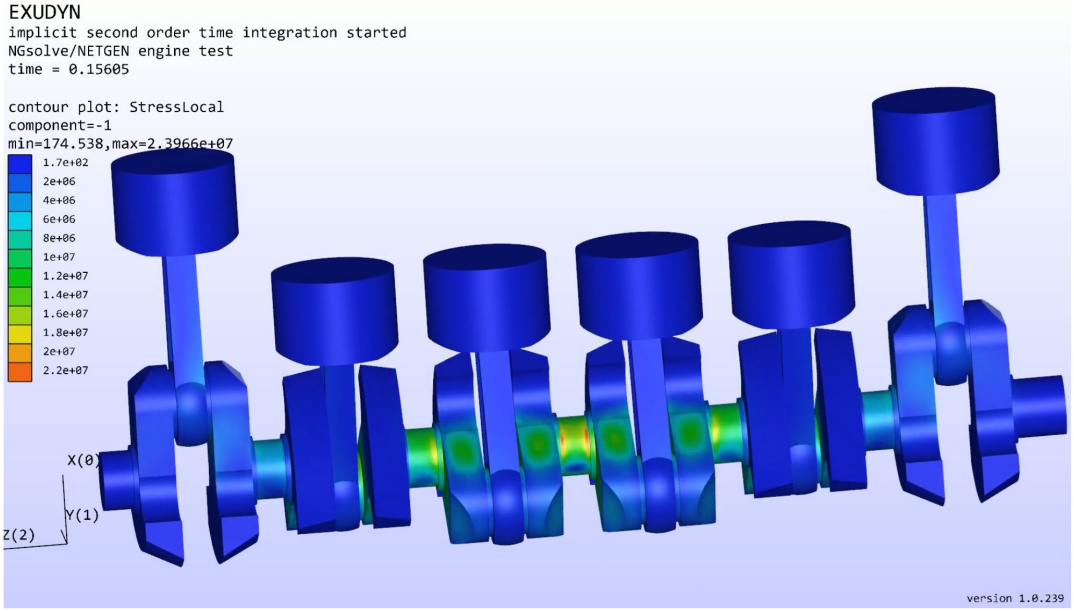
Flexible Multibody Dynamics Systems with Python and C++

Exudyn

USER DOCUMENTATION



(mesh and FEM-model generated with NETGEN and NGsolve – 647058 total coordinates)

Exudyn version = 1.9.201.dev1

(build date and time=2025-06-18 17:00)

CHECK Section 12 for changes from previous versions!!!

University of Innsbruck, Department of Mechatronics, June 18, 2025,

Johannes Gerstmayr

Table of Contents

List of abbreviations (for theDoc, C++ and Python codes)

This section shows typical abbreviations. For further symbols in notation, see Section Notation Section 4.

2D two dimensions or planar

3D three dimensions or spatial

abs absolute (e.g., absolute error), absolute value

AE algebraic equations

CMS component mode synthesis

coeffs coefficients

COM center of mass

DOF degree of freedom

EOM equations of motion

EP Euler parameters

FFRF floating frame of reference formulation

HT homogeneous transformation

LHS Left-Hand-Side (of equation)

LTG local-to-global

mbs multibody system

min minimum

max maximum

ODE ordinary differential equation

ODE1 first order ordinary differential equations

ODE2 second order ordinary differential equations

pos position

quad quadrangle, polygon with 4 vertices

rel relative (e.g., relative error)

RHS Right-Hand-Side (of equation)

Rot rotation

Rxyz rotation parameterization: consecutive rotations around x, y and z-axis (Tait-Bryan)

STL STereoLithography

T66 Plücker transformation

trig triangle

Chapter 1

Installation and Getting Started

The documentation for Exudyn is split into this introductory section, including a quick start up, installation, code structure and important hints, as well as a larger part containing the Exudyn module description and interfaces, theory, references to the accessible items and data structures, solvers, settings and much more.

How to cite:

> Johannes Gerstmayr. Exudyn – a C++-based Python package for flexible multibody systems. Multibody System Dynamics, Vol. 60, pp. 533-561, 2024. https://doi.org/10.1007/s11044-023-09937-1 [19]

Tutorial videos can be found in the youtube channel of Exudyn. Exudyn is hosted on GitHub [18]:

- https://github.com/jgerstmayr/EXUDYN

Online documentation is available at:

- https://jgerstmayr.github.io/EXUDYN
- https://exudyn.readthedocs.io

For any comments, requests, issues, bug reports, send an email to:

- email: reply.exudyn@gmail.com

Thanks for your contribution!

1.1 Getting started

This section will show:

1. What is Exudyn?
2. Who is developing Exudyn?
3. How to install Exudyn
4. How to link Exudyn and Python

5. Goals of Exudyn
6. Run a simple example in Python
7. FAQ – Frequently asked questions

1.1.1 What is Exudyn?

Exudyn– (fl**EX**ible m**U**ltibody **DYN**amics – **EX**tend yo**U**r **DYN**amics)

Exudyn is a C++ based Python library for efficient simulation of flexible multibody dynamics systems. It is the follow up code of the previously developed multibody code HOTINT, which Johannes Gerstmayr started during his PhD-thesis. It seemed that the previous code HOTINT reached limits of further (efficient) development and it seemed impossible to continue from this code as it was outdated regarding programming techniques and the numerical formulation at the time Exudyn was started.

Exudyn is designed to easily set up complex multibody models, consisting of rigid and flexible bodies with joints, loads and other components. It shall enable automatized model setup and parameter variations, which are often necessary for system design but also for analysis of technical problems. The broad usability of Python allows to couple a multibody simulation with environments such as optimization, statistics, data analysis, machine learning and others.

The multibody formulation is mainly based on redundant coordinates. This means that computational objects (rigid bodies, flexible bodies, ...) are added as independent bodies to the system. Hereafter, connectors (e.g., springs or constraints) are used to interconnect the bodies. The connectors are using Markers on the bodies as interfaces, in order to transfer forces and displacements. For details on the interaction of nodes, objects, markers and loads see Section 2.2. For a non-redundant formulation, see ObjectKinematicTree – this allows to create tree-structures with minimal coordinates in Exudyn.

There are several journal papers of the developers which were using Exudyn (list may be incomplete):

- J. Gerstmayr. Exudyn – a C++-based Python package for flexible multibody systems. Multibody System Dynamics (2023). https://doi.org/10.1007/s11044-023-09937-1 [19]
- J. Gerstmayr, P. Manzl, M. Pieber. Multibody Models Generated from Natural Language, Preprint, Research Square, 2023. https://doi.org/10.21203/rs.3.rs-3552291/v1 [23]
- P. Manzl, O. Rogov, J. Gerstmayr, A. Mikkola, G. Orzechowski. Reliability Evaluation of Reinforcement Learning Methods for Mechanical Systems with Increasing Complexity. Preprint, Research Square, 2023. [41] https://doi.org/10.21203/rs.3.rs-3066420/v1
- S. Holzinger, M. Arnold, J. Gerstmayr. Evaluation and Implementation of Lie Group Integration Methods for Rigid Multibody Systems. Preprint, Research Square, 2023. [31] https://doi.org/10.21203/rs.3.rs-2715112/v1
- M. Sereinig, P. Manzl, and J. Gerstmayr. Task Dependent Comfort Zone, a Base Placement Strategy for Autonomous Mobile Manipulators using Manipulability Measures, Robotics and Autonomous Systems, submitted. [54]

- R. Neurauter, J. Gerstmayr. A novel motion reconstruction method for inertial sensors with constraints, Multibody System Dynamics, 2022. [44]
- M. Pieber, K. Ntarladima, R. Winkler, J. Gerstmayr. A Hybrid ALE Formulation for the Investigation of the Stability of Pipes Conveying Fluid and Axially Moving Beams, ASME Journal of Computational and Nonlinear Dynamics, 2022. [48]
- S. Holzinger, M. Schieferle, C. Gutmann, M. Hofer, J. Gerstmayr. Modeling and Parameter Identification for a Flexible Rotor with Impacts. Journal of Computational and Nonlinear Dynamics, 2022. [33]
- S. Holzinger, J. Gerstmayr. Time integration of rigid bodies modelled with three rotation parameters, Multibody System Dynamics, Vol. 53(5), 2021. [32]
- A. Zwölfer, J. Gerstmayr. The nodal-based floating frame of reference formulation with modal reduction. Acta Mechanica, Vol. 232, pp. 835–851 (2021). [67]
- A. Zwölfer, J. Gerstmayr. A concise nodal-based derivation of the floating frame of reference formulation for displacement-based solid finite elements, Journal of Multibody System Dynamics, Vol. 49(3), pp. 291 – 313, 2020. [66]
- S. Holzinger, J. Schöberl, J. Gerstmayr. The equations of motion for a rigid body using non-redundant unified local velocity coordinates. Multibody System Dynamics, Vol. 48, pp. 283 – 309, 2020. [34]

1.1.2 Developers of Exudyn and thanks

`Exudyn` is currently (6-2025) developed at the University of Innsbruck. In the first phase most of the core code is written by Johannes Gerstmayr, implementing ideas that followed out of the project HOTINT [20]. 15 years of development led to a lot of lessons learned and after 20 years, a code must be re-designed.

Some important tests for the coupling between C++ and Python have been written by Stefan Holzinger. Stefan also helped to set up the previous upload to GitLab and to test parallelization features. For the interoperability between C++ and Python, we extensively use **Pybind11**[37], originally written by Jakob Wenzel, see `https://github.com/pybind/pybind11`. Without Pybind11 we couldn't have made this project – Thanks a lot!

Important discussions with researchers from the community were important for the design and development of `Exudyn`, where we like to mention Joachim Schöberl from TU-Vienna who boosted the design of the code with great concepts.

The cooperation and funding within the EU H2020-MSCA-ITN project 'Joint Training on Numerical Modelling of Highly Flexible Structures for Industrial Applications' contributes to the development of the code.

The following people have contributed to Python and C++ library implementations, testing, examples or theory:

- Joachim Schöberl, TU Vienna (Providing specialized NGsolve [51, 52, 14] core library with `taskmanager` for **multi-threaded parallelization**, which is now replaced by a simplified internal version but

closely following the original implementation; NGsolve mesh and FE-matrices import; highly efficient eigenvector computations)

- Stefan Holzinger, University of Innsbruck (Lie group module and solvers in Python, Lie group node; helped with Lie group solvers, geometrically exact beam; testing)
- Peter Manzl, University of Innsbruck (ConvexRoll Python and C++ implementation; revised artificialIntelligence, ParameterVariation, robotics and MPI parallelization; providing many figures for theDoc; pip install on linux, wsl with graphics)
- Andreas Zwölfer, Technical University Munich (theory and examples for FFRF, CMS formulation and ANCF 2D cable prototypes in MATLAB)
- Martin Sereinig, University of Innsbruck (special robotics functionality, mobile robots, manipulability measures, robot models)
- Michael Pieber, University of Innsbruck (helped in several Python libraries; ComputeODE2Eigenvalues with constraints, FEM and CMS testing; Abaqus import and test files; ANCFCable2D+ALE theory improvements and equations check)
- Grzegorz Orzechowski, Lappeenranta University of Technology (coupling with openAI gym and running machine learning algorithms)
- Aaron Bacher, University of Innsbruck (helped to integrated OpenVR, connection with Franka Emika Panda)
- Martin Arnold, Martin-Luther-University of Halle-Wittenberg (support for explicit and implicit Lie group solvers, especially to theory / jacobians and automatic step size)
- Konstantina Ntarladima, University of Innsbruck (ANCFCable2D+ALE theory improvements and equations check)
- Alexander Humer, Johannes Kepler University Linz (initial discussions on structure and C++ code)
- Qasim Khadim, University of Oulu (suggestion for improved model of HydraulicsActuatorSimple with effective bulk modulus)
- Michael Gerbl, University of Innsbruck (figures in the documentation, taken from lecture notes)
- examples provided by: Manuel Schieferle, Martin Knapp, Lukas March, Dominik Sponring, David Wibmer, Simon Scheiber

– thanks a lot! –

1.2 Installation instructions

1.2.1 Requirements for Exudyn?

Exudyn only works with Python. Thus, you need an appropriate Python installation. So far (2021-07), we tested

- **Anaconda 2023-07, 64bit, Python 3.11** with Spyder 5.4
- **Anaconda 2021-11, 64bit, Python 3.9**[1]
- Currently, we support Python 3.8 - Python 3.12 **conda environments** on Windows, Linux and MacOS (please report configurations that do not work on GitHub/issues).
- **Spyder 5.1.5** (with Python 3.9.7, 64bit) and **Spyder 4.1.3** (with Python 3.7.7, 64bit), which is included in the Anaconda installation[2]; Spyder works with all virtual environments

Many alternative options exist:

- Users report successful use of Exudyn with **Visual Studio Code**. **Jupyter** has been tested with some examples; both environments should work with default settings.
- Anaconda 2020-11 with **Python 3.8** and Spyder 4.1.5: no problems except some regular crashes of Spyder, TestSuite runs without problems since Exudyn version 1.0.182.
- Alternative option with more stable Spyder (as compared to Spyder 4.1.3): Anaconda, 64bit, Python 3.6.5)[3]

If you plan to extend the C++ code, we recommend to use Microsoft Visual Studio (VS2022) and previously VS2017[4] to compile your code, which offers Python 3.x compatibility. Remember that Python versions and the version of the Exudyn module must be identical (e.g., Python 3.9 64 bit **both** in the Exudyn module and e.g. in Spyder).

1.2.1.1 Run without Anaconda

If you do not install Anaconda (e.g., under Linux), make sure that you have the according Python packages installed:

- `numpy` (used throughout the code, inevitable)
- `matplotlib` (for any plot, also PlotSensor(...))
- `tkinter` (for interactive dialogs, SolutionViewer, etc.)
- `scipy` (needed for eigenvalue computation)

[1] older Anaconda3 versions can be downloaded via the repository archive `https://repo.anaconda.com/archive/`

[2] Note that it is important that Spyder, Python and Exudyn are **either** 32bit **or** 64bit and are compiled up to the same minor version, i.e., 3.7.x. There will be a strange .DLL error, if you mix up 32/64bit. It is possible to install both, Anaconda 32bit and Anaconda 64bit – then you should follow the recommendations of paths as suggested by Anaconda installer.

[3] Anaconda 64bit with Python3.6 can be downloaded via the repository archive `https://repo.anaconda.com/archive/` choosing `Anaconda3-5.2.0-Windows-x86_64.exe` for 64bit.

[4] Note: VS2019 has problems with the library 'Eigen' and therefore leads to erroneous results with the sparse solver.

You can install most of these packages using `pip install numpy` (Windows) or `pip3 install numpy` (Linux). NOTE: as there is only `numpy` needed (but not for all sub-packages) and `numpy` supports many variants, we do not add a particular requirement for installation of depending packages. It is not necessary to install `scipy` as long as you are not using features of `scipy`. Same reason for `tkinter` and `matplotlib`.

For interaction (right-mouse-click, some key-board commands) you need the Python module `tkinter`. This is included in regular Anaconda distributions (recommended, see below), but on Ubuntu you need to type alike (do not forget the '3', otherwise it installs for Python2 ...):

```
sudo apt-get install python3-tk
```

see also common blogs for your operating system.

1.2.2 Install Exudyn with PIP INSTALLER (pypi.org)

Pre-built versions of `Exudyn` are hosted on `pypi.org`, see the project

- https://pypi.org/project/exudyn

As with most other packages, in the regular case (if your binary has been pre-built) you just need to do

```
pip install exudyn
```

On Ubuntu/Linux, make sure that pip is installed and up-to-date (**update pip to at least 20.3**; otherwise the manylinux wheels will not be accepted!):

```
sudo apt install python3-pip
python3 -m pip install -upgrade
```

Depending on installation the command may read `pip3` or `pip`:

```
pip3 install exudyn
```

For pre-releases (use with care!), add `--pre` flag:

```
pip install exudyn --pre -U
```

The `-U` (identical to `-upgrade`) flag ensures that the current installed version is also updated in case of a change of the micro version (e.g., from version 1.6.119 to version 1.6.164), otherwise, it will only update if you switch to a newer minor version.

In some cases (e.g. for AppleM1 or special Linux versions), your pre-built binary will not work due to some incompatibilities. Then you need to build from source as described in the 'Build and install' sections, Section 1.2.4.

1.2.2.1 Troubleshooting pip install

Pip install may fail, if your linux version does not support the current manylinux version. This was known for Red Hat, CentOS, Rocky Linux or simlilar systems which usually support manylinux2014. In this case, you had to build `Exudyn` from source, see Section 1.2.6. Since version 1.7.116, the manylinux2014 version is supported and according problems should be solved.

Sometimes, you install exudyn, but when running python, the `import exudyn` fails. In case of several environments, check where your installation goes. To guarantee that the pip install goes to the python call, use:

```
python -m pip install exudyn
```

which ensures that the used python is calling its associated pip module.

If the PyPi index is not updated, it may help to use

```
pip install -i https://pypi.org/project/ exudyn
```

1.2.3 Install from specific Wheel (Ubuntu and Windows)

A way to install the Python package `Exudyn` is to use the so-called 'wheels' (file ending `.whl`). NOTE that this approach usually is not required; just use the pip installer of the previous section!
Ubuntu:

Wheels can be downloaded directly from https://pypi.org/project/exudyn/#files, for many Python versions and architectures.

For Ubuntu 18.04 (which by default uses Python 3.6) this may read (version number 1.0.20 may be different):

- `Python 3.6, 64bit`: pip3 install dist\exudyn-1.0.20-cp36-cp36-linux_x86_64.whl

For Ubuntu 20.04 (which by default uses Python 3.8) this may read (version number 1.0.20 may be different):

- `Python 3.8, 64bit`: pip3 install dist\exudyn-1.0.20-cp38-cp38-linux_x86_64.whl

NOTE that your installation may have environments with different Python versions, so install that `Exudyn` version appropriately! If the wheel installation does not work on Ubuntu, it is highly recommended to build `Exudyn` for your specific system as given in Section 1.2.6.
Windows:

First, open an Anaconda prompt:

- EITHER calling: START->Anaconda->... OR go to anaconda/Scripts folder and call activate.bat
- You can check your Python version then, by running `python`[5], the output reads like:

```
Python 3.6.5 |Anaconda, Inc.| (default, Mar 29 2018, 13:32:41) [MSC v.1900 64 bit
(AMD64)] on win32
```

[5] `python3` under Ubuntu 18.04

...

- type `exit()` to close Python

For Windows the installation commands may read (version number 1.0.20 may be different):

- `Python 3.6, 32bit`: pip install dist\exudyn-1.0.20-cp36-cp36m-win32.whl
- `Python 3.6, 64bit`: pip install dist\exudyn-1.0.20-cp36-cp36m-win_amd64.whl
- `Python 3.7, 64bit`: pip install dist\exudyn-1.0.20-cp37-cp37m-win_amd64.whl

1.2.4 Build and install Exudyn under Windows 10?

Note that there are a couple of pre-requisites, depending on your system and installed libraries. For Windows 10, the following steps proved to work:

- you need an appropriate compiler (tested with Microsoft Visual Studio; recommended: VS2017)
- install your Anaconda distribution including Spyder
- close all Python programs (e.g. Spyder, Jupyter, ...)
- run an Anaconda prompt (may need to be run as administrator)
- if you cannot run Anaconda prompt directly, do:
  - open windows shell (cmd.exe) as administrator (START → search for cmd.exe → right click on app → 'run as administrator' if necessary) [may not be necessary]
  - go to your Scripts folder inside the Anaconda folder (e.g. `C:\ProgramData\Anaconda\Scripts`) [may not be necessary]
  - run 'activate.bat' [may not be necessary]
- go to 'main' of your cloned github folder of `Exudyn`
- run:[6] `pip wheel .  -v -w dist -no-deps`
- Before version 1.7.116: run:[7] `python setup.py install -parallel`
- read the output; if there are errors, try to solve them by installing appropriate modules

You can also create your own wheels, doing the above steps to activate the according Python version and then calling:

```
python setup.py bdist_wheel -parallel
```

This will add a wheel in the `dist` folder.

[6] Since version 1.7.116 a PEP518 compatible build is available. This should work with Windows, MacOS and linux. The `setupPyConfig.json` file includes some flags such as the parallel compilation, GLFW, etc.; the `-v` flag adds verbosity.

[7] the `-parallel` option performs parallel compilation on multithreaded CPUs and can speedup by 2x - 8x

1.2.5 Build and install Exudyn under Mac OS X?

Installation and building on Mac OS X is less frequently tested, but successful compilation including GLFW has been achieved. Requirements are an according Anaconda (or Miniconda) installation.
Tested configurations:

- Mac OS 11.x 'Big Sur', Mac Mini (2021), Apple M1, 16GB Memory
- Miniconda with conda environments (x86 / i368 based with Rosetta 2) with Python 3.7 - 3.11
- Miniconda with conda environments (ARM) with Python 3.8 - 3.11
  → wheels are available on pypi since `Exudyn` 1.5.0

NOTE:

- New `universal2` wheels should support x86 (APPLE Intel and Python/Rosetta on APPLE Silicon)
- Multi-threading is not fully supported on MacOS, but may work in some applications
- On Apple M1 processors the newest Anaconda supports now all required features; environments with Python 3.8-3.11 have been successfully tested;
- The Rosetta (x86 emulation) mode on Apple M1 also works now without much restrictions; these files should also work on older Macs
- If you have a MacOS version< 11, it worked to download wheels from PyPI, change wheel names, e.g., from `exudyn-1.7.116.dev1-cp311-cp311-macosx_11_0_x86_64.whl` to `exudyn-1.7.116.dev1-cp311-c`
  This also works for universal2 files. Installation worked and wheels were running smoothly.
- `tkinter` has been adapted (some workarounds needed on MacOS!), available since `Exudyn` 1.5.15.dev1
- Some optimization and processing functions do not run (especially multiprocessing and tqdm);

Alternatively, we tested on:

- Mac OS X 10.11.6 'El Capitan', Mac Pro (2010), 3.33GHz 6-Core Intel Xeon, 4GB Memory, Anaconda Navigator 1.9.7, Python 3.7.0, Spyder 3.3.6

Compile from source:
If you would like to compile from source, just use a bash terminal on your Mac, and do the following steps inside the `main` directory of your repository and type

- uninstall if old version exists (may need to repeat this!): `pip uninstall exudyn`
- remove the `build` directory if you would like to re-compile without changes
- to perform compilation from source, write:[8]
- Since version 1.7.116: `pip wheel .  -v -w dist -no-deps`
- Until version 1.7.116: `python setup.py bdist_wheel -parallel`
- which takes 75 seconds on Apple M1 in parallel mode, otherwise 5 minutes. To install `Exudyn`, run

[8] the `-parallel` option performs parallel compilation on multithreaded CPUs and can speedup by 2x - 8x

`python setup.py install`

→ this will only install, but not re-compile. Otherwise, just use pip install from the created wheel in the dist folder

NOTE that conda environments are highly recommended

Then just go to the `pythonDev/Examples` folder and run an example:

```
python springDamperUserFunctionTest.py
```

If there are other issues, we are happy to receive your detailed bug reports.
Note that you need to run

```
SC.renderer.Start()
SC.renderer.DoIdleTasks()
```

in order to interact with the render window, as there is only a single-threaded version available for Mac OS.

1.2.6 Build and install Exudyn under Ubuntu?

Having a new Ubuntu 18.04 standard installation (e.g. using a VM virtual box environment), the following steps need to be done (Python **3.6** is already installed on Ubuntu 18.04, otherwise use `sudo apt install python3`)[9]:
First update ...

```
sudo apt-get update
```

Install necessary Python libraries and pip3; `matplotlib` and `scipy` are not required for installation but used in `Exudyn` examples:

```
sudo dpkg --configure -a
sudo apt install python3-pip
pip3 install numpy
pip3 install matplotlib
pip3 install scipy
```

Install pybind11 (needed for running the setup.py file derived from the pybind11 example):

```
pip3 install pybind11
```

To have dialogs enabled, you need to install `Tk`/`tkinter` (may be already installed in your case). `Tk` is installed on Ubuntu via apt-get and should then be available in Python:

```
sudo apt-get install python3-tk
```

If graphics is used (`#define USE_GLFW_GRAPHICS` in `BasicDefinitions.h`), you must install the according GLFW libs:

```
sudo apt-get install libglfw3 libglfw3-dev
```

[9] see also the youtube video: https://www.youtube.com/playlist?list=PLZduTa9mdcmOh5KVUqatD9GzVg_jtl6fx

In some cases, it may be required to install OpenGL and some of the following libraries:

```
sudo apt-get install freeglut3 freeglut3-dev
sudo apt-get install mesa-common-dev
sudo apt-get install libx11-dev xorg-dev libglew1.5 libglew1.5-dev libglu1-mesa libglu1-
  mesa-dev libgl1-mesa-glx libgl1-mesa-dev
```

With all of these libs, you can run the setup.py installer (go to `Exudyn_git/main` folder), which takes some minutes for compilation (the –user option is used to install in local user folder)[10]:

```
sudo python3 setup.py install --user --parallel
```

Since version 1.7.116, a PEP518 compatible way to compile sources and install the current repository has been added (the `-v` flag activates a verbose mode):

```
pip install . -v --no-deps
```

Congratulation! **Now, run a test example** (will also open an OpenGL window if successful):

```
python3 pythonDev/Examples/rigid3Dexample.py
```

You can also create a Ubuntu wheel which can be easily installed on the same machine (x64), same operating system (Ubuntu 18.04) and with same Python version (e.g., 3.6):

```
sudo pip3 install wheel
sudo python3 setup.py bdist_wheel -parallel
```

Since version 1.7.116, the PEP518 compatible way which puts wheels into the `dist` folder reads:

```
pip wheel .  -v -w dist -no-deps
```

Exudyn under Ubuntu / WSL:

- Note that `Exudyn` also nicely works under WSL (Windows subsystem for linux; tested for Ubuntu 18.04) and an according xserver (VcXsrv).
- In case of old WSL2, just set the display variable in your .bashrc file accordingly and you can enjoy the OpenGL windows and settings.
- It shall be noted that WSL + xserver works better than on MacOS, even for tkinter, multitasking, etc.! So, if you have troubles with your Mac, use a virtual machine with ubuntu and a xserver, that may do better
- In case of WSLg (since 2021), only the software-OpenGL works; therefore, you have to set (possibly in .bashrc file): `export LIBGL_ALWAYS_SOFTWARE=0`

Exudyn under RaspberryPi 4b:

- Exudyn also compiles under RaspberryPi 4b, Ubuntu Mate 20.04, Python 3.8; current version should compile out of the box using `python3 setup.py install` command.
- Performance is quite ok and it is even capable to use all cores (but you should add a fan!)

[10] the `-parallel` option performs parallel compilation on multithreaded CPUs and can speedup by 2x - 8x

- → this could lead to a nice cluster project!

KNOWN issues for linux builds:

- Using **WSL2** (Windows subsystem for linux), there occur some conflicts during build because of incompatible windows and linux file systems and builds will not be copied to the dist folder; workaround: go to explorer, right click on 'build' directory and set all rights for authenticated user to 'full access'
- **compiler (gcc,g++) conflicts**: It seems that `Exudyn` works well on Ubuntu 18.04 with the original `Python 3.6.9` and `gcc-7.5.0` version as well as with Ubuntu 20.04 with `Python 3.8.5` and `gcc-9.3.0`. Upgrading `gcc` on a Linux system with Python 3.6 to, e.g., `gcc-8.2` showed us a linker error when loading the `Exudyn` module in Python – there are some common restriction using `gcc` versions different from those with which the Python version has been built. Starting `python` or `python3` on your linux machine shows you the `gcc` version it had been build with. Check your current `gcc` version with: `gcc -version`

1.2.7 Uninstall Exudyn

To uninstall exudyn under Windows, run (may require admin rights):

```
pip uninstall exudyn
```

To uninstall under Ubuntu, run:

```
sudo pip3 uninstall exudyn
```

If you upgrade to a newer version, uninstall is usually not necessary!

1.2.8 How to install Exudyn and use the C++ source code (advanced)?

`Exudyn` is still under intensive development of core modules. There are several ways of using the code, but you **cannot** install `Exudyn` as compared to other executable programs and apps.

In order to make full usage of the C++ code and extending it, you can use:

- Windows / Microsoft Visual Studio 2017 and above:
  - get the files from git
  - put them into a local directory (recommended: `C:/DATA/cpp/EXUDYN_git`)
  - start `main_sln.sln` with Visual Studio (recommended version: 2017, otherwise you have to manually adapt)
  - compile the code and run `main/pythonDev/pytest.py` example code
  - adapt `pytest.py` for your applications
  - extend the C++ source code
  - link it to your own code
  - NOTE: on Linux systems, you mostly need to replace '/' with '\'
- Linux, etc.: Use the build methods described above; Visual Studio Code may allow native Python and C++ debugging; switching to other build mechanisms (CMakeLists or scikit-build-core).

1.3 Further notes

1.3.1 Goals of Exudyn

After the first development phase (2019-2023), it

- is a moderately large [11] multibody library, which can be easily linked to other projects,
- contains basic multibody rigid bodies, flexible bodies, joints, contact, etc.,
- includes a large Python utility library for convenient building and post processing of models,
- allows to efficiently simulate small scale systems (compute 100 000s of time steps per second for systems with $n_{DOF} < 10$),
- allows to efficiently simulate medium scaled systems for problems with $n_{DOF} < 1\,000\,000$,
- is a safe and widely accessible module for Python,
- allows to add user defined objects and solvers in C++,
- allows to add user defined objects and solvers in Python,
- allows multi-threaded parallel computing,
- includes Lie group integration,
- includes interfaces for robotics and ROS,
- includes interfaces for reinforcement learning (stable-baselines3), pytorch and artificial intelligence,
- includes kinematical trees with minimal coordinates.

Future goals (2024-2026) are:

- add specific and advanced connectors/constraints (extended wheels, contact, control connector)
- automatic step size selection for second order solvers (planned 2025),
- add 3D beams and plates (first attempts exist; planned 2024),
- export equations (planned, 2025),
- add GPU support (planned, 2025).

For solved issues (and new features), see section 'Issues and Bugs', Section 12. For specific open issues, see `trackerlog.html` – a document only intended for developers!

1.4 Run a simple example in Python

After performing the steps of the previous section, this section shows a simplistic model which helps you to check if Exudyn runs on your computer.

In order to start, run the Python interpreter Spyder (or any preferred Python environment). In order to test the following example, which creates a multibody system (mbs), adds a node, an object, a marker and a load and simulates everything with default values,

[11] wheels have only sizes of 2MB on Windows and 4MB on Linux, without fast Exudyn options

Listing 1.1: My first example

```
#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# This is an EXUDYN example
#
# Details:  Micro example from documentation; use this to check if Exudyn works
#
# Author:   Johannes Gerstmayr
# Date:     2019-08-01
#
# Copyright:This file is part of Exudyn. Exudyn is free software. You can redistribute it
    and/or modify it under the terms of the Exudyn license. See 'LICENSE.txt' for more
    details.
#
#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

import exudyn as exu                #EXUDYN package including C++ core part
from exudyn.itemInterface import * #conversion of data to exudyn dictionaries

SC = exu.SystemContainer()          #container of systems
mbs = SC.AddSystem()                #add a new system to work with

nMP = mbs.AddNode(NodePoint2D(referenceCoordinates=[0,0]))
mbs.AddObject(ObjectMassPoint2D(physicsMass=10, nodeNumber=nMP ))
mMP = mbs.AddMarker(MarkerNodePosition(nodeNumber = nMP))
mbs.AddLoad(Force(markerNumber = mMP, loadVector=[0.001,0,0]))

mbs.Assemble()                      #assemble system and solve
simulationSettings = exu.SimulationSettings()
simulationSettings.timeIntegration.verboseMode=1 #provide some output
mbs.SolveDynamic(simulationSettings)
```

- open Spyder and copy the example provided in Listing 1.1 into a new file, or
- open `myFirstExample.py` from your `Examples` folder.

Hereafter, press the play button or `F5` in Spyder.

If successful, the IPython Console of Spyder will print something like:

```
runfile('C:/DATA/cpp/EXUDYN_git/main/pythonDev/Examples/myFirstExample.py',
  wdir='C:/DATA/cpp/EXUDYN_git/main/pythonDev/Examples')
++++++++++++++++++++++++++++++
EXUDYN V1.2.9 solver: implicit second order time integration
STEP100, t = 1 sec, timeToGo = 0 sec, Nit/step = 1
solver finished after 0.0007824 seconds.
```

If you check your current directory (where `myFirstExample.py` lies), you will find a new file `coordinatesSolution.txt`, which contains the results of your computation (with default values for

time integration). The beginning and end of the file should look like:

```
#Exudyn implicit second order time integration solver solution file
#simulation started=2022-04-07,19:02:19
#columns contain: time, ODE2 displacements, ODE2 velocities, ODE2 accelerations
#number of system coordinates [nODE2, nODE1, nAlgebraic, nData] = [2,0,0,0]
#number of written coordinates [nODE2, nVel2, nAcc2, nODE1, nVel1, nAlgebraic, nData] =
  [2,2,2,0,0,0,0]
#total columns exported  (excl. time) = 6
#number of time steps (planned) = 100
#Exudyn version = 1.2.33.dev1; Python3.9.11; Windows AVX2 FLOAT64
#
0,0,0,0,0,0.0001,0
0.01,5e-09,0,1e-06,0,0.0001,0
0.02,2e-08,0,2e-06,0,0.0001,0
0.03,4.5e-08,0,3e-06,0,0.0001,0
0.04,8e-08,0,4e-06,0,0.0001,0
0.05,1.25e-07,0,5e-06,0,0.0001,0

...

0.96,4.608e-05,0,9.6e-05,0,0.0001,0
0.97,4.7045e-05,0,9.7e-05,0,0.0001,0
0.98,4.802e-05,0,9.8e-05,0,0.0001,0
0.99,4.9005e-05,0,9.9e-05,0,0.0001,0
1,5e-05,0,0.0001,0,0.0001,0
#simulation finished=2022-04-07,19:02:19
#Solver Info: stepReductionFailed(or step failed)=0,discontinuousIterationSuccessful=1,
  newtonSolutionDiverged=0,massMatrixNotInvertible=1,total time steps=100,total Newton
  iterations=100,total Newton jacobians=100
```

Within this file, the first column shows the simulation time and the following columns provide coordinates, their derivatives and Lagrange multipliers on system level. For relation of local to global coordinates, see Section 2.3. As expected, the x-coordinate of the point mass has constant acceleration $a = f/m = 0.001/10 = 0.0001$, the velocity grows up to 0.0001 after 1 second and the point mass moves 0.00005 along the x-axis.

Note that line 8 contains the `Exudyn` and Python versions[12] provided in the solution file are the versions at which Exudyn has been compiled with. The Python micro version (last digit) may be different from the Python version from which you were running Exudyn. This information is also provided in the sensor output files.

[12] as well as some other specific information on the platform and compilation settings (which may help you identify with which computer, etc., you created results)

1.5 Trouble shooting and FAQ

1.5.1 Trouble shooting

Exudyn has a solid exception handling and does lots of error checking on inputs as well as during computations. This should usually not lead to an unexpected crash as in early days of scientific codes.

For basic information on exception handling, see also the according section on Exceptions and Error Messages. In the following, typical error messages are listed.

Python import errors:

- Sometimes the Exudyn module cannot be loaded into Python. Typical **error messages if Python versions are not compatible** are:

```
Traceback (most recent call last):

  File "<ipython-input-14-df2a108166a6>", line 1, in <module>
    import exudynCPP

ImportError: Module use of python36.dll conflicts with this version of Python.
```

  Typical **error messages if 32/64 bits versions are mixed**:

```
Traceback (most recent call last):

  File "<ipython-input-2-df2a108166a6>", line 1, in <module>
    import exudynCPP

ImportError: DLL load failed: \%1 is not a valid Win32 application.
```

  There are several reasons and workarounds:

  → You mixed up 32 and 64 bits version (see below)

  → You are using an exudyn version for Python $x_1.y_1$ (e.g., 3.6.z_1) different from the Python $x_2.y_2$ version in your Anaconda (e.g., 3.7.z_2); note that $x_1 = x_2$ and $y_1 = y_2$ must be obeyed while z_1 and z_2 may be different

- **Import of exudyn C++ module failed Warning: ...**:

  → ... and similar messages with: ModuleNotFoundError, Warning, with AVX2, without AVX2

  → A known reason is that your CPU **does not support AVX2**, while Exudyn is compiled with the AVX2 option[13].

[13] modern Intel Core-i3, Core-i5 and Core-i7 processors as well as AMD processors, especially Zen and Zen-2 architectures should have no problems with AVX2; however, low-cost Celeron, Pentium and older AMD processors do **not** support AVX2, e.g., Intel Celeron G3900, Intel core 2 quad q6600, Intel Pentium Gold G5400T; check the system settings of your computer to find out the processor type; typical CPU manufacturer pages or Wikipedia provide information on this

→ **solution**: the release versions without the .dev1 ending in the wheel contain C++ libraries which are compiled without AVX/AVX2; the module loader will usually detect automatically, if your CPU supports AVX/AVX2; if not, it will load the exudynCPPnoAVX.cp ... .pyd file; if this does not work, try

```
import sys
sys.exudynCPUhasAVX2 = False
```

to explicitly load the version without AVX2.

→ you can also compile for your specific Python version without AVX if you adjust the `setup.py` file in the `main` folder.

→ **DEPRECATED workaround** to solve the AVX problem: use the Python 3.6 version (up to `Exudyn` V1.2.28 only the 32bit version), which is compiled without AVX2.

→ The `ModuleNotFoundError` may also happen if something went wrong during installation (paths, problems with Anaconda, ..) → very often a new installation of Anaconda and `Exudyn` helps.

Typical Python errors:

- Typical Python **syntax error** with missing braces:

```
File "C:\DATA\cpp\EXUDYN_git\main\pythonDev\Examples\springDamperTutorial.py", line 42
    nGround=mbs.AddNode(NodePointGround(referenceCoordinates = [0,0,0]))
           ^
SyntaxError: invalid syntax
```

→ such an error points to the line of your code (line 42), but in fact the error may have been caused in previous code, such as in this case there was a missing brace in the line 40, which caused the error:

```
38  n1=mbs.AddNode(Point(referenceCoordinates = [L,0,0],
39                       initialCoordinates = [u0,0,0],
40                       initialVelocities= [v0,0,0])
41  #ground node
42  nGround=mbs.AddNode(NodePointGround(referenceCoordinates = [0,0,0]))
43
```

- Typical Python **import error** message on Linux / Ubuntu if Python modules are missing:

```
Python WARNING [file '/home/johannes/.local/lib/python3.6/site-packages/exudyn/solver.
  py', line 236]:
Error when executing process ShowVisualizationSettingsDialog':
ModuleNotFoundError: No module named 'tkinter'
```

→ see installation instructions to install missing Python modules, Section 1.2.

- Problems with **tkinter**, especially on MacOS:
  `Exudyn` uses `tkinter`, based on tcl/tk, to provide some basic dialogs, such as visualizationSettings As Python is not suited for multithreading, this causes problems in window and dialog workflows. Especially on MacOS `tkinter` is less stable and compatible with the window manager. Especially, `tkinter` already needs to run before the application's OpenGL window (renderer) is

opened. Therefore, on MacOS `tkinter.Tk()` is called before the renderer is started. In some cases, visualizationSettings dialog may not be available and changes have to be made inside the code.

→ To resolve issues, the following visualizationSettings may help (before starting renderer!), but may reduce functionality: dialogs.multiThreadedDialogs = False, general.useMultiThreadedRendering = False

Typical solver errors:

- Consider the example for a mixed error message comes from the solver when called for a (possibly empty) `mbs` with no prior call to `mbs.Assemble()`:

```
import exudyn as exu
from exudyn.utilities import *
SC = exu.SystemContainer()
mbs = SC.AddSystem()
sims=exu.SimulationSettings()
exu.SolveDynamic(mbs, sims)
```

- This will results in error messages similar to:

```
==========================================
User ERROR [file 'C:\Users\username\.conda\envs\venvP39\lib\site-packages\exudyn\solver
  .py', line 245]:
Solver: system is inconsistent and cannot be solved (call Assemble() and check error
  messages)
==========================================

==========================================
SYSTEM ERROR [file 'C:\Users\username\.conda\envs\venvP39\lib\site-packages\exudyn\
  solver.py', line 245]:
EXUDYN raised internal error in 'CSolverBase::InitializeSolver':
Exudyn: parsing of Python file terminated due to Python (user) error
==========================================

*****************************
DYNAMIC SOLVER FAILED:
  use showHints=True to show helpful information
*****************************

Traceback (most recent call last):

  File "C:\Users\username\AppData\Local\Temp\ipykernel_24988\3348856385.py", line 1, in
   <module>
    exu.SolveDynamic(mbs, sims)

  File "C:\Users\username\.conda\envs\venvP39\lib\site-packages\exudyn\solver.py", line
   255, in SolveDynamic
    raise ValueError("SolveDynamic terminated")
```

```
ValueError: SolveDynamic terminated
```

→ it seems clear that you should read this error from top as it indicates that you just forgot to call `mbs.Assemble()`

- WSL/Ubuntu: render window crashes after left or right mouse click:

  → This happens, in some (all?) Linux installations during mouse selection
  → workaround: setting `SC.visualizationSettings.interactive.selectionLeftMouse = False` and `SC.visualizationSettings.interactive.selectionRightMouse = False` removes the option to select with mouse, but avoids crashes

- `SolveDynamic` or `SolveStatic` **terminated due to errors**:

  → use flag `showHints = True` in `SolveDynamic` or `SolveStatic`

- Very simple example **without loads** leads to error: `SolveDynamic` or `SolveStatic` **terminated due to errors**:

  → see also 'Convergence problems', Section 2.4.15
  → may be caused due to nonlinearity of formulation and round off errors, which restrict Newton to achieve desired tolerances; adjust `.newton.relativeTolerance`/`.newton.absoluteTolerance` in static solver or in time integration

- Typical **solver error due to redundant constraints or missing inertia terms**, could read as follows:

```
========================================
SYSTEM ERROR [file 'C:\ProgramData\Anaconda3_64b37\lib\site-packages\exudyn\solver.py',
   line 207]:
CSolverBase::Newton: System Jacobian seems to be singular / not invertible!
time/load step #1, time = 0.0002
causing system equation number (coordinate number) = 42
========================================
```

  → this solver error shows that equation 42 is not solvable. The according coordinate is shown later in such an error message:

```
...
The causing system equation 42 belongs to a algebraic variable (Lagrange multiplier)
Potential object number(s) causing linear solver to fail: [7]
    object 7, name='object7', type=JointGeneric
```

  → object 7 seems to be the reason, possibly there are too much (joint) constraints applied to your system, check this object.
  → show typical REASONS and SOLUTIONS, by using `showHints=True` in `exu.SolveDynamic(...)` or `exu.SolveStatic(...)`

→ You can also **highlight** object 7 by using the following code in the iPython console:

```
SC.renderer.Start()
HighlightItem(SC,mbs,7)
```

which draws the according object in red and others gray/transparent (but sometimes objects may be hidden inside other objects!). See the command's description for further options, e.g., to highlight nodes.

- Typical **solver error if Newton does not converge**:

```
++++++++++++++++++++++++++++++
EXUDYN V1.0.200 solver: implicit second order time integration
  Newton (time/load step #1): convergence failed after 25 iterations; relative error =
  0.079958, time = 2
  Newton (time/load step #1): convergence failed after 25 iterations; relative error =
  0.0707764, time = 1
  Newton (time/load step #1): convergence failed after 25 iterations; relative error =
  0.0185745, time = 0.5
  Newton (time/load step #2): convergence failed after 25 iterations; relative error =
  0.332953, time = 0.5
  Newton (time/load step #2): convergence failed after 25 iterations; relative error =
  0.0783815, time = 0.375
  Newton (time/load step #2): convergence failed after 25 iterations; relative error =
  0.0879718, time = 0.3125
  Newton (time/load step #2): convergence failed after 25 iterations; relative error =
  2.84704e-06, time = 0.28125
  Newton (time/load step #3): convergence failed after 25 iterations; relative error =
  1.9894e-07, time = 0.28125
STEP348, t = 20 sec, timeToGo = 0 sec, Nit/step = 7.00575
solver finished after 0.258349 seconds.
```

→ this solver error is caused, because the nonlinear system cannot be solved using Newton's method.

→ the static or dynamic solver by default tries to reduce step size to overcome this problem, but may fail finally (at minimum step size).

→ possible reasons are: too large time steps (reduce step size by using more steps/second), inappropriate initial conditions, or inappropriate joints or constraints (remove joints to see if they are the reason), usually within a singular configuration. Sometimes a system may be just unsolvable in the way you set it up.

→ see also 'Convergence problems', Section 2.4.15

- Typical solver error if (e.g., syntax) **error in user function** (output may be very long, **read always message on top!**):

```
========================================
SYSTEM ERROR [file 'C:\ProgramData\Anaconda3_64b37\lib\site-packages\exudyn\solver.py',
   line 214]:
Error in Python USER FUNCTION 'LoadCoordinate::loadVectorUserFunction' (referred line
  number my be wrong!):
NameError: name 'sin' is not defined

At:
  C:\DATA\cpp\DocumentationAndInformation\tests\springDamperUserFunctionTest.py(48):
  Sweep
  C:\DATA\cpp\DocumentationAndInformation\tests\springDamperUserFunctionTest.py(54):
  userLoad
  C:\ProgramData\Anaconda3_64b37\lib\site-packages\exudyn\solver.py(214): SolveDynamic
  C:\DATA\cpp\DocumentationAndInformation\tests\springDamperUserFunctionTest.py(106): <
  module>
  C:\ProgramData\Anaconda3_64b37\lib\site-packages\spyder_kernels\customize\
  spydercustomize.py(377): exec_code
  C:\ProgramData\Anaconda3_64b37\lib\site-packages\spyder_kernels\customize\
  spydercustomize.py(476): runfile
  <ipython-input-14-323569bebfb4>(1): <module>
  C:\ProgramData\Anaconda3_64b37\lib\site-packages\IPython\core\interactiveshell.py
  (3331): run_code
...
...
; check your Python code!
========================================

Solver stopped! use showHints=True to show helpful information
```

→ this indicates an error in the user function `LoadCoordinate::loadVectorUserFunction`, because `sin` function has not been defined (must be imported, e.g., from `math`). It indicates that the error occurred in line 48 in `springDamperUserFunctionTest.py` within function `Sweep`, which has been called from function `userLoad`, etc.

1.5.2 FAQ

Some frequently asked questions:

1. When **importing** `Exudyn` in Python (windows) I get an error

   → see trouble shooting instructions above!

2. I do not understand the **Python errors** – how can I find the reason of the error or crash?

   → Read trouble shooting section above!

→ First, you should read all error messages and warnings: from the very first to the last message. Very often, there is a definite line number which shows the error. Note, that if you are executing a string (or module) as a Python code, the line numbers refer to the local line number inside the script or module.

→ If everything fails, try to execute only part of the code to find out where the first error occurs. By omiting parts of the code, you should find the according source of the error.

→ If you think, it is a bug: send an email with a representative code snippet, version, etc. to `reply.exudyn@gmail.com`

3. Spyder **console hangs** up, does not show error messages, ...:

→ very often a new start of Spyder helps; most times, it is sufficient to restart the kernel or to just press the 'x' in your IPython console, which closes the current session and restarts the kernel (this is much faster than restarting Spyder)

→ restarting the IPython console also brings back all error messages

4. Where do I find the **'.exe' file**?

→ `Exudyn` is only available via the Python interface as a module '`exudyn`', the C++ code being inside of `exudynCPP.pyd`, which is located in the exudyn folder where you installed the package. This means that you need to **run Python** (best: Spyder) and import the `Exudyn` module.

5. I get the error message 'check potential mixing of different (object, node, marker, ...) indices', what does it mean?

→ probably you used wrong item indexes, see beginning of command interface in Section 6.

→ E.g., an object number `oNum = mbs.AddObject(...)` is used at a place where a `NodeIndex` is expected, e.g., `mbs.AddObject(MassPoint(nodeNumber=oNum, ...))`

→ Usually, this is an ERROR in your code, it does not make sense to mix up these indexes!

→ In the exceptional case, that you want to convert numbers, see beginning of Section 6.

6. Why does **type auto completion** / intellisense not work for mbs (MainSystem)?

→ in earlier versions of `Exudyn` type completion did not work properly for more complex structures

→ since version 1.6.103 type completion works for most functions types and structures: tested in Spyder 5.2.2 and Visual Studio Code 1.78.1); with an added stub file (.pyi) the standard type completion fetches information about structures or functions; this even works for example with `SC.visualizationSettings.bodies.kinematicTree.showJointFrames`. If you still have problems, try to restart your environment / computer or switch to a different version, and create an issue on GitHub.

7. How to add graphics?

→ Graphics (lines, text, 3D triangular / STL mesh) can be added to all BodyGraphicsData items in objects. Graphics objects which are fixed with the background can be attached to a

ObjectGround object. Moving objects must be attached to the BodyGraphicsData of a moving body. Other moving bodies can be realized, e.g., by adding a ObjectGround and changing its reference with time. Furthermore, ObjectGround allows to add fully user defined graphics.

8. In `GenerateStraightLineANCFCable2D`

   → coordinate constraints can be used to constrain position and rotation, e.g., `fixedConstraintsNode0 = [1,1,0,1]` for a beam aligned along the global x-axis;
   → this **does not work** for beams with arbitrary rotation in reference configuration, e.g., 45°. Use a GenericJoint with a rotationMarker instead.

9. What is the difference between MarkerBodyPosition and MarkerBodyRigid?

   → Position markers (and nodes) do not have information on the orientation (rotation). For that reason, there is a difference between position based and rigid-body based markers. In case of a rigid body attached to ground with a SpringDamper, you can use both, MarkerBodyPosition or MarkerBodyRigid, markers. For a prismatic joint, you will need a MarkerBodyRigid.

10. I get an error in `exu.SolveDynamic(mbs, ...)` OR in `exu.SolveStatic(mbs, ...)` but no further information – how can I solve it?

    → Typical **time integration errors** may look like:

    ```
    File "C:/DATA/cpp/EXUDYN\_git/main/pythonDev/...<file name>", line XXX, in <module
      >
    solver.SolveSystem(...)
    SystemError: <built-in method SolveSystem of PyCapsule object at 0x0CC63590>
      returned a result with an error set
    ```

    → The pre-checks, which are performed to enable a crash-free simulation are insufficient for your model
    → As a first try, **restart the IPython console** in order to get all error messages, which may be blocked due to a previous run of `Exudyn`.
    → Very likely, you are using Python user functions inside `Exudyn`: They lead to an internal Python error, which is not always catched by `Exudyn`; e.g., a load user function UFload(mbs, t, load), which tries to access component load[3] of a load vector with 3 components will fail internally;
    → Use the print(...) command in Python at many places to find a possible error in user functions (e.g., put `print("Start user function XYZ")` at the beginning of every user function; test user functions from iPython console
    → It is also possible, that you are using inconsistent data, which leads to the crash. In that case, you should try to change your model: omit parts and find out which part is causing your error
    → see also **I do not understand the Python errors – how can I find the cause?**

11. Why can't I get the focus of the simulation window on startup (render window hidden)?

    → Starting `Exudyn` out of Spyder might not bring the simulation window to front, because of specific settings in Spyder(version 3.2.8), e.g., Tools→Preferences→Editor→Advanced settings:

uncheck 'Maintain focus in the Editor after running cells or selections'; Alternatively, set `SC.visualizationSettings.window.alwaysOnTop=True` **before** starting the renderer with `SC.renderer.Start()`

Chapter 2

Overview on Exudyn

This section provides a general overview on important parts of `Exudyn`. It is recommended to read these parts as it alleviates your life in creating models, understanding the behavior of the system and resolving errors.

2.1 Module structure

This section will show:

- Overview of modules
- Conventions: dimension of nodes, objects and vectors
- Coordinates: reference coordinates and displacements
- Nodes, Objects, Markers and Loads

For an introduction to the solvers, see Section 11.

2.1.1 Overview of modules

Currently, the `Exudyn` module structure is split into a C++ core part and a set of Python parts, see Fig. 2.1.

- **C++ parts**, see Fig. 2.2 and Fig. 2.3:
  - `exudyn`: on this level, there are just very few functions: `SystemContainer()`, `SC.renderer.Start()`, `SC.renderer.Stop()`, `SolveStatic(...)`, `SolveDynamic(...)`, ... as well as system and user variable dictionaries `exudyn.variables` and `exudyn.sys`
  - `config`, `special`: substructures for configuration (global settings) and special settings; use e.g. `exudyn.config.outputPrecision=4`
  - `symbolic`: tools for symbolic computation in user functions (speedup!)
  - `SystemContainer`: contains the systems (most important), solvers (static, dynamics, ...), visualization settings

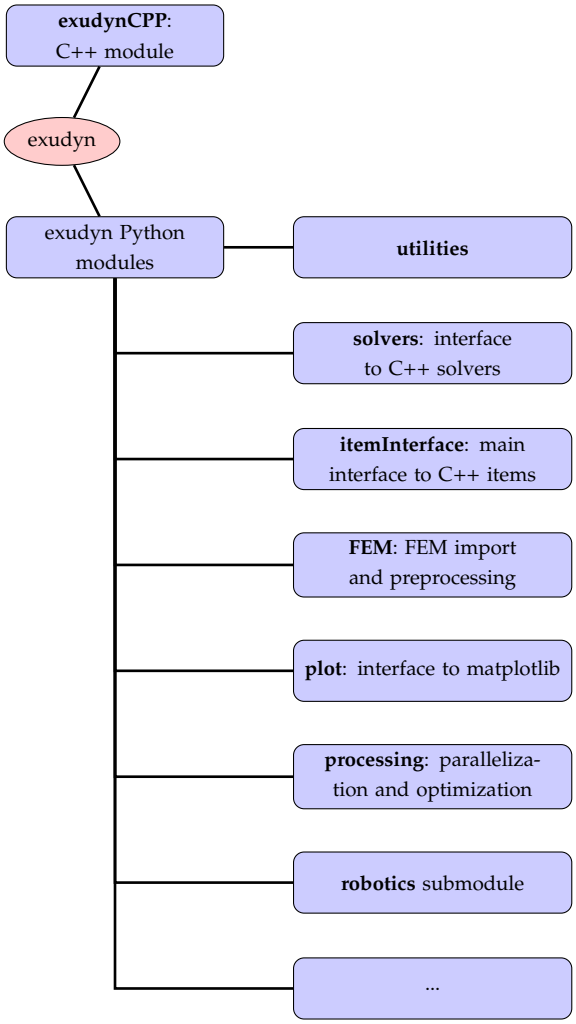


Figure 2.1: Overview of exudyn C++ and Python modules.

  - `MainSystem mbs`: multibody system (mbs) created with `mbs = SC.AddSystem()`, this structure contains everything that defines a solvable multibody system; a large set of nodes, objects, markers, loads can added to the system, see Section 8;
  - `mbs.systemData`: contains the initial, current, visualization, ... states of the system and holds the items, see Fig. 2.3
  - `SimulationSettings`: contains the systems (most important), solvers (static, dynamics, ...), visualization settings

- **Python parts** (this list is continuously extended, see Section 7):

  - `exudyn.artificialIntelligence`: interface to stablebaselines, interface to pytorch training (coming soon)
  - `exudyn.basicUtilities`: contains basic helper classes, without importing numpy
  - `exudyn.beams`: helper functions for creation of beams along straight lines and curves, sliding joints, etc.
  - `exudyn.graphics`: provides some basic drawing utilities, definition of colors and basic drawing objects (including STL import); rotation/translation of graphicsData objects
  - `exudyn.interactive`: helper classes to create interactive models (e.g. for teaching or demos)
  - `exudyn.itemInterface`: contains the interface, which transfers Python classes (e.g., of a NodePoint) to dictionaries that can be understood by the C++ module
  - `exudyn.FEM`: everything related to finite element import and creation of model order reduction flexible bodies
  - `exudyn.lieGroupBasics`: a collection of Python functions for Lie group methods (SO3, SE3, log, exp, Texp, ...)
  - `exudyn.mainSystemExtensions`: mapping of some functions to MainSystem (mbs)
  - `exudyn.physics`: containing helper functions, which are physics related such as friction
  - `exudyn.plot`: contains PlotSensor(...), a very versatile interface to matplotlib and other valuable helper functions
  - `exudyn.processing`: methods for optimization, parameter variation, sensitivity analysis, etc.
  - `exudyn.rigidBodyUtilities`: contains important helper classes for creation of rigid body inertia, rigid bodies, and rigid body joints; includes helper functions for rotation parameterization, rotation matrices, homogeneous transformations, etc.
  - `exudyn.robotics`: submodule containing several helper modules related to manipulators (`robotics`, `robotics.models`), mobile robots (`robotics.mobile`), trajectory generation (`robotics.motion`) etc.
  - `exudyn.signalProcessing`: filters, FFT, etc.; interfaces to scipy and numpy methods
  - `exudyn.solver`: functions imported when loading `exudyn`, containing main solvers
  - `exudyn.utilities`: constains helper classes in Python and includes `Exudyn` main modules `basicUtilities`, `rigidBodyUtilities`, `graphics`, and `itemInterface`, which is recommended to be loaded at beginning of your model file in order to have most necessary functionality at hand

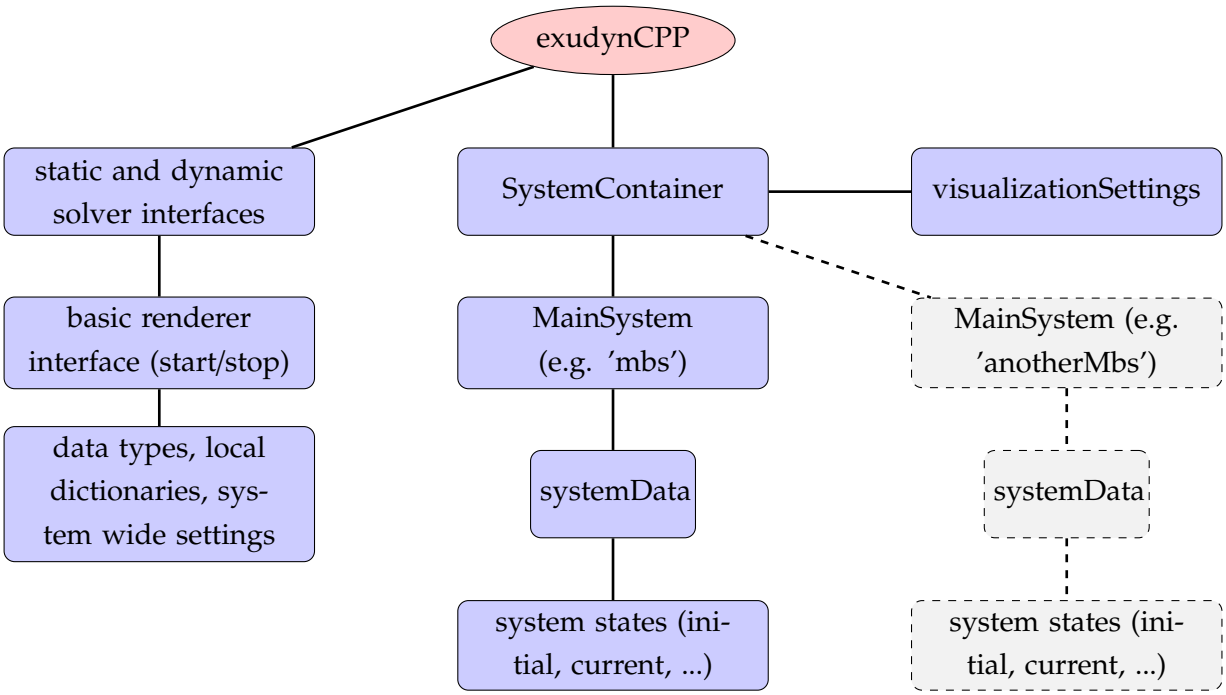


Figure 2.2: Overview of exudyn C++ module.

2.1.2 Conventions: items, indexes, coordinates

In this documentation, we will use the term **item** to identify nodes, objects, markers, loads and sensors:

item ∈ {node, object, marker, load, sensor}

Indexes: arrays and vectors starting with 0:

As known from Python, all **indexes** of arrays, vectors, matrices, ... are starting with 0. This means that the first component of the vector `v=[1,2,3]` is accessed with `v[0]` in Python (and also in the C++ part of `Exudyn`). The range is usually defined as `range(0,3)`, in which '3' marks the index after the last valid component of an array or vector.

Dimensionality of objects and vectors:

two dimensions or planar (2D) vs. three dimensions or spatial (3D)

As a convention, quantities in `Exudyn` are 3D, such as nodes, objects, markers, loads, measured quantities, etc. For that reason, we denote planar nodes, objects, etc. with the suffix 2D, but 3D objects do not get this suffix[1].

Output and input to objects, markers, loads, etc. is usually given by 3D vectors (or matrices), such as (local) position, force, torque, rotation, etc. However, initial and reference values for nodes depend on their dimensionality. As an example, consider a `NodePoint2D`:

- `referenceCoordinates` is a 2D vector (but could be any dimension in general nodes)

[1]There are some rare exceptions, such as Beam3D as the pure beam may easily lead to name space conflicts in Python

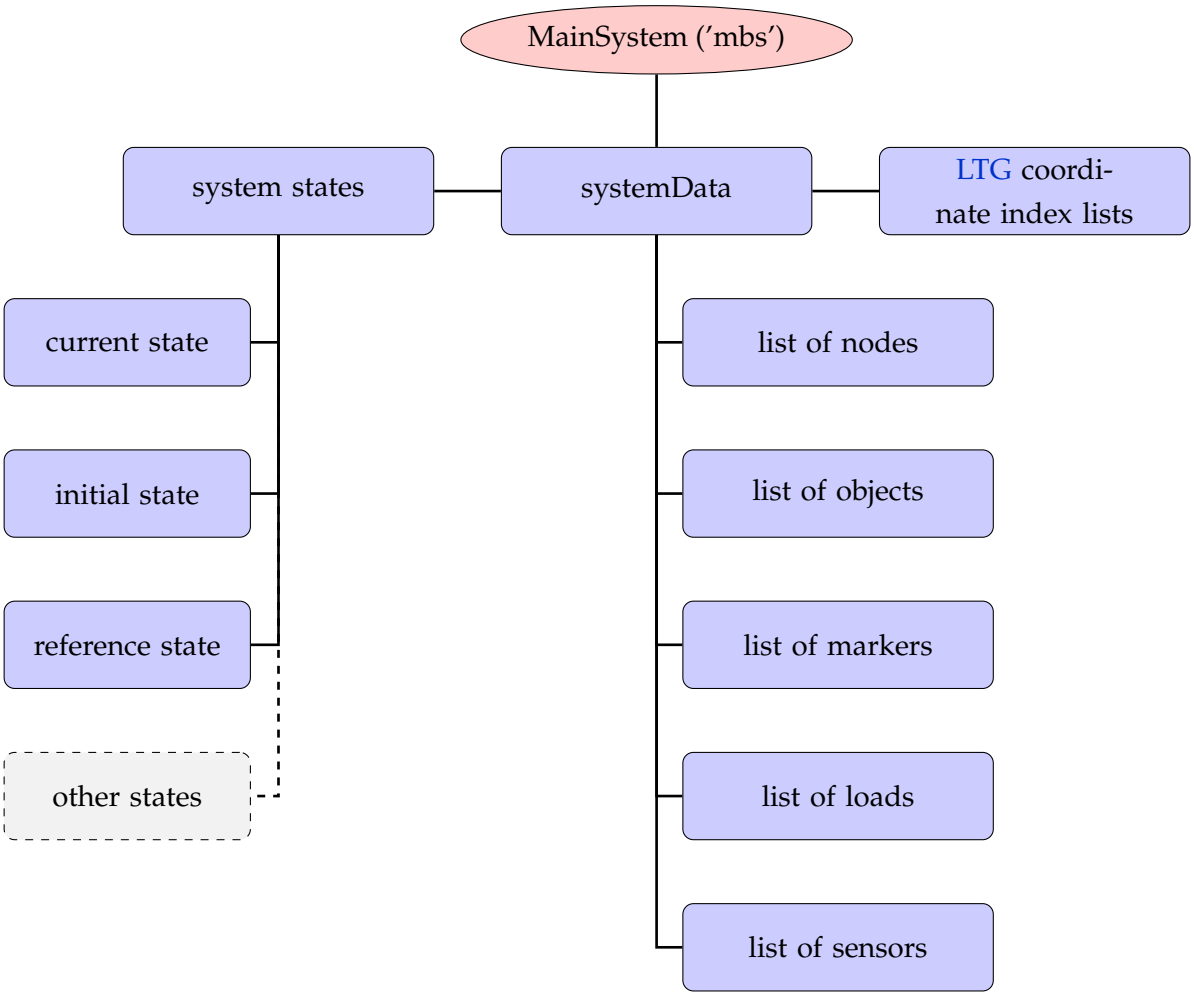


Figure 2.3: Overview of systemData, which connects items, states and stores the LTG. Note that access to items is provided via functions in `MainSystem`.

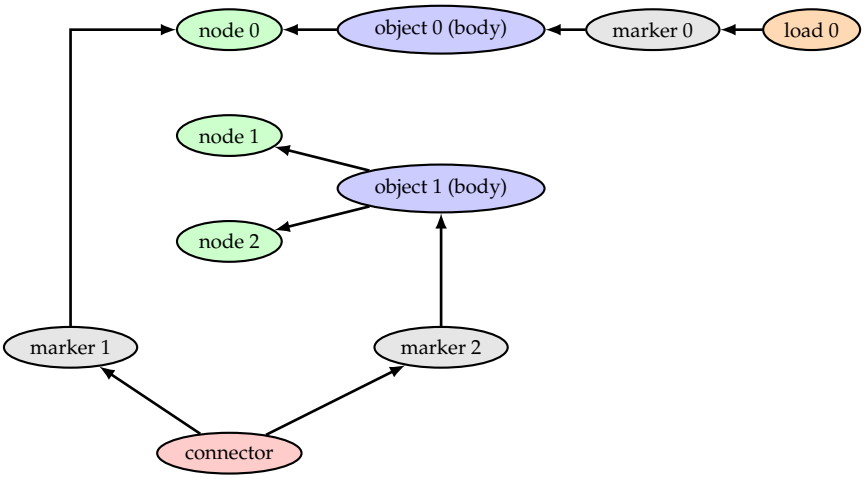


Figure 2.4: Interaction of items in a multibody system. Note that both, bodies and connectors (including constraints) are – computational – objects. The arrows indicate, that, e.g., object 1 has node 1 and node 2 (indexes) and that marker 0 is attached to object 0, while load 0 uses marker 0 to apply the load. Sensors could additionally be attached to certain items.

- measuring the current position of `NodePoint2D` gives a 3D vector
- when attaching a `MarkerNodePosition` and a `LoadForceVector`, the force will be still a 3D vector

Furthermore, the local position in 2D objects is provided by a 3D vector. Usually, the dimensionality is given in the reference manual. User errors in the dimensionality will be usually detected either by the Python interface (i.e., at the time the item is created) or by the system-preprocessor

2.2 Items: Nodes, Objects, Loads, Markers, Sensors, ...

In this section, the most important part of `Exudyn` are provided. An overview of the interaction of the items is given in Fig. 2.4

2.2.1 Nodes

Nodes provide the coordinates (and the degrees of freedom) to the system. They have no mass, stiffness or whatsoever assigned. Without nodes, the system has no unknown coordinates. Adding a node provides (for the system unknown) coordinates. In addition we also need equations for every nodal coordinate – otherwise the system cannot be computed (NOTE: this is currently not checked by the preprocessor). In general, adding nodes and objects (e.g., to represent rigid bodies), leads to a **redundant coordinate formulation**. In order to determine the degree of freedom, you may use the Grübler-Kutzbach criterion. This can also be done numerically, using the function `ComputeSystemDegreeOfFreedom`, see the module `exudyn.solver`. Furthermore, **minimal** coordinates can be used for open tree systems, using `ObjectKinematicTree`.

2.2.2 Objects

Objects are 'computational objects' and they provide equations to your system. Objects often provide derivatives and have measurable quantities (e.g. displacement) and they provide access, which can be used to apply, e.g., forces. Some of this functionality is only available in C++, but not in Python.

Objects can be a:

- general object (e.g. a controller, user defined object, ...; no example yet)
- body: has a mass or mass distribution; markers can be placed on bodies; loads can be applied; constraints can be attached via markers; bodies can be:
  - ground object: has no nodes
  - simple body: has one node (e.g. mass point, rigid body)
  - finite element and more complicated body (e.g. FFRF-object): has more than one node
- 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
  - spring-damper or penalty constraint

2.2.3 Markers

Markers are interfaces between objects/nodes and constraints/loads. A constraint (which is also an object) or load cannot act directly on a node or object without a marker. As a benefit, the constraint or load does not need to know whether it is applied, e.g., to a node or to a local position of a body.

Typical situations are:

- Node – Marker – Load
- Node – Marker – Constraint (object)
- Body(object) – Marker – Load
- Body1 – Marker1 – Joint(object) – Marker2 – Body2

2.2.4 Loads

Loads are used to apply forces and torques to the system. The load values are static values. However, you can use Python functionality to modify loads either by linearly increasing them during static computation or by using the 'mbs.SetPreStepUserFunction(...)' structure in order to modify loads in every integration step depending on time or on measured quantities (thus, creating a controller).

2.2.5 Sensors

Sensors are only used to measure output variables (values) in order to simpler generate the requested output quantities. They have a very weak influence on the system, because they are only evaluated after certain solver steps as requested by the user.

2.2.6 Reference coordinates and displacements

Nodes usually have separated reference and initial quantities. Here, `referenceCoordinates` are the coordinates for which the system is defined upon creation. Reference coordinates are needed, e.g., for definition of joints and for the reference configuration of finite elements. In many cases it marks the undeformed configuration (e.g., with finite elements), but not, e.g., for `ObjectConnectorSpringDamper`, which has its own reference length.

Initial displacement (or rotation) values are provided separately, in order to start a system from a configuration different from the reference configuration. As an example, the initial configuration of a `NodePoint` is given by `referenceCoordinates + initialCoordinates`, while the initial state of a dynamic system additionally needs `initialVelocities`. See also Section 5.1.11!

Note that commonly the `OutputVariableType Coordinates` returns coordinates without reference values, which are usually displacements (or changes in the rotation parameters), which is required if you are interested, e.g., in motion of finite element nodes. In contrast the `OutputVariableType CoordinatesTotal` returns (Since Exudyn1.9.25) the sum of reference and displacement (or rotation) coordinates for any configuration (e.g., current, initial or visualization).

2.3 Mapping between local and global coordinate indices

The local-to-global (LTG)-index-mappings[2] between local coordinate **indices**, on node or object level, and global (=system) coordinate **indices** follows the following rules:

- LTG-index-mappings are computed during `mbs.Assemble()` and are not available before.
- Nodes own a global index which relates the local coordinates to global (system) coordinate. E.g., for a second order ordinary differential equations (ODE2) node with node number `i`, this index can be obtained via the function `mbs.GetNodeODE2Index(i)`.
- The order of global coordinates is simply following the node numbering. If we add three nodes `NodePoint`, the system will contain 9 coordinates, where the first triple (starting index 0) belongs to node 0, the second triple (starting index 3) belongs to node 1 and the third triple (starting index 6) belongs to node 2. After `mbs.Assemble()`, you can access the system coordinates via `mbs.systemData.GetODE2Coordinates()`, which returns a numpy array with 9 coordinates, containing the initial values provided in `NodePoint` (default: zero).
- Objects have their own LTG-index-mappings for their respective coordinate types. The ODE2 coordinates of an object `j` can be retrieved via `mbs.systemData.GetObjectLTGODE2(j)`. For a body, these are the global ODE2 coordinates representing the body; for a connector, these are the coordinates to which the connector is linked (usually coordinates of two bodies); for a ground object, the LTG-index-mapping is empty; see also Section 6.6.2.
- Constraints create algebraic variables (Lagrange multipliers) automatically. For a constraint with object number `k`, the global index to algebraic variables (of algebraic equations (AE)-type) can be accessed via `mbs.systemData.GetObjectLTGAE(k)`.

[2] local-to-global coordinate index mappings containing transformation from local object coordinate indices to global (system) coordinate indices; this is different for **coordinate transformations**!

2.4 Exudyn Basics

This section will show:

- Interaction with the Exudyn module
- Simulation settings
- Visualization settings
- Generating output and results
- Graphics pipeline
- Generating animations

2.4.1 Interaction with the Exudyn module

It is important that the Exudyn module is basically a state machine, where you create items on the C++ side using the Python interface. This helps you to easily set up models using many other Python modules (numpy, sympy, matplotlib, ...) while the computation will be performed in the end on the C++ side in a very efficient manner.

Where do objects live?

Whenever a system container is created with `SC = exu.SystemContainer()`, the structure `SC` becomes a variable in the Python interpreter, but it is managed inside the C++ code and it can be modified via the Python interface. Usually, the system container will hold at least one system, usually called `mbs`. Commands such as `mbs.AddNode(...)` add objects to the system `mbs`. The system will be prepared for simulation by `mbs.Assemble()` and can be solved (e.g., using `exu.SolveDynamic(...)`) and evaluated hereafter using the results files. Using `mbs.Reset()` will clear the system and allows to set up a new system. Items can be modified (`ModifyObject(...)`) after first initialization, even during simulation.

2.4.2 Simulation settings

The simulation settings consists of a couple of substructures, e.g., for `solutionSettings`, `staticSolver`, `timeIntegration` as well as a couple of general options – for details see Section 9.2.0.1 and Section 9.2.

Simulation settings are needed for every solver. They contain solver-specific parameters (e.g., the way how load steps are applied), information on how solution files are written, and very specific control parameters, e.g., for the Newton solver.
The simulation settings structure is created with

```
simulationSettings = exu.SimulationSettings()
```

Hereafter, values of the structure can be modified, e.g.,

```
tEnd = 10 #10 seconds of simulation time:
h = 0.01  #step size (gives 1000 steps)
simulationSettings.timeIntegration.endTime = tEnd
#steps for time integration must be integer:
```

```
simulationSettings.timeIntegration.numberOfSteps = int(tEnd/h)
#assigns a new tolerance for Newton's method:
simulationSettings.timeIntegration.newton.relativeTolerance = 1e-9
#write some output while the solver is active (SLOWER):
simulationSettings.timeIntegration.verboseMode = 2
#write solution every 0.1 seconds:
simulationSettings.solutionSettings.solutionWritePeriod = 0.1
#use sparse matrix storage and solver (package Eigen):
simulationSettings.linearSolverType = exu.LinearSolverType.EigenSparse
```

2.4.3 Generating output and results

The solvers provide a number of options in `solutionSettings` to generate a solution file. As a default, exporting the solution of all system coordinates (on position, velocity, ... level) to the solution file is activated with a writing period of 0.01 seconds.
Typical output settings are:

```
#create a new simulationSettings structure:
simulationSettings = exu.SimulationSettings()

#activate writing to solution file:
simulationSettings.solutionSettings.writeSolutionToFile = True
#write results every 1ms:
simulationSettings.solutionSettings.solutionWritePeriod = 0.001

#assign new filename to solution file
simulationSettings.solutionSettings.coordinatesSolutionFileName= "myOutput.txt"

#do not export certain coordinates:
simulationSettings.solutionSettings.exportDataCoordinates = False
```

Furthermore, you can use sensors to record particular information, e.g., the displacement of a body's local position, forces or joint data. For viewing sensor results, use the `PlotSensor` function of the `exudyn.plot` tool, see the rigid body and joints tutorial. Finally, the render window allows to show traces (trajectories) of position sensors, sensor vector quantities (e.g., velocity vectors), or triads given by rotation matrices. For further information, see the `sensors.traces` structure of `VisualizationSettings`, Section 9.3.0.10.

2.4.4 Renderer and 3D graphics

A 3D renderer is attached to the simulation. Visualization is started with `SC.renderer.Start()`, see the examples and tutorials. In order to show your model in the render window, you have to provide 3D graphics data to the bodies. Flexible bodies (e.g., FFRF-like) can visualize their meshes. Further items (nodes, markers, ...) can be visualized with default settings, however, often you have to turn on drawing or enlarge default sizes to make items visible. Item number can also be shown.

Finally, since version 1.6.188, sensor traces (trajectories) can be shown in the render window, see the `VisualizationSettings` in Section 9.3.

The renderer uses an OpenGL window of a library called GLFW, which is platform-independent. The renderer is set up in a minimalistic way, just to ensure that you can check that the modeling is correct.

Note:

- For closing the render window, press key 'Q' or Escape or just close the window.
- There is no way to contruct models inside the renderer (no 'GUI').
- Try to avoid huge number of triangles in STL files or by creating large number of complex objects, such as spheres or cylinders.
- After `visualizationSettings.window.reallyQuitTimeLimit` seconds a 'do you really want to quit' dialog opens for safety on pressing 'Q'; if no tkinter is available, you just have to press 'Q' twice. For closing the window, you need to click a second time on the close button of the window after `reallyQuitTimeLimit` seconds (usually 900 seconds).

Here are the **main features of the renderer**, using keyboard and mouse, for details see Section 10:

- press key H to show help in renderer
- move model by pressing left mouse button and drag
- rotate model by pressing right mouse button and drag
- for further mouse functionality, see Section 10.1
- change visibility (wire frame, solid, transparent, ...) by pressing T
- zoom all: key A
- open visualization dialog: key V, see Section 2.4.5
- open Python command dialog: key X, see Section 2.4.6
- show item number: click on graphics element with left mouse button
- show item dictionary: click on graphics element with right mouse button
- for further keys, see Section 10.2 or press H in renderer
- raytracing mode, see Section 2.4.8

Depending on your model (size, place, ...), you **may need to adjust the following general visualization** and `openGL` **parameters** in `visualizationSettings`, see Section 9.3:

- change window size
- light and light position; switch `openGL.lightPositionsInCameraFrame` to switch between model-fixed or camera-fixed lights
- shadow (turned off by using shadow=0; turned on by using, e.g., a value of 0.3) and shadow polygon offset; shadow slows down graphics performance by a factor of 2-3, depending on your graphics card
- visibility of nodes, markers, etc. in according bodies, nodes, markers, ..., `visualizationSettings`
- move camera with a selected marker: adjust `trackMarker` in `visualizationSettings.interactive`

NOTE: changing `visualizationSettings` is not thread-safe, as it allows direct access to the C++ variables. In most cases, this is not problematic, e.g., turning on/off some view parameters my just lead to some short-time artifacts if they are changed during redraw. However, more advanced quantities (e.g., `trackMarker` or changing strings) may lead to problems, which is why it is strongly recommended to:

- set all `visualizationSettings` **before start of renderer**

2.4.5 Visualization settings dialog

Visualization settings are used for user interaction with the model. E.g., the nodes, markers, loads, etc., can be visualized for every model. There are default values, e.g., for the size of nodes, which may be inappropriate for your model. Therefore, you can adjust those parameters. In some cases, huge models require simpler graphics representation, in order not to slow down performance – e.g., the number of faces to represent a cylinder should be small if there are 10000s of cylinders drawn. Even computation performance can be slowed down, if visualization takes lots of CPU power. However, visualization is performed in a separate thread, which usually does not influence the computation exhaustively.

Details on visualization settings and its substructures are provided in Section 9.3. These settings may also be edited by pressing 'V' in the active render window (does not work, if there is no active render loop using, e.g., `SC.renderer.DoIdleTasks()`). The visualization settings dialog is shown exemplarily in Fig. 2.5. Note that this dialog is automatically created and uses Python's `tkinter`, which is lightweight, but not very well suited if display scalings are large (e.g., on high resolution laptop screens). If working with Spyder, it is recommended to restart Spyder, if display scaling is changed, in order to adjust scaling not only for Spyder but also for `Exudyn`.

The appearance of visualization settings dialogs may be adjusted by directly modifying `exudyn.GUI` variables (this may change in the future). For example write in your code before opening the render window[3]:

```
import exudyn.GUI
exudyn.GUI.dialogDefaultWidth             #unscaled width of, e.g., right-mouse-button
  dialog
exudyn.GUI.treeEditDefaultWidth = 800
exudyn.GUI.treeEditDefaultHeight = 600
exudyn.GUI.treeEditMaxInitialHeight = 600 #otherwise height is increased for larger
  screens
exudyn.GUI.treeEditOpenItems = ['general','contact'] #these tree items are opened each
  time the dialog is opened
#
exudyn.GUI.treeviewDefaultFontSize        #this is the base font size of the dialog (also
  right-mouse-button dialog)
exudyn.GUI.useRenderWindowDisplayScaling  #if True, the scaling will follow the current
  scaling of the render window; if False, it will use the \texttt{tkinter} internal
```

[3]treeEdit and treeview both mean the settings dialog currently used for visualization settings and partially for right-mouse-click

```
  scaling, which uses the main screen where the dialog is created (which won't scale well,
   if the window is moved to another screen).
#
exudyn.GUI.textHeightFactor = 1.45        #this factor is used to increase height of lines
   in tree view as compared to font size
```

The visualization settings structure can be accessed in the system container SC (access per reference, no copying!), accessing every value or structure directly, e.g.,

```
SC.visualizationSettings.nodes.defaultSize = 0.001      #draw nodes very small

#change openGL parameters; current values can be obtained from SC.renderer.GetState()
#change zoom factor:
SC.visualizationSettings.openGL.initialZoom = 0.2
#set the center point of the scene (can be attached to moving object):
SC.visualizationSettings.openGL.initialCenterPoint = [0.192, -0.0039,-0.075]

#turn of auto-fit:
SC.visualizationSettings.general.autoFitScene = False

#change smoothness of a cylinder:
SC.visualizationSettings.general.cylinderTiling = 100

#make round objects flat:
SC.visualizationSettings.openGL.shadeModelSmooth = False

#turn on coloured plot, using y-component of displacements:
SC.visualizationSettings.contour.outputVariable = exu.OutputVariableType.Displacement
SC.visualizationSettings.contour.outputVariableComponent = 1 #0=x, 1=y, 2=z
```

2.4.6 Execute Command and Help

In addition to the Visualization settings dialog, a simple help window opens upon pressing key 'H'. It is also possible to execute single Python commands during simulation by pressing 'X', which opens a dialog, saying 'Exudyn Command Window'. Note that the dialog may appear behind the visualization window! This dialog may be very helpful in long running computations or in case that you may evaluate variables for debugging. The Python commands are evaluated in the global python scope, meaning that `mbs` or other variables of your scripts are available. User errors are caught by exceptions, but in severe cases this may lead to crash. To print values, always use `print(...)` to see the string representation of an object.
Useful examples (single lines) may be:

```
x=5 #or change any other variable used in Python user functions
print(mbs) #print current mbs overview
print(mbs.GetSensorValues(0))
#adjust simulation end time, in long-run simulations:
mbs.sys['dynamicSolver'].it.endTime = 1
```

Figure 2.5: View of visualization settings (as of Exudyn 1.5.51.dev1) (press 'V' in render window to open dialog)

```
#adjust output behavior
mbs.sys['dynamicSolver'].output.verboseMode = 0
```

You can also do quite fancy things during simulation, e.g., to deactivate joints (of course this may result in strange behavior):

```
n=mbs.systemData.NumberOfObjects()
for i in range(n):
    d = mbs.GetObject(i)
    #if 'Joint' in d['objectType']:
    if 'activeConnector' in d:
        mbs.SetObjectParameter(i, 'activeConnector', False)
```

Note that you could also change `visualizationSettings` in this way, but the Visualization settings dialog is much more convenient. Changing `simulationSettings` within the execute command is dangerous and must be treated with care. Some parameters, such as `simulationSettings.timeIntegration.end` are copied into the internal solver's `mbs.sys['dynamicSolver'].it` structure.

Thus, changing `simulationSettings.timeIntegration.endTime` has no effect during simulation. As a rule of thumb, all variables that are not stored inside the solvers structures may be adjusted by the `simulationSettings` passed to the solver (which are then not copied internally); see the C++ code for details. However, behavior may change in future and unexpected behavior or and changing `simulationSettings` will likely cause crashes if you do not know exactly the behavior, e.g., changing output format from text to binary ... ! Specifically, `newton` and `discontinuous` settings cannot be changed on the fly as they are copied internally.

2.4.7 Graphics pipeline

There are basically two loops during simulation, which feed the graphics pipeline. The solver runs a loop:

- compute step (or set up initial values)
- finish computation step; results are in current state
- copy current state to visualization state (thread safe)
- signal graphics pipeline that new visualization data is available
- the renderer may update the visualization depending on `graphicsUpdateInterval` in `visualizationSettings.general`

The openGL graphics thread (=separate thread) runs the following loop:

- render openGL scene with a given graphicsData structure (containing lines, faces, text, ...)
- go idle for some milliseconds
- check if openGL rendering needs an update (e.g. due to user interaction)
  → if update is needed, the visualization of all items is updated – stored in a graphicsData structure)
- check if new visualization data is available and the time since last update is larger than a presribed value, the graphicsData structure is updated with the new visualization state

2.4.8 Raytracing

In order to compensate the limited functionality (but high compatibility) of OpenGL 1.3, an option for CPU-based software rendering (raytracing) has been added. This allows to include shadows and transparency correctly, with additional support for relections, refraction, emission, fog and materials. In the future, textures may be added as well.

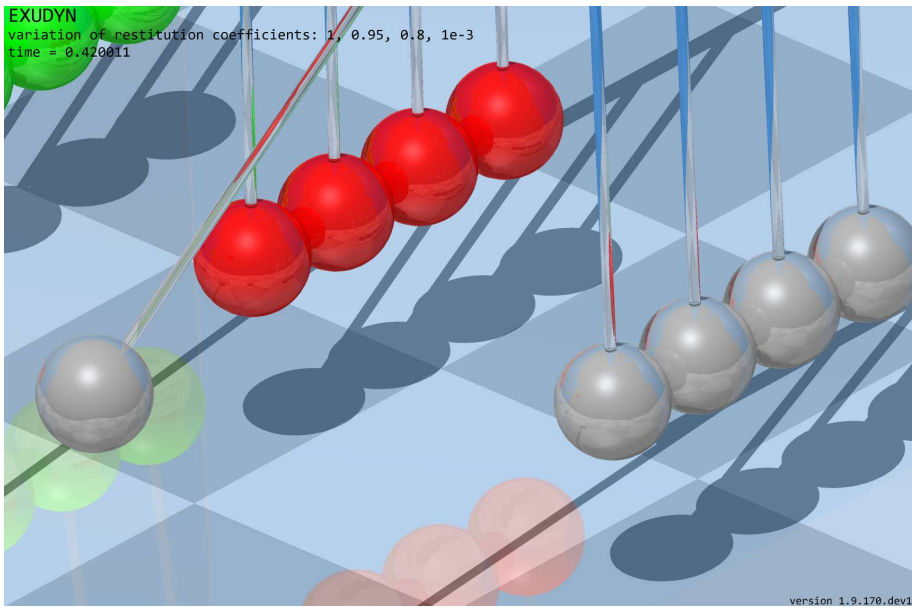


Figure 2.6: Example image of raytraced renderer view.

The basic things to know are:

- Raytracing settings are collected in `SC.visualizationSettings.raytracer` (in the following, we omit 'SC.visualizationSettings').
- Raytracing is activated by setting `raytracer.enable=True`. Please make sure that you start with small render window sizes / complexity first.
- The render window size is adjusted by `window.renderWindowSize`. Be careful with this settings.
- Adjust the `raytracer.numberOfThreads` for optimal performance, use `raytracer.verbose` to see render times for different settings. For testing, use `raytracer.imageSizeFactor`>1 to decrease the raytracer's resolution (with same image size), while `openGL.multiSampling` will increase the resolution (anti-aliasing). Note that switching from `openGL.multiSampling=1` and `raytracer.imageSizeFactor=4` to `openGL.multiSampling=3` and `raytracer.imageSizeFactor=1` increases computational costs by a factor $4 \times 4 \times 3 \times 3 = 144$. Use only one light, if sufficient (set `openGL.enableLight1=False`).
- Scene, lights, shadow, clipping plane, etc. settings are taken form OpenGL settings and directly used in the software renderer, like `openGL.light0position`, `openGL.shadow`, `openGL.perspective`, `openGL.clippingPlaneNormal`, `openGL.showLines`, etc.; some settings are in general, like `general.backgroun` or `general.drawWorldBasis`;

- In order to see the advantages of the software renderer, materials have to be used, see below.

Materials:

- Materials have the type `VSettingsMaterial`, see description in Section 9.3.0.15, for adjusting color, reflectivity, shininess, alpha-transparency, etc.;
- Materials can only be used within triangulated geometries (GraphicsData `TriangleList`) using a material-flag in the color, like `graphics.Sphere(..., color=graphics.material.chrome)`. In the RGBA color, the alpha-channel is replaced by a material index which starts at 1000 (where 1000 represents material index 0). Note that in the regular OpenGL-rendering, alpha>1 is equivalent to alpha=1. The first 10 materials are linked to `raytracer.material0 ... raytracer.material9`.
- The material's `baseColor` is used if the color red-channel is set to −1. Note that this allows to globally change the color of objects by changing `baseColor` in the material settings in `visualizationSettings`. Summarizing, using `color=[1,0,0,graphics.material.indexSteel]` chooses red color with steel material settings, while `color=[-1,-1,-1,graphics.material.indexSteel]` will use the color of steel (but will be black for OpenGL renderer), identical with `color=graphics.material.steel`.
- The setting `backgroundColorReflections` can be used to represent the background which is used for rendering, while the background is independently set to black or white. Otherwise, black background leads to black regions on highly reflective objects or very light regions for white backgrounds.
- System text messages (solver, version, etc.) are overlayed over raytracing and can be turned off using the settings in `general.showComputationInfo` and similar. However, note that **item texts are currently not shown** in raytracer, affecting node numbers, etc.!

Limitations and risks:

- Raytracing is CPU-based and therefore slow. Do not use very high resolution (4K) together with multisampling > 1. Start with small render window sizes (e.g. 600 × 400)
- Raytracing usually uses multithreading with speedups > 10 on 16 cores. However, this cannot be combined with multithreaded simulations. It is therefore recommended to use raytracing in the solution viewer, not during simulation.
- If software rendering of a single frame gets to long (>4 seconds), timeouts become active and it may occasionally not work. There are some options to compensate, see above.
- In general, it is **recommended to start with default settings and experiment** with changes using the visualization settings dialog.

To add raytracing to your project, do like this:

```
...
#sphere with chrome
graphics.Sphere(radius=radius,
                color=graphics.color.dodgerblue[0:3]+[graphics.material.indexChrome],
                nTiles=32)
ground = mbs.CreateGround(referencePosition=[0,0,0],
                          graphicsDataList=[gSphere])
```

```
#add mbs components
#assemble
#solve
...
#after computation, switch to raytracing
SC.visualizationSettings.openGL.multiSampling = 1
#SC.visualizationSettings.openGL.imageSizeFactor = 4 #reduce resolution for first tests!
SC.visualizationSettings.openGL.enableLight1 = False
SC.visualizationSettings.raytracer.numberOfThreads = 16 #adjust to your n-threads
SC.visualizationSettings.raytracer.enable = True

mbs.SolutionViewer()
```

Have fun!

2.4.9 Storing the model view

There is a simple way to store the current view (zoom, centerpoint, orientation, etc.) by using `SC.renderer.GetState()` and `SC.renderer.SetState()`, see also Section 10.3. A simple way is to reload the stored render state (model view) after simulating your model once at the end of the simulation[4]:

```
import exudyn as exu
SC=exu.SystemContainer()
SC.visualizationSettings.general.autoFitScene = False #prevent from autozoom
SC.renderer.Start()
if 'renderState' in exu.sys:
    SC.renderer.SetState(exu.sys['renderState'])
#+++++++++++++++
#do simulation here and adjust model view settings with mouse
#+++++++++++++++

#store model view for next run:
SC.renderer.Stop() #stores render state in exu.sys['renderState']
```

Alternatively, you can obtain the current model view from the console after a simulation, e.g.,

```
In[1] : SC.renderer.GetState()
Out[1]:
{'centerPoint': [1.0, 0.0, 0.0],
 'maxSceneSize': 2.0,
 'zoom': 1.0,
 'currentWindowSize': [1024, 768],
 'modelRotation': [[ 0.34202015,  0.        , 0.9396926 ],
```

[4] note that `visualizationSettings.general.autoFitScene` should be set False if you want to use the stored zoom factor

```
                    [-0.60402274,  0.76604444, 0.21984631],
                    [-0.7198463 , -0.6427876 , 0.26200265]])}
```

which contains the last state of the renderer. Now copy the output and set this with `SC.renderer.SetState` in your Python code to have a fixed model view in every simulation (`SC.renderer.SetState` AFTER `SC.renderer.Start()`):

```
SC.visualizationSettings.general.autoFitScene = False #prevent from autozoom
SC.renderer.Start()
renderState={'centerPoint': [1.0, 0.0, 0.0],
             'maxSceneSize': 2.0,
             'zoom': 1.0,
             'currentWindowSize': [1024, 768],
             'modelRotation':     [[ 0.34202015,  0.        ,  0.9396926 ],
                                   [-0.60402274,  0.76604444,  0.21984631],
                                   [-0.7198463 , -0.6427876 ,  0.26200265]])
SC.renderer.SetState(renderState)
#.... further code for simulation here
```

Note that in the current version of `Exudyn` there is more data stored in render state, which is not used in `SC.renderer.SetState`, see also Section 10.3.

2.4.10 Graphics user functions via Python

There are some user functions in order to customize drawing:

- You can assign graphicsData to the visualization to most bodies, such as rigid bodies in order to change the shape. Graphics can also be imported from files (`exu.graphics.FromSTLfileASCII`, `exu.graphics.FromSTLfile`,) using the established format STereoLithography (STL)[5].
- Some objects, e.g., `ObjectGenericODE2` or `ObjectRigidBody`, provide customized a function `graphicsDataUserFunction`. This user function just returns a list of GraphicsData, see Section 10.4. With this function you can change the shape of the body in every step of the computation.
- Specifically, the `graphicsDataUserFunction` in `ObjectGround` can be used to draw any moving background in the scene.

Note that all kinds of `graphicsDataUserFunction`s need to be called from the main (=computation) process as Python functions may not be called from separate threads (GIL). Therefore, the computation thread is interrupted to execute the `graphicsDataUserFunction` between two time steps, such that the graphics Python user function can be executed. There is a timeout variable for this interruption of the computation with a warning if scenes get too complicated.

2.4.11 Color, RGBA and alpha-transparency

Many functions and objects include color information. In order to allow alpha-transparency, all colors contain a list of 4 RGBA values, all values being in the range [0..1]:

[5] STereoLithography or Standard Triangle Language; file format available in nearly all CAD systems

- red (R) channel
- green (G) channel
- blue (B) channel
- alpha (A) value, representing the so-called **alpha-transparency** (A=0: fully transparent, A=1: solid)

E.g., red color with no transparency is obtained by the color=[1,0,0,1]. Color predefinitions are found in `graphics.py`, e.g., using `graphics.color.red` or `graphics.color.steelblue` as well a list of 16 colors `graphics.colorList`, which is convenient to be used in a loop creating objects. Earlier, special colors were given in `exudyn.graphicsDataUtilities.py`, e.g., `color4red` or `color4steelblue` as well as `color4list`, which are marked as deprecated.

2.4.12 Solution viewer

Exudyn offers a convenient WYSIWYS – 'What you See is What you Simulate' interface, showing you the computation results during simulation in the render window. If you are running large models, it may be more convenient to watch results after simulation has been finished. For this, you can use

- `interactive.SolutionViewer`, see Section 6.5.2
- `interactive.AnimateModes`, lets you view the animation of computed modes, see Section 7.11

shown exemplary in Fig. 2.7. The `SolutionViewer` adds a `tkinter` interactive dialog, which lets

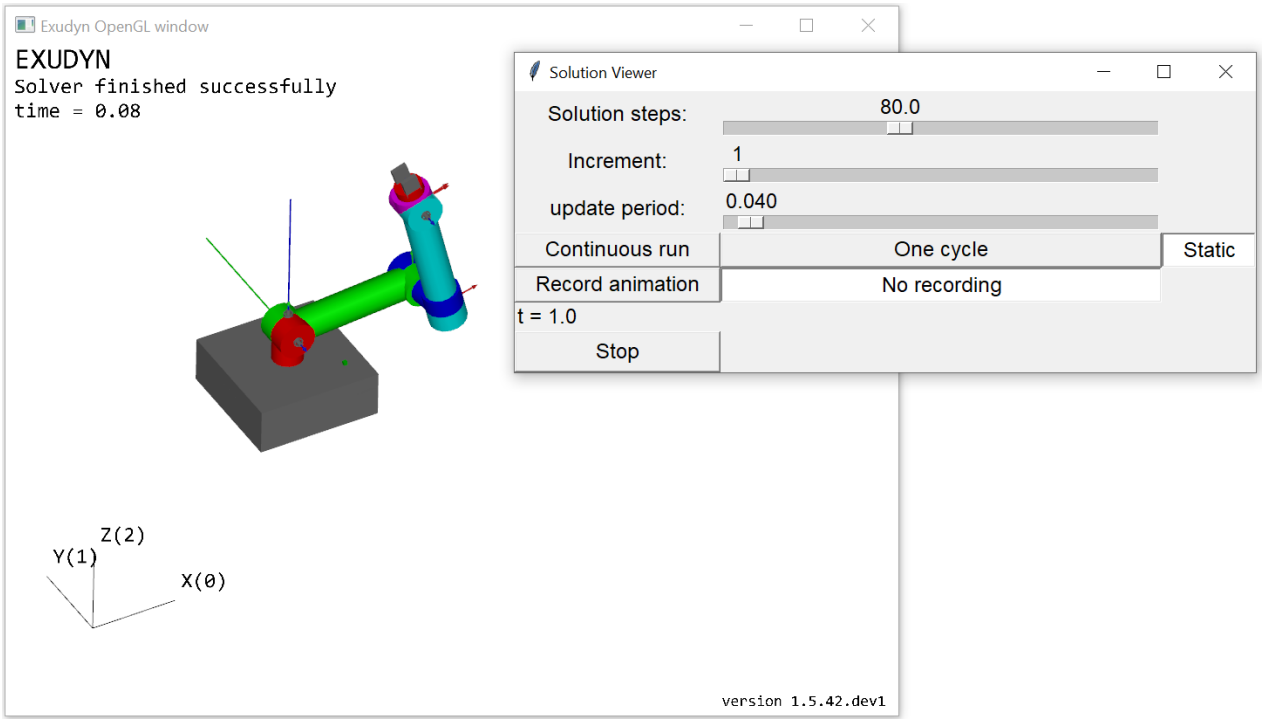


Figure 2.7: View of `SolutionViewer` (as of Exudyn 1.5.42.dev1)

you interact with the model, with the following features:

- The SolutionViewer represents a 'Player' for the dynamic solution or a series of static solutions, which is available after simulation if `solutionSettings.writeSolutionToFile = True`
- The parameter `solutionSettings.solutionWritePeriod` represents the time period used to store solutions during dynamic computations.
- As soon as 'Run' is pressed, the player runs (and it may be started automatically as well)
- In the 'Static' mode, drag the slider 'Solution steps' to view the solution steps
- In the 'Continuous run' mode, the player runs in an infinite loop
- In the 'One cycle' mode, the player runs from the current position to the end; this is perfectly suited to record series of images for **creating animations**, see Section 2.4.13 and works together with the visualization settings dialog.
- In the 'Record animation' mode, the player records frames that are shown in the render window; before pressing on 'Record animation', press 'Stop' and switch to 'One cycle'. Then put the solution steps slider to the first frame and press 'Record animation', which stores images in the current subfolder 'images' as 'frame00001.png' with increasing number, using PNG by default. The number is increased and can only be reset after new start of SolutionViewer.
- Since `Exudyn` V1.9.83, the button 'Make mp4' allows to directly generate animation files, see next section.

The solution should be loaded with `LoadSolutionFile('coordinatesSolution.txt')`, where 'coordinatesSolution.txt' represents the stored solution file, see

- `exu.SimulationSettings().solutionSettings.coordinatesSolutionFileName`

You can call the `SolutionViewer` either in the model, or at the command line / IPython to load a previous solution (belonging to the same mbs underlying the solution!):

```
from exudyn.utilities import LoadSolutionFile
sol = LoadSolutionFile('coordinatesSolution.txt')
mbs.SolutionViewer(solution=sol)
```

By default and as a recommended way, if no solution is provided, `SolutionViewer` tries to reload the solution of the previous simulation that is referred to from `mbs.sys['simulationSettings']`:

```
#... mbs has been previously solved
mbs.SolutionViewer()
```

An example for the `SolutionViewer` is integrated into the `Examples/` directory, see `solutionViewerTest.py`.

2.4.13 Generating animations

In many dynamics simulations, it is very helpful to create animations in order to better understand the motion of bodies. Specifically, the animation can be used to visualize the model much slower or faster than the model is computed.

Animations are created based on a series of images (frames, snapshots) taken during simulation. It is important, that the current view is used to record these images – this means that the view should

not be changed during the recording of images. The easiest way to create animations, is using the SolutionViewer with its integrated features, see Section 2.4.12.

To turn on recording of images during solving, set the following flag to a positive value

- `simulationSettings.solutionSettings.recordImagesInterval = 0.01`

which means, that after every 0.01 seconds of simulation time, an image of the current view is taken and stored in the directory and filename (without filename ending) specified by

- `SC.visualizationSettings.exportImages.saveImageFileName = "myFolder/frame"`

By default, a consecutive numbering is generated for the image, e.g., 'frame0000.png, frame0001.png,...'. Note that the standard file format PNG with ending '.png' uses compression libraries included in glfw, while the alternative TGA format produces '.tga' files which contain raw image data and therefore can become very large.

To create animation files, an external tool FFMPEG is used to efficiently convert a series of images into an animation. Since `Exudyn` V1.9.83, ffmpeg is integrated into the solution viewer (button 'Make mp4'), which requires prior installation using `pip install ffmpeg-python`. Note that you may also need to install ffmpeg itself, depending on your platform. In Windows, simple DOS batch files can do the job to convert frames given in the local directory to animations, e.g.:

```
echo off
REM 2019-12-23, Johannes Gerstmayr
REM helper file for EXUDYN to convert all frame00000.png, frame00001.png, ...  files to a video
REM for higher quality use crf option (standard: -crf 23, range: 0-51, lower crf value means higher
    quality)

IF EXIST animation.mp4 (
    echo "animation.mp4 already exists! rename the file"
) ELSE (
    "C:\Program Files (x86)\FFMPEG\bin\ffmpeg.exe" -r 25 -start_number 0 -i frame%%05d.png -c:v
    libx264 -vf "fps=25,format=yuv420p" animation.mp4
)
```

After the video has been created, you should delete the single images:

```
REM 2022-03-28, Johannes Gerstmayr
REM helper file for EXUDYN
REM delete all .png images of current directory

del *.png
```

2.4.14 Examples, test models and test suite

The main collection of examples and models is available under

- `main/pythonDev/Examples`
- `main/pythonDev/TestModels`

You can use these examples to build up your own realistic models of multibody systems. Very often, these models show the way which already works. Alternative ways may exist, but sometimes there are limitations in the underlying C++ code, such that they won't work as you expect.

We would like to note that, even that some examples and test models contain comparison to papers of the literature or analytical solutions, there are many models which may not contain real mechanical values and these models may not be converged in space or time (in order to keep running our test suite in less than a minute).

Finally, note that the `main/pythonDev/TestModels` are often only intended to preserve functionality in the Python and C++ code (e.g., if global methods are changed), but they should not be misinterpreted as validation of the implemented methods. The `TestModels` are used in the `Exudyn`**TestSuite** `TestModels/runTestSuite.py` which is run after a full build of Python versions. Output for very version is written to `main/pythonDev/TestSuiteLogs` containing the `Exudyn` version and Python version. At the end of these files, a summary is included to show if all models completed successfully (which means that a certain error level is achieved, which is rather small and different for the models). There are also performance tests (e.g., if a certain implementation leads to a significant drop of performance). However, the output of the performance tests is not stored on github.

We are trying hard to achieve error-free algorithms of physically correct models, but there may always be some errors in the code.

2.4.15 Removing convergence problems and solver failures

Nonlinear formulations (such as most multibody systems, especially nonlinear finite elements) cause problems and there is no general nonlinear solver which may reliably and accurately solve such problems. Tuning solver parameters is at hand of the user. In general, the Newton solver tries to reduce the error by the factor given in

- `simulationSettings.staticSolver.newton.relativeTolerance` (for static solver),

which is not possible for very small (or zero) initial residuals. The absolute tolerance is helping out as a lower bound for the error, given in

- `simulationSettings.staticSolver.newton.absoluteTolerance` (for static solver),

which is by default rather low (1e-10) – in order to achieve accurate results for small systems or small motion (in mm or μm regime). Increasing this value helps to solve such problems. Nevertheless, you should usually set tolerances as low as possible because otherwise, your solution may become inaccurate.
The following hints / rules for described problems shall be followed.

- **static solver**: **load steps get very small** even if the solution seems to be smooth (or linear) and less steps are expected:

→ this may happen for **system without loads**; larger number of steps may happen for finer discretization;

→ you may adjust (increase) `.newton.relativeTolerance` / `.newton.absoluteTolerance` in static solver or in time integration to resolve such problems, but check if solution achieves according accuracy

- **static solver**: load steps are reduced significantly for **highly nonlinear problems**:

  → solver repeatedly writes that steps are reduced → try to use `loadStepGeometric` and use a large `loadStepGeometricRange`: this allows to start with very small loads in which the system is nearly linear (e.g. for thin strings or belts under gravity).

- **static solver**: system is (nearly) **kinematic**:

  → a static solution can be achieved using `stabilizerODE2term`, which adds mass-proportional stiffness terms during load steps < 1; see also hints for singular Jacobians below

- very small loads or even **zero loads** do not converge: `SolveDynamic` or `SolveStatic` **terminated due to errors**

  → the reason is the nonlinearity of formulations (nonlinear kinematics, nonlinear beam, etc.) and round off errors, which restrict Newton to achieve desired tolerances

  → adjust (increase) `.newton.relativeTolerance` / `.newton.absoluteTolerance` in static solver or in time integration

  → in many cases, especially for static problems, the `.newton.newtonResidualMode = 1` evaluates the increments; the nonlinear problems is assumed to be converged, if increments are within given absolute/relative tolerances; this also works usually better for kinematic solutions

- for **discontinuous problems**:

  → try to adjust solver parameters; especially the `discontinuous.iterationTolerance` and `discontinuous.maxIterations`; try to make smaller load or time steps in order to resolve switching points of contact or friction; generalized alpha solvers may cause troubles when reducing step sizes → use TrapezoidalIndex2 solver

  → in case of **user functions**, make sure that there is no switching inside the user function (if or `sign` function); switching must be done in the PostNewtonStep, otherwise convergence severely suffers

- **singular Jacobians** or **redundant constraints**:

  → in case of systems that lead to a singular Jacobian due to redundant constraints or kinematic DOF in static solutions, you may switch to Eigen's FullPivotLU solver using:

  → `simulationSettings.linearSolverType = exu.LinearSolverType.EigenDense` and

  → `simulationSettings.linearSolverSettings.ignoreSingularJacobian=True` ;

  → however, check your results as they may be erroneous, because the solver tries to find an optimal solution / compromise which may not be what you intend to get!

- if you see further problems, please post them (including relevant example) at the Exudyn github page!

2.4.16 Performance and ways to speed up computations

Multibody dynamics simulation should be accurate and reliable on the one hand side. Most solver settings are such that they lead to comparatively reliable results. However, in some cases there is a significant possibility for speeding up computations, which are described in the following list. Not all recommendations may apply to your models.

The following examples refer to `simulationSettings = exu.SimulationSettings()`. In general, to see where CPU time is lost, use the option turn on `simulationSettings.displayComputationTime = True` to see which parts of the solver need most of the time (deactivated in exudynFast versions!). In addition to Exudyn's internal time measurements, in Spyder (or IPython) you can use magic commands such as `%timeit -n10 mbs.SolveDynamic()` to evaluate the time spent for a specific command with number of repetitions given after `-n`. This may be particularly interesting in Python user functions to see where time is lost.

To activate the Exudyn C++ versions without range checks, which may be approx. 30 percent faster in some situations, use the following code snippet before first import of exudyn:

```
import sys
sys.exudynFast = True #this variable is used to signal to load the fast exudyn module
import exudyn as exu
```

The faster versions are available for all release versions, but only for some `.dev1` development versions (Python 3.10), which can be determined by trying `import exudyn.exudynCPPfast`.
However, there are many **ways to speed up Exudyn in general**:

- for models with more than 50 coordinates, switching to sparse solvers might greatly improve speed: `simulationSettings.linearSolverType = exu.LinearSolverType.EigenSparse`
- when preferring dense direct solvers, switching to Eigen's PartialPivLU solver might greatly improve speed: `simulationSettings.linearSolverType = exu.LinearSolverType.EigenDense`; however, the flag `simulationSettings.linearSolverSettings.ignoreSingularJacobian=True` will switch to the much slower (but more robust) Eigen's FullPivLU
- try to avoid Python functions or try to speed up Python functions; if this is not possible, see solutions below
- instead of user functions in objects or loads (computed in every iteration), some problems would also work if these parameters are only updated in `mbs.SetPreStepUserFunction(...)`
- Python user functions can be speed up (since Exudyn V1.7.40) by converting conventional Python functions into Exudyn (internal) symbolic user functions, which have similar performance as C++ functions with the ability to parallelize; see Section 6.7
- Alternatively, Python user functions can be speed up using the Python numba package, using `@jit` in front of functions (for more options, see https://numba.pydata.org/numba-doc/dev/user/index.html); Example given in `Examples/springDamperUserFunctionNumbaJIT.py` showing speedups of factor 4; more complicated Python functions may see speedups of 10 - 50

- for **discontinuous problems**, try to adjust solver parameters; especially the discontinuous.iterationTolerance which may be too tight and cause many iterations; iterations may be limited by discontinuous.maxIterations, which at larger values solely multiplies the computation time with a factor if all iterations are performed
- For multiple computations / multiple runs of Exudyn (parameter variation, optimization, compute sensitivities), you can use the processing sub module of `Exudyn` to parallelize computations and achieve speedups proporional to the number of cores/threads of your computer; specifically using the `multiThreading` option or even using a cluster (using `dispy`, see `ParameterVariation(...)` function)
- In case of multiprocessing and cluster computing, you may see a very high CPU usage of "Antimalware Service Executable", which is the Microsoft Defender Antivirus; you can turn off such problems by excluding `python.exe` from the defender (on your own risk!) in your settings:
  Settings → Update & Security → Windows Security → Virus & threat protection settings → Manage settings → Exclusions → Add or remove exclusions

Possible speed ups for dynamic simulations:

- for implicit integration, turn on **modified Newton**, which updates jacobians only if needed: `simulationSettings.timeIntegration.newton.useModifiedNewton = True`
- use **multi-threading**: `simulationSettings.parallel.numberOfThreads = ...`, depending on the number of cores (larger values usually do not help); improves greatly for contact problems, but also for some objects computed in parallel; will improve significantly in future
- decrease number of steps (`simulationSettings.timeIntegration.numberOfSteps = int(tEnd/h)`) by increasing the step size h if not needed for accuracy reasons; not that in general, the solver will reduce steps in case of divergence, but not for accuracy reasons, which may still lead to divergence if step sizes are too large
- switch off measuring computation time, if not needed: `simulationSettings.displayComputationTime = False`
- try to switch to **explicit solvers**, if problem has no constraints and if problem is not stiff
- try to have **constant mass matrices** (see according objects, which have constant mass matrices; e.g. rigid bodies using RotationVector Lie group node have constant mass matrix)
- for explicit integration, set `computeEndOfStepAccelerations = False`, if you do not need accurate evaluation of accelerations at end of time step (will then be taken from beginning)
- for explicit integration, set `explicitIntegration.computeMassMatrixInversePerBody=True`, which avoids factorization and back substitution, which may speed up computations with many bodies / particles
- if you are sure that your mass matrix is constant, set:

  `simulationSettings.timeIntegration.reuseConstantMassMatrix = True`; check results!
- check that `simulationSettings.timeIntegration.simulateInRealtime = False`; if set True, it breaks down simulation to real time

- do not record images, if not needed: `simulationSettings.solutionSettings.recordImagesInterval = -1`
- in case of bad convergence, decreasing the step size might also help; check also other flags for adaptive step size and for Newton
- use `simulationSettings.timeIntegration.verboseMode = 1`; larger values create lots of output which drastically slows down
- use `simulationSettings.timeIntegration.verboseModeFile = 0`, otherwise output written to file
- adjust `simulationSettings.solutionSettings.sensorsWritePeriod` to avoid time spent on writing sensor files
- use `simulationSettings.timeIntegration.writeSolutionToFile = False`, otherwise much output may be written to file;
- if solution file is needed, adjust `simulationSettings.solutionSettings.solutionWritePeriod` to larger values and also adjust `simulationSettings.solutionSettings.outputPrecision`, e.g., to 6, in order to avoid larger files; also adjust `simulationSettings.solutionSettings.exportVelocities = False` and `simulationSettings.solutionSettings.exportAccelerations = False` to avoid large output files

2.5 Advanced topics

This section covers some advanced topics, which may be only relevant for a smaller group of people. Functionality may be extended but also removed in future

2.5.1 Camera following objects and interacting with model view

For some models, it may be advantageous to track the translation and/or rotation of certain bodies, e.g., for cars, (wheeled) robots or bicycles. Since `Exudyn` 1.4.18 you can attach view to a marker, using the visualization setting

```
SC.visualizationSettings.interactive.trackMarker = nMarker
```

in which `nMarker` represents the desired marker number to follow. See also related options in `SC.visualizationSettings.interactive` in Section 9.3.0.20.

The following paragraph represents a slower, slightly outdated approach, which may be interesting for advanced usage of object tracking. To do so, the current render state (`SC.renderer.GetState()`, `SC.renderer.SetState(...)`) can be obtained and modified, in order to always follow a certain position. As this needs to be done during redraw of every frame, it is conveniently done in a graphicsUserFunction, e.g., within the ground body. This is shown in the following example, in which `mbs.variables['nTrackNode']` is a node number to be tracked:

```
#mbs.variables['nTrackNode'] contains node number
def UFgraphics(mbs, objectNum):
    n = mbs.variables['nTrackNode']
    p = mbs.GetNodeOutput(n,exu.OutputVariableType.Position,
```

```
                                configuration=exu.ConfigurationType.Visualization)
    rs=SC.renderer.GetState() #get current render state
    A = np.array(rs['modelRotation'])
    p = A.T @ p #transform point into model view coordinates
    rs['centerPoint']=[p[0],p[1],p[2]]
    SC.renderer.SetState(rs)  #modify render state
    return []

#add object with graphics user function
oGround2 = mbs.AddObject(ObjectGround(visualization=
                VObjectGround(graphicsDataUserFunction=UFgraphics)))
#.... further code for simulation here
```

NOTE that this approach is slower and it may lead to a (usually silient) crash after closing the renderer, as the renderer thread is somehow coupled to Python which is prohibited from Python side.

2.5.2 Contact problems

Since Q4 2021 a contact module is available in `Exudyn`. This separate module `GeneralContact` [**still under development, consider with care!**] is highly optimized and implemented with parallelization (multi-threaded) for certain types of contact elements.

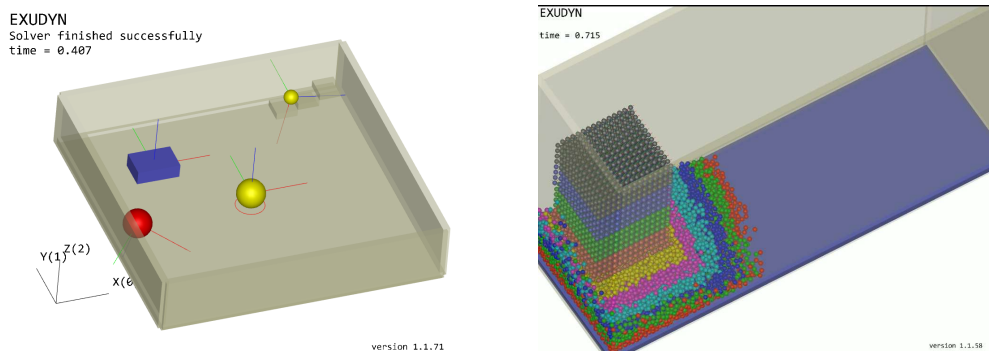


Figure 2.8: Some tests and examples using `GeneralContact`.

Note:

- `GeneralContact` is (in most cases) restricted to dynamic simulation (explicit or implicit [**still under development, consider with care!**]) if friction is used; without friction, it also works in the static case
- in addition to `GeneralContact` there are special objects, in particular for rolling and simple 1D contacts, that are available as single objects, cf. `ObjectConnectorRollingDiscPenalty`
- `GeneralContact` is recommended to be used for large numbers of contacts, while the single objects are integrated more directly into mbs.

Currently, `GeneralContact` includes:

- Sphere-Sphere contact (attached to any marker); may represent circle-circle contact in 2D

- Triangles mounted on rigid bodies, in contact with Spheres [only explicit]
- ANCFCable2D contacting with spheres (which then represent circles in 2D) [partially implicit, needs revision]

For details on the contact formulations, see Section 5.6.

2.5.3 OpenVR

The general open source libraries from Valve, see

https://github.com/ValveSoftware/openvr

have been linked to `Exudyn`. In order to get OpenVR fully integrated, you need to run `setup.py Exudyn` with the `-openvr` flag. For general installation instructions, see Section 1.2.

Running OpenVR either requires an according head mounted display (HMD) or a virtualization using, e.g., Riftcat 2 to use a mobile phone with an according adapter. Visualization settings are available in `interactive.openVR`, but need to be considered with care. An example is provided in `openVRengine.py`, showing some optimal flags like locking the model rotation, zoom or translation.

Everything is experimental, but contributions are welcome!

2.5.4 Interaction with Julia

The scientific community gets increasingly interested into the language Julia. There is a very simple interoperability with julia – at least from julia to Python – which has been tests. The other way – calling Python from julia – is also possible, but it is left to the reader.

After installing julia (tested on Windows 10 with julia 1.6.7), you need to add Python accessibility via `PyCall` in **julia**:

```
using Pkg
Pkg.add("PyCall")
```

Ideally, you have a certain Python installation where `Exudyn` is already installed (and for the following examples, you also need `matplotlib`). Find the according Python path in any **Python** console:

```
import sys
print(sys.executable)
```

Use this path and adapt the following **julia** script ('raw' allows to use single backslash) in **julia**:

```
ENV["PYTHON"]=raw"C:\Users\xyz\.conda\envs\venvP38\python.exe"
Pkg.build("PyCall")
```

Now we can interact with Python, using Python objects in **julia** almost natively, try:

```
py"""
import exudyn
from exudyn.demos import *

Demo1()
"""
```

This will run the very simple Exudyn Demo1. As exudyn is now imported into this Python session, you can access it, e.g., py"exudyn".Help() will write the help message.

To show the interoperability with julia, test the following example (similar to Demo1) in **julia**:

```
py"""
import exudyn as exu                #EXUDYN package including C++ core part
import exudyn.itemInterface as eii #conversion of data to exudyn dictionaries

SC = exu.SystemContainer()          #container of systems
mbs = SC.AddSystem()                #add a new system to work with

nMP = mbs.AddNode(eii.NodePoint2D(referenceCoordinates=[0,0]))
mbs.AddObject(eii.ObjectMassPoint2D(physicsMass=10, nodeNumber=nMP ))
mMP = mbs.AddMarker(eii.MarkerNodePosition(nodeNumber = nMP))
mbs.AddLoad(eii.Force(markerNumber = mMP, loadVector=[0.001,0,0]))

#add a sensor:
s = mbs.AddSensor(eii.SensorNode(nodeNumber=nMP,
                  outputVariableType=exu.OutputVariableType.Position,
                  storeInternal=True))

mbs.Assemble()                      #assemble system and solve
simulationSettings = exu.SimulationSettings()
simulationSettings.timeIntegration.verboseMode=1 #provide some output
simulationSettings.solutionSettings.coordinatesSolutionFileName = 'solution/demo1.txt'

exu.SolveDynamic(mbs, simulationSettings)
print('results can be found in local directory: solution/demo1.txt')
"""
```

We can access Python variables from julia via py"..." to read out, e.g., mbs:

```
py"mbs".systemData.Info()
```

We can use variables (or objects) directly in julia, e.g.,

```
mbs=py"mbs"
print(mbs)
```

Finally, we can also plot values via PlotSensor (matplotlib in the background):

```
eplt=pyimport("exudyn.plot")
eplt.PlotSensor(py"mbs", py"s")
```

We could also access the stored sensor data in julia, using

```
x = py"mbs".GetSensorStoredData(py"s")
```

and we could just print (or use) the first 10 rows of this data generated on the Python side, using it in **julia**:

```
x[1:10,:]
```

NOTE the 1-based indexing in julia, which highlights the limitations of this approach.

To finally check if the GLFW renderer also runs via julia, just use:

```
py"""
from exudyn.demos import *
Demo2()
"""
```

For the full range of possibilities, see github.com/JuliaPy/PyCall.jl.

2.5.5 Interaction with other codes

Interaction with other codes and computers (E.g., MATLAB or other C++ codes, or other Python versions) is possible. To connect to any other code, it is convenient to use a TCP/IP connection. This is enabled via the `exudyn.utilities` functions

- `CreateTCPIPconnection`
- `TCPIPsendReceive`
- `CloseTCPIPconnection`

Basically, data can be transmitted in both directions, e.g., within a preStepUserFunction. In Examples, you can find TCPIPexudynMatlab.py which shows a basic example for such a connectivity.

2.5.6 ROS

Basic interaction with ROS has been tested. However, make sure to use Python 3, as there is no (and will never be any) Python 2 support for `Exudyn`.

2.6 C++ Code

This section covers some information on the C++ code. For more information see the Open source code and use doxygen.

Exudyn was developed for the efficient simulation of flexible multi-body systems. Exudyn was designed for rapid implementation and testing of new formulations and algorithms in multibody systems, whereby these algorithms can be easily implemented in efficient C++ code. The code is applied to industry-related research projects and applications.

2.6.1 Focus of the C++ code

The code focuses on four principles, starting with highest priority:

1. developer-friendly
2. error minimization
3. user-friendliness
4. efficiency

The focus is therefore on:

- A developer-friendly basic structure regarding the C++ class library and the possibility to add new components.
- The basic libraries are slim, but extensively tested; only the necessary components are available
- Complete unit tests are added to new program parts during development; for more complex processes, tests are available in Python
- In order to implement the sometimes difficult formulations and algorithms without errors, error avoidance is always prioritized.
- To generate efficient code, classes for parallelization (vectorization and multithreading) are provided. We live the principle that parallelization takes place on multi-core processors with a central main memory, and thus an increase in efficiency through parallelization is only possible with small systems, as long as the program runs largely in the cache of the processor cores. Vectorization is tailored to SIMD commands as they have Intel processors, but could also be extended to GPGPUs in the future.
- The user interface (Python) provides a nearly 1:1 image of the system and the processes running in it, which can be controlled with the extensive possibilities of Python.

2.6.2 C++ Code structure

The following **entry points** into the C++ code can be found:

- Python – C++: the creation of the module `exudyn` is found in:

  `main/src/Pymodules/PybindModule.cpp`
  it includes large header files, which are automatically created for binding C++ code with Python.
- The object factory for creation of items (calling `mbs.AddNode(...)` and similar):
  `main/src/Main/MainObjectFactory.h / .cpp`
- Using the VisualStudio `.sln` file and using the Debug mode allows you to smoothly walk from Python to C++ code (though that this takes some time to start up and it does not work always; and it does not work for graphics if it runs in a separate thread).

The functionality of the code is mainly based on systems (MainSystem and CSystem), items and solvers representing the multibody system or similar physical systems to be simulated. Parts of the core structure of Exudyn are:

- CSystem / MainSystem: a multibody system which consists of nodes, objects, markers, loads, etc.
- SystemContainer: holds a set of systems; connects to visualization (container)
- items: node, (computational) object, marker, load, sensor
- computational objects: efficient objects for computation = bodies, connectors, connectors, loads, nodes, ...
- visualization objects: interface between computational objects and 3D graphics

- main (manager) objects: do all tasks (e.g. interface to visualization objects, GUI, Python, ...) which are not needed during computation
- static solver, kinematic solver, time integration
- Python interface via pybind11; items are accessed with a dictionary interface; system structures and settings read/written by direct access to the structure (e.g. SimulationSettings, VisualizationSettings)
- interfaces to linear solvers; future: optimizer, eigenvalue solver, ... (mostly external or in Python)
- **autogenerated**: this folder in `main/src` contains many item definitions as well as other interface files; they are all automatically generated by some Python code and should not be changed manually as they will be overwritten.

2.6.3 C++ Code: Modules

The following internal modules are used, which are represented by directories in `main/src`:

- Autogenerated: item (nodes, objects, markers and loads) classes split into main (management, Python connection), visualization and computation
- Graphics: a general data structure for 2D and 3D graphical objects and a tiny openGL visualization; linkage to GLFW
- Linalg: Linear algebra with vectors and matrices; separate classes for small vectors (SlimVector), large vectors (Vector and ResizableVector), vectors without copying data (LinkedDataVector), and vectors with constant size (ConstVector)
- Main: mainly contains SystemContainer, System and ObjectFactory
- Objects: contains the implementation part of the autogenerated items
- Pymodules: manually created libraries for linkage to Python via pybind; remaining linking to Python is located in autogenerated folder
- pythonGenerator: contains Python files for automatic generation of C++ interfaces and Python interfaces of items;
- Solver: contains all solvers for solving a CSystem
- System: contains core item files (e.g., MainNode, CNode, MainObject, CObject, ...)
- Tests: files for testing of internal linalg (vector/matrix), data structure libraries (array, etc.) and functions
- Utilities: array structures for administrative/managing tasks (indexes of objects ... bodies, forces, connectors, ...); basic classes with templates and definitions

The following main external libraries are linked to Exudyn:

- LEST: for testing of internal functions (e.g. linalg)
- GLFW: 3D graphics with openGL; cross-platform capabilities
- Eigen: linear algebra for large matrices, linear solvers, sparse matrices and link to special solvers
- pybind11: linking of C++ to Python

2.6.4 Code style and conventions

This section provides general coding rules and conventions, partly applicable to the C++ and Python parts of the code. Many rules follow common conventions (e.g., google code style, but not always – see notation):

- write simple code (no complicated structures or uncommon coding)
- write readable code (e.g., variables and functions with names that represent the content or functionality; AVOID abbreviations)
- put a header in every file, according to Doxygen format
- put a comment to every (global) function, member function, data member, template parameter
- ALWAYS USE curly brackets for single statements in 'if', 'for', etc.; example: if (i<n) {i += 1;}
- use Doxygen-style comments (use '//!' Qt style and '@ date' with '@' instead of '́for commands)
- use Doxygen (with preceeding '@') 'test' for tests, 'todo' for todos and 'bug' for bugs
- USE 4-spaces-tab
- use C++11 standards when appropriate, but not exhaustively
- ONE class ONE file rule (except for some collectors of single implementation functions)
- add complete unit test to every function (every file has link to LEST library)
- avoid large classes (>30 member functions; > 15 data members)
- split up god classes (>60 member functions)
- mark changed code with your name and date
- REPLACE tabs by spaces: Extras->Options->C/C++->Tabstopps: tab stopp size = 4 (=standard) + KEEP SPACES=YES

2.6.5 Notation conventions

The following notation conventions are applied (**no exceptions!**):

- use lowerCamelCase for names of variables (including class member variables), consts, c-define variables, ...; EXCEPTION: for algorithms following formulas, e.g., $f = M * q_{tt} + K * q$, GBar, ...
- use UpperCamelCase for functions, classes, structs, ...
- Special cases for CamelCase (with some exceptions that happened in the past ...):
  - continue upper case after upper case abbreviations in case of **functions or classes**: 'ODESystem', 'Point2DClass', 'ANCFCable2D', 'ANCFALE', 'ComputeODE1Equations', ... (this is not always nice to read, but has become a standard and will be further used!)
  - for variables and class member variables continue **lower case**: 'nODE1variables', 'dim2Dspecial', 'ANCFsize'
  - abbreviations at beginning of expressions: for functions or classes use `ODEComputeCoords()`, for variables avoid 'ODE' at beginning: use 'nODE' or write 'odeCoordinates'
- '[...]Init' ... in arguments, for initialization of variables; e.g. 'valueInit' for initialization of member variable 'value'

- use American English throughout: Visualization, etc.
- AVOID consecutive capitalized words, e.g., avoid 'ODEAE'
- do not use '_' within variable or function names; exception: derivatives
- use name which exactly describes the function/variable: 'numberOfItems' instead of 'size' or 'l'
- examples for variable names: secondOrderSize, massMatrix, mThetaTheta
- examples for function/class names: `SecondOrderSize`, `EvaluateMassMatrix`, `Position(const Vector3D& localPosition)`
- use the Get/Set...() convention if data is retrieved from a class (Get) or something is set in a class (Set); Use `const T& Get()/T& Get` if direct access to variables is needed; Use Get/Set for pybind11
- example Get/Set: `Real* GetDataPointer()`, `Vector::SetAll(Real)`, `GetTransposed()`, `SetRotationalParar` `SetColor(...)`, ...
- use 'Real' instead of double or float: for compatibility, also for AVX with SP/DP
- use 'Index' for array/vector size and index instead of size_t or int
- item: object, node, marker, load: anything handled within the computational/visualization systems
- Do not use numbers (3 for 3D or any other number which represents, e.g., the number of rotation parameters). Use const Index or constexpr to define constants.

2.6.6 No-abbreviations-rule

The code uses a **minimum set of abbreviations**; however, the following abbreviation rules are used throughout: In general: DO NOT ABBREVIATE function, class or variable names: GetDataPointer() instead of GetPtr(); exception: cnt, i, j, k, x or v in cases where it is really clear (short, 5-line member functions).

Exceptions to the NO-ABBREVIATIONS-RULE, see also List of Abbreviations:

- ordinary differential equation (ODE)
- ODE2: marks parts related to second order differential equations (SOS2, EvalF2 in HOTINT)
- first order ordinary differential equations (ODE1): marks parts related to first order differential equations (ES, EvalF in HOTINT)
- AE; note: using the term 'AEcoordinates' for 'algebraicEquationsCoordinates'
- 'C[...]' ... Computational, e.g. for ComputationalNode ==> use 'CNode'
- mbs
- minimum (min), maximum (max)
- absolute (e.g., absolute error), absolute value (abs), relative (e.g., relative error) (rel)
- triangle (trig)
- quadrangle, polygon with 4 vertices (quad)
- RHS
- LHS
- Euler parameters (EP)

- rotation parameterization: consecutive rotations around x, y and z-axis (Tait-Bryan) (Rxyz)
- coefficients (coeffs)
- position (pos)
- Plücker transformation (T66); based on 6×6 matrix transformations
- write time derivatives with underscore: _t, _tt; example: Position_t, Position_tt, ...
- write space-wise derivatives ith underscore: _x, _xx, _y, ...
- if a scalar, write coordinate derivative with underscore: _q, _v (derivative w.r.t. velocity coordinates)
- for components, elements or entries of vectors, arrays, matrices: use 'item' throughout
- '[...]Init' ... in arguments, for initialization of variables; e.g. 'valueInit' for initialization of member variable 'value'

2.6.7 Implementation of new computational items in C++

This section should sketch which changes will be needed to integrate new C++ items. In general, it is recommended to first start with a Python implementation with user functions based on `NodeGeneric...`, `ObjectGeneric...`, `ObjectConnectorCoordinateVector` for constraints and any suitable connector for new nodes or objects. New sensors can be based on the `SensorUserFunction`.

If such an implementation is successful, but too slow, a C++ implementation can be considered. In the following, two use cases are shown, which show the simplicity of the procedure:

- **Case 1**: user object (body):
  It is recommended to first search for a body with a similar behavior. Copy the definition of such an object inside the file `objectDefinition.py` and edit the according lines. There is not much description of this file yet (except from the first lines of the file), as it will be transformed into another format in the future. Basically, you need to edit the interface, which contains parameters (which are linked to Python) and functions, which go to the header file. When you finished editing, run `pythonAutoGenerateObjects.py`. This generates the header file in `src/autogenerated` but also adds description to some docs files and adds the `pybind11` interface. Now copy the implementation (`.cpp`) file of the same connector from which you copied from and rename and edit all functions. For the body

  - `ComputeMassMatrix`: computes the mass matrix either in sparse or dense mode; this function is performance-critical if the mass matrix is non-constant
  - `ComputeODE2LHS`: computes the LHS generalized forces of the body; this function is performance-critical
  - `GetAccessFunctionTypes`: specifies, which access functions are available in `GetAccessFunctionBody(...`
  - `GetAccessFunctionBody`: needs to compute functions for 'access' to the body, in the sense that e.g. forces or torques can be applied.
  - `GetAvailableJacobians`: shall return the flags which jacobians of `ComputeODE2LHS` need to be computed and which are available as functions; binary flags added up

- `GetOutputVariableBody`: function needs to implement the output variables, such as position, acceleration, forces, etc. as defined in `GetOutputVariableTypes()`
- `HasConstantMassMatrix`: specifies, if mass matrix is constant
- `GetNumberOfNodes`: number of nodes of object
- `GetODE2Size`: total number of ODE2 coordinates
- `GetType`: some flags for objects, such as `Body`, `SingleNoded`, `SuperElement`, ...; these flags are needed for connectivity and special treatment in the system
- `GetPosition, GetVelocity, ...`: provide this functions as far as possible; rigid bodies need to provide positions and rotation matrix, as well as velocity and angular velocity for markers; if functions do not exist, some marker or sensor functions may fail
- ... possibly some helper functions, which you should implement for the functionality of your object.

- **Case 2**: user connector:
  It is recommended to search for a connector with similar behavior; first check, if you would like to implement an algebraic constraint or a spring-damper-like connector. Again, copy a similar connector in `objectDefinition.py` and edit the according lines. When you finished editing, run `pythonAutoGenerateObjects.py` and make a copy of the copied implementation (`.cpp`) file. The implementation file usually consists of

  - `ComputeODE2LHS`: this function shall compute the LHS generalized forces on the two marker objects
  - `ComputeJacobianODE2_ODE2`: computes the `GetAvailableJacobians()` is not providing any '..._function' flag, which indicates that these jacobians are available as function
  - `GetOutputVariableConnector`: this function needs to compute all output variables as given in `GetOutputVariableTypes()`
  - ... possibly some helper functions, which you should implement for the functionality of your object.

Chapter 3

Tutorial

This section will show:

- A basic tutorial for a 1D mass and spring-damper with initial displacements, shortest possible model with practically no special settings
- A more advanced rigid-body model, including 3D rigid bodies and revolute joints
- Links to examples section

A large number of examples, some of them quite advanced, can be found in:

`main/pythonDev/Examples`

`main/pythonDev/TestModels`

3.1 Mass-Spring-Damper tutorial

The Python source code of the first tutorial can be found in the file:

`main/pythonDev/Examples/springDamperTutorial.py`

A similar version based on a simplified approach (using a 3D mass point) is available as, which uses simplified approaches:

`main/pythonDev/Examples/springDamperTutorialNew.py`

The following tutorial will set up a mass point and a spring damper, dynamically compute the solution and evaluate the reference solution.

We import the exudyn library and the interface for all nodes, objects, markers, loads and sensors:

```
import exudyn as exu
from exudyn.utilities import Point, NodePointGround, MassPoint, MarkerNodeCoordinate,\
                             CoordinateSpringDamper, LoadCoordinate, SensorObject
import exudyn.graphics as graphics #only import if it does not conflict
import numpy as np #for postprocessing
```

Instead of the named import of `exudyn.utilities` functions and classes, you can use a star import, which includes `itemInterface`, `rigidBodyUtilities` and some helper functions:

```
from exudyn.utilities import *
```

Next, we need a `SystemContainer`, which contains all computable systems and add a new MainSystem `mbs`. Per default, you always should name your system 'mbs' (multibody system), in order to copy/paste code parts from other examples, tutorials and other projects:

```
SC = exu.SystemContainer()
mbs = SC.AddSystem()
```

In order to check, which version you are using, you can printout the current `Exudyn` version. The version shown is in line with the issue tracker and marks the number of open/closed issues added to `Exudyn` . Adding `True` as argument will also print platform-specific information, which is helpful in case of reporting some compatibility issues:

```
print('EXUDYN version='+exu.config.Version(True))
```

Using the powerful Python language, we can define some variables for our problem, which will also be used for the analytical solution:

```
L=0.5                #reference position of mass
mass = 1.6           #mass in kg
spring = 4000        #stiffness of spring-damper in N/m
damper = 8           #damping constant in N/(m/s)
f =80                #force on mass
```

For the simple spring-mass-damper system, we need initial displacements and velocities:

```
u0=-0.08             #initial displacement
v0=1                 #initial velocity
x0=f/spring          #static displacement
print('resonance frequency = '+str(np.sqrt(spring/mass)))
print('static displacement = '+str(x0))
```

We first need to add nodes, which provide the coordinates (and the degrees of freedom) to the system. The following line adds a 3D node for 3D mass point[1]:

```
n1=mbs.AddNode(Point(referenceCoordinates = [L,0,0],
                     initialCoordinates = [u0,0,0],
                     initialVelocities = [v0,0,0]))
```

Here, `Point` (=`NodePoint`) is a Python class, which takes a number of arguments defined in the reference manual. The arguments here are `referenceCoordinates`, which are the coordinates for which the system is defined. The initial configuration is given by `referenceCoordinates` + `initialCoordinates`, while the initial state additionally gets `initialVelocities`. The command `mbs.AddNode(...)` returns a `NodeIndex n1`, which basically contains an integer, which can only be used as node number. This node number will be used lateron to use the node in the object or in the marker.

While `Point` adds 3 unknown coordinates to the system, which need to be solved, we also can add ground nodes, which can be used similar to nodes, but they do not have unknown coordinates

[1]Note: Point is an abbreviation for NodePoint, defined in `itemInterface.py`.

– and therefore also have no initial displacements or velocities. The advantage of ground nodes (and ground bodies) is that no constraints are needed to fix these nodes. Such a ground node is added via:

```
nGround=mbs.AddNode(NodePointGround(referenceCoordinates = [0,0,0]))
```

In the next step, we add an object[2], which provides equations for coordinates. The `MassPoint` needs at least a mass (kg) and a node number to which the mass point is attached. Additionally, graphical objects could be attached:

```
massPoint = mbs.AddObject(MassPoint(physicsMass = mass, nodeNumber = n1))
```

Note that instead of adding a `NodePoint` and a `MassPoint` with `mbs.AddNode(...)` and `mbs.AddObject(...)`, there is also a convenient function `mbs.CreateMassPoint(...)`, which can do everything at once including the option to add gravity.

In order to apply constraints and loads, we need markers. These markers are used as local positions (and frames), where we can attach a constraint lateron. In this example, we work on the coordinate level, both for forces as well as for constraints. Markers are attached to the according ground and regular node number, additionally using a coordinate number (0 ... first coordinate):

```
groundMarker=mbs.AddMarker(MarkerNodeCoordinate(nodeNumber= nGround,
                                                coordinate = 0))
#marker for springDamper for first (x-)coordinate:
nodeMarker = mbs.AddMarker(MarkerNodeCoordinate(nodeNumber= n1,
                                                coordinate = 0))
```

This means that constraints are be applied to the first coordinate of node `n1` via marker with number `nodeMarker`, which is in fact of type `MarkerNodeCoordinate`.

Now we add a spring-damper to the markers with numbers `groundMarker` and the `nodeMarker`, providing stiffness and damping parameters:

```
nC = mbs.AddObject(CoordinateSpringDamper(markerNumbers = [groundMarker, nodeMarker],
                                          stiffness = spring,
                                          damping = damper))
```

A load is added to marker `nodeMarker`, with a scalar load with value `f`:

```
nLoad = mbs.AddLoad(LoadCoordinate(markerNumber = nodeMarker,
                                   load = f))
```

Again, instead of adding a `MarkerNodeCoordinate` and a `LoadCoordinate` with `mbs.AddLoad(...)`, we could just use `mbs.CreateForce(...)` to add a 3D force vector. For specific joints, there are also `mbs.Create...(...)` functions.

Finally, a sensor is added to the coordinate constraint object with number `nC`, requesting the `outputVariableType Force`:

```
mbs.AddSensor(SensorObject(objectNumber=nC, fileName='groundForce.txt',
                           outputVariableType=exu.OutputVariableType.Force))
```

[2]For the moment, we just need to know that objects either depend on one or more nodes, which are usually bodies and finite elements, or they can be connectors, which connect (the coordinates of) objects via markers, see Section 2.1.

Note that sensors can be attached, e.g., to nodes, bodies, objects (constraints) or loads. As our system is fully set, we can print the overall information and assemble the system to make it ready for simulation:

```
print(mbs)      #show system properties
mbs.Assemble() #prepare for simulation
```

We will use time integration and therefore define a number of steps (fixed step size; must be provided) and the total time span for the simulation:

```
tEnd = 1      #end time of simulation
h = 0.001     #step size; leads to 1000 steps
```

All settings for simulation, see according reference section, can be provided in a structure given from `exu.SimulationSettings()`. Note that this structure will contain all default values, and only non-default values need to be provided:

```
simulationSettings = exu.SimulationSettings()
simulationSettings.solutionSettings.solutionWritePeriod = 5e-3 #output interval general
simulationSettings.solutionSettings.sensorsWritePeriod = 5e-3  #output interval of sensors
simulationSettings.timeIntegration.numberOfSteps = tEnd/h
simulationSettings.timeIntegration.endTime = tEnd
simulationSettings.displayComputationTime = True               #show how fast
```

In order to see some solver output, we must set `verboseMode` to 1 (higher values gives detailed output per step). Furthermore, we can show information on computation time (which may cost some overhead in computation!):

```
simulationSettings.timeIntegration.verboseMode = 1             #show some solver output
simulationSettings.displayComputationTime = True               #show how fast
```

We are using a generalized alpha solver, where numerical damping is needed for index 3 constraints. As we have only spring-dampers, we can set the spectral radius to 1, meaning no numerical damping:

```
simulationSettings.timeIntegration.generalizedAlpha.spectralRadius = 1
```

In order to visualize the results online, a renderer can be started. As our computation will be very fast, it is a good idea to wait for the user to press SPACE, before starting the simulation (uncomment second line):

```
SC.renderer.Start()             #start graphics visualization
#SC.renderer.DoIdleTasks()      #wait for SPACE bar or 'Q' to continue (in render window
  !)
```

As the simulation is still very fast, we will not see the motion of our node. Using a very small step size of, e.g., `h=1e-7` in the lines above allows us to visualize the resulting oscillations in realtime.

Finally, we start the solver, by telling which system to be solved, solver type and the simulation settings:

```
exu.SolveDynamic(mbs, simulationSettings)
```

After simulation, our renderer needs to be stopped (otherwise it will stop unsafely as soon as the Python kernel is stopped or restarted). Sometimes you would like to wait until closing the render window, using `WaitForRenderEngineStopFlag()`:

```
#SC.renderer.DoIdleTasks()       #wait for pressing 'Q' to quit
SC.renderer.Stop()               #safely close rendering window!
```

If you run this code, e.g. in Spyder or Visual Studio Code, it may take a 1-2 seconds to complete. However, the time spent is only related to some overhead in the Python environment and for the visualization. The simulation itself will only take around 3-10 milliseconds, in which a large overhead is due to file writing.

There are several ways to evaluate results, see the reference pages. In the following we take the final value of node n1 and read its 3D position vector:

```
#evaluate final (=current) output values
u = mbs.GetNodeOutput(n1, exu.OutputVariableType.Position)
print('displacement=',u)
```

The following code generates a reference (exact) solution for our example:

```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

omega0 = np.sqrt(spring/mass)          #eigen frequency of undamped system
dRel = damper/(2*np.sqrt(spring*mass)) #dimensionless damping
omega = omega0*np.sqrt(1-dRel**2)      #eigen freq of damped system
C1 = u0-x0 #static solution needs to be considered!
C2 = (v0+omega0*dRel*C1) / omega       #C1, C2 are coeffs for solution
steps = int(tEnd/h)                    #use same steps for reference solution

refSol = np.zeros((steps+1,2))
for i in range(0,steps+1):
  t = tEnd*i/steps
  refSol[i,0] = t
  refSol[i,1] = np.exp(-omega0*dRel*t)*(C1*np.cos(omega*t)+C2*np.sin(omega*t))+x0

plt.plot(refSol[:,0], refSol[:,1], 'r-', label='displacement (m); exact solution')
```

Now we can load our results from the default solution file `coordinatesSolution.txt`, which is in the same directory as your Python tutorial file. **Note** that the visualization of results can be simplified considerably using the `PlotSensor(...)` utility function as shown in the **Rigid body and joints tutorial**!

For reading the file containing commented lines (this does not work in binary mode!), we use a numpy feature and finally plot the displacement of coordinate 0 or our mass point[3]:

```
data = np.loadtxt('coordinatesSolution.txt', comments='#', delimiter=',')
plt.plot(data[:,0], data[:,1], 'b-', label='displacement (m); numerical solution')
```

Note that the coordinates do not include the reference position (which is 0.5 in this case). For information on displacement and reference coordinates, see Section 2.2.6.

[3] `data[:,0]` contains the simulation time, `data[:,1]` contains displacement of (global) coordinate 0, `data[:,2]` contains displacement of (global) coordinate 1, ...)

The sensor result can be loaded in the same way. The sensor output format contains time in the first column and sensor values in the remaining columns. The number of columns depends on the sensor and the output quantity (scalar, vector, ...):

```
data = np.loadtxt('groundForce.txt', comments='#', delimiter=',')
plt.plot(data[:,0], data[:,1]*1e-3, 'g-', label='force (kN)')
```

In order to get a nice plot within Spyder, the following options can be used[4]:

```
ax=plt.gca() # get current axes
ax.grid(True, 'major', 'both')
ax.xaxis.set_major_locator(ticker.MaxNLocator(10))
ax.yaxis.set_major_locator(ticker.MaxNLocator(10))
plt.legend() #show labels as legend
plt.tight_layout()
plt.show()
```

The matplotlib output should look as shown in Fig. 3.1.

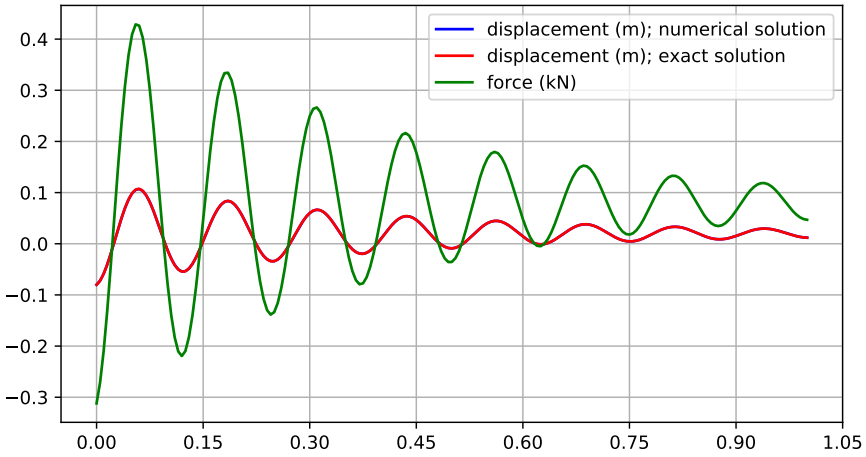


Figure 3.1: Output of spring-damper tutorial.

[4]note, in some environments you need finally the command `plt.show()`

3.2 Rigid body and joints tutorial

The Python source code of the first tutorial, based on the simple description of revolute joints, can be found in the file:

`main/pythonDev/Examples/rigidBodyTutorial3.py`

For alternative approaches, see

`main/pythonDev/Examples/rigidBodyTutorial3withMarkers.py`

`main/pythonDev/Examples/rigidBodyTutorial2.py`

NOTE that the youtube video uses a slightly older way of creating graphics using `GraphicsData...` functions, which can be easily replaced by the newer `graphics. ...` commands shown here. Their interface is identical.

This tutorial will set up a multibody system containing a ground, two rigid bodies and two revolute joints driven by gravity, compare a 3D view of the example in Fig. 3.2.

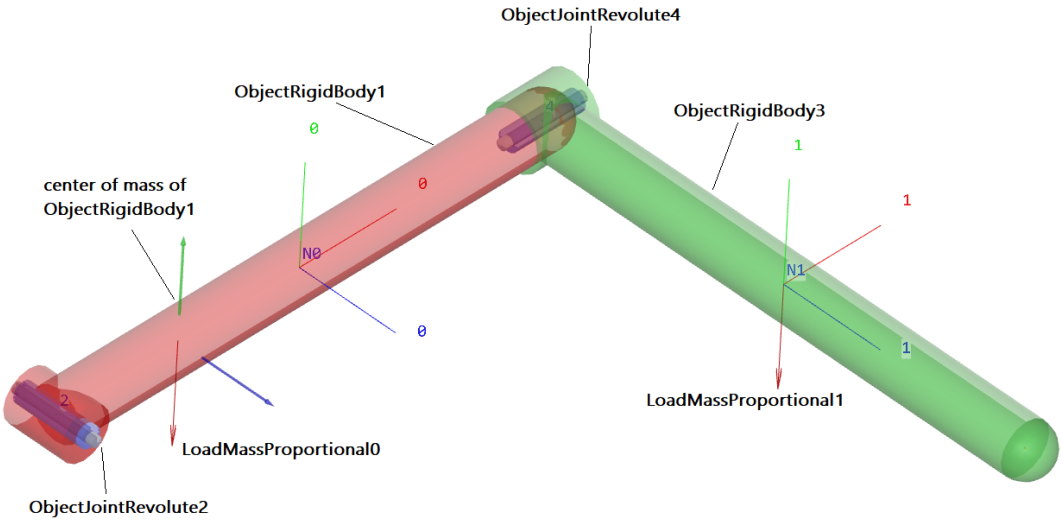


Figure 3.2: Render view of rigid body tutorial, showing objects, nodes (N0, N1), and loads.

We first import the exudyn library and the interface for all nodes, objects, markers, loads and sensors:

```
import exudyn as exu
#import specific items, inertia class for general cuboid (hexahedral) block, etc.
from exudyn.utilities import InertiaCuboid, ObjectRigidBody, MarkerBodyRigid, \
                             GenericJoint, VObjectJointGeneric, SensorBody
import exudyn.graphics as graphics #only import as graphics if it does not conflict
import numpy as np #for postprocessing
```

The submodule `exudyn.utilities` contains helper functions for graphics representation, 3D rigid bodies and joints. For simplicity, many examples just use a star import instead, which is recommended for rapid model development:

```
from exudyn.utilities import *
```

As in the first tutorial, we need a `SystemContainer` and add a new MainSystem `mbs`:

```
SC = exu.SystemContainer()
mbs = SC.AddSystem()
```

We define some geometrical parameters for lateron use.

```
#physical parameters
g =     [0,-9.81,0] #gravity
L = 1               #length
w = 0.1             #width
bodyDim=[L,w,w]     #body dimensions
p0 =    [0,0,0]     #origin of pendulum
pMid0 = np.array([L*0.5,0,0]) #center of mass, body0
```

We add an empty ground body, using default values. It's origin is at [0,0,0] and here we use no visualization.

```
#ground body, located at specific position (there could be several ground objects)
oGround = mbs.CreateGround(referencePosition=[0,0,0])
```

On the background, the `CreateGround` function creates an object, which is equivalent to:

```
oGround = mbs.AddObject(ObjectGround(referencePosition=[0,0,0]))
```

For physical parameters of the rigid body, we can use the class `RigidBodyInertia`, which allows to define mass, center of mass (COM) and inertia parameters, as well as shifting COM or adding inertias. The `RigidBodyInertia` can be used directly to create rigid bodies. Special derived classes can be use to define rigid body inertias for cylinders, cubes, etc., so we use a cube here:

```
#first link:
#inertia for cubic body with dimensions in sideLengths
iCube0 = InertiaCuboid(density=5000, sideLengths=bodyDim)
iCube0 = iCube0.Translated([-0.25*L,0,0]) #transform COM, COM not at reference point!
```

Note that the COM is translated in axial direction, while it would be at the body's local position [0,0,0] by default!
For visualization, we add some graphics for the body defined as a 3D cube with center point and dimensions; additionally we draw a basis (three RGB-vectors) at the COM:

```
#graphics for body
graphicsBody0 = graphics.Brick(centerPoint=[0,0,0],size=[L,w,w],
                               color=graphics.color.red)
graphicsCOM0 = graphics.Basis(origin=iCube0.com, length=2*w)
```

Now we have defined all data for the link (rigid body). We could use `mbs.AddNode(NodeRigidBodyEP(...))` and `mbs.AddObject(ObjectRigidBody(...))` to create a node and a body, but the `MainSystem` since V1.6.110 offers a much more comfortable function:

```
#create node, add body and gravity load:
b0=mbs.CreateRigidBody(inertia = iCube0, #includes COM
                       referencePosition = pMid0,
                       gravity = g,
                       graphicsDataList = [graphicsCOM0, graphicsBody0])
```

which also adds a gravity load and could also set initial velocities, if wanted. Note that much more options are available for this function, e.g., we could define a `nodeType` for the underlying formulation of the rigid body node, see Section 6.10.4. We can use

- `RotationEulerParameters`: for fast computation, but leads to an additional algebraic equation and thus needs an implicit solver
- `RotationRxyz`: contains a singularity if the second angle reaches +/- 90 degrees, but no algebraic equations
- `RotationRotationVector`: for usage with Lie group integrators, especially with explicit integration, singularities are bypassed; leads to fewest unknowns and usually less Newton iterations

We now add a revolute joint around the (global) z-axis. We have several possibilities, which are shown in the following. For the **first two possibilities only**, following `rigidBodyTutorial3withMarkers.py`, we need the following markers

```
#markers for ground and rigid body (not needed for option 3):
markerGround = mbs.AddMarker(MarkerBodyRigid(bodyNumber=oGround, localPosition=[0,0,0]))
markerBody0J0 = mbs.AddMarker(MarkerBodyRigid(bodyNumber=b0, localPosition=[-0.5*L,0,0]))
```

The very general **option 1** is to use the `GenericJoint`, that can be used to define any kind of joint with translations and rotations fixed or free,

```
#revolute joint option 1:
mbs.AddObject(GenericJoint(markerNumbers=[markerGround, markerBody0J0],
                           constrainedAxes=[1,1,1,1,1,0],
                           visualization=VObjectJointGeneric(axesRadius=0.2*w,
                                                             axesLength=1.4*w)))
```

In addition, transformation matrices (`rotationMarker0/1`) can be added, see the joint description.
Option 2 is using the revolute joint, which allows a free rotation around the local z-axis of marker 0 (`markerGround` in our example)

```
#revolute joint option 2:
mbs.AddObject(ObjectJointRevoluteZ(markerNumbers = [markerGround, markerBody0J0],
                                   rotationMarker0=np.eye(3),
                                   rotationMarker1=np.eye(3),
                                   visualization=VObjectJointRevoluteZ(axisRadius=0.2*w,
                                                                       axisLength=1.4*w)
                                   ))
```

Additional transformation matrices (`rotationMarker0/1`) can be added in order to chose any rotation axis.
Note that an error in the definition of markers for the joints can be also detected in the render window (if you completed the example), e.g., if you change the following marker in the lines above,

```
#example if wrong marker position is chosen:
markerBody0J0 = mbs.AddMarker(MarkerBodyRigid(bodyNumber=b0, localPosition=[-0.4*L,0,0]))
```

→ you will see a misalignment of the two parts of the joint by `0.1*L`. The latter approach is very general and will also work for any kind of flexible bodies.
Due to the fact that the definition of markers for general joints is tedious, **option 3** is based on a MainSystem function, which allows to attach revolute joints immediately to **rigid bodies** and defining the rotation axis only once for the joint:

```
#revolute joint option 3 (simplest):
mbs.CreateRevoluteJoint(bodyNumbers=[oGround, b0], position=[0,0,0],
                        axis=[0,0,1], axisRadius=0.2*w, axisLength=1.4*w)
```

Note that `axis` and `position` are defined in global coordinates, and local coordinates are computed according to the reference configuration of the bodies. There exist more arguments that may be specified, e.g., the axis and position can also be defined in the local frame of the first body.

The second link and the according joint can be set up in a very similar way. For visualization, we need to add some graphics for the body defined as a RigidLink graphics function:

```
#second link, simple graphics:
graphicsBody1 = graphics.RigidLink(p0=[0,0,-0.5*L],p1=[0,0,0.5*L],
                                   axis0=[1,0,0], axis1=[0,0,0], radius=[0.06,0.05],
                                   thickness = 0.1, width = [0.12,0.12],
                                   color=graphics.color.lightgreen)

b1=mbs.CreateRigidBody(inertia = InertiaCuboid(density=5000, sideLengths=[0.1,0.1,1]),
                       referencePosition = np.array([L,0,0.5*L]),
                       gravity = g,
                       graphicsDataList = [graphicsBody1])
```

The revolute joint in this case has a free rotation around the global x-axis:

```
#revolute joint (free x-axis)
mbs.CreateRevoluteJoint(bodyNumbers=[b0, b1], position=[L,0,0],
                        axis=[1,0,0], axisRadius=0.2*w, axisLength=1.4*w)
```

Optionally, we could also add forces or torques onto bodies

```
#forces can be added like in the following
force = [0,0.5,0]       #0.5N   in y-direction
torque = [0.1,0,0]      #0.1Nm around x-axis
mbs.CreateForce(bodyNumber=b1,
                loadVector=force,
                localPosition=[0,0,0.5], #at tip
                bodyFixed=False) #if True, direction would corotate with body
mbs.CreateTorque(bodyNumber=b1,
                loadVector=torque,
                localPosition=[0,0,0],   #at body's reference point/center
                bodyFixed=False) #if True, direction would corotate with body
```

Finally, we also add a sensor for some output of the double pendulum:

```
#position sensor at tip of body1
sens1=mbs.AddSensor(SensorBody(bodyNumber = b1, localPosition = [0,0,0.5*L],
                               fileName = 'solution/sensorPos.txt',
                               outputVariableType = exu.OutputVariableType.Position))
```

Before simulation, we need to call `Assemble()` for our system, which links objects, nodes, ..., assigns initial values and does further pre-computations and checks:

```
mbs.Assemble()
```

After `Assemble()`, markers, nodes, objects, etc. are linked and we can analyze the internal structure. First, we can print out useful information, either just typing `mbs` in the iPython console to print out overal information:

```
<systemData:
  Number of nodes= 2
  Number of objects = 5
  Number of markers = 8
  Number of loads = 4
  Number of sensors = 1
  Number of ODE2 coordinates = 14
  Number of ODE1 coordinates = 0
  Number of AE coordinates   = 12
  Number of data coordinates   = 0
  For details see mbs.systemData, mbs.sys and mbs.variables
>
```

Note that there are 2 nodes for the two rigid bodies. The five objects are due to ground object, 2 rigid bodies and 2 revolute joints. The meaning of markers can be seen in the graphical representation described below.
Furthermore, we can print the full internal information as a dictionary using:

```
mbs.systemData.Info() #show detailed information
```

which results in the following output (shortened):

```
node0:
    {'nodeType': 'RigidBodyEP', 'referenceCoordinates': [0.5, 0.0, 0.0, 1.0, 0.0, 0.0,
  0.0], 'addConstraintEquation': True, 'initialCoordinates': [0.0, 0.0, 0.0, 0.0, 0.0,
  0.0, 0.0], 'initialVelocities': [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], 'name': 'node0', '
  Vshow': True, 'VdrawSize': -1.0, 'Vcolor': [-1.0, -1.0, -1.0, -1.0]}
node1:
    {'nodeType': 'RigidBodyEP', 'referenceCoordinates': [1.0, 0.0, 0.5, 1.0, 0.0, 0.0,
  0.0], 'addConstraintEquation': True, 'initialCoordinates': [0.0, 0.0, 0.0, 0.0, 0.0,
  0.0, 0.0], 'initialVelocities': [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], 'name': 'node1', '
  Vshow': True, 'VdrawSize': -1.0, 'Vcolor': [-1.0, -1.0, -1.0, -1.0]}
object0:
```

```
    {'objectType': 'Ground', 'referencePosition': [0.0, 0.0, 0.0], 'name': 'object0', '
  Vshow': True, 'VgraphicsDataUserFunction': 0, 'Vcolor': [-1.0, -1.0, -1.0, -1.0], '
  VgraphicsData': {'TODO': 'Get graphics data to be implemented'}}
object1:
    {'objectType': 'RigidBody', 'physicsMass': 50.0, 'physicsInertia':
  [0.08333333333333336, 7.333333333333334, 7.333333333333334, 0.0, 0.0, 0.0], '
  physicsCenterOfMass': [-0.25, 0.0, 0.0], 'nodeNumber': 0, 'name': 'object1', 'Vshow':
  True, 'VgraphicsDataUserFunction': 0, 'VgraphicsData': {'TODO': 'Get graphics data to be
   implemented'}}
object2:
    {'objectType': 'JointRevolute', 'markerNumbers': [3, 4], 'rotationMarker0': [[0.0,
  1.0, 0.0], [-1.0, 0.0, 0.0], [0.0, 0.0, 1.0]], 'rotationMarker1': [[0.0, 1.0, 0.0],
  [-1.0, 0.0, 0.0], [0.0, 0.0, 1.0]], 'activeConnector': True, 'name': 'object2', 'Vshow':
   True, 'VaxisRadius': 0.019999999552965164, 'VaxisLength': 0.14000000059604645, 'Vcolor'
  : [-1.0, -1.0, -1.0, -1.0]}
object3:
...
```

Sometimes it is hard to understand the degree of freedom for the constrained system. Furthermore, we may have added – by error – redundant constraints, which are not solvable or at least cause solver problems. Both can be checked with the command:

```
mbs.ComputeSystemDegreeOfFreedom(verbose=True) #print out DOF and further information
```

This will print:

```
ODE2 coordinates          = 14
total constraints         = 12
redundant constraints     = 0
pure algebraic constraints= 0
degree of freedom         = 2
```

We see that there are 14 ODE2 coordinates from the two nodes that are based on Euler parameters. The two joints add 2×5 constraints and there are 2 additional Euler parameter constraints, giving a degree of freedom of 2 (as expected ...).
You can try and duplicate the code for the second revolute joint:

```
#add a second constraint for bodies b0 and b1:
mbs.CreateRevoluteJoint(bodyNumbers=[b0, b1], ...)
```

such that we have two identical joints (which would be unwanted, in general). This would give

```
ODE2 coordinates          = 14
total constraints         = 17
redundant constraints     = 5
pure algebraic constraints= 0
degree of freedom         = 2
```

which clearly shows the 5 redundant constraints, which will lead to a solver failure (except for the `EigenDense` solver, see there). In practical cases, redundant constraints may be much more involved, but can be detected in this way.

A graphical representation of the internal structure of the model can be shown using the command `DrawSystemGraph`:

```
mbs.DrawSystemGraph(useItemTypes=True) #draw nice graph of system
```

For the output see Fig. 3.3. Note that obviously, markers are always needed to connect objects (or nodes) as well as loads. We can also see, that 2 markers MarkerBodyRigid1 and MarkerBodyRigid2 are unused, which is no further problem for the model and also does not require additional computational resources (except for some bytes of memory). Having isolated nodes or joints that are not connected (or having too many connections) may indicate that you did something wrong in setting up your model. Furthermore, it can be seen that the function `CreateRigidBody` added a body `ObjectRigidBody`, a node `NodeRigidBodyEP`, a `LoadMassProportional` for gravity load with a `MarkerBodyMass`, and the function `CreateRevoluteJoint` created two `MarkerBodyRigid` and a `ObjectJointRevoluteZ` which represents a revolute joint about a Z-axis in the joint coordinate system. For further information, consult the respective pages in the Items reference manual.

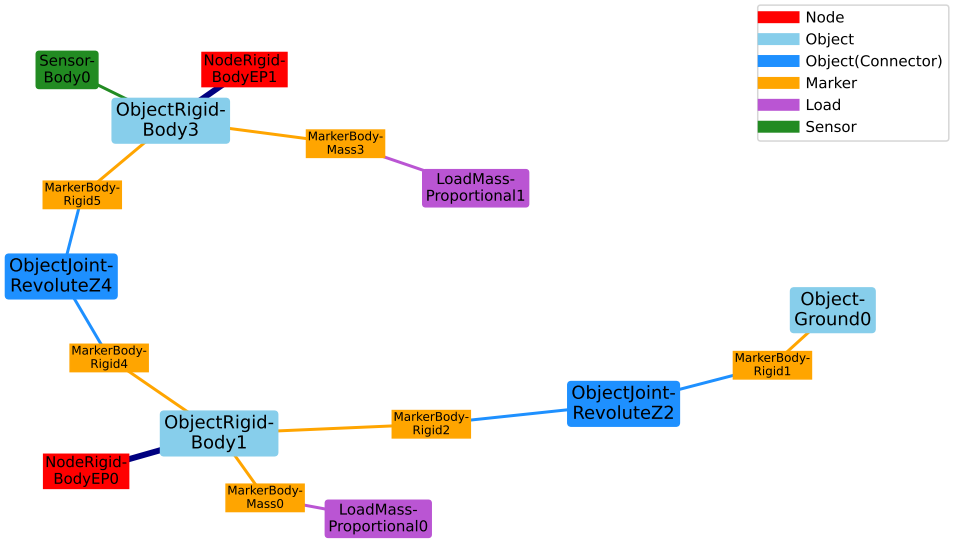


Figure 3.3: System graph for rigid body tutorial (with option 3 for the first revolute joint). Numbers are always related to the node number, object number, etc.; note that colors are used to distinguish nodes, objects, markers, loads and sensors

---

Before starting our simulation, we should adjust the solver parameters, especially the end time and the step size (no automatic step size for implicit solvers available!):

```
simulationSettings = exu.SimulationSettings() #takes currently set values or default
  values

tEnd = 4 #simulation time
h = 1e-3 #step size
```

```
simulationSettings.timeIntegration.numberOfSteps = int(tEnd/h)
simulationSettings.timeIntegration.endTime = tEnd
simulationSettings.timeIntegration.verboseMode = 1
#simulationSettings.timeIntegration.simulateInRealtime = True
simulationSettings.solutionSettings.solutionWritePeriod = 0.005 #store every 5 ms
```

The `verboseMode` tells the solver the amount of output during solving. Higher values (2, 3, ...) show residual vectors, jacobians, etc. for every time step, but slow down simulation significantly. The option `simulateInRealtime` is used to view the model during simulation, while setting this false, the simulation finishes after fractions of a second. It should be set to false in general, while solution can be viewed using the `SolutionViewer()`. With `solutionWritePeriod` you can adjust the frequency which is used to store the solution of the whole model, which may lead to very large files and may slow down simulation, but is used in the `SolutionViewer()` to reload the solution after simulation.
In order to improve visualization, there are hundreds of options, see Visualization settings in Section 9.3, some of them used here:

```
SC.visualizationSettings.window.renderWindowSize = [1600,1200]
SC.visualizationSettings.openGL.multiSampling = 4  #improved OpenGL rendering
SC.visualizationSettings.general.autoFitScene = False

SC.visualizationSettings.nodes.drawNodesAsPoint = False
SC.visualizationSettings.nodes.showBasis = True #shows three RGB (=xyz) lines for node
  basis
```

The option `autoFitScene` is used in order to avoid zooming while loading the last saved render state, see below.
We can start the 3D visualization (Renderer) now:

```
SC.renderer.Start()
```

In order to reload the model view of the last simulation (if there is any), we can use the following commands:

```
if 'renderState' in exu.sys: #reload old view
    SC.renderer.SetState(exu.sys['renderState'])

SC.renderer.DoIdleTasks()    #stop before simulating
```

the function `WaitForUserToContinue()` waits with simulation until we press SPACE bar. This allows us to make some pre-checks.
Finally, the **index 2** (velocity level) implicit time integration (simulation) is started with:

```
mbs.SolveDynamic(simulationSettings = simulationSettings,
                 solverType = exu.DynamicSolverType.TrapezoidalIndex2)
```

This solver is used in the present example, but should be considered with care as it leads to (small) drift of position constraints, linearly increasing in time. Using sufficiently small time steps, this effect is often negligible on the advantage of having a **energy-conserving integrator** (guaranteed for linear systems, but very often also for the nonlinear multibody system). Due to the velocity level, the

integrator is less sensitive to consistent initial conditions on position level and compatible to frequent step size changes, however, initial jumps in velocities may never damp out in undamped systems. Alternatively, an **index 3** implicit time integration – the generalized-α method – is started with the default settings for `solverType`:

```
mbs.SolveDynamic(simulationSettings = simulationSettings)
```

Note that the **generalized-α method** includes numerical damping (adjusted with the spectral radius) for stabilization of index 3 constraints. This leads to effects every time the integrator is (re-)started, e.g., when adapting time step sizes. For fixed step sizes, this is **the recommended integrator**.
After simulation, the library would immediately exit (and jump back to iPython or close the terminal window). In order to avoid this, we can use `WaitForRenderEngineStopFlag()` to wait until we press key 'Q'.

```
SC.renderer.DoIdleTasks()   #stop before closing
SC.renderer.Stop()          #safely close rendering window!
```

If you entered everything correctly, the render window should show a nice animation of the 3D double pendulum after pressing the SPACE key. If we do not stop the renderer (`SC.renderer.Stop()`), it will stay open for further simulations. However, it is safer to always close the renderer at the end.
As the simulation will run very fast, if you did not set `simulateInRealtime` to true. However, you can reload the stored solution and view the stored steps interactively:

```
mbs.SolutionViewer()
#alternatively, we could load solution from a file:
#from exudyn.utilities import LoadSolutionFile
#sol = LoadSolutionFile('coordinatesSolution.txt')
#mbs.SolutionViewer(sol)
```

Finally, we can plot our sensor, drawing the y-component of the sensor (check out the many options in `PlotSensor(...)` to conveniently represent results!):

```
mbs.PlotSensor(sensorNumbers=[sens1],components=[1],closeAll=True)
```

Congratulations! You completed the rigid body tutorial, which gives you the ability to model multibody systems. Note that much more complicated models are possible, including feedback control or flexible bodies, see the Examples!

3.3 Flexible beams tutorial

This tutorial briefly introduces two simple planar beams and how to work with them with utility functions. The python source code of the beam tutorial can be found at:

```
main/pythonDev/Examples/beamTutorial.py
```

The tutorial uses the GeometricallyExactBeam2D, which is a shear deformable Reissner-Timoshenko beam, and a thin cable ANCFCable2D, which represents a large deformation Bernoulli-Euler beam. The model includes two highly flexible planar with length 2m, height 0.005m, width 0.01m, Young's modulus 1e9N/m^2 and density 2000kg/m^3. The shear deformable beam is rigidly attached to ground and the cable is rigidly attached to a moving ground.

We first import necessary libraries and set up a mbs. Note that utilities also include pi, sin, cos and sqrt.

```
import exudyn as exu
from exudyn.utilities import *
import numpy as np

SC = exu.SystemContainer()
mbs = SC.AddSystem()
```

We define a set of beam parameters, discretization and geometry for both beam models.

```
numberOfElements = 16
L = 2                       # length of pendulum
E=2e11                      # steel
rho=7800                    # elastomer
h=0.005                     # height of rectangular beam element in m
b=0.01                      # width of rectangular beam element in m
A=b*h                       # cross sectional area of beam element in m^2
I=b*h**3/12                 # second moment of area of beam element in m^4
nu = 0.3                    # Poisson's ratio

EI = E*I
EA = E*A
rhoA = rho*A
rhoI = rho*I
ks = 10*(1+nu)/(12+11*nu) # shear correction factor
G = E/(2*(1+nu))          # shear modulus
GA = ks*G*A               # shear stiffness of beam

g = [0,-9.81,0]           # gravity vector

positionOfNode0 = [0,0,0] # 3D vector
positionOfNode1 = [L,0,0] # 3D vector
```

Build geometrically exact 2D beam template (Timoshenko-Reissner), which includes all parameters In order to create beams, we usually need to create 2D rigid body nodes, create beam elements, add constraints and loads.
However, there is a utility function `GenerateStraightBeam(...)`, which conveniently does this for straight beams, including discretization, constraints and gravity. First, we create a beam template, which includes all beam parameters (this could also be another beam type):

```
beamTemplate = Beam2D(nodeNumbers = [-1,-1], #added later
                      physicsMassPerLength=rhoA,
                      physicsCrossSectionInertia=rhoI,
                      physicsBendingStiffness=EI,
                      physicsAxialStiffness=EA,
                      physicsShearStiffness=GA,
                      physicsBendingDamping=0.02*EI,
```

```
                    visualization=VObjectBeamGeometricallyExact2D(drawHeight = h))
```

Now we use this template and call `GenerateStraightBeam`, which takes the nodal positions, calculates according beam element lengths from discretization and could add constraints, if `fixedConstraintsNode0` or `fixedConstraintsNode1` are not `None`.

```
beamData = GenerateStraightBeam(mbs, positionOfNode0, positionOfNode1,
                                numberOfElements, beamTemplate, gravity= g,
                                fixedConstraintsNode0=[1,1,1],
                                fixedConstraintsNode1=None)
```

We perform the same operations for ANCF cable elemente (Bernoulli-Euler), but in this case, we do not add constraints:

```
beamTemplate = Cable2D(nodeNumbers = [-1,-1], #added later
                       physicsMassPerLength=rhoA,
                       physicsBendingStiffness=EI,
                       physicsAxialStiffness=EA,
                       physicsBendingDamping=0.02*EI,
                       visualization=VCable2D(drawHeight = h))

cableData = GenerateStraightBeam(mbs, positionOfNode0, positionOfNode1,
                                 numberOfElements, beamTemplate, gravity= g,
                                 fixedConstraintsNode0=None,
                                 fixedConstraintsNode1=None)
```

Now, we create a ground object and markers to attach cable with generic joint

```
oGround = mbs.CreateGround(referencePosition=[0,0,0])
mGround = mbs.AddMarker(MarkerBodyRigid(bodyNumber=oGround, localPosition=[0,0,0]))
mCable = mbs.AddMarker(MarkerNodeRigid(nodeNumber=cableData[0][0]))
```

As we like to move the cable relative to ground, we employ a simple user function which prescribes relative rotation and (corotated) translation in the joint:

```
def UFoffset(mbs, t, itemNumber, offsetUserFunctionParameters):
    x = SmoothStep(t, 2, 4, 0, 0.5)   #translate in local joint coordinates
    phi = SmoothStep(t, 5, 10, 0, pi) #rotates frame of mGround
    return [x, 0,0,0,0,phi]
```

Finally, we add the rigid joint (2D displacements and rotation around Z fixed) as GenericJoint. Note that for 2D objects, we may only fix X- and Y-translations, as well as the Z-rotation

```
mbs.AddObject(GenericJoint(markerNumbers=[mGround, mCable],
                           constrainedAxes=[1,1,0, 0,0,1],
                           offsetUserFunction=UFoffset,
                           visualization=VGenericJoint(axesRadius=0.01,
                                                       axesLength=0.02)))
```

As in the previous example, we just need to assemble and set up simulation parameters:

```
mbs.Assemble()
```

```
simulationSettings = exu.SimulationSettings()

tEnd = 10
stepSize = 0.005
simulationSettings.timeIntegration.numberOfSteps = int(tEnd/stepSize)
simulationSettings.timeIntegration.endTime = tEnd
simulationSettings.timeIntegration.verboseMode = 1
simulationSettings.solutionSettings.solutionWritePeriod = 0.005
simulationSettings.solutionSettings.writeSolutionToFile = True

simulationSettings.linearSolverType = exu.LinearSolverType.EigenSparse
simulationSettings.timeIntegration.newton.useModifiedNewton = True #for faster simulation

## add some visualization settings
SC.visualizationSettings.nodes.defaultSize = 0.01
SC.visualizationSettings.nodes.drawNodesAsPoint = False #show beam nodes
SC.visualizationSettings.bodies.beams.crossSectionFilled = True
```

Now start the solver with visualization and run the solution viewer afterwards, because simulation may be faster than you can follow:

```
SC.renderer.Start()
mbs.SolveDynamic(simulationSettings)
SC.renderer.Stop()

## visualize computed solution:
mbs.SolutionViewer()
```

The visualization window for the solution drawn at 6.5s is shown in Fig. 3.4.
This example should give you a good starting point to create beam models. See further examples, and more advanced functions, e.g., to create curved beam or reeving systems. There are also a sliding joint as well as axially moving beams and contact between beams and sheaves.
3D beams are still under development and include less functionality. In case, send a request at the GitHub discussion or issues.

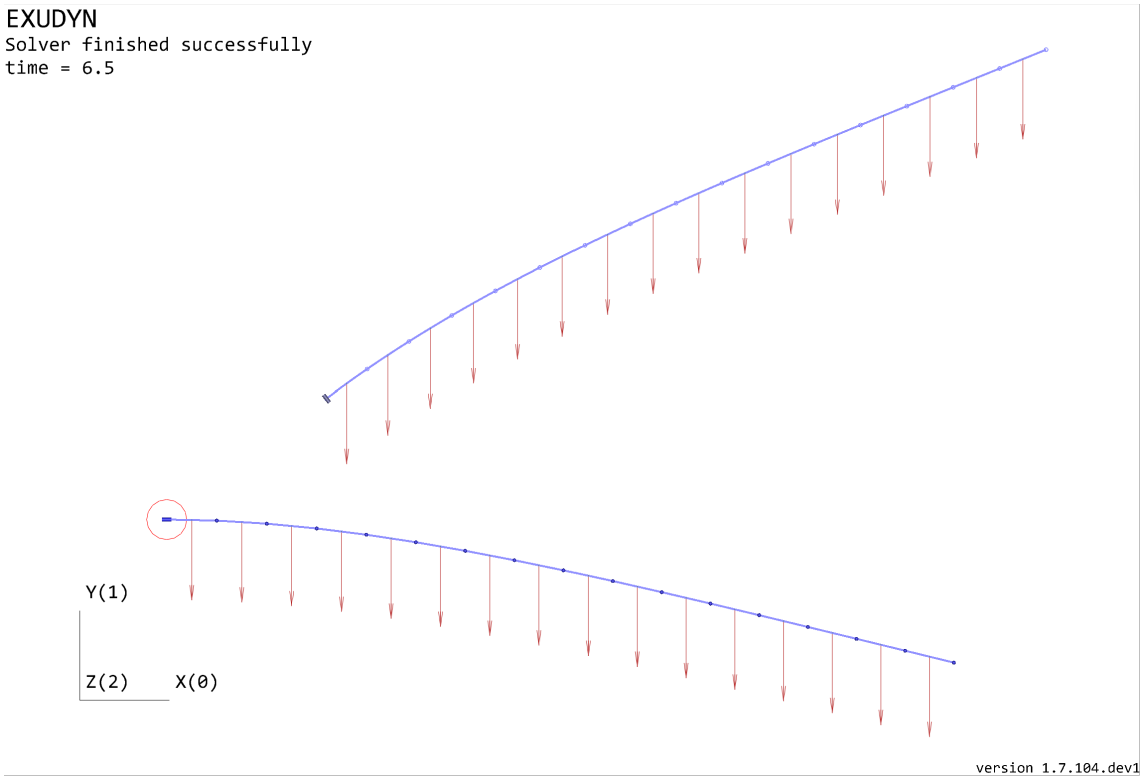


Figure 3.4: Render window showing the deformed state of the two beams at 6.5s. The lower beam is fixed at the left end, while the upper beam's support is translated and rotated.

3.4 Symbolic user function tutorial

The following tutorial demonstrates the setup of a nonlinear oscillator with a mass point and a force user function, using a Cartesian spring damper defined by symbolic user functions.

First, we import the necessary libraries, create a system container and a main system:

```
import exudyn as exu
from exudyn.utilities import *  # includes itemInterface and rigidBodyUtilities
import exudyn.graphics as graphics  # only import if it does not conflict
SC = exu.SystemContainer()
mbs = SC.AddSystem()
```

We set an abbreviation for the symbolic library for convenient access; often you can replace that with numpy:

```
esym = exu.symbolic
```

Now, define the parameters for the linear spring-damper system:

```
L = 0.5
mass = 1.6
k = 4000
omega0 = 50  # sqrt(k / mass)
dRel = 0.05
d = dRel * 2 * omega0
u0 = -0.08
v0 = 1
f = 80
```

Create the ground object and the mass point with initial conditions:

```
objectGround = mbs.CreateGround(referencePosition=[0, 0, 0])
massPoint = mbs.CreateMassPoint(referencePosition=[L, 0, 0],
                                initialDisplacement=[u0, 0, 0],
                                initialVelocity=[v0, 0, 0],
                                physicsMass=mass)
```

Set up the Cartesian spring damper between the ground and the mass point, and apply an external force on the mass point:

```
csd = mbs.CreateCartesianSpringDamper(bodyNumbers=[objectGround, massPoint],
                                      stiffness=[k, k, k],
                                      damping=[d, 0, 0],
                                      offset=[L, 0, 0])
load = mbs.CreateForce(bodyNumber=massPoint, loadVector=[f, 0, 0])
```

Add a sensor to monitor the position of the mass point:

```
sMass = mbs.AddSensor(SensorBody(bodyNumber=massPoint,
                                 storeInternal=True,
                                 outputVariableType=exu.OutputVariableType.Position))
```

Define a user function for the Cartesian spring damper, which may use Python or symbolic expressions:

```
def springForceUserFunction(mbs, t, itemNumber, u, v, k, d, offset):
    return [0.5 * u[0]**2 * k[0] + esym.sign(v[0]) * 10, k[1] * u[1], k[2] * u[2]]
```

We assign `CSDuserFunction` to the Python user function. This is used if no symbolic user function is used:

```
CSDuserFunction = springForceUserFunction
```

Up to now, everything looks like regular user functions. We now add an optional way to create a symbolic user function, which runs much faster in this case:

```
doSymbolic = True
if doSymbolic:
    CSDuserFunction = CreateSymbolicUserFunction(mbs, springForceUserFunction,
                                                 'springForceUserFunction', csd)
    # Check function:
    print('user function:\n', CSDuserFunction)
```

Set the user function to the object, assemble the system, and configure the simulation settings:

```
mbs.SetObjectParameter(csd, 'springForceUserFunction', CSDuserFunction)
mbs.Assemble()

simulationSettings = exu.SimulationSettings()
tEnd = 2
steps = 200000
simulationSettings.timeIntegration.numberOfSteps = steps
simulationSettings.timeIntegration.endTime = tEnd
simulationSettings.timeIntegration.verboseMode = 1
simulationSettings.solutionSettings.writeSolutionToFile = False
simulationSettings.solutionSettings.sensorsWritePeriod = 0.001
```

Finally, start the renderer and solver, then evaluate the solution:

```
SC.renderer.Start()
mbs.SolveDynamic(simulationSettings, solverType=exu.DynamicSolverType.ExplicitMidpoint)
SC.renderer.Stop()  # safely close rendering window!
n1 = mbs.GetObject(massPoint)['nodeNumber']
u = mbs.GetNodeOutput(n1, exu.OutputVariableType.Position)
print('u=', u)
mbs.PlotSensor(sMass)
```

NOTE: this tutorial has been mostly created with ChatGPT-4, and curated hereafter!

3.5 Flexible body – FFRF tutorial

The following tutorial includes flexible bodies, using the floating frame of reference formulation (FFRF), including Netgen and NGsolve for mesh and finite element data generation and uses modal

reduction for simulation. The tutorial will set up a body with Hurty-Craig-Bampton modes, giving a simple flexible pendulum meshed hinged with a revolute joint.

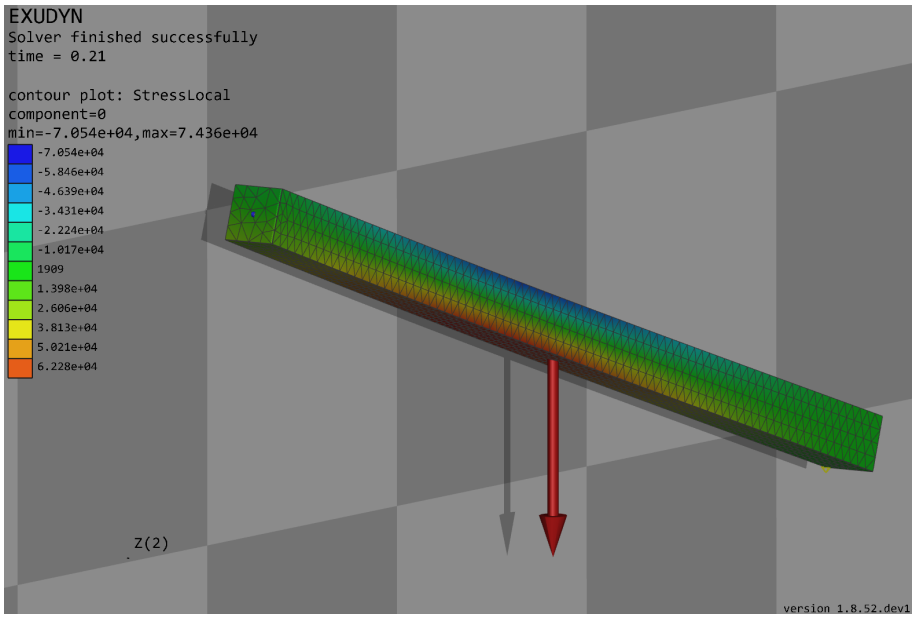


Figure 3.5: Screen shot of pendulum modeled with floating frame of reference formulation, using HCB modes, and meshed with Netgen.

We import the exudyn library, utilities, and other necessary modules:

```
import exudyn as exu
from exudyn.utilities import *  # includes itemInterface and rigidBodyUtilities
import exudyn.graphics as graphics  # only import if it does not conflict
from exudyn.FEM import *
import numpy as np
import time
import ngsolve as ngs
from netgen.meshing import *
from netgen.csg import *
```

Next, we need a `SystemContainer`, which contains all computable systems and adds a new MainSystem `mbs`:

```
SC = exu.SystemContainer()
mbs = SC.AddSystem()
```

Define the parameters and setup the Netgen mesh, using Netgen's CSG geometry. We create a simple brick in order to simplify the application of boundary interfaces and joints:

```
useGraphics = True
fileName = 'testData/netgenBrick'  # for load/save of FEM data

a = 0.025  # height/width of beam
b = a
```

```
h = 0.5 * a
L = 1     # Length of beam
nModes = 8

rho = 1000
Emodulus = 1e7 * 10
nu = 0.3
meshCreated = False
meshOrder = 1  # use order 2 for higher accuracy, but more unknowns

geo = CSGeometry()
block = OrthoBrick(Pnt(0, -a, -b), Pnt(L, a, b))
geo.Add(block)
mesh = ngs.Mesh(geo.GenerateMesh(maxh=h))
mesh.Curve(1)
```

When creating the geometry and mesh, we sometimes would like to verify that with Netgen's GUI. In Jupyter, this works smoother with `webgui_jupyter_widgets` – see the Netgen documention. Here we use a simple loop, which has to set `True` if visualization shall run:

```
if False:  # set this to true, if you want to visualize the mesh inside netgen/ngsolve
    import netgen.gui
    ngs.Draw(mesh)
    for i in range(10000000):
        netgen.Redraw()  # this makes the window interactive
        time.sleep(0.05)
```

Use the FEM interface to import the FEM model and create the FFRF reduced-order data stored in fem. Note that on importing the FEM structure from NGsolve (the FEM-module and solver of Netgen), we have to specify the mechanical properties related to the mesh:

```
fem = FEMinterface()
[bfM, bfK, fes] = fem.ImportMeshFromNGsolve(mesh, density=rho,
                                            youngsModulus=Emodulus,
                                            poissonsRatio=nu,
                                            meshOrder=meshOrder)
```

We could now just compute eigenmodes of the free bodies. However, as mentioned in the theory part, they do not respect boundary conditions and lead to low accuracy. Therefore, we use Hurty-Craig-Bampton modes. For them, we have to define boundary interfaces given as lists of node numbers as well as weights. We can use convenient functions from the `FEMinterface` class to retrieve nodes from planar or cylindrical surfaces (or we may define them in the finite element model ourselves):

```
pLeft = [0, -a, -b]
pRight = [L, -a, -b]
nTip = fem.GetNodeAtPoint(pRight) #for sensor
nodesLeftPlane = fem.GetNodesInPlane(pLeft, [-1, 0, 0])
weightsLeftPlane = fem.GetNodeWeightsFromSurfaceAreas(nodesLeftPlane)
nodesRightPlane = fem.GetNodesInPlane(pRight, [-1, 0, 0])
weightsRightPlane = fem.GetNodeWeightsFromSurfaceAreas(nodesRightPlane)
```

We define a list of boundaries, which are then passed to the function which computes modes. Note that by default the first boundary modes are eliminated as they are fixed to the reference frame of the FFRF object:

```
boundaryList = [nodesLeftPlane]

print("nNodes=", fem.NumberOfNodes())
print("compute HCB modes... ")
start_time = time.time()
fem.ComputeHurtyCraigBamptonModes(boundaryNodesList=boundaryList,
                                  nEigenModes=nModes,
                                  useSparseSolver=True,
                                  computationMode=HCBstaticModeSelection.RBE2)
print("HCB modes needed %.3f seconds" % (time.time() - start_time))
```

Compute stress modes for postprocessing, which is not needed for simulation, but useful for post-processing:

```
if True:
    mat = KirchhoffMaterial(Emodulus, nu, rho)
    varType = exu.OutputVariableType.StressLocal
    print("ComputePostProcessingModes ... (may take a while)")
    start_time = time.time()
    fem.ComputePostProcessingModesNGsolve(fes, material=mat,
                                          outputVariableType=varType)
    print("   ... needed %.3f seconds" % (time.time() - start_time))
    SC.visualizationSettings.contour.reduceRange = False
    SC.visualizationSettings.contour.outputVariable = varType
    SC.visualizationSettings.contour.outputVariableComponent = 0  # x-component
else:
    SC.visualizationSettings.contour.outputVariable = exu.OutputVariableType.
DisplacementLocal
    SC.visualizationSettings.contour.outputVariableComponent = 1
```

Having all data prepared now, we create the CMS element which is an object added then to `mbs`:

```
print("create CMS element ...")
cms = ObjectFFRFreducedOrderInterface(fem)
objFFRF = cms.AddObjectFFRFreducedOrder(mbs, positionRef=[0, 0, 0],
                                        initialVelocity=[0, 0, 0],
                                        initialAngularVelocity=[0, 0, 0],
                                        gravity=[0, -9.81, 0],
                                        color=[0.1, 0.9, 0.1, 1.])
```

Add markers and revolute joint, using a superelement marker:

```
nodeDrawSize = 0.0025  # for joint drawing

mRB = mbs.AddMarker(MarkerNodeRigid(nodeNumber=objFFRF['nRigidBody']))
oGround = mbs.AddObject(ObjectGround(referencePosition=[0, 0, 0]))
leftMidPoint = [0, 0, 0]
```

```
mGround = mbs.AddMarker(MarkerBodyRigid(bodyNumber=oGround, localPosition=leftMidPoint))
mLeft = mbs.AddMarker(MarkerSuperElementRigid(bodyNumber=objFFRF['oFFRFreducedOrder'],
                                              meshNodeNumbers=np.array(nodesLeftPlane),
                                              weightingFactors=weightsLeftPlane))
mbs.AddObject(GenericJoint(markerNumbers=[mGround, mLeft],
                           constrainedAxes=[1, 1, 1, 1, 1, 1 * 0],
                           visualization=VGenericJoint(axesRadius=0.1 * a, axesLength
=0.1 * a)))
```

Note that the marker is using weights, which are needed to compute accurate average (midpoint) positions from the non-uniform triangular surface mesh.
Now we finally add a sensor and assemble the system:

```
fileDir = 'solution/'
sensTipDispl = mbs.AddSensor(SensorSuperElement(bodyNumber=objFFRF['oFFRFreducedOrder'],
                                                meshNodeNumber=nTip,
                                                fileName=fileDir + 'nMidDisplacementCMS'
 + str(nModes) + 'Test.txt',
                                                outputVariableType=exu.
OutputVariableType.Displacement))

mbs.Assemble()
```

Set simulation settings and run the simulation:

```
simulationSettings = exu.SimulationSettings()

SC.visualizationSettings.nodes.defaultSize = nodeDrawSize
SC.visualizationSettings.nodes.drawNodesAsPoint = False
SC.visualizationSettings.connectors.defaultSize = 2 * nodeDrawSize
SC.visualizationSettings.nodes.show = False
SC.visualizationSettings.sensors.show = True
SC.visualizationSettings.sensors.defaultSize = 0.01
SC.visualizationSettings.markers.show = False
SC.visualizationSettings.loads.drawSimplified = False

h = 1e-3
tEnd = 4

simulationSettings.timeIntegration.numberOfSteps = int(tEnd / h)
simulationSettings.timeIntegration.endTime = tEnd
simulationSettings.timeIntegration.verboseMode = 1
simulationSettings.timeIntegration.newton.useModifiedNewton = True
simulationSettings.solutionSettings.sensorsWritePeriod = h
simulationSettings.timeIntegration.generalizedAlpha.spectralRadius = 0.8
simulationSettings.displayComputationTime = True

mbs.SolveDynamic(simulationSettings=simulationSettings)
```

```
uTip = mbs.GetSensorValues(sensTipDispl)[1]
print("nModes=", nModes, ", tip displacement=", uTip)

mbs.SolutionViewer()
```

When the solution viewer starts, it should show the stresses in a flexible swinging pendulum, see Fig. 3.5.

NOTE: this tutorial has been mostly created with ChatGPT-4, and curated hereafter!

Chapter 4

Notation

The notation is used to explain:

- typical symbols in equations and formulas (e.g., q)
- common types used as parameters in items (e.g., `PInt`)
- typical annotation of equations (or parts of it) and symbols (e.g., `ODE2`)

4.0.1 Common types

Common types are especially used in the definition of items. These types indicate how they need to be set (e.g., a `Vector3D` is set as a list of 3 floats or as a numpy array with 3 floats), and they usually include some range or size check (e.g., `PReal` is checked for being positive and non-zero):

- `float` ... a single-precision floating point number (note: in Python, '`float`' is used also for double precision numbers; in EXUDYN, internally floats are single precision numbers especially for graphics objects and OpenGL)
- `Real` ... a double-precision floating point number (note: in Python this is also of type '`float`')
- `UReal` ... same as `Real`, but may not be negative
- `PReal` ... same as `Real`, but must be positive, non-zero (e.g., step size may never be zero)
- `Index` ... deprecated, represents unsined integer, `UInt`
- `Int` ... a (signed) integer number, which converts to '`int`' in Python, '`int`' in C++
- `UInt` ... an unsigned integer number, which converts to '`int`' in Python
- `PInt` ... an positive integer number (> 0), which converts to '`int`' in Python
- `NodeIndex, MarkerIndex, ...` ... a special (non-negative) integer type to represent indices of nodes, markers, ...; specifically, an unintentional conversion from one index type to the other is not possible (e.g., to convert `NodeIndex` to `MarkerIndex`); see Section 6.1.1
- `String` ... a string
- `ArrayIndex` ... a list of integer numbers (either list or in some cases `numpy` arrays may be allowed)
- `ArrayNodeIndex` ... a list of node indices
- `Bool` ... a boolean parameter: either `True` or `False` ('`bool`' in Python)

- `VObjectMassPoint`, `VObjectRigidBody`, `VObjectGround`, etc. ... represents the visualization object of the underlying object; 'V' is put in front of object name
- `BodyGraphicsData` ... see Section 10.4
- `Vector2D` ... a list or `numpy` array of 2 real numbers
- `Vector3D` ... a list or `numpy` array of 3 real numbers
- `Vector'X'D` ... a list or `numpy` array of 'X' real numbers
- `Float4` ... a list of 4 float numbers
- `Vector` ... a list or `numpy` array of real numbers (length given by according object)
- `NumpyVector` ... a 1D `numpy` array with real numbers (size given by according object); similar as Vector, but not accepting list
- `Matrix3D` ... a list of lists or `numpy` array with 3×3 real numbers
- `Matrix6D` ... a list of lists or `numpy` array with 6×6 real numbers
- `NumpyMatrix` ... a 2D `numpy` array (matrix) with real numbers (size given by according object)
- `NumpyMatrixI` ... a 2D `numpy` array (matrix) with integer numbers (size given by according object)
- `MatrixContainer` ... a versatile representation for dense and sparse matrices, see Section 6.9.1
- `PyFunctionGraphicsData` ... a user function providing GraphicsData, see the user function description of the according object
- `PyFunctionMbsScalar...` ... a user function for the according object; the name is chosen according to the interface (arguments containing scalars, vectors, etc.) and is only used internally for code generation; see the according user function description

Note that for integers, there is also the `exu.InvalidIndex()` which is used to uniquely mark invalid indices, e.g., for default values of node numbers in objects or for other functions, often marked as `invalid (-1)` in the documentation. Currently, the invalid index is set to -1, but it may change in the future!

4.0.2 States and coordinate attributes

The following subscripts are used to define configurations of a quantity, e.g., for a vector of displacement coordinates $\mathbf{q}$:

- $\mathbf{q}_{\text{config}}$ ... $\mathbf{q}$ in any configuration
- $\mathbf{q}_{\text{ref}}$ ... $\mathbf{q}$ in reference configuration, e.g., reference coordinates: $\mathbf{c}_{\text{ref}}$
- $\mathbf{q}_{\text{ini}}$ ... $\mathbf{q}$ in initial configuration, e.g., initial displacements: $\mathbf{u}_{\text{ini}}$
- $\mathbf{q}_{\text{cur}}$ ... $\mathbf{q}$ in current configuration
- $\mathbf{q}_{\text{vis}}$ ... $\mathbf{q}$ in visualization configuration
- $\mathbf{q}_{\text{start of step}}$ ... $\mathbf{q}$ in start of step configuration

Note that the reference configuration is not included in other configurations, such that coordinates have to be usually understood relative to the reference configuration, see also Section 2.2.6.

As written in the introduction, the coordinates are attributed to certain types of equations and therefore, the following attributes are used (usually as subscript, e.g., $\mathbf{q}_{ODE2}$):

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

Time is usually defined as 'time' or t. The cross product or vector product '$\times$' is often replaced by the skew symmetric matrix using the tilde '~' symbol,

$$\mathbf{a} \times \mathbf{b} = \tilde{\mathbf{a}}\,\mathbf{b} = -\tilde{\mathbf{b}}\,\mathbf{a}\,, \tag{4.1}$$

in which ~ transforms a vector $\mathbf{a}$ into a skew-symmetric matrix $\tilde{\mathbf{a}}$. If the components of $\mathbf{a}$ are defined as $\mathbf{a} = [a_0, a_1, a_2]^{\mathrm{T}}$, then the skew-symmetric matrix reads

$$\tilde{\mathbf{a}} = \begin{bmatrix} 0 & -a_2 & a_1 \\ a_2 & 0 & -a_0 \\ -a_1 & a_0 & 0 \end{bmatrix}. \tag{4.2}$$

The inverse operation is denoted as vec, resulting in $\mathrm{vec}(\tilde{\mathbf{a}}) = \mathbf{a}$.

For the length of a vector we often use the abbreviation

$$\|\mathbf{a}\| = \sqrt{\mathbf{a}^T\mathbf{a}}\,. \tag{4.3}$$

A vector $\mathbf{a} = [x, y, z]^{\mathrm{T}}$ can be transformed into a diagonal matrix, e.g.,

$$\mathbf{A} = \mathrm{diag}(\mathbf{a}) = \begin{bmatrix} x & 0 & 0 \\ 0 & y & 0 \\ 0 & 0 & z \end{bmatrix} \tag{4.4}$$

4.0.3 Symbols in item equations

The following tables contains the common notation General **coordinates** are:

python name (or description)	**symbol**	**description**
displacement coordinates (ODE2)	$\mathbf{q} = [q_0, \ldots, q_n]^{\mathrm{T}}$	vector of n displacement based coordinates in any configuration; used for second order differential equations
rotation coordinates (ODE2)	$\boldsymbol{\psi} = [\psi_0, \ldots, \psi_\eta]^{\mathrm{T}}$	vector of η **rotation based coordinates** in any configuration; these coordinates are added to reference rotation parameters to provide the current rotation parameters; used for second order differential equations
coordinates (ODE1)	$\mathbf{y} = [y_0, \ldots, y_n]^{\mathrm{T}}$	vector of n coordinates for first order ordinary differential equations (ODE1) in any configuration
algebraic coordinates	$\mathbf{z} = [z_0, \ldots, z_m]^{\mathrm{T}}$	vector of m algebraic coordinates if not Lagrange multipliers in any configuration

Lagrange multipliers	$\boldsymbol{\lambda} = [\lambda_0, \ldots, \lambda_m]^\mathrm{T}$	vector of m Lagrange multipliers (=algebraic coordinates) in any configuration
data coordinates	$\mathbf{x} = [x_0, \ldots, x_l]^\mathrm{T}$	vector of l data coordinates in any configuration

The following parameters represent possible **OutputVariable** (list is not complete):

python name	**symbol**	**description**
Coordinate	$\mathbf{c} = [c_0, \ldots, c_n]^\mathrm{T}$	coordinate vector with n generalized coordinates c_i in any configuration; the letter c is used both for ODE1 and ODE2 coordinates
Coordinate_t	$\dot{\mathbf{c}} = [c_0, \ldots, c_n]^\mathrm{T}$	time derivative of coordinate vector
Displacement	${}^0\mathbf{u} = [u_0, u_1, u_2]^\mathrm{T}$	global displacement vector with 3 displacement coordinates u_i in any configuration; in 1D or 2D objects, some of there coordinates may be zero
Rotation	$[\varphi_0, \varphi_1, \varphi_2]^\mathrm{T}_\mathrm{config}$	vector with 3 components of the Euler angles in xyz-sequence (${}^{0b}\mathbf{A}_\mathrm{config} =: \mathbf{A}_0(\varphi_0) \cdot \mathbf{A}_1(\varphi_1) \cdot \mathbf{A}_2(\varphi_2)$), recomputed from rotation matrix
Rotation (alt.)	$\boldsymbol{\theta} = [\theta_0, \ldots, \theta_n]^\mathrm{T}$	vector of **rotation parameters** (e.g., Euler parameters, Tait Bryan angles, ...) with n coordinates θ_i in any configuration
Identity matrix	$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & 1 \end{bmatrix}$	the identity matrix, very often $\mathbf{I} = \mathbf{I}_{3\times3}$, the 3×3 identity matrix
Identity transformation	${}^{0b}\mathbf{I}_{3\times3} = \mathbf{I}_{3\times3}$	converts body-fixed into global coordinates, e.g., ${}^0\mathbf{x} = {}^{0b}\mathbf{I}_{3\times3}\ {}^b\mathbf{x}$, thus resulting in ${}^0\mathbf{x} = {}^b\mathbf{x}$ in this case
RotationMatrix	${}^{0b}\mathbf{A} = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix}$	a 3D rotation matrix, which transforms local (e.g., body b) to global coordinates (0): ${}^0\mathbf{x} = {}^{0b}\mathbf{A}\ {}^b\mathbf{x}$
RotationMatrixX	${}^{01}\mathbf{A}_0(\theta_0) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_0) & -\sin(\theta_0) \\ 0 & \sin(\theta_0) & \cos(\theta_0) \end{bmatrix}$	rotation matrix for rotation around X axis (axis 0), transforming a vector from frame 1 to frame 0
RotationMatrixY	${}^{01}\mathbf{A}_1(\theta_1) = \begin{bmatrix} \cos(\theta_1) & 0 & \sin(\theta_1) \\ 0 & 1 & 0 \\ -\sin(\theta_1) & 0 & \cos(\theta_1) \end{bmatrix}$	rotation matrix for rotation around Y axis (axis 1), transforming a vector from frame 1 to frame 0
RotationMatrixZ	${}^{01}\mathbf{A}_2(\theta_2) = \begin{bmatrix} \cos(\theta_2) & -\sin(\theta_2) & 0 \\ \sin(\theta_2) & \cos(\theta_2) & 0 \\ 0 & 0 & 1 \end{bmatrix}$	rotation matrix for rotation around Z axis (axis 2), transforming a vector from frame 1 to frame 0
Position	${}^0\mathbf{p} = [p_0, p_1, p_2]^\mathrm{T}$	global position vector with 3 position coordinates p_i in any configuration
Velocity	${}^0\mathbf{v} = {}^0\dot{\mathbf{u}} = [v_0, v_1, v_2]^\mathrm{T}$	global velocity vector with 3 displacement coordinates v_i in any configuration

AngularVelocity	${}^0\boldsymbol{\omega} = [\omega_0, \ldots, \omega_2]^\mathrm{T}$	global angular velocity vector with 3 coordinates ω_i in any configuration
Acceleration	${}^0\mathbf{a} = {}^0\ddot{\mathbf{u}} = [a_0, a_1, a_2]^\mathrm{T}$	global acceleration vector with 3 displacement coordinates a_i in any configuration
AngularAcceleration	${}^0\boldsymbol{\alpha} = {}^0\dot{\boldsymbol{\omega}} = [\alpha_0, \ldots, \alpha_2]^\mathrm{T}$	global angular acceleration vector with 3 coordinates α_i in any configuration
VelocityLocal	${}^b\mathbf{v} = [v_0, v_1, v_2]^\mathrm{T}$	local (body-fixed) velocity vector with 3 displacement coordinates v_i in any configuration
AngularVelocityLocal	${}^b\boldsymbol{\omega} = [\omega_0, \ldots, \omega_2]^\mathrm{T}$	local (body-fixed) angular velocity vector with 3 coordinates ω_i in any configuration
Force	${}^0\mathbf{f} = [f_0, \ldots, f_2]^\mathrm{T}$	vector of 3 force components in global coordinates
Torque	${}^0\boldsymbol{\tau} = [\tau_0, \ldots, \tau_2]^\mathrm{T}$	vector of 3 torque components in global coordinates

The following table collects some typical **input parameters** for nodes, objects and markers:

python name	**symbol**	**description**
referenceCoordinates	$\mathbf{c}_\mathrm{ref} = [c_0, \ldots, c_n]^\mathrm{T}_\mathrm{ref} = [c_{\mathrm{Ref},0}, \ldots, c_{\mathrm{Ref},n}]^\mathrm{T}_\mathrm{ref}$	n coordinates of reference configuration (can usually be set at initialization of nodes)
initialCoordinates	$\mathbf{c}_\mathrm{ini}$	initial coordinates with generalized or mixed displacement/rotation quantities (can usually be set at initialization of nodes)
reference point	${}^0\mathbf{r} = [r_0, r_1, r_2]^\mathrm{T}$	reference point of body, e.g., for rigid bodies or floating frame of reference formulation (FFRF) bodies, in any configuration; NOTE: for ANCF elements, ${}^0\mathbf{r}$ is used for the position vector to the beam centerline
localPosition	${}^b\mathbf{b} = [{}^bb_0, {}^bb_1, {}^bb_2]^\mathrm{T}$	local (body-fixed) position vector with 3 position coordinates b_i in any configuration, measured relative to reference point; NOTE: for rigid bodies, ${}^0\mathbf{p} = {}^0\mathbf{r} + {}^{0b}\mathbf{A}\,{}^b\mathbf{b}$; localPosition is used for definition of body-fixed local position of markers, sensors, COM, etc.

4.1 LHS–RHS naming conventions in EXUDYN

The general idea of the `Exudyn` is to have objects, which provide equations (ODE2, ODE1, AE). The solver then assembles these equations and solves the static or dynamic problem. The system structure and solver are similar but much more advanced and modular as earlier solvers by the main developer [24, 17, 20].

Functions and variables contain the abbreviations LHS and RHS, sometimes lower-case, in order to distinguish if terms are computed at the LHS or RHS.

The objects have the following LHS–RHS conventions:

- the acceleration term, e.g., $m \cdot \ddot{q}$ is always positive on the LHS
- objects, connectors, etc., use LHS conventions for most terms: mass, stiffness matrix, elastic forces, damping, etc., are computed at LHS of the object equation
- object forces are written at the RHS of the object equation
- in case of constraint or connector equations, there is no LHS or RHS, as there is no acceleration term.

Therefore, the computation function evaluates the term as given in the description of the object, adding it to the LHS. Object equations may read, e.g., for one coordinate q, mass m, damping coefficient d, stiffness k and applied force f,

$$\underbrace{m \cdot \ddot{q} + d \cdot \dot{q} + k \cdot q}_{LHS} = \underbrace{f}_{RHS} \tag{4.5}$$

In this case, the C++ function `ComputeODE2LHS(const Vector& ode2Lhs)` will compute the term $d \cdot \dot{q} + k \cdot q$ with positive sign. Note that the acceleration term $m \cdot \ddot{q}$ is computed separately, as it is computed from mass matrix and acceleration.

However, system quantities (e.g. within the solver) are always written on RHS[1]:

$$\underbrace{M_{sys} \cdot \ddot{q}_{sys}}_{LHS} = \underbrace{f_{sys}}_{RHS} \ . \tag{4.6}$$

In the case of the object equation

$$m \cdot \ddot{q} + d \cdot \dot{q} + k \cdot q = f \ , \tag{4.7}$$

the RHS term becomes $f_{sys} = -(d \cdot \dot{q} + k \cdot q) + f$ and it is computed by the C++ function `ComputeSystemODE2RHS`. This means, that during computation, terms which appear at the LHS of the object are transferred to the RHS of the system equation. This enables a simpler setup of equations for the solver.

4.2 System assembly

Assembling equations of motion is done within the C++ class `CSystem`, see the file `CSystem.cpp`. The general idea is to assemble, i.e. to sum up, (parts of) residuals attributed by different objects. The summation process is based on coordinate indices to which the single equations belong to. Let's assume that we have two simple `ObjectMass1D` objects, with object indices $o0$ and $o1$ and having mass m_0 and m_1. They are connected to nodes of type `Node1D` $n0$ and $n1$, with global coordinate indices $c0$ and $c1$. The partial object residuals, which are fully independent equations, read

$$m_0 \cdot \ddot{q}_{c0} = RHS_{c0} \ , \tag{4.8}$$

$$m_1 \cdot \ddot{q}_{c1} = RHS_{c1} \ , \tag{4.9}$$

where RHS_{c0} and RHS_{c1} the right-hand-side of the respective equations/coordinates. They represent forces, e.g., from `LoadCoordinate` items (which directly are applied to coordinates of nodes), say f_{c0}

[1] except for the acceleration $\times$ mass matrix and constraint reaction forces, see Eq. (11.1)

and f_{c1}, that are in case also summed up on the right hand side. Let us for now assume that

$$RHS_{c0} = f_{c0} \quad \text{and} \quad RHS_{c1} = f_{c1} \,. \tag{4.10}$$

Now we add another `ObjectMass1D` object with object index $o2$, having mass m_2, but letting the object *again* use node $n0$ with coordinate $c0$. In this case, the total object residuals read

$$(m_0 + m_2) \cdot \ddot{q}_{c0} = RHS_{c0} \,, \tag{4.11}$$

$$m_1 \cdot \ddot{q}_{c1} = RHS_{c1} \,. \tag{4.12}$$

It is clear, that now the mass in the first equation is increased due to the fact that two objects contribute to the same coordinate. The same would happen, if several loads are applied to the same coordinate.

Finally, if we add a `CoordinateSpringDamper`, assuming a spring k between coordinates $c0$ and $c1$, the RHS of equations related to $c0$ and $c1$ is now augmented to

$$RHS_{c0} = f_{c0} + k \cdot (q_{c1} - q_{c0}) \,, \tag{4.13}$$

$$RHS_{c1} = f_{c1} + k \cdot (q_{c0} - q_{c1}) \,. \tag{4.14}$$

The system of equation would therefore read

$$(m_0 + m_2) \cdot \ddot{q}_{c0} = f_{c0} + k \cdot (q_{c1} - q_{c0}) \,, \tag{4.15}$$

$$m_1 \cdot \ddot{q}_{c1} = f_{c1} + k \cdot (q_{c0} - q_{c1}) \,. \tag{4.16}$$

It should be noted, that all (components of) residuals ('equations') are summed up for the according coordinates, and also all contributions to the mass matrix. Only constraint equations, which are related to Lagrange parameters always get their 'own' Lagrange multipliers, which are automatically assigned by the system and therefore independent for every constraint.

4.3 Nomenclature for system equations of motion and solvers

Using the basic notation for coordinates in Section 4, we use the following quantities and symbols for equations of motion and solvers:

quantity	**symbol**	**description**
number of ODE2 coordinates	n_q	second order ordinary differential equations (ODE2)
number of ODE1 coordinates	n_y	first order ordinary differential equations (ODE1)
number of AE coordinates	m	algebraic equations (AE)
number of system coordinates	n_s	system equations
ODE2 coordinates	$\mathbf{q} = [q_0, \ldots, q_{n_q}]^T$	ODE2, displacement-based coordinates (could also be rotation or deformation coordinates)
ODE2 velocities	$\mathbf{v} = \dot{\mathbf{q}} = [\dot{q}_0, \ldots, \dot{q}_{n_q}]^T$	ODE2 velocity coordinates
ODE2 accelerations	$\ddot{\mathbf{q}} = [\ddot{q}_0, \ldots, \ddot{q}_{n_q}]^T$	ODE2 acceleration coordinates

ODE1 coordinates	$\mathbf{y} = [y_0, \ldots, y_{n_y}]^\mathrm{T}$	vector of n_y coordinates for first order ordinary differential equations (ODE1)
ODE1 velocities	$\dot{\mathbf{y}} = [\dot{y}_0, \ldots, \dot{y}_{n_y}]^\mathrm{T}$	vector of n velocities for first order ordinary differential equations (ODE1)
ODE2 Lagrange multipliers	$\boldsymbol{\lambda} = [\lambda_0, \ldots, \lambda_m]^\mathrm{T}$	vector of m Lagrange multipliers (=algebraic coordinates), representing the linear factors (often forces or torques) to fulfill the algebraic equations; for ODE1 and ODE2 coordinates
data coordinates	$\mathbf{x} = [x_0, \ldots, x_l]^\mathrm{T}$	vector of l data coordinates in any configuration
RHS ODE2	$\mathbf{f}_q \in \mathbb{R}^{n_q}$	right-hand-side of ODE2 equations; (all terms except mass matrix $\times$ acceleration and joint reaction forces)
RHS ODE1	$\mathbf{f}_q \in \mathbb{R}^{n_y}$	right-hand-side of ODE1 equations
AE	$\mathbf{g} \in \mathbb{R}^m$	algebraic equations
mass matrix	$\mathbf{M} \in \mathbb{R}^{n_q \times n_q}$	mass matrix, only for ODE2 equations
(tangent) stiffness matrix	$\mathbf{K} \in \mathbb{R}^{n_q \times n_q}$	includes all derivatives of $\mathbf{f}_q$ w.r.t. $\mathbf{q}$
damping/gyroscopic matrix	$\mathbf{D} \in \mathbb{R}^{n_q \times n_q}$	includes all derivatives of $\mathbf{f}_q$ w.r.t. $\mathbf{v}$
step size	h	current step size in time integration method
residual	$\mathbf{r}_q \in \mathbb{R}^{n_q}$, $\mathbf{r}_y \in \mathbb{R}^{n_y}$, $\mathbf{r}_\lambda \in \mathbb{R}^m$	residuals for each type of coordinates within static/time integration – depends on method
system residual	$\mathbf{r} \in \mathbb{R}^{n_s}$	system residual – depends on method
system coordinates	$\boldsymbol{\xi}$	system coordinates and unknowns for solver; definition depends on solver
Jacobian	$\mathbf{J} \in \mathbb{R}^{n_s \times n_s}$	system Jacobian – depends on method

Chapter 5

Theory and formulations

This section includes some material on formulations and general theoretical backgrounds for `Exudyn`. Note that the formulation of nodes, objects, markers, etc. is presented right at the subsection of each object, see Section 8. Furthermore, the general notations are described in Section 4 and an overview on the system assembly and system equations of motion is given in Section 4, solvers are described in Section 11.

5.1 Introduction to multibody systems

The intention of the subsequent sections is to give brief introductions into subjects which are essential for working with multibody systems. Some of the contents may not go deep enough, which is why we refer to the according literature. In particular, the contents follow the books of

- C. Woernle [64]
- A.A. Shabana [56]
- P.E. Nikravesh [46]

Multibody systems are mechanical systems of bodies (sometimes only one body), which are interconnected. Multibody systems may include linear dynamical systems as used in vibration analysis and rotordynamics. However, multibody systems are often characterized by nonlinear and differential-algebraic equations, and they typically cover arbitrary rigid body motions. In a broader sense, multibody systems represent the mechanical part of mechatronics systems. The range of multibody systems might vary from simple one or two-mass oscillators, or single or double pendulums, up to complex full-scale vehicles or aerospace systems.

Multibody systems (MBS) are characterized by rigid or flexible bodies that are interconnected by means of joints, contacts, springs and dampers and usually driven by electrical, hydraulic or pneumatic actuators, and possibly subject to external forces. Sophisticated models of joints include the deformation of the joint surface and consider contact and friction models.

5.1.1 Historical development

The roots of analysis tools for multibody systems trace back to Newton's laws and Euler's contributions to the motion of rigid bodies. Lagrange introduced a formalism to derive the equations of motions for complex rigid or flexible multibody systems, which is still state of the art. Multibody system dynamics attracted the attention of engineers as soon as computer power was large enough to simulate the nonlinear dynamic behavior of such systems numerically, however only using ordinary differential equations for modeling in the beginning. Since the 1960s, flexibility of structures has been introduced, e.g., in order to understand the instability of a satellite with flexible antennas or a helicopter's rotor blades. Since the 1990s flexible multibody system methods have been applied in vehicle and aerospace industry, in particular because model order reduction became available and real life components could be simulated. Models where mainly made out of kinematic trees, using minimum coordinate models in the beginning. For long time, algebraic constraints could only be approximated, such as with Baumgarte's stabilization introduced in the 1970. Major advances on differential-algebraic solvers have been made in the 1990s, introducing Raudau-5 of Hairer and Wanner, as well as BDF solvers of Linda Petzold. Because both approaches had drawbacks, the implicit generalized-alpha solver (Chung, Hubert; Arnold, Brüls) has become a quasi-standard in the 2000s, as it allows to solve almost any index 3 (position level) constraint problem with high efficiency.

5.1.2 Simulation tools in computational engineering

As in Exudyn, the main intention is to create computer models for mechanical systems. The further analysis of the system is left to the users' needs. In particular, engineers are interested in

(Forward) dynamic analysis:

- Dynamic simulation, by setting initial conditions and using time integration for the equations of motion of the model
- Outputs may be displacements, rotations, velocities, angular velocities, accelerations, stresses, joint or support forces, etc.

(Forward) static analysis:

- (Nonlinear) static computation, by solving the equations of motion without inertia forces and without time-dependency (usually settings accelerations and velocities to zero, but possibly also computing quasi-static or stationary solutions)
- Outputs may be displacements, rotations, stresses, joint or support forces, etc.

Kinematic analysis:

- Similar to a static analysis, inertia forces are neglected, and in addition forces are neglected as well.
- Inputs may be prescribed translations or rotations to specific bodies or joints.
- Outputs may be ratios of angular velocities, or the motion of bodies due to prescribed motion in specific joints

Analysis of a multibody system

- Kinematic analysis regarding system degree of freedom (DOF), redundant constraints, jacobians (e.g., how joint velocities affect velocities of a particular body)
- Critical loads (buckling and bifurcation)
- Eigenfrequency and eigenmode analysis
- Instability analysis (e.g. critical velocity of trains, fluids, etc.)

Inverse static and dynamic analysis

- Computation of required loads in order to reach certain displacements
- Inverse kinematics: computation of joint angles of robots in order to obtain a certain tool-center position
- Inverse dynamics: computation of joint forces to compensate weight and inertia forces of robot mechanism under motion

Optimization

- Minimization of maximum stress within bodies, or of stress to yield-stress ratio
- Optimizing kinematic behavior
- Minimization of energy
- Minimization of damage and wear

Sensitivity analysis

- Investigate the influence of small variations of the parameters (dimensions, material, etc.) on the measured values, such as displacements, forces, etc.

Realtime simulation

- Hardware-in-the-loop simulation: e.g. in virtual reality, the user acts in reality, while the machine input and feedback is simulated in realtime.
- Software-in-the-loop simulation: while the real machine, or parts of it, are considered, other hardware, controllers or humans are emulated by software.

5.1.3 Components of a multibody system

Any multibody system basically consists of a set of components, each of which with specific properties and dependencies. Main components are:

Bodies:

- mass points
- rigid bodies
- flexible bodies (in general)

- finite elements (often representing flexible bodies)

Connectors:

- spring-dampers (bushings)
- joints (represented by algebraic equations)
- general constraints (such as on coordinates)

Further components are:

- **Loads**: forces, moments, body loads, gravity, applied to bodies; loads are usually not coupled to the motion of bodies, as compared to springs; however, follower loads may depend on body rotation. Loads may be time-dependent, which can be realized by defining a load together with a user function, or prescribing a load in each time step.
- **Sensors**: measure displacements, velocities, stresses, strains, bending moments, loads, etc., and the do not affect the system behavior, as long as the are not used in controllers or control units.
- **graphical representation** for bodies and system components
- **Joint control**: certain feed-back control laws (e.g., PID) for each joint axis, in order to prescribe joint rotation, velocity, etc., which may in particular result in coupling of system equations.
- **Motion control, path planning**: prescribing a desired path for one or several joints, using a feed-forward control
- **Control units**: controlling the behavior of the system (such as a sequence of tasks, feeding a list of motion control tasks for several joint axes); often realized as scripts

5.1.4 Kinematics basics

Kinematics, also referred to as the **geometry of motion**, is a fundamental aspect of multibody systems that focuses on the description of **motion without considering the forces that cause it**. It involves the study of the positions, velocities, and accelerations of body parts in a system and how these quantities change over time. For rigid bodies or bodies that employ frames, it also includes rotations, angular velocities and angular accelerations. This branch of mechanics provides the essential framework for understanding how individual components of a multibody system move relative to one another, laying the groundwork for the subsequent analysis of dynamics where forces are taken into account.

Kinematics is also related to to **trajectories** (including frames co-moving along trajectories), **curve length**, motion of **frames**, and **homogeneous transformations**. **Kinematic constraints** are used to define joints, basically by constraining specific relative motion, such as the distance between two particles, the relative rotation or the relative translation along axis. For **non-holonomic constraints**, the constraint equations have to be defined at velocity level, such as the idealized rolling (wheel) or slipping conditions (sled skid).

Kinematic synthesis involves designing a mechanism's geometry, including its links and joints, to achieve specific motions. In contrast, **kinematic analysis** focuses on determining the kinematic quantities, such as positions, velocities, and accelerations, of certain points or components within a mechanism that is performing a prescribed motion.

5.1.4.1 Links, joints, kinematic chains and trees

Kinematics has been studied much earlier than the development of multibody system dynamics. Traditionally in kinematics, joints were the essential part to define specific (relative) motion. The various joints in a system were linked by "**links**", which had no further role in the geometry of motion, without consideration for forces or inertia. However, with advancements in the field of rigid and flexible multibody dynamics, the significance of links has been elevated, and they are more commonly referred to as bodies, considering their dynamic properties.

A series of interconnected links forms a kinematic chain. In most mechanisms, except for those designed for flight, at least one link is stationary, serving as a reference point or frame for the system. This stationary link is typically referred to as the ground or frame. Mechanisms can be classified based on the motion of their links: if all links move within a single plane, the mechanism is called a planar mechanism; otherwise, it is known as a spatial mechanism.

It is also crucial to differentiate between open-loop and closed-loop mechanisms. An open-loop mechanism is characterized by a configuration where traversing through the links in sequence does not lead back to the starting point. In the general case, such an open-loop system is represented by a kinematic tree, which has a root link, and every link can have arbitrary many joints – as long as it leads to no single closed loop. Conversely, a closed-loop mechanism features a configuration where at least one path forms a loop, including the ground link, allowing for the possibility of returning to the starting link through the sequence of connections.

Fig. 5.1a shows a double pendulum, which is an example of an open-loop mechanism. A closed-loop mechanism is shown in Fig. 5.1b, which is a four-bar linkage. As mentioned before, the fourth link is the ground link.

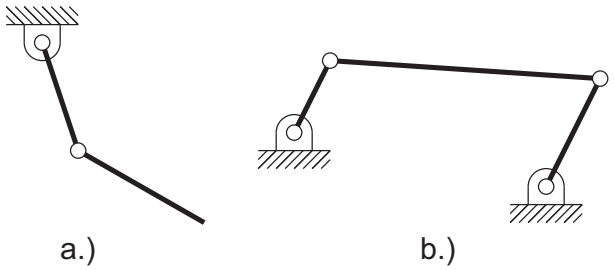


Figure 5.1: a.) Open loop mechanism (double pendulum), b.) Closed loops mechanism (four-bar linkage).

5.1.4.2 Kinematic pairs

During early achievements in kinematics, joints have been denoted as kinematic pairs, where the pair represents two bodies where joint constraints are imposed at. There are joints denoted as **lower pairs**, which are defined by the idealized contact of two rigid surfaces, each of which attached to one of the two bodies. The most prominent types, with mention of the respective object names in `Exudyn`, are:

- spherical joint (S), ball and socket joint; three constraints on relative translation of body-fixed

points; `ObjectJointSpherical`

- revolute (R) joint, hinged joint; three spherical joint constraints plus two on tilting around axes normal to the joint axis; `ObjectJointRevolute`
- prismatic (P) joint; tree constraints on relative rotation; `ObjectJointPrismatic`
- screw (helical, H) joint; `not yet available`
- cylindrical (C) joint; `ObjectJointGeneric`
- planar joint; restricts motion to plane, thus adding one relative translation and two rotation constraints; `ObjectJointGeneric`

A very practical joint, which has no relevance for kinematics, is the rigid joint (6 constraints, `ObjectJointGeneric`). It simply constrains the rigid body motion of two bodies, making the two bodies identical from the rigid body kinematics viewpoint.

Higher pairs involve changing surfaces or curves of contact, such as in a cam and follower or gears.

Indeed, some cases in constraints may not be represented by a set of kinematic pairs, such as several mass points arranged along an inextensible string. Therefore, codeName usually employs kinematic pairs, but also allows an arbitrary number of bodies to be coupled.

5.1.5 Euler's and Chasles's Theorems

In the following, we consider so-called active rotations of bodies. Consider a body rotating with an angular velocity ω. Thereing, the orientation of the body can be given with respect a previous orientation using a transformation matrix (rotation matrix). Here, the components of the transformation matrix are given as a function of time. When reconsidering the DOF of a mechanism, it is thus the question, how many independent coordinates are required to describe a spatial rotation?

The answer to this question is given by Euler's theorem:

- **Euler's theorem**: The general displacement of a body with one point fixed is a rotation about some axis.

The latter theorem states that at any current time t and after any rotations, the orientation of the body can be described by a rotation axis and a single rotation angle. Note that this rotation axis is used to describe the rotation of the body from the initial to the current time at once, although, the rotation might have been performed by means of many different successive rotations. This rotation axis is different from the so-called **instantaneous axis of rotation** of the body. As a consequence, points lying at the rotation axis are not affected by this rotation.

In addition to Euler's theorem, we mention **Chasles's theorem**, which reads as follows:

- **Chasles's theorem**: The most general displacement of a body is a translation plus a rotation.

Therefore, the general motion of the rigid body follows from Euler's theorem plus a translation of the point which was originally fixed.

5.1.6 Degree of freedom – DOF

According to Nikravesh, "The minimum number of coordinates required to fully describe the configuration of a system is called the number of degrees of freedom of the system". We may consider two examples given in Fig. 5.2, to understand the DOF. A double pendulum is shown, which can be described with no less than two independent angles ϕ_1 and ϕ_2, thus the system has two degrees of freedom.

As a second example, a four-bar mechanism is shown, see Fig. 5.2b. There are three angles ϕ_a, ϕ_b and ϕ_c related with the current configuration of the system. However, there are two algebraic relations (constraint conditions) between these three angles, given as

$$\begin{aligned} l_a \sin(\phi_a) + l_b \sin(\phi_b) - l_c \sin(\phi_c) - d_1 &= 0 \\ l_a \cos(\phi_a) - l_b \cos(\phi_b) - l_c \cos(\phi_c) + h_1 &= 0 \end{aligned} \tag{5.1}$$

In the latter two equations, the angles ϕ_a and ϕ_b can be expressed in terms of ϕ_c. Thus, ϕ_c is the only remaining independent (minimum) coordinate of the system, and as a consequence, the system has only one degree of freedom. Regarding the four-bar mechanism, there exist some configurations, which can lead to bifurcation and a change in the degrees of freedom – but this is usually avoided in practical cases.

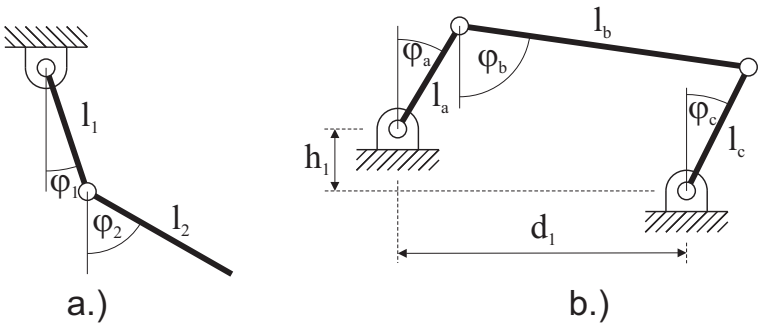


Figure 5.2: Examples of mechanisms with different degrees of freedom.

5.1.7 Non-holonomic constraints

It should be noted, that the above given definition works well with holonomic constraints. In case of non-holonomic systems, the non-holonomic constraints, which may only be given at velocity level, lead to different degrees of freedom on position and velocity level. Due to the fact that time integration integrates velocities and accelerations, and constraints may be formulated on position or velocity in `Exudyn`, non-holonomic constraints are not largely influencing the structure of the system of equations. However, in view of the constraint jacobian, it usually reflects the constraints on velocity level, as this is the matrix which the incremental solver uses to update coordinates.

5.1.8 Dependent and independent coordinates

Assuming a holonomic mechanical system with k degrees of freedom, one can find k **independent coordinates** (**minimum coordinates**) to completely describe the system configuration. Note that these coordinates do not necessarily have the meaning of displacement, length or angle, but they may be more general (generalized coordinates).

In addition to the independent coordinates, there may be **dependent coordinates**, similar to the angles ϕ_a and ϕ_b which are dependent on ϕ_c in the example of the four-bar mechanism. It is left to the engineer, whether to find the minimum coordinates of a system and to write all equations in terms of these, or to work with a larger set of independent and dependent coordinates together with algebraic constraint conditions.

5.1.9 Chebychev-Grübler-Kutzbach criterion

A **rigid body in space has six degrees of freedom**, and thus, six independent coordinates can be used to describe its configuration. Thus for a system with n_b bodies, there are $6 \cdot n_b$ coordinates, to describe the bodies. Assuming that there are a set of spheric, revolute, prismatic and other joints, which reduce the degrees of freedom, we denote the number of **independent constraints** to be of size n_c. It is important, that the constraint equations are (linearly) independent, because in kinematics it is possible to restrict the motion with redundant constraints, see later.

Finally, having n_b rigid bodies n_c scalar, independent constraint equations, the degrees of freedom are given as

$$n_{\text{DOF}} = 6 \cdot n_b - n_c \tag{5.2}$$

which is denoted sometimes as the **Kutzbach**, or **Chebychev-Grübler-Kutzbach criterion**.

In the case of planar mechanisms, a body obtains only three degrees of freedom. Therefore the Chebychev-Grübler-Kutzbach criterion reads

$$DOF_{\text{planar}} = 3 \cdot n_b - n_c \tag{5.3}$$

As a spatial example, consider again the double pendulum Fig. 5.2a as a spatial mechanism. In this case, there are two bodies, $n_b = 2$ and two revolute joints with 5 constraints each, giving $n_c = 10$. Thus, the Chebychev-Grübler-Kutzbach criterion gives $n_{\text{DOF}} = 12 - 10 = 2$, which we expect.

As another example, consider a spatial four-bar mechanism according to Fig. 5.2b. In this case, there are four bodies, $n_b = 4$, four revolute joints with 5 constraints each and a ground joints with 6 constraints, totalling at $n_c = 4 \cdot 5 + 6 = 26$. Thus, Eq. (5.2) gives

$$n_{\text{DOF}} = 24 - 26 = -2. \tag{5.4}$$

This is certainly not what we expect, as we know that the mechanism can move and has $n_{\text{DOF}} = 1$. The reason for this number lies in redundant constraints, which may not be counted for the Chebychev-Grübler-Kutzbach criterion. A solution to this problem is to replace one revolute joint by a planar revolute joint (2 constraints), which then gives

$$n_c = 3 \cdot 5 + 2 + 6 = 23 \quad \text{and} \quad n_{\text{DOF}} = 1. \tag{5.5}$$

Therefore, Exudyn uses an **extended Chebychev-Grübler-Kutzbach criterion** for redundant constraints,

$$n^*_{\text{DOF}} = n_{\text{ODE2}} - (n_c - n_{ca} - n_r) \tag{5.6}$$

where n_r is the number of redundant constraints and n_{ODE2} is the number of ODE2 coordinates, which may be $6 \cdot n_b$ for purely spatial rigid bodies, n_{ca} are pure algebraic constraints and n_r are redundant constraints.

In Exudyn, there is a function ComputeSystemDegreeOfFreedom in the solver module, available as

```
mbs.ComputeSystemDegreeOfFreedom(verbose=True)
```

which allows to compute ODE2 coordinates (= n_{ODE2}), total constraints (= n_c), redundant constraints (= n_r), and the degree of freedom (= n^*_{DOF}). In addition, the function also computes the number of pure algebraic constraints (= n_{ca}), which are internal constraints, which only act on the algebraic quantities but do not affect ODE2 coordinates. As an example for pure algebraic constraints, in a GenericJoint inactive constraints are replaced by $\lambda_i = 0$ (λ being the Lagrange multiplier), thus reducing the number of effective constraints in the system.

5.1.10 Generalized coordinates

In **Cartesian coordinates**, three parameters (denoted as coordinates) are used to uniquely define the position of any point with respect to a Cartesian coordinate system. The position and orientation of a rigid body can be uniquely defined by means of three position and three rotation parameters (rigid body coordinates).

The position and orientation as well as the deformation in deformable bodies can be defined by a **set of coordinates**. We require that any set of coordinates uniquely defines the position of all points of the bodies. As the bodies move, the coordinates vary with time. The set of n generalized coordinates is commonly represented by a (column) vector

$$\mathbf{q} = [q_0, q_1, \ldots, q_{n-1}]^{\text{T}}, \tag{5.7}$$

in which n represents the total number of coordinates that are used. In Exudyn, position, orientation and deformation coordinates (of bodies) are denoted as ODE2 coordinates, as they are related to second order differential equations, the main criterion to distinguish computational coordinates in the system.

The **generalized (Lagrangian) coordinates**, which are employed in Lagrange's equations of motion, are another set of well known coordinates used for mechanisms. As known from Lagrange's formalism, coordinates may be defined relative to each other.

5.1.11 Reference and current coordinates

An important fact on the coordinates used in Exudyn is upon the **additive**[1] splitting of quantities (e.g. position, rotation parameters, etc.) into **reference** and **current** (initial/visualization/...) coordinates.

[1]This additive splitting is also used for rotations: therefore, only the sum of reference and current (or visualization) coordinates has a geometrical meaning, while the parts are only used within the solver for incrementing.

The current position vector of a point node is computed from the reference position plus the current displacement, reading

$$\mathbf{p}_{\text{cur}} = \mathbf{p}_{\text{ref}} + \mathbf{u}_{\text{cur}} \tag{5.8}$$

In the same way rotation parameters are computed from,

$$\boldsymbol{\theta}_{\text{cur}} = \boldsymbol{\theta}_{\text{ref}} + \boldsymbol{\psi}_{\text{cur}} \tag{5.9}$$

which are based on reference quantities plus displacements or changes. Note that these changes are additive, even for rotation parameters. Needless to say, $\boldsymbol{\psi}_{\text{cur}}$ do not represent rotation parameters, while $\boldsymbol{\theta}_{\text{ref}}$ should be chosen such that they represent the orientation of a node in reference configuration. The necessity for reference coordinates originates from finite elements, which usually split nodal position into displacements and reference position. However, we also use the reference position here in order to define joints, e.g., using the utility function `CreateRevoluteJoint(...)`.

Note that this splitting is only employed for position coordinates, but not for velocities, accelerations or special coordinates. See also Section 2.2.6.

5.2 Dynamics: Mechanical principles

Following kinematics, dynamics comes into play, which involves the study of the forces and torques that cause the motion, bridging the gap between the motion observed and the reasons behind it. In this section, we shortly recap mechanical principles, which are used throughout for the simulation of multibody systems.

5.2.1 Newton's basic principles

In the study of multibody systems, Newton's basic principles lay the foundational framework. At the heart of these principles are three laws that govern the motion of bodies:
1. Newton's first law (law of inertia): A body remains at rest, or in uniform motion in a straight line, unless acted upon by a force $\mathbf{f}$, expressed as:

$$\mathbf{f} = 0 \implies \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} = 0 \tag{5.10}$$

where $\mathbf{f}$ is the net force applied to the body, and $\frac{d\mathbf{v}}{dt}$ is the acceleration of the body.
2. Newton's second law (law of motion): The sum of the forces F acting on a point mass equals the change in momentum of this mass,

$$\mathbf{f} = \frac{\mathrm{d}\,\mathbf{J}}{\mathrm{d}t} = \dot{\mathbf{J}} \tag{5.11}$$

where $\mathbf{J}$ is the (linear, translational) momentum of the mass m. Assuming a constant mass of the body (which is not true for some cases such as rockets), we find that the acceleration $\mathbf{a}$ of an object is directly proportional to the net force $\mathbf{f}$ acting upon it and inversely proportional to its mass, given by the equation:

$$\mathbf{f} = m\mathbf{a} \tag{5.12}$$

where m is the mass of the object.

3. Newton's third law (action and reaction): For every action, there is an equal and opposite reaction. This principle is fundamental in analyzing the interactions within multibody systems and can be summarized as:

$$\mathbf{f}_{12} = -\mathbf{f}_{21} \tag{5.13}$$

in which we observe $\mathbf{f}_{12}$ as the force exerted by body 1 on body 2, and $\mathbf{f}_{21}$ as the force exerted by body 2 on body 1.

These principles are pivotal in understanding and modeling the dynamics of multibody systems, providing the basis for further exploration and analysis in this field. Newton's principles represent linear (translational) motion, but may be transformed to rotations by using angular momentum, as well. This leads to Euler's equations, see the description of the `ObjectRigidBody`.

5.2.2 The Lagrange-d'Alembert principle

For a constant mass m, it follows from Newton's second law that

$$\sum \mathbf{f} = m\mathbf{a} \ , \tag{5.14}$$

Eq. (5.14) can also be written in the form

$$\sum \mathbf{f} - m\mathbf{a} = \mathbf{0} \tag{5.15}$$

The vector $(-m\mathbf{a})$ is referred to as the inertia force, since it can apparently be balanced to zero in the sense of a generalized equilibrium. This equilibrium is called dynamic equilibrium, in analogy to statics.

In **D'Alembert's Principle**, the inertia force $\mathbf{f}_i = -m\mathbf{a}$ is introduced, and the sum of the forces is written as

$$\mathbf{f}_e + \mathbf{f}_c + \mathbf{f}_i = \mathbf{0} \tag{5.16}$$

where applied forces $\mathbf{f}_e$ and constraint forces $\mathbf{f}_c$ are distinguished.

A key property of the constraint forces is that the virtual work done by constraint forces always vanishes,

$$\mathbf{f}_c \cdot \delta\mathbf{r} = 0 \ . \tag{5.17}$$

where here $\delta\mathbf{r}$ is the virtual displacement associated with the constraint force.

5.2.3 Generalized Principle of Virtual Work

With the generalized principle of virtual work, it follows

$$(\mathbf{f}_e + \mathbf{f}_i) \cdot \delta\mathbf{r} = 0 \quad \text{or} \tag{5.18}$$

Thus, we obtain the **Lagrange-d'Alembert** principle as

$$\delta W_e + \delta W_T = 0 \quad \text{or} \tag{5.19}$$

with the virtual work W_e of applied forces and W_i of inertia forces,

$$\delta W_e = \mathbf{f}_e \cdot \delta\mathbf{r} \quad \text{and} \quad \delta W_T = \mathbf{f}_i \cdot \delta\mathbf{r} \ . \tag{5.20}$$

For a system of n mass points, it follows

$$\sum_{i=0}^{n-1} (\mathbf{f}_{e,i} \cdot \delta\mathbf{r}_i - m_i \mathbf{a}_i \cdot \delta\mathbf{r}_i) = 0 \tag{5.21}$$

It is ultimately characterized by the fact that, in comparison to D'Alembert's principle, constraint forces do not appear.

Note that this principle is used in `Exudyn` at many places to derive equations for rigid and flexible bodies, as well as for connectors, such as spring-dampers.

5.2.4 Virtual displacements

The virtual displacement $\delta\mathbf{r}_j$ or the variation of any quantity can be written in terms of variation of the underlying coordinates, see Section 5.1.10,

$$\delta\mathbf{r}_j = \sum_{i=1}^{n} \frac{\partial \mathbf{r}_j}{\partial q_i} \delta q_i \ . \tag{5.22}$$

5.2.5 Generalized Forces

To derive the Lagrangian equations, we first consider the virtual work done by N forces on N mass points

$$\delta W = \sum_{j=1}^{N} \mathbf{f}_j \cdot \delta\mathbf{r}_j \ . \tag{5.23}$$

Here, $\mathbf{f}_j$ represents the force on, and $\delta\mathbf{r}_j$ represents the displacement of, the jth mass point.

Using Eq. (5.22), the virtual work can be expressed as

$$\delta W = \sum_{j=1}^{N} \mathbf{f}_j \cdot \sum_{i=1}^{n} \frac{\partial \mathbf{r}_j}{\partial q_i} \delta q_i = \sum_{i=1}^{n} \left(\sum_{j=1}^{N} \mathbf{f}_j \cdot \frac{\partial \mathbf{r}_j}{\partial q_i} \right) \delta q_i \ . \tag{5.24}$$

In the process of translating the virtual work of forces or moments, we identify what are called generalized forces Q_i,

$$Q_i = \sum_{j=1}^{N} \mathbf{f}_j \cdot \frac{\partial \mathbf{r}_j}{\partial q_i} \ . \tag{5.25}$$

Consequently, the virtual work can also be written as,

$$\delta W = \sum_{i=1}^{n} Q_i \, \delta q_i \ . \tag{5.26}$$

5.2.6 Lagrange's Equations of Motion

We define the kinetic energy as T, which for N mass points is given by

$$T = \frac{1}{2} \sum_{j=1}^{N} m_j \mathbf{v}_j^2 \tag{5.27}$$

Using the kinetic energy T, the following **Lagrangian equations** can be written,

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial T}{\partial \dot{q}_i} - \frac{\partial T}{\partial q_i} = Q_i, \quad i = 1, 2, \ldots, n \,. \tag{5.28}$$

These are valid under the assumption of minimal coordinates with holonomic constraints, resulting in the fact that all virtual displacements δq_i are independent of each other.

For forces that possess a potential V, i.e. the variation of the potential reads

$$\delta V = \sum_{i=1}^{n} \frac{\partial V}{\partial q_i}\delta q_i = -\sum_{i=1}^{n} Q_i^{(V)}\delta q_i \,, \tag{5.29}$$

and assuming that Q_i now only includes forces **without a potential**, the Lagrange equations can be rewritten as follows,

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial T}{\partial \dot{q}_i} - \frac{\partial T}{\partial q_i} = Q_i + Q_i^{(V)} = Q_i - \frac{\partial V}{\partial q_i}, \quad i = 1, 2, \ldots, n \,. \tag{5.30}$$

With the **definition** $L = T - V$ and the relationship $\frac{\partial V}{\partial \dot{q}_i} = 0$, a common form of Lagrange's equations is obtained:

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} = Q_i, \quad i = 1, 2, \ldots, n \,. \tag{5.31}$$

This equation is also used sometimes to derive equations of motion for rigid or flexible bodies in `Exudyn`.

5.2.7 Multibody formulations: redundant and minimal coordinates

In multibody system dynamics, we primarily distinguish between two formulations:

- **redundant coordinates**
- **minimal coordinates**

Formulations based on minimal coordinates appeared naturally and early, in principle every model in dynamics, such as a 1D motion of a mass point according to Newton's laws, see Eq. (5.12), Euler's equations, or a single DOF mass-spring-damper model. The **advantages of a minimal-coordinates** formulation are:

- coordinates represent the degrees of freedom
- direct access to relevant coordinates
- direct application of forces to the coordinates
- application of any class of solvers, as long as equations are non-stiff

The **disadvantages** are related to the **additional efforts to derive the equations of motion**, which has to be done for every different system, and the general restriction to open-loop systems, making it difficult to be applied to closed-loop systems (but not always impossible). There are many modifications, which allow closed-loop systems, e.g., via augmented Lagrangians or similar approaches.
A **redundant-coordinates** formulation has the following **advantages**:

- being **extremely versatile** and allowing practically any combination of (closed-loop) constraints
- easy extension to flexible bodies, sliding constraints, etc.
- multibody systems can be created easily by **combining a set of bodies and constraints** (→ user friendly)

Disadvantages are clearly the **resulting index 3** (position-level) constraints, for which only very few implicit solvers (time integration) exist, and it requires always matrix factorization during solving. A further problem is that this formulation may potentially lead to redundant constraints, which need to be treated specially with solvers, and that the degrees of freedom are less obvious and that relevant **(e.g., joint) coordinates are not directly accessible** on the equations level.
It is left to the user, which formulation to chose when working with multibody systems. In `Exudyn`, there is the option to create tree-like rigid body systems using the `ObjectKinematicTree`, which allows to use minimal coordinates for open-loop systems. In general, `Exudyn` is a redundant multibody dynamics formulation for reasons of simpler assembly of equations of motion.
As an example, we investigate the equations of motion for a **mathematical pendulum** with mass m, distance L between support and mass, under gravity g.
In the minimal coordinates formulation, see Fig. 5.3, we define the angle φ, being zero in the horizontal configuration,

$$mL\ddot{\varphi} = mg\cos\varphi \tag{5.32}$$

leading to one 2nd order ordinary differential equation (ODE2), using the minimal coordinate φ.

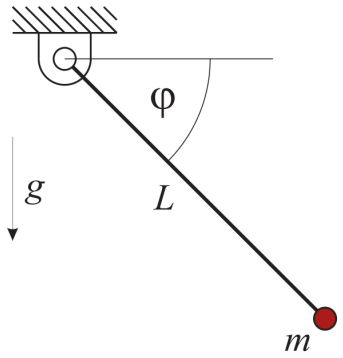


Figure 5.3: Mathematical pendulum with minimal coordinate φ.

With redundant coordinates, see Fig. 5.4, we may introduce two Cartesian coordinates (x, y), to define the location of the mass in the plane, leading to two differential equations for the mass point,

$$m\ddot{x} = f_x, \quad m\ddot{y} = f_y \tag{5.33}$$

together with a constraint equation

$$c(x, y) = \sqrt{x^2 + y^2} - L = 0 \tag{5.34}$$

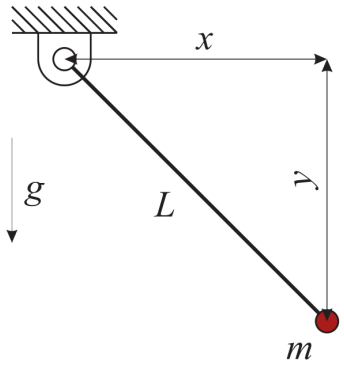


Figure 5.4: Mathematical pendulum with redundant coordinates (x, y).

Note that Eq. (5.33) and Eq. (5.34) include 3 equations for only two unknowns (x, y). Therefore, we need to introduce an additional unknown Lagrange multiplier λ for the redundant multibody formulation, which represents a force in direction of the pendulum's string, exactly such that the length L is preserved. The factor λ has to be projected with the constraint derivatives (Jacobian) onto the coordinates (x, y), giving the forces

$$f_x = -\frac{\partial c}{\partial x}\lambda, \quad f_y = mg - \frac{\partial c}{\partial y}\lambda \; . \tag{5.35}$$

Therefore, we can rewrite Eq. (5.33) as

$$m\ddot{x} + \frac{\partial c}{\partial x}\lambda = 0, \quad m\ddot{y} + \frac{\partial c}{\partial y}\lambda = mg \tag{5.36}$$

with the algebraic constraint, written in a computationally more efficient form,

$$c(x, y) = x^2 + y^2 - L^2 = 0 \tag{5.37}$$

The equations of motion are formed by both Eq. (5.36) and Eq. (5.37), which are **differential-algebraic equations (DAEs) of index 3** and therefore much harder to solve than ordinary differential equations in case of minimal coordinates.
In `Exudyn`, we therefore use either the generalized-α solver, or an implicit trapezoidal rule in combination with an index 2 reduction, see the section on solvers for more details.

5.3 Frames, rotations and coordinate systems

In this section, we introduce frames, which provide reference points and rotations attached to the ground or to bodies, thus providing a bases for coordinate systems.

5.3.1 Reference points and reference frames

In contrast to the reference coordinates (which may include reference position and rotations), the reference point of a body, marker or joint is available in any configuration (current, initial, etc.). The

reference point and orientation (or reference frame) of a rigid body coincides with the position and orientation of the underlying node. The local position of a point of the body is expressed relative to the body's reference frame, which may be different from the body's center of mass. The reference frame is also used for FFRF objects. In most cases, reference points are denoted by $\mathbf{r}$.

5.3.2 Coordinate systems of frames

Many considerations and computations in kinematics can be formulated coordinate-free. However, ultimately local positions are given by the user in different coordinates from results given in global (world) coordinates.

For this reason, we use left upper indices in symbols to indicate underlying frames of points or vectors, e.g., ${}^{0}\mathbf{u}$ represents a displacement vector in the global (world) coordinate system 0, while ${}^{m1}\mathbf{u}$ represents the displacement vector in marker 1 ($m1$) coordinates.

Typical coordinate systems (frames) are:

- ${}^{0}\mathbf{u}$ ... global or world coordinates
- ${}^{b}\mathbf{u}$ ... body-fixed, local coordinates
- ${}^{m0}\mathbf{u}$ ... local coordinates of (the body or node of) marker $m0$
- ${}^{m1}\mathbf{u}$ ... local coordinates of (the body or node of) marker $m1$
- ${}^{J0}\mathbf{u}$ ... local coordinates of joint 0, related to marker $m0$
- ${}^{J1}\mathbf{u}$ ... local coordinates of joint 1, related to marker $m1$

To transform the local coordinates ${}^{m0}\mathbf{u}$ of marker 0 into global coordinates ${}^{0}\mathbf{x}$, we use the relation

$$ {}^{0}\mathbf{u} = {}^{0,m0}\mathbf{A}\ {}^{m0}\mathbf{u} \tag{5.38} $$

in which ${}^{0,m0}\mathbf{A}$ is the transformation matrix of (the body or node of) the underlying marker 0.

In particular, note that any transformation, e.g., of the angular velocity in body (b) and world coordinates,

$$ {}^{0}\boldsymbol{\omega} = {}^{0,b}\mathbf{A}\ {}^{b}\boldsymbol{\omega} \tag{5.39} $$

only transforms the vector from one frame to the other, but does not change the vector $\boldsymbol{\omega}$ itself.

5.3.3 Homogeneous transformations

In order to considere rotations and translations in frames, homogeneous transformations are used. An arbitrary vector ${}^{i}\mathbf{r}$ in coordinates of frame $\mathcal{F}_i$ can be expressed relative to $\mathcal{F}_j$ with the vector ${}^{j}\mathbf{p}_i$, pointing from O_j to O_i and a rotation matrix according to Eq. (5.38):

$$ {}^{j}\mathbf{r} = {}^{ji}\mathbf{R}\ {}^{i}\mathbf{r} + {}^{j}\mathbf{p}_i \tag{5.40} $$

Eq. (5.40) can be written as a linear mapping,

$$ \begin{bmatrix} {}^{j}\mathbf{r} \\ 1 \end{bmatrix} = \begin{bmatrix} {}^{ji}\mathbf{R} & {}^{j}\mathbf{p}_i \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} {}^{i}\mathbf{r} \\ 1 \end{bmatrix}, \tag{5.41} $$

in which the matrix

$$ {}^{ji}\mathbf{T} = \begin{bmatrix} {}^{ji}\mathbf{R} & {}^{j}\mathbf{p}_i \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{5.42} $$

is referred to as the **homogeneous transformation matrix**, and ${}^{j}\hat{\mathbf{r}} = [{}^{j}\mathbf{r} \quad 1]^T$ is the homogeneous vector corresponding to ${}^{j}\mathbf{r}$.

Analogous to the rotation matrix, ${}^{ji}\mathbf{T}$ has an inverse, which follows as

$$ {}^{ji}\mathbf{T}^{-1} = {}^{ij}\mathbf{T} = \begin{bmatrix} {}^{ji}\mathbf{R}^T & -{}^{ji}\mathbf{R}^T\,{}^{j}\mathbf{p}_i \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} {}^{j}\mathbf{r} \\ 1 \end{bmatrix} . \tag{5.43} $$

Sequential application of homogeneous transformations simplifies to

$$ {}^{ki}\mathbf{T} = {}^{kj}\mathbf{T}\;{}^{ji}\mathbf{T} . \tag{5.44} $$

We could also actively move vectors, by assuming that the homogeneous transformation matrix ${}^{i0,i1}\mathbf{T}$ transforms coordinates within the same frame $\mathcal{F}_i$, between two (time) steps 0 and 1,

$$ {}^{i1}\hat{\mathbf{r}} = {}^{i1,i0}\mathbf{T}\;{}^{i0}\hat{\mathbf{r}} \tag{5.45} $$

which advances the vector ${}^{i0}\hat{\mathbf{r}}$ in time.

Note that an efficient implementation would only include $\mathbf{R}$ and $\mathbf{p}$, without necessarily computing 4×4 matrices.

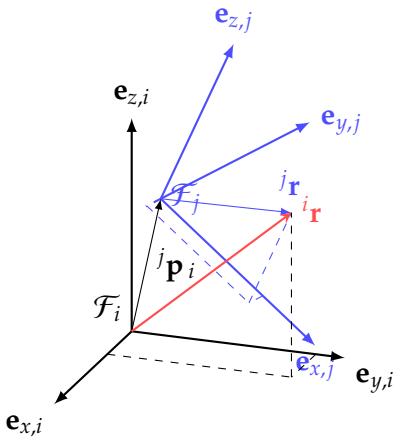


Figure 5.5: Homogeneous transformation from $\mathcal{F}_i$ to $\mathcal{F}_j$ using the relation ${}^{j}\hat{\mathbf{r}} = {}^{ji}\mathbf{T}\;{}^{i}\hat{\mathbf{r}}$.

All homogeneous transformations form the **Special Euclidean Group SE(3)** – the motion group – for describing rigid body movements in 3D. Therefore, homogeneous transformations also meet the criteria for groups (closure, associativity, existence of neutral and inverse elements).

Elementary rotations about the $\mathbf{e}_z$-axis and translations along the $\mathbf{e}_x$-axis, for example, are given by:

$$ \mathbf{Rot}(\mathbf{e}_z, \theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{Trans}(\mathbf{e}_x, d) = \begin{bmatrix} 1 & 0 & 0 & d \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.46} $$

Arbitrary composite transformations can be constructed from elementary transformations. In general, the following applies:

- Pre-multiplication (**Pre-Multiplication**): Transformation in the global coordinate system
- Post-multiplication (**Post-Multiplication**): Transformation in the rotated coordinate system

By exploiting the tools of so-called Lie groups, rigid body movements can be interpolated using matrix logarithm (`LogSE3(...)`) and matrix exponential function (`ExpSE3(...)`).

5.3.4 Rotations

In this section, we discuss in particular rotations, as already introduced for coordinate systems and homogeneous transformations. Rotations can be represented by transformation matrices, leading to a linear transformation

$$ {}^{0}\mathbf{r} = {}^{0,1}\mathbf{A}\ {}^{1}\mathbf{r} \tag{5.47} $$

which transforms the vector ${}^{1}\mathbf{r}$ accordingly. However, rotations are inherently nonlinear. First, rotation parametrizations – except for the components of the rotation matrix itself – are highly nonlinear functions of rotation parameters. Second, subsequent rotations couple in a multiplicative (i.e., nonlinear) way. Therefore rotations have to be treated differently from translations.

5.3.4.1 Derivation of the transformation matrix from coordinate transformations

The vector $\mathbf{a}$ can be represented in coordinates of frame $\mathcal{F}_1$ with basis vectors ($\mathbf{e}_{x1}$, $\mathbf{e}_{y1}$, $\mathbf{e}_{z1}$) and of frame $\mathcal{F}_2$ with basis vectors ($\mathbf{e}_{x2}$, $\mathbf{e}_{y2}$, $\mathbf{e}_{z2}$),

$$ \mathbf{a} = {}^{1}a_x\,\mathbf{e}_{x1} + {}^{1}a_y\,\mathbf{e}_{y1} + {}^{1}a_z\,\mathbf{e}_{z1} = {}^{2}a_x\,\mathbf{e}_{x2} + {}^{2}a_y\,\mathbf{e}_{y2} + {}^{2}a_z\,\mathbf{e}_{z2}\ . \tag{5.48} $$

Multiplying Eq. (5.48) with the basis vectors ($\mathbf{e}_{x1}$, $\mathbf{e}_{y1}$, $\mathbf{e}_{z1}$) consecutively, we obtain three relations,

$$ \begin{aligned} {}^{1}a_x &= {}^{2}a_x\,\mathbf{e}_{x1}^T\mathbf{e}_{x2} + {}^{2}a_y\,\mathbf{e}_{x1}^T\mathbf{e}_{y2} + {}^{2}a_z\,\mathbf{e}_{x1}^T\mathbf{e}_{z2}\ , \\ {}^{1}a_y &= {}^{2}a_x\,\mathbf{e}_{y1}^T\mathbf{e}_{x2} + {}^{2}a_y\,\mathbf{e}_{y1}^T\mathbf{e}_{y2} + {}^{2}a_z\,\mathbf{e}_{y1}^T\mathbf{e}_{z2}\ , \\ {}^{1}a_z &= {}^{2}a_x\,\mathbf{e}_{z1}^T\mathbf{e}_{x2} + {}^{2}a_y\,\mathbf{e}_{z1}^T\mathbf{e}_{y2} + {}^{2}a_z\,\mathbf{e}_{z1}^T\mathbf{e}_{z2}\ . \end{aligned} \tag{5.49} $$

where we apply the orthonormality relationships $\mathbf{e}_{x1}^T\mathbf{e}_{x1} = 1, \ldots$ as well as $\mathbf{e}_{x1}^T\mathbf{e}_{y1} = 0$, $\mathbf{e}_{x1}^T\mathbf{e}_{z1} = 0, \ldots$ througout.

We can represent the result as linear transformation

$$ \begin{bmatrix} {}^{1}a_x \\ {}^{1}a_y \\ {}^{1}a_z \end{bmatrix} = \begin{bmatrix} \mathbf{e}_{x1}^T\mathbf{e}_{x2} & \mathbf{e}_{x1}^T\mathbf{e}_{y2} & \mathbf{e}_{x1}^T\mathbf{e}_{z2} \\ \mathbf{e}_{y1}^T\mathbf{e}_{x2} & \mathbf{e}_{y1}^T\mathbf{e}_{y2} & \mathbf{e}_{y1}^T\mathbf{e}_{z2} \\ \mathbf{e}_{z1}^T\mathbf{e}_{x2} & \mathbf{e}_{z1}^T\mathbf{e}_{y2} & \mathbf{e}_{z1}^T\mathbf{e}_{z2} \end{bmatrix} \begin{bmatrix} {}^{2}a_x \\ {}^{2}a_y \\ {}^{2}a_z \end{bmatrix} \tag{5.50} $$

or in short

$$ {}^{1}\mathbf{a} = {}^{12}\mathbf{A}\ {}^{2}\mathbf{a}\ . \tag{5.51} $$

The **Transformationsmatrix** ${}^{12}\mathbf{A}$ transforms coordinates of vector **a** from $\mathcal{F}_2$ into $\mathcal{F}_1$. We observe that the columns of ${}^{12}\mathbf{A}$ contain unit vectors $\mathbf{e}_{x2}$, $\mathbf{e}_{y2}$, and $\mathbf{e}_{z2}$ represented in frame $\mathcal{F}_1$, and that the rows can be identified as the unit vectors $\mathbf{e}_{x1}^T$, $\mathbf{e}_{y1}^T$, and $\mathbf{e}_{z1}^T$ represented in frame $\mathcal{F}_2$:

$$ {}^{12}\mathbf{A} = \begin{bmatrix} {}^1\mathbf{e}_{x2} & {}^1\mathbf{e}_{y2} & {}^1\mathbf{e}_{z2} \end{bmatrix} = \begin{bmatrix} {}^2\mathbf{e}_{x1}^{\mathrm{T}} \\ {}^2\mathbf{e}_{y1}^{\mathrm{T}} \\ {}^2\mathbf{e}_{z1}^{\mathrm{T}} \end{bmatrix} . \tag{5.52} $$

Note that:

- Columns and rows of ${}^{12}\mathbf{A}$ represent an orthonormal right-hand system.
- Because the determinant $\det({}^{12}\mathbf{A}) = +1$, we call ${}^{12}\mathbf{A}$ to be proper-orthogonal.
- Rotations conserve length, $|\mathbf{Ax}| = |\mathbf{x}|$.
- For the same reason, rotations conserve angles.
- During transformations, we observe that reading left upper indices from left to right, indices are changed only by transformation matrices (which are not transposed): ${}^2\mathbf{a} = {}^{21}\mathbf{A}\ {}^1\mathbf{a}$

We note the following rules:

$$ \begin{aligned} {}^{21}\mathbf{A} &= {}^{12}\mathbf{A}^{-1} , \\ {}^{12}\mathbf{A}^{-1} &= {}^{12}\mathbf{A}^{T} , \\ {}^{12}\mathbf{A}\ {}^{12}\mathbf{A}^{T} &= \mathbf{I} , \\ {}^{12}\mathbf{A}^{T}\ {}^{12}\mathbf{A} &= \mathbf{I} . \end{aligned} \tag{5.53} $$

5.3.4.2 Elementary rotations

Elementary rotations are rotations about a single axis, using one of the orthogonal basis vectors. Consider basis ($\mathbf{e}_{x1}$, $\mathbf{e}_{y1}$, $\mathbf{e}_{z1}$) and a rotation around axis $\mathbf{e}_{x1}$ with angle φ_1 in positive rotation sense, obtaining a rotated frame ($\mathbf{e}_{x2}$, $\mathbf{e}_{y2}$, $\mathbf{e}_{z2}$), see Fig. 5.6. The rotation matrix for this case reads:

$$ {}^{12}\mathbf{A} = \begin{bmatrix} {}^1\mathbf{e}_{x2} & {}^1\mathbf{e}_{y2} & {}^1\mathbf{e}_{z2} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\varphi_1 & -s\varphi_1 \\ 0 & s\varphi_1 & c\varphi_1 \end{bmatrix} . \tag{5.54} $$

In this case, the coordinates of a vector **r** are transformed according to

$$ {}^1\mathbf{r} = {}^{12}\mathbf{A}\ {}^2\mathbf{r} \tag{5.55} $$

In a second example, a rotation with angle φ_2 around $\mathbf{e}_{y2}$ is performed to transform from basis ($\mathbf{e}_{x2}$, $\mathbf{e}_{y2}$, $\mathbf{e}_{z2}$) into ($\mathbf{e}_{x3}$, $\mathbf{e}_{y3}$, $\mathbf{e}_{z3}$), see Fig. 5.7. The rotation matrix for this case reads:

$$ {}^{23}\mathbf{A} = \begin{bmatrix} {}^2\mathbf{e}_{x3} & {}^2\mathbf{e}_{y3} & {}^2\mathbf{e}_{z3} \end{bmatrix} = \begin{bmatrix} c\varphi_2 & 0 & s\varphi_2 \\ 0 & 1 & 0 \\ -s\varphi_2 & 0 & c\varphi_2 \end{bmatrix} . \tag{5.56} $$

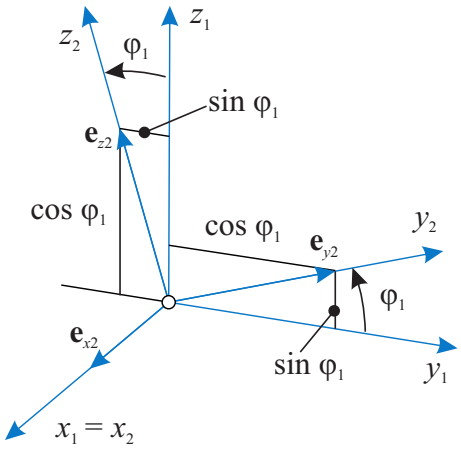


Figure 5.6: Elementary rotation around axis $\mathbf{x}_1$.

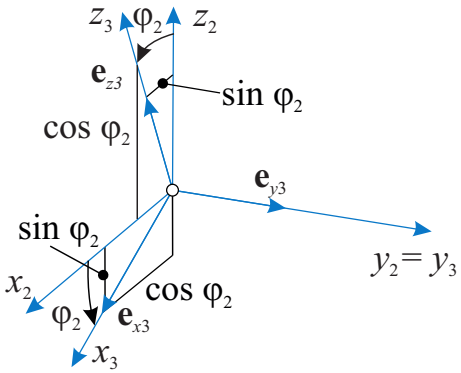


Figure 5.7: Elementary rotation around axis $\mathbf{y}_2$.

Finally, a rotation around $\mathbf{e}_{z3}$ with angle φ_3 would give similarly,

$$ {}^{34}\mathbf{A} = \begin{bmatrix} {}^{3}\mathbf{e}_{x4} & {}^{3}\mathbf{e}_{y4} & {}^{3}\mathbf{e}_{z4} \end{bmatrix} = \begin{bmatrix} c\varphi_3 & -s\varphi_3 & 0 \\ s\varphi_3 & c\varphi_3 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{5.57} $$

5.3.4.3 Linearized rotations

In this case we can interpret the small rotations as a constant angular velocity over small time Δt, $\boldsymbol{\varphi} = \Delta t \cdot \boldsymbol{\omega}$, which results in

$$ \mathbf{r}_1' = \mathbf{r}_1 + \Delta t \cdot (\boldsymbol{\omega} \times \mathbf{r}_1) = (\mathbf{I} + \tilde{\boldsymbol{\varphi}})\mathbf{r}_1 \tag{5.58} $$

Leading to the linearized rotation matrix

$$ \mathbf{A}_{\text{lin}} = \mathbf{I} - \tilde{\boldsymbol{\varphi}} \tag{5.59} $$

Alternatively, using linearized rotations $\boldsymbol{\varphi} = [\varphi_1,\ \varphi_2,\ \varphi_3]^{\mathrm{T}}$, with $|\boldsymbol{\varphi}| \ll 1$, we can approximate $\sin\varphi_1$ as:

$$ \sin\varphi_1 = \varphi_1 - \frac{\varphi_1^3}{3!} + \frac{\varphi_1^5}{5!} - \ldots \approx \varphi_1 \tag{5.60} $$

Similarly, we can approximate $\cos\varphi_1 \approx 1$. This immediately gives a linearization for elementary rotations – where we observe that contributions for each axis can be added.

The axis-angle representation (see later) leads to the same result by linearization of the Rodrigues formula,

$$ \mathbf{A}(\mathbf{u}, \varphi) \approx \mathbf{I} + \tilde{\mathbf{u}}\varphi, \quad \mathbf{A}^{\mathrm{T}} \approx \mathbf{I} - \tilde{\mathbf{u}}\varphi \tag{5.61} $$

in which φ represents the infinitesimal angle and $\mathbf{u}$ is the rotation axis.

Note: whenever you would like to work with linearized rotations, or if you do not know the sequence of small rotations, still do not use the linearized formula as it gives immediately rotation matrices without strict orthogonality and determinant of 1. In order to avoid computational problems, use for example `RotationVector2RotationMatrix(phi)` to compute a consistent rotation matrix based on the matrix exponential.

5.3.4.4 Active and passive rotations

It is important to distinguish two fundamentally different concepts of rotations. There is the **active rotation**, which can be represented by a coordinate-free relation,

$$ \mathbf{v}' = \mathbf{A}\mathbf{v} \tag{5.62} $$

in which a vector $\mathbf{v}$ is actively rotated into a new configuration $\mathbf{v}'$ using a rotation tensor $\mathbf{A}$. Eq. (5.62) can be represented in any coordinate system, but keeping it fixed.

Furthermore, we denote a **passive rotation** as a coordinate transformation, as used many times before,

$$ {}^{1}\mathbf{v} = {}^{12}\mathbf{A}\ {}^{2}\mathbf{v} \tag{5.63} $$

in which the vector $\mathbf{v}$ is not changed at all, but only its coordinates are represented in two different coordinate systems.

While passive rotations (and homogeneous transformations) are used, e.g., to compute current positions of body-attached points through several relative coordinate systems, active rotations can represent the rotations from one time step to the next one.

5.3.4.5 Successive rotations

The successive application of rotation matrices is non-commutative. An exception is the planar case, i.e., all rotations occur around the same axis. Specifically, we see that for successive rotations, the following must be considered:

$$\mathbf{A}_2\mathbf{A}_1\mathbf{v} \neq \mathbf{A}_1\mathbf{A}_2\mathbf{v} \; . \tag{5.64}$$

As an example, we consider in Figure 5.8 the different order of rotations of a block.

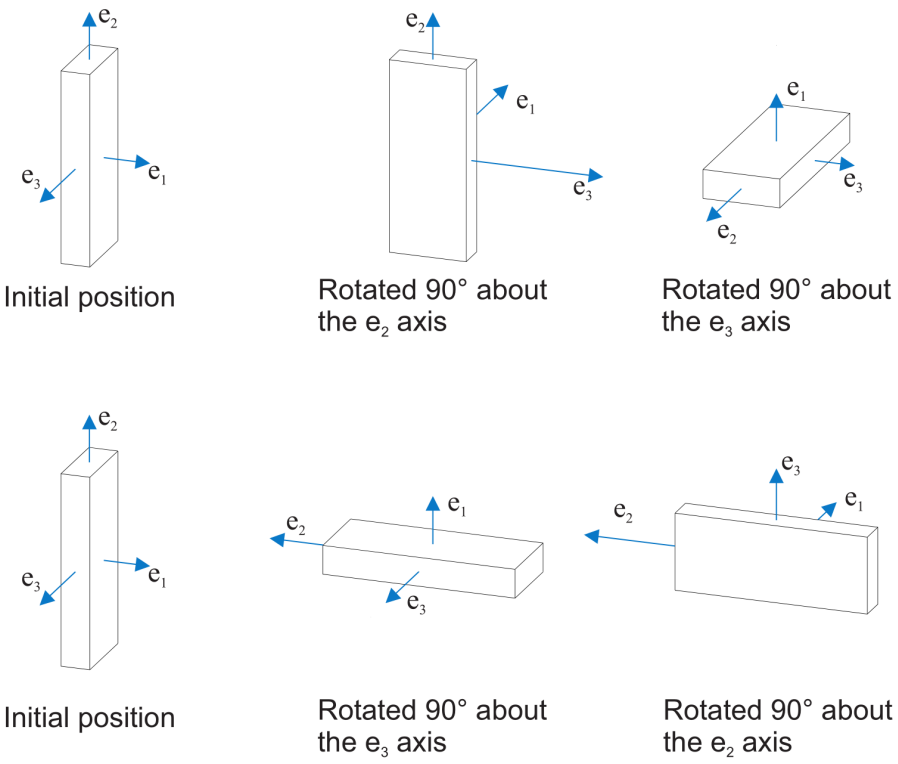


Figure 5.8: Successive rotations are not commutative.

In Fig. 5.8a, the block shown is first rotated 90°about the $\mathbf{e}_2$ axis and then 90°about the $\mathbf{e}_3$ axis. In Figure 5.8b, the same rotations are applied in reverse order, i.e., the block is first rotated 90°about the $\mathbf{e}_3$ axis and then 90°about the $\mathbf{e}_2$ axis. It is immediately apparent that the resulting orientation of the block is different in both cases.

Finally, it should be noted that there are two different types of successive rotations. In the first variant, also known as the **single-frame method**, the same reference frame is chosen for each rotation.

This variant is best understood in the active rotation of vectors, where these rotations always take place, for example, in the global reference system.

In the second variant, also known as the **multi-frame method**, a new reference frame created by the preceding rotation is used for each rotation. This variant can be illustrated by applications in robotics (e.g., articulated arm robots), where (passive) rotations of the reference systems of the robot's respective arms occur, with each rotation taking place in the reference system of the preceding arm.

5.3.5 Rotation tensor: axis-angle representation

In the following, we will elaborate the rotation tensor, inherently linked to the axis-angle representation. Note that the rotation axis times the angle is denoted as rotation vector throughout.

Considering Euler's theorem on rotations, we assume a transformation

$$\mathbf{r}_0 \rightarrow \mathbf{r}(t) \tag{5.65}$$

which purely follows from a rotation. This transformation thus can be represented by a rotation axis $\mathbf{u}$ and an angle $\varphi(t)$,

$$(\mathbf{u}(t),\, \varphi(t)) \tag{5.66}$$

both of them given in a coordinate-free manner. We may therefore conclude

$$\mathbf{r}(t) = \mathbf{A}(t)\,\mathbf{r}_0 \qquad \text{with} \qquad \mathbf{A}(t) = \mathbf{A}(\mathbf{u}(t), \varphi(t)) \tag{5.67}$$

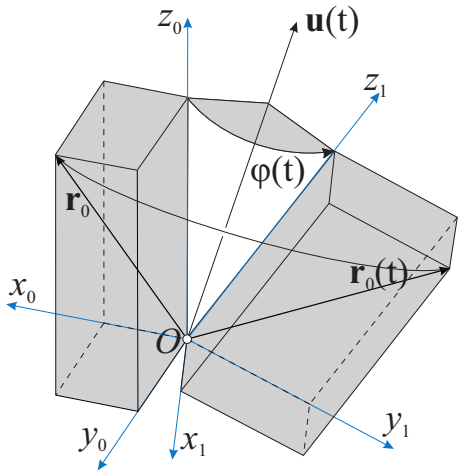


Figure 5.9: Rotation of a vector $\mathbf{r}_0$ by means of the angle-axis tuple $(\mathbf{u}(t),\, \varphi(t))$.

Using Fig. 5.9, we may now consider relations of the two frames $(\mathbf{e}_{x0},\, \mathbf{e}_{y0},\, \mathbf{e}_{z0})$ and $(\mathbf{e}_{x1},\, \mathbf{e}_{y1},\, \mathbf{e}_{z1})$, solely defined by the angle-axis $(\mathbf{u}(t),\, \varphi(t))$ relation.

According to Fig. 5.10, we may split the vector $\mathbf{r}$, which has the length $r = |\mathbf{r}|$, into

$$\mathbf{r} = c_1\mathbf{e}_1 + c_2\mathbf{e}_2 + c_3\mathbf{e}_3 \tag{5.68}$$

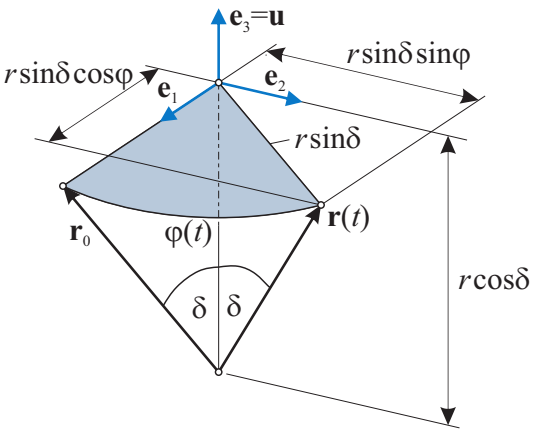


Figure 5.10: Relations for derivation of rotation tensor and Rodrigues' formula.

with the unit vectors given as

$$\mathbf{e}_1 = \frac{\mathbf{r}_0 - \mathbf{u}(\mathbf{u}^{\mathrm{T}}\mathbf{r}_0)}{r\sin\delta}, \quad \mathbf{e}_2 = \frac{\tilde{\mathbf{u}}\mathbf{r}_0}{r\sin\delta}, \quad \text{and} \quad \mathbf{e}_3 = \mathbf{u}\,. \tag{5.69}$$

The coefficients can be calculated as follows,

$$c_1 = r\sin\delta\cos\varphi, \quad c_2 = r\sin\delta\sin\varphi, \quad \text{and} \quad c_3 = r\cos\delta \tag{5.70}$$

Putting everything together gives

$$\mathbf{r} = \cos\varphi\,\mathbf{r}_0 + \sin\varphi\,\tilde{\mathbf{u}}\,\mathbf{r}_0 + (1-\cos\varphi)\,\mathbf{u}\,(\mathbf{u}^{\mathrm{T}}\,\mathbf{r}_0) \tag{5.71}$$

From the relation $\mathbf{r} = \mathbf{A}(\mathbf{u},\varphi)\mathbf{r}_0$, we can reinterpret the **rotation tensor A**

$$\mathbf{A}(\mathbf{u},\varphi) = \cos\varphi\,\mathbf{I} + \sin\varphi\,\tilde{\mathbf{u}} + (1-\cos\varphi)\,\mathbf{u}\,\mathbf{u}^{\mathrm{T}} \tag{5.72}$$

With the additional relations

$$\mathbf{u}\,\mathbf{u}^{\mathrm{T}} = \tilde{\mathbf{u}}\,\tilde{\mathbf{u}} + (\mathbf{u}^{\mathrm{T}}\mathbf{u})\mathbf{I} \quad \text{and} \quad \mathbf{u}^{\mathrm{T}}\mathbf{u} = 1 \tag{5.73}$$

we finally obtain

$$\mathbf{A}(\mathbf{u},\varphi) = \mathbf{I} + \sin\varphi\,\tilde{\mathbf{u}} + (1-\cos\varphi)\,\tilde{\mathbf{u}}\,\tilde{\mathbf{u}} \tag{5.74}$$

which is also known as the **Rodrigues formula** for rotations. While this formula is coordinate-free, it may be represented in any coordinate system, such as

$${}^0\mathbf{r}\,(t) = {}^0\mathbf{A}\,(t)\;{}^0\mathbf{r}_0 \qquad \text{with} \qquad {}^0\mathbf{A}\,(t) = \mathbf{A}({}^0\mathbf{u}\,(t),\varphi(t)) \tag{5.75}$$

5.3.5.1 Rotation vector

Using the rotation vector $\mathbf{v}_{\mathrm{rot}} = \varphi\cdot\mathbf{u}$ and $\varphi = |\mathbf{v}_{\mathrm{rot}}|$, another representation of the rotation tensor follows as

$$\mathbf{A}(\mathbf{u},\varphi) = \mathbf{I} + \frac{\sin\varphi}{\varphi}\tilde{\mathbf{v}}_{\mathrm{rot}} + \frac{(1-\cos\varphi)}{\varphi^2}\,\tilde{\mathbf{v}}_{\mathrm{rot}}\,\tilde{\mathbf{v}}_{\mathrm{rot}} \tag{5.76}$$

Note, that we inherently assume that $\varphi \in [0,\pi]$. In `Exudyn`, there are the following **functions and items related to the rotation vector** and the axis-angle representation:

- `NodeRigidBodyRotVecLG`: A 3D rigid body node based on rotation vector and Lie group methods; can be used for explicit integration methods, not leading to singularities when integrating with Lie-group methods
- `RotationVector2RotationMatrix`: computes the rotation matrix from a given rotation vector $\mathbf{v}_{\text{rot}}$
- `RotationMatrix2RotationVector`: computes the rotation vector $\mathbf{v}_{\text{rot}}$ from a given rotation matrix, based on a reconstruction of Euler parameters
- `ComputeRotationAxisFromRotationVector`: computes the rotation axis $\mathbf{u}$ from the rotation vector by using $\varphi = |\mathbf{v}_{\text{rot}}|$

Note the following **properties of the rotation tensor**:

- The axis-angle representations $(\mathbf{u}, \varphi)$ and $(-\mathbf{u}, -\varphi)$ represent the same tensor $\mathbf{A}(\mathbf{u}, \varphi) = \mathbf{A}(-\mathbf{u}, -\varphi)$
- The rotation tensor $\mathbf{A}$ is orthogonal, i.e., $\mathbf{A}^{\mathrm{T}}\,\mathbf{A} = \mathbf{E}$.
- The inverse rotation tensor $\mathbf{A}^{-1} = \mathbf{A}^{\mathrm{T}}$ is represented by the identical axis-angle tuples $(\mathbf{u}, -\varphi)$ and $(-\mathbf{u}, \varphi)$, leading to $\mathbf{r}_0 = \mathbf{A}(\mathbf{u}, -\varphi)\mathbf{r}$.
- We furthermore not the identities: $\mathbf{A}(\mathbf{u}, -\varphi) \equiv \mathbf{A}(-\mathbf{u}, \varphi) \equiv \mathbf{A}^{\mathrm{T}}(\mathbf{u}, \varphi)$
- The rotation axis is invariant to the rotation: $\mathbf{A}(\mathbf{u}, \varphi)\mathbf{u} = \mathbf{u}$

5.3.6 Euler angles: Tait-Bryan angles

In the following section, we discuss rotation matrices built by Tait-Bryan angles, which are consecutive xyz-rotations. Note that there are many other representations of Euler angles, however, they are not implemented or used in `Exudyn`. The output variable `Rotation` gives Tait-Bryan angles, which is why it is important to define their properties.

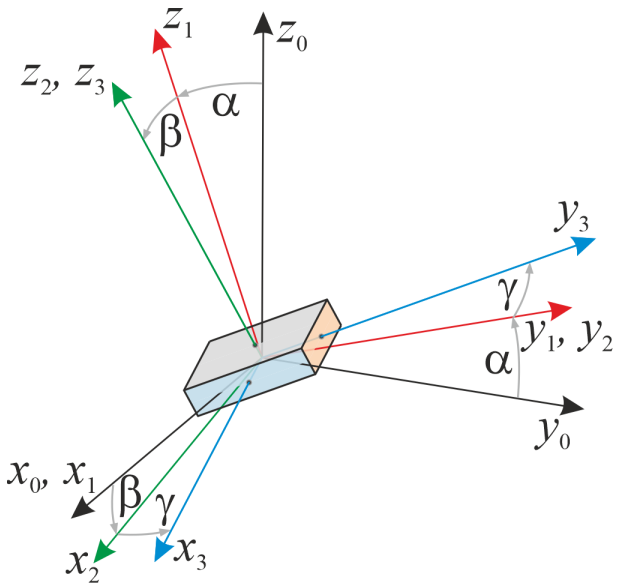


Figure 5.11: Definition of Tait-Bryan angles.

The transformation given by Tait-Bryan angles (α, β, γ) follows from three successive rotations,

$$ {}^0\mathbf{r} = {}^{01}\mathbf{A}(\alpha)\ {}^{12}\mathbf{A}(\beta)\ {}^{23}\mathbf{A}(\gamma)\ {}^3\mathbf{r} \quad \text{and} \quad {}^0\mathbf{r} = {}^{03}\mathbf{A}(\alpha,\beta,\gamma)\ {}^3\mathbf{r} \tag{5.77} $$

The **Tait-Bryan rotation matrix** is thus defined as

$$ {}^{03}\mathbf{A}(\alpha,\beta,\gamma) = \begin{bmatrix} \mathrm{c}\beta\,\mathrm{c}\gamma & -\mathrm{c}\beta\,\mathrm{s}\gamma & \mathrm{s}\beta \\ \mathrm{c}\alpha\,\mathrm{s}\gamma + \mathrm{s}\alpha\,\mathrm{s}\beta\,\mathrm{c}\gamma & \mathrm{c}\alpha\,\mathrm{c}\gamma - \mathrm{s}\alpha\,\mathrm{s}\beta\,\mathrm{s}\gamma & -\mathrm{s}\alpha\,\mathrm{c}\beta \\ \mathrm{s}\alpha\,\mathrm{s}\gamma - \mathrm{c}\alpha\,\mathrm{s}\beta\,\mathrm{c}\gamma & \mathrm{s}\alpha\,\mathrm{c}\gamma + \mathrm{c}\alpha\,\mathrm{s}\beta\,\mathrm{s}\gamma & \mathrm{c}\alpha\,\mathrm{c}\beta \end{bmatrix} \tag{5.78} $$

While not fully obvious from the latter equations, we note that there is a singularity at $|\beta| = \frac{\pi}{2}$, which causes any computations to stall whenever getting close enough to this value.
It is also possible to reconstruct Tait-Bryan angles from a given rotation matrix. Assume, we have the rotation matrix ${}^{03}\mathbf{A} = (A_{ij})$. There are two solutions for β which follow from

$$ \cos\beta = \pm\sqrt{1 - A_{13}^2}, \quad \text{and} \quad \sin\beta = A_{13} \tag{5.79} $$

For beta fulfilling $|\beta| \neq \frac{\pi}{2}$, we can uniquely compute γ and α,

$$ \cos\gamma = \frac{A_{11}}{\cos\beta}, \quad \sin\gamma = -\frac{A_{12}}{\cos\beta}, \quad \cos\alpha = \frac{A_{33}}{\cos\beta}, \quad \text{and} \quad \sin\alpha = -\frac{A_{23}}{\cos\beta}\,. \tag{5.80} $$

Note that improvements are possible using the atan2() function. We can finally resolve non-uniqueness by restricting β within the range $-\frac{\pi}{2} < \beta < \frac{\pi}{2}$.

5.3.6.1 Tait-Bryan angles: angular velocity vector

Angular velocities are essential for the formulation of equations of motion of rigid bodies, for constraints and for evaluation. Therefore, basic relations are shown for Tait-Bryan angles, in particular the velocity transformation matrix $\mathbf{G}_{TB}$.

The angular velocity vector ω can either be computed from the basic relation $\dot{\mathbf{A}} = \tilde{\omega}\mathbf{A}$, which however involves many trigonometric terms, or from partial angular velocities, being part of the co-rotated rotation axes. With the latter approach, we see that

$$ \boldsymbol{\omega}_{30} = \dot{\alpha}\mathbf{e}_{x0} + \dot{\beta}\mathbf{e}_{y1} + \dot{\gamma}\mathbf{e}_{z2} \quad \text{or} \quad \boldsymbol{\omega}_{30} = \underbrace{[\mathbf{e}_{x0} \quad \mathbf{e}_{y1} \quad \mathbf{e}_{z2}]}_{\mathbf{G}_{TB}} \begin{bmatrix} \dot{\alpha} \\ \dot{\beta} \\ \dot{\gamma} \end{bmatrix} \tag{5.81} $$

We can now evaluate this equation in frame $\mathcal{F}_0$ with $\boldsymbol{\beta} = [\alpha\ \beta\ \gamma]^{\mathrm{T}}$, written as

$$ {}^0\boldsymbol{\omega}_{30} = \underbrace{[{}^0\mathbf{e}_{x0} \quad {}^0\mathbf{e}_{y1} \quad {}^0\mathbf{e}_{z2}]}_{{}^0\mathbf{G}_{TB}} \begin{bmatrix} \dot{\alpha} \\ \dot{\beta} \\ \dot{\gamma} \end{bmatrix} = {}^0\mathbf{G}_{TB}(\boldsymbol{\beta})\dot{\boldsymbol{\beta}}\,. \tag{5.82} $$

Evaluating the consecutive rotations, we find the respective unit vectors as

$$ {}^0\mathbf{e}_{x0} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad {}^0\mathbf{e}_{y1} = \begin{bmatrix} 0 \\ \mathrm{c}\alpha \\ \mathrm{s}\alpha \end{bmatrix}, \quad \text{and} \quad {}^0\mathbf{e}_{z2} = \begin{bmatrix} \mathrm{s}\beta \\ -\mathrm{s}\alpha\mathrm{c}\beta \\ \mathrm{c}\alpha\mathrm{c}\beta \end{bmatrix}\,. \tag{5.83} $$

which results in the relations for the (global) angular velocity vector

$$\begin{bmatrix} {}^0\omega_{30x} \\ {}^0\omega_{30y} \\ {}^0\omega_{30z} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \mathrm{s}\beta \\ 0 & \mathrm{c}\alpha & -\mathrm{s}\alpha\mathrm{c}\beta \\ 0 & \mathrm{s}\alpha & \mathrm{c}\alpha\mathrm{c}\beta \end{bmatrix} \begin{bmatrix} \dot{\alpha} \\ \dot{\beta} \\ \dot{\gamma} \end{bmatrix} \tag{5.84}$$

The matrix ${}^0\mathbf{G}_{TB}$ is used to compute partial derivatives of the angular velocity vector w.r.t. rotations. However, it is also used to project torques and angular velocities into a space given by $\mathbf{G}_{TB}^{\mathrm{T}}$ (which is indeed not equivalent to $\mathbf{G}_{TB}^{-1}$, but gives a symmetric mass matrix – see the equations of motion for the rigid body).

The inverse relation between $\dot{\boldsymbol{\beta}}$ and ${}^0\boldsymbol{\omega}_{30}$ gives

$$\begin{bmatrix} \dot{\alpha} \\ \dot{\beta} \\ \dot{\gamma} \end{bmatrix} = \frac{1}{\mathrm{c}\beta} \begin{bmatrix} \mathrm{c}\beta & \mathrm{s}\alpha\,\mathrm{s}\beta & -\mathrm{c}\alpha\,\mathrm{s}\beta \\ 0 & \mathrm{c}\alpha\,\mathrm{c}\beta & \mathrm{s}\alpha\mathrm{c}\beta \\ 0 & -\mathrm{s}\alpha & \mathrm{c}\alpha \end{bmatrix} \begin{bmatrix} {}^0\omega_{30x} \\ {}^0\omega_{30y} \\ {}^0\omega_{30z} \end{bmatrix} \tag{5.85}$$

or in compact form

$$\dot{\boldsymbol{\beta}} = {}^0\mathbf{G}_{TB}^{-1}(\boldsymbol{\beta})\,{}^0\boldsymbol{\omega}_{30} \tag{5.86}$$

We again see that for the case $|\beta| = \pi/2$ the matrix${}^0\mathbf{G}_{TB}$ becomes singular, and Eq. (5.85) cannot be resolved for $\dot{\boldsymbol{\beta}}$!

Finally, we also provide the relations of $\boldsymbol{\omega}_{30}$ in body-fixed (local) coordinates $\mathcal{F}_3$, which are frequently used in the implementation,

$$\begin{bmatrix} {}^3\omega_{30x} \\ {}^3\omega_{30y} \\ {}^3\omega_{30z} \end{bmatrix} = \begin{bmatrix} \mathrm{c}\beta\,\mathrm{c}\gamma & \mathrm{s}\gamma & 0 \\ -\mathrm{c}\beta\,\mathrm{s}\gamma & \mathrm{c}\gamma & 0 \\ \mathrm{s}\beta & 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{\alpha} \\ \dot{\beta} \\ \dot{\gamma} \end{bmatrix}, \tag{5.87}$$

which reads in compact form (${}^3\mathbf{G}_{TB} = {}^b\mathbf{G}_{TB}$),

$${}^3\boldsymbol{\omega}_{30} = {}^3\mathbf{G}_{TB}(\boldsymbol{\beta})\dot{\boldsymbol{\beta}} \tag{5.88}$$

In `Exudyn`, there are the following **functions and items related to Tait-Bryan angles**:

- `NodeRigidBodyRxyz`: A 3D rigid body node based on Tait-Bryan angles
- `RotXYZ2RotationMatrix`: computes the rotation matrix from given Tait-Bryan angles
- `RotationMatrix2RotXYZ`: computes Tait-Bryan angles from a given rotation matrix
- `RotXYZ2G`: computes the matrix ${}^0\mathbf{G}$ relating time derivatives of Tait-Bryan angles to the (global) angular velocity vector
- `RotXYZ2Glocal`: computes the matrix ${}^b\mathbf{G}$ relating time derivatives of Tait-Bryan angles to the (local, body-fixed) angular velocity vector

5.3.7 Euler parameters and unit quaternions

As one of the most important forms of rotational parameters for multibody systems, the 4 Euler parameters are introduced. They can be mathematically represented as unit quaternions and provide

particularly simple expressions and equations of motion, albeit at the expense of an additional constraint.

The Euler parameters are defined as

$$\underline{\mathbf{p}} = \begin{bmatrix} p_\mathrm{s} \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \cos\frac{\varphi}{2} \\ \mathbf{u}\,\sin\frac{\varphi}{2} \end{bmatrix} \quad \text{bzw.} \quad \underline{\mathbf{p}} = \begin{bmatrix} p_\mathrm{s} \\ p_x \\ p_y \\ p_z \end{bmatrix} = \begin{bmatrix} \cos\frac{\varphi}{2} \\ u_x\,\sin\frac{\varphi}{2} \\ u_y\,\sin\frac{\varphi}{2} \\ u_z\,\sin\frac{\varphi}{2} \end{bmatrix} \tag{5.89}$$

Here, $p_s = \cos\frac{\varphi}{2}$ is the scalar part and $\mathbf{p} = \mathbf{u}\,\sin\frac{\varphi}{2}$ is the vector part of the Euler parameters $\underline{\mathbf{p}}$. The four Euler parameters $\underline{\mathbf{p}}$ are subject to the normalization condition

$$\phi(\underline{\mathbf{p}}) \equiv p_s^2 + p_x^2 + p_y^2 + p_z^2 - 1 = 0 \quad \text{or} \quad p_s^2 + \mathbf{p}^\mathrm{T}\,\mathbf{p} - 1 = 0\,. \tag{5.90}$$

First, the following trigonometric transformations are introduced,

$$\cos\varphi = 2\cos^2\frac{\varphi}{2} - 1, \quad \sin\varphi = 2\sin\frac{\varphi}{2}\cos\frac{\varphi}{2}, \quad 1 - \cos\varphi = 2\sin^2\frac{\varphi}{2}\,. \tag{5.91}$$

When these are substituted into the rotation tensor using Eq. (5.89), it follows

$$\mathbf{A}(\underline{\mathbf{p}}) = (2p_\mathrm{s}^2 - 1)\,\mathbf{E} + 2p_\mathrm{s}\,\tilde{\mathbf{p}} + 2\,\mathbf{p}\,\mathbf{p}^\mathrm{T}\,. \tag{5.92}$$

Euler parameters can initially be expressed in the global reference frame $\mathcal{F}_0$,

$${}^0\underline{\mathbf{p}} = \begin{bmatrix} p_\mathrm{s} \\ {}^0\mathbf{p} \end{bmatrix} = \begin{bmatrix} \cos\frac{\varphi}{2} \\ {}^0\mathbf{u}\,\sin\frac{\varphi}{2} \end{bmatrix}\,. \tag{5.93}$$

Thus, the coordinate representation of the rotation tensor is given by

$$\mathbf{A}({}^0\underline{\mathbf{p}}) = 2\,{}^0\begin{bmatrix} p_\mathrm{s}^2 + p_x^2 - \frac{1}{2} & p_x p_y - p_\mathrm{s} p_z & p_x p_z + p_\mathrm{s} p_y \\ p_x p_y + p_\mathrm{s} p_z & p_\mathrm{s}^2 + p_y^2 - \frac{1}{2} & p_y p_z - p_\mathrm{s} p_x \\ p_x p_z - p_\mathrm{s} p_y & p_y p_z + p_\mathrm{s} p_x & p_\mathrm{s}^2 + p_z^2 - \frac{1}{2} \end{bmatrix} = {}^{01}\mathbf{A} \tag{5.94}$$

Note that the axis of rotation is invariant to the rotation, ${}^0\mathbf{u} = {}^1\mathbf{u}$, thus ${}^0\underline{\mathbf{p}} = {}^1\underline{\mathbf{p}}$ Due to the ambiguity $\mathbf{A}(\underline{\mathbf{p}}) = \mathbf{A}(-\underline{\mathbf{p}})$, the scalar part of the Euler parameters is usually normalized, i.e., $p_s \geq 0$.
The careful reader may observe that matrix $\mathbf{A}$ in Eq. (5.94) is not identical to the implementation in `Exudyn`. This is true, because there is always an alternative formulation for the diagonal terms by adding the normalization condition times a factor.

5.3.7.1 Quaternion operations

Calculations with Euler parameters are efficiently performed through the rules of quaternion algebra. Active perspective as vector rotation: The rotation of vector $\mathbf{r}_0$ into $\mathbf{r}(t)$,

$${}^0\mathbf{r}\,(t) = {}^0\mathbf{A}\,(t)\;{}^0\mathbf{r}_0 \quad \text{mit} \quad {}^0\mathbf{A}\,(t) = \mathbf{A}({}^0\mathbf{u}\,(t), \varphi(t)) \tag{5.95}$$

is formulated using the quaternion ${}^0\underline{\mathbf{p}} = \underline{\mathbf{p}}({}^0\mathbf{u}, \varphi)$ and its conjugate $\overline{\underline{\mathbf{p}}}$ with the help of the double quaternion product (operator $\circ$),

$$ {}^0\underline{\mathbf{r}}(t) = {}^0\underline{\mathbf{p}}(t) \circ {}^0\underline{\mathbf{r}}_0 \circ {}^0\overline{\underline{\mathbf{p}}}(t) , \tag{5.96} $$

or in short form,

$$ \begin{bmatrix} 0 \\ {}^0\mathbf{r}(t) \end{bmatrix} = \begin{bmatrix} p_\mathrm{s}(t) \\ {}^0\mathbf{p}(t) \end{bmatrix} \circ \begin{bmatrix} 0 \\ {}^0\mathbf{r}_0 \end{bmatrix} \circ \begin{bmatrix} p_\mathrm{s}(t) \\ -{}^0\mathbf{p}(t) \end{bmatrix} . \tag{5.97} $$

With the multiplication rule for quaternions, the scalar part yields $0 = 0$ and the vector part the vector rotation (5.95). For multiple rotations, we have

$$ {}^0\underline{\mathbf{r}}_2 = {}^0\underline{\mathbf{p}}_2 \circ {}^0\underline{\mathbf{p}}_1 \circ {}^0\underline{\mathbf{r}}_0 \circ {}^0\overline{\underline{\mathbf{p}}}_1 \circ {}^0\overline{\underline{\mathbf{p}}}_2 . \tag{5.98} $$

Given two quaternions

$$ \underline{\mathbf{a}} = a_\mathrm{s} + \mathrm{i}\, a_x + \mathrm{j}\, a_y + \mathrm{k}\, a_z, \quad \text{and} \quad \underline{\mathbf{b}} = b_\mathrm{s} + \mathrm{i}\, b_x + \mathrm{j}\, b_y + \mathrm{k}\, b_z , \tag{5.99} $$

we provide two important **rules of quaternion algebra**, which is the sum,

$$ \underline{\mathbf{c}} = \underline{\mathbf{a}} + \underline{\mathbf{b}} \quad \rightarrow \quad \begin{bmatrix} c_\mathrm{s} \\ \mathbf{c} \end{bmatrix} = \begin{bmatrix} a_\mathrm{s} \\ \mathbf{a} \end{bmatrix} + \begin{bmatrix} b_\mathrm{s} \\ \mathbf{b} \end{bmatrix} = \begin{bmatrix} a_\mathrm{s} + b_\mathrm{s} \\ \mathbf{a} + \mathbf{b} \end{bmatrix} , \tag{5.100} $$

and the multiplication

$$ \underline{\mathbf{c}} = \underline{\mathbf{a}} \circ \underline{\mathbf{b}} \neq \underline{\mathbf{b}} \circ \underline{\mathbf{a}} \quad \rightarrow \quad \begin{bmatrix} c_\mathrm{s} \\ \mathbf{c} \end{bmatrix} = \begin{bmatrix} a_\mathrm{s} \\ \mathbf{a} \end{bmatrix} \circ \begin{bmatrix} b_\mathrm{s} \\ \mathbf{b} \end{bmatrix} = \begin{bmatrix} a_\mathrm{s} b_\mathrm{s} - \mathbf{a}^\mathrm{T}\mathbf{b} \\ a_\mathrm{s}\mathbf{b} + b_\mathrm{s}\mathbf{a} + \tilde{\mathbf{a}}\mathbf{b} \end{bmatrix} . \tag{5.101} $$

5.3.7.2 Angular velocity vector

To derive the angular velocity from Euler parameters, there are both the possibilities of deriving the rotation matrix with respect to time and directly deriving the angular velocity from Euler parameters. For an infinitesimal rotation $\mathrm{d}\psi$, the quaternion follows

$$ \underline{\mathbf{p}}(\mathbf{e}, \mathrm{d}\psi) = \begin{bmatrix} \cos\frac{\mathrm{d}\psi}{2} \\ \mathbf{e}\sin\frac{\mathrm{d}\psi}{2} \end{bmatrix} \approx \begin{bmatrix} 1 \\ \mathbf{e}\frac{\mathrm{d}\psi}{2} \end{bmatrix} \tag{5.102} $$

from which we conclude that from $\underline{\mathbf{p}}(\mathbf{e}, \mathrm{d}\psi) \circ \underline{\mathbf{p}}(t) = \underline{\mathbf{p}}(t) + \mathrm{d}\underline{\mathbf{p}}$ and thus $\mathrm{d}\underline{\mathbf{p}} = \begin{bmatrix} 0 \\ \mathbf{e}\frac{\mathrm{d}\psi}{2} \end{bmatrix}$. Dividing by $\mathrm{d}t$ and with the angular velocity $\boldsymbol{\omega}_{10} = \mathbf{e}\,\dot{\psi}$, it follows

$$ \begin{bmatrix} \dot{p}_\mathrm{s} \\ {}_0\dot{\mathbf{p}} \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 0 \\ \boldsymbol{\omega}_{10} \end{bmatrix} \circ \begin{bmatrix} p_\mathrm{s} \\ \mathbf{p} \end{bmatrix} \tag{5.103} $$

or in short form

$$ {}^0\dot{\underline{\mathbf{p}}} = \frac{1}{2}\underline{\boldsymbol{\omega}}_{10} \circ \underline{\mathbf{p}} \tag{5.104} $$

With the quaternion product in matrix notation, it follows

$$ \begin{bmatrix} \dot{p}_\mathrm{s} \\ {}^0\dot{\mathbf{p}} \end{bmatrix} = \frac{1}{2} \begin{bmatrix} p_\mathrm{s} & -\mathbf{p}^\mathrm{T} \\ \mathbf{p} & p_\mathrm{s}\mathbf{E} - \tilde{\mathbf{p}} \end{bmatrix} \begin{bmatrix} 0 \\ \boldsymbol{\omega}_{10} \end{bmatrix} \tag{5.105} $$

or with the matrix $\underline{\mathbf{P}}$ it can be written as

$$ {}^0\underline{\dot{\mathbf{p}}} = \frac{1}{2}\underline{\mathbf{P}}(\mathbf{p})\underline{\boldsymbol{\omega}}_{10} \tag{5.106} $$

This form can be represented in global coordinates due to $\underline{\mathbf{P}}^{-1} = \underline{\mathbf{P}}^{\mathrm{T}}$ as

$$ \begin{bmatrix} 0 \\ {}^0\boldsymbol{\omega}_{10} \end{bmatrix} = 2 \begin{bmatrix} p_\mathrm{s} & {}^0\mathbf{p}^{\mathrm{T}} \\ -{}^0\mathbf{p} & p_\mathrm{s}\mathbf{E} + {}^0\tilde{\mathbf{p}} \end{bmatrix} \begin{bmatrix} \dot{p_\mathrm{s}} \\ {}_0\dot{\mathbf{p}} \end{bmatrix} \tag{5.107} $$

Thus, the velocity transformation follows to

$$ {}^0\mathbf{G}_{EP} = \left[-2\,{}^0\mathbf{p}\,,\; 2p_\mathrm{s}\mathbf{E} + 2\,{}^0\tilde{\mathbf{p}} \right] \tag{5.108} $$

as a matrix with 3 rows and 4 columns. With ${}^0\mathbf{G}_{EP}$, the angular velocity vector can be conveniently calculated,

$$ {}^0\boldsymbol{\omega}_{10} = \left[-2\,{}^0\mathbf{p}\,,\; 2p_\mathrm{s}\mathbf{E} + 2\,{}^0\tilde{\mathbf{p}} \right] {}^0\underline{\dot{\mathbf{p}}} \tag{5.109} $$

In `Exudyn`, there are the following **functions and items related to Euler parameters**:

- `NodeRigidBodyEP`: A 3D rigid body node based on Euler parameters
- `EulerParameters2RotationMatrix`: computes the rotation matrix from given Euler parameters
- `RotationMatrix2EulerParameters`: computes Euler parameters from a given rotation matrix
- `EulerParameters2G`: computes the matrix ${}^0\mathbf{G}_{EP}$ relating time derivatives of Euler parameters to the (global) angular velocity vector
- `EulerParameters2Glocal`: computes the matrix ${}^b\mathbf{G}_{EP}$ relating time derivatives of Euler parameters to the (local, body-fixed) angular velocity vector

5.4 Integration Points

For several tasks, especially for finite elements and contact, different integration rules are used, which are summarized here. The interval of all integration rules is $\in [-1, 1]$, thus giving a total sum for integration weights of 2. The points ξ_{ip} and weights w_{ip} for Gauss rules read:

The following table collects some typical **input parameters** for nodes, objects and markers:

type/order	**point 0**	**point 1**	**point 2**	**point 3**
Gauss 1	0			
Gauss 3	$-\sqrt{1/3}$	$\sqrt{1/3}$		
Gauss 5	$-\sqrt{3/5}$	0	$\sqrt{3/5}$	
Gauss 7	$-\sqrt{3/7 + \sqrt{120}/35}$	$-\sqrt{3/7 - \sqrt{120}/35}$	$\sqrt{3/7 - \sqrt{120}/35}$	$\sqrt{3/7 + \sqrt{120}/35}$
type/order	**weight 0**	**weight 1**	**weight 2**	**weight 3**
Gauss 1	2			
Gauss 3	1	1		
Gauss 5	5/9	8/9	5/9	
Gauss 7	$1/2 - 5/(3\sqrt{120})$	$1/2 + 5/(3 * \sqrt{120})$	$1/2 + 5/(3 * \sqrt{120})$	$1/2 - 5/(3 * \sqrt{120})$

The points ξ_{ip} and weights w_{ip} for Lobatto rules read:

type/order	point 0	point 1	point 2	point 3
Lobatto 1	-1	1		
Lobatto 3	-1	0	1	
Lobatto 5	-1	$-\sqrt{1/5}$	$\sqrt{1/5}$	1
type/order	**weight 0**	**weight 1**	**weight 2**	**weight 3**
Lobatto 1	1	1		
Lobatto 3	1/3	4/3	1/3	
Lobatto 5	1/6	5/6	5/6	1/6

Further integration rules can be found in the C++ code of `Exudyn`, see file `BasicLinalg.h`.

5.5 Model order reduction and component mode synthesis

This section describes the process how to create general flexible multibody system models using the floating frame of reference formulation with model order reduction (here also denoted as component mode synthesis (CMS)). The according object `ObjectFFRFreducedOrder` is described in Section 8.3.4.

5.5.1 Import of flexible bodies

For flexible bodies in multibody systems, specifically for model order reduction, a standard input data (SID) has been defined in the past [53]. A recent formulation for FFRF [67] showed that significantly less information is required for the computation of the dynamics of displacement-based solid finite elements:

- nodal reference positions (given in `FEMinterface` member variable `nodes['Position']`)
- stiffness matrix (given in `FEMinterface` member variable `stiffnessMatrix`)
- mass matrix (given in `FEMinterface` member variable `massMatrix`)
- (given in `FEMinterface` member variable `nodes['Position']`)

In addition, the following data may be needed:

- element connectivity: needed for visualization, surface reconstruction and stress computation
- list of surface elements: if not computed internally in `Exudyn` (stored in `FEMinterface` member variable `surface`)
- information on how to obtain stresses from the set of (reduced) coordinates: if not computed internally in `Exudyn` (stored in `FEMinterface` member variable `postProcessingModes` for stresses at nodal positions)

This data can be generated by an appropriate interface to NGsolve:

- `FEMinterface.ImportMeshFromNGsolve(...)`,

or imported from ABAQUS with `FEMinterface` functions

- `ReadMassMatrixFromAbaqus(...)`,
- `ReadStiffnessMatrixFromAbaqus(...)`,
- `ImportFromAbaqusInputFile(...)`)

and similar functionality exists for ANSYS. Importing data may be time consuming, which is why all FEMinterface data, including computed modes, can be saved and loaded via `SaveToFile` and `LoadFromFile`.

It is assumed that there exists an underlying solid finite element mesh, given by e.g., tetrahedral or hexahedral finite elements. However, this mesh is not needed for computations. If the surface or any part of the flexible body shall be visualized, surface elements (triangles or quads) need to be provided with indices of the mesh nodes. The computation of surface elements is done by `FEMinterface` function `VolumeToSurfaceElements`, making use of solid finite elements stored in `FEMinterface.elements`.

A major advantage of the `FEMinterface` data is that it is widely independent of underlying finite element technologies, specifically finite element order, reduced integration, etc., however, stress or strain can not be computed as well. A conventional way is to store computed body deformations and perform post processing in the original finite element code, which gives highest quality of stress, strain, and other quantities. A second way is, due to linearity of the small deformation assumptions, to use post-processing modes, such as modes to represent stress components. Post processing modes may be defined in the `FEMinterface` member `postProcessingModes`, which is a dictionary containing the modes stored in columns of the `matrix`, cyclic for every (stress / strain) component and for all modes, and an extra field `outputVariableType` which denotes the type of modes. The post processing modes may be directly computed in NGsolve with `ComputePostProcessingModesNGsolve`, using efficient and accurate internal functionality, or calculated with a much slower and very basic Python function `ComputePostProcessingModes`, which can compute simplified stresses or strains for 4-noded tetrahedral elements.

5.5.2 Eigenmodes

This section will describe the computation of eigenmodes using FEMinterface.

The `FEMinterface` in the module `FEM` has various functionality to import finite element meshes from finite element software. We create a `FEMinterface` by means of

```
fem = FEMinterface()
```

which allows us to use the variable `fem` from now.

Meshes can be imported from NETGEN/NGsolve (Section 7.7.8), Abaqus (see Section 7.7.8 and other sections related to ABAQUS), ANSYS (see Section 7.7.8 and other sections related to ANSYS). The import procedure, which can also be done manually, needs to include `massMatrix` $\mathbf{M}$ and `stiffnessMatrix` $\mathbf{K}$ from any finite element model. Note that many functions are based on the requirement that nodes are 3D displacement-based nodes, without rotation or other coordinates.

For any functionality with `ObjectFFRFreducedOrder` and for the computation of Hurty-Craig-Bampton modes as described in the next section, `nodes` are required. Finally, `elements` need to be included for visualization, and a surface needs to be reconstructed from the element connectivity, which is available for tetrahedral and hexahedral elements for most import functions.

As an example, we consider a part denoted as 'hinge' in the following, see Fig. 5.12. The test example can be found in `Examples/NGsolveCMStutorial.py` with lots of additional features.

After import of mass and stiffness matrix, eigenmodes and eigenfrequencies can be computed using `fem.ComputeEigenFrequencies(...)`, which computes the quantities `fem.modeBasis` and `fem.eigenValues`. The eigenvalues in Hz can be retrieved also with the function `fem.GetEigenFrequenciesHz()`. The function `fem.ComputeEigenFrequencies(...)` is available for dense and sparse matrices, and uses `scipy.linalg` to compute eigenvalues of the linear, undamped mechanical system

$$\mathbf{M}\ddot{\mathbf{q}}(t) + \mathbf{K}\mathbf{q}(t) = \mathbf{f}(t) \,. \tag{5.110}$$

Here, the total number of coordinates of the system is n, thus having the vector of system coordinates $\mathbf{q} \in \mathbb{R}^n$, vector of applied forces $\mathbf{f} \in \mathbb{R}^n$, mass matrix $\mathbf{M} \in \mathbb{R}^{n\times n}$ and stiffness matrix $\mathbf{K} \in \mathbb{R}^{n\times n}$. If we are

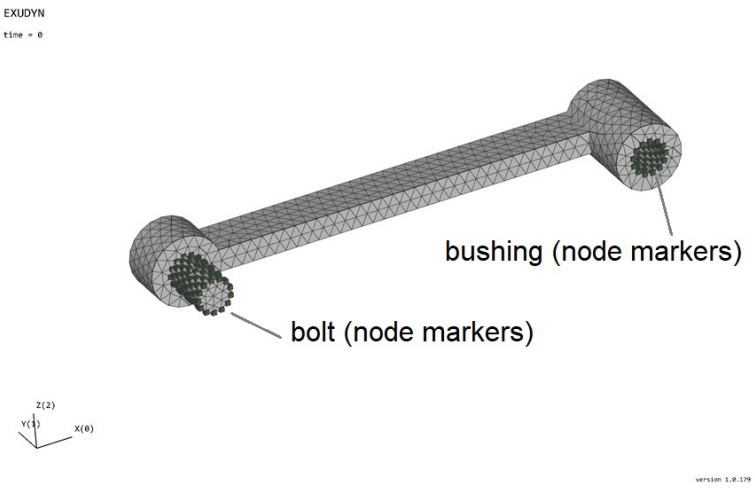


Figure 5.12: Test model and mesh for hinge created with Netgen (linear tetrahedral elements).

interested in free vibrations of the system, without any boundary conditions or interconnections to other bodies, Eq. (5.110) can be converted to a generalized eigenvalue problem. Using the approach $\mathbf{q}(t) = \mathbf{v}e^{i\omega t}$ in Eq. (5.110), and thus $\ddot{\mathbf{q}}(t) = -\omega^2\mathbf{q}(t)$, we obtain

$$\left[\left(-\omega^2\mathbf{M} + \mathbf{K}\right)\mathbf{v}\right]e^{i\omega t} = \mathbf{0}\,. \tag{5.111}$$

Assuming that Eq. (5.111) is valid for all times, the **generalized eigenvalue problem** follows that

$$\left(-\omega^2\mathbf{M} + \mathbf{K}\right)\mathbf{v} = \mathbf{0}\,, \tag{5.112}$$

which can be rewritten as

$$\det\left(-\omega^2\mathbf{M} + \mathbf{K}\right) = 0\,, \tag{5.113}$$

and which defines the eigenvalues ω_i^2 of the linear system, where $i \in \{0, \ldots, n-1\}$. Note that in this case, the eigenvalues are the squared eigenfrequencies (in rad/s). We can use eigenvalue algorithms to compute the eigenvalues ω_i^2 and according eigenvectors $\mathbf{v}_i$ from Python. The function `fem.ComputeEigenmodes(...)` uses `eigh(...)` from `scipy.linalg` in the dense matrix mode, and in the sparse mode `eigsh(...)` from `scipy.sparse.linalg`, the latter being restricted to pure symmetric matrices. Using special shift-inverted techniques in `eigsh(...)`, it performs much better than standard settings. However, you may tune your specific eigenvalue problem by modifying the solver procedure (just copy that function and adjust to your needs). As an output, we obtain the smallest `nModes` eigenvectors (=eigenmodes)[2] of the system. Here, we will also use synonymously the terms 'eigenmodes' and 'normal modes', which result from an eigenvalue/eigenvector computation using certain (or even no) boundary conditions.

Clearly, if there are no supports included in the stiffness matrix, the resulting eigenmodes will contain 6 rigid body modes and we will also call this case for the computation of eigenmodes the free-free case, in analogy to a simply supported beam. This rigid body modes, which are usually not

[2]Eigenvectors are the result of the eigenvalue algorithm, such as the QR algorithm. The mechanical interpretation of eigenvectors are eigenmodes, that can be visualized as shown in the figures of this section.

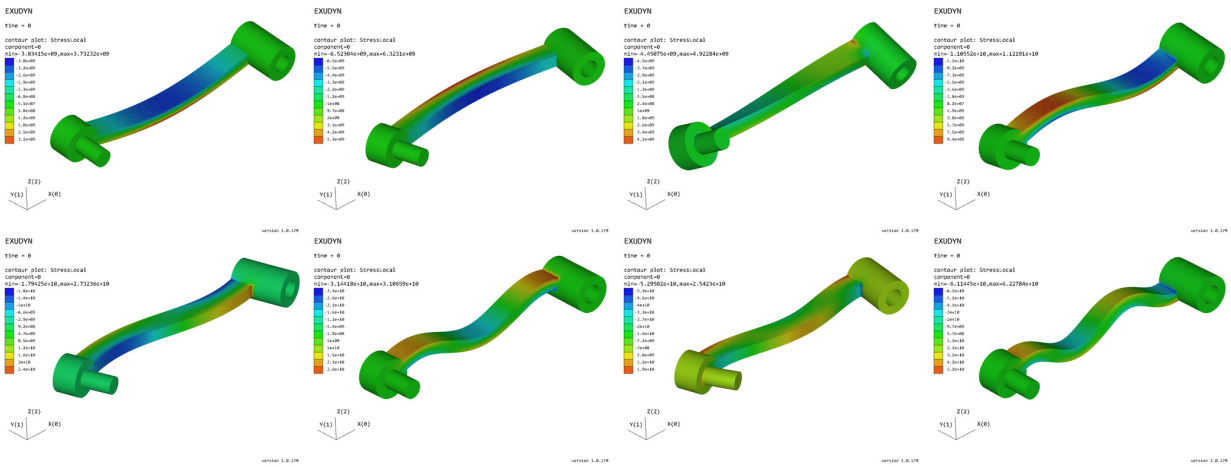


Figure 5.13: Lowest 8 free-free modes for hinge finite element model, contour plot for xx-stress component.

needed (=unwanted) in the succeeding computation, can be excluded with an according option in

```
fem.ComputeEigenFrequencies(excludeRigidBodyModes = ...)
```

For our test example, 8 eigenmodes are shown in Fig. 5.13, where the 6 rigid body modes have been excluded (so in total, 14 eigenvectors were computed). The 8 eigenfrequencies for the chosen coarse mesh with mesh size $h = 0.01$ and 1216 nodes result as

$$f_{0..7} = [671.59, 707.17, 1298.50, 1929.97, 1971.76, 3141.47, 3595.34, 4317.51]Hz \tag{5.114}$$

Note, that a computation with a finer mesh, using mesh size $h = 0.002$ and 100224 nodes, leads to significantly different eigenfrequencies, starting with $f_0 = 371.50$ Hz. This shows that quadratic finite elements would be more appropriate for this case.

After the computation of modes, it is always a good idea to visualize and/or animate these modes. We can do this, using the function `AnimateModes(...)` available in `exudyn.interactive`, which allows us to inspect and animate modes and to create animations for these modes, see the mentioned example.

Clearly, the free-free modes in Fig. 5.13 are not well suited for the modeling of the deformations within the hinge, if the bolt and the bushing shall be fixed to ground or to another part. Therefore, we can use modes based on ideas of Hurty [36] and Craig-Bampton [2], as shown in the following.

5.5.3 Hurty-Craig-Bampton modes

This section will describe the computation of static and eigen (normal) modes using FEMinterface. The theory is based on Hurty [36] and Craig-Bampton [2], but often only attributed to Craig-Bampton. Furthermore, boundaries are also called interfaces[3], as they either represent surface sections of our

[3]Here, and in the description of various Python functions, we will use boundary and interface often synonymously, as flexible bodies can be either connected to ground in the sense of a classical 'support-type' boundary condition, or they can represent the boundary of the flexible body as an interface to joints (via markers).

finite element model which are connected to the ground or they represent interfaces to joints and are connected to other bodies.

The computation of so-called static and normal modes follows a simple concept based on finite element mass and stiffness matrices. The final goal of the computation of modes is to approximate the solution $\mathbf{q} \in \mathbb{R}^n$ by means of a reduction basis $\mathbf{\Psi} \in \mathbb{R}^{n \times m}$ and a reduced set of coordinates $\mathbf{p} \in \mathbb{R}^m$, for which we assume $m \ll n$.

In order to include boundary/interface effects, we separate our nodes and the nodal coordinates into

a) boundary nodes $\mathbf{q}_b \in \mathbb{R}^{n_b}$ and

b) internal or inner nodes $\mathbf{q}_i \in \mathbb{R}^{n_i}$.

We assume that internal nodes are not exposed to boundary/interface conditions or to forces.

Therefore, we may rewrite Eq. (5.110) as follows

$$\begin{bmatrix} \mathbf{M}_{bb} & \mathbf{M}_{bi} \\ \mathbf{M}_{ib} & \mathbf{M}_{ii} \end{bmatrix} \begin{bmatrix} \ddot{\mathbf{q}}_b \\ \ddot{\mathbf{q}}_i \end{bmatrix} + \begin{bmatrix} \mathbf{K}_{bb} & \mathbf{K}_{bi} \\ \mathbf{K}_{ib} & \mathbf{K}_{ii} \end{bmatrix} \begin{bmatrix} \mathbf{q}_b \\ \mathbf{q}_i \end{bmatrix} = \begin{bmatrix} \mathbf{f}_b \\ \mathbf{0} \end{bmatrix} \tag{5.115}$$

or, equivalently,

$$\mathbf{M}_{bb}\ddot{\mathbf{q}}_b + \mathbf{M}_{bi}\ddot{\mathbf{q}}_i + \mathbf{K}_{bb}\mathbf{q}_b + \mathbf{K}_{bi}\mathbf{q}_i = \mathbf{f}_b \tag{5.116}$$

$$\mathbf{M}_{ib}\ddot{\mathbf{q}}_b + \mathbf{M}_{ii}\ddot{\mathbf{q}}_i + \mathbf{K}_{ib}\mathbf{q}_b + \mathbf{K}_{ii}\mathbf{q}_i = \mathbf{0} \,. \tag{5.117}$$

A pure static condensation follows from Eq. (5.115) with the assumption that inertia terms are neglected, leading to the static result for internal nodes,

$$\mathbf{q}_{i,stat} = -\mathbf{K}_{ii}^{-1}\mathbf{K}_{ib}\mathbf{q}_b \,. \tag{5.118}$$

A pure static condensation, also denoted as Guyan-Irons method, keeps boundary coordinates but removes all internal modes, using the approximation

$$\begin{bmatrix} \mathbf{q}_b \\ \mathbf{q}_i \end{bmatrix} \approx \begin{bmatrix} \mathbf{I} \\ -\mathbf{K}_{ii}^{-1}\mathbf{K}_{ib} \end{bmatrix} \mathbf{q}_b = \mathbf{\Psi}^{GI}\mathbf{q}_b \,, \tag{5.119}$$

which leads to no approximations ('exact') results for the static case, but poor performance in highly dynamic problems.

Significant improvement result from the Hurty-Craig-Bampton method, which adds eigenmodes of the internal coordinates (internal nodes). We assume that $\mathbf{\Psi}_{ii}$ is the matrix of eigenvectors as a solution to the eigenvalue problem

$$\left(-\omega^2\mathbf{M}_{ii} + \mathbf{K}_{ii}\right)\mathbf{v} = \mathbf{0} \,, \tag{5.120}$$

Hereafter, we will only keep the lowest (or other appropriate) m eigenmodes in a reduced eigenmode matrix,

$$\mathbf{\Psi}_{ii}^{(red)} = [\mathbf{\Psi}_{ii,0}, \ldots, \mathbf{\Psi}_{ii,m-1}] \tag{5.121}$$

Combining these 'fixed-fixed' eigenvectors with the Guyan-Irons reduction (5.119), we obtain the Hurty-Craig-Bampton modes as

$$\begin{bmatrix} \mathbf{q}_b \\ \mathbf{q}_i \end{bmatrix} \approx \begin{bmatrix} \mathbf{I} \\ -\mathbf{K}_{ii}^{-1}\mathbf{K}_{ib} \end{bmatrix} \mathbf{q}_b + \begin{bmatrix} \mathbf{0} \\ \mathbf{\Psi}_{r,i} \end{bmatrix} \mathbf{p}_r \ , \tag{5.122}$$

or in matrix form

$$\begin{bmatrix} \mathbf{q}_b \\ \mathbf{q}_i \end{bmatrix} \approx \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{K}_{ii}^{-1}\mathbf{K}_{ib} & \mathbf{\Psi}_{r,i} \end{bmatrix} \begin{bmatrix} \mathbf{q}_b \\ \mathbf{p}_r \end{bmatrix} = \mathbf{\Psi}^{HCB}\mathbf{p}^{HCB} \ . \tag{5.123}$$

The disadvantage of Eq. (5.123) is evident by the fact that there may be a large number of boundary/interface nodes, leading to a huge number of static modes (100s or 1000s) and thus making the model reduction inefficient. Therefore, we can switch to other interfaces, as described in the following.

5.5.3.1 Definition of RBE2 / RBE3 interfaces

A powerful extension, which is available in many finite element as well as flexible multibody codes, is the definition of special boundary/interface conditions, based on pure rigid body motion. The so-called RBE2 boundaries are defined such that they are firmly connected to a rigid frame, thus the boundary or interface can only undergo rigid body motion. The advantage of this procedure is that, in comparison to Eq. (5.123), the number of boundary/interface modes is given by 6 *rigid body* modes, which allow simple integration into standard joints of multibody systems, e.g., the `GenericJoint`. The disadvantage is that such modes usually lead to artificial stiffening and stresses close to the boundary.

For so-called RBE3 boundaries, the kinematics is significantly different. The displacement of RBE3 boundaries is the (weighted) average displacement of all boundary nodes. The resulting forces at the RBE3 boundary are equally distributed, again using node-weighting. The (linearized) rotation of RBE3 boundaries is computed as the weighted displacements of the boundaries and including the distance to the rotation axes. Forces due to torques at RBE3 boundaries are computed according to the weighting, again considering the distance to the rotation axes, see the according formulas later on. The computation of RBE3 boundaries widely follows the formulation of the `MarkerSuperElementRigid`, see Section 8.9.11.

5.5.3.2 Computation of Hurty-Craig-Bampton modes with RBE2 interfaces

In the following section, we show the procedure for the computation of static modes for the RBE2 rigid-body interfaces. Note that eigenmodes directly follow from matrices $\mathbf{M}_{ii}$ and $\mathbf{K}_{ii}$ as described in Section 5.5.3. The implementation is given in `fem.ComputeHurtyCraigBamptonModes(...)`, see Section 7.7.8.

First, we use the index j here as a node index, having the clear correspondence to the coordinate index i, that node j has coordinates $[3 \cdot j,\ 3 \cdot j + 1,\ 3 \cdot j + 2]$. Furthermore, nodes are split into boundary and internal nodes, which then leads to according internal and boundary coordinates. We shall note that this sorting is never done in the finite element model or matrices, but just some indexing (referencing) lists are generated and used throughout, using valuable features of `numpy.linalg` and `scipy.sparse`.

For a certain boundary node set $B = [j_0,\ j_1,\ j_2,\ ...] \in \mathbb{N}^{n_b}$ with certain n_b node indices $j_0, ...$, we define one boundary set. The following transformations need to be performed for every set of boundary node lists. We also assume that weighting of all boundary nodes is equal, which may not be appropriate in all cases.

If we assume that there may only occur rigid body translation and rotation for the whole boundary node set, which is according to the idea of so-called RBE2 boundary conditions, it follows that the translation of all boundary nodes is given by

$$\mathbf{T}_t = \begin{bmatrix} \mathbf{I} \\ \vdots \\ \mathbf{I} \end{bmatrix} \in \mathbb{R}^{3n_b \times 3} \tag{5.124}$$

with $\mathbf{I} \in \mathbb{R}^{3\times 3}$ identity matrices. The nodal translation coordinates on boundary B are denoted as $\mathbf{q}_{B,t} \in \mathbb{R}^3$. The translation of the boundary/interface is mapped to the boundary coordinates as follows (assuming only one boundary B),

$$\mathbf{q}_{b,t} = \mathbf{T}_t\, \mathbf{q}_{B,t} \tag{5.125}$$

The nodal rotation coordinates on boundary B are denoted as $\mathbf{q}_{B,r} \in \mathbb{R}^3$. The rotation of the boundary/interface is mapped to the boundary coordinates as follows (assuming only one boundary B),

$$\mathbf{q}_{b,r} = \mathbf{T}_r\, \mathbf{q}_{B,r} \tag{5.126}$$

The computation of matrix $\mathbf{T}_r$ is more involved. It is based on nodal (reference) position vectors $\mathbf{r}_j^{(0)}$, $j \in B$, the midpoint of all boundary nodes,

$$\mathbf{r}^{(m)} = \frac{1}{n_b} \sum_{j=0}^{n_b-1} \mathbf{r}_j^{(0)} \tag{5.127}$$

and the position relative to the midpoint, denoted as

$$\mathbf{r}_j = \mathbf{r}_j^{(0)} - \mathbf{r}^{(m)}\ . \tag{5.128}$$

Note that the coordinate system refers to the system used in the underlying finite element mesh. The transformation for rotation follows from

$$\mathbf{T}_r = \begin{bmatrix} \tilde{\mathbf{r}}_0 \\ \vdots \\ \tilde{\mathbf{r}}_{n_b-1} \end{bmatrix} \in \mathbb{R}^{3n_b \times 3}\ . \tag{5.129}$$

The total nodal coordinates at the boundary, representing translations and rotations, follow as

$$\mathbf{q}_B = \begin{bmatrix} \mathbf{q}_{B,t} \\ \mathbf{q}_{B,r} \end{bmatrix}\ , \tag{5.130}$$

and the transformation matrix for the translation and rotation simply reads

$$\mathbf{T} = [\mathbf{T}_t\ \ \mathbf{T}_r] \in \mathbb{R}^{3n_b \times 6}\ , \tag{5.131}$$

which provides the total mapping of boundary rigid body motion

$$\mathbf{q}_b = \mathbf{T}\,\mathbf{q}_B\ , \tag{5.132}$$

which is the sum of translation and rotation.

As an example, having the boundary nodes sorted for two boundary node set B_0 and B_1, we obtain the following transformation for the Hurty-Craig-Bampton method with only 6 modes per boundary node set,

$$\begin{bmatrix}\mathbf{q}_b\\ \mathbf{q}_i\end{bmatrix} \approx \begin{bmatrix} \mathbf{T}_0 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{T}_1 & \mathbf{0} \\ -\mathbf{K}_{ii}^{-1}\mathbf{K}_{ib}\begin{bmatrix}\mathbf{T}_0\\ \mathbf{0}\end{bmatrix} & -\mathbf{K}_{ii}^{-1}\mathbf{K}_{ib}\begin{bmatrix}\mathbf{0}\\ \mathbf{T}_1\end{bmatrix} & \mathbf{\Psi}_{r,i} \end{bmatrix} \begin{bmatrix}\mathbf{q}_{B_0}\\ \mathbf{q}_{B_1}\\ \mathbf{p}_r\end{bmatrix}\ . \tag{5.133}$$

with the new boundary node vector $\mathbf{q}_b = [\mathbf{q}_{B_0}^{\mathrm{T}}\ \ \mathbf{q}_{B_1}^{\mathrm{T}}]^{\mathrm{T}}$.
Notes:

- The inverse $\mathbf{K}_{ii}^{-1}$ is not computed, but this matrix is LU-factorized using sparse techniques.
- The factorization only needs to be applied to six vectors for every relevant boundary node set.
- One set of boundary nodes can be omitted from the final static modes in Eq. (5.133), because keeping all boundary modes, would introduce six rigid body motions to our mode basis, what is usually not wanted nor needed.

Using again the examples given in Fig. 5.12, we now obtain a set of modified modes using the function `fem.ComputeHurtyCraigBamptonModes(...)`. Fig. 5.14 shows the first 6 rigid body modes. Note that these modes are automatically removed in the function `fem.ComputeHurtyCraigBamptonModes(...)` with default settings. Fig. 5.15 shows the second set of 6 rigid body modes. Finally, 8 eigenmodes have been computed for the fixed-fixed case (where all boundary/interfaces nodes are fixed), see Fig. 5.16. The eigenfrequencies for this case now are significantly higher than in the free-free case, reading

$$f_{0..7} = [1277.35, 1469.86, 3336.91, 3584.28, ...] \tag{5.134}$$

5.5.3.3 Computation of Hurty-Craig-Bampton modes with RBE3 interfaces

we are currently finishing a paper, after which this section will be completed!

5.5.4 Computation of stresses and strains for CMS modes

The computation of stresses and strains is not directly possible if only knowing nodal displacements, stiffness matrix and mass matrix. In the following, we assume that we have a vector of nodal displacements $\mathbf{q}$, reduced coordinates $\mathbf{p}^R$, as well as a reduction matrix $\mathbf{\Psi}^R$, compare Eq. (5.123),

$$\mathbf{q} \approx \mathbf{\Psi}^R\mathbf{p}^R\ . \tag{5.135}$$

Knowing all nodal displacements of a finite element allows to compute displacement, stress, and strain field within the element. This procedure is usually done within the finite element codes. In

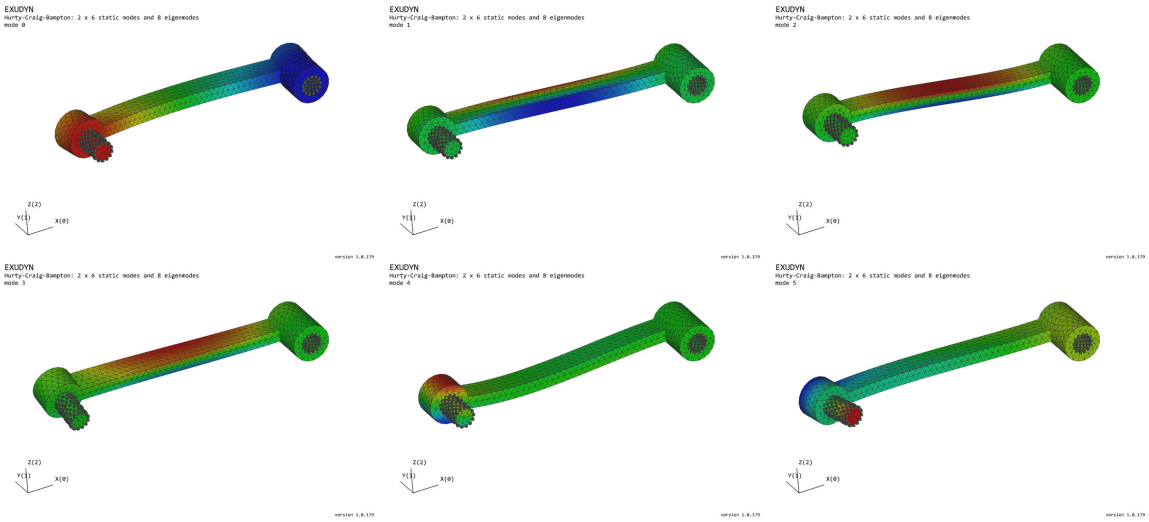


Figure 5.14: Static modes for bolt rigid body interface, using Hurty-Craig-Bampton method; top three images show (x,y,z)-translation modes, bottom three images show (x,y,z)-rotation modes; contour color represents norm of displacements.

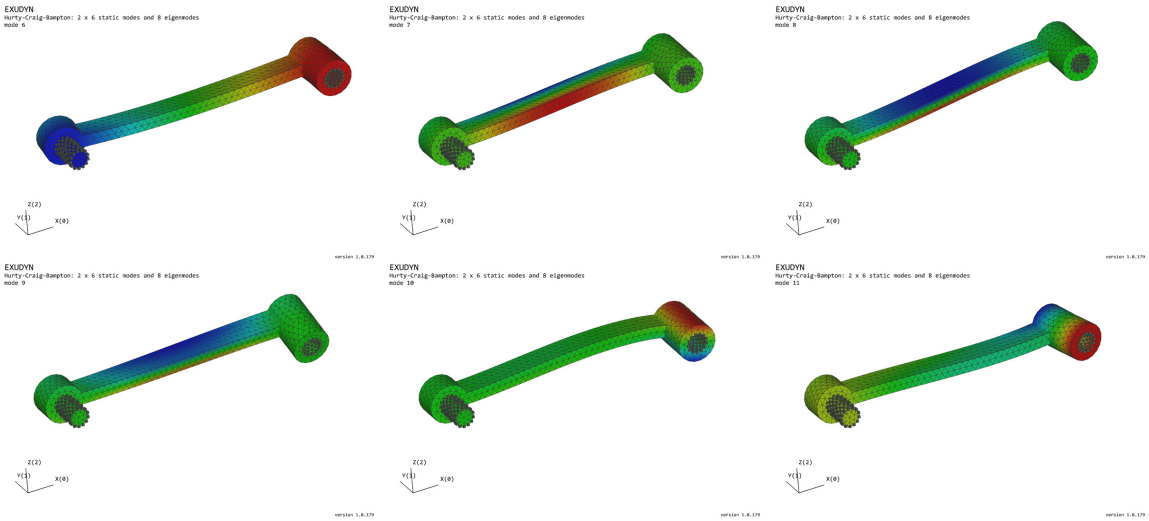


Figure 5.15: Static modes for bushing rigid body interface, using Hurty-Craig-Bampton method; top three images show (x,y,z)-translation modes, bottom three images show (x,y,z)-rotation modes; contour color represents norm of displacements.

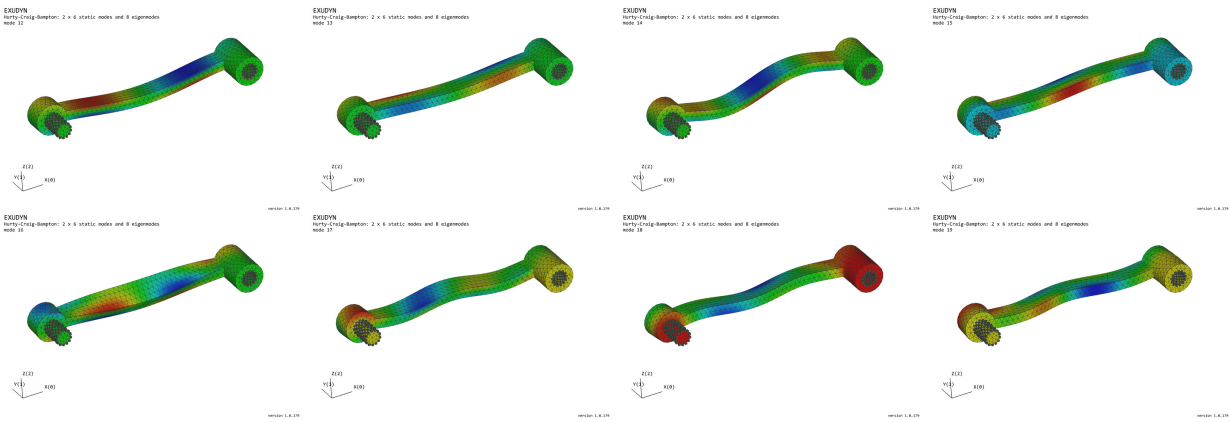


Figure 5.16: Eigenmodes for fixed-fixed case, resulting from Hurty-Craig-Bampton method; contour color represents norm of displacements.

particular, one should know that stress and strain quantities are having a lower order of accuracy than displacements and they may be more accurate in certain points, e.g., integration points. Furthermore, stress and strain quantities may have jumps along element boundaries, which is why they are usually post-processed in order to at least look smoother but in general also are more accurate.

In `Exudyn`, we have the option to pre-compute stress or strain components at finite element nodes, see the options below. Due to the fact that the FFRF / CMS formulation is assuming small (linearized) strains only, we are able to superimpose stress and strain for each mode. Independently of the quantity we intend to compute (stress, strain or similar), we use post-processing modes, which allow to represent special output variables.

Having a modal coordinate $\mathbf{p}_k^R$, we define a post-processing mode (pm) such that

$$\mathbf{s}_k^{\text{pm}} = \boldsymbol{\Psi}_k^{\text{pm}} \mathbf{p}_k^R \,, \tag{5.136}$$

in which $\mathbf{s}_k^{\text{pm}}$ represents for example the stress component σ_{xx} for the mode k. Putting together all stress modes for σ_{xx}, σ_{yy}, σ_{zz}, σ_{yz}, σ_{xz}, and σ_{xy},

$$\boldsymbol{\Psi}^{\sigma_{xx}} = \left[\boldsymbol{\Psi}_0^{\sigma_{xx}}, \boldsymbol{\Psi}_1^{\sigma_{xx}}, \ldots, \boldsymbol{\Psi}_{m-1}^{\sigma_{xx}}\right] \,, \tag{5.137}$$

we are able to compute σ_{xx} for all nodes from the relation

$$\mathbf{s}^{\sigma_{xx}} = \boldsymbol{\Psi}^{\sigma_{xx}} \mathbf{p}^R \tag{5.138}$$

Using the `FEMinterface` member `postProcessingModes`, the `FEM` module, we can define the `matrix` as $\boldsymbol{\Psi}^{\sigma_{ij}}$ for every node of the finite element mesh. In particular, one has to store all stress (or strain) components consecutively for each mode, which means that for mode k, $\boldsymbol{\Psi}$ contains the columns

$$\boldsymbol{\Psi}_k^{\sigma} = \left[\boldsymbol{\Psi}_k^{\sigma_{xx}}, \boldsymbol{\Psi}_k^{\sigma_{yy}}, \boldsymbol{\Psi}_k^{\sigma_{zz}}, \boldsymbol{\Psi}_k^{\sigma_{yz}}, \boldsymbol{\Psi}_k^{\sigma_{xz}}, \boldsymbol{\Psi}_k^{\sigma_{xy}}\right] \,, \tag{5.139}$$

For more details, see the `FEM` module in Section 7.7.8, either for function `ComputePostProcessingModes` or `ComputePostProcessingModesNGsolve`.

In order to retrieve modes, we currently have three options:

- Re-compute stress or strain quantities for given material parameters from nodal displacements for linear tetrahedral elements (Tet4), using the function `ComputePostProcessingModes` within the `FEM` module. This function is implemented in Python and therefore comparatively slow.
- For NGsolve models, you can use the `ComputePostProcessingModesNGsolve`, which takes the finite element space and material to compute post-processing modes directly in NGsolve, which is comparatively fast, if you do not have an excessive amount of modes and nodes.
- You can compute the post-processing modes within your finite element tool, such as Ansys or Simulia(ABAQUS) and import them manually. There exists no functionality in `Exudyn` to do so.

In general, one should know that the size of postprocessing modes may be huge. If you have 200 000 nodes and 100 modes, the matrix $\boldsymbol{\Psi}^{\sigma}$ would have the size $200\,000 \times (6 \cdot 100)$, thus leading to 120 000 000 components, close to 1GB of memory. In other words, it could make sense to consider computation of stresses in a post-computing phase.

5.5.5 Interfaces and boundaries

Being able to model a sole flexible body is not sufficient for the modeling of industrial problems. An important part of component mode synthesis is the appropriate definition of boundaries or interfaces. The term interface is widely used and may be more appropriate when connecting two bodies via such interfaces. However, in some cases the flexible body may be fixed to ground via such a boundary. In order to distinguish boundary/interface (b) and internal nodes (i), boundary seems to be appropriate and boundary/interface will be used synonymously in the context of flexible bodies.

An boundary/interface is represented by a certain surface area of a body, usually defined by surface elements and underlying nodes. For simplicity, it may just be defined by means of a node set. This is sufficient, in order for most of the previously described algorithms to work. If node sets are not imported from the underlying finite element codes, practical functions exist for the definition of node sets from geometrical operations, specifically[4]:

- `GetNodeAtPoint`: returns node number of a single node (if found) at given spatial position, with certain tolerance
- `GetNodesInPlane`: returns all nodes lying on a defined plane with certain tolerance
- `GetNodesInCube`: returns all nodes lying in a axis-parallel cube
- `GetNodesOnLine`: returns all nodes lying on a line defined by two points, with certain tolerance
- `GetNodesOnCylinder`: returns all nodes lying on a cylinder defined by two points and radius, with certain tolerance
- `GetNodesOnCircle`: returns all nodes lying on a circle defined by point, normal and radius, with certain tolerance

In order to compute according weighting factors, surface elements need to exist, either importing them the finite element code, or by using the `FEMinterface` member `surface`. The surface of tetrahedral or hexahedral meshes, which follow a standard node numbering, can be computed using the `FEMinterface` function `VolumeToSurfaceElements`.

[4]Note that these functions perform a linear search in the whole mesh, which is computationally inefficient if it is called many times.

5.5.6 Node weighting

As mentioned in the literature [29], there are certain advantages to use regular meshes on boundaries/interfaces. However, industrial relevant geometries often cannot be meshed by regular hexahedral meshes which leads to unstructured tetrahedral elements with (nearly) arbitrary triangular surfaces. While being a more general approach, an according nodal weighting is inevitable for unstructured surface meshes. As a drawback, accurate nodal weighting for application of forces or for computation of average displacements or rotations requires the information of underlying finite element interpolation functions, which are avoided in the present approach. A simplified, first order accurate functionality is provided by `GetNodeWeightsFromSurfaceAreas`, which reconstructs nodal weights for a set of node numbers from a given triangulated surface in `FEMinterface`. After identification of surface triangles and computation of according triangle areas, the weight w_i of every node i is built upon the according area of all connected triangles j,

$$w_i = \frac{1}{3A_B} \sum_j A_j \,, \quad \text{and} \quad \sum_i w_i = 1 \tag{5.140}$$

using the total area A_B of the boundary. This weighting leads to nearly constant strain distribution along the cross section of a fixed bar with equally distributed axial forces.

5.5.7 Reference conditions

Currently, there is no specific functionality to define reference conditions for FFRF objects in `Exudyn`. In the `ObjectFFRF`, a `ObjectConnectorCoordinateVector` needs to be used to define constraints of a so-called Tisserand frame.

In the `ObjectFFRFreducedOrder`, there are in general two approaches:

- The computed modes do not include rigid body motions, by using the appropriate flag `excludeRigidBodyModes = True` for most of such functions; in this case, the reference conditions are defined such that the reference node positions of the mesh are rigidly attached to the reference frame. In case of Hurty-Craig-Bampton modes, one boundary set (the first one) is attached to the reference frame.
- Alternatively, `excludeRigidBodyModes` can be set False, or arbitrary modes can be imported from elsewhere. In this case, rigid body motion must be excluded by appropriate constraints, e.g., a `ObjectConnectorCoordinateVector` applied to the `NodeGenericODE2` of `ObjectFFRFreducedOrder`. This task is completely left to the user.

It should be noted that regarding efficiency or highest accuracy, better reference conditions may exists, which are not fully supported in the current code and may only be applied with user functions.

5.6 Modeling of Contact in Exudyn

The `GeneralContact` module, see Section 6.8, which is

still under developement, consider with care!

provides a simple, efficient and versatile interface to a general contact module. The movivation for this module is based on the need for simple contact modeling in robotics, but also for the efficient modeling of beam-cylinder or beam-beam contact, as well as contact between deformable meshes (not yet available).

Note that there are currently only simplistic contact models, such as linear contact and simple damping, which are not representing realistic Hertzian contact (which will be implemented in near future). Furthermore, read the notes in `GeneralContact` carefully, how stiffness and damping is realized – e.g., stiffness may be a serial spring against the other object, while damping is implemented as parallel damper.

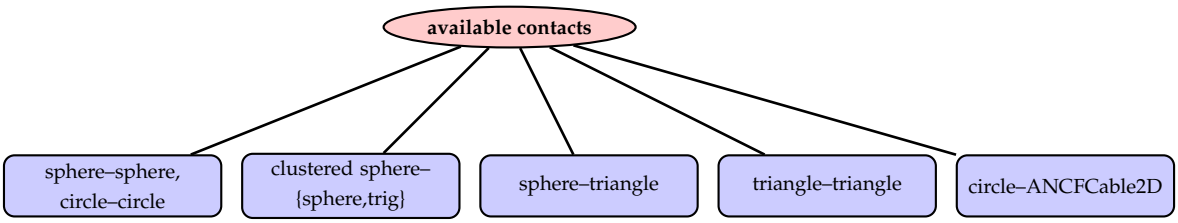


Figure 5.17: Contact: possible coupling of geometrical objects in `Exudyn`

Fig. 5.17 shows the implemented / possible coupling of contact objects:

1) simulate spherical particles; in 2D, spheres are represented as circles
2) simulate clustered spherical [circular] particles which consist of rigid bodies made of a cluster of spheres; several contact spheres are attached to one rigid body by using rigid body markers; in 2D, spheres are represented as circles
3) simulate the contact of spheres with triangular meshes, e.g., in order to provide some limitations of the range of motion for your objects
4) simulate the contact between arbitrarily shaped rigid bodies
5) simulate the contact between rolls (spheres) and ANCF cable elements; to enable cable-cable contact, spheres must be attached and distributed along the cable elements

In all use cases, explicit integrators are much faster and they are recommended, as long as your problem allows to do so.

5.6.1 Contact of meshed rigid bodies

Case 4) is more involved and needs further explanations (which more or less also applies to case 4). The geometry is approximated by a mesh consisting of flat triangles, which are attached to a rigid body (marker). The triangles obtain a contact stiffness against spheres. There are special cases, depending if the sphere gets in contact with the triangle plane or with the triangle edge. Having

contact with edges, usually involves several triangles at the same time (specifically at vertices), which leads to higher contact stiffness as compared to a planar contact. Nevertheless, for flat planes, contact computation takes care, that contact stiffness is constant in the whole plane, independently of the number of involved triangles. In order to realize contact between meshed rigid bodies, both body-attached meshes are added via the `GeneralContact` function

- `AddTrianglesRigidBodyBased(...)`

In addition, all mesh vertices are added as spheres with markers using

- `AddSphereWithMarker(...)`

However, as we need a certain finite radius of the spheres, the mesh must be shrinked for this purpuse (and it needs to have according thickness). Shrinking of the (consistent) triangular mesh can be done by the utility function

- `ShrinkMeshNormalToSurface(...);`
- in order to reduce artifacts at object edges, it is recommended to refine the mesh, using the utility function `RefineMesh(...)`

According examples can be found in test models, but there will be a more convenient function for contact of meshes attached to rigid bodies in the future.

All contacts can be created in a `GeneralContact` object – which is not a regular object in mbs – created by

- `gContact = mbs.AddGeneralContact()`

Note that one can create several, independent contact objects. Hereafter, spheres, triangles, ... are added with appropriate functions, see Section 6.8. Note that triangles need to be correctly numbered (see correct normals in Fig. 10.1), which defines inside/outside of a triangluar mesh.

5.6.2 Regularized friction

Within a regularized friction law, similar to a well known law attributed to Haff-Werner, the friction force $\mathbf{f}_f$ is computed from static (dry) friction coefficient μ_s and the friction regularization velocity[5] $v_{\mu,reg}$

$$\begin{aligned} v_t &= |{}^0\mathbf{v}_t|, \\ \mathbf{f}_f(v_t, |f_c|, \mu_s, v_{\mu,reg}, v_t, {}^0\mathbf{v}_t) &= \begin{cases} \frac{\mu_s \cdot |f_c|}{v_{\mu,reg}} {}^0\mathbf{v}_t, & \text{if} \quad v_t < v_{\mu,reg} \\ \mu_s \cdot |f_c| \frac{{}^0\mathbf{v}_t}{v_t}, & \text{else} \end{cases} \end{aligned} \tag{5.141}$$

Note that the following equations represent the computed contact relations in high detail, but minor cases and flags, such as the `intraSpheresContact` are not described here, but must be carefully considered in the description of `GeneralContact`, see Section 6.8.

[5] global regularization coefficient stored in `GeneralContact.frictionProportionalZone`

5.6.3 Sphere-sphere contact: Equations

The contact model between two spheres follows a penalty formulation, using a spring and optional damper to model the contact. Currently, only linear springs are utilized, however, the user is free to modify the equations in the code to model any nonlinear case as well. The equations for sphere-sphere contact in contact normal direction are very similar to the `ObjectConnectorSpringDamper`, see Section 8.6.1. Every sphere is attached to a position-based marker [6]. In C++, the sphere attached to marker 0 is denoted as `sphereI` and the sphere attached to marker 1 is denoted as `sphereJ`. Input parameters for this contact model are

intermediate variables	**symbol**	**description**
sphere i radius	r_i	radius of sphere i, attached to marker 0
sphere j radius	r_j	radius of sphere j, attached to marker 1
sphere i contact stiffness	k_i	N/m
sphere j contact stiffness	k_j	N/m
sphere i contact damping	d_i	N/m
sphere j contact damping	d_j	N/m
friction pairing coefficient	μ_{ij}	the friction coefficient stored in the the friction pairings matrix, resulting from the friction indices of spheres i and j

Marker positions and velocities are given by the relations:

intermediate variables	**symbol**	**description**
sphere i position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker m0
sphere j position	${}^0\mathbf{p}_{m1}$	current global position which is provided by marker m1
marker m0 position Jacobian	${}^0\mathbf{J}_{pos,m0}$	with interpretation as variation of the position $\delta\,{}^0\mathbf{p}_{m0} = {}^0\mathbf{J}_{pos,m0}\,\delta\mathbf{q}_{m0}$; assuming that $\mathbf{q}_{m0}$ represents the generalized coordinates of marker $m0$
marker m1 position Jacobian	${}^0\mathbf{J}_{pos,m1}$	with interpretation as variation of the position $\delta\,{}^0\mathbf{p}_{m1} = {}^0\mathbf{J}_{pos,m1}\,\delta\mathbf{q}_{m1}$; assuming that $\mathbf{q}_{m0}$ represents the generalized coordinates of marker $m0$
sphere i velocity	${}^0\mathbf{v}_i$	current global velocity which is provided by marker m0
sphere j velocity	${}^0\mathbf{v}_j$	
relative position	${}^0\mathbf{n}$	${}^0\mathbf{p}_j - {}^0\mathbf{p}_i$
Distance*	$L = \lvert{}^0\mathbf{n}\rvert$	
unit vector*	${}^0\mathbf{n}_0 = \frac{1}{L}\,{}^0\mathbf{n}$	vector in contact normal direction
gap*	$g = L - (r_i + r_j)$	
penetration*	$p = -g = r_i + r_j - L$	

In case of rigid bodies and non-zero friction, we also compute angular velocities and orientation,

[6] which can be attached itself to position nodes, rigid body nodes, point masses, rigid bodies as well as flexible bodies. NOTE, that in case of implicit integration, flexible bodies are not fully implemented!

intermediate variables: rigid bodies and friction	symbol	description
sphere i angular velocity	${}^{m0}\boldsymbol{\omega}_i$	current local angular velocity provided by marker $m0$
marker $m0$ orientation	${}^{0,m0}\mathbf{A}$	transformation from marker m0 (body-fixed) to global coordinates
sphere j angular velocity	${}^{m1}\boldsymbol{\omega}_j$	current local angular velocity provided by marker $m1$
marker $m1$ orientation	${}^{0,m1}\mathbf{A}$	transformation from marker m0 (body-fixed) to global coordinates
marker $m0$ rotation Jacobian	${}^{0}\mathbf{J}_{rot,m0}$	with interpretation as derivative of the global angular velocity ${}^{0}\boldsymbol{\omega}_{m0} = {}^{0}\mathbf{J}_{rot,m0}\,\dot{\mathbf{q}}_{m0}$; assuming that $\dot{\mathbf{q}}_{m0}$ represents the generalized velocities of marker $m0$
marker $m1$ rotation Jacobian	${}^{0}\mathbf{J}_{rot,m1}$	with interpretation as derivative of the global angular velocity ${}^{0}\boldsymbol{\omega}_{m1} = {}^{0}\mathbf{J}_{rot,m1}\,\dot{\mathbf{q}}_{m1}$; assuming that $\dot{\mathbf{q}}_{m1}$ represents the generalized velocities of marker $m1$

Contact between spheres with global index g_i and another sphere with global index g_j is active, if

- Bounding box of sphere g_j intersects with a box in the searchtree which also intersects with bounding box of sphere g_i AND
- if Bounding box of sphere g_j intersects with bounding box of sphere g_i AND
- if the condition $L^2 < \left(r_i + r_j\right)^2$ holds OR if g_j belongs to the active set of g_i (computed in PostNewtonStep).

Note that quantities * are only computed if contact is active.

5.6.3.1 Contact relations for sphere g_i (marker $m0$) and sphere g_j (marker $m1$)

If contact is active, we compute the global position of the contact point[7], see Fig. 5.18,[8]

$$ {}^{0}\mathbf{p}_c = {}^{0}\mathbf{p}_{m0} + r_i^* \cdot \mathbf{n}_0 \,, \tag{5.142} $$

the velocities of the spheres at the contact point[9],

$$ \begin{aligned} {}^{0}\mathbf{v}_{c,i} &= {}^{0}\mathbf{v}_i + \left({}^{0,m0}\mathbf{A}\ {}^{m0}\boldsymbol{\omega}_i\right) \times \left({}^{0}\mathbf{p}_c - {}^{0}\mathbf{p}_i\right) = {}^{0}\mathbf{v}_i + r_i^* \cdot \left({}^{0,m0}\mathbf{A}\ {}^{m0}\boldsymbol{\omega}_i\right) \times {}^{0}\mathbf{n}_0 \,, \\ {}^{0}\mathbf{v}_{c,j} &= {}^{0}\mathbf{v}_j + \left({}^{0,m1}\mathbf{A}\ {}^{m1}\boldsymbol{\omega}_j\right) \times \left({}^{0}\mathbf{p}_c - {}^{0}\mathbf{p}_j\right) = {}^{0}\mathbf{v}_j - r_j^* \cdot \left({}^{0,m1}\mathbf{A}\ {}^{m1}\boldsymbol{\omega}_j\right) \times {}^{0}\mathbf{n}_0 \,, \end{aligned} \tag{5.143} $$

the velocity in contact normal direction, which can be computed from sphere's center points,

$$ v_n = {}^{0}\mathbf{n}_0^{\mathrm{T}} \left({}^{0}\mathbf{v}_j - {}^{0}\mathbf{v}_i\right) = {}^{0}\mathbf{n}_0^{\mathrm{T}} \left({}^{0}\mathbf{v}_{c,j} - {}^{0}\mathbf{v}_{c,i}\right) \,, \tag{5.144} $$

[7] considering also penetration, a more consistent contact point would be ${}^{0}\mathbf{p}_c^* = {}^{0}\mathbf{p}_{m0} + (r_i - \frac{p}{2}) \cdot \mathbf{n}_0$, subtracting the half penetration.

[8] note that we have to consider the penetration when computing the contact points, attributing penetration equally to both sides; we introduce therefore the modified radii $r_i^* = r_i - \frac{p}{2}$ and $r_j^* = r_j - \frac{p}{2}$.

[9] In case of no friction, the angular velocities are not included in these relations

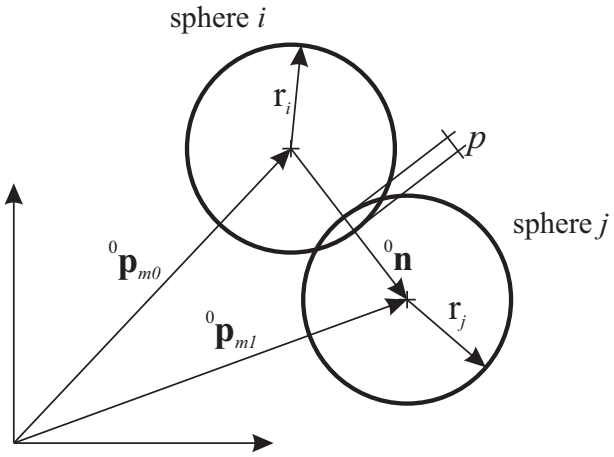


Figure 5.18: Geometrical relations for contact of two spheres i and j with according markers $m0$ and $m1$.

the velocity in tangential direction, considering the tangential velocities,

$$\begin{aligned} {}^0\mathbf{v}_t &= {}^0\mathbf{v}_{c,j} - {}^0\mathbf{v}_{c,i} - v_n \cdot {}^0\mathbf{n}_0 = \left(\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0\right)\left({}^0\mathbf{v}_{c,j} - {}^0\mathbf{v}_{c,i}\right), \\ &= \left(\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0\right)\left({}^0\mathbf{v}_j + r_j^* \cdot {}^0\tilde{\mathbf{n}}_0 \, {}^{0,m1}\mathbf{A} \, {}^{m1}\boldsymbol{\omega}_j - {}^0\mathbf{v}_i + r_i^* \cdot {}^0\tilde{\mathbf{n}}_0 \, {}^{0,m0}\mathbf{A} \, {}^{m0}\boldsymbol{\omega}_i\right) \\ &= \left(\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0\right)\left({}^0\mathbf{v}_j + r_j^* \cdot {}^0\tilde{\mathbf{n}}_0 \, {}^0\boldsymbol{\omega}_j - {}^0\mathbf{v}_i + r_i^* \cdot {}^0\tilde{\mathbf{n}}_0 \, {}^0\boldsymbol{\omega}_i\right) , \end{aligned} \tag{5.145}$$

the effective contact stiffness coefficient based on the stiffness k_i of sphere i and stiffness k_j of sphere j,

$$k_c = \frac{k_i \cdot k_j}{k_i + k_j} , \tag{5.146}$$

and the effective contact damping coefficient[10] based on the damping k_i of sphere i and damping k_j of sphere j

$$d_c = d_i + d_j . \tag{5.147}$$

The contact force (negative contact pressure) is computed from gap g and normal velocity v_n,

$$f_c = k_c \cdot g + d_c \cdot v_n , \tag{5.148}$$

and the total vectorial contact force is computed with the help of Eq. (5.141), defining the friction force $\mathbf{f}_f(v_t, |f_c|, \mu_s, v_{\mu,reg}, v_t, {}^0\mathbf{v}_t)$,

$${}^0\mathbf{f}_c = f_c \cdot \mathbf{n}_0 + \mathbf{f}_f . \tag{5.149}$$

The torque due to friction for sphere i and sphere j results into[11]

$${}^0\boldsymbol{\tau}_{f,i} = (-r_i^* \cdot \mathbf{n}_0) \times {}^0\mathbf{f}_f , \quad {}^0\boldsymbol{\tau}_{f,j} = (-r_j^* \cdot \mathbf{n}_0) \times {}^0\mathbf{f}_f . \tag{5.150}$$

[10] Note that this simplicial damping law is used according to the idea of parallel dampers, because serial dampers would not allow to adjust damping for different particles

[11] note that both signs are the same and that ${}^0\mathbf{f}_f$ could be replaced by $\mathbf{f}_c$

5.6.3.2 Generalized forces due to contact

Based on the contact pressure and the friction forces, forces and torques are applied via the markers' Jacobians, resulting in generalized forces to whatever the marker is attached to.

The generalized forces to the marker $m0$ and $m1$ (sphere i and j) are computed as

$$\begin{aligned} \mathbf{f}_{m0,LHS} &= -\,{}^0\mathbf{J}_{pos,m0}^{\mathrm{T}}\,{}^0\mathbf{f}_c + {}^0\mathbf{J}_{rot,m0}^{\mathrm{T}}\,{}^0\boldsymbol{\tau}_{f,i}\,, \\ \mathbf{f}_{m1,LHS} &= {}^0\mathbf{J}_{pos,m1}^{\mathrm{T}}\,{}^0\mathbf{f}_c + {}^0\mathbf{J}_{rot,m1}^{\mathrm{T}}\,{}^0\boldsymbol{\tau}_{f,j}\,. \end{aligned} \tag{5.151}$$

5.6.3.3 Jacobi matrix for sphere g_i and sphere g_j

For implicit time integration, the (contact) Jacobian[12] represents the derivative of the generalized forces

$$\mathbf{f}_{LHS} = \begin{bmatrix} \mathbf{f}_{m0,LHS} \\ \mathbf{f}_{m1,LHS} \end{bmatrix} \tag{5.152}$$

with respect to the generalized coordinates affected by the two markers,

$$\mathbf{q} = \begin{bmatrix} \mathbf{q}_{m0} \\ \mathbf{q}_{m1} \end{bmatrix}. \tag{5.153}$$

The Jacobian thus reads[13]

$$\mathbf{J}_c = \frac{\partial \mathbf{f}_{LHS}}{\partial \mathbf{q}} = \begin{bmatrix} \frac{\partial\left(-{}^0\mathbf{J}_{pos,m0}^{\mathrm{T}}\,{}^0\mathbf{f}_c + {}^0\mathbf{J}_{rot,m0}^{\mathrm{T}}\,{}^0\boldsymbol{\tau}_{f,i}\right)}{\partial \mathbf{q}_{m0}} & \frac{\partial\left(-{}^0\mathbf{J}_{pos,m0}^{\mathrm{T}}\,{}^0\mathbf{f}_c + {}^0\mathbf{J}_{rot,m0}^{\mathrm{T}}\,{}^0\boldsymbol{\tau}_{f,i}\right)}{\partial \mathbf{q}_{m1}} \\ \frac{\partial\left({}^0\mathbf{J}_{pos,m1}^{\mathrm{T}}\,{}^0\mathbf{f}_c + {}^0\mathbf{J}_{rot,m1}^{\mathrm{T}}\,{}^0\boldsymbol{\tau}_{f,j}\right)}{\partial \mathbf{q}_{m0}} & \frac{\partial\left({}^0\mathbf{J}_{pos,m1}^{\mathrm{T}}\,{}^0\mathbf{f}_c + {}^0\mathbf{J}_{rot,m1}^{\mathrm{T}}\,{}^0\boldsymbol{\tau}_{f,j}\right)}{\partial \mathbf{q}_{m1}} \end{bmatrix} \tag{5.154}$$

The single terms may be expressed as

$$\begin{aligned} &\frac{\partial\left(-{}^0\mathbf{J}_{pos,m0}^{\mathrm{T}}\,{}^0\mathbf{f}_c + {}^0\mathbf{J}_{rot,m0}^{\mathrm{T}}\,{}^0\boldsymbol{\tau}_{f,i}\right)}{\partial \mathbf{q}_{m0,1}} = \\ &-\frac{\partial\,{}^0\mathbf{J}_{pos,m0}^{\mathrm{T}}}{\partial \mathbf{q}_{m0,1}}\,{}^0\mathbf{f}_c - {}^0\mathbf{J}_{pos,m0}^{\mathrm{T}}\frac{\partial\,{}^0\mathbf{f}_c}{\partial \mathbf{q}_{m0,1}} + \frac{\partial\,{}^0\mathbf{J}_{rot,m0}^{\mathrm{T}}}{\partial \mathbf{q}_{m0,1}}\,{}^0\boldsymbol{\tau}_{f,i} + {}^0\mathbf{J}_{rot,m0}^{\mathrm{T}}\frac{\partial\,{}^0\boldsymbol{\tau}_{f,i}}{\partial \mathbf{q}_{m0,1}}\,, \end{aligned} \tag{5.155}$$

and similar for m_1. In order to simplify implementation (avoiding arrays with 3 indices) and improve computational efficiency, derivatives of Jacobians are realized as

$$\frac{\partial\,{}^0\mathbf{J}_{pos,m0}^{\mathrm{T}}}{\partial \mathbf{q}_{m0}}\,{}^0\mathbf{f}_c = \frac{\partial\,{}^0\mathbf{J}_{pos,m0}^{\mathrm{T}}\,{}^0\bar{\mathbf{f}}_c}{\partial \mathbf{q}_{m0}}\,, \tag{5.156}$$

in which ${}^0\bar{\mathbf{f}}_c = {}^0\mathbf{f}_c$, but assumed to be a constant and not depending on $\mathbf{q}$ in the computation of derivatives. Note that derivatives for position Jacobians, e.g., $\frac{\partial\,{}^0\mathbf{J}_{pos,m0}^{\mathrm{T}}\,{}^0\bar{\mathbf{f}}_c}{\partial \mathbf{q}_{m0}}$ or rotation Jacobians are provided by the according markers (will be described there in the near future).

For the jacobians, we need to compute the derivatives of the following terms[14][15]:

[12] Here, we only consider the local Jacobian related to the coordinates underlying the two markers $m0$ and $m1$; in the implementation, the parts of the Jacobian are added to the sparse system

[13] **NOTE** that only terms marked in **green** are currently fully implemented and terms in **blue** are approximated, while other terms are neglected

[14] terms that are implemented are marked in green; black terms are not implemented or unused

[15] to keep derivations short, we use ${}^0\mathbf{J}_{pos}$, which represents $-\,{}^0\mathbf{J}_{pos,m0}$ in case of $\frac{\partial}{\partial \mathbf{q}_{m0}}$ and ${}^0\mathbf{J}_{pos,m1}$ in case of $\frac{\partial}{\partial \mathbf{q}_{m1}}$

- $L = \left({}^0\mathbf{n}^{\mathrm{T}}\,{}^0\mathbf{n}\right)^{\frac{1}{2}}$:

$$\frac{\partial L}{\partial \mathbf{q}_{m0,m1}} = \frac{\partial \left({}^0\mathbf{n}^{\mathrm{T}}\,{}^0\mathbf{n}\right)^{\frac{1}{2}}}{\partial \mathbf{q}_{m0,m1}} = \frac{1}{L}\left({}^0\mathbf{n}^{\mathrm{T}}\frac{\partial\,{}^0\mathbf{n}}{\partial \mathbf{q}_{m0,m1}}\right) = \frac{1}{L}\left({}^0\mathbf{n}^{\mathrm{T}}\,{}^0\mathbf{J}_{pos}\right) = \left({}^0\mathbf{n}_0{}^{\mathrm{T}}\,{}^0\mathbf{J}_{pos}\right) \tag{5.157}$$

- $L^{-1} = \left({}^0\mathbf{n}^{\mathrm{T}}\,{}^0\mathbf{n}\right)^{-\frac{1}{2}}$:

$$\frac{\partial L^{-1}}{\partial \mathbf{q}_{m0,m1}} = \frac{\partial \left({}^0\mathbf{n}^{\mathrm{T}}\,{}^0\mathbf{n}\right)^{-\frac{1}{2}}}{\partial \mathbf{q}_{m0,m1}} = -\frac{1}{L^3}\left({}^0\mathbf{n}^{\mathrm{T}}\frac{\partial\,{}^0\mathbf{n}}{\partial \mathbf{q}_{m0,m1}}\right) = -\frac{1}{L^2}\left({}^0\mathbf{n}_0{}^{\mathrm{T}}\,{}^0\mathbf{J}_{pos}\right) \tag{5.158}$$

- ${}^0\mathbf{n} = {}^0\mathbf{p}_j - {}^0\mathbf{p}_i$:

$$\frac{\partial\,{}^0\mathbf{n}}{\partial \mathbf{q}_{m0,m1}} = {}^0\mathbf{J}_{pos} \tag{5.159}$$

- ${}^0\mathbf{n}_0 = \frac{1}{L}\,{}^0\mathbf{n}$:[16]

$$\frac{\partial\,{}^0\mathbf{n}_0}{\partial \mathbf{q}_{m0,m1}} = -\frac{1}{L^3}\left({}^0\mathbf{n} \otimes {}^0\mathbf{n}\right){}^0\mathbf{J}_{pos} + \frac{1}{L}\,{}^0\mathbf{J}_{pos} = \frac{1}{L}\left(\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0\right){}^0\mathbf{J}_{pos} \tag{5.160}$$

- $g = L - r_i + r_j$:

$$\frac{\partial g}{\partial \mathbf{q}_{m0,m1}} = {}^0\mathbf{n}_0{}^{\mathrm{T}}\,{}^0\mathbf{J}_{pos} \tag{5.161}$$

- $v_n = \left({}^0\mathbf{v}_j - {}^0\mathbf{v}_i\right)^{\mathrm{T}}\,{}^0\mathbf{n}_0$ (**NOTE**: only valid in case that markers are attached to node or body reference point!!!):

$$\frac{\partial v_n}{\partial \mathbf{q}_{m0,m1}} = \left({}^0\mathbf{v}_j - {}^0\mathbf{v}_i\right)^{\mathrm{T}}\left(\frac{1}{L}\left(\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0\right){}^0\mathbf{J}_{pos}\right) \tag{5.162}$$

$$\frac{\partial v_n}{\partial \dot{\mathbf{q}}_{m0,m1}} = {}^0\mathbf{n}_0^{\mathrm{T}}\,{}^0\mathbf{J}_{pos} \tag{5.163}$$

- $f_c = k_c \cdot g + d_c \cdot v_n$:

$$\frac{\partial f_c}{\partial \mathbf{q}_{m0,m1}} = k_c\frac{\partial g}{\partial \mathbf{q}_{m0,m1}} + d_c\frac{\partial v_n}{\partial \mathbf{q}_{m0,m1}} = k_c \cdot {}^0\mathbf{n}_0{}^{\mathrm{T}}\,{}^0\mathbf{J}_{pos} + d_c \cdot \left({}^0\mathbf{v}_j - {}^0\mathbf{v}_i\right)^{\mathrm{T}}\left(\frac{1}{L}\left(\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0\right){}^0\mathbf{J}_{pos}\right) \tag{5.164}$$

$$\frac{\partial f_c}{\partial \dot{\mathbf{q}}_{m0,m1}} = d_c\frac{\partial v_n}{\partial \dot{\mathbf{q}}_{m0,m1}} = d_c\,{}^0\mathbf{n}_0^{\mathrm{T}}\,{}^0\mathbf{J}_{pos} \tag{5.165}$$

- ${}^0\mathbf{v}_t = {}^0\mathbf{v}_{c,j} - {}^0\mathbf{v}_{c,i} - v_n \cdot {}^0\mathbf{n}_0 = \left(\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0\right)\left({}^0\mathbf{v}_j - {}^0\mathbf{v}_i\right) + r_j^* \cdot {}^0\tilde{\mathbf{n}}_0\,{}^0\boldsymbol{\omega}_j + r_i^* \cdot {}^0\tilde{\mathbf{n}}_0\,{}^0\boldsymbol{\omega}_i$:

$$\begin{aligned}\frac{\partial\,{}^0\mathbf{v}_t}{\partial \mathbf{q}_{m0,m1}} = & -{}^0\mathbf{n}_0 \otimes \left({}^0\mathbf{v}_j - {}^0\mathbf{v}_i\right)\left(\frac{1}{L}\left(\mathbf{I} - {}^0\mathbf{n} \otimes {}^0\mathbf{n}\right){}^0\mathbf{J}_{pos}\right) \\ & -v_n\left(\frac{1}{L}\left(\mathbf{I} - {}^0\mathbf{n} \otimes {}^0\mathbf{n}\right){}^0\mathbf{J}_{pos}\right) - v_n \cdot \left(\frac{1}{L}\left(\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0\right){}^0\mathbf{J}_{pos}\right) \\ & +r_{i,j} \cdot \left(-\frac{1}{L}\,{}^0\tilde{\boldsymbol{\omega}}_{i,j}\left(\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0\right) + {}^0\tilde{\mathbf{n}}_0\frac{\partial\,{}^0\boldsymbol{\omega}_{i,j}}{\partial \mathbf{q}_{m0,m1}}\right)\end{aligned} \tag{5.166}$$

[16] NOTE: dyadic product $\otimes$

- velocity coordinate derivatives for ${}^0\mathbf{v}_t$:

$$\frac{\partial\, {}^0\mathbf{v}_t}{\partial \dot{\mathbf{q}}_{m0}} = \left(\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0\right)\left(-\,{}^0\mathbf{J}_{pos,m0}\right) + r_i^* \cdot {}^0\tilde{\mathbf{n}}_0\ {}^0\mathbf{J}_{rot,m0}\ , \tag{5.167}$$

$$\frac{\partial\, {}^0\mathbf{v}_t}{\partial \dot{\mathbf{q}}_{m1}} = \left(\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0\right)\left({}^0\mathbf{J}_{pos,m1}\right) + r_j^* \cdot {}^0\tilde{\mathbf{n}}_0\ {}^0\mathbf{J}_{rot,m1} \tag{5.168}$$

The contact force reads (note that because f_c is always negative, the sign of regularization term is negative),

$${}^0\mathbf{f}_c = f_c \cdot {}^0\mathbf{n}_0 + \mathbf{f}_f = \begin{cases} f_c\left({}^0\mathbf{n}_0 - \frac{\mu_s}{v_{\mu,reg}}\,{}^0\mathbf{v}_t\right), & \text{if} \quad |\,{}^0\mathbf{v}_t\,| < v_{\mu,reg} \\ f_c \cdot {}^0\mathbf{n}_0 + \mathbf{f}_f, & \text{else with } \mathbf{f}_f = const. \end{cases} \tag{5.169}$$

Thus we introduce a factor δ_f, which is $\delta_f = 1$ in the regularized small velocity state, and in the saturated (constant) friction force we use $\delta_f = 0$. Thus, the jacobian of the contact force ${}^0\mathbf{f}_c$ reads (note the diadic product $\otimes$),

$$\begin{aligned} \frac{\partial\, {}^0\mathbf{f}_c}{\partial \mathbf{q}_{m0,m1}} &= \left({}^0\mathbf{n}_0 - \delta_f \frac{\mu_s}{v_{\mu,reg}}\,{}^0\mathbf{v}_t\right) \otimes \frac{\partial f_c}{\partial \mathbf{q}_{m0,m1}} + f_c\left(\frac{\partial\, {}^0\mathbf{n}_0}{\partial \mathbf{q}_{m0,m1}} - \delta_f \frac{\mu_s}{v_{\mu,reg}} \frac{\partial\, {}^0\mathbf{v}_t}{\partial \mathbf{q}_{m0,m1}}\right) \\ &= \left({}^0\mathbf{n}_0 - \delta_f \frac{\mu_s}{v_{\mu,reg}}\,{}^0\mathbf{v}_t\right)\left(k_c \frac{\partial g}{\partial \mathbf{q}_{m0,m1}} + d_c \frac{\partial v_n}{\partial \mathbf{q}_{m0,m1}}\right) + f_c\left(\frac{\partial\, {}^0\mathbf{n}_0}{\partial \mathbf{q}_{m0,m1}} - \delta_f \frac{\mu_s}{v_{\mu,reg}} \frac{\partial\, {}^0\mathbf{v}_t}{\partial \mathbf{q}_{m0,m1}}\right) \\ &= \left({}^0\mathbf{n}_0 - \delta_f \frac{\mu_s}{v_{\mu,reg}}\,{}^0\mathbf{v}_t\right) \otimes \left(k_c \cdot {}^0\mathbf{n}_0^{\mathrm{T}} + d_c \cdot \left({}^0\mathbf{v}_j - {}^0\mathbf{v}_i\right)^{\mathrm{T}} \left(\frac{1}{L}\left(\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0\right)\right)\right) {}^0\mathbf{J}_{pos} + f_c \cdot \end{aligned} \tag{5.170}$$

The jacobian for the contact force ${}^0\mathbf{f}_c$ w.r.t. velocity marker coordinates reads, note that ${}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0 - \mathbf{I} = -\left(\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0\right)$,

$$\frac{\partial\, {}^0\mathbf{f}_c}{\partial \dot{\mathbf{q}}_{m0,1}} = \left({}^0\mathbf{n}_0 - \delta_f \frac{\mu_s}{v_{\mu,reg}}\,{}^0\mathbf{v}_t\right) \otimes \frac{\partial\, {}^0 f_c}{\partial \dot{\mathbf{q}}_{m0,1}} - f_c \cdot \left(\delta_f \frac{\mu_s}{v_{\mu,reg}} \frac{\partial\, {}^0\mathbf{v}_t}{\partial \dot{\mathbf{q}}_{m0,1}}\right) \tag{5.171}$$

$$\begin{aligned} \frac{\partial\, {}^0\mathbf{f}_c}{\partial \dot{\mathbf{q}}_{m0}} &= -d_c\left({}^0\mathbf{n}_0 - \delta_f \frac{\mu_s}{v_{\mu,reg}}\,{}^0\mathbf{v}_t\right) \otimes \left({}^0\mathbf{n}_0^{\mathrm{T}}\,{}^0\mathbf{J}_{pos,m0}\right) - \\ &\quad f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}}\left(-\left(\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0\right){}^0\mathbf{J}_{pos,m0} + r_i^* \cdot {}^0\tilde{\mathbf{n}}_0\ {}^0\mathbf{J}_{rot,m0}\right) \\ \frac{\partial\, {}^0\mathbf{f}_c}{\partial \dot{\mathbf{q}}_{m1}} &= d_c\left({}^0\mathbf{n}_0 - \delta_f \frac{\mu_s}{v_{\mu,reg}}\,{}^0\mathbf{v}_t\right) \otimes \left({}^0\mathbf{n}_0^{\mathrm{T}}\,{}^0\mathbf{J}_{pos,m1}\right) - \\ &\quad f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}}\left(\left(\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0\right){}^0\mathbf{J}_{pos,m1} + r_j^* \cdot {}^0\tilde{\mathbf{n}}_0\ {}^0\mathbf{J}_{rot,m1}\right) \end{aligned} \tag{5.172}$$

The jacobians for torques are computed for the case that friction is in the regularized small velocity state ($\delta_f = 1$), while otherwise derivatives of ${}^0\boldsymbol{\tau}_{f,(i,j)}$ are zero,

$${}^0\boldsymbol{\tau}_{f,i} = (-r_i^* \cdot {}^0\mathbf{n}_0) \times \left(-f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}}\,{}^0\mathbf{v}_t\right), \quad {}^0\boldsymbol{\tau}_{f,j} = (-r_j^* \cdot {}^0\mathbf{n}_0) \times \left(-f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}}\,{}^0\mathbf{v}_t\right) \tag{5.173}$$

in case of no friction or constant friction forces, $\delta_f = 0$. The jacobians follow from (accordingly for $\boldsymbol{\tau}_{f,i}$, $\boldsymbol{\tau}_{f,j}$ and derivatives w.r.t $\mathbf{q}_{m0,1}$):

$$\begin{aligned}
\frac{\partial\, {}^0\boldsymbol{\tau}_{f,(i,j)}}{\partial \mathbf{q}_{m0,1}} &= \frac{\partial \left(-r_{(i,j)} \cdot {}^0\mathbf{n}_0\right) \times \left(-f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}}\, {}^0\mathbf{v}_t\right)}{\partial \mathbf{q}_{m0,1}} \\
&= \left(-r_{(i,j)} \frac{\partial\, {}^0\mathbf{n}_0}{\partial \mathbf{q}_{m0,1}}\right) \times \left(-f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}}\, {}^0\mathbf{v}_t\right) + \left(-r_{(i,j)} \cdot {}^0\mathbf{n}_0\right) \times \left(-f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}} \frac{\partial\, {}^0\mathbf{v}_t}{\partial \mathbf{q}_{m0,1}}\right) + \frac{\partial f_c}{\partial \mathbf{q}_{m0,m1}}(...) \\
&= \left(-f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}}\, {}^0\tilde{\mathbf{v}}_t\right)\left(r_{(i,j)} \frac{\partial\, {}^0\mathbf{n}_0}{\partial \mathbf{q}_{m0,1}}\right) + \left(-r_{(i,j)} \cdot {}^0\tilde{\mathbf{n}}_0\right)\left(-f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}} \frac{\partial\, {}^0\mathbf{v}_t}{\partial \mathbf{q}_{m0,1}}\right) + \frac{\partial f_c}{\partial \mathbf{q}_{m0,m1}}(...)
\end{aligned} \tag{5.174}$$

and (note that ${}^0\tilde{\mathbf{n}}_0\ {}^0\mathbf{n}_0 = \mathbf{0}$),

$$\begin{aligned}
\frac{\partial\, {}^0\boldsymbol{\tau}_{f,(i,j)}}{\partial \dot{\mathbf{q}}_{m0}} &= \left(-r_{(i,j)} \cdot {}^0\tilde{\mathbf{n}}_0\right)\left(-f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}} \frac{\partial\, {}^0\mathbf{v}_t}{\partial \dot{\mathbf{q}}_{m0}} - \frac{\partial f_c}{\partial \dot{\mathbf{q}}_{m0}} \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}}\, {}^0\mathbf{v}_t\right) \\
&= \left(r_{(i,j)} \cdot {}^0\tilde{\mathbf{n}}_0\right)\left(f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}}\left(-\,{}^0\mathbf{J}_{pos,m0} + r_i^* \cdot {}^0\tilde{\mathbf{n}}_0\, {}^0\mathbf{J}_{rot,m0}\right) - d_c \delta_f \frac{\mu_s}{v_{\mu,reg}} \cdot {}^0\mathbf{v}_t \otimes ({}^0\mathbf{n}_0\, {}^0\mathbf{J}_{pos,m0})\right)
\end{aligned} \tag{5.175}$$

$$\begin{aligned}
\frac{\partial\, {}^0\boldsymbol{\tau}_{f,(i,j)}}{\partial \dot{\mathbf{q}}_{m1}} &= \left(-r_{(i,j)} \cdot \tilde{\mathbf{n}}_0\right)\left(-f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}} \frac{\partial\, {}^0\mathbf{v}_t}{\partial \dot{\mathbf{q}}_{m1}} - \frac{\partial f_c}{\partial \dot{\mathbf{q}}_{m1}} \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}}\, {}^0\mathbf{v}_t\right) \\
&= \left(r_{(i,j)} \cdot \tilde{\mathbf{n}}_0\right)\left(f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}}\left({}^0\mathbf{J}_{pos,m1} + r_j^* \cdot {}^0\tilde{\mathbf{n}}_0\, {}^0\mathbf{J}_{rot,m1}\right) + d_c \delta_f \frac{\mu_s}{v_{\mu,reg}} \cdot {}^0\mathbf{v}_t \otimes ({}^0\mathbf{n}_0\, {}^0\mathbf{J}_{pos,m1})\right)
\end{aligned} \tag{5.176}$$

5.6.4 Sphere-triangle contact: Equations

The sphere-triangle contact model follows a penalty formulation, using a spring and optional damper to model the unilateral contact behavior. Note that the model can be used for for planar (2D) contact between circles and lineas accordingly, where triangles are placed perpendicular to the $X-Y$ plane, representing lines for the contact with circles. Currently, only linear springs are utilized, however, the user is free to modify the equations in the code to model any nonlinear

The spheres are attached to a position-based marker, as they are the same spheres as in the sphere-sphere contact. The triangles currently may only be attached to a rigid body (but may be attached to three position-based markers in the future). In C++, the sphere attached to marker 0 is denoted as `sphereI` with index i and the triangle attached to a rigid body is denoted as `trigJ` and becomes contact object with index j.

Input parameters for this contact model are

intermediate variables	**symbol**	**description**
sphere i position	${}^0\mathbf{p}_{s,i}$	global position of sphere i
sphere i radius	r_i	radius of sphere i, attached to marker 0
sphere i contact stiffness	k_i	N/m
sphere i contact damping	d_i	N/m
triangle j points	${}^0\mathbf{p}_{t,j,k}$	global position of vertex k of triangle j
triangle j contact stiffness	k_j	N/m
triangle j contact damping	d_j	N/m

friction pairing coefficient	μ_{ij}	the friction coefficient stored in the the friction pairings matrix, resulting from the friction indices of spheres i and j

The main geometrical contact parameters are computed from a function, which computes the projected point on the triangle j with the minimal distance to the sphere's position ${}^0\mathbf{p}_{s,i}$. The triangle j is given by the vertices $({}^0\mathbf{p}_{t,j,0}, {}^0\mathbf{p}_{t,j,1}, {}^0\mathbf{p}_{t,j,2})$. As a result, we obtain the closest (projected) point on the triangle ${}^0\mathbf{p}_p$, which is either inside the triangle or on one of the (closest) edges, potentially also at a vertex. If the projected point is inside the triangle, the flag `inside` becomes true and the closest distance is the normal distance to the plane, otherwise the flag is false. The function for the minimal distance reads

```
[pp, inside] = MinDistTP(pt0, pt1, pt2, ps)
```

We compute the vector between the two points,

$$ {}^0\boldsymbol{\delta}_p = {}^0\mathbf{p}_p - {}^0\mathbf{p}_{s,i} \tag{5.177} $$

and the distance

$$ d = |\,{}^0\boldsymbol{\delta}_p\,| \tag{5.178} $$

Contact is only considered, if $d < r$.

In case of contact, we compute the normalized vector

$$ {}^0\boldsymbol{\delta}_{p0} = \frac{1}{d}\,{}^0\boldsymbol{\delta}_p \tag{5.179} $$

as well as the penetration

$$ \Delta = r_i - d \tag{5.180} $$

In the case of a linear model, the normal contact force results as

$$ f_n = -k \cdot \Delta^p - d \cdot \dot{\Delta} \tag{5.181} $$

in which p is the exponent of the penetration, which is set to 1 by default, but could have different values according to the geometry (however, this has to be adjusted in the C++ part).

For tangential contact and damping, the relative velocity has to be computed. The velocity of the sphere at the contact point reads

$$ {}^0\mathbf{p}_{sp} = \left({}^{0s}\mathbf{A}\ {}^s\boldsymbol{\omega}_s\right) \times \left({}^0\mathbf{p}_p - {}^0\mathbf{p}_{s,i}\right) + {}^0\mathbf{v}_{s,i} \tag{5.182} $$

Here, ${}^0\mathbf{v}_{s,i}$ is the sphere's velocity at the midpoint. The velocity of the rigid body (at which the triangle is attached) at the contact point reads

$$ {}^0\mathbf{p}_{tp} = \left({}^{0r}\mathbf{A}\ {}^r\boldsymbol{\omega}_r\right) \times \left({}^0\mathbf{p}_p - {}^0\mathbf{p}_r\right) + {}^0\mathbf{v}_r \tag{5.183} $$

in which ${}^0\mathbf{v}_r$ is the rigid body's velocity at the reference point and ${}^0\mathbf{p}_r$ is the rigid body's reference point.

From the latter two quantities, we are able to compute the penetration velocity

$$\dot{\Delta} = {}^0\boldsymbol{\delta}_{p0}^{\mathrm{T}} \left({}^0\mathbf{p}_{sp} - {}^0\mathbf{p}_{tp} \right) \tag{5.184}$$

The tangent velocity vector is then computed as

$${}^0\boldsymbol{\delta}_{vt} = \left({}^0\mathbf{p}_{sp} - {}^0\mathbf{p}_{tp} \right) - \dot{\Delta}\, {}^0\boldsymbol{\delta}_{p0} \tag{5.185}$$

The friction model follows again Eq. (5.141), defining the friction force ${}^0\mathbf{f}_f\,(v_t, |f_c|, \mu_s, v_{\mu,reg}, v_t, {}^0\mathbf{v}_t)$, and resulting in the contact force

$${}^0\mathbf{f}_c = f_c \cdot {}^0\boldsymbol{\delta}_{p0} + {}^0\mathbf{f}_f \ . \tag{5.186}$$

The torque due to friction for sphere marker i and for triangle j rigid body results into[17]

$${}^0\boldsymbol{\tau}_{f,i} = {}^0\boldsymbol{\delta}_p \times {}^0\mathbf{f}_f \,, \quad {}^0\boldsymbol{\tau}_{f,j} = \left({}^0\mathbf{p}_p - {}^0\mathbf{p}_r \right) \times {}^0\mathbf{f}_f \ . \tag{5.187}$$

Jacobians for the derivative of contact forces w.r.t. marker positions and rotations only include the main dependencies of normal and tangential forces.

5.6.5 Contact relations for ANCF cable g_i (marker $m0$) and sphere g_j (marker $m1$)

If contact is active, we have two relative axial reference coordinates s_0 and s_1, which define start and end location at the beam, for which the span in between intersects with the circle, see Fig. 5.19. The intersection points are either computed based on the exact 6th order polynomial equations or using a set of linear segments for interpolation. In this model, due to the active set strategy, the reference coordinates spanning $[s_0, s_1]$ are kept fixed, even though that they would change during Newton iterations.

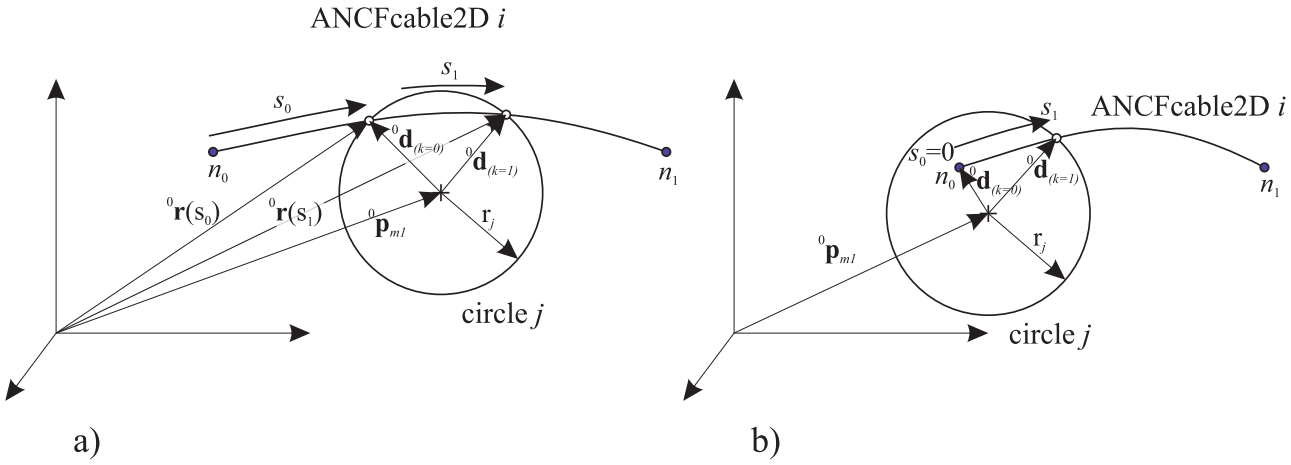


Figure 5.19: Geometrical relations for contact of ANCFCable2D i and circle j with according marker $m1$; case a) shows a cable with nodes n_0 and n_1, partially penetrating at the midspan of the cable; case b) shows the case of a cable where node n_0 is inside the cable.

[17] note that both signs are the same and that ${}^0\mathbf{f}_f$ could be replaced by ${}^0\mathbf{f}_c$

Normal contact and tangential friction forces are then computed based on integrals over the coordinates $[s_0, s_1]$. The integration is performed over n_{ip} integration points $x_k \in [x_{i0}, x_{i1}, \ldots]$. In case of a 3 point Lobatto integration, we chose the integration points

$$x_k \in [s_0, (s_0 + s_1)/2, s_1] \, . \tag{5.188}$$

According weights are

$$w_k \in [1/3, 4/3, 1/3] \, . \tag{5.189}$$

The ANCF cable provides the global position of an integration point $k \in \{i0, i1, \ldots\}$ via

$${}^0\mathbf{r}(x_k) = {}^0\mathbf{S}(x_k)\, \mathbf{q} \tag{5.190}$$

with ANCF shape function matrix $\mathbf{S}$ and current ANCF coordinates $\mathbf{q}$. Note that in the simplified case with linear segments, ${}^0\mathbf{r}(x_k)$ is computed from linear interpolation of the segment which is attached to the cable. The velocity is computed in the same way,

$${}^0\dot{\mathbf{r}}(x_k) = {}^0\mathbf{S}(x_k)\, \dot{\mathbf{q}} \tag{5.191}$$

again using linear interpolation of the velocities along the straight segment, if linear segments are used.

In order to perform the integration of contact forces due to penetration as well as tangential (friction) forces, we iterate over all integration points, and sum up the according generalized forces on the cable and the circle marker object.

The integration factor for integration point k follows from

$$f_k = \frac{s_1 - s_0}{2} w_k \, , \tag{5.192}$$

assuming axial stretch of the cable element being moderately small. The vector ${}^0\mathbf{d}_k$ which points from the center of the circle to the cable (integration) point reads

$${}^0\mathbf{d}_k = {}^0\mathbf{r}(x_k) - {}^0\mathbf{p}_j \tag{5.193}$$

The velocity of the circle at the contact integration point k follows as

$${}^0\mathbf{v}_{c,k} = {}^0\mathbf{v}_j + \left({}^{0,m0}\mathbf{A}\ {}^{m0}\boldsymbol{\omega}_j\right) \times {}^0\mathbf{d}_k \tag{5.194}$$

The distance L between cable and circle center point, gap g and the contact normal vector read

$$L_k = |\,{}^0\mathbf{d}_k\,|, \quad g = L_k - (r + h_{1/2}), \quad {}^0\mathbf{d}_{0,k} = \frac{1}{L_k}\,{}^0\mathbf{d}_k \tag{5.195}$$

with the half height of the ANCF element $h_{1/2}$, which gives additional penetration. Note that this height is added on the side of the circle, which virtually represents a larger circle, behaving slightly different from a cable with thickness h.

The velocity in contact normal direction reads (note that we use the velocity of the circle's center point),

$$v_n = {}^0\mathbf{d}_{0,k} \left({}^0\dot{\mathbf{r}}(x_k) - {}^0\mathbf{v}_j\right) \tag{5.196}$$

The contact force (tension! is always negative) follows in the simplistic case of a linear contact model as

$$f_{c,k} = k_c \cdot g + d \cdot v_n \tag{5.197}$$

with contact stiffness k_c and contact normal damping d_c.
In case of tangential friction, the tangential velocity reads

$$\begin{aligned} \mathbf{v}_{t,k} &= {}^0\dot{\mathbf{r}}(x_k) - {}^0\mathbf{v}_{c,k} - v_n \cdot {}^0\mathbf{d}_{0,k} = {}^0\dot{\mathbf{r}}(x_k) - {}^0\mathbf{v}_j - \left({}^{0,m0}\mathbf{A}\ {}^{m0}\boldsymbol{\omega}_j\right) \times {}^0\mathbf{d}_k - v_n \cdot {}^0\mathbf{d}_{0,k} \\ &= -\left({}^0\mathbf{d}_{0,k} \otimes {}^0\mathbf{d}_{0,k} - \mathbf{I}\right)\left({}^0\dot{\mathbf{r}}(x_k) - {}^0\mathbf{v}_{c,k}\right) + {}^0\tilde{\mathbf{d}}_k\ {}^{0,m0}\mathbf{A}\ {}^{m0}\boldsymbol{\omega}_j \end{aligned} \tag{5.198}$$

and the friction force is computed from Eq. (5.141) using the contact pressure $-f_{c,k}$ from Eq. (5.197), while otherwise ${}^0\mathbf{f}_f = \mathbf{0}$.

The force vector for the contact point for integration point k, including integration weight f_k[18] thus reads

$${}^0\mathbf{f}_k = f_k \cdot \left(f_{c,k} \cdot {}^0\mathbf{d}_{0,k} + {}^0\mathbf{f}_f\right) \tag{5.199}$$

The total force and torque on the circle j is found by summation over all integration points k,

$${}^0\mathbf{f}_{circ} = \sum_k {}^0\mathbf{f}_{circ,k} = \sum_k {}^0\mathbf{f}_k\,, \quad {}^0\mathbf{t}_{circ} = \sum_k {}^0\mathbf{t}_{circ,k} = \sum_k \left(r_j \cdot {}^0\mathbf{d}_{0,k}\right) \times {}^0\mathbf{f}_k \tag{5.200}$$

and the contribution to the generalized forces of the ANCF cable element (with generalized coordinates $\mathbf{q}_{ANCF}$) read

$$\mathbf{f}_{ANCF} = \sum_k \mathbf{f}_{ANCF,k} = \sum_k {}^0\mathbf{S}(x_k)^{\mathrm{T}} \cdot {}^0\mathbf{f}_k \tag{5.201}$$

The generalized LHS forces for marker $m1$ (with generalized coordinates $\mathbf{q}_{m1}$) thus read

$$\mathbf{f}_{m1,LHS} = {}^0\mathbf{J}_{pos,m1}^{\mathrm{T}}\ {}^0\mathbf{f}_{circ} + {}^0\mathbf{J}_{rot,m1}^{\mathrm{T}}\ {}^0\mathbf{t}_{circ}\ . \tag{5.202}$$

The Jacobian matrix for the circle-ANCF contact on position level thus reads[19],

$$\mathbf{J}_c = \begin{bmatrix} \dfrac{\partial\, {}^0\mathbf{f}_{ANCF}}{\partial \mathbf{q}_{ANCF}} & \dfrac{\partial\, {}^0\mathbf{f}_{ANCF}}{\partial \mathbf{q}_{m1}} \\ -\dfrac{\partial\left({}^0\mathbf{J}_{pos,m1}^{\mathrm{T}}\ {}^0\mathbf{f}_{circ} + {}^0\mathbf{J}_{rot,m1}^{\mathrm{T}}\ {}^0\mathbf{t}_{circ}\right)}{\partial \mathbf{q}_{ANCF}} & \dfrac{\partial\left({}^0\mathbf{J}_{pos,m1}^{\mathrm{T}}\ {}^0\mathbf{f}_{circ} + {}^0\mathbf{J}_{rot,m1}^{\mathrm{T}}\ {}^0\mathbf{t}_{circ}\right)}{\partial \mathbf{q}_{m1}} \end{bmatrix} \tag{5.203}$$

and on velocity level, it follows as

$$\mathbf{J}_c = \begin{bmatrix} \dfrac{\partial\, {}^0\mathbf{f}_{ANCF}}{\partial \dot{\mathbf{q}}_{ANCF}} & \dfrac{\partial\, {}^0\mathbf{f}_{ANCF}}{\partial \dot{\mathbf{q}}_{m1}} \\ -\dfrac{\partial\left({}^0\mathbf{J}_{pos,m1}^{\mathrm{T}}\ {}^0\mathbf{f}_{circ} + {}^0\mathbf{J}_{rot,m1}^{\mathrm{T}}\ {}^0\mathbf{t}_{circ}\right)}{\partial \dot{\mathbf{q}}_{ANCF}} & \dfrac{\partial\left({}^0\mathbf{J}_{pos,m1}^{\mathrm{T}}\ {}^0\mathbf{f}_{circ} + {}^0\mathbf{J}_{rot,m1}^{\mathrm{T}}\ {}^0\mathbf{t}_{circ}\right)}{\partial \dot{\mathbf{q}}_{m1}} \end{bmatrix} \tag{5.204}$$

For the calculation of the jacobian, the derivatives of the following terms are needed:

- ${}^0\mathbf{d}_k = {}^0\mathbf{r}(x_k) - {}^0\mathbf{p}_j$:

$$\frac{\partial\, {}^0\mathbf{d}_k}{\partial \mathbf{q}_{ANCF}} = \frac{\partial\, {}^0\mathbf{r}(x_k) - {}^0\mathbf{p}_j}{\partial \mathbf{q}_{ANCF}} = {}^0\mathbf{S}(x_k) \tag{5.205}$$

$$\frac{\partial\, {}^0\mathbf{d}_k}{\partial \mathbf{q}_{m1}} = \frac{\partial\, {}^0\mathbf{r}(x_k) - {}^0\mathbf{p}_j}{\partial \mathbf{q}_{m1}} = -{}^0\mathbf{J}_{pos,m1} \tag{5.206}$$

[18] this is done, because all further terms are proportional to $\mathbf{f}_k$.
[19] terms that are implemented are marked in green; black terms are not implemented or unused

- ${}^0\dot{\mathbf{d}}_k = {}^0\dot{\mathbf{r}}(x_k) - {}^0\dot{\mathbf{v}}_j$:

$$\frac{\partial\, {}^0\dot{\mathbf{d}}_k}{\partial \dot{\mathbf{q}}_{ANCF}} = \frac{\partial\, {}^0\dot{\mathbf{r}}(x_k) - {}^0\mathbf{v}_j}{\partial \mathbf{q}_{ANCF}} = {}^0\mathbf{S}(x_k) \tag{5.207}$$

$$\frac{\partial\, {}^0\dot{\mathbf{d}}_k}{\partial \dot{\mathbf{q}}_{m1}} = \frac{\partial\, {}^0\dot{\mathbf{r}}(x_k) - {}^0\mathbf{v}_j}{\partial \mathbf{q}_{m1}} = -{}^0\mathbf{J}_{pos,m1} \tag{5.208}$$

- $L_k = |\,{}^0\mathbf{d}_k\,| = \left({}^0\mathbf{d}_k^{\mathrm{T}}\ {}^0\mathbf{d}_k\right)^{1/2}$:

$$\frac{\partial |\,{}^0\mathbf{d}_k\,|}{\partial \mathbf{q}_{ANCF,m1}} = \frac{1}{L_k}\left({}^0\mathbf{d}_k^{\mathrm{T}}\,\frac{\partial\, {}^0\mathbf{d}_k}{\partial \mathbf{q}_{ANCF,m1}}\right) = {}^0\mathbf{d}_{0,k}^{\mathrm{T}}\,\frac{\partial\, {}^0\mathbf{d}_k}{\partial \mathbf{q}_{ANCF,m1}} \tag{5.209}$$

- $L_k^{-1} = \left({}^0\mathbf{d}_k^{\mathrm{T}}\ {}^0\mathbf{d}_k\right)^{-1/2}$ (note different sign as in L-term due to $-1/2$):

$$\frac{\partial L_k^{-1}}{\partial \mathbf{q}_{ANCF,m1}} = \frac{\partial \left({}^0\mathbf{d}_k^{\mathrm{T}}\ {}^0\mathbf{d}_k\right)^{-1/2}}{\partial \mathbf{q}_{ANCF,m1}} = -\frac{1}{L_k}\left({}^0\mathbf{d}_{0,k}^{\mathrm{T}}\,\frac{\partial\, {}^0\mathbf{d}_k}{\partial \mathbf{q}_{ANCF,m1}}\right) \tag{5.210}$$

- ${}^0\mathbf{d}_{0,k} = \frac{1}{L_k}\,{}^0\mathbf{d}_k$:

$$\frac{\partial\, {}^0\mathbf{d}_{0,k}}{\partial \mathbf{q}_{ANCF,m1}} = \frac{\partial \frac{1}{L_k}\,{}^0\mathbf{d}_k}{\partial \mathbf{q}_{ANCF,m1}} = -\frac{1}{L_k^2}\,{}^0\mathbf{d}_k \otimes \left({}^0\mathbf{d}_{0,k}^{\mathrm{T}}\,\frac{\partial\, {}^0\mathbf{d}_k}{\partial \mathbf{q}_{ANCF,m1}}\right) + \frac{1}{L_k}\frac{\partial\, {}^0\mathbf{d}_k}{\partial \mathbf{q}_{ANCF,m1}} = \frac{1}{L_k}\left(\mathbf{I} - {}^0\mathbf{d}_{0,k} \otimes {}^0\mathbf{d}_{0,k}\right)\frac{\partial\, {}^0\mathbf{d}_k}{\partial \mathbf{q}_{ANCF,m1}} \tag{5.211}$$

- $g = L_k - (r + h_{1/2})$:

$$\frac{\partial g}{\partial \mathbf{q}_{ANCF,m1}} = \frac{\partial L_k}{\partial \mathbf{q}_{ANCF,m1}} = {}^0\mathbf{d}_{0,k}^{\mathrm{T}}\,\frac{\partial\, {}^0\mathbf{d}_k}{\partial \mathbf{q}_{ANCF,m1}} \tag{5.212}$$

- velocity at circle contact point k: ${}^0\mathbf{v}_{c,k} = {}^0\mathbf{v}_j + \left({}^{0,m1}\mathbf{A}\ {}^{m1}\boldsymbol{\omega}_j\right) \times {}^0\mathbf{d}_k = {}^0\mathbf{v}_j - {}^0\tilde{\mathbf{d}}_k\ {}^{0,m1}\mathbf{A}\ {}^{m1}\boldsymbol{\omega}_j$:

$$\frac{\partial\, {}^0\mathbf{v}_{c,k}}{\partial \dot{\mathbf{q}}_{m1}} = {}^0\mathbf{J}_{pos,m1} - {}^0\tilde{\mathbf{d}}_k\ {}^0\mathbf{J}_{rot,m1} \tag{5.213}$$

- $v_n = {}^0\mathbf{d}_{0,k}\left({}^0\dot{\mathbf{r}}(x_k) - {}^0\mathbf{v}_{c,k}\right)$:[20]

$$\frac{\partial v_n}{\partial \dot{\mathbf{q}}_{ANCF,m1}} = {}^0\mathbf{d}_{0,k}\,\frac{\partial \left({}^0\dot{\mathbf{r}}(x_k) - {}^0\mathbf{v}_{c,k}\right)}{\partial \dot{\mathbf{q}}_{ANCF}} \approx {}^0\mathbf{d}_{0,k}^{\mathrm{T}}\,\frac{\partial\, {}^0\mathbf{d}_k}{\partial \mathbf{q}_{ANCF,m1}} \tag{5.214}$$

- $\mathbf{v}_{t,k} = {}^0\dot{\mathbf{r}}(x_k) - {}^0\mathbf{v}_j - v_n \cdot {}^0\mathbf{d}_{0,k} = \left(\mathbf{I} - {}^0\mathbf{d}_{0,k} \otimes {}^0\mathbf{d}_{0,k}\right)\left({}^0\dot{\mathbf{r}}(x_k) - {}^0\mathbf{v}_j\right) + {}^0\tilde{\mathbf{d}}_k\ {}^0\boldsymbol{\omega}_j$:

$$\frac{\partial \mathbf{v}_{t,k}}{\partial \dot{\mathbf{q}}_{ANCF}} = \left(\mathbf{I} - {}^0\mathbf{d}_{0,k} \otimes {}^0\mathbf{d}_{0,k}\right){}^0\mathbf{S}(x_k) \tag{5.215}$$

$$\frac{\partial \mathbf{v}_{t,k}}{\partial \dot{\mathbf{q}}_{m1}} = -\left(\mathbf{I} - {}^0\mathbf{d}_{0,k} \otimes {}^0\mathbf{d}_{0,k}\right){}^0\mathbf{J}_{pos,m1} + {}^0\tilde{\mathbf{d}}_k\ {}^0\mathbf{J}_{rot,m1} \tag{5.216}$$

[20] the approximate sign is used, because ${}^0\mathbf{v}_{c,k}$ includes a normal component if ANCF cable is not fully tangential, which is not considered here.

- $f_{c,k} = k_c \cdot g + d_c \cdot v_n$:

$$\frac{\partial f_{c,k}}{\partial \mathbf{q}_{ANCF}} = k_c \cdot \frac{\partial g}{\partial \mathbf{q}_{ANCF}} + d_c \cdot \frac{\partial v_n}{\partial \mathbf{q}_{ANCF}} \approx k_c \cdot {}^0\mathbf{d}_{k,0}^{\mathrm{T}}\, {}^0\mathbf{S}(x_k) \tag{5.217}$$

$$\frac{\partial f_{c,k}}{\partial \mathbf{q}_{m1}} = k_c \cdot \frac{\partial p}{\partial \mathbf{q}_{m1}} + d_c \cdot \frac{\partial v_n}{\partial \mathbf{q}_{m1}} \approx -k_c \cdot {}^0\mathbf{d}_{k,0}^{\mathrm{T}}\, {}^0\mathbf{J}_{pos,m1} \tag{5.218}$$

$$\frac{\partial f_{c,k}}{\partial \dot{\mathbf{q}}_{ANCF}} = d_c \cdot \frac{\partial v_n}{\partial \dot{\mathbf{q}}_{ANCF}} = d_c \cdot {}^0\mathbf{d}_{0,k}\, {}^0\mathbf{S}(x_k) \tag{5.219}$$

$$\frac{\partial f_{c,k}}{\partial \dot{\mathbf{q}}_{m1}} = d_c \cdot \frac{\partial v_n}{\partial \dot{\mathbf{q}}_{m1}} = -d_c \cdot {}^0\mathbf{d}_{0,k}\, {}^0\mathbf{J}_{pos,m1} \tag{5.220}$$

The contact force reads (note that because f_c is always negative, the sign of regularization term is negative),

$${}^0\mathbf{f}_k = f_k \cdot \left(f_{c,k} \cdot {}^0\mathbf{d}_{0,k} + \mathbf{f}_f\right) = \begin{cases} f_k \cdot f_{c,k}\left({}^0\mathbf{d}_{0,k} - \frac{\mu_s}{v_{\mu,reg}}\, {}^0\mathbf{v}_t\right), & \text{if} \quad |\,{}^0\mathbf{v}_t\,| < v_{\mu,reg} \\ f_k \cdot \left(f_{c,k} \cdot {}^0\mathbf{d}_{0,k} + \mathbf{f}_f\right), & \text{else with } \mathbf{f}_f = const. \end{cases} \tag{5.221}$$

We introduce a factor δ_f, which is $\delta_f = 1$ in the regularized small velocity state, and in the saturated (constant) friction force we use $\delta_f = 0$. Thus, the derivative of $\mathbf{f}_f$, using $|f_{c,k}| = -f_{c,k}$, reads:

$$\begin{aligned} \frac{\partial \mathbf{f}_f}{\partial \dot{\mathbf{q}}_{ANCF,m1}} &= \delta_f \frac{\mu_s}{v_{\mu,reg}} \frac{\partial(-f_{c,k}\, {}^0\mathbf{v}_t)}{\partial \dot{\mathbf{q}}_{ANCF,m1}} \approx -\delta_f f_{c,k} \frac{\mu_s}{v_{\mu,reg}} \frac{\partial\, {}^0\mathbf{v}_t}{\partial \dot{\mathbf{q}}_{ANCF,m1}} \\ \frac{\partial \mathbf{f}_f}{\partial \dot{\mathbf{q}}_{ANCF}} &\approx -\delta_f f_{c,k} \frac{\mu_s}{v_{\mu,reg}} \left(\mathbf{I} - {}^0\mathbf{d}_{0,k} \otimes {}^0\mathbf{d}_{0,k}\right) {}^0\mathbf{S}(x_k) \\ \frac{\partial \mathbf{f}_f}{\partial \dot{\mathbf{q}}_{m1}} &\approx -\delta_f f_{c,k} \frac{\mu_s}{v_{\mu,reg}} \left(-\left(\mathbf{I} - {}^0\mathbf{d}_{0,k} \otimes {}^0\mathbf{d}_{0,k}\right) {}^0\mathbf{J}_{pos,m1} + {}^0\tilde{\mathbf{d}}_k\, {}^0\mathbf{J}_{rot,m1}\right) \end{aligned} \tag{5.222}$$

The term ${}^0\mathbf{f}_k$ gives:

$$\begin{aligned} \frac{\partial\, {}^0\mathbf{f}_k}{\partial \mathbf{q}_{ANCF,m1}} &= f_k \cdot \left(\left({}^0\mathbf{d}_{0,k} - \frac{\mu_s}{v_{\mu,reg}}\, {}^0\mathbf{v}_t\right) \otimes \frac{\partial f_{c,k}}{\partial \mathbf{q}_{ANCF,m1}} + f_{c,k} \cdot \frac{\partial\, {}^0\mathbf{d}_{0,k}}{\partial \mathbf{q}_{ANCF,m1}} + \frac{\partial\, {}^0\mathbf{f}_f}{\partial \mathbf{q}_{ANCF,m1}}\right) \\ &\approx f_k \cdot k_c \cdot \left(\left({}^0\mathbf{d}_{0,k} - \frac{\mu_s}{v_{\mu,reg}}\, {}^0\mathbf{v}_t\right) \otimes {}^0\mathbf{d}_{0,k} \frac{\partial\, {}^0\mathbf{d}_k}{\partial \mathbf{q}_{ANCF,m1}}\right) \end{aligned} \tag{5.223}$$

and the velocity terms yield

$$\begin{aligned} \frac{\partial\, {}^0\mathbf{f}_k}{\partial \dot{\mathbf{q}}_{ANCF}} &= f_k \cdot \left(d_c \cdot \left({}^0\mathbf{d}_{0,k} - \frac{\mu_s}{v_{\mu,reg}}\, {}^0\mathbf{v}_t\right) \otimes \frac{\partial v_n}{\partial \dot{\mathbf{q}}_{ANCF}} + \frac{\partial\, {}^0\mathbf{f}_f}{\partial \dot{\mathbf{q}}_{ANCF}}\right) \\ &\approx f_k \cdot \left(d_c \cdot \left({}^0\mathbf{d}_{0,k} - \frac{\mu_s}{v_{\mu,reg}}\, {}^0\mathbf{v}_t\right) \otimes {}^0\mathbf{d}_{0,k} - \delta_f f_{c,k} \frac{\mu_s}{v_{\mu,reg}} \left(\mathbf{I} - {}^0\mathbf{d}_{0,k} \otimes {}^0\mathbf{d}_{0,k}\right)\right) \frac{\partial\, {}^0\dot{\mathbf{d}}_k}{\partial \dot{\mathbf{q}}_{ANCF}} \end{aligned} \tag{5.224}$$

$$\begin{aligned} \frac{\partial\, {}^0\mathbf{f}_k}{\partial \dot{\mathbf{q}}_{m1}} &= f_k \cdot \left(d_c \cdot \left({}^0\mathbf{d}_{0,k} - \frac{\mu_s}{v_{\mu,reg}}\, {}^0\mathbf{v}_t\right) \otimes \frac{\partial v_n}{\partial \dot{\mathbf{q}}_{m1}} + \frac{\partial\, {}^0\mathbf{f}_f}{\partial \dot{\mathbf{q}}_{m1}}\right) \\ &\approx f_k \cdot \left(\left(d_c \cdot \left({}^0\mathbf{d}_{0,k} - \frac{\mu_s}{v_{\mu,reg}}\, {}^0\mathbf{v}_t\right) \otimes {}^0\mathbf{d}_{0,k} - \delta_f f_{c,k} \frac{\mu_s}{v_{\mu,reg}} \left(\mathbf{I} - {}^0\mathbf{d}_{0,k} \otimes {}^0\mathbf{d}_{0,k}\right)\right) \frac{\partial\, {}^0\dot{\mathbf{d}}_k}{\partial \dot{\mathbf{q}}_{m1}} \right. \\ &\quad \left. + \delta_f f_{c,k} \frac{\mu_s}{v_{\mu,reg}}\, {}^0\tilde{\mathbf{d}}_k\, {}^0\mathbf{J}_{rot,m1}\right) \end{aligned} \tag{5.225}$$

The single jacobian terms w.r.t. $\mathbf{q}_{ANCF}$ and $\mathbf{q}_{m1}$ may be expressed as

$$\frac{\partial \left({}^{0}\mathbf{J}_{pos,m1}^{\mathrm{T}} \, {}^{0}\mathbf{f}_{circ} + {}^{0}\mathbf{J}_{rot,m1}^{\mathrm{T}} \, {}^{0}\mathbf{t}_{circ} \right)}{\partial \mathbf{q}_{ANCF,m1}} = \frac{\partial \, {}^{0}\mathbf{J}_{pos,m1}^{\mathrm{T}}}{\partial \mathbf{q}_{ANCF,m1}} \, {}^{0}\mathbf{f}_{circ} + {}^{0}\mathbf{J}_{pos,m1}^{\mathrm{T}} \frac{\partial \, {}^{0}\mathbf{f}_{circ}}{\partial \mathbf{q}_{ANCF,m1}} + \frac{\partial \, {}^{0}\mathbf{J}_{rot,m1}^{\mathrm{T}}}{\partial \mathbf{q}_{m0,1}} \, {}^{0}\mathbf{t}_{circ} + {}^{0}\mathbf{J}_{rot,m1}^{\mathrm{T}} \frac{\partial \, {}^{0}\mathbf{t}_{circ}}{\partial \mathbf{q}_{ANCF,m1}} \tag{5.226}$$

Note that similar relations follow for the time derivatives $\frac{\partial}{\partial \dot{\mathbf{q}}_{ANCF,m1}}$ ().
The derivatives of ${}^{0}\mathbf{f}_{ANCF}$, ${}^{0}\mathbf{f}_{circ}$, and

$${}^{0}\mathbf{t}_{circ} = \sum_{k} r_j \cdot {}^{0}\mathbf{d}_{0,k} \times {}^{0}\mathbf{f}_{k} = \sum_{k} r_j \cdot {}^{0}\tilde{\mathbf{d}}_{0,k} \, {}^{0}\mathbf{f}_{k} \tag{5.227}$$

follow from

$$\begin{aligned} \frac{\partial \, {}^{0}\mathbf{f}_{ANCF}}{\partial \mathbf{q}_{ANCF,m1}} &= \sum_{k} {}^{0}\mathbf{S}(x_k)^{\mathrm{T}} \frac{\partial \, {}^{0}\mathbf{f}_{k}}{\partial \mathbf{q}_{ANCF,m1}}, \\ \frac{\partial \, {}^{0}\mathbf{f}_{circ}}{\partial \mathbf{q}_{ANCF,m1}} &= \sum_{k} \frac{\partial \, {}^{0}\mathbf{f}_{k}}{\partial \mathbf{q}_{ANCF,m1}}, \\ \frac{\partial \, {}^{0}\mathbf{t}_{circ}}{\partial \mathbf{q}_{ANCF,m1}} &\approx \sum_{k} \left(r_j \cdot {}^{0}\tilde{\mathbf{d}}_{0,k} \frac{\partial \, {}^{0}\mathbf{f}_{k}}{\partial \mathbf{q}_{ANCF,m1}} - r_j \cdot {}^{0}\tilde{\mathbf{f}}_{k} \frac{\partial \, {}^{0}\mathbf{d}_{0,k}}{\partial \mathbf{q}_{ANCF,m1}} \right) \end{aligned} \tag{5.228}$$

Chapter 6

Python-C++ command interface

This chapter lists the basic interface functions which can be used to set up a Exudyn model in Python.

6.1 General information on Python-C++ interface

This chapter lists the basic interface functions which can be used to set up a Exudyn model in Python. Note that some functions or classes will be used in examples, which are explained in detail later on. In the following, some basic steps and concepts for usage are shown, references to all functions are placed hereafter:

To import the module, just include the Exudyn module in Python:

```
import exudyn as exu
```

For compatibility with examples and other users, we recommend to use the exu abbreviation throughout. In addition, you may work with a convenient interface for your items, therefore also always include:

```
from exudyn.itemInterface import *
```

Note that including exudyn.utilities will cover itemInterface. Also note that from ... import * is not recommended in general and it will not work in certain cases, e.g., if you like to compute on a cluster. However, it greatly simplifies life for smaller models and you may replace imports in your files afterwards by removing the star import.

The general hub to multibody dynamics models is provided by the classes SystemContainer and MainSystem, except for some very basic system functionality (which is inside the Exudyn module).

You can create a new SystemContainer, which is a class that is initialized by assigning a system container to a variable, usually denoted as SC:

```
SC = exu.SystemContainer()
```

Note that creating a second exu.SystemContainer() will be independent of SC and therefore makes no sense if you do not intend to work with two different containers.

To add a MainSystem to system container SC and store as variable mbs, write:

```
mbs = SC.AddSystem()
```

Furthermore, there are a couple of commands available directly in the exudyn module, given in the following subsections. Regarding the **(basic) module access**, functions are related to the exudyn = exu module, see these examples:

```
#  import exudyn module:
import exudyn as exu
#  print detailed exudyn version, Python version (at which it is compiled):
exu.config.Version(addDetails = True)
#  set precision of C++ output to console
exu.config.precision = numberOfDigits
#  turn on/off output to console
exu.config.printToConsole = False
#  invalid index, may depend on compilation settings:
nInvalid = exu.InvalidIndex() #the invalid index, depends on architecture and version
#  run basic demos (without/with graphics):
exu.demos.Demo1()
exu.demos.Demo2()
```

Understanding the usage of functions for python object SystemContainer of the module exudyn, the following examples might help:

```
#import exudyn module:
import exudyn as exu
#  import utilities (includes itemInterface, basicUtilities,
#                  advancedUtilities, rigidBodyUtilities, graphics):
from exudyn.utilities import *
#  create system container and store in SC:
SC = exu.SystemContainer()
#  add a MainSystem (multibody system) to system container SC and store as mbs:
mbs = SC.AddSystem()
#  add a second MainSystem to system container SC and store as mbs2:
mbs2 = SC.AddSystem()
#  print number of systems available:
nSys = SC.NumberOfSystems()
exu.Print(nSys) #or just print(nSys)
#  delete reference to mbs and mbs2 (usually not necessary):
del mbs, mbs2
#  reset system container (mbs becomes invalid):
SC.Reset()
```

If you run a parameter variation (check Examples/parameterVariationExample.py), you may reset or delete the created MainSystem mbs and the SystemContainer SC before creating new instances in order to avoid memory growth.

6.1.1 Item index

Many functions will work with node numbers (`NodeIndex`), object numbers (`ObjectIndex`),marker numbers (`MarkerIndex`) and others. These numbers are special Python objects, which have been introduced in order to avoid mixing up, e.g., node and object numbers.

For example, the command `mbs.AddNode(...)` returns a `NodeIndex`. For these indices, the following rules apply:

`mbs.Add[Node|Object|...](...)` returns a specific `NodeIndex`, `ObjectIndex`, ...

You can create any item index, e.g., using `ni = NodeIndex(42)` or `oi = ObjectIndex(42)`

The benefit of these indices comes as they may not be mixed up, e.g., using an object index instead of a node index.

You can convert any item index, e.g., NodeIndex `ni` into an integer number using `int(ni)` of `ni.GetIndex()`

Still, you can use integers as initialization for item numbers, e.g.:
`mbs.AddObject(MassPoint(nodeNumber=13, ...))`
However, it must be a pure integer type.

You can make integer calculations with such indices, e.g., `oi = 2*ObjectIndex(42)+1` restricing to addition, subtraction and multiplication. Currently, the result of such calculations is a `int` type andoperating on mixed indices is not checked (but may raise exceptions in future).

You can also print item indices, e.g., `print(ni)` as it converts to string by default.

If you are unsure about the type of an index, use `ni.GetTypeString()` to show the index type.

6.1.2 Copying and referencing C++ objects

As a key concept to working with `Exudyn` , most data which is retrieved by C++ interface functions is copied. Experienced Python users may know that it is a key concept to Python to often use references instead of copying, which is sometimes error-prone but offers a computationally efficient behavior. There are only a few very important cases where data is referenced in `Exudyn` , the main ones are `SystemContainer`, `MainSystem`, `VisualizationSettings`, and `SimulationSettings` which are always references to internal C++ classes. The following code snippets and comments should explain this behavior:

```
import copy                         #for real copying
import exudyn as exu
from exudyn.utilities import *
#create system container, referenced from SC:
SC = exu.SystemContainer()
SC2 = SC                            #this will only put a reference to SC
                                    #SC2 and SC represent the SAME C++ object
#add a MainSystem (multibody system):
mbs = SC.AddSystem()                #get reference mbs to C++ system
mbs2=mbs                            #again, mbs2 and mbs refer to the same C++ object
```

```
og = mbs.AddObject(ObjectGround()) #copy data of ObjectGround() into C++
o0 = mbs.GetObject(0)              #get copy of internal data as dictionary

mbsCopy=copy.copy(mbs)             #mbsCopy is now a real copy of mbs; uses pickle;
    experimental!
SC.Append(mbsCopy)                 #this is needed to work with mbsCopy

del o0                             #delete the local dictionary; C++ data not affected
del mbs, mbs2                      #references to mbs deleted (C++ data still available)
del mbsCopy                        #now also copy of mbs destroyed
del SC                             #references to SystemContainer deleted
#at this point, mbs and SC are not available any more (data will be cleaned up by Python)
```

6.1.3 Exceptions and Error Messages

There are several levels of type and argument checks, leading to different types of errors and exceptions. The according error messages are non-unique, because they may be raised in Python modules or in C++, and they may be raised on different levels of the code. Error messages depend on Python version and on your iPython console. Very often the exception may be called `ValueError`, but it mustnot mean that it is a wrong error, but it could also be, e.g., a wrong order of function calls.

As an example, a type conversion error is raised when providing wrong argument types, e.g., try `exu.config.Version('abc')`:

```
Traceback (most recent call last):

File "C:\Users\username\AppData\Local\Temp\ipykernel_24988\2212168679.py", line 1, in <
    module>
    exu.config.Version('abc')

TypeError: Version(): incompatible function arguments. The following argument types are
    supported:
    1. (addDetails: bool = False) -> str

Invoked with: 'abc'
```

Note that your particular error message may be different.

Another error results from internal type and range checking, saying User ERROR, as it is due to a wrong input of the user. For this, we try

```
mbs.AddObject('abc')
```

Which results in an error message similar to:

```
=========================================
User ERROR [file 'C:\Users\username\AppData\Local\Temp\ipykernel_24988\2838049308.py', line
    1]:
```

```
Error in AddObject(...):
Check your python code (negative indices, invalid or undefined parameters, ...)

=========================================

Traceback (most recent call last):

  File "C:\Users\username\AppData\Local\Temp\ipykernel_24988\2838049308.py", line 1, in <
    module>
    mbs.AddObject('abc')

RuntimeError: Exudyn: parsing of Python file terminated due to Python (user) error
```

Finally, there may be system errors. They may be caused due to previous wrong input, but if there is no reason seen, it may be appropriate to report this error on github.com/jgerstmayr/EXUDYN/ .

Be careful in reading and interpreting such error messages. You should **read them from top to bottom**, as the cause may be in the beginning. Often files and line numbers of errors are provided (e.g., if you have a longer script). In the ultimate case, try to comment parts of your code or deactivate items to see where the error comes from. See also section on Trouble shooting and FAQ.

6.2 Exudyn

These are the access functions to the `Exudyn` module. General usage is explained in Section 6.1 and examples are provided there. The C++ module `exudyn` is the root level object linked between Python and C++.In the installed site-packages, the according file is usually denoted as `exudynCPP.pyd` for the regular module, `exudynCPPfast.pyd` for the module without range checks and `exudynCPPnoAVX.pyd` for the module compiled without AVX vector extensions (may depend on your installation).

```
#import exudyn module:
import exudyn as exu
#create systemcontainer and mbs:
SC = exu.SystemContainer()
mbs = SC.AddSystem()
```

function/structure name	description
Help()	Show basic help information
RequireVersion(requiredVersionString)	Checks if the installed version is according to the required version. Major, micro and minor version must agree the required level. This function is defined in the `__init__.py` file EXAMPLE: `exu.RequireVersion("1.0.31")`

SetWriteToFile(filename, flagWriteToFile = True, flagAppend = False, flagFlushAlways = False)	set flag to write (True) or not write to console; default value of flagWriteToFile = False; flagAppend appends output to file, if set True; in order to finalize the file, write `exu.SetWriteToFile(", False)` to close the output file; in case of flagFlushAlways=True, file will be finalized immediately in every print command, but may be slower; **EXAMPLE**:
	`exudyn.config.printToConsole = False #no output to console`
	`exu.SetWriteToFile(filename='testOutput.log', flagWriteToFile=True, flagAppend=False, flagFlushAlways=False)`
	`exu.Print('print this to file')`
	`exu.SetWriteToFile('', False) #terminate writing to file which closes the file`

Print()	this allows printing via exudyn with similar syntax as in Python print(args) except for keyword arguments: exu.Print('test=',42,sep=' ',end='',flush=True); allows to redirect all output to file given by SetWriteToFile(...); does not print to console in case that exudyn.config.printToConsole eis set to False
InvalidIndex()	This function provides the invalid index, which may depend on the kind of 32-bit, 64-bit signed or unsigned integer; e.g. node index or item index in list; currently, the InvalidIndex() gives -1, but it may be changed in future versions, therefore you should use this function
__version__	contains the current version of the Exudyn package
symbolic	the symbolic submodule for creating symbolic variables in Python, see documentation of Symbolic; For details, see Section Symbolic.
config	global config settings, like precision, print behavior, warnings, etc.
config.suppressWarnings	flag to suppress all warnings (default=False)
config.outputPrecision	change precision (number of digits) in C++ and Python output
config.linalgOutputFormatPython	True (default): use Python format for output of vectors and matrices; False: use Matlab format
config.printDelayMilliSeconds	add some delay (in milliSeconds) to printing to console (exudyn.Print), in order to let console (e.g. Spyder) process the output; default = 0
config.printFlushAlways	flush always buffers when using exudyn.Print(...) to write to file or console; this is needed if you are streaming text or showing counters in parameter variation; default=False
config.printToConsole	enables or disables writing to console with exudyn.Print(...); default=True
config.printToFile	flag that shows if writing to file with exudyn.Print(...) is enabled; flag is readonly
config.printFileName	file name for writing to file with exudyn.Print(...); flag is readonly

config.printToFileAppend	flag that shows if append mode is used for writing to file with exudyn.Print(...); flag is readonly
config.Version(addDetails = False)	Get Exudyn built version as string (if addDetails=True, adds more information on compilation Python version, platform, etc.; the Python micro version may differ from that you are working with; AVX2 shows that you are running a AVX2 compiled version)
experimental	Experimental features, not intended for regular users; for available features, see the C++ code class PyExperimental
special	special attributes and functions, such as global (solver) flags or helper functions; not intended for regular users; for available features, see the C++ code class PySpecial
special.InfoStat(writeOutput = True)	Retrieve list of global information on memory allocation and other counts as list:[array_new_counts, array_delete_counts, vector_new_counts, vector_delete_counts, matrix_new_counts, matrix_delete_counts, linkedDataVectorCast_counts]; May be extended in future; if writeOutput==True, it additionally prints the statistics; counts for new vectors and matrices should not depend on numberOfSteps, except for some objects such as ObjectGenericODE2 and for (sensor) output to files; Not available if code is compiled with __FAST_EXUDYN_LINALG flag
special.solver	special solver attributes and functions; not intended for regular users; for available features, see the C++ code class PySpecialSolver
special.solver.timeout	if >= 0, the solver stops after reaching accoring CPU time specified with timeout; makes sense for parameter variation, automatic testing or for long-running simulations; default=-1 (no timeout)
special.solver.multiThreadingLoadBalancing	if True (=default), multithreaded code parts (in particular solver and raytracing) use load balancing, which may give better performance in case of non-equilibrated loads; (mobile) Intel CPUs may perform significantly better without load balancing
variables	this dictionary may be used by the user to store exudyn-wide data in order to avoid global Python variables; usage: exu.variables["myvar"] = 42; can be used in particular to exchange data between different mbs or between packages by importing exudyn.variables wherever needed.
sys	this dictionary is used and reserved by the system, e.g. for testsuite, graphics or system function to store module-wide data in order to avoid global Python variables; the variable exu.sys['renderState'] contains the last render state after SC.renderer.Stop() and can be used for subsequent simulations

6.3 SystemContainer

The SystemContainer is the top level of structures in Exudyn. The container holds all (multibody) systems, solvers and all other data structures for computation and it is the hub for the OpenGL renderer. A SystemContainer is created by `SC = exu.SystemContainer()`, understanding `exu.SystemContainer` as a class like Python's internal list class, creating a list instance with `x=list()`. Currently, only one container shall be used, while multiple containers are possible – e.g. for reasons of different behavior. The SystemContainer contains visualizationSettings, see Section 9.3, which can be edited when pressing the key V in the render window and it holds the renderer substructure to start and stop the renderer, and to interact with the renderer. Regarding the **(basic) module access**, functions are related to the `exudyn = exu` module, see also the introduction of this chapter and this example:

```
import exudyn as exu
#create system container and store by reference in SC:
SC = exu.SystemContainer()
#add MainSystem to SC:
mbs = SC.AddSystem()
```

function/structure name	description
Reset()	delete all multibody systems and reset SystemContainer (including graphics); this also releases SystemContainer from the renderer, which requires SC.renderer.Attach() to be called in order to reconnect to rendering; a safer way is to delete the current SystemContainer and create a new one (SC=SystemContainer())
AddSystem()	add a new computational system
Append(mainSystem)	append an exsiting computational system to the system container; returns the number of MainSystem in system container
NumberOfSystems()	obtain number of multibody systems available in system container
GetSystem(systemNumber)	obtain multibody systems with index from system container
visualizationSettings	this structure is read/writeable and contains visualization settings, which are immediately applied to the rendering window. EXAMPLE: SC = exu.SystemContainer() SC.visualizationSettings.autoFitScene=False
GetDictionary()	[UNDER DEVELOPMENT]: return the dictionary of the system container data, e.g., to copy the system or for pickling
SetDictionary(systemDict)	[UNDER DEVELOPMENT]: set system container data from given dictionary; used for pickling
renderer	The substructure in SystemContainer responsible for rendering (except visualizationSettings)

visualizationSettings	Structure representing the settings for renderer; for details of visualizationSettings see Section Structures and Settings

6.4 Renderer

This is the substructure of SystemContainer that collects rendering and visualization interaction. Rendering is done for a SystemContainer, which may include several MainSystems. Note that visualizationSettings are directly accessible from the SystemContainer.

```
import exudyn as exu
from exudyn.utilities import *
import exudyn.graphics as graphics
SC = exu.SystemContainer()
mbs = SC.AddSystem()
mbs.CreateMassPoint(physicsMass=1)

mbs.Assemble()
SC.visualizationSettings.general.drawWorldBasis = True
SC.renderer.Start()
SC.renderer.DoIdleTasks() #wait until user presses space, etc.
mbs.SolveDynamic()
SC.renderer.Stop()
```

function/structure name	**description**
Start(verbose = 0)	Start OpenGL rendering engine (in separate thread) for visualization of rigid or flexible multibody system; use verbose=1 to output information during OpenGL window creation; verbose=2 produces more output and verbose=3 gives a debug level; some of the information will only be seen in windows command (powershell) windows or linux shell, but not inside iPython of e.g. Spyder
Stop()	Stop OpenGL rendering engine; uses timeout in multi-threading.
IsActive()	returns True if GLFW renderer is available and running; otherwise False
Attach()	Links the SystemContainer to the render engine, such that the changes in the graphics structure drawn upon updates, etc.; done automatically on creation of SystemContainer; return False, if no renderer exists (e.g., compiled without GLFW) or cannot be linked (if other SystemContainer already linked)

Detach()	DEPRECATED; Releases the SystemContainer from the render engine; return True if successfully released, False if no GLFW available or detaching failed
DoIdleTasks(waitSeconds = -1., printPauseMessage = True)	Interrupt further computation until user input (Space, 'Q', Escape-key), representing a PAUSE function; this command runs a loop in the background to have active response of the render window, e.g., to open the visualization dialog or use the right-mouse-button; replaces former SC.WaitForRenderEngineStopFlag() and mbs.WaitForUserToContinue(); call this function in order to interact with Renderer window; use waitSeconds in order to run this idle tasks while animating a model (e.g. waitSeconds=0.04), use waitSeconds=0 without waiting, or use waitSeconds=-1 (default) to wait until window is closed **EXAMPLE**: `SC.renderer.DoIdleTasks()`
ZoomAll()	Send zoom all signal, which will perform zoom all at next redraw request
RedrawAndSaveImage()	Redraw openGL scene and save image (command waits until process is finished)
SendRedrawSignal()	This function is used to send a signal to the renderer that all MainSystems (mbs) shall be redrawn
GetRenderCount()	Returns the number of rendered OpenGL images; can be used to determine if image has been drawn by comparing to previous counter; also shows that first image has been drawn (needed for zoom all)

GetState()	Get dictionary with current render state (openGL zoom, modelview, etc.); will have no effect if GLFW_GRAPHICS is deactivated
	EXAMPLE:
	`SC = exu.SystemContainer()`
	`renderState = SC.renderer.GetState()`
	`print(renderState['zoom'])`

SetState(renderState, waitForRendererFullStartup = True)	Set current render state (openGL zoom, modelview, etc.) with given dictionary; usually, this dictionary has been obtained with GetRenderState; waitForRendererFullStartup is used to wait at startup for the first frame to be drawn (and zoom all to be set), but be be set False in case of performance issues; will have no effect if GLFW_GRAPHICS is deactivated
	EXAMPLE:
	`SC = exu.SystemContainer()`
	`SC.renderer.SetState(renderState)`

GetMouseCoordinates(useOpenGLcoordinates = False)	Get current mouse coordinates as list [x, y]; x and y being floats, as returned by GLFW, measured from top left corner of window; use GetCurrentMouseCoordinates(useOpenGLcoordinates=True) to obtain OpenGLcoordinates of projected plane

GetItemSelection(resetSelection = True)	Get selected item in render state; option to reset selected item afterwards; item is selected in render window by clicking left mouse button; returns [mbs number, ItemType, ItemIndex, depth] where depth is the Z-depth in the current view; note that only items of the categories activated in visualizationSettings.interactive.selectionLeftMouseItemTypes are returned; if itemType == 0 if no item has been selected
materials	GraphicsMaterialList used for raytracer (possibly for OpenGL in future); list can be accessed with [] operator, reset and extended. Note that after Reset() there are at least 10 materials available, which are copied from visualizationSettings.raytracer.materials which are synced continuously

6.5 MainSystem

This is the class which defines a (multibody) system. The MainSystem shall only be created by `SC.AddSystem()`, not with `exu.MainSystem()`, as the latter one would not be linked to a SystemContainer. In some cases, you may use SC.Append(mbs). In C++, there is a MainSystem (the part which links to Python) and a System (computational part). For that reason, the name is MainSystem on the Python side, but it is often just called 'system'. For compatibility, it is recommended to denote the variable holding this system as mbs, the multibody dynamics system. It can be created, visualized and computed. Use the following functions for system manipulation.

```
import exudyn as exu
SC = exu.SystemContainer()
mbs = SC.AddSystem()
```

function/structure name	**description**
Assemble()	assemble items (nodes, bodies, markers, loads, ...) of multibody system; Calls CheckSystemIntegrity(...), AssembleCoordinates(), AssembleLTGLists(), AssembleInitializeSystemCoordinates(), and AssembleSystemInitialize()
AssembleCoordinates()	assemble coordinates: assign computational coordinates to nodes and constraints (algebraic variables)
AssembleLTGLists()	build LTG coordinate lists for objects (used to build global ODE2RHS, MassMatrix, etc. vectors and matrices) and store special object lists (body, connector, constraint, ...)
AssembleInitializeSystemCoordinates()	initialize all system-wide coordinates based on initial values given in nodes

AssembleSystemInitialize()	initialize some system data, e.g., generalContact objects (searchTree, etc.)
Reset()	reset all lists of items (nodes, bodies, markers, loads, ...) and temporary vectors; deallocate memory
GetSystemContainer()	return the systemContainer where the mainSystem (mbs) was created
SendRedrawSignal()	this function is used to send a signal to the renderer that the scene shall be redrawn because the visualization state has been updated
GetRenderEngineStopFlag()	get the current stop simulation flag; True=user wants to stop simulation
SetRenderEngineStopFlag(stopFlag)	set the current stop simulation flag; set to False, in order to continue a previously user-interrupted simulation
ActivateRendering(flag = True)	activate (flag=True) or deactivate (flag=False) rendering for this system

SetPreStepUserFunction(value)	Sets a user function PreStepUserFunction(mbs, t) executed at beginning of every computation step; in normal case return True; return False to stop simulation after current step; set to 0 (integer) in order to erase user function. Note that the time returned is already the end of the step, which allows to compute forces consistently with trapezoidal integrators; for higher order Runge-Kutta methods, step time will be available only in object-user functions.
	EXAMPLE:
	`def PreStepUserFunction(mbs, t):`
	`  print(mbs.systemData.NumberOfNodes())`
	`  if(t>1):`
	`  return False`
	`  return True`
	`mbs.SetPreStepUserFunction(PreStepUserFunction)`

GetPreStepUserFunction(asDict = False)	Returns the preStepUserFunction.

SetPostStepUserFunction(value)	Sets a user function PostStepUserFunction(mbs, t) executed at beginning of every computation step; in normal case return True; return False to stop simulation after current step; set to 0 (integer) in order to erase user function.
	EXAMPLE:
	`def PostStepUserFunction(mbs, t):`
	`  print(mbs.systemData.NumberOfNodes())`
	`  if(t>1):`
	`  return False`
	`  return True`
	`mbs.SetPostStepUserFunction(PostStepUserFunction)`

GetPostStepUserFunction(asDict = False)	Returns the postStepUserFunction.

SetPostNewtonUserFunction(value)	Sets a user function PostNewtonUserFunction(mbs, t) executed after successful Newton iteration in implicit or static solvers and after step update of explicit solvers, but BEFORE PostNewton functions are called by the solver; function returns list [discontinuousError, recommendedStepSize], containing a error of the PostNewtonStep, which is compared to [solver].discontinuous.iterationTolerance. The recommendedStepSize shall be negative, if no recommendation is given, 0 in order to enforce minimum step size or a specific value to which the current step size will be reduced and the step will be repeated; use this function, e.g., to reduce step size after impact or change of data variables; set to 0 (integer) in order to erase user function. Similar described by Flores and Ambrosio, https://doi.org/10.1007/s11044-010-9209-8
	EXAMPLE:
	`def PostNewtonUserFunction(mbs, t):`
	`  if(t>1):`
	`  return [0, 1e-6]`
	`  return [0,0]`
	`mbs.SetPostNewtonUserFunction(PostNewtonUserFunction)`

GetPostNewtonUserFunction(asDict = False)	Returns the postNewtonUserFunction.

SetPreNewtonResidualUserFunction(value)	Sets a user function PreNewtonResidualUserFunction(mbs, t, newtonIt, discontinuousIt) executed prior to computation of the Newton residual in implicit or static solvers. This function returns nothing. The arguments newtonIt and discontinuousIt may be used to distinguish if the call is done at the beginning of a discontinuous iteration (newtonIt=0) or during Newton iterations (newtonIt>0). The typical use case would be to modify objects or loads in every iteration. Note that this user function is not called during Jacobian computation. If needed, the jacobian can be modified with the user function set by SetSystemJacobianUserFunction.
	EXAMPLE:
	`def PreNewtonResidualUserFunction(mbs, t,`
	`newtonIt, discontinuousIt):`
	`  print("t=",t,", newtonIt=",newtonIt,",`
	`discIt=",discontinuousIt)`
	`mbs.SetPreNewtonResidualUserFunction(PreNewtonResidualUserFunc`

GetPreNewtonResidualUserFunction(asDict = False)	Returns the preNewtonResidualUserFunction.

SetSystemJacobianUserFunction(value)	Sets a user function SystemJacobianUserFunction(mbs, t, factorODE2, factorODE2_t, factorODE1) executed after computation of the Newton jacobian of a static solver or an implicit timeintegrator; The function shall return additional terms for the jacobian at RHS, e.g., related to dependencies that are added by the user in the PreNewtonResidualUserFunction; RHS means that for a spring with stiffness K, the jacobian would be -K as it is computed for the RHS, see the RHS-LHS convention. If you like to completely replace the jacobian, consider using the solver's user function SetUserFunctionComputeNewtonJacobian which can be used to replace the jacobian computation; the factors factorODE2, factorODE2_t, factorODE1 must be multiplied with quantities related to ODE2 coordinates (like stiffness terms), ODE2_t velocity coordinates (like damping terms) and ODE1 quantities. The functions returns a MatrixContainer, for which the sparse format is recommended for efficiency reasons.
	EXAMPLE:
	`def SystemJacobianUserFunction(mbs, t, factorODE2, factorODE2_t, factorODE1):`
	`  return MatrixContainer([[factorODE2*10,0],[0,0]])`
	`mbs.SetSystemJacobianUserFunction(SystemJacobianUserFunction)`

GetSystemJacobianUserFunction(asDict = False)	Returns the systemJacobianUserFunction.
AddGeneralContact()	add a new general contact, used to enable efficient contact computation between objects (nodes or markers)
GetGeneralContact(generalContactNumber)	get read/write access to GeneralContact with index generalContactNumber stored in mbs; Examples shows how to access the GeneralContact object added with last AddGeneralContact() command: **EXAMPLE**: `gc=mbs.GetGeneralContact(mbs.NumberOfGeneralContacts()-1)`
DeleteGeneralContact(generalContactNumber)	delete GeneralContact with index generalContactNumber in mbs; other general contacts are resorted (index changes!)
NumberOfGeneralContacts()	Return number of GeneralContact objects in mbs
GetAvailableFactoryItems()	get all available items to be added (nodes, objects, etc.); this is useful in particular in case of additional user elements to check if they are available; the available items are returned as dictionary, containing lists of strings for Node, Object, etc.
GetDictionary()	[UNDER DEVELOPMENT]: return the dictionary of the system data (todo: and state), e.g., to copy the system or for pickling
SetDictionary(systemDict)	[UNDER DEVELOPMENT]: set system data (todo: and state) from given dictionary; used for pickling
__repr__()	return the representation of the system, which can be, e.g., printed **EXAMPLE**: `print(mbs)`

systemIsConsistent	this flag is used by solvers to decide, whether the system is in a solvable state; this flag is set to False as long as Assemble() has not been called; any modification to the system, such as Add...(), Modify...(), etc. will set the flag to False again; this flag can be modified (set to True), if a change of e.g. an object (change of stiffness) or load (change of force) keeps the system consistent, but would normally lead to systemIsConsistent=False
interactiveMode	set this flag to True in order to invoke a Assemble() command in every system modification, e.g. AddNode, AddObject, ModifyNode, ...; this helps that the system can be visualized in interactive mode.
variables	this dictionary may be used by the user to store model-specific data, in order to avoid global Python variables in complex models; mbs.variables["myvar"] = 42
sys	this dictionary is used by exudyn Python libraries, e.g., solvers, to avoid global Python variables
solverSignalJacobianUpdate	this flag is used by solvers to decide, whether the jacobian should be updated; at beginning of simulation and after jacobian computation, this flag is set automatically to False; use this flag to indicate system changes, e.g. during time integration
systemData	Access to SystemData structure; enables access to number of nodes, objects, ... and to (current, initial, reference, ...) state variables (ODE2, AE, Data,...)

6.5.1 MainSystem extensions (create)

This section represents extensions to MainSystem, which are direct calls to Python functions; the 'create' extensions to simplify the creation of multibody systems, such as CreateMassPoint(...); these extensions allow a more intuitive interaction with the MainSystem class, see the following example. For activation, import `exudyn.mainSystemExtensions` or `exudyn.utilities`

```
import exudyn as exu
from exudyn.utilities import *
#alternative: import exudyn.mainSystemExtensions
SC = exu.SystemContainer()
mbs = SC.AddSystem()
#
#create rigid body
b1=mbs.CreateRigidBody(inertia = InertiaCuboid(density=5000, sideLengths=[0.1,0.1,1]),
                       referencePosition = [1,0,0],
                       gravity = [0,0,-9.81])
```

def **CreateGround** (*name*= '', *referencePosition*= [0.,0.,0.], *referenceRotationMatrix*= np.eye(3), *graphicsDataList*= [], *graphicsDataUserFunction*= 0, *show*= True)

– **function description**:

helper function to create a ground object, using arguments of ObjectGround; this function is mainly added for consistency with other mainSystemExtensions

- NOTE that this function is added to MainSystem via Python function MainSystemCreateGround.

– **input**:

name: name string for object

referencePosition: reference coordinates for point node (always a 3D vector, no matter if 2D or 3D mass)

referenceRotationMatrix: reference rotation matrix for rigid body node (always 3D matrix, no matter if 2D or 3D body)

graphicsDataList: list of GraphicsData for optional ground visualization

graphicsDataUserFunction: a user function graphicsDataUserFunction(mbs, itemNumber)->BodyGraphics (list of GraphicsData), which can be used to draw user-defined graphics; this is much slower than regular GraphicsData

color: color of node

show: True: show ground object;

– **output**: ObjectIndex; returns ground object index

– **example**:

```
import exudyn as exu
from exudyn.utilities import * #includes itemInterface and rigidBodyUtilities
import numpy as np
SC = exu.SystemContainer()
mbs = SC.AddSystem()
ground=mbs.CreateGround(referencePosition = [2,0,0],
                        graphicsDataList = [exu.graphics.CheckerBoard(point
 =[0,0,0], normal=[0,1,0],size=4)])
```

For examples on CreateGround see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ballBearningModel.py (Ex), basicTutorial2024.py (Ex), beamTutorial.py (Ex), bicycleIftommBenchmark.py (Ex), bungeeJump.py (Ex), ... , contactCurveExample.py (TM), contactSphereSphereTest.py (TM), contactSphereSphereTestEAPM.py (TM), ...

def **CreateMassPoint** (*name*= '', *referencePosition*= [0.,0.,0.], *initialDisplacement*= [0.,0.,0.], *initialVelocity*= [0.,0.,0.], *physicsMass*= 0, *gravity*= [0.,0.,0.], *graphicsDataList*= [], *drawSize*= -1, *color*= [-1.,-1.,-1.,-1.], *show*= True, *create2D*= False, *returnDict*= False)

– **function description**:

helper function to create 2D or 3D mass point object and node, using arguments as in NodePoint and MassPoint

- NOTE that this function is added to MainSystem via Python function MainSystemCreateMassPoint.

– **input**:

name: name string for object, node is 'Node:'+name

referencePosition: reference coordinates for point node (always a 3D vector, no matter if 2D or 3D mass)

initialDisplacement: initial displacements for point node (always a 3D vector, no matter if 2D or 3D mass)

initialVelocity: initial velocities for point node (always a 3D vector, no matter if 2D or 3D mass)

physicsMass: mass of mass point

gravity: gravity vevtor applied (always a 3D vector, no matter if 2D or 3D mass)

graphicsDataList: list of GraphicsData for optional mass visualization

drawSize: general drawing size of node

color: color of node

show: True: if graphicsData list is empty, node is shown, otherwise body is shown; False: nothing is shown

create2D: if True, create NodePoint2D and MassPoint2D

returnDict: if False, returns object index; if True, returns dict of all information on created object and node

– **output**: Union[dict, ObjectIndex]; returns mass point object index or dict with all data on request (if returnDict=True)

– **example**:

```
import exudyn as exu
from exudyn.utilities import * #includes itemInterface and rigidBodyUtilities
import numpy as np
SC = exu.SystemContainer()
mbs = SC.AddSystem()
b0=mbs.CreateMassPoint(referencePosition = [0,0,0],
                       initialVelocity = [2,5,0],
                       physicsMass = 1, gravity = [0,-9.81,0],
                       drawSize = 0.5, color=exu.graphics.color.blue)
mbs.Assemble()
simulationSettings = exu.SimulationSettings() #takes currently set values or
  default values
simulationSettings.timeIntegration.numberOfSteps = 1000
simulationSettings.timeIntegration.endTime = 2
mbs.SolveDynamic(simulationSettings = simulationSettings)
```

For examples on CreateMassPoint see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- basicTutorial2024.py (Ex), cartesianSpringDamper.py (Ex), cartesianSpringDamperUserFunction.py (Ex), chatGPTupdate.py (Ex), NGsolveOCCgeometry.py (Ex), ... , createFunctionsTest.py (TM), deleteItemsTest.py (TM), loadUserFunctionTest.py (TM), ...

def **CreateRigidBody** (*name*= '', *referencePosition*= [0.,0.,0.], *referenceRotationMatrix*= np.eye(3), *initialVelocity*= [0.,0.,0.], *initialAngularVelocity*= [0.,0.,0.], *initialDisplacement*= None, *initialRotationMatrix*= None, *inertia*= None, *gravity*= [0.,0.,0.], *nodeType*= exudyn.NodeType.RotationEulerParameters, *graphicsDataList*= [], *graphicsDataUserFunction*= 0, *drawSize*= -1, *color*= [-1.,-1.,-1.,-1.], *show*= True, *create2D*= False, *returnDict*= False)

- **function description**:

  helper function to create 3D (or 2D) rigid body object and node; all quantities are global (angular velocity, etc.); use this function to easily create a rigid body; graphics can be directly obtained from inertia object, e.g. in case of cylindrical or cuboid shape

  - NOTE that this function is added to MainSystem via Python function MainSystemCreateRigidBody.

- **input**:

  name: name string for object, node is 'Node:'+name

  referencePosition: reference position vector for rigid body node (always a 3D vector, no matter if 2D or 3D body)

  referenceRotationMatrix: reference rotation matrix for rigid body node (always 3D matrix, no matter if 2D or 3D body)

  initialVelocity: initial translational velocity vector for node (always a 3D vector, no matter if 2D or 3D body)

  initialAngularVelocity: initial angular velocity vector for node (always a 3D vector, no matter if 2D or 3D body)

  initialDisplacement: initial translational displacement vector for node (always a 3D vector, no matter if 2D or 3D body); these displacements are deviations from reference position, e.g. for a finite element node [None: unused]

  initialRotationMatrix: initial rotation provided as matrix (always a 3D matrix, no matter if 2D or 3D body); this rotation is superimposed to reference rotation [None: unused]

  inertia: an instance of class RigidBodyInertia, see rigidBodyUtilities; may also be from derived class (InertiaCuboid, InertiaMassPoint, InertiaCylinder, ...)

  gravity: gravity vevtor applied (always a 3D vector, no matter if 2D or 3D mass)

  graphicsDataList: list of GraphicsData for rigid body visualization; use exudyn.graphics functions to create GraphicsData for basic solids

graphicsDataUserFunction: a user function graphicsDataUserFunction(mbs, itemNumber)->BodyGraphics
(list of GraphicsData), which can be used to draw user-defined graphics; this is much slower than regular GraphicsData

drawSize: general drawing size of node

color: color of node

show: True: if graphicsData list is empty, node is shown, otherwise body is shown; False: nothing is shown

create2D: if True, create NodeRigidBody2D and ObjectRigidBody2D

returnDict: if False, returns object index; if True, returns dict of all information on created object and node

- **output**: Union[dict, ObjectIndex]; returns rigid body object index (or dict with 'nodeNumber', 'objectNumber' and possibly 'loadNumber' and 'markerBodyMass' if returnDict=True)

- **example**:

```
import exudyn as exu
from exudyn.utilities import * #includes itemInterface and rigidBodyUtilities
import numpy as np
SC = exu.SystemContainer()
mbs = SC.AddSystem()
b0 = mbs.CreateRigidBody(inertia = InertiaCuboid(density=5000,
                                                 sideLengths=[1,0.1,0.1]),
                         referencePosition = [1,0,0],
                         initialVelocity = [2,5,0],
                         initialAngularVelocity = [5,0.5,0.7],
                         gravity = [0,-9.81,0],
                         graphicsDataList = [exu.graphics.Brick(size=[1,0.1,0.1],
                                                                color=exu.
  graphics.color.red)])
mbs.Assemble()
simulationSettings = exu.SimulationSettings() #takes currently set values or
  default values
simulationSettings.timeIntegration.numberOfSteps = 1000
simulationSettings.timeIntegration.endTime = 2
mbs.SolveDynamic(simulationSettings = simulationSettings)
```

For examples on CreateRigidBody see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- addPrismaticJoint.py (Ex), addRevoluteJoint.py (Ex), ANCFrotatingCable2D.py (Ex), ballBearningModel.py (Ex), bicycleIftommBenchmark.py (Ex), ... , bricardMechanism.py (TM), carRollingDiscTest.py (TM), complexEigenvaluesTest.py (TM), ...

def **CreateSpringDamper** (*name*= '', *bodyNumbers*= [None, None], *localPosition0*= [0.,0.,0.], *localPosition1*= [0.,0.,0.], *referenceLength*= None, *stiffness*= 0., *damping*= 0., *force*= 0., *velocityOffset*= 0., *springForceUserFunction*= 0, *bodyOrNodeList*= [None, None], *bodyList*= [None, None], *show*= True, *drawSize*= -1, *color*= exudyn.graphics.color.default)

– **function description**:

helper function to create SpringDamper connector, using arguments from ObjectConnectorSpringDamper; similar interface as CreateDistanceConstraint(...), see there for for further information

- NOTE that this function is added to MainSystem via Python function MainSystemCreateSpringDamper.

– **input**:

name: name string for connector; markers get Marker0:name and Marker1:name

bodyNumbers: a list of two body numbers (ObjectIndex) to be connected

localPosition0: local position (as 3D list or numpy array) on body0, if not a node number

localPosition1: local position (as 3D list or numpy array) on body1, if not a node number

referenceLength: if None, length is computed from reference position of bodies or nodes; if not None, this scalar reference length is used for spring

stiffness: scalar stiffness coefficient

damping: scalar damping coefficient

force: scalar additional force applied

velocityOffset: scalar offset: if referenceLength is changed over time, the velocityOffset may be changed accordingly to emulate a reference motion

springForceUserFunction: a user function springForceUserFunction(mbs, t, itemNumber, deltaL, deltaL_t, stiffness, damping, force)->float ; this function replaces the internal connector force computation

bodyOrNodeList: alternative to bodyNumbers; a list of object numbers (with specific localPosition0/1) or node numbers; may alse be mixed types; to use this case, set bodyNumbers = [None,None]

show: if True, connector visualization is drawn

drawSize: general drawing size of connector

color: color of connector

– **output**: ObjectIndex; returns index of newly created object

– **example**:

```
import exudyn as exu
from exudyn.utilities import * #includes itemInterface and rigidBodyUtilities
import numpy as np
SC = exu.SystemContainer()
mbs = SC.AddSystem()
b0 = mbs.CreateMassPoint(referencePosition = [2,0,0],
                         initialVelocity = [2,5,0],
                         physicsMass = 1, gravity = [0,-9.81,0],
```

```
                                drawSize = 0.5, color=exu.graphics.color.blue)
oGround = mbs.AddObject(ObjectGround())
#add vertical spring
oSD = mbs.CreateSpringDamper(bodyNumbers=[oGround, b0],
                             localPosition0=[2,1,0],
                             localPosition1=[0,0,0],
                             stiffness=1e4, damping=1e2,
                             drawSize=0.2)
mbs.Assemble()
simulationSettings = exu.SimulationSettings() #takes currently set values or
  default values
simulationSettings.timeIntegration.numberOfSteps = 1000
simulationSettings.timeIntegration.endTime = 2
SC.visualizationSettings.nodes.drawNodesAsPoint=False
mbs.SolveDynamic(simulationSettings = simulationSettings)
```

For examples on CreateSpringDamper see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ballBearningModel.py (Ex), basicTutorial2024.py (Ex), camFollowerExample.py (Ex), chatGPTupdate.py (Ex), contactCurveWithLongCurve.py (Ex), ... , createFunctionsTest.py (TM), loadUserFunctionTest.py (TM), mainSystemExtensionsTests.py (TM), ...

def **CreateCartesianSpringDamper** (*name*= '', *bodyNumbers*= [None, None], *localPosition0*= [0.,0.,0.], *localPosition1*= [0.,0.,0.], *stiffness*= [0.,0.,0.], *damping*= [0.,0.,0.], *offset*= [0.,0.,0.], *springForceUserFunction*= 0, *bodyOrNodeList*= [None, None], *bodyList*= [None, None], *show*= True, *drawSize*= -1, *color*= exudyn.graphics.color.default)

- **function description**:

  helper function to create CartesianSpringDamper connector, using arguments from ObjectConnectorCartesianSpringDamper

  - NOTE that this function is added to MainSystem via Python function MainSystemCreateCartesianSpringDamper.

- **input**:

  name: name string for connector; markers get Marker0:name and Marker1:name

  bodyNumbers: a list of two body numbers (ObjectIndex) to be connected

  localPosition0: local position (as 3D list or numpy array) on body0, if not a node number

  localPosition1: local position (as 3D list or numpy array) on body1, if not a node number

  stiffness: stiffness coefficients (as 3D list or numpy array)

  damping: damping coefficients (as 3D list or numpy array)

  offset: offset vector (as 3D list or numpy array)

  springForceUserFunction: a user function springForceUserFunction(mbs, t, itemNumber, displacement, velocity, stiffness, damping, offset)->[float,float,float] ; this function replaces the internal connector force computation

bodyOrNodeList: alternative to bodyNumbers; a list of object numbers (with specific localPosition0/1) or node numbers; may alse be mixed types; to use this case, set bodyNumbers = [None,None]

show: if True, connector visualization is drawn

drawSize: general drawing size of connector

color: color of connector

- **output**: ObjectIndex; returns index of newly created object
- **example**:

```
import exudyn as exu
from exudyn.utilities import * #includes itemInterface and rigidBodyUtilities
import numpy as np
SC = exu.SystemContainer()
mbs = SC.AddSystem()
b0 = mbs.CreateMassPoint(referencePosition = [7,0,0],
                         physicsMass = 1, gravity = [0,-9.81,0],
                         drawSize = 0.5, color=exu.graphics.color.blue)
oGround = mbs.AddObject(ObjectGround())
oSD = mbs.CreateCartesianSpringDamper(bodyNumbers=[oGround, b0],
                              localPosition0=[7.5,1,0],
                              localPosition1=[0,0,0],
                              stiffness=[200,2000,0], damping=[2,20,0],
                              drawSize=0.2)
mbs.Assemble()
simulationSettings = exu.SimulationSettings() #takes currently set values or
  default values
simulationSettings.timeIntegration.numberOfSteps = 1000
simulationSettings.timeIntegration.endTime = 2
SC.visualizationSettings.nodes.drawNodesAsPoint=False
mbs.SolveDynamic(simulationSettings = simulationSettings)
```

For examples on CreateCartesianSpringDamper see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- cartesianSpringDamper.py (Ex), cartesianSpringDamperUserFunction.py (Ex), chatGPTupdate.py (Ex), complexEigenvaluesTest.py (TM), computeODE2AEeigenvaluesTest.py (TM), createFunctionsTest.py (TM), mainSystemExtensionsTests.py (TM), mainSystemUserFunctionsTest.py (TM), ...

def **CreateRigidBodySpringDamper** (*name*= '', *bodyNumbers*= [None, None], *localPosition0*= [0.,0.,0.], *localPosition1*= [0.,0.,0.], *stiffness*= np.zeros((6,6)), *damping*= np.zeros((6,6)), *offset*= [0.,0.,0.,0.,0.,0.], *rotationMatrixJoint*= np.eye(3), *useGlobalFrame*= True, *intrinsicFormulation*= True, *springForceTorqueUserFunction*= 0, *postNewtonStepUserFunction*= 0, *bodyOrNodeList*= [None, None], *bodyList*= [None, None], *show*= True, *drawSize*= -1, *color*= exudyn.graphics.color.default)

- **function description**:

helper function to create RigidBodySpringDamper connector, using arguments from ObjectConnectorRigidBodySpringDamper, see there for the full documentation

- NOTE that this function is added to MainSystem via Python function MainSystemCreateRigidBodySpringDamper.

– **input**:

name: name string for connector; markers get Marker0:name and Marker1:name

bodyNumbers: a list of two body numbers (ObjectIndex) to be connected

localPosition0: local position (as 3D list or numpy array) on body0, if not a node number

localPosition1: local position (as 3D list or numpy array) on body1, if not a node number

stiffness: stiffness coefficients (as 6D matrix or numpy array)

damping: damping coefficients (as 6D matrix or numpy array)

offset: offset vector (as 6D list or numpy array)

rotationMatrixJoint: additional rotation matrix; in case useGlobalFrame=False, it transforms body0/node0 local frame to joint frame; if useGlobalFrame=True, it transforms global frame to joint frame

useGlobalFrame: if False, the rotationMatrixJoint is defined in the local coordinate system of body0

intrinsicFormulation: if True, uses intrinsic formulation of Maserati and Morandini, which uses matrix logarithm and is independent of order of markers (preferred formulation); otherwise, Tait-Bryan angles are used for computation of torque, see documentation

springForceTorqueUserFunction: a user function springForceTorqueUserFunction(mbs, t, itemNumber, displacement, rotation, velocity, angularVelocity, stiffness, damping, rotJ0, rotJ1, offset)->[float,float,float, float,float,float] ; this function replaces the internal connector force / torque computation

postNewtonStepUserFunction: a special user function postNewtonStepUserFunction(mbs, t, Index itemIndex, dataCoordinates, displacement, rotation, velocity, angularVelocity, stiffness, damping, rotJ0, rotJ1, offset)->[PNerror, recommendedStepSize, data[0], data[1], ...] ; for details, see RigidBodySpringDamper for full docu

bodyOrNodeList: alternative to bodyNumbers; a list of object numbers (with specific localPosition0/1) or node numbers; may alse be mixed types; to use this case, set bodyNumbers = [None,None]

show: if True, connector visualization is drawn

drawSize: general drawing size of connector

color: color of connector

– **output**: ObjectIndex; returns index of newly created object

– **example**:

```
#coming later
```

For examples on CreateRigidBodySpringDamper see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- bricardMechanism.py (TM), rigidBodySpringDamperIntrinsic.py (TM)

def **CreateTorsionalSpringDamper** (*name*= '', *bodyNumbers*= [None, None], *position*= [0.,0.,0.], *axis*= [0.,0.,0.], *stiffness*= 0., *damping*= 0., *offset*= 0., *velocityOffset*= 0., *torque*= 0., *useGlobalFrame*= True, *springTorqueUserFunction*= 0, *unlimitedRotations*= True, *show*= True, *drawSize*= -1, *color*= exudyn.graphics.color.default)

- **function description**:

  helper function to create TorsionalSpringDamper connector, using arguments from Object-ConnectorTorsionalSpringDamper, see there for the full documentation

  - NOTE that this function is added to MainSystem via Python function MainSystemCreate-TorsionalSpringDamper.

- **input**:

  name: name string for connector; markers get Marker0:name and Marker1:name

  bodyNumbers: a list of two body numbers (ObjectIndex) to be connected

  position: a 3D vector as list or np.array: if useGlobalFrame=True it describes the global position of the joint in reference configuration; else: local position in body0

  axis: a 3D vector as list or np.array containing the axis around which the spring acts, either in local body0 coordinates (useGlobalFrame=False), or in global reference configuration (useGlobalFrame=True)

  stiffness: scalar stiffness of spring

  damping: scalar damping added to spring

  offset: scalar offset, which can be used to realize a P-controlled actuator

  velocityOffset: scalar velocity offset, which can be used to realize a D-controlled actuator

  torque: additional constant torque added to spring-damper, acting between the two bodies

  useGlobalFrame: if False, the position and axis vectors are defined in the local coordinate system of body0, otherwise in global (reference) coordinates

  springTorqueUserFunction : a user function springTorqueUserFunction(mbs, t, itemNumber, rotation, angularVelocity, stiffness, damping, offset)->float ; this function replaces the internal connector torque computation

  unlimitedRotations: if True, an additional generic data node is added to enable measurement of rotations beyond +/- pi; this also allows the spring to cope with multiple turns.

  show: if True, connector visualization is drawn

  drawSize: general drawing size of connector

color: color of connector

- **output**: ObjectIndex; returns index of newly created object
- **example**:

```
#coming later
```

For examples on CreateTorsionalSpringDamper see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- createFunctionsTest.py (TM)

def **CreateRevoluteJoint** (*name*= '', *bodyNumbers*= [None, None], *position*= [], *axis*= [], *useGlobalFrame*= True, *show*= True, *axisRadius*= 0.1, *axisLength*= 0.4, *color*= exudyn.graphics.color.default)

- **function description**:

  Create revolute joint between two bodies; definition of joint position and axis in global coordinates (alternatively in body0 local coordinates) for reference configuration of bodies; all markers, markerRotation and other quantities are automatically computed

  - NOTE that this function is added to MainSystem via Python function MainSystemCreateRevoluteJoint.

- **input**:

  name: name string for joint; markers get Marker0:name and Marker1:name

  bodyNumbers: a list of object numbers for body0 and body1; must be rigid body or ground object

  position: a 3D vector as list or np.array: if useGlobalFrame=True it describes the global position of the joint in reference configuration; else: local position in body0

  axis: a 3D vector as list or np.array containing the joint axis either in local body0 coordinates (useGlobalFrame=False), or in global reference configuration (useGlobalFrame=True)

  useGlobalFrame: if False, the position and axis vectors are defined in the local coordinate system of body0, otherwise in global (reference) coordinates

  show: if True, connector visualization is drawn

  axisRadius: radius of axis for connector graphical representation

  axisLength: length of axis for connector graphical representation

  color: color of connector

- **output**: ObjectIndex; returns index of created joint
- **example**:

```
import exudyn as exu
from exudyn.utilities import * #includes itemInterface and rigidBodyUtilities
import numpy as np
SC = exu.SystemContainer()
mbs = SC.AddSystem()
b0 = mbs.CreateRigidBody(inertia = InertiaCuboid(density=5000,
                                                 sideLengths=[1,0.1,0.1]),
                         referencePosition = [3,0,0],
                         gravity = [0,-9.81,0],
                         graphicsDataList = [exu.graphics.Brick(size=[1,0.1,0.1],
                                                                color=exu.
  graphics.color.steelblue)])
oGround = mbs.AddObject(ObjectGround())
mbs.CreateRevoluteJoint(bodyNumbers=[oGround, b0], position=[2.5,0,0], axis
  =[0,0,1],
                        useGlobalFrame=True, axisRadius=0.02, axisLength=0.14)
mbs.Assemble()
simulationSettings = exu.SimulationSettings() #takes currently set values or
  default values
simulationSettings.timeIntegration.numberOfSteps = 1000
simulationSettings.timeIntegration.endTime = 2
mbs.SolveDynamic(simulationSettings = simulationSettings)
```

For examples on CreateRevoluteJoint see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- addRevoluteJoint.py (Ex), ballBearningModel.py (Ex), bicycleIftommBenchmark.py (Ex), chatGPTupdate.py (Ex), chatGPTupdate2.py (Ex), ... , bricardMechanism.py (TM), createFunctionsTest.py (TM), createRollingDiscPenaltyTest.py (TM), ...

def **CreatePrismaticJoint** (*name*= '', *bodyNumbers*= [None, None], *position*= [], *axis*= [], *useGlobalFrame*= True, *show*= True, *axisRadius*= 0.1, *axisLength*= 0.4, *color*= exudyn.graphics.color.default)

- **function description**:

  Create prismatic joint between two bodies; definition of joint position and axis in global coordinates (alternatively in body0 local coordinates) for reference configuration of bodies; all markers, markerRotation and other quantities are automatically computed

  - NOTE that this function is added to MainSystem via Python function MainSystemCreatePrismaticJoint.

- **input**:

  name: name string for joint; markers get Marker0:name and Marker1:name

  bodyNumbers: a list of object numbers for body0 and body1; must be rigid body or ground object

position: a 3D vector as list or np.array: if useGlobalFrame=True it describes the global position of the joint in reference configuration; else: local position in body0

axis: a 3D vector as list or np.array containing the joint axis either in local body0 coordinates (useGlobalFrame=False), or in global reference configuration (useGlobalFrame=True)

useGlobalFrame: if False, the position and axis vectors are defined in the local coordinate system of body0, otherwise in global (reference) coordinates

show: if True, connector visualization is drawn

axisRadius: radius of axis for connector graphical representation

axisLength: length of axis for connector graphical representation

color: color of connector

- **output**: ObjectIndex; returns index of created joint
- **example**:

```
import exudyn as exu
from exudyn.utilities import * #includes itemInterface and rigidBodyUtilities
import numpy as np
SC = exu.SystemContainer()
mbs = SC.AddSystem()
b0 = mbs.CreateRigidBody(inertia = InertiaCuboid(density=5000,
                                                 sideLengths=[1,0.1,0.1]),
                         referencePosition = [4,0,0],
                         initialVelocity = [0,4,0],
                         gravity = [0,-9.81,0],
                         graphicsDataList = [exu.graphics.Brick(size=[1,0.1,0.1],
                                                                color=exu.
  graphics.color.steelblue)])
oGround = mbs.AddObject(ObjectGround())
mbs.CreatePrismaticJoint(bodyNumbers=[oGround, b0], position=[3.5,0,0], axis
  =[0,1,0],
                         useGlobalFrame=True, axisRadius=0.02, axisLength=1)
mbs.Assemble()
simulationSettings = exu.SimulationSettings() #takes currently set values or
  default values
simulationSettings.timeIntegration.numberOfSteps = 1000
simulationSettings.timeIntegration.endTime = 2
mbs.SolveDynamic(simulationSettings = simulationSettings)
```

For examples on CreatePrismaticJoint see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- addPrismaticJoint.py (Ex), chatGPTupdate.py (Ex), chatGPTupdate2.py (Ex), createFunctionsTest.py (TM), mainSystemExtensionsTests.py (TM), pickleCopyMbs.py (TM)

def **CreateSphericalJoint** (*name*= '', *bodyNumbers*= [None, None], *position*= [], *constrainedAxes*= [1,1,1], *useGlobalFrame*= True, *show*= True, *jointRadius*= 0.1, *color*= exudyn.graphics.color.default)

– **function description**:

Create spherical joint between two bodies; definition of joint position in global coordinates (alternatively in body0 local coordinates) for reference configuration of bodies; all markers are automatically computed

- NOTE that this function is added to MainSystem via Python function MainSystemCreateSphericalJoint.

– **input**:

name: name string for joint; markers get Marker0:name and Marker1:name

bodyNumbers: a list of object numbers for body0 and body1; must be mass point, rigid body or ground object

position: a 3D vector as list or np.array: if useGlobalFrame=True it describes the global position of the joint in reference configuration; else: local position in body0

constrainedAxes: flags, which determines which (global) translation axes are constrained; each entry may only be 0 (=free) axis or 1 (=constrained axis)

useGlobalFrame: if False, the point and axis vectors are defined in the local coordinate system of body0

show: if True, connector visualization is drawn

jointRadius: radius of sphere for connector graphical representation

color: color of connector

– **output**: ObjectIndex; returns index of created joint

– **example**:

```
import exudyn as exu
from exudyn.utilities import * #includes itemInterface and rigidBodyUtilities
import numpy as np
SC = exu.SystemContainer()
mbs = SC.AddSystem()
b0 = mbs.CreateRigidBody(inertia = InertiaCuboid(density=5000,
                                                 sideLengths=[1,0.1,0.1]),
                         referencePosition = [5,0,0],
                         initialAngularVelocity = [5,0,0],
                         gravity = [0,-9.81,0],
                         graphicsDataList = [exu.graphics.Brick(size=[1,0.1,0.1],
                                                                color=exu.
  graphics.color.orange)])
oGround = mbs.AddObject(ObjectGround())
mbs.CreateSphericalJoint(bodyNumbers=[oGround, b0], position=[5.5,0,0],
                         useGlobalFrame=True, jointRadius=0.06)
mbs.Assemble()
simulationSettings = exu.SimulationSettings() #takes currently set values or
  default values
```

```
simulationSettings.timeIntegration.numberOfSteps = 1000
simulationSettings.timeIntegration.endTime = 2
mbs.SolveDynamic(simulationSettings = simulationSettings)
```

For examples on CreateSphericalJoint see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- newtonsCradle.py (Ex), createFunctionsTest.py (TM), driveTrainTest.py (TM), mainSystemExtensionsTests.py (TM)

def **CreateGenericJoint** (*name*= '', *bodyNumbers*= [None, None], *position*= [], *rotationMatrixAxes*= np.eye(3), *constrainedAxes*= [1,1,1, 1,1,1], *useGlobalFrame*= True, *offsetUserFunction*= 0, *offsetUserFunction_t*= 0, *show*= True, *axesRadius*= 0.1, *axesLength*= 0.4, *color*= exudyn.graphics.color.default)

- **function description**:

  Create generic joint between two bodies; definition of joint position (position) and axes (rotationMatrixAxes) in global coordinates (useGlobalFrame=True) or in local coordinates of body0 (useGlobalFrame=False), where rotationMatrixAxes is an additional rotation to body0; all markers, markerRotation and other quantities are automatically computed

  - NOTE that this function is added to MainSystem via Python function MainSystemCreateGenericJoint.

- **input**:

  name: name string for joint; markers get Marker0:name and Marker1:name

  bodyNumber0: a object number for body0, must be rigid body or ground object

  bodyNumber1: a object number for body1, must be rigid body or ground object

  position: a 3D vector as list or np.array: if useGlobalFrame=True it describes the global position of the joint in reference configuration; else: local position in body0

  rotationMatrixAxes: rotation matrix which defines orientation of constrainedAxes; if useGlobalFrame, this rotation matrix is global, else the rotation matrix is post-multiplied with the rotation of body0, identical with rotationMarker0 in the joint

  constrainedAxes: flag, which determines which translation (0,1,2) and rotation (3,4,5) axes are constrained; each entry may only be 0 (=free) axis or 1 (=constrained axis); ALL constrained Axes are defined relative to reference rotation of body0 times rotation0

  useGlobalFrame: if False, the position is defined in the local coordinate system of body0, otherwise it is defined in global coordinates

  offsetUserFunction: a user function offsetUserFunction(mbs, t, itemNumber, offsetUserFunctionParameters)
  >float ; this function replaces the internal (constant) by a user-defined offset. This allows to realize rheonomic joints and allows kinematic simulation

  offsetUserFunction_t: a user function offsetUserFunction_t(mbs, t, itemNumber, offsetUserFunctionParamet
  >float ; this function replaces the internal (constant) by a user-defined offset velocity; this function is used instead of offsetUserFunction, if velocityLevel (index2) time integration

show: if True, connector visualization is drawn

axesRadius: radius of axes for connector graphical representation

axesLength: length of axes for connector graphical representation

color: color of connector

- **output**: ObjectIndex; returns index of created joint
- **example**:

```
import exudyn as exu
from exudyn.utilities import * #includes itemInterface and rigidBodyUtilities
import numpy as np
SC = exu.SystemContainer()
mbs = SC.AddSystem()
b0 = mbs.CreateRigidBody(inertia = InertiaCuboid(density=5000,
                                                 sideLengths=[1,0.1,0.1]),
                         referencePosition = [6,0,0],
                         initialAngularVelocity = [0,8,0],
                         gravity = [0,-9.81,0],
                         graphicsDataList = [exu.graphics.Brick(size=[1,0.1,0.1],
                                                                color=exu.
  graphics.color.orange)])
oGround = mbs.AddObject(ObjectGround())
mbs.CreateGenericJoint(bodyNumbers=[oGround, b0], position=[5.5,0,0],
                       constrainedAxes=[1,1,1, 1,0,0],
                       rotationMatrixAxes=RotationMatrixX(0.125*pi), #tilt axes
                       useGlobalFrame=True, axesRadius=0.02, axesLength=0.2)
mbs.Assemble()
simulationSettings = exu.SimulationSettings() #takes currently set values or
  default values
simulationSettings.timeIntegration.numberOfSteps = 1000
simulationSettings.timeIntegration.endTime = 2
mbs.SolveDynamic(simulationSettings = simulationSettings)
```

For examples on CreateGenericJoint see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ballBearningModel.py (Ex), bungeeJump.py (Ex), pistonEngine.py (Ex), universalJoint.py (Ex), bricardMechanism.py (TM), complexEigenvaluesTest.py (TM), computeODE2AEeigenvaluesTest.py (TM), driveTrainTest.py (TM), ...

def **CreateDistanceConstraint** (*name*= '', *bodyNumbers*= [None, None], *localPosition0*= [0.,0.,0.], *localPosition1*= [0.,0.,0.], *distance*= None, *bodyOrNodeList*= [None, None], *bodyList*= [None, None], *show*= True, *drawSize*= -1., *color*= exudyn.graphics.color.default)

- **function description**:

  Create distance joint between two bodies; definition of joint positions in local coordinates of bodies or nodes; if distance=None, it is computed automatically from reference length; all markers are automatically computed

- NOTE that this function is added to MainSystem via Python function MainSystemCreateDistanceConstraint.

– **input**:

name: name string for joint; markers get Marker0:name and Marker1:name

bodyNumbers: a list of two body numbers (ObjectIndex) to be constrained

localPosition0: local position (as 3D list or numpy array) on body0, if not a node number

localPosition1: local position (as 3D list or numpy array) on body1, if not a node number

distance: if None, distance is computed from reference position of bodies or nodes; if not None, this distance is prescribed between the two positions; if distance = 0, it will create a SphericalJoint as this case is not possible with a DistanceConstraint

bodyOrNodeList: alternative to bodyNumbers; a list of object numbers (with specific localPosition0/1) or node numbers; may alse be mixed types; to use this case, set bodyNumbers = [None,None]

show: if True, connector visualization is drawn

drawSize: general drawing size of node

color: color of connector

– **output**: ObjectIndex; returns index of created joint

– **example**:

```
import exudyn as exu
from exudyn.utilities import * #includes itemInterface and rigidBodyUtilities
import numpy as np
SC = exu.SystemContainer()
mbs = SC.AddSystem()
b0 = mbs.CreateRigidBody(inertia = InertiaCuboid(density=5000,
                                                 sideLengths=[1,0.1,0.1]),
                         referencePosition = [6,0,0],
                         gravity = [0,-9.81,0],
                         graphicsDataList = [exu.graphics.Brick(size=[1,0.1,0.1],
                                                                color=exu.
  graphics.color.orange)])
m1 = mbs.CreateMassPoint(referencePosition=[5.5,-1,0],
                         physicsMass=1, drawSize = 0.2)
n1 = mbs.GetObject(m1)['nodeNumber']
oGround = mbs.AddObject(ObjectGround())
mbs.CreateDistanceConstraint(bodyNumbers=[oGround, b0],
                             localPosition0 = [6.5,1,0],
                             localPosition1 = [0.5,0,0],
                             distance=None, #automatically computed
                             drawSize=0.06)
mbs.CreateDistanceConstraint(bodyOrNodeList=[b0, n1],
```

```
                                    localPosition0 = [-0.5,0,0],
                                    localPosition1 = [0.,0.,0.], #must be [0,0,0] for Node
                                    distance=None, #automatically computed
                                    drawSize=0.06)
mbs.Assemble()
simulationSettings = exu.SimulationSettings() #takes currently set values or
  default values
simulationSettings.timeIntegration.numberOfSteps = 1000
simulationSettings.timeIntegration.endTime = 2
mbs.SolveDynamic(simulationSettings = simulationSettings)
```

For examples on CreateDistanceConstraint see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- chatGPTupdate.py (Ex), chatGPTupdate2.py (Ex), newtonsCradle.py (Ex), createFunctionsTest.py (TM), deleteItemsTest.py (TM), mainSystemExtensionsTests.py (TM), taskmanagerTest.py (TM)

def **CreateCoordinateConstraint** (*name*= '', *bodyNumbers*= [None, None], *coordinates*= [None, None], *offset*= 0., *factorValue1*= 1., *velocityLevel*= False, *offsetUserFunction*= 0, *offsetUserFunction_t*= 0, *show*= True, *drawSize*= -1., *color*= exudyn.graphics.color.default)

- **function description**:

  Create coordinate constraint for two bodies, or body on ground; markers and NodePointGround are automatically created when needed

  - NOTE that this function is added to MainSystem via Python function MainSystemCreateCoordinateConstraint.

- **input**:

  name: name string for joint; markers get Marker0:name and Marker1:name

  bodyNumbers: a list of two body numbers (ObjectIndex) to be constrained

  coordinates: a list of two coordinates for the respective bodies (in case of ground, it shall be None)

  offset: an fixed offset between the two coordinate values

  factorValue1: an additional factor multiplied with coordinate value1 used in algebraic equation, to enable (e.g. gear) ratio between coordinates

  velocityLevel: If true: connector constrains velocities (only works for ODE2 coordinates!); offset is used between velocities; if True, the offsetUserFunction_t is considered and offsetUserFunction is ignored

  offsetUserFunction: a Python function which defines the time-dependent offset; see description in CoordinateConstraint

  offsetUserFunction_t: time derivative of offsetUserFunction; needed for velocity level constraints; see description in CoordinateConstraint

show: if True, connector visualization is drawn

drawSize: general drawing size of node

color: color of connector

- **output**: ObjectIndex; returns index of created joint
- **example**:

```
import exudyn as exu
from exudyn.utilities import * #includes itemInterface and rigidBodyUtilities
import numpy as np
SC = exu.SystemContainer()
mbs = SC.AddSystem()
b0 = mbs.CreateRigidBody(inertia = InertiaCuboid(density=5000,
                                                 sideLengths=[1,0.1,0.1]),
                         referencePosition = [6,0,0],
                         gravity = [0,-9.81,0],
                         graphicsDataList = [exu.graphics.Brick(size=[1,0.1,0.1],
                                                                color=exu.
  graphics.color.orange)])
m1 = mbs.CreateMassPoint(referencePosition=[5.5,-1,0],
                         physicsMass=1, drawSize = 0.2)
mbs.CreateCoordinateConstraint(bodyNumbers=[None, b0],
                               coordinates=[None, 0]) #constraints X-coordinate
#constrain Y-coordinate of b0 to Z-coordinate of m1:
mbs.CreateCoordinateConstraint(bodyNumbers=[b0, m1],
                               coordinates=[1, 2])
mbs.Assemble()
simulationSettings = exu.SimulationSettings() #takes currently set values or
  default values
simulationSettings.timeIntegration.numberOfSteps = 1000
simulationSettings.timeIntegration.endTime = 2
mbs.SolveDynamic(simulationSettings = simulationSettings)
```

For examples on CreateCoordinateConstraint see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ballBearningModel.py (Ex), camFollowerExample.py (Ex), contactCurveExample.py (TM), createFunctionsTest.py (TM)

def **CreateRollingDisc** (*name*= '', *bodyNumbers*= [None, None], *axisPosition*= [], *axisVector*= [1,0,0], *discRadius*= 0., *planePosition*= [0,0,0], *planeNormal*= [0,0,1], *constrainedAxes*= [1,1,1], *activeConnector*= True, *show*= True, *discWidth*= 0.1, *color*= exudyn.graphics.color.default)

- **function description**:

  Create an ideal rolling disc joint between wheel rigid body and ground; the disc is infinitely thin and the ground is a perfectly flat plane; the wheel may lift off; definition of joint position and axis in global coordinates (alternatively in wheel (body1) local coordinates)

for reference configuration of bodies; all markers and other quantities are automatically computed; some constraint conditions may be deactivated, e.g. to resolve redundancy of constraints for multi-wheel vehicles

- NOTE that this function is added to MainSystem via Python function MainSystemCreateRollingDisc.

– **input**:

name: name string for joint; markers get Marker0:name and Marker1:name

bodyNumbers: a list of object numbers for body0=ground and body1=wheel; must be rigid body or ground object

axisPosition: a 3D vector as list or np.array: position of wheel axis in local body1=wheel coordinates

axisVector: a 3D vector as list or np.array containing the joint (=wheel) axis in local body1=wheel coordinates

discRadius: radius of the disc

planePosition: any 3D position vector of plane in ground object; given as local coordinates in ground object

planeNormal: 3D normal vector of the rolling (contact) plane on ground; given as local coordinates in ground object

constrainedAxes: [j0,j1,j2] flags, which determine which constraints are active, in which j0 represents the constraint for lateral motion, j1 longitudinal (forward/backward) motion and j2 represents the normal (contact) direction

activeConnector: flag to activate or deactivate the joint

show: if True, connector visualization is drawn

discWidth: disc with, only used for drawing

color: color of connector

– **output**: ObjectIndex; returns index of created joint

– **example**:

```
import exudyn as exu
from exudyn.utilities import * #includes itemInterface and rigidBodyUtilities
import numpy as np
SC = exu.SystemContainer()
mbs = SC.AddSystem()
r = 0.2
oDisc = mbs.CreateRigidBody(inertia = InertiaCylinder(density=5000, length=0.1,
  outerRadius=r, axis=0),
                            referencePosition = [1,0,r],
                            initialAngularVelocity = [-3*2*pi,0,0],
                            initialVelocity = [0,r*3*2*pi,0],
```

```
                        gravity = [0,0,-9.81],
                        graphicsDataList = [exu.graphics.Cylinder(pAxis =
  [-0.05,0,0], vAxis = [0.1,0,0], radius = r*0.99,
                                                                color=exu.
  graphics.color.blue),
                                                exu.graphics.Basis(length=2*r)])
oGround = mbs.CreateGround(graphicsDataList=[exu.graphics.CheckerBoard(size=4)])
mbs.CreateRollingDisc(bodyNumbers=[oGround, oDisc],
                      axisPosition=[0,0,0], axisVector=[1,0,0], #on local wheel
  frame
                      planePosition = [0,0,0], planeNormal = [0,0,1],  #in ground
  frame
                      discRadius = r,
                      discWidth=0.01, color=exu.graphics.color.steelblue)
mbs.Assemble()
simulationSettings = exu.SimulationSettings()
simulationSettings.timeIntegration.numberOfSteps = 1000
simulationSettings.timeIntegration.endTime = 2
mbs.SolveDynamic(simulationSettings = simulationSettings)
```

For examples on CreateRollingDisc see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- createFunctionsTest.py (TM), createRollingDiscTest.py (TM)

def **CreateRollingDiscPenalty** (*name*= '', *bodyNumbers*= [None, None], *axisPosition*= [], *axisVector*= [1,0,0], *discRadius*= 0., *planePosition*= [0,0,0], *planeNormal*= [0,0,1], *contactStiffness*= 0., *contactDamping*= 0., *dryFriction*= [0,0], *dryFrictionAngle*= 0., *dryFrictionProportionalZone*= 0., *viscousFriction*= [0,0], *rollingFrictionViscous*= 0., *useLinearProportionalZone*= False, *activeConnector*= True, *show*= True, *discWidth*= 0.1, *color*= exudyn.graphics.color.default)

- **function description**:

  Create penalty-based rolling disc joint between wheel rigid body and ground; the disc is infinitely thin and the ground is a perfectly flat plane; the wheel may lift off; definition of joint position and axis in global coordinates (alternatively in wheel (body1) local coordinates) for reference configuration of bodies; all markers and other quantities are automatically computed

  - NOTE that this function is added to MainSystem via Python function MainSystemCreateRollingDiscPenalty.

- **input**:

  name: name string for joint; markers get Marker0:name and Marker1:name

  bodyNumbers: a list of object numbers for body0=ground and body1=wheel; must be rigid body or ground object

  axisPosition: a 3D vector as list or np.array: position of wheel axis in local body1=wheel coordinates

axisVector: a 3D vector as list or np.array containing the joint (=wheel) axis in local body1=wheel coordinates

discRadius: radius of the disc

planePosition: any 3D position vector of plane in ground object; given as local coordinates in ground object

planeNormal: 3D normal vector of the rolling (contact) plane on ground; given as local coordinates in ground object

dryFrictionAngle: angle (radiant) which defines a rotation of the local tangential coordinates dry friction; this allows to model Mecanum wheels with specified roll angle

contactStiffness: normal contact stiffness

contactDamping: normal contact damping

dryFriction: 2D list of friction parameters; dry friction coefficients in local wheel coordinates, where for dryFrictionAngle=0, the first parameter refers to forward direction and the second parameter to lateral direction

viscousFriction: 2D list of viscous friction coefficients [SI:1/(m/s)] in local wheel coordinates; proportional to slipping velocity, leading to increasing slipping friction force for increasing slipping velocity; directions are same as in dryFriction

dryFrictionProportionalZone: limit velocity [m/s] up to which the friction is proportional to velocity (for regularization / avoid numerical oscillations)

rollingFrictionViscous: rolling friction [SI:1], which acts against the velocity of the trail on ground and leads to a force proportional to the contact normal force;

useLinearProportionalZone: if True, a linear proportional zone is used; the linear zone performs better in implicit time integration as the Jacobian has a constant tangent in the sticking case

activeConnector: flag to activate or deactivate the connector

show: if True, connector visualization is drawn

discWidth: disc with, only used for drawing

color: color of connector

- **output**: ObjectIndex; returns index of created joint

- **example**:

```
import exudyn as exu
from exudyn.utilities import * #includes itemInterface and rigidBodyUtilities
import numpy as np
SC = exu.SystemContainer()
mbs = SC.AddSystem()
r = 0.2
oDisc = mbs.CreateRigidBody(inertia = InertiaCylinder(density=5000, length=0.1,
  outerRadius=r, axis=0),
```

```
                            referencePosition = [1,0,r],
                            initialAngularVelocity = [-3*2*pi,0,0],
                            initialVelocity = [0,r*3*2*pi,0],
                            gravity = [0,0,-9.81],
                            graphicsDataList = [exu.graphics.Cylinder(pAxis =
    [-0.05,0,0], vAxis = [0.1,0,0], radius = r*0.99,
                                                                    color=exu.
    graphics.color.blue),
                                                    exu.graphics.Basis(length=2*r)])
oGround = mbs.CreateGround(graphicsDataList=[exu.graphics.CheckerBoard(size=4)])
mbs.CreateRollingDiscPenalty(bodyNumbers=[oGround, oDisc], axisPosition=[0,0,0],
    axisVector=[1,0,0],
                                discRadius = r, planePosition = [0,0,0], planeNormal
    = [0,0,1],
                                dryFriction = [0.2,0.2],
                                contactStiffness = 1e5, contactDamping = 2e3,
                                discWidth=0.01, color=exu.graphics.color.steelblue)
mbs.Assemble()
simulationSettings = exu.SimulationSettings()
simulationSettings.timeIntegration.numberOfSteps = 1000
simulationSettings.timeIntegration.endTime = 2
mbs.SolveDynamic(simulationSettings = simulationSettings)
```

For examples on CreateRollingDiscPenalty see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- createFunctionsTest.py (TM), createRollingDiscPenaltyTest.py (TM)

def **CreateKinematicTree** (*name*= '', *listOfTreeLinks*= [], *referenceCoordinates*= None, *initialCoordinates*= None, *initialCoordinates_t*= None, *gravity*= [0.,0.,0.], *baseOffset*= [0.,0.,0.], *linkForces*= None, *linkTorques*= None, *jointForceVector*= None, *jointPositionOffsetVector*= None, *jointVelocityOffsetVector*= None, *forceUserFunction*= 0, *jointRadius*= 0.05, *jointWidth*= 0.12, *colors*= exudyn.graphics.color.default, *colorsJoints*= exudyn.graphics.color.default, *baseGraphicsDataList*= None, *linkRoundness*= 0.2, *show*= True)

- **function description**:

  helper function to create 2D or 3D mass point object and node, using arguments as in NodePoint and MassPoint; uses TreeLink as defined in exudyn.rigidBodyUtilities

  - NOTE that this function is added to MainSystem via Python function MainSystemCreateKinematicTree.

- **input**:

  name: name string for object, node is 'Node:'+name

  listOfTreeLinks: list of TreeLink (from exudyn.rigidBodyUtilities) which characterize the KinematicTree

referenceCoordinates: reference coordinates all kinematic tree coordinates (configuration when current coordinates are zero)

initialCoordinates: initial deviation from reference coordinates

initialVelocities: initial velocities for point node (always a 3D vector, no matter if 2D or 3D mass)

gravity: gravity vevtor applied to kinematic tree (always a 3D vector, no matter if 2D or 3D mass)

baseOffset: constant 3D vector representing the origin of the kinematic tree

linkForces: Vector3DList of forces per link (at joint origin) or None

linkTorques: Vector3DList of torques per link or None

jointForceVector: a list or numpy array of scalar forces per joint, representing joint forces (prismatic joint) or joint torques (revolute joint)

jointPositionOffsetVector: a list or numpy array of scalar set coordinates per joint; use PreStepUserFunction to change values over time

jointVelocityOffsetVector: a list or numpy array of scalar set velocities per joint; use PreStepUserFunction to change values over time

forceUserFunction: A Python user function which computes the generalized force vector on RHS with identical action as jointForceVector; for description see ObjectKinematicTree

show: show kinematic tree

showLinks: set true, if links shall be shown; if graphicsDataList is empty, a standard drawing for links is used (drawing a cylinder from previous joint or base to next joint; size relative to frame size in KinematicTree visualization settings); else graphicsDataList are used per link; NOTE visualization of joint and COM frames can be modified via visualizationSettings.bodies.kinematicTree

showJoints: set true, if joints shall be shown; if graphicsDataList is empty, a standard drawing for joints is used (drawing a cylinder for revolute joints; size relative to frame size in KinematicTree visualization settings)

jointRadius: for generic visualization of joints and links

jointWidth: for generic visualization of joints and links

colors: either one general color for kinematic tree, or list with one color per link

colorsJoints: either one color for all joints or list with one color per joint

baseGraphicsDataList: graphics for base; if None, it is computed automatically; otherwise a list of graphicsData or empty list

linkRoundness: for automatic generation of graphics for links, roundness=0 give brick-shape, roundness<1 give transition of brick to ellipsoid and roundness=1 give cylinders

show: show kinematic tree

– **output**: ObjectIndex; returns kinematic tree object index

For examples on CreateKinematicTree see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- humanRobotInteraction.py (Ex), kinematicTreeAndMBS.py (Ex), kinematicTreePendulum.py (Ex), openAIgymNLinkAdvanced.py (Ex), openAIgymNLinkContinuous.py (Ex), ... , createKinematicTreeTest.py (TM), kinematicTreeAndMBStest.py (TM), kinematicTreeConstraintTest.py (TM), ...

def **CreateForce** (*name*= '', *bodyNumber*= None, *loadVector*= [0.,0.,0.], *localPosition*= [0.,0.,0.], *bodyFixed*= False, *loadVectorUserFunction*= 0, *show*= True)

– **function description**:

helper function to create force applied to given body

- NOTE that this function is added to MainSystem via Python function MainSystemCreateForce.

– **input**:

name: name string for object

bodyNumber: body number (ObjectIndex) at which the force is applied to

loadVector: force vector (as 3D list or numpy array)

localPosition: local position (as 3D list or numpy array) where force is applied

bodyFixed: if True, the force is corotated with the body; else, the force is global

loadVectorUserFunction: A Python function f(mbs, t, load)->loadVector which defines the time-dependent load and replaces loadVector in every time step; the arg load is the static loadVector

show: if True, load is drawn

– **output**: LoadIndex; returns load index

– **example**:

```
import exudyn as exu
from exudyn.utilities import * #includes itemInterface and rigidBodyUtilities
import numpy as np
SC = exu.SystemContainer()
mbs = SC.AddSystem()
b0=mbs.CreateMassPoint(referencePosition = [0,0,0],
                       initialVelocity = [2,5,0],
                       physicsMass = 1, gravity = [0,-9.81,0],
                       drawSize = 0.5, color=exu.graphics.color.blue)
f0=mbs.CreateForce(bodyNumber=b0, loadVector=[100,0,0],
                   localPosition=[0,0,0])
mbs.Assemble()
```

```
simulationSettings = exu.SimulationSettings() #takes currently set values or
  default values
simulationSettings.timeIntegration.numberOfSteps = 1000
simulationSettings.timeIntegration.endTime = 2
mbs.SolveDynamic(simulationSettings = simulationSettings)
```

For examples on CreateForce see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ballBearningModel.py (Ex), cartesianSpringDamper.py (Ex), cartesianSpringDamperUserFunction.py (Ex), chatGPTupdate.py (Ex), chatGPTupdate2.py (Ex), ... , createFunctionsTest.py (TM), loadUserFunctionTest.py (TM), mainSystemExtensionsTests.py (TM), ...

def **CreateTorque** (*name*= '', *bodyNumber*= None, *loadVector*= [0.,0.,0.], *localPosition*= [0.,0.,0.], *bodyFixed*= False, *loadVectorUserFunction*= 0, *show*= True)

- **function description**:

  helper function to create torque applied to given body

  - NOTE that this function is added to MainSystem via Python function MainSystemCreateTorque.

- **input**:

  name: name string for object

  bodyNumber: body number (ObjectIndex) at which the torque is applied to

  loadVector: torque vector (as 3D list or numpy array)

  localPosition: local position (as 3D list or numpy array) where torque is applied

  bodyFixed: if True, the torque is corotated with the body; else, the torque is global

  loadVectorUserFunction: A Python function f(mbs, t, load)->loadVector which defines the time-dependent load and replaces loadVector in every time step; the arg load is the static loadVector

  show: if True, load is drawn

- **output**: LoadIndex; returns load index

- **example**:

```
import exudyn as exu
from exudyn.utilities import * #includes itemInterface and rigidBodyUtilities
import numpy as np
SC = exu.SystemContainer()
mbs = SC.AddSystem()
b0 = mbs.CreateRigidBody(inertia = InertiaCuboid(density=5000,
                                                 sideLengths=[1,0.1,0.1]),
                         referencePosition = [1,3,0],
                         gravity = [0,-9.81,0],
                         graphicsDataList = [exu.graphics.Brick(size=[1,0.1,0.1],
```

```
                                                                    color=exu.
  graphics.color.red)])
f0=mbs.CreateTorque(bodyNumber=b0, loadVector=[0,100,0])
mbs.Assemble()
simulationSettings = exu.SimulationSettings() #takes currently set values or
  default values
simulationSettings.timeIntegration.numberOfSteps = 1000
simulationSettings.timeIntegration.endTime = 2
mbs.SolveDynamic(simulationSettings = simulationSettings)
```

For examples on CreateTorque see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ballBearningModel.py (Ex), chatGPTupdate.py (Ex), chatGPTupdate2.py (Ex), rigidBodyTutorial3.py (Ex), createFunctionsTest.py (TM), mainSystemExtensionsTests.py (TM), pickleCopyMbs.py (TM), simulatorCouplingTwoMbs.py (TM), ...

6.5.2 MainSystem extensions (general)

This section represents general extensions to MainSystem, which are direct calls to Python functions, such as PlotSensor or SolveDynamic; these extensions allow a more intuitive interaction with the MainSystem class, see the following example. For activation, import `exudyn.mainSystemExtensions` or `exudyn.utilities`

```
#this example sketches the usage
#for complete examples see Examples/ or TestModels/ folders
#create some multibody system (mbs) first:
# ...
#
#compute system degree of freedom:
mbs.ComputeSystemDegreeOfFreedom(verbose=True)
#
#call solver function directly from mbs:
mbs.SolveDynamic(exu.SimulationSettings())
#
#plot sensor directly from mbs:
mbs.PlotSensor(...)
```

def **SolutionViewer** (*solution*= None, *rowIncrement*= 1, *timeout*= 0.04, *runOnStart*= True, *runMode*= 2, *fontSize*= 12, *title*= '', *checkRenderEngineStopFlag*= True)

- **function description**:

  open interactive dialog and visulation (animate) solution loaded with LoadSolutionFile(...); Change slider 'Increment' to change the automatic increment of time frames; Change mode between continuous run, one cycle (fits perfect for animation recording) or 'Static' (to change Solution steps manually with the mouse); update period also lets you change

the speed of animation; Press Run / Stop button to start/stop interactive mode (updating of grpahics)

- NOTE that this function is added to MainSystem via Python function SolutionViewer.

– **input**:

solution: solution dictionary previously loaded with exudyn.utilities.LoadSolutionFile(...); will be played from first to last row; if solution==None, it tries to load the file coordinatesSolutionFileName as stored in mbs.sys['simulationSettings'], which are the simulationSettings of the previous simulation

rowIncrement: can be set larger than 1 in order to skip solution frames: e.g. rowIncrement=10 visualizes every 10th row (frame)

timeout: in seconds is used between frames in order to limit the speed of animation; e.g. use timeout=0.04 to achieve approximately 25 frames per second

runOnStart: immediately go into 'Run' mode

runMode: 0=continuous run, 1=one cycle, 2=static (use slider/mouse to vary time steps)

fontSize: define font size for labels in InteractiveDialog

title: if empty, it uses default; otherwise define specific title

checkRenderEngineStopFlag: if True, stopping renderer (pressing Q or Escape) also causes stopping the interactive dialog

– **output**: None; updates current visualization state, renders the scene continuously (after pressing button 'Run')

– **example**:

```
#HERE, mbs must contain same model as solution stored in coordinatesSolution.txt
#adjust autoFitScence, otherwise it may lead to unwanted fit to scene
SC.visualizationSettings.general.autoFitScene = False
from exudyn.interactive import SolutionViewer #import function
sol = LoadSolutionFile('coordinatesSolution.txt') #load solution: adjust to your
  file name
mbs.SolutionViewer(sol) #call via MainSystem
```

For examples on SolutionViewer see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- addPrismaticJoint.py (Ex), addRevoluteJoint.py (Ex), ANCFcableCantilevered.py (Ex), ANCFrotatingCable2D.py (Ex), ballBearningModel.py (Ex), ... , ACFtest.py (TM), ANCFbeltDrive.py (TM), ANCFgeneralContactCircle.py (TM), ...

def **PlotSensor** (*sensorNumbers*= [], *components*= 0, *xLabel*= 'time (s)', *yLabel*= None, *labels*= [], *colorCodeOffset*= 0, *newFigure*= True, *closeAll*= False, *componentsX*= [], *title*= '', *figureName*= '', *fontSize*= 16, *colors*= [], *lineStyles*= [], *lineWidths*= [], *markerStyles*= [], *markerSizes*= [], *markerDensity*= 0.08, *rangeX*= [], *rangeY*= [], *majorTicksX*= 10, *majorTicksY*= 10, *offsets*= [], *factors*= [], *subPlot*= [], *sizeInches*= [6.4,4.8], *fileName*= '', *useXYZcomponents*= True, ***kwargs*)

– **function description**:

Helper function for direct and easy visualization of sensor outputs, without need for loading text files, etc.; PlotSensor can be used to simply plot, e.g., the measured x-Position over time in a figure. PlotSensor provides an interface to matplotlib (which needs to be installed). Default values of many function arguments can be changed using the exudyn.plot function PlotSensorDefaults(), see there for usage.

- NOTE that this function is added to MainSystem via Python function PlotSensor.

– **input**:

sensorNumbers: consists of one or a list of sensor numbers (type SensorIndex or int) as returned by the mbs function AddSensor(...); sensors need to set writeToFile=True and/or storeInternal=True for PlotSensor to work; alternatively, it may contain FILENAMES (incl. path) to stored sensor or solution files OR a numpy array instead of sensor numbers; the format of data (file or numpy array) must contain per row the time and according solution values in columns; if components is a list and sensorNumbers is a scalar, sensorNumbers is adjusted automatically to the components

components: consists of one or a list of components according to the component of the sensor to be plotted at y-axis; if components is a list and sensorNumbers is a scalar, sensorNumbers is adjusted automatically to the components; as always, components are zero-based, meaning 0=X, 1=Y, etc.; for regular sensor files, time will be component=-1; to show the norm (e.g., of a force vector), use component=[plot.componentNorm] for according sensors; norm will consider all values of sensor except time (for 3D force, it will be $\sqrt{f_0^2 + f_1^2 + f_2^2}$); offsets and factors are mapped on norm (plot value=factor*(norm(values) + offset)), not on component values

componentsX: default componentsX=[] uses time in files; otherwise provide componentsX as list of components (or scalar) representing x components of sensors in plotted curves; DON'T forget to change xLabel accordingly!

Using componentsX=[...] with a list of column indices specifies the respective columns used for the x-coordinates in all sensors; by default, values are plotted against the first column in the files, which is time; according to counting in PlotSensor, this represents componentX=-1;

plotting y over x in a position sensor thus reads: components=[1], componentsX=[0];

plotting time over x reads: components=[-1], componentsX=[0];

the default value reads componentsX=[-1,-1,...]

xLabel: string for text at x-axis

yLabel: string for text at y-axis (default: None==> label is automatically computed from sensor value types)

labels: string (for one sensor) or list of strings (according to number of sensors resp. components) representing the labels used in legend; if labels=[], automatically generated legend is used

rangeX: default rangeX=[]: computes range automatically; otherwise use rangeX to set range (limits) for x-axis provided as sorted list of two floats, e.g., rangeX=[0,4]

rangeY: default rangeY=[]: computes range automatically; otherwise use rangeY to set range (limits) for y-axis provided as sorted list of two floats, e.g., rangeY=[-1,1]

figureName: optional name for figure, if newFigure=True

fontSize: change general fontsize of axis, labels, etc. (matplotlib default is 12, default in PlotSensor: 16)

title: optional string representing plot title

offsets: provide as scalar, list of scalars (per sensor) or list of 2D numpy.arrays (per sensor, having same rows/columns as sensor data; in this case it will also influence x-axis if componentsX is different from -1) to add offset to each sensor output; for an original value fOrig, the new value reads fNew = factor*(fOrig+offset); for offset provided as numpy array (with same time values), the 'time' column is ignored in the offset computation; can be used to compute difference of sensors; if offsets=[], no offset is used

factors: provide as scalar or list (per sensor) to add factor to each sensor output; for an original value fOrig, the new value reads fNew = factor*(fOrig+offset); if factor=[], no factor is used

majorTicksX: number of major ticks on x-axis; default: 10

majorTicksY: number of major ticks on y-axis; default: 10

colorCodeOffset: int offset for color code, color codes going from 0 to 27 (see PlotLineCode(...)); automatic line/color codes are used if no colors and lineStyles are used

colors: color is automatically selected from colorCodeOffset if colors=[]; otherwise chose from 'b', 'g', 'r', 'c', 'm', 'y', 'k' and many other colors see https://matplotlib.org/stable/gallery/color/named_c

lineStyles: line style is automatically selected from colorCodeOffset if lineStyles=[]; otherwise define for all lines with string or with list of strings, chosing from '-', '–', '-.', ':', or ''

lineWidths: float to define line width by float (default=1); either use single float for all sensors or list of floats with length >= number of sensors

markerStyles: if different from [], marker styles are defined as list of marker style strings or single string for one sensor; chose from '.', 'o', 'x', '+' ... check listMarkerStylesFilled and listMarkerStyles in exudyn.plot and see https://matplotlib.org/stable/api/markers_api.html ; ADD a space to markers to make them empty (transparent), e.g. 'o ' will create an empty circle

markerSizes: float to define marker size by float (default=6); either use single float for all sensors or list of floats with length >= number of sensors

markerDensity: if int, it defines approx. the total number of markers used along each graph; if float, this defines the distance of markers relative to the diagonal of the plot (default=0.08); if None, it adds a marker to every data point if marker style is specified for sensor

newFigure: if True, a new matplotlib.pyplot figure is created; otherwise, existing figures are overwritten

subPlot: given as list [nx, ny, position] with nx, ny being the number of subplots in x and y direction (nx=cols, ny=rows), and position in [1,..., nx*ny] gives the position in the subplots; use the same structure for first PlotSensor (with newFigure=True) and all subsequent PlotSensor calls with newFigure=False, which creates the according subplots; default=[](no subplots)

sizeInches: given as list [sizeX, sizeY] with the sizes per (sub)plot given in inches; default: [6.4, 4.8]; in case of sub plots, the total size of the figure is computed from nx*sizeInches[0] and ny*sizeInches[1]

fileName: if this string is non-empty, figure will be saved to given path and filename (use figName.pdf to safe as PDF or figName.png to save as PNG image); use matplotlib.use('Agg') in order not to open figures if you just want to save them

useXYZcomponents: of True, it will use X, Y and Z for sensor components, e.g., measuring Position, Velocity, etc. wherever possible

closeAll: if True, close all figures before opening new one (do this only in first PlotSensor command!)

*[*kwargs]*:

minorTicksXon: if True, turn minor ticks for x-axis on

minorTicksYon: if True, turn minor ticks for y-axis on

logScaleX: use log scale for x-axis

logScaleY: use log scale for y-axis

fileCommentChar: if exists, defines the comment character in files (#,

fileDelimiterChar: if exists, defines the character indicating the columns for data (',', ' ', ';', ...)

- **output**: [Any, Any, Any, Any]; plots the sensor data; returns [plt, fig, ax, line] in which plt is matplotlib.pyplot, fig is the figure (or None), ax is the axis (or None) and line is the return value of plt.plot (or None) which could be changed hereafter

- **notes**: adjust default values by modifying the variables exudyn.plot.plotSensorDefault..., e.g., exudyn.plot.plotSensorDefaultFontSize

- **example**:

```
#assume to have some position-based nodes 0 and 1:
s0=mbs.AddSensor(SensorNode(nodeNumber=0, fileName='s0.txt',
                            outputVariableType=exu.OutputVariableType.Position))
s1=mbs.AddSensor(SensorNode(nodeNumber=1, fileName='s1.txt',
                            outputVariableType=exu.OutputVariableType.Position))
mbs.PlotSensor(s0, 0) #plot x-coordinate
#plot x for s0 and z for s1:
mbs.PlotSensor(sensorNumbers=[s0,s1], components=[0,2], yLabel='this is the
  position in meter')
mbs.PlotSensor(sensorNumbers=s0, components=plot.componentNorm) #norm of position
```

```
mbs.PlotSensor(sensorNumbers=s0, components=[0,1,2], factors=1000., title='Answers
  to the big questions')
mbs.PlotSensor(sensorNumbers=s0, components=[0,1,2,3],
           yLabel='Coordantes with offset 1\nand scaled with $\\frac{1}{1000}$',
           factors=1e-3, offsets=1,fontSize=12, closeAll=True)
#assume to have body sensor sBody, marker sensor sMarker:
mbs.PlotSensor(sensorNumbers=[sBody]*3+[sMarker]*3, components=[0,1,2,0,1,2],
           colorCodeOffset=3, newFigure=False, fontSize=10,
           yLabel='Rotation $\\alpha, \\beta, \\gamma$ and\n Position $x,y,z$',
           title='compare marker and body sensor')
#assume having file plotSensorNode.txt:
mbs.PlotSensor(sensorNumbers=[s0]*3+ [filedir+'plotSensorNode.txt']*3,
           components=[0,1,2]*2)
#plot y over x:
mbs.PlotSensor(sensorNumbers=s0, componentsX=[0], components=[1], xLabel='x-
  Position', yLabel='y-Position')
#for further examples, see also Examples/plotSensorExamples.py
```

For examples on PlotSensor see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ANCFALEtest.py (Ex), beltDriveALE.py (Ex), beltDriveReevingSystem.py (Ex), beltDrivesComparison.py (Ex), bicycleIftommBenchmark.py (Ex), ... , ACFtest.py (TM), ANCFbeltDrive.py (TM), ANCFgeneralContactCircle.py (TM), ...

def **SolveStatic** (*simulationSettings*= exudyn.SimulationSettings(), *updateInitialValues*= False, *storeSolver*= True, *showHints*= False, *showCausingItems*= True, *autoAssemble*= True)

- **function description**:

  solves the static mbs problem using simulationSettings; check theDoc.pdf for MainSolverStatic for further details of the static solver; this function is also available in exudyn (using exudyn.SolveStatic(...))

  - NOTE that this function is added to MainSystem via Python function SolveStatic.

- **input**:

  simulationSettings: specific simulation settings out of exu.SimulationSettings(), as described in Section 9.2.0.1; use options for newton, discontinuous settings, etc., from staticSolver sub-items

  updateInitialValues: if True, the results are written to initial values, such at a consecutive simulation uses the results of this simulation as the initial values of the next simulation

  storeSolver: if True, the staticSolver object is stored in the mbs.sys dictionary as mbs.sys['staticSolver'], and simulationSettings are stored as mbs.sys['simulationSettings']

  showHints: show additional hints, if solver fails

  showCausingItems: if linear solver fails, this option helps to identify objects, etc. which are related to a singularity in the linearized system matrix

autoAssemble: if True: if mbs.systemIsConsistent=False (system is not assembled), call mbs.Assemble() before solver calls

- **output**: bool; returns True, if successful, False if fails; if storeSolver = True, mbs.sys contains staticSolver, which allows to investigate solver problems (check theDoc.pdf Section 9.4 and the items described in Section 9.4.0.5)

- **example**:

```
import exudyn as exu
from exudyn.itemInterface import *
SC = exu.SystemContainer()
mbs = SC.AddSystem()
#create simple system:
ground = mbs.AddObject(ObjectGround())
mbs.AddNode(NodePoint())
body = mbs.AddObject(MassPoint(physicsMass=1, nodeNumber=0))
m0 = mbs.AddMarker(MarkerBodyPosition(bodyNumber=ground))
m1 = mbs.AddMarker(MarkerBodyPosition(bodyNumber=body))
mbs.AddObject(CartesianSpringDamper(markerNumbers=[m0,m1], stiffness=[100,100,100])
  )
mbs.AddLoad(LoadForceVector(markerNumber=m1, loadVector=[10,10,10]))
mbs.Assemble()
simulationSettings = exu.SimulationSettings()
simulationSettings.timeIntegration.endTime = 10
success = mbs.SolveStatic(simulationSettings, storeSolver = True)
exu.Print("success =", success)
exu.Print("iterations = ", mbs.sys['staticSolver'].it)
exu.Print("pos=", mbs.GetObjectOutputBody(body,localPosition=[0,0,0],
      variableType=exu.OutputVariableType.Position))
```

For examples on SolveStatic see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- 3SpringsDistance.py (Ex), ALEANCFpipe.py (Ex), ANCFALEtest.py (Ex), ANCFcantileverTest.py (Ex), ANCFcontactCircle.py (Ex), ... , ANCFBeamTest.py (TM), ANCFbeltDrive.py (TM), ANCFcontactCircleTest.py (TM), ...

def **SolveDynamic** (*simulationSettings*= exudyn.SimulationSettings(), *solverType*= exudyn.DynamicSolverType.GeneralizedAlpha, *updateInitialValues*= False, *storeSolver*= True, *showHints*= False, *showCausingItems*= True, *autoAssemble*= True)

- **function description**:

  solves the dynamic mbs problem using simulationSettings and solver type; check theDoc.pdf for MainSolverImplicitSecondOrder for further details of the dynamic solver; this function is also available in exudyn (using exudyn.SolveDynamic(...))

  - NOTE that this function is added to MainSystem via Python function SolveDynamic.

- **input**:

simulationSettings: specific simulation settings out of exu.SimulationSettings(), as described in Section 9.2.0.1; use options for newton, discontinuous settings, etc., from timeIntegration; therein, implicit second order solvers use settings from generalizedAlpha and explict solvers from explicitIntegration; be careful with settings, as the influence accuracy (step size!), convergence and performance (see special Section 2.4.16)

solverType: use exudyn.DynamicSolverType to set specific solver (default=generalized alpha)

updateInitialValues: if True, the results are written to initial values, such at a consecutive simulation uses the results of this simulation as the initial values of the next simulation

storeSolver: if True, the staticSolver object is stored in the mbs.sys dictionary as mbs.sys['staticSolver'], and simulationSettings are stored as mbs.sys['simulationSettings']

showHints: show additional hints, if solver fails

showCausingItems: if linear solver fails, this option helps to identify objects, etc. which are related to a singularity in the linearized system matrix

autoAssemble: if True: if mbs.systemIsConsistent=False (system is not assembled), call mbs.Assemble() before solver calls

– **output**: bool; returns True, if successful, False if fails; if storeSolver = True, mbs.sys contains staticSolver, which allows to investigate solver problems (check theDoc.pdf Section 9.4 and the items described in Section 9.4.0.5)

– **example**:

```
import exudyn as exu
from exudyn.itemInterface import *
SC = exu.SystemContainer()
mbs = SC.AddSystem()
#create simple system:
ground = mbs.AddObject(ObjectGround())
mbs.AddNode(NodePoint())
body = mbs.AddObject(MassPoint(physicsMass=1, nodeNumber=0))
m0 = mbs.AddMarker(MarkerBodyPosition(bodyNumber=ground))
m1 = mbs.AddMarker(MarkerBodyPosition(bodyNumber=body))
mbs.AddObject(CartesianSpringDamper(markerNumbers=[m0,m1], stiffness=[100,100,100])
  )
mbs.AddLoad(LoadForceVector(markerNumber=m1, loadVector=[10,10,10]))
#
mbs.Assemble()
simulationSettings = exu.SimulationSettings()
simulationSettings.timeIntegration.endTime = 10
success = mbs.SolveDynamic(simulationSettings, storeSolver = True)
exu.Print("success =", success)
exu.Print("iterations = ", mbs.sys['dynamicSolver'].it)
exu.Print("pos=", mbs.GetObjectOutputBody(body,localPosition=[0,0,0],
      variableType=exu.OutputVariableType.Position))
```

For examples on SolveDynamic see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- 3SpringsDistance.py (Ex), addPrismaticJoint.py (Ex), addRevoluteJoint.py (Ex), ALEANCFpipe.py (Ex), ANCFALEtest.py (Ex), ... , abaqusImportTest.py (TM), ACFtest.py (TM), ANCFBeamEigTest.py (TM), ...

def **ComputeLinearizedSystem** (*simulationSettings*= exudyn.SimulationSettings(), *projectIntoConstraintNullspace*= False, *singularValuesTolerance*= 1e-12, *returnConstraintJacobian*= False, *returnConstraintNullspace*= False, *autoAssemble*= True)

- **function description**:

  compute linearized system of equations for ODE2 part of mbs, not considering the effects of algebraic constraints; for computation of eigenvalues and advanced computation with constrained systems, see ComputeODE2Eigenvalues; the current implementation is also able to project into the constrained space, however, this currently does not generally work with non-holonomic systems

  - NOTE that this function is added to MainSystem via Python function ComputeLinearizedSystem.

- **input**:

  simulationSettings: specific simulation settings used for computation of jacobian (e.g., sparse mode in static solver enables sparse computation)

  projectIntoConstraintNullspace: if False, algebraic equations (and constraint jacobian) are not considered for the linearized system; if True, the equations are projected into the nullspace of the constraints in the current configuration, using singular value decomposition; in the latter case, the returned list contains [M, K, D, C, N] where C is the constraint jacobian and N is the nullspace matrix (C and N may be an empty list, depending on the following flags)

  singularValuesTolerance: tolerance used to distinguish between zero and nonzero singular values for algebraic constraints projection

  returnConstraintJacobian: if True, the returned list contains [M, K, D, C, N] where C is the constraint jacobian and N is the nullspace matrix (may be empty)

  returnConstraintNullspace: if True, the returned list contains [M, K, D, C, N] where C is the constraint jacobian (may be empty) and N is the nullspace matrix

  autoAssemble: if True: if mbs.systemIsConsistent=False (system is not assembled), call mbs.Assemble() before solver calls

- **output**: [ArrayLike, ArrayLike, ArrayLike]; [M, K, D]; list containing numpy mass matrix M, stiffness matrix K and damping matrix D; for constraints, see options with arguments above, return values may change to [M, K, D, C, N]

- **notes**: consider paper of Agundez, Vallejo, Freire, Mikkola, "The dependent coordinates in the linearization of constrained multibody systems: Handling and elimination", https://www.sciencedirect.com/scie

– **example**:

```
import exudyn as exu
from exudyn.utilities import *
import numpy as np
SC = exu.SystemContainer()
mbs = SC.AddSystem()
#
b0 = mbs.CreateMassPoint(referencePosition = [2,0,0],
                         initialVelocity = [2*0,5,0],
                         physicsMass = 1, gravity = [0,-9.81,0],
                         drawSize = 0.5, color=graphics.color.blue)
#
oGround = mbs.AddObject(ObjectGround())
#add vertical spring
oSD = mbs.CreateSpringDamper(bodyOrNodeList=[oGround, b0],
                             localPosition0=[2,1,0],
                             localPosition1=[0,0,0],
                             stiffness=1e4, damping=1e2,
                             drawSize=0.2)
#
mbs.Assemble()
[M,K,D] = mbs.ComputeLinearizedSystem()
exu.Print('M=\n',M,'\nK=\n',K,'\nD=\n',D)
```

For examples on ComputeLinearizedSystem see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ANCFBeamEigTest.py (TM), ANCFBeamTest.py (TM), geometricallyExactBeamTest.py (TM), mainSystemExtensionsTests.py (TM)

def **ComputeODE2Eigenvalues** (*simulationSettings*= exudyn.SimulationSettings(), *useSparseSolver*= False, *numberOfEigenvalues*= 0, *constrainedCoordinates*= [], *convert2Frequencies*= False, *useAbsoluteValues*= True, *computeComplexEigenvalues*= False, *ignoreAlgebraicEquations*= False, *singularValuesTolerance*= 1e-12, *autoAssemble*= True)

– **function description**:

compute eigenvalues for unconstrained ODE2 part of mbs, which represent the square of the eigenfrequencies (in radiant) of the undamped system; the computation may include constraints in case that ignoreAlgebraicEquations=False (however, this currently does not generally work with non-holonomic systems); for algebraic constraints, however, a dense singular value decomposition of the constraint jacobian is used for the nullspace projection; the computation is done for the initial values of the mbs, independently of previous computations. If you would like to use the current state for the eigenvalue computation, you need to copy the current state to the initial state (using GetSystemState, SetSystemState, see Section 6.6); note that mass and stiffness matrices are computed in dense mode so far, while eigenvalues are computed according to useSparseSolver.

- NOTE that this function is added to MainSystem via Python function ComputeODE2Eigenvalues.

– **input**:

simulationSettings: specific simulation settings used for computation of jacobian (e.g., sparse mode in static solver enables sparse computation)

useSparseSolver: if False (only for small systems), all eigenvalues are computed in dense mode (slow for large systems!); if True, only the numberOfEigenvalues are computed (numberOfEigenvalues must be set!); Currently, the matrices are exported only in DENSE MODE from mbs, which means that intermediate matrices may become huge for more than 5000 coordinates! NOTE that the sparsesolver accuracy is much less than the dense solver

numberOfEigenvalues: number of eigenvalues and eivenvectors to be computed; if numberOfEigenvalues==0, all eigenvalues will be computed (may be impossible for larger or sparse problems!)

constrainedCoordinates: if this list is non-empty (and there are no algebraic equations or ignoreAlgebraicEquations=True), the integer indices represent constrained coordinates of the system, which are fixed during eigenvalue/vector computation; according rows/columns of mass and stiffness matrices are erased; in this case, algebraic equations of the system are ignored

convert2Frequencies: if True, the square root is computed for eigenvalues, they are converted into frequencies (Hz), and the output is [eigenFrequencies, eigenVectors]

useAbsoluteValues: if True, abs(eigenvalues) is used, which avoids problems for small (close to zero) eigenvalues; needed, when converting to frequencies

computeComplexEigenvalues: if True, the system is converted into a system of first order differential equations, including damping terms; returned eigenvalues are complex and contain the 'damping' (=real) part and the eigenfrequency (=complex) part; for this case, set useAbsoluteValues=False (otherwise you will not get the complex values; values are unsorted, however!); also, convert2Frequencies must be False in this case! only implemented for dense solver

ignoreAlgebraicEquations: if True, algebraic equations (and constraint jacobian) are not considered for eigenvalue computation; otherwise, the solver tries to automatically project the system into the nullspace kernel of the constraint jacobian using a SVD; this gives eigenvalues of the constrained system; eigenvectors are not computed

singularValuesTolerance: tolerance used to distinguish between zero and nonzero singular values for algebraic constraints projection

autoAssemble: if True: if mbs.systemIsConsistent=False (system is not assembled), call mbs.Assemble() before solver calls

– **output**: [ArrayLike, ArrayLike]; [eigenValues, eigenVectors]; eigenValues being a numpy array of eigen values (ω_i^2, being the squared eigen frequencies in (ω_i in rad/s)!), eigenVectors a numpy array containing the eigenvectors in every column

– **author**: Johannes Gerstmayr, Michael Pieber

– **example**:

```
 #take any example from the Examples or TestModels folder, e.g., '
  cartesianSpringDamper.py' and run it
 #specific example:
import exudyn as exu
from exudyn.utilities import *
import numpy as np
SC = exu.SystemContainer()
mbs = SC.AddSystem()
#
b0 = mbs.CreateMassPoint(referencePosition = [2,0,0],
                         physicsMass = 1, gravity = [0,-9.81,0],
                         drawSize = 0.5, color=graphics.color.blue)
#
oGround = mbs.AddObject(ObjectGround())
#add vertical spring
oSD = mbs.CreateSpringDamper(bodyOrNodeList=[oGround, b0],
                             localPosition0=[2,1,0],
                             localPosition1=[0,0,0],
                             stiffness=1e4, damping=1e2,
                             drawSize=0.2)
#
mbs.Assemble()
#
[eigenvalues, eigenvectors] = mbs.ComputeODE2Eigenvalues()
#==>eigenvalues contain the eigenvalues of the ODE2 part of the system in the
  current configuration
#
#compute eigenfrequencies in Hz (analytical: 100/2/pi Hz for y-direction):
[eigenvaluesHz, ev] = mbs.ComputeODE2Eigenvalues(convert2Frequencies=True)
#
#compute complex eigenvalues:
[eigenvaluesComplex, ev] = mbs.ComputeODE2Eigenvalues(computeComplexEigenvalues=
  True,
                                                      useAbsoluteValues=False)
```

For examples on ComputeODE2Eigenvalues see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- nMassOscillator.py (Ex), nMassOscillatorEigenmodes.py (Ex), nMassOscillatorInteractive.py (Ex), ANCFBeamEigTest.py (TM), bricardMechanism.py (TM), complexEigenvaluesTest.py (TM), computeODE2AEeigenvaluesTest.py (TM), computeODE2EigenvaluesTest.py (TM), ...

def **ComputeSystemDegreeOfFreedom** (*simulationSettings*= exudyn.SimulationSettings(), *threshold*= 1e-12, *verbose*= False, *useSVD*= False, *autoAssemble*= True)

– **function description**:

compute system DOF numerically, considering Grübler-Kutzbach formula as well as redundant constraints; uses numpy matrix rank or singular value decomposition of scipy (useSVD=True)

- NOTE that this function is added to MainSystem via Python function ComputeSystemDegreeOfFreedom.

– **input**:

simulationSettings: used e.g. for settings regarding numerical differentiation; default settings may be used in most cases

threshold: threshold factor for singular values which estimate the redundant constraints

useSVD: use singular value decomposition directly, also showing SVD values if verbose=True

verbose: if True, it will show the singular values and one may decide if the threshold shall be adapted

autoAssemble: if True: if mbs.systemIsConsistent=False (system is not assembled), call mbs.Assemble() before solver calls

– **output**: dict; returns dictionary with key words 'degreeOfFreedom', 'redundantConstraints', 'nODE2', 'nODE1', 'nAE', 'nPureAE', where: degreeOfFreedom = the system degree of freedom computed numerically, redundantConstraints=the number of redundant constraints, nODE2=number of ODE2 coordinates, nODE1=number of ODE1 coordinates, nAE=total number of constraints, nPureAE=number of constraints on algebraic variables (e.g., lambda=0) that are not coupled to ODE2 coordinates

– **notes**: this approach could possibly fail with special constraints! Currently only works with dense matrices, thus it will be slow for larger systems

– **example**:

```
import exudyn as exu
from exudyn.utilities import *
import numpy as np
SC = exu.SystemContainer()
mbs = SC.AddSystem()
#
b0 = mbs.CreateRigidBody(inertia = InertiaCuboid(density=5000,
                                                 sideLengths=[1,0.1,0.1]),
                         referencePosition = [6,0,0],
                         initialAngularVelocity = [0,8,0],
                         gravity = [0,-9.81,0],
                         graphicsDataList = [exu.graphics.Brick(size=[1,0.1,0.1],
                                                                color=
  graphics.color.orange)])
oGround = mbs.AddObject(ObjectGround())
mbs.CreateGenericJoint(bodyNumbers=[oGround, b0], position=[5.5,0,0],
                       constrainedAxes=[1,1,1, 1,0,0],
```

```
                                rotationMatrixAxes=RotationMatrixX(0.125*pi), #tilt axes
                                useGlobalFrame=True, axesRadius=0.02, axesLength=0.2)
#
mbs.Assemble()
dof = mbs.ComputeSystemDegreeOfFreedom(verbose=1)['degreeOfFreedom'] #print out
  details
```

For examples on ComputeSystemDegreeOfFreedom see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- fourBarMechanism3D.py (Ex), rigidBodyTutorial3.py (Ex), bricardMechanism.py (TM), mainSystemExtensionsTests.py (TM)

def **CreateDistanceSensorGeometry** (*meshPoints*, *meshTrigs*, *rigidBodyMarkerIndex*, *searchTreeCellSize*= [8,8,8])

- **function description**:

  Add geometry for distance sensor given by points and triangles (point indices) to mbs; use a rigid body marker where the geometry is put on;

  Creates a GeneralContact for efficient search on background. If you have several sets of points and trigs, first merge them or add them manually to the contact

  - NOTE that this function is added to MainSystem via Python function CreateDistanceSensorGeometry.

- **input**:

  meshPoints: list of points (3D), as returned by graphics.ToPointsAndTrigs()

  meshTrigs: list of trigs (3 node indices each), as returned by graphics.ToPointsAndTrigs()

  rigidBodyMarkerIndex: rigid body marker to which the triangles are fixed on (ground or moving object)

  searchTreeCellSize: size of search tree (X,Y,Z); use larger values in directions where more triangles are located

- **output**: int; returns ngc, which is the number of GeneralContact in mbs, to be used in CreateDistanceSensor(...); keep the gContact as deletion may corrupt data

- **notes**: should be used by CreateDistanceSensor(...) and AddLidar(...) for simple initialization of GeneralContact; old name: DistanceSensorSetupGeometry(...)

For examples on CreateDistanceSensorGeometry see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- mobileMecanumWheelRobotWithLidar.py (Ex), laserScannerTest.py (TM)

def **CreateDistanceSensor** (*generalContactIndex*, *positionOrMarker*, *dirSensor*, *minDistance*= -1e7, *maxDistance*= 1e7, *cylinderRadius*= 0, *selectedTypeIndex*= exudyn.ContactTypeIndex.IndexEndOfEnumList, *storeInternal*= False, *fileName*= '', *measureVelocity*= False, *addGraphicsObject*= False, *drawDisplaced*= True, *color*= exudyn.graphics.color.red)

- **function description**:

  Function to create distance sensor based on GeneralContact in mbs; sensor can be either placed on absolute position or attached to rigid body marker; in case of marker, dirSensor is relative to the marker

  - NOTE that this function is added to MainSystem via Python function CreateDistanceSensor.

- **input**:

  generalContactIndex: the number of the GeneralContact object in mbs; the index of the GeneralContact object which has been added with last AddGeneralContact(...) command is generalContactIndex=mbs.NumberOfGeneralContacts()-1

  positionOrMarker: either a 3D position as list or np.array, or a MarkerIndex with according rigid body marker

  dirSensor: the direction (no need to normalize) along which the distance is measured (must not be normalized); in case of marker, the direction is relative to marker orientation if marker contains orientation (BodyRigid, NodeRigid)

  minDistance: the minimum distance which is accepted; smaller distance will be ignored

  maxDistance: the maximum distance which is accepted; items being at maxDistance or futher are ignored; if no items are found, the function returns maxDistance

  cylinderRadius: in case of spheres (selectedTypeIndex=ContactTypeIndex.IndexSpheresMarkerBased), a cylinder can be used which measures the shortest distance at a certain radius (geometrically interpreted as cylinder)

  selectedTypeIndex: either this type has default value, meaning that all items in GeneralContact are measured, or there is a specific type index, which is the only type that is considered during measurement

  storeInternal: like with any SensorUserFunction, setting to True stores sensor data internally

  fileName: if defined, recorded data of SensorUserFunction is written to specified file

  measureVelocity: if True, the sensor measures additionally the velocity (component 0=distance, component 1=velocity); velocity is the velocity in direction 'dirSensor' and does not account for changes in geometry, thus it may be different from the time derivative of the distance!

  addGraphicsObject: if True, the distance sensor is also visualized graphically in a simplified manner with a red line having the length of dirSensor; NOTE that updates are ONLY performed during computation, not in visualization; for this reason, solutionSettings.sensorsWritePeriod should be accordingly small

drawDisplaced: if True, the red line is drawn backwards such that it moves along the measured surface; if False, the beam is fixed to marker or position

color: optional color for 'laser beam' to be drawn

- **output**: SensorIndex; creates sensor and returns according sensor number of SensorUserFunction
- **notes**: use generalContactIndex = CreateDistanceSensorGeometry(...) before to create GeneralContact module containing geometry; old name: AddDistanceSensor(...)

For examples on CreateDistanceSensor see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- distanceSensor.py (TM), laserScannerTest.py (TM)

def **DrawSystemGraph** (*showLoads*= True, *showSensors*= True, *useItemNames*= False, *useItemTypes*= False, *addItemTypeNames*= True, *multiLine*= True, *fontSizeFactor*= 1., *layoutDistanceFactor*= 3., *layoutIterations*= 100, *showLegend*= True, *tightLayout*= True)

- **function description**:

  helper function which draws system graph of a MainSystem (mbs); several options let adjust the appearance of the graph; the graph visualization uses randomizer, which results in different graphs after every run!

  - NOTE that this function is added to MainSystem via Python function DrawSystemGraph.

- **input**:

  showLoads: toggle appearance of loads in mbs

  showSensors: toggle appearance of sensors in mbs

  useItemNames: if True, object names are shown instead of basic object types (Node, Load, ...)

  useItemTypes: if True, object type names (MassPoint, JointRevolute, ...) are shown instead of basic object types (Node, Load, ...); Note that Node, Object, is omitted at the beginning of itemName (as compared to theDoc.pdf); item classes become clear from the legend

  addItemTypeNames: if True, type nymes (Node, Load, etc.) are added

  multiLine: if True, labels are multiline, improving readability

  fontSizeFactor: use this factor to scale fonts, allowing to fit larger graphs on the screen with values < 1

  showLegend: shows legend for different item types

  layoutDistanceFactor: this factor influences the arrangement of labels; larger distance values lead to circle-like results

  layoutIterations: more iterations lead to better arrangement of the layout, but need more time for larger systems (use 1000-10000 to get good results)

  tightLayout: if True, uses matplotlib plt.tight_layout() which may raise warning

– **output**: [Any, Any, Any]; returns [networkx, G, items] with nx being networkx, G the graph and item what is returned by nx.draw_networkx_labels(...)

For examples on DrawSystemGraph see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- fourBarMechanism3D.py (Ex), rigidBodyTutorial3.py (Ex), rigidBodyTutorial3withMarkers.py (Ex), mainSystemExtensionsTests.py (TM)

6.5.3 MainSystem: Node

This section provides functions for adding, reading and modifying nodes. Nodes are used to define coordinates (unknowns to the static system and degrees of freedom if constraints are not present). Nodes can provide various types of coordinates for second/first order differential equations (ODE2/ODE1), algebraic equations (AE) and for data (history) variables – which are not providing unknowns in the nonlinear solver but will be solved in an additional nonlinear iteration for e.g. contact, friction or plasticity.

```
import exudyn as exu                #EXUDYN package including C++ core part
from exudyn.itemInterface import * #conversion of data to exudyn dictionaries
SC = exu.SystemContainer()          #container of systems
mbs = SC.AddSystem()                #add a new system to work with
nMP = mbs.AddNode(NodePoint2D(referenceCoordinates=[0,0]))
```

function/structure name	description

AddNode(pyObject)	add a node with nodeDefinition from Python node class; returns (global) node index (type NodeIndex) of newly added node; use int(nodeIndex) to convert to int, if needed (but not recommended in order not to mix up index types of nodes, objects, markers, ...)
	EXAMPLE:
	`item = Rigid2D( referenceCoordinates= [1,0.5,0], initialVelocities= [10,0,0])`
	`mbs.AddNode(item)`
	`nodeDict = {'nodeType': 'Point', 'referenceCoordinates': [1.0, 0.0, 0.0], 'initialCoordinates': [0.0, 2.0, 0.0], 'name': 'example node'}`
	`mbs.AddNode(nodeDict)`

DeleteNode(nodeNumber, suppressWarnings = False)	delete the node with nodeNumber in MainSystem; consistently renames nodes according to their new node numbers; adapts node numbers in sensors and in markers; items using deleted nodeNumber obtain invalid nodeNumber **EXAMPLE**: `mbs.DeleteNode(nodeNumber=42)`

GetNodeNumber(nodeName)	get node's number by name (string) **EXAMPLE:** `n = mbs.GetNodeNumber('example node')`
GetNode(nodeNumber)	get node's dictionary by node number (type NodeIndex) **EXAMPLE:** `nodeDict = mbs.GetNode(0)`
ModifyNode(nodeNumber, nodeDict)	modify node's dictionary by node number (type NodeIndex) **EXAMPLE:** `mbs.ModifyNode(nodeNumber, nodeDict)`

GetNodeDefaults(typeName)	get node's default values for a certain nodeType as (dictionary)
	EXAMPLE:
	`nodeType = 'Point'`
	`nodeDict = mbs.GetNodeDefaults(nodeType)`

GetNodeOutput(nodeNumber, variableType, configuration = exu.ConfigurationType.Current)	get the ouput of the node specified with the OutputVariableType; output may be scalar or array (e.g. displacement vector)
	EXAMPLE:
	`mbs.GetNodeOutput(nodeNumber=0,`
	`variableType=exu.OutputVariableType.Displacement)`

GetNodeODE2Index(nodeNumber)	get index in the global ODE2 coordinate vector for the first node coordinate of the specified node **EXAMPLE:** `mbs.GetNodeODE2Index(nodeNumber=0)`
GetNodeODE1Index(nodeNumber)	get index in the global ODE1 coordinate vector for the first node coordinate of the specified node **EXAMPLE:** `mbs.GetNodeODE1Index(nodeNumber=0)`
GetNodeAEIndex(nodeNumber)	get index in the global AE coordinate vector for the first node coordinate of the specified node **EXAMPLE:** `mbs.GetNodeAEIndex(nodeNumber=0)`
GetNodeParameter(nodeNumber, parameterName)	get nodes's parameter from node number (type NodeIndex) and parameterName; parameter names can be found for the specific items in the reference manual; for visualization parameters, use a 'V' as a prefix **EXAMPLE:** `mbs.GetNodeParameter(0, 'referenceCoordinates')`
SetNodeParameter(nodeNumber, parameterName, value)	set parameter 'parameterName' of node with node number (type NodeIndex) to value; parameter names can be found for the specific items in the reference manual; for visualization parameters, use a 'V' as a prefix **EXAMPLE:** `mbs.SetNodeParameter(0, 'Vshow', True)`

6.5.4 MainSystem: Object

This section provides functions for adding, reading and modifying objects, which can be bodies (mass point, rigid body, finite element, ...), connectors (spring-damper or joint) or general objects. Objects provided terms to the residual of equations resulting from every coordinate given by the nodes. Single-noded objects (e.g. mass point) provides exactly residual terms for its nodal coordinates. Connectors constrain or penalize two markers, which can be, e.g., position, rigid or coordinate markers. Thus, the dependence of objects is either on the coordinates of the marker-objects/nodes or on nodes which the objects possess themselves.

```
import exudyn as exu                #EXUDYN package including C++ core part
from exudyn.itemInterface import * #conversion of data to exudyn dictionaries
SC = exu.SystemContainer()          #container of systems
mbs = SC.AddSystem()                #add a new system to work with
nMP = mbs.AddNode(NodePoint2D(referenceCoordinates=[0,0]))
mbs.AddObject(ObjectMassPoint2D(physicsMass=10, nodeNumber=nMP ))
```

function/structure name	description

AddObject(pyObject)	add an object with objectDefinition from Python object class; returns (global) object number (type ObjectIndex) of newly added object
	EXAMPLE:
	`item = MassPoint(name='heavy object',`
	`nodeNumber=0, physicsMass=100)`
	`mbs.AddObject(item)`
	`objectDict = {'objectType': 'MassPoint',`
	`'physicsMass': 10,`
	`'nodeNumber': 0,`
	`'name': 'example object'}`
	`mbs.AddObject(objectDict)`

DeleteObject(objectNumber, deleteDependentItems = True, suppressWarnings = False)	delete the object with objectNumber in MainSystem; consistently renames objects according to their new object numbers; adapts object numbers in sensors and in markers; items using deleted objectNumber obtain invalid objectNumber; with the option deleteDependentItems (default=True) the function also delete nodes and markers which are used by the object **EXAMPLE**: `mbs.DeleteObject(objectNumber=42)`
GetObjectNumber(objectName)	get object's number by name (string) **EXAMPLE**: `n = mbs.GetObjectNumber('heavy object')`

GetObject(objectNumber, addGraphicsData = False)	get object's dictionary by object number (type ObjectIndex); NOTE: visualization parameters have a prefix 'V'; in order to also get graphicsData written, use addGraphicsData=True (which is by default False, as it would spoil the information) **EXAMPLE**: `objectDict = mbs.GetObject(0)`
ModifyObject(objectNumber, objectDict)	modify object's dictionary by object number (type ObjectIndex); NOTE: visualization parameters have a prefix 'V' **EXAMPLE**: `mbs.ModifyObject(objectNumber, objectDict)`

GetObjectDefaults(typeName)	get object's default values for a certain objectType as (dictionary)
	EXAMPLE:
	`objectType = 'MassPoint'`
	`objectDict = mbs.GetObjectDefaults(objectType)`

GetObjectOutput(objectNumber, variableType, configuration = exu.ConfigurationType.Current)	get object's current output variable from object number (type ObjectIndex) and OutputVariableType; for connectors, it can only be computed for exu.ConfigurationType.Current configuration!
GetObjectOutputBody(objectNumber, variableType, localPosition = [0,0,0], configuration = exu.ConfigurationType.Current)	get body's output variable from object number (type ObjectIndex) and OutputVariableType, using the localPosition as defined in the body, and as used in MarkerBody and SensorBody **EXAMPLE**: `u = mbs.GetObjectOutputBody(objectNumber = 1, variableType = exu.OutputVariableType.Position, localPosition=[1,0,0], configuration = exu.ConfigurationType.Initial)`
GetObjectOutputSuperElement(objectNumber, variableType, meshNodeNumber, configuration = exu.ConfigurationType.Current)	get output variable from mesh node number of object with type SuperElement (GenericODE2, FFRF, FFRFreduced - CMS) with specific OutputVariableType; the meshNodeNumber is the object's local node number, not the global node number! **EXAMPLE**: `u = mbs.GetObjectOutputSuperElement(objectNumber = 1, variableType = exu.OutputVariableType.Position, meshNodeNumber = 12, configuration = exu.ConfigurationType.Initial)`
GetObjectParameter(objectNumber, parameterName)	get objects's parameter from object number (type ObjectIndex) and parameterName; parameter names can be found for the specific items in the reference manual; for visualization parameters, use a 'V' as a prefix; NOTE that BodyGraphicsData cannot be get or set, use dictionary access instead **EXAMPLE**: `mbs.GetObjectParameter(objectNumber = 0, parameterName = 'nodeNumber')`

SetObjectParameter(objectNumber, parameterName, value)	set parameter 'parameterName' of object with object number (type ObjectIndex) to value;; parameter names can be found for the specific items in the reference manual; for visualization parameters, use a 'V' as a prefix; NOTE that BodyGraphicsData cannot be get or set, use dictionary access instead **EXAMPLE**: `mbs.SetObjectParameter(objectNumber = 0, parameterName = 'Vshow', value=True)`

6.5.5 MainSystem: Marker

This section provides functions for adding, reading and modifying markers. Markers define how to measure primal kinematical quantities on objects or nodes (e.g., position, orientation or coordinates themselves), and how to act on the quantities which are dual to the kinematical quantities (e.g., force, torque and generalized forces). Markers provide unique interfaces for loads, sensors and constraints in order to address these quantities independently of the structure of the object or node (e.g., rigid or flexible body).

```
import exudyn as exu                #EXUDYN package including C++ core part
from exudyn.itemInterface import * #conversion of data to exudyn dictionaries
SC = exu.SystemContainer()          #container of systems
mbs = SC.AddSystem()                #add a new system to work with
nMP = mbs.AddNode(NodePoint2D(referenceCoordinates=[0,0]))
mbs.AddObject(ObjectMassPoint2D(physicsMass=10, nodeNumber=nMP ))
mMP = mbs.AddMarker(MarkerNodePosition(nodeNumber = nMP))
```

function/structure name	**description**

AddMarker(pyObject)	add a marker with markerDefinition from Python marker class; returns (global) marker number (type MarkerIndex) of newly added marker
	EXAMPLE:
	`item = MarkerNodePosition(name='my marker',nodeNumber=1)`
	`mbs.AddMarker(item)`
	`markerDict = {'markerType': 'NodePosition',`
	`'nodeNumber': 0,`
	`'name': 'position0'}`
	`mbs.AddMarker(markerDict)`

DeleteMarker(markerNumber, suppressWarnings = False)	delete the marker with markerNumber in MainSystem; consistently renames markers according to their new marker numbers; adapts marker numbers in objects, loads and sensors; items using deleted markerNumber obtain invalid markerNumber **EXAMPLE:** `mbs.DeleteMarker(markerNumber=42)`
GetMarkerNumber(markerName)	get marker's number by name (string) **EXAMPLE:** `n = mbs.GetMarkerNumber('my marker')`
GetMarker(markerNumber)	get marker's dictionary by index **EXAMPLE:** `markerDict = mbs.GetMarker(0)`
ModifyMarker(markerNumber, markerDict)	modify marker's dictionary by index **EXAMPLE:** `mbs.ModifyMarker(markerNumber, markerDict)`

GetMarkerDefaults(typeName)	get marker's default values for a certain markerType as (dictionary)
	EXAMPLE:
	`markerType = 'NodePosition'`
	`markerDict = mbs.GetMarkerDefaults(markerType)`

GetMarkerParameter(markerNumber, parameterName)	get markers's parameter from markerNumber and parameterName; parameter names can be found for the specific items in the reference manual
SetMarkerParameter(markerNumber, parameterName, value)	set parameter 'parameterName' of marker with markerNumber to value; parameter names can be found for the specific items in the reference manual

GetMarkerOutput(markerNumber, variableType, configuration = exu.ConfigurationType.Current)	get the ouput of the marker specified with the OutputVariableType; currently only provides Displacement, Position and Velocity for position based markers, and RotationMatrix, Rotation and AngularVelocity(Local) for markers providing orientation; Coordinates and Coordinates_t available for coordinate markers
	EXAMPLE:
	`mbs.GetMarkerOutput(markerNumber=0,`
	`variableType=exu.OutputVariableType.Position)`

6.5.6 MainSystem: Load

This section provides functions for adding, reading and modifying operating loads. Loads are used to act on the quantities which are dual to the primal kinematic quantities, such as displacement and rotation. Loads represent, e.g., forces, torques or generalized forces.

```
import exudyn as exu                #EXUDYN package including C++ core part
from exudyn.itemInterface import * #conversion of data to exudyn dictionaries
SC = exu.SystemContainer()          #container of systems
mbs = SC.AddSystem()                #add a new system to work with
```

```
nMP = mbs.AddNode(NodePoint2D(referenceCoordinates=[0,0]))
mbs.AddObject(ObjectMassPoint2D(physicsMass=10, nodeNumber=nMP ))
mMP = mbs.AddMarker(MarkerNodePosition(nodeNumber = nMP))
mbs.AddLoad(Force(markerNumber = mMP, loadVector=[0.001,0,0]))
```

function/structure name	description

AddLoad(pyObject)	add a load with loadDefinition from Python load class; returns (global) load number (type LoadIndex) of newly added load
	EXAMPLE:
	`item = mbs.AddLoad(LoadForceVector(loadVector=[1,0,0], markerNumber=0, name='heavy load'))`
	`mbs.AddLoad(item)`
	`loadDict = {'loadType': 'ForceVector', 'markerNumber': 0, 'loadVector': [1.0, 0.0, 0.0], 'name': 'heavy load'}`
	`mbs.AddLoad(loadDict)`

DeleteLoad(loadNumber, deleteDependentMarkers = True, suppressWarnings = False)	delete the load with loadNumber in MainSystem; consistently renames loads according to their new load numbers; deleteDependentMarkers (default=True) also deletes the corresponding marker **EXAMPLE**: `mbs.DeleteLoad(loadNumber=42)`
GetLoadNumber(loadName)	get load's number by name (string) **EXAMPLE**: `n = mbs.GetLoadNumber('heavy load')`
GetLoad(loadNumber)	get load's dictionary by index **EXAMPLE**: `loadDict = mbs.GetLoad(0)`
ModifyLoad(loadNumber, loadDict)	modify load's dictionary by index **EXAMPLE**: `mbs.ModifyLoad(loadNumber, loadDict)`

GetLoadDefaults(typeName)	get load's default values for a certain loadType as (dictionary)
	EXAMPLE:
	`loadType = 'ForceVector'`
	`loadDict = mbs.GetLoadDefaults(loadType)`

GetLoadValues(loadNumber)	Get current load values, specifically if user-defined loads are used; can be scalar or vector-valued return value
GetLoadParameter(loadNumber, parameterName)	get loads's parameter from loadNumber and parameterName; parameter names can be found for the specific items in the reference manual
SetLoadParameter(loadNumber, parameterName, value)	set parameter 'parameterName' of load with loadNumber to value; parameter names can be found for the specific items in the reference manual

6.5.7 MainSystem: Sensor

This section provides functions for adding, reading and modifying operating sensors. Sensors are used to measure information in nodes, objects, markers, and loads for output in a file.

```
import exudyn as exu                #EXUDYN package including C++ core part
from exudyn.itemInterface import * #conversion of data to exudyn dictionaries
SC = exu.SystemContainer()          #container of systems
mbs = SC.AddSystem()                #add a new system to work with
nMP = mbs.AddNode(NodePoint(referenceCoordinates=[0,0,0]))
mbs.AddObject(ObjectMassPoint(physicsMass=10, nodeNumber=nMP ))
mMP = mbs.AddMarker(MarkerNodePosition(nodeNumber = nMP))
mbs.AddLoad(Force(markerNumber = mMP, loadVector=[2,0,5]))
sMP = mbs.AddSensor(SensorNode(nodeNumber=nMP, storeInternal=True,
                               outputVariableType=exu.OutputVariableType.Position))
mbs.Assemble()
mbs.SolveDynamic(exu.SimulationSettings())
from exudyn.plot import PlotSensor
PlotSensor(mbs, sMP, components=[0,1,2])
```

function/structure name	description

AddSensor(pyObject)	add a sensor with sensor definition from Python sensor class; returns (global) sensor number (type SensorIndex) of newly added sensor
	EXAMPLE:
	`item = mbs.AddSensor(SensorNode(sensorType=`
	`exu.SensorType.Node, nodeNumber=0, name='test`
	`sensor'))`
	`mbs.AddSensor(item)`
	`sensorDict = {'sensorType': 'Node',`
	`'nodeNumber': 0,`
	`'fileName': 'sensor.txt',`
	`'name': 'test sensor'}`
	`mbs.AddSensor(sensorDict)`

DeleteSensor(sensorNumber, suppressWarnings = False)	delete the marker with sensorNumber in MainSystem; consistently renames sensors according to their new sensor numbers; adapts sensor numbers in sensors; items using deleted sensorNumber obtain invalid sensorNumber **EXAMPLE:** `mbs.DeleteSensor(sensorNumber=42)`
GetSensorNumber(sensorName)	get sensor's number by name (string) **EXAMPLE:** `n = mbs.GetSensorNumber('test sensor')`
GetSensor(sensorNumber)	get sensor's dictionary by index **EXAMPLE:** `sensorDict = mbs.GetSensor(0)`

ModifySensor(sensorNumber, sensorDict)	modify sensor's dictionary by index **EXAMPLE:** `mbs.ModifySensor(sensorNumber, sensorDict)`

GetSensorDefaults(typeName)	get sensor's default values for a certain sensorType as (dictionary)
	EXAMPLE:
	`sensorType = 'Node'`
	`sensorDict = mbs.GetSensorDefaults(sensorType)`

GetSensorValues(sensorNumber, configuration = exu.ConfigurationType.Current)	get sensors's values for configuration; can be a scalar or vector-valued return value!
GetSensorStoredData(sensorNumber)	get sensors's internally stored data as matrix (all time points stored); rows are containing time and sensor values as obtained by sensor (e.g., time, and x, y, and z value of position)
GetSensorParameter(sensorNumber, parameterName)	get sensors's parameter from sensorNumber and parameterName; parameter names can be found for the specific items in the reference manual
SetSensorParameter(sensorNumber, parameterName, value)	set parameter 'parameterName' of sensor with sensorNumber to value; parameter names can be found for the specific items in the reference manual

6.6 SystemData

This is the data structure of a system which contains Objects (bodies/constraints/...), Nodes, Markers and Loads. The SystemData structure allows advanced access to this data, which HAS TO BE USED WITH CARE, as unexpected results and system crash might happen.

```
import exudyn as exu                #EXUDYN package including C++ core part
from exudyn.itemInterface import * #conversion of data to exudyn dictionaries
SC = exu.SystemContainer()          #container of systems
mbs = SC.AddSystem()                #add a new system to work with
nMP = mbs.AddNode(NodePoint(referenceCoordinates=[0,0,0]))
mbs.AddObject(ObjectMassPoint(physicsMass=10, nodeNumber=nMP ))
mMP = mbs.AddMarker(MarkerNodePosition(nodeNumber = nMP))
mbs.AddLoad(Force(markerNumber = mMP, loadVector=[2,0,5]))
mbs.Assemble()
mbs.SolveDynamic(exu.SimulationSettings())

#obtain current ODE2 system vector including reference values:
uTotal = mbs.systemData.GetODE2CoordinatesTotal()

#obtain current ODE2 system vector without reference values
#  (e.g. after static simulation finished):
u = mbs.systemData.GetODE2Coordinates()
#set initial ODE2 vector for next simulation (only coordinates!):
```

```
mbs.systemData.SetODE2Coordinates(coordinates=u,
                configuration=exu.ConfigurationType.Initial)

#faster access with reference access (copy=False):
u3 = mbs.systemData.GetODE2Coordinates(copy=False)[3]
#we can also modify data, but this may be dangerous!
u3 += 1
#NOTE: reference access is possible throughout simulation and may
#      allow faster user functions, but is potentially dangerous
#      to erroneous behavior: for safety, compare with copy=True results!

#print detailed information on items:
mbs.systemData.Info()
#print LTG lists for objects and loads:
mbs.systemData.InfoLTG()
```

function/structure name	**description**
NumberOfLoads()	return number of loads in system **EXAMPLE:** `print(mbs.systemData.NumberOfLoads())`
NumberOfMarkers()	return number of markers in system **EXAMPLE:** `print(mbs.systemData.NumberOfMarkers())`
NumberOfNodes()	return number of nodes in system **EXAMPLE:** `print(mbs.systemData.NumberOfNodes())`
NumberOfObjects()	return number of objects in system **EXAMPLE:** `print(mbs.systemData.NumberOfObjects())`
NumberOfSensors()	return number of sensors in system **EXAMPLE:** `print(mbs.systemData.NumberOfSensors())`
ODE2Size(configurationType = exu.ConfigurationType.Current)	get size of ODE2 coordinate vector for given configuration (only works correctly after mbs.Assemble()) **EXAMPLE:** `print('ODE2 size=',mbs.systemData.ODE2Size())`
ODE1Size(configurationType = exu.ConfigurationType.Current)	get size of ODE1 coordinate vector for given configuration (only works correctly after mbs.Assemble()) **EXAMPLE:** `print('ODE1 size=',mbs.systemData.ODE1Size())`
AEsize(configurationType = exu.ConfigurationType.Current)	get size of AE coordinate vector for given configuration (only works correctly after mbs.Assemble()) **EXAMPLE:** `print('AE size=',mbs.systemData.AEsize())`
DataSize(configurationType = exu.ConfigurationType.Current)	get size of Data coordinate vector for given configuration (only works correctly after mbs.Assemble()) **EXAMPLE:** `print('Data size=',mbs.systemData.DataSize())`

SystemSize(configurationType = exu.ConfigurationType.Current)	get size of System coordinate vector for given configuration (only works correctly after mbs.Assemble()) **EXAMPLE:** `print('System size=',mbs.systemData.SystemSize())`
GetTime(configurationType = exu.ConfigurationType.Current)	get configuration dependent time. **EXAMPLE:** `mbs.systemData.GetTime(exu.ConfigurationType.Initial)`

SetTime(newTime, configurationType = exu.ConfigurationType.Current)	set configuration dependent time; use this access with care, e.g. in user-defined solvers.
	EXAMPLE:
	`mbs.systemData.SetTime(10.,`
	`exu.ConfigurationType.Initial)`

AddODE2LoadDependencies(loadNumber, globalODE2coordinates)	advanced function for adding special dependencies of loads onto ODE2 coordinates, taking a list / numpy array of global ODE2 coordinates; this function needs to be called after Assemble() and needs to contain global ODE2 coordinate indices; this list only affects implicit or static solvers if timeIntegration.computeLoadsJacobian or staticSolver.computeLoadsJacobian is set to 1 (ODE2) or 2 (ODE2 and ODE2_t dependencies); if set, it may greatly improve convergence if loads with user functions depend on some system states, such as in a load with feedback control loop; the additional dependencies are not required, if doSystemWideDifferentiation=True, however the latter option being much less efficient. For more details, consider the file doublePendulum2DControl.py in the examples directory.
	EXAMPLE:
	`mbs.systemData.AddODE2LoadDependencies(0,[0,1,2])`
	`#add dependency of load 5 onto node 2 coordinates:`
	`nodeLTG2 = mbs.systemData.GetNodeLTGODE2(2)`
	`mbs.systemData.AddODE2LoadDependencies(5,nodeLTG2)`

Info()	print detailed information on every item; for short information use print(mbs) **EXAMPLE:** `mbs.systemData.Info()`
InfoLTG()	print LTG information of objects and load dependencies **EXAMPLE:** `mbs.systemData.InfoLTG()`

6.6.1 SystemData: Access coordinates

This section provides access functions to global coordinate vectors. Assigning invalid values or using wrong vector size might lead to system crash and unexpected results.

function/structure name	**description**

GetODE2CoordinatesTotal(configuration = exu.ConfigurationType.Current)	get ODE2 system coordinates (displacements/rotation) including reference values for given configuration (default: exu.Configuration.Current); in case of exu.ConfigurationType.Reference, it only includes reference values once and is identical to GetODE2Coordinates; note that faster access to coordinates is possibly with GetODE2Coordinates(copy=False), which is not possible with GetODE2CoordinatesTotal !
	EXAMPLE:
	`uTotal = mbs.systemData.GetODE2CoordinatesTotal()`
	`#this is equivalent to:`
	`uTotal=mbs.systemData.GetODE2Coordinates()+mbs.systemData.GetO`

GetODE2Coordinates(configuration = exu.ConfigurationType.Current, copy = True)	get ODE2 system coordinates (displacements/rotations) for given configuration (default: exu.Configuration.Current) **EXAMPLE:** `uCurrent = mbs.systemData.GetODE2Coordinates()`
SetODE2Coordinates(coordinates, configuration = exu.ConfigurationType.Current)	set ODE2 system coordinates (displacements/rotations) for given configuration (default: exu.Configuration.Current); invalid vector size may lead to system crash! **EXAMPLE:** `mbs.systemData.SetODE2Coordinates(uCurrent)`
GetODE2Coordinates_t(configuration = exu.ConfigurationType.Current, copy = True)	get ODE2 system coordinates (velocities) for given configuration (default: exu.Configuration.Current) **EXAMPLE:** `vCurrent = mbs.systemData.GetODE2Coordinates_t()`
SetODE2Coordinates_t(coordinates, configuration = exu.ConfigurationType.Current)	set ODE2 system coordinates (velocities) for given configuration (default: exu.Configuration.Current); invalid vector size may lead to system crash! **EXAMPLE:** `mbs.systemData.SetODE2Coordinates_t(vCurrent)`
GetODE2Coordinates_tt(configuration = exu.ConfigurationType.Current, copy = True)	get ODE2 system coordinates (accelerations) for given configuration (default: exu.Configuration.Current) **EXAMPLE:** `vCurrent = mbs.systemData.GetODE2Coordinates_tt()`
SetODE2Coordinates_tt(coordinates, configuration = exu.ConfigurationType.Current)	set ODE2 system coordinates (accelerations) for given configuration (default: exu.Configuration.Current); invalid vector size may lead to system crash! **EXAMPLE:** `mbs.systemData.SetODE2Coordinates_tt(aCurrent)`
GetODE1Coordinates(configuration = exu.ConfigurationType.Current, copy = True)	get ODE1 system coordinates (displacements) for given configuration (default: exu.Configuration.Current) **EXAMPLE:** `qCurrent = mbs.systemData.GetODE1Coordinates()`
SetODE1Coordinates(coordinates, configuration = exu.ConfigurationType.Current)	set ODE1 system coordinates (velocities) for given configuration (default: exu.Configuration.Current); invalid vector size may lead to system crash! **EXAMPLE:** `mbs.systemData.SetODE1Coordinates_t(qCurrent)`

GetODE1Coordinates_t(configuration = exu.ConfigurationType.Current, copy = True)	get ODE1 system coordinates (velocities) for given configuration (default: exu.Configuration.Current) **EXAMPLE:** `qCurrent = mbs.systemData.GetODE1Coordinates_t()`
SetODE1Coordinates_t(coordinates, configuration = exu.ConfigurationType.Current)	set ODE1 system coordinates (displacements) for given configuration (default: exu.Configuration.Current); invalid vector size may lead to system crash! **EXAMPLE:** `mbs.systemData.SetODE1Coordinates(qCurrent)`
GetAECoordinates(configuration = exu.ConfigurationType.Current, copy = True)	get algebraic equations (AE) system coordinates for given configuration (default: exu.Configuration.Current) **EXAMPLE:** `lambdaCurrent = mbs.systemData.GetAECoordinates()`
SetAECoordinates(coordinates, configuration = exu.ConfigurationType.Current)	set algebraic equations (AE) system coordinates for given configuration (default: exu.Configuration.Current); invalid vector size may lead to system crash! **EXAMPLE:** `mbs.systemData.SetAECoordinates(lambdaCurrent)`
GetDataCoordinates(configuration = exu.ConfigurationType.Current, copy = True)	get system data coordinates for given configuration (default: exu.Configuration.Current) **EXAMPLE:** `dataCurrent = mbs.systemData.GetDataCoordinates()`
SetDataCoordinates(coordinates, configuration = exu.ConfigurationType.Current)	set system data coordinates for given configuration (default: exu.Configuration.Current); invalid vector size may lead to system crash! **EXAMPLE:** `mbs.systemData.SetDataCoordinates(dataCurrent)`
GetSystemState(configuration = exu.ConfigurationType.Current)	get system state for given configuration (default: exu.Configuration.Current); state vectors do not include the non-state derivatives ODE1_t and ODE2_tt and the time; function is copying data - not highly efficient; format of pyList: [ODE2Coords, ODE2Coords_t, ODE1Coords, AEcoords, dataCoords] **EXAMPLE:** `sysStateList = mbs.systemData.GetSystemState()`
SetSystemState(systemStateList, configuration = exu.ConfigurationType.Current)	set system data coordinates for given configuration (default: exu.Configuration.Current); invalid list of vectors / vector size may lead to system crash; write access to state vectors (but not the non-state derivatives ODE1_t and ODE2_tt and the time); function is copying data - not highly efficient; format of pyList: [ODE2Coords, ODE2Coords_t, ODE1Coords, AEcoords, dataCoords] **EXAMPLE:** `mbs.systemData.SetSystemState(sysStateList, configuration = exu.ConfigurationType.Initial)`

GetSystemStateDict(configuration = exu.ConfigurationType.Current, reference = False)	get dictionary with copies of (or references to) system states for given configuration (default: exu.Configuration.Current), with at least the following quantities: ODE1Coords, ODE1Coords_t, ODE2Coords, ODE2Coords_t, ODE2Coords_tt, AECoords, dataCoords; we can obtain copies OR references to vectors without copying, meaning that these vectors then have read-write properties and have to be treated carefully! The dictionary's contents are subject to changes in the future; if reference=False, data is copied **EXAMPLE:** `d = mbs.systemData.GetSystemStateDict()`

6.6.2 SystemData: Get object LTG coordinate mappings

This section provides access functions the LTG-lists for every object (body, constraint, ...) in the system. For details on the LTG mapping, see Section 2.3.

function/structure name	**description**
GetObjectLTGODE2(objectNumber)	get object local-to-global coordinate mapping (list of global coordinate indices) for ODE2 coordinates; only available after Assemble() **EXAMPLE:** `ltgObject4 = mbs.systemData.GetObjectLTGODE2(4)`
GetObjectLTGODE1(objectNumber)	get object local-to-global coordinate mapping (list of global coordinate indices) for ODE1 coordinates; only available after Assemble() **EXAMPLE:** `ltgObject4 = mbs.systemData.GetObjectLTGODE1(4)`
GetObjectLTGAE(objectNumber)	get object local-to-global coordinate mapping (list of global coordinate indices) for algebraic equations (AE) coordinates; only available after Assemble() **EXAMPLE:** `ltgObject4 = mbs.systemData.GetObjectLTGAE(4)`
GetObjectLTGData(objectNumber)	get object local-to-global coordinate mapping (list of global coordinate indices) for data coordinates; only available after Assemble() **EXAMPLE:** `ltgObject4 = mbs.systemData.GetObjectLTGData(4)`
GetNodeLTGODE2(nodeNumber)	get node local-to-global coordinate mapping (list of global coordinate indices) for ODE2 coordinates; only available after Assemble() **EXAMPLE:** `ltgNode4 = mbs.systemData.GetNodeLTGODE2(4)`

GetNodeLTGODE1(nodeNumber)	get node local-to-global coordinate mapping (list of global coordinate indices) for ODE1 coordinates; only available after Assemble() **EXAMPLE:** `ltgNode4 = mbs.systemData.GetNodeLTGODE1(4)`
GetNodeLTGAE(nodeNumber)	get node local-to-global coordinate mapping (list of global coordinate indices) for AE coordinates; only available after Assemble() **EXAMPLE:** `ltgNode4 = mbs.systemData.GetNodeLTGAE(4)`
GetNodeLTGData(nodeNumber)	get node local-to-global coordinate mapping (list of global coordinate indices) for Data coordinates; only available after Assemble() **EXAMPLE:** `ltgNode4 = mbs.systemData.GetNodeLTGData(4)`

6.7 Symbolic

The Symbolic sub-module in `exudyn.symbolic` allows limited symbolic manipulations in `Exudyn` and is currently under development In particular, symbolic user functions can be created, which allow significant speedup of Python user functions. However, **always veryfy your symbolic expressions or user functions**, as behavior may be unexpected in some cases.

6.7.1 symbolic.Real

The symbolic Real type allows to replace Python's float by a symbolic quantity. The `symbolic.Real` may be directly set to a float and be evaluated as float. However, turing on recording by using extttexudyn.symbolic.SetRecording(True) (on by default), results are stored as expression trees, which may be evaluated in C++ or Python, in particular in user functions, see the following example:

```
import exudyn as exu
esym = exu.symbolic     #abbreviation
SymReal = esym.Real     #abbreviation

#create some variables
a = SymReal('a',42.)    #use named expression
b = SymReal(13)         #b is 13
c = a+b*7.+1.-3         #c stores expression tree
d = c                   #d and c are containing same tree!
print('a: ',a,' = ',a.Evaluate())
print('c: ',c,' = ',c.Evaluate())

#use special functions:
d = a+b*esym.sin(a)+esym.cos(SymReal(7))
```

```
print('d: ',d,' = ',d.Evaluate())

a.SetValue(14)          #variable a set to new value; influences d
print('d: ',d,' = ',d.Evaluate())

a = SymReal(1000)       #a is now a new variable; not updated in d!
print('d: ',d,' = ',d.Evaluate())

#compute derivatives (automatic differentiation):
x = SymReal("x",0.5)
f = a+b*esym.sin(x)+esym.cos(SymReal(7))+x**4
print('f=',f.Evaluate(), ', diff=',f.Diff(x))

#turn off recording of trees (globally for all symbolic.Real!):
esym.SetRecording(False)
x = SymReal(42) #now, only represents a value
y = x/3.        #directly evaluates to 14

#back to default behavior, otherwise expr. only evaluated:
esym.SetRecording(True)
```

To create a symbolic Real, use `aa=symbolic.Real(1.23)` to build a Python object aa with value 1.23. In order to use a named value, use `pi=symbolic.Real('pi',3.14)`. Note that in the following, we use the abbreviation `SymReal=exudyn.symbolic.Real`. Member functions of `SymReal`, which are **not recorded**, are:

function/structure name	**description**
__init__(value)	Construct symbolic.Real from float.
__init__(name, value)	Construct named symbolic.Real from name and float.

SetValue(valueInit)	Set either internal float value or value of named expression; cannot change symbolic expressions.
	EXAMPLE:
	`b = SymReal(13)`
	`b.SetValue(14) #now b is 14`
	`#b.SetValue(a+3.)  #not possible!`

Evaluate()	return evaluated expression (prioritized) or stored Real value.

Diff(var)	(UNTESTED!) return derivative of stored expression with respect to given symbolic named variable; NOTE: when defining the expression of the variable which shall be differentiated, the variable may only be changed with the SetValue(...) method hereafter!
	EXAMPLE:
	`x=SymReal('x',2)`
	`f=3*x+x**2*sin(x)`
	`f.Diff(x) #evaluate derivative w.r.t.  x`

value	access to internal float value, which is used in case that symbolic.Real has been built from a float (but without a name and without symbolic expression)
operator __float__()	evaluation of expression and conversion of symbolic.Real to Python float
operator __str__()	conversion of symbolic.Real to string
operator __repr__()	representation of symbolic.Real in Python

The remaining operators and mathematical functions are recorded within expressions. Main mathematical operators for `SymReal` exist, similar to Python, such as:

```
a = SymReal(1)
b = SymReal(2)

r1 = a+b
r1 = a-b
r1 = a*b
r1 = a/b
r1 = -a
r1 = a**b

c = SymReal(3.3)
c += b
c -= b
c *= b
c /= b

c = (a == b)
c = (a != b)
c = (a < b)
c = (a > b)
c = (a <= b)
c = (a >= b)

#in most cases, we can also mix with float:
c = a*7 + SymReal.sin(8)
```

Mathematical functions may be called with an `SymReal` or with a `float`. Most standard mathematical functions exist for `symbolic`, e.g., as `symbolic.abs`. **HINT**: function names are lower-case for compatibility with Python's math library. Thus, you can easily exchange math.sin with esym.sin, and you may want to use a generic name, such as myMath=symbolic in order to switch between Python and symbolic user functions. The following functions exist:

function/structure name	**description**

isfinite(x)	according to specification of C++ std::isfinite
abs(x)	according to specification of C++ std::fabs
round(x)	according to specification of C++ std::round
ceil(x)	according to specification of C++ std::ceil
floor(x)	according to specification of C++ std::floor
sqrt(x)	according to specification of C++ std::sqrt
exp(x)	according to specification of C++ std::exp
log(x)	according to specification of C++ std::log
sin(x)	according to specification of C++ std::sin
cos(x)	according to specification of C++ std::cos
tan(x)	according to specification of C++ std::tan
asin(x)	according to specification of C++ std::asin
acos(x)	according to specification of C++ std::acos
atan(x)	according to specification of C++ std::atan
sinh(x)	according to specification of C++ std::sinh
cosh(x)	according to specification of C++ std::cosh
tanh(x)	according to specification of C++ std::tanh
asinh(x)	according to specification of C++ std::asinh
acosh(x)	according to specification of C++ std::acosh
atanh(x)	according to specification of C++ std::atanh

The following table lists special functions for `SymReal`:

function/structure name	**description**
sign(x)	returns 0 for x=0, -1 for x<0 and 1 for x>1.
Not(x)	returns logical not of expression, equal to Python's 'not'. Not(True)=False, Not(0.)=True, Not(-0.1)=False
min(x, y)	return minimum of x and y.
max(x, y)	return maximum of x and y.
mod(x, y)	return floating-point remainder of the division operation x / y. For example, mod(5.1, 3) gives 2.1 as a remainder.
pow(x, y)	return x^y.
max(x, y)	return maximum of x and y.

IfThenElse(condition, ifTrue, ifFalse)	Symbolic function for conditional evaluation. If the condition evaluates to True, the expression ifTrue is evaluated, while otherwise expression ifFalse is evaluated
	EXAMPLE:
	`x=SymReal(-1)`
	`y=SymReal(2,'y')`
	`a=SymReal.IfThenElse(x<0, y+1, y-1))`

SetRecording(flag)	Set current (global / module-wide) status of expression recording. By default, recording is on. **EXAMPLE**: `SymReal.SetRecording(True)`

GetRecording()	Get current (global / module-wide) status of expression recording. **EXAMPLE**: `symbolic.Real.GetRecording()`

6.7.2 symbolic.Vector

A symbolic Vector type to replace Python's (1D) numpy array in symbolic expressions. The `symbolic.Vector` may be directly set to a list of floats or (1D) numpy array and be evaluated as array. However, turing on recording by using extttexudyn.symbolic.SetRecording(True) (on by default), results are stored as expression trees, which may be evaluated in C++ or Python, in particular in user functions, see the following example:

```
import exudyn as exu
import numpy as np
esym = exu.symbolic

SymVector = esym.Vector
SymReal = esym.Real

a = SymReal('a',42.)
b = SymReal(13)
c = a-3*b

#create from list:
v1 = SymVector([1,3,2])
print('v1: ',v1)

#create from numpy array:
v2 = SymVector(np.array([1,3,2]))
print('v2 initial: ',v2)

#create from list, mixing symbolic expressions and numbers:
v2 = SymVector([a,42,c])

print('v2 now: ',v2,"=",v2.Evaluate())
print('v1+v2: ',v1+v2,"=",(v1+v2).Evaluate()) #evaluate as vector

print('v1*v2: ',v1*v2,"=",(v1*v2).Evaluate()) #evaluate as Real

#access of vector component:
print('v1[2]: ',v1[2],"=",v1[2].Evaluate())   #evaluate as Real
```

To create a symbolic Vector, use `aa=symbolic.Vector([3,4.2,5]` to build a Python object aa with values [3,4.2,5]. In order to use a named vector, use `v=symbolic.Vector('myVec',[3,4.2,5])`. Vec-

tors can be also created from mixed symbolic expressions and numbers, such as `v=symbolic.Vector([x,x**2,3.1` however, this cannot become a named vector as it contains expressions. There is a significance difference to numpy, such that '*' represents the scalar vector multplication which gives a scalar. Furthermore, the comparison operator '==' gives only True, if all components are equal, and the operator '!=' gives True, if any component is unequal. Note that in the following, we use the abbreviation `SymVector=exudyn.symbolic.Vector`. Note that only functions are able to be recorded. Member functions of `SymVector` are:

function/structure name	**description**
__init__(vector)	Construct symbolic.Vector from vector represented as numpy array or list (which may contain symbolic expressions).
__init__(name, vector)	Construct named symbolic.Vector from name and vector represented as numpy array or list (which may contain symbolic expressions).
Evaluate()	Return evaluated expression (prioritized) or stored vector value. (not recorded)
SetVector(vector)	Set stored vector or named vector expression to new given (non-symbolic) vector. Only works, if SymVector contains no expression. (may lead to inconsistencies in recording)
NumberOfItems()	Get size of Vector (may require to evaluate expression; not recording)
operator __setitem__(index)	bracket [] operator for setting a component of the vector. Only works, if SymVector contains no expression. (may lead to inconsistencies in recording)

NormL2()	return (symbolic) L2-norm of vector.
	EXAMPLE:
	`v1 = SymVector([1,4,8])`
	`length = v1.NormL2() #gives 9.`

MultComponents(other)	Perform component-wise multiplication of vector times other vector and return result. This corresponds to the numpy multiplication using '*'.
	EXAMPLE:
	`v1 = SymVector([1,2,4])`
	`v2 = SymVector([1,0.5,0.25])`
	`v3 = v1.MultComponents(v2)`

operator __getitem__(index)	bracket [] operator to return (symbolic) component of vector, allowing read-access. Index may also evaluate from an expression.
operator __str__()	conversion of SymVector to string
operator __repr__()	representation of SymVector in Python

Standard vector operators are available for `SymVector`, see the following examples:

```
v = SymVector([1,3,2])
w = SymVector([3.3,2.2,1.1])
```

```
u = v+w
u = v-w
u = -v
#scalar multiplication; evaluates to SymReal:
x = v*w
#NOTE: component-wise multiplication, returns SymVector:
u = v.MultComponents(w)

#inplace operators:
v += w
v -= w
v *= SymReal(0.5)
```

6.7.3 symbolic.Matrix

A symbolic Matrix type to replace Python's (2D) numpy array in symbolic expressions. The `symbolic.Matrix` may be directly set to a list of list of floats or (2D) numpy array and be evaluated as array. However, turing on recording by using extttexudyn.symbolic.SetRecording(True) (on by default), results are stored as expression trees, which may be evaluated in C++ or Python, in particular in user functions, see the following example:

```
import exudyn as exu
import numpy as np
esym = exu.symbolic

SymMatrix = esym.Matrix
SymReal = esym.Real

a = SymReal('a',42.)
b = SymReal(13)

#create matrix from list of lists
m1 = SymMatrix([[1,3,2],[4,5,6]])

#create symbolic matrix from list of lists
m3 = SymMatrix([[a,3*b,2],[4,5,6]])

#create from numpy array
m2 = SymMatrix(np.ones((3,3))-np.eye(3))

m1 += m3
m1 *= 3
m1 -= 3*m3
print('m1: ',m1)
print('m2: ',m2)
```

To create a symbolic Matrix, use `aa=symbolic.Matrix([[3,4.2],[3.3,1.2]]` to build a Python object aa. In order to use a named matrix, use `v=symbolic.Matrix('myMat',[3,4.2,5])`. Matrixs can be also created from mixed symbolic expressions and numbers, such as `v=symbolic.Matrix([x,x**2,3.14])`, however, this cannot become a named matrix as it contains expressions. There is a significance difference to numpy, such that '*' represents the matrix multplication (compute components from row times column operations). Note that in the following, we use the abbreviation `SymMatrix=exudyn.symbolic.Matrix`. Member functions of `SymMatrix` are:

function/structure name	**description**
__init__(matrix)	Construct symbolic.Matrix from vector represented as numpy array or list of lists (which may contain symbolic expressions).
__init__(name, matrix)	Construct named symbolic.Matrix from name and vector represented as numpy array or list of lists (which may contain symbolic expressions).
Evaluate()	Return evaluated expression (prioritized) or stored Matrix value. (not recorded)
SetMatrix(matrix)	Set stored Matrix or named Matrix expression to new given (non-symbolic) Matrix. Only works, if SymMatrix contains no expression. (may lead to inconsistencies in recording)
NumberOfRows()	Get number of rows (may require to evaluate expression; not recording)
NumberOfColumns()	Get number of columns (may require to evaluate expression; not recording)
operator __setitem__(row, column)	bracket [] operator for (symbolic) component of Matrix (write-access). Only works, if SymMatrix contains no expression. (may lead to inconsistencies in recording)
operator __getitem__(row, column)	bracket [] operator for (symbolic) component of Matrix (read-access). Row and column may also evaluate from an expression.
operator __str__()	conversion of SymMatrix to string
operator __repr__()	representation of SymMatrix in Python

Standard Matrix operators are available for `SymMatrix`, see the following examples:

```
m1 = SymMatrix([[1,7],[4,5]])
m2 = SymMatrix([[1,2.2],[4,4.3]])
v = SymVector([1.5,3])

m3 = m1+m2
m3 = m1-m2
m3 = m1*m2

#multiply with scalar
m3 = 13*m2
```

```
m3 = m2*3.14

#multiply with vector
m3 = m2*v

#transposed:
m3 = v*m2 #equals numpy operation m2.T @ v

#inplace operators:
m1 += m1
m1 -= m1
m1 *= 3.14
```

6.7.4 symbolic.VariableSet

A container for symbolic variables, in particular for exchange between user functions and the model. For details, see the following example:

```
import exudyn as exu
import math
SymReal = exu.symbolic.Real

#use global variable set:
variables = exu.symbolic.variables

#create a named Real
a = SymReal('a',42.)

#regular way to add variable:
variables.Add('pi', math.pi)

#add named variable (doesn't need a name):
variables.Add(a)

#print current variable set
print(variables)

print('pi=',variables.Get('pi').Evaluate()) #3.14
print('a=',variables.Get('a')) #prints 'a'

x=variables.Get('a')
print('x=',x.Evaluate()) #x=42

#override a
variables.Set('a',3.33)
```

```
#x is depending on a:
print('x:',x,"=",x.Evaluate()) #3.33

#create your own variable set
mySet = esym.VariableSet()
```

function/structure name	description
Add(name, value)	Add a variable with name and value (name may not exist)
Add(namedReal)	Add a variable with named real (name may not exist)
Set(name, value)	Set a variable with name and value (adds new or overrides existing)
Get(name)	Get a variable by name
Exists(name)	Return True, if variable name exists
Reset()	Erase all variables and reset VariableSet
NumberOfItems(name)	Return True, if variable name exists
GetNames()	Get list of stored variable names
data[index]= ...name, value	bracket [] operator for setting a variable to a specific value
... = data[index]name	bracket [] operator for getting a specific variable by name
operator __str__()	create string of set of variables
operator __repr__()	representation of SymMatrix in Python

6.7.5 symbolic.UserFunction

A class for creating and handling symbolic user functions in C++. Use these functions for high performance extensions, e.g., of existing objects or loadsFor details, see the following example:

```
import exudyn as exu
esym = exu.symbolic
from exudyn.utilities import * #advancedUtilities with user function utilities included
SymReal = exu.symbolic.Real

SC = exu.SystemContainer()
mbs = SC.AddSystem()

#regular Python user function with esym math functions
def UFload(mbs, t, load):
    return load*esym.sin(10*(2*pi)*t)

#create symbolic user function from Python user function:
symFuncLoad = CreateSymbolicUserFunction(mbs, UFload, load, 'loadUserFunction',verbose=1)

#add ground and mass point:
oGround = mbs.CreateGround()
oMassPoint = mbs.CreateMassPoint(referencePosition=[1.+0.05,0,0], physicsMass=1)
```

```
#add marker and load:
mc = mbs.AddMarker(MarkerNodeCoordinate(nodeNumber=mbs.GetObject(oMassPoint)['nodeNumber'],
    coordinate=0))
load = mbs.AddLoad(LoadCoordinate(markerNumber=mc, load=10,
                                  loadUserFunction=symFuncLoad))

#print string of symbolic expression of user function (to check if it looks ok):
print('load user function: ',symFuncLoad)

#test evaluate user function; requires args of user function:
print('load user function: ',symFuncLoad.Evaluate(mbs, 0.025, 10.))

#now you could add further items or simulate ...
```

function/structure name	description
Evaluate()	Evaluate symbolic function with test values; requires exactly same args as Python user functions; this is slow and only intended for testing
SetUserFunctionFromDict(mainSystem, fcnDict, itemIndex, userFunctionName)	Create C++ std::function (as requested in C++ item) with symbolic user function as recorded in given dictionary, as created with ConvertFunctionToSymbolic(...).
operator __repr__()	Representation of Symbolic function
operator __str__()	Convert stored symbolic function to string

6.8 GeneralContact

Structure to define general and highly efficient contact functionality in multibody systems[1]. For further explanations and theoretical backgrounds, see Section 5.6. Internally, the contacts are stored with global indices, which are in the following list: [numberOfSpheresMarkerBased, numberOfANCFCable2D, numberOfTrigsRigidBodyBased], see alsothe output of GetPythonObject().

```
#...
#code snippet, must be placed anywhere before mbs.Assemble()
#Add GeneralContact to mbs:
gContact = mbs.AddGeneralContact()
#Add contact elements, e.g.:
gContact.AddSphereWithMarker(...) #use appropriate arguments
gContact.SetFrictionPairings(...) #set friction pairings and adjust searchTree if needed.
```

function/structure name	description

[1]Note that GeneralContact is still developed, use with care.

GetPythonObject()	convert member variables of GeneralContact into dictionary; use this for debug only!
Reset(freeMemory = True)	remove all contact objects and reset contact parameters
isActive	default = True (compute contact); if isActive=False, no contact computation is performed for this contact set
verboseMode	default = 0; verboseMode = 1 or higher outputs useful information on the contact creation and computation
visualization	access visualization data structure
resetSearchTreeInterval	(default=10000) number of search tree updates (contact computation steps) after which the search tree cells are re-created; this costs some time, will free memory in cells that are not needed any more
sphereSphereContact	activate/deactivate contact between spheres
sphereSphereFrictionRecycle	False: compute static friction force based on tangential velocity; True: recycle friction from previous PostNewton step, which greatly improves convergence, but may lead to unphysical artifacts; will be solved in future by step reduction
minRelDistanceSpheresTriangles	(default=1e-10) tolerance (relative to sphere radiues) below which the contact between triangles and spheres is ignored; used for spheres directly attached to triangles
frictionProportionalZone	(default=0.001) velocity $v\_\{\mu, reg\}$ upon which the dry friction coefficient is interpolated linearly (regularized friction model); must be greater 0; very small values cause oscillations in friction force
excludeOverlappingTrigSphereContacts	(default=True) for consistent, closed meshes, we can exclude overlapping contact triangles (which would cause holes if mesh is overlapping and not consistent!!!)
excludeDuplicatedTrigSphereContactPoints	(default=False) run additional checks for double contacts at edges or vertices, being more accurate but can cause additional costs if many contacts
computeExactStaticTriangleBins	(default=True) if True, search tree bins are computed exactly for static triangles while if False, it uses the overall (=very inaccurate) AABB of each triangle in the search tree
computeContactForces	(default=False) if True, additional system vector is computed which contains all contact force and torque contributions. In order to recover forces on a single rigid body, the respective LTG-vector has to be used and forces need to be extracted from this system vector; may slow down computations.
ancfCableUseExactMethod	(default=True) if True, uses exact computation of intersection of 3rd order polynomials and contacting circles
ancfCableNumberOfContactSegments	(default=1) number of segments to be used in case that ancfCableUseExactMethod=False; maximum number of segments=3
ancfCableMeasuringSegments	(default=20) number of segments used to approximate geometry for ANCFCable2D elements for measuring with ShortestDistanceAlongLine; with 20 segments the relative error due to approximation as compared to 10 segments usually stays below 1e-8

parallelTaskSplit	(default=12) general number of tasks per thread (min)
parallelTaskSplitBoundingBoxes	(default=48) number of tasks per thread for bounding box computations
parallelTaskSplitThreshold	(default=12) general threshold below which only one task per thread is used
parallelTaskSplitBoundingBoxesThreshold	(default=400) threshold below which only one task per thread is used, for bounding box computations

SetFrictionPairings(frictionPairings)	set Coulomb friction coefficients for pairings of materials (e.g., use material 0,1, then the entries (0,1) and (1,0) define the friction coefficients for this pairing); matrix should be symmetric!
	EXAMPLE:
	`#set 3 surface friction types, all being 0.1:`
	`gContact.SetFrictionPairings(0.1*np.ones((3,3)));`

SetFrictionProportionalZone(frictionProportionalZone)	regularization for friction (m/s); used for all contacts
SetSearchTreeCellSize(numberOfCells)	set number of cells of search tree (boxed search) in x, y and z direction **EXAMPLE**: `gContact.SetSearchTreeInitSize([10,10,10])`

SetSearchTreeBox(pMin, pMax)	set geometric dimensions of searchTreeBox (point with minimum coordinates and point with maximum coordinates); if this box becomes smaller than the effective contact objects, contact computations may slow down significantly
	EXAMPLE:
	`gContact.SetSearchTreeBox(pMin=[-1,-1,-1],`
	`pMax=[1,1,1])`

AddSphereWithMarker(markerIndex, radius, contactStiffness, contactDamping, frictionMaterialIndex)	add contact object using a marker (Position or Rigid), radius and contact/friction parameters and return localIndex of the contact item in GeneralContact; frictionMaterialIndex refers to frictionPairings in GeneralContact; contact is possible between spheres (circles in 2D) (if intraSphereContact = True), spheres and triangles and between sphere (=circle) and ANCFCable2D; contactStiffness is computed as serial spring between contacting objects, while damping is computed as a parallel damper
AddANCFCable(objectIndex, halfHeight, contactStiffness, contactDamping, frictionMaterialIndex)	add contact object for an ANCF cable element, using the objectIndex of the cable element and the cable's half height as an additional distance to contacting objects (currently not causing additional torque in case of friction), and return localIndex of the contact item in GeneralContact; currently only contact with spheres (circles in 2D) possible; contact computed using exact geometry of elements, finding max 3 intersecting contact regions

AddTrianglesRigidBodyBased(rigidBodyMarkerIndex, contactStiffness, contactDamping, frictionMaterialIndex, pointList, triangleList, staticTriangles = False)	add contact object using a rigidBodyMarker (of a body), contact/friction parameters, a list of points (as 3D numpy arrays or lists; coordinates relative to rigidBodyMarker) and a list of triangles (3 indices as numpy array or list) according to a mesh attached to the rigidBodyMarker; the flag staticTriangles=True can be used to inform the contact solver that these triangles are static (fixed in space); note that static triangles have to be added before dynamic triangles; function returns starting local index of trigsRigidBodyBased at which the triangles are stored; mesh can be produced with GraphicsData2TrigsAndPoints(...); contact is possible between sphere (circle) and Triangle but yet not between triangle and triangle; frictionMaterialIndex refers to frictionPairings in GeneralContact; contactStiffness is computed as serial spring between contacting objects, while damping is computed as a parallel damper (otherwise the smaller damper would always dominate); the triangle normal must point outwards, with the normal of a triangle given with local points (p0,p1,p2) defined as n=(p1-p0) x (p2-p0), see function ComputeTriangleNormal(...)
GetItemsInBox(pMin, pMax)	Get all items in box defined by minimum coordinates given in pMin and maximum coordinates given by pMax, accepting 3D lists or numpy arrays; in case that no objects are found, False is returned; otherwise, a dictionary is returned, containing numpy arrays with indices of obtained MarkerBasedSpheres, TrigsRigidBodyBased, ANCFCable2D, ...; the indices refer to the local index in GeneralContact which can be evaluated e.g. by GetMarkerBasedSphere(localIndex) **EXAMPLE**: `gContact.GetItemsInBox(pMin=[0,1,1], pMax=[2,3,2])`
GetSphereMarkerBased(localIndex, addData = False)	Get dictionary with current position, orientation, velocity, angular velocity as computed in last contact iteration; if addData=True, adds stored data of contact element, such as radius, markerIndex and contact parameters; localIndex is the internal index of contact element, as returned e.g. from GetItemsInBox
SetSphereMarkerBased(localIndex, contactStiffness = -1., contactDamping = -1., radius = -1., frictionMaterialIndex = -1)	Set data of marker based sphere with localIndex (as internally stored) with given arguments; arguments that are < 0 (default) imply that current values are not overwritten
GetTriangleRigidBodyBased(localIndex)	Get dictionary with rigid body index, local position of triangle vertices (nodes) and triangle normal; NOTE: the mesh added to contact is different from this structure, as it contains nodes and connectivity lists; the triangle index corresponds to the order as triangles are added to GeneralContact

SetTriangleRigidBodyBased(localIndex, points, contactRigidBodyIndex = -1)	Set data of marker based sphere with localIndex (triangle index); points are provided as 3x3 numpy array, with point coordinates in rows; contactRigidBodyIndex<0 indicates no change of the current index (and changing this index should be handled with care)
ShortestDistanceAlongLine(pStart = [0,0,0], direction = [1,0,0], minDistance = -1e-7, maxDistance = 1e7, asDictionary = False, cylinderRadius = 0, typeIndex = Contact.IndexEndOfEnumList)	Find shortest distance to contact objects in GeneralContact along line with pStart (given as 3D list or numpy array) and direction (as 3D list or numpy array with no need to be normalized); the function returns the distance which is >= minDistance and < maxDistance; in case of beam elements, it measures the distance to the beam centerline; the distance is measured from pStart along given direction and can also be negative; if no item is found along line, the maxDistance is returned; if asDictionary=False, the result is a float, while otherwise details are returned as dictionary (including distance, velocityAlongLine (which is the object velocity in given direction and may be different from the time derivative of the distance; works similar to a laser Doppler vibrometer - LDV), itemIndex and itemType in GeneralContact); the cylinderRadius, if not equal to 0, will be used for spheres to find closest sphere along cylinder with given point and direction; the typeIndex can be set to a specific contact type, e.g., which are searched for (otherwise all objects are considered)
UpdateContacts(mainSystem)	Update contact sets, e.g. if no contact is simulated (isActive=False) but user functions need up-to-date contact states for GetItemsInBox(...) or for GetActiveContacts(...) **EXAMPLE**: `gContact.UpdateContacts(mbs)`

GetActiveContacts(typeIndex, itemIndex)	Get list of global item numbers which are in contact with itemIndex of type typeIndex in case that the global itemIndex is smaller than the abs value of the contact pair index; a negative sign indicates that the contacting (spheres) is in Coloumb friction, a positive sign indicates a regularized friction region; in case of itemIndex==-1, it will return the list of numbers of active contacts per item for the contact type; for interpretation of global contact indices, see gContact.GetPythonObject() and documentation; requires either implicit contact computation or UpdateContacts(...) needs to be called prior to this function
	EXAMPLE:
	`#if explicit solver is used, we first need to update contacts:`
	`gContact.UpdateContacts(mbs)`
	`#obtain active contacts of marker based sphere 42:`
	`gList = gContact.GetActiveContacts(exu.ContactTypeIndex.IndexS 42)`

GetSystemODE2RhsContactForces(copy = False)	Get numpy array of system vector containing contribution of contact forces to system ODE2 Rhs vector; if copy=False, it will give direct (reference) access to the internal vector (note: modifications to this vector do not influence simulation!), however, which may cause problems if the system size changes or simulation is restarted; if copy=True, the vector is copied (time consuming); contributions to single objects may be extracted by checking the according LTG-array of according objects (such as rigid bodies); the contact forces vector is computed in each contact iteration;
__repr__()	return the string representation of the GeneralContact, containing basic information and statistics

6.8.1 VisuGeneralContact

This structure may contains some visualization parameters in future. Currently, all visualization settings are controlled via SC.visualizationSettings

function/structure name	**description**
Reset()	reset visualization parameters to default values

6.9 Data structures

This section describes a set of special data structures which are used in the Python-C++ interface, such as a MatrixContainer for dense/sparse matrices or a list of 3D vectors. Note that there are many native data types, such as lists, dicts and numpy arrays (e.g. 3D vectors), which are not described here as they are native to Pybind11, but can be passed as arguments when appropriate.

6.9.1 MatrixContainer

The MatrixContainer is a versatile representation for dense and sparse matrices. NOTE: if the MatrixContainer is constructed from a numpy array or a list of lists, both representing a dense matrix, it will go into dense mode; if it is initialized with a scipy sparse csr matrix, it will go into sparse mode. Examples:

```
#Create empty MatrixContainer:
from scipy.sparse import csr_matrix
from exudyn import MatrixContainer
mc = MatrixContainer() #empty matrix, dense mode

#Create MatrixContainer with dense matrix:
```

```
#container can be initialized with a dense matrix, using list of lists or a numpy array, e.g
    .:
matrix = np.eye(3)
#stores matrices internally in dense mode:
mcDense1 = MatrixContainer(matrix)
mcDense2 = MatrixContainer([[1,2],[3,4]])

#container can be initialized with a scipy csr sparse matrix, then being stored as sparse
    matrix
mcSparse = MatrixContainer(csr_matrix(matrix))

#Set with dense pyArray (a numpy array):
pyArray = np.array(matrix)
mc.SetWithDenseMatrix(pyArray, useDenseMatrix = True)

#Set empty matrix:
mc.SetWithDenseMatrix([[]], useDenseMatrix = True)

#Set with list of lists, stored as sparse matrix:
mc.SetWithDenseMatrix([[1,2],[3,4]], useDenseMatrix = False)

#Set with sparse triplets (list of lists or numpy array):
mc.SetWithSparseMatrix([[0,0,13.3],[1,1,4.2],[1,2,42.]],
                       numberOfRows=2, numberOfColumns=3,
                       useDenseMatrix=True)

print(mc)
#gives dense matrix:
#[[13.3  0.   0. ]
# [ 0.   4.2 42. ]]

#Set with scipy matrix:
#WARNING: only use csr_matrix
#         csc_matrix would basically run, but gives the transposed!!!
spmat = csr_matrix(matrix)
mc.SetWithSparseMatrix(spmat) #takes rows and column format automatically

#initialize and add triplets later on
mc.Initialize(3,3,useDenseMatrix=False)
mc.AddSparseMatrix(spmat, factor=1)
#can also add smaller matrix
mc.AddSparseMatrix(csr_matrix(np.eye(2)), factor=0.5)
print('mc8=',mc)
```

function/structure name	description

Initialize(numberOfRows, numberOfColumns, useDenseMatrix = True)	initialize MatrixContainer with number of rows and columns and set dense/sparse mode
SetWithDenseMatrix(pyArray, useDenseMatrix = False, factor = 1.)	set MatrixContainer with dense numpy array of size (n x m); array (=matrix) contains values and matrix size information; if useDenseMatrix=True, matrix will be stored internally as dense matrix, otherwise it will be converted and stored as sparse matrix (which may speed up computations for larger problems); pyArray is multiplied with given factor
SetWithSparseMatrix(sparseMatrix, numberOfRows = invalid (-1), numberOfColumns = invalid (-1), useDenseMatrix = False, factor = 1.)	set with scipy sparse csr_matrix (NOT: csc_matrix!) or with internal sparse triplet format (denoted as CSR): 'sparseMatrix' either contains a scipy matrix create with csr_matrix or a list of lists of sparse triplets (row, col, value) or the list of lists converted into numpy array; numberOfRowsInit and numberOfColumnsInit denote the size of the matrices, which are ignored in case of a scipy sparse matrix; if useDenseMatrix=True, matrix will be converted and stored internally as dense matrix, otherwise it will be stored as sparse matrix triplets; the values of sparseMatrix are multiplied with the given factor before storing
AddSparseMatrix(sparseMatrix, factor = 1.)	add scipy sparse csr_matrix with factor to already initilized MatrixContainer; sparseMatrix must contain according scipy csr format, otherwise the behavior is undefined! This function allows to efficiently add submatrices to the MatrixContainer
GetPythonObject()	convert MatrixContainer to numpy array (dense) or dictionary (sparse): containing nr. of rows, nr. of columns, numpy matrix with sparse triplets
Convert2DenseMatrix()	convert MatrixContainer to dense numpy array (SLOW and may fail for too large sparse matrices)
UseDenseMatrix()	returns True if dense matrix is used, otherwise False
SetAllZero()	Set all values to zero; dense mode: set all matrix entries to zero (slow); sparse mode: set number of triplets to zero (fast)
SetWithSparseMatrixCSR(numberOfRowsInit, numberOfColumnsInit, pyArrayCSR, useDenseMatrix = False, factor = 1.)	DEPRECATED: set with sparse CSR matrix format: numpy array 'pyArrayCSR' contains sparse triplet (row, col, value) per row; numberOfRows and numberOfColumns given extra; if useDenseMatrix=True, matrix will be converted and stored internally as dense matrix, otherwise it will be stored as sparse matrix; the values of pyArrayCSR are multiplied by the given factor
__repr__()	return the string representation of the MatrixContainer

6.9.2 GraphicsMaterialList

The GraphicsMaterialList contains the list of materials (material properties) for visualization; currently, only the raytracer uses materials. Materials can be accessed via the variable materials in renderer of SystemContainer.

```
#access material 0:
mat0 = SC.renderer.materials[0]
#convert into dictionary for easier processing:
matDict = mat0.GetDictionary()
matDict['alpha'] = 0.5
#change material (e.g. using a data base):
mat0.SetDictionary(matDict)
#or directly update material
mat0.name = 'new name'
mat0.emission = [0.8,0.6,0.]

#update material in renderer:
SC.renderer.materials.Set(0,mat0)
#update material directly with dictionary:
SC.renderer.materials.Set(0,matDict)

#create new material:
mat10 = SC.renderer.materials.New()
mat10.reflectivity = 0.8
SC.renderer.materials.Append(mat10) #returns index of mat10

#10 default graphics materials in Exudyn
#listed here with default color and some properties:
#note the increased computational costs for reflection & transparency
#the material names are as follows:
SC.renderer.materials[0].name == "default"    #steel blue
SC.renderer.materials[1].name == "matt"       #green
SC.renderer.materials[2].name == "steel"      #grey (reflection)
SC.renderer.materials[3].name == "plastic"    #red (reflection)
SC.renderer.materials[4].name == "chrome"     #light grey (reflection)
SC.renderer.materials[5].name == "shiny"      #orange (reflection)
SC.renderer.materials[6].name == "transparent"#(transparency,slight refraction)
SC.renderer.materials[7].name == "glass"      #light grey (reflection,transparency,
    refraction)
SC.renderer.materials[8].name == "mirror"     #light grey (reflection)
SC.renderer.materials[9].name == "emission"   #light yellow
```

function/structure name	**description**
Reset()	reset materials to 10 default materials
Append(material)	add single material as dict or VSettingsMaterial to list; returns index of newly added material
New()	Get new default material, which can be modified or appended to materials list
Set(indexOrName, material)	set material with index 'materialIndex' as dict or VSettingsMaterial

Get(indexOrName)	get material with index 'materialIndex' as VSettingsMaterial
GetDict(indexOrName)	get material with index 'materialIndex' as dict
len(data)	return length of the Vector3DList, using len(data) where data is the Vector3DList
... = data[index]	get reference access of material with 'index' as VSettingsMaterial
__repr__()	return the string representation of the GraphicsMaterialList

6.9.3 Vector3DList

The Vector3DList is used to represent lists of 3D vectors. This is used to transfer such lists from Python to C++.

Usage:

- Create empty `Vector3DList` with `x = Vector3DList()`
- Create `Vector3DList` with list of numpy arrays:`x = Vector3DList([ numpy.array([1.,2.,3.]), numpy.array([4.,5.,6.]) ])`
- Create `Vector3DList` with list of lists `x = Vector3DList([[1.,2.,3.], [4.,5.,6.]])`
- Append item: `x.Append([0.,2.,4.])`
- Convert into list of numpy arrays: `x.GetPythonObject()`

function/structure name	**description**
Append(pyArray)	add single array or list to Vector3DList; array or list must have appropriate dimension!
GetPythonObject()	convert Vector3DList into (copied) list of numpy arrays
len(data)	return length of the Vector3DList, using len(data) where data is the Vector3DList
data[index]= ...	set list item 'index' with data, write: data[index] = ...
... = data[index]	get copy of list item with 'index' as vector
__copy__()	copy method to be used for copy.copy(...); in fact does already deep copy
__deepcopy__()	deepcopy method to be used for copy.copy(...)
__repr__()	return the string representation of the Vector3DList data, e.g.: print(data)

6.9.4 Vector2DList

The Vector2DList is used to represent lists of 2D vectors. This is used to transfer such lists from Python to C++.

Usage:

- Create empty `Vector2DList` with `x = Vector2DList()`
- Create `Vector2DList` with list of numpy arrays:
  `x = Vector2DList([ numpy.array([1.,2.]), numpy.array([4.,5.])  ])`
- Create `Vector2DList` with list of lists `x = Vector2DList([[1.,2.], [4.,5.]])`
- Append item: `x.Append([0.,2.])`
- Convert into list of numpy arrays: `x.GetPythonObject()`
- similar to Vector3DList !

function/structure name	**description**
Append(pyArray)	add single array or list to Vector2DList; array or list must have appropriate dimension!
GetPythonObject()	convert Vector2DList into (copied) list of numpy arrays
len(data)	return length of the Vector2DList, using len(data) where data is the Vector2DList
data[index]= ...	set list item 'index' with data, write: data[index] = ...
... = data[index]	get copy of list item with 'index' as vector
__copy__()	copy method to be used for copy.copy(...); in fact does already deep copy
__deepcopy__()	deepcopy method to be used for copy.copy(...)
__repr__()	return the string representation of the Vector2DList data, e.g.: print(data)

6.9.5 Vector6DList

The Vector6DList is used to represent lists of 6D vectors. This is used to transfer such lists from Python to C++.

Usage:

- Create empty `Vector6DList` with `x = Vector6DList()`
- Convert into list of numpy arrays: `x.GetPythonObject()`
- similar to Vector3DList !

function/structure name	**description**
Append(pyArray)	add single array or list to Vector6DList; array or list must have appropriate dimension!
GetPythonObject()	convert Vector6DList into (copied) list of numpy arrays
len(data)	return length of the Vector6DList, using len(data) where data is the Vector6DList
data[index]= ...	set list item 'index' with data, write: data[index] = ...

... = data[index]	get copy of list item with 'index' as vector
__copy__()	copy method to be used for copy.copy(...); in fact does already deep copy
__deepcopy__()	deepcopy method to be used for copy.copy(...)
__repr__()	return the string representation of the Vector6DList data, e.g.: print(data)

6.9.6 Matrix3DList

The Matrix3DList is used to represent lists of 3D Matrices. . This is used to transfer such lists from Python to C++.

Usage:

- Create empty `Matrix3DList` with `x = Matrix3DList()`
- Create `Matrix3DList` with list of numpy arrays:
  `x = Matrix3DList([ numpy.eye(3), numpy.array([[1.,2.,3.],[4.,5.,6.],[7.,8.,9.]])  ])`
- Append item: `x.Append(numpy.eye(3))`
- Convert into list of numpy arrays: `x.GetPythonObject()`
- similar to Vector3DList !

function/structure name	**description**
Append(pyArray)	add single 3D array or list of lists to Matrix3DList; array or lists must have appropriate dimension!
GetPythonObject()	convert Matrix3DList into (copied) list of 3x3 numpy arrays
len(data)	return length of the Matrix3DList, using len(data) where data is the Matrix3DList
data[index]= ...	set list item 'index' with matrix, write: data[index] = ...
... = data[index]	get copy of list item with 'index' as matrix
__repr__()	return the string representation of the Matrix3DList data, e.g.: print(data)

6.9.7 Matrix6DList

The Matrix6DList is used to represent lists of 6D Matrices. . This is used to transfer such lists from Python to C++.

Usage:

- Create empty `Matrix6DList` with `x = Matrix6DList()`

- Create `Matrix6DList` with list of numpy arrays:
  `x = Matrix6DList([ numpy.eye(6), 2*numpy.eye(6) ])`
- Append item: `x.Append(numpy.eye(6))`
- Convert into list of numpy arrays: `x.GetPythonObject()`
- similar to Matrix3DList !

function/structure name	**description**
Append(pyArray)	add single 6D array or list of lists to Matrix6DList; array or lists must have appropriate dimension!
GetPythonObject()	convert Matrix6DList into (copied) list of 6x6 numpy arrays
len(data)	return length of the Matrix6DList, using len(data) where data is the Matrix6DList
data[index]= ...	set list item 'index' with matrix, write: data[index] = ...
... = data[index]	get copy of list item with 'index' as matrix
__repr__()	return the string representation of the Matrix6DList data, e.g.: print(data)

6.10 Type definitions

This section defines a couple of structures (C++: enum aka enumeration type), which are used to select, e.g., a configuration type or a variable type. In the background, these types are integer numbers, but for safety, the types should be used as type variables. See this examples:

```
#Conversion to integer is possible:
x = int(exu.OutputVariableType.Displacement)
#also conversion from integer:
varType = exu.OutputVariableType(8)
#use in settings:
SC.visualizationSettings.contour.outputVariable = exu.OutputVariableType.StressLocal
#use outputVariableType in sensor:
mbs.AddSensor(SensorBody(bodyNumber=rigid, storeInternal=True,
                         outputVariableType=exu.OutputVariableType.Displacement))
#
```

6.10.1 OutputVariableType

This section shows the OutputVariableType structure, which is used for selecting output values, e.g. for GetObjectOutput(...) or for selecting variables for contour plot.

Available output variables and the interpreation of the output variable can be found at the object definitions. The OutputVariableType does not provide information about the size of the output variable, which can be either scalar or a list (vector). For vector output quantities, the contour plot

option offers an additional parameter for selection of the component of the OutputVariableType. The components are usually out of {0,1,2}, representing {x,y,z} components (e.g., of displacements, velocities, ...), or {0,1,2,3,4,5} representing {xx,yy,zz,yz,xz,xy} components (e.g., of strain or stress). In order to compute a norm, chose component=-1, which will result in the quadratic norm for other vectors and to a norm specified for stresses (if no norm is defined for an outputVariable, it does not compute anything)

function/structure name	**description**
_None	no value; used, e.g., to select no output variable in contour plot
Distance	e.g., measure distance in spring damper connector
Position	measure 3D position, e.g., of node or body
Displacement	measure displacement; usually difference between current position and reference position
DisplacementLocal	measure local displacement, e.g. in local joint coordinates
Velocity	measure (translational) velocity of node or object
VelocityLocal	measure local (translational) velocity, e.g. in local body or joint coordinates
Acceleration	measure (translational) acceleration of node or object
AccelerationLocal	measure (translational) acceleration of node or object in local coordinates
RotationMatrix	measure rotation matrix of rigid body node or object
Rotation	measure, e.g., scalar rotation of 2D body, Euler angles of a 3D object or rotation within a joint
AngularVelocity	measure angular velocity of node or object
AngularVelocityLocal	measure local (body-fixed) angular velocity of node or object
AngularAcceleration	measure angular acceleration of node or object
AngularAccelerationLocal	measure angular acceleration of node or object in local coordinates
CoordinatesTotal	measure the total coordinates (including reference configuration) of a node or object; otherwise the same as Coordinates
Coordinates	measure the coordinates of a node or object; coordinates just contain displacements, but not the reference (position or rotation) values - see also definition of respective nodes or objects
Coordinates_t	measure the time derivative of coordinates (= velocity coordinates) of a node or object
Coordinates_tt	measure the second time derivative of coordinates (= acceleration coordinates) of a node or object
SlidingCoordinate	measure sliding coordinate in sliding joint
Director1	measure a director (e.g. of a rigid body frame), or a slope vector in local 1 or x-direction
Director2	measure a director (e.g. of a rigid body frame), or a slope vector in local 2 or y-direction
Director3	measure a director (e.g. of a rigid body frame), or a slope vector in local 3 or z-direction

Force	measure global force, e.g., in joint or beam (resultant force), or generalized forces; see description of according object
ForceLocal	measure local force, e.g., in joint or beam (resultant force)
Torque	measure torque, e.g., in joint or beam (resultant couple/-moment)
TorqueLocal	measure local torque, e.g., in joint or beam (resultant couple/moment)
StrainLocal	measure local strain, e.g., axial strain in cross section frame of beam or Green-Lagrange strain
StressLocal	measure local stress, e.g., axial stress in cross section frame of beam or Second Piola-Kirchoff stress; choosing component==-1 will result in the computation of the Mises stress
CurvatureLocal	measure local curvature; may be scalar or vectorial: twist and curvature of beam in cross section frame
ConstraintEquation	evaluates constraint equation (=current deviation or drift of constraint equation)

6.10.2 ConfigurationType

This section shows the ConfigurationType structure, which is used for selecting a configuration for reading or writing information to the module. Specifically, the ConfigurationType.Current configuration is usually used at the end of a solution process, to obtain result values, or the ConfigurationType.Initial is used to set initial values for a solution process.

function/structure name	**description**
_None	no configuration; usually not valid, but may be used, e.g., if no configurationType is required
Initial	initial configuration prior to static or dynamic solver; is computed during mbs.Assemble() or AssembleInitializeSystemCoordinates()
Current	current configuration during and at the end of the computation of a step (static or dynamic)
Reference	configuration used to define deformable bodies (reference configuration for finite elements) or joints (configuration for which some joints are defined)
StartOfStep	during computation, this refers to the solution at the start of the step = end of last step, to which the solver falls back if convergence fails
Visualization	this is a state completely de-coupled from computation, used for visualization
EndOfEnumList	this marks the end of the list, usually not important to the user

6.10.3 ItemType

This section shows the ItemType structure, which is used for defining types of indices, e.g., in render window and will be also used in item dictionaries in future.

function/structure name	description
_None	item has no type
Node	item or index is of type Node
Object	item or index is of type Object
Marker	item or index is of type Marker
Load	item or index is of type Load
Sensor	item or index is of type Sensor

6.10.4 NodeType

This section shows the NodeType structure, which is used for defining node types for 3D rigid bodies.

function/structure name	description
_None	node has no type
Ground	ground node
Position2D	2D position node
Orientation2D	node with 2D rotation
Point2DSlope1	2D node with 1 slope vector
Position	3D position node
Orientation	3D orientation node
RigidBody	node that can be used for rigid bodies
RotationEulerParameters	node with 3D orientations that are modelled with Euler parameters (unit quaternions)
RotationRxyz	node with 3D orientations that are modelled with Tait-Bryan angles
RotationRotationVector	node with 3D orientations that are modelled with the rotation vector
LieGroupWithDirectUpdate	node to be solved with Lie group methods, without data coordinates
GenericODE2	node with general ODE2 variables
GenericODE1	node with general ODE1 variables
GenericAE	node with general algebraic variables
GenericData	node with general data variables
PointSlope1	node with 1 slope vector
PointSlope12	node with 2 slope vectors in x and y direction
PointSlope23	node with 2 slope vectors in y and z direction

6.10.5 JointType

This section shows the JointType structure, which is used for defining joint types, used in KinematicTree.

function/structure name	description
_None	node has no type
RevoluteX	revolute joint type with rotation around local X axis
RevoluteY	revolute joint type with rotation around local Y axis
RevoluteZ	revolute joint type with rotation around local Z axis
PrismaticX	prismatic joint type with translation along local X axis
PrismaticY	prismatic joint type with translation along local Y axis
PrismaticZ	prismatic joint type with translation along local Z axis

6.10.6 DynamicSolverType

This section shows the DynamicSolverType structure, which is used for selecting dynamic solvers for simulation.

function/structure name	description
GeneralizedAlpha	an implicit solver for index 3 problems; intended to be used for solving directly the index 3 constraints using the spectralRadius sufficiently small (usually 0.5 .. 1)
TrapezoidalIndex2	an implicit solver for index 3 problems with index2 reduction; uses generalized alpha solver with settings for Newmark with index2 reduction
ExplicitEuler	an explicit 1st order solver (generally not compatible with constraints)
ExplicitMidpoint	an explicit 2nd order solver (generally not compatible with constraints)
RK33	an explicit 3 stage 3rd order Runge-Kutta method, aka "Heun third order"; (generally not compatible with constraints)
RK44	an explicit 4 stage 4th order Runge-Kutta method, aka "classical Runge Kutta" (generally not compatible with constraints), compatible with Lie group integration and elimination of CoordinateConstraints
RK67	an explicit 7 stage 6th order Runge-Kutta method, see 'On Runge-Kutta Processes of High Order', J. C. Butcher, J. Austr Math Soc 4, (1964); can be used for very accurate (reference) solutions, but without step size control!
ODE23	an explicit Runge Kutta method with automatic step size selection with 3rd order of accuracy and 2nd order error estimation, see Bogacki and Shampine, 1989; also known as ODE23 in MATLAB

DOPRI5	an explicit Runge Kutta method with automatic step size selection with 5th order of accuracy and 4th order error estimation, see Dormand and Prince, 'A Family of Embedded Runge-Kutta Formulae.', J. Comp. Appl. Math. 6, 1980
DVERK6	[NOT IMPLEMENTED YET] an explicit Runge Kutta solver of 6th order with 5th order error estimation; includes adaptive step selection
VelocityVerlet	[TEST phase] a special explicit time integration scheme, the 'velocity Verlet' method (similar to leap frog method), with second order accuracy for conservative second order differential equations, often used for particle dynamics and contact; implementation uses Explicit Euler for ODE1 equations

6.10.7 CrossSectionType

This section shows the CrossSectionType structure, which is used for defining beam cross section types.

function/structure name	**description**
Polygon	cross section profile defined by polygon
Circular	cross section is circle or elliptic

6.10.8 KeyCode

This section shows the KeyCode structure, which is used for special key codes in keyPressUserFunction.

function/structure name	**description**
SPACE	space key
ENTER	enter (return) key
TAB	
BACKSPACE	
RIGHT	cursor right
LEFT	cursor left
DOWN	cursor down
UP	cursor up
F1	function key F1
F2	function key F2
F3	function key F3
F4	function key F4
F5	function key F5

F6	function key F6
F7	function key F7
F8	function key F8
F9	function key F9
F10	function key F10

6.10.9 LinearSolverType

This section shows the LinearSolverType structure, which is used for selecting linear solver types, which are dense or sparse solvers.

function/structure name	**description**
_None	no value; used, e.g., if no solver is selected
EXUdense	use dense matrices and according solvers for densly populated matrices (usually the CPU time grows cubically with the number of unknowns)
EigenSparse	use sparse matrices and according solvers; additional overhead for very small multibody systems; specifically, memory allocation is performed during a factorization process
EigenSparseSymmetric	use sparse matrices and according solvers; NOTE: this is the symmetric mode, which assumes symmetric system matrices; this is EXPERIMENTAL and should only be used of user knows that the system matrices are (nearly) symmetric; does not work with scaled GeneralizedAlpha matrices; does not work with constraints, as it must be symmetric positive definite
EigenDense	use Eigen's LU factorization with partial pivoting (faster than EXUdense) or full pivot (if linearSolverSettings.ignoreSingularJacobian=True; is much slower)

6.10.10 ContactTypeIndex

This section shows the ContactTypeIndex structure, which is in GeneralContact to select specific contact items, such as spheres, ANCFCable or triangle items.

function/structure name	**description**
IndexSpheresMarkerBased	spheres attached to markers
IndexANCFCable2D	ANCFCable2D contact items
IndexTrigsRigidBodyBased	triangles attached to rigid body (or rigid body marker)
IndexEndOfEnumList	signals end of list

Chapter 7

Python utility functions

This chapter describes in every subsection the functions and classes of the utility modules. These modules help to create multibody systems with the EXUDYN core module. Functions are implemented in Python and can be easily changed, extended and also verified by the user. **Check the source code** by entering these functions in Sypder and pressing **CTRL + left mouse button**. These Python functions are much slower than the functions available in the C++ core. Some matrix computations with larger matrices implemented in numpy and scipy, however, are parallelized and therefore very efficient.

Note that in general functions accept lists and numpy arrays. If not, an error will occur, which is easily tracked. Furthermore, angles are generally provided in radian (2π equals 360^o) and no units are used for distances, but it is recommended to use SI units (m, kg, s) throughout.

Functions have been implemented, if not otherwise mentioned, by Johannes Gerstmayr.

7.1 Utility: ResultsMonitor

The results monitor is a special tool, which allows to monitor results during simulation. This is intended, e.g., to show results for long-term simulations or to visualize results for teaching. The tool can visualize time dependent data (e.g., from sensors or solution files) or data from optimization. The tool automatically detects the type of file and visualizes all given columns (default) or selected columns of the file.

For running the results monitor, start a terminal (linux) or an Anaconda prompt (windows). Either you copy the `resultsLoader.py`, located in the `main/pythonDev/exudyn` subfolder of the repository, to your desired/current directory or you call it from a relative path from your current directory. Usage is described by typing `python resultsMonitor.py -h`, as given in the following listing:

```
usage for resultsLoader:
  python resultsLoader.py file.txt
options:
  -xcols i,j,..: comma-separated columns (NO SPACES!) to be plotted on x-axis
  -ycols i,j,..: comma-separated columns (NO SPACES!) to be plotted on y-axis
  -logx: use log scale for x-axis
  -logy: use log scale for y-axis
  -sizex float: float = x-size of one subplot in inches (default=5)
  -sizey float: float = y-size of one subplot in inches (default=5)
```

```
  -update float: float = update period in seconds
  -color char: char = line color code according to pyplot, default=b (blue)
  -style char: char = line symbol according to pyplot, default="-"
example: (to be called from windows Anaconda prompt or in linux terminal in the directory
    where file.txt lies)
  python resultsMonitor.py file.txt -logy -xcols 0,1 -ycols 2,3 -update 0.2
```

7.2 Module: advancedUtilities

Advanced utility functions only depending on numpy or specified exudyn modules; Here, we gather special functions, which are depending on other modules and do not fit into exudyn.utilities as they cannot be imported e.g. in rigidBodyUtilities

Author: Johannes Gerstmayr

Date: 2023-01-06 (created)

def **PlotLineCode** (*index*)

- **function description**: helper functions for matplotlib, returns a list of 28 line codes to be used in plot, e.g. 'r-' for red solid line
- **input**: index in range(0:28)
- **output**: a color and line style code for matplotlib plot

For examples on PlotLineCode see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- serialRobotInteractiveLimits.py (Ex), serialRobotTSD.py (Ex)

---

def **FindObjectIndex** (*i*, *globalVariables*)

- **function description**: simple function to find object index i within the local or global scope of variables
- **input**: i, the integer object number and globalVariables=globals()
- **example**:

```
FindObjectIndex(2, locals() )  #usually sufficient
FindObjectIndex(2, globals() ) #wider search
```

---

def **FindNodeIndex** (*i*, *globalVariables*)

- **function description**: simple function to find node index i within the local or global scope of variables
- **input**: i, the integer node number and globalVariables=globals()
- **example**:

```
FindObjectIndex(2, locals() )  #usually sufficient
FindObjectIndex(2, globals() ) #wider search
```

---

def **IsListOrArray** (*data*, *checkIfNoneEmpty*= False)

- **function description**: checks, if data is of type list or np.array; used in functions to check input data
- **input**:

  data: any type, preferrably list or numpy.array

  checkIfNoneEmpty: if True, function only returns True if type is list or array AND if length is non-zero
- **output**: returns True/False

---

def **RaiseTypeError** (*where*= '', *argumentName*= '', *received*= None, *expectedType*= None, *dim*= None, *cols*= None)

- **function description**: internal function which is used to raise common errors in case of wrong types; dim is used for vectors and square matrices, cols is used for non-square matrices

---

def **IsNone** (*x*)

- **function description**: return True, if x is None; works also for numpy arrays or structures

---

def **IsNotNone** (x)

- **function description**: return True, if x is not None; works also for numpy arrays or structures

---

def **IsValidBool** (x)

- **function description**: return True, if x is int, float, np.double, np.integer or similar types that can be automatically casted to pybind11

---

def **IsValidRealInt** (x)

- **function description**: return True, if x is int, float, np.double, np.integer or similar types that can be automatically casted to pybind11

---

def **IsValidPRealInt** (x)

- **function description**: return True, if x is valid Real/Int and positive

---

def **IsValidURealInt** (x)

- **function description**: return True, if x is valid Real/Int and unsigned (non-negative)

---

def **IsReal** (x)

- **function description**: return True, if x is any python or numpy float type; could also be called IsFloat(), but Real has special meaning in Exudyn

---

def **IsInteger** (*x*)

- **function description**: return True, if x is any python or numpy float type

---

def **IsVector** (*v*, *expectedSize*= None)

- **function description**: check if v is a valid vector with floats or ints; if expectedSize!=None, the length is also checked

---

def **IsIntVector** (*v*, *expectedSize*= None)

- **function description**: check if v is a valid vector with floats or ints; if expectedSize!=None, the length is also checked

---

def **IsSquareMatrix** (*m*, *expectedSize*= None)

- **function description**: check if v is a valid vector with floats or ints; if expectedSize!=None, the length is also checked

---

def **IsValidObjectIndex** (*x*)

- **function description**: return True, if x is valid exudyn object index

---

def **IsValidNodeIndex** (*x*)

- **function description**: return True, if x is valid exudyn node index

---

def **IsValidMarkerIndex** (*x*)

- **function description**: return True, if x is valid exudyn marker index

---

def **IsEmptyList** (*x*)

- **function description**: return True, if x is an empty list (or empty list converted from numpy array), otherwise return False

---

def **FillInSubMatrix** (*subMatrix*, *destinationMatrix*, *destRow*, *destColumn*)

- **function description**: fill submatrix into given destinationMatrix; all matrices must be numpy arrays
- **input**:

  subMatrix: input matrix, which is filled into destinationMatrix

  destinationMatrix: the subMatrix is entered here

  destRow: row destination of subMatrix

  destColumn: column destination of subMatrix
- **output**: destinationMatrix is changed after function call
- **notes**: may be erased in future!

For examples on FillInSubMatrix see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `objectFFRFTest.py` (TM)

---

def **SweepSin** (*t*, *t1*, *f0*, *f1*)

- **function description**: compute sin sweep at given time t
- **input**:

  t: evaluate of sweep at time t

  t1: end time of sweep frequency range

f0: start of frequency interval [f0,f1] in Hz

f1: end of frequency interval [f0,f1] in Hz

- **output**: evaluation of sin sweep (in range -1..+1)

For examples on SweepSin see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- objectGenericODE2Test.py (TM)

---

def **SweepCos** (*t*, *t1*, *f0*, *f1*)

- **function description**: compute cos sweep at given time t
- **input**:

  t: evaluate of sweep at time t

  t1: end time of sweep frequency range

  f0: start of frequency interval [f0,f1] in Hz

  f1: end of frequency interval [f0,f1] in Hz
- **output**: evaluation of cos sweep (in range -1..+1)

For examples on SweepCos see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- rigidRotor3DbasicBehaviour.py (Ex), rigidRotor3Drunup.py (Ex), objectGenericODE2Test.py (TM)

---

def **FrequencySweep** (*t*, *t1*, *f0*, *f1*)

- **function description**: frequency according to given sweep functions SweepSin, SweepCos
- **input**:

  t: evaluate of frequency at time t

  t1: end time of sweep frequency range

  f0: start of frequency interval [f0,f1] in Hz

  f1: end of frequency interval [f0,f1] in Hz
- **output**: frequency in Hz

For examples on FrequencySweep see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- objectGenericODE2Test.py (TM)

---

def **SmoothStep** (*x*, *x0*, *x1*, *value0*, *value1*)

- **function description**: step function with smooth transition from value0 to value1; transition is computed with cos function
- **input**:

  x: argument at which function is evaluated

  x0: start of step (f(x) = value0)

  x1: end of step (f(x) = value1)

  value0: value before smooth step

  value1: value at end of smooth step
- **output**: returns f(x)

For examples on SmoothStep see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- beamTutorial.py (Ex), beltDriveALE.py (Ex), beltDriveReevingSystem.py (Ex), chainDriveExample.py (Ex), craneReevingSystem.py (Ex), ... , createKinematicTreeTest.py (TM)

---

def **SmoothStepDerivative** (*x*, *x0*, *x1*, *value0*, *value1*)

- **function description**: derivative of SmoothStep using same arguments
- **input**:

  x: argument at which function is evaluated

  x0: start of step (f(x) = value0)

  x1: end of step (f(x) = value1)

  value0: value before smooth step

  value1: value at end of smooth step
- **output**: returns d/dx(f(x))

For examples on SmoothStepDerivative see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- leggedRobot.py (Ex)

---

def **IndexFromValue** (*data*, *value*, *tolerance*= 1e-7, *assumeConstantSampleRate*= False, *rangeWarning*= True)

- **function description**: get index from value in given data vector (numpy array); usually used to get specific index of time vector; this function is slow (linear search), if sampling rate is non-constant; otherwise set assumeConstantSampleRate=True!
- **input**:

  data: containing (almost) equidistant values of time

  value: e.g., time to be found in data

  tolerance: tolerance, which is accepted (default: tolerance=1e-7)

  rangeWarning: warn, if index returns out of range; if warning is deactivated, function uses the closest value
- **output**: index
- **notes**: to obtain the interpolated value of a time-signal array, use GetInterpolatedSignalValue() in exudyn.signalProcessing

---

def **RoundMatrix** (*matrix*, *treshold*= 1e-14)

- **function description**: set all entries in matrix to zero which are smaller than given treshold; operates directly on matrix
- **input**: matrix as np.array, treshold as positive value
- **output**: changes matrix

---

def **ConvertScipySparseToDict** (*sparseMatrix*)

- **function description**: Function to convert a scipy sparse matrix to a dictionary

---

def **ConvertDictToScipySparse** (*sparseDict*)

- **function description**: Function to convert a dictionary back to a scipy sparse matrix

---

def **SaveDictToHDF5** (*fileName*, *dataDict*)

- **function description**: recursively saves a hierarchical dictionary dataDict to a HDF5 file with given fileName; limitations for certain types and Python or symbolic user functions
- **input**:

  fileName: file name (possibly including path) for HDF5 file, including file ending

  dataDict: the dictionary containing the hierarchical data to be saved; the data may contain the following data types in hierarchical form: int, bool, float, str (utf-8), list, dict, numpy array, scipy csr_matrix, Python function
- **output**: None

For examples on SaveDictToHDF5 see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `NGsolvePistonEngine.py` (Ex), `serialRobotURDF - Kopie.py` (Ex), `serialRobotURDF.py` (Ex), `testHDF5loadSave.py` (Ex), `pickleCopyMbs.py` (TM)

---

def **LoadDictFromHDF5** (*fileName*, *callerGlobals*= None)

- **function description**: recursively loads a hierarchical dictionary from a HDF5 file with given fileName
- **input**:

  fileName: file name (possibly including path) for HDF5 file, including file ending

  callerGlobals: optional: if your data contains functions, the callerGlobals must contain, e.g., globals() of the caller, where the Python functions are defined at which the HDF5 function refers to
- **output**: dict which contains loaded data

For examples on LoadDictFromHDF5 see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- serialRobotURDF - Kopie.py (Ex), serialRobotURDF.py (Ex), testHDF5loadSave.py (Ex), pickleCopyMbs.py (TM)

---

def **ConvertFunctionToSymbolic** (*mbs*, *function*, *userFunctionName*, *itemIndex*= None, *itemTypeName*= None, *verbose*= 0)

- **function description**:

  Internal function to convert a Python user function into a dictionary containing the symbolic representation;

  this function is under development and should be used with care

- **input**:

  mbs: MainSystem, needed currently for interface

  function: Python function with interface according to desired user function

  itemIndex: item index, such as ObjectIndex or LoadIndex; -1 indicates MainSystem; if None, itemTypeName must be provided instead

  itemTypeName: use of type name, such as ObjectConnectorSpringDamper; in this case, itemIndex must be None

  itemIndex: item index, such as ObjectIndex or LoadIndex; -1 indicates MainSystem

  userFunctionName: name of user function item, see documentation; this is required, because some items have several user functions, which need to be distinguished

  verbose: if > 0, according output is printed

- **output**: return dictionary with 'functionName', 'argList', and 'returnList'

---

def **CreateSymbolicUserFunction** (*mbs*, *function*, *userFunctionName*, *itemIndex*= None, *itemTypeName*= None, *verbose*= 0)

- **function description**:

  Helper function to convert a Python user function into a symbolic user function;

  this function is under development and should be used with care

- **input**:

  mbs: MainSystem, needed currently for interface

  function: Python function with interface according to desired user function

itemIndex: item index, such as ObjectIndex or LoadIndex; -1 indicates MainSystem; if None, itemTypeName must be provided instead

itemTypeName: use of type name, such as ObjectConnectorSpringDamper; in this case, itemIndex must be None

userFunctionName: name of user function item, see documentation; this is required, because some items have several user functions, which need to be distinguished

verbose: if > 0, according output may be printed

- **output**: returns symbolic user function; this can be transfered into an item using TransferUserFunction2Item

- **notes**: keep the return value alive in a variable (or list), as it contains the expression tree which must exist for the lifetime of the user function

- **example**:

```
oGround = mbs.AddObject(ObjectGround())
node = mbs.AddNode(NodePoint(referenceCoordinates = [1.05,0,0]))
oMassPoint = mbs.AddObject(MassPoint(nodeNumber = node, physicsMass=1))
symbolicFunc = CreateSymbolicUserFunction(mbs, function=springForceUserFunction,
                                          userFunctionName='springForceUserFunction
  ',
                                          itemTypeName='ObjectConnectorSpringDamper
  ')
m0 = mbs.AddMarker(MarkerBodyPosition(bodyNumber=oGround, localPosition=[0,0,0]))
m1 = mbs.AddMarker(MarkerBodyPosition(bodyNumber=oMassPoint, localPosition=[0,0,0])
  )
co = mbs.AddObject(ObjectConnectorSpringDamper(markerNumbers=[m0,m1],
                   referenceLength = 1, stiffness = 100, damping = 1,
                   springForceUserFunction=symbolicFunc))
exudyn.Print(symbolicFunc.Evaluate(mbs, 0., 0, 1.1, 0.,  100., 0., 13.) )
```

For examples on CreateSymbolicUserFunction see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- cartesianSpringDamperUserFunction.py (Ex), SpringDamperMassUserFunction.py (Ex), symbolicUserFunctionMasses.py (Ex),
  loadUserFunctionTest.py (TM), symbolicUserFunctionTest.py (TM)

7.2.1 CLASS ExpectedType(Enum) (in module advancedUtilities)

class description: internal type which is used for type checking in exudyn Python user functions; used to create unique error messages

7.3 Module: artificialIntelligence

This library collects interfaces and functionality for artificial intelligence This library is under construction (2022-05); To make use of this libraries, you need to install openAI gym with 'pip install gym'; For standard machine learning algorithms, install e.g. stable_baselines3 using 'pip install stable_baselines3'

Author: Johannes Gerstmayr

Date: 2022-05-21 (created)

7.3.1 CLASS OpenAIGymInterfaceEnv(Env) (in module artificialIntelligence)

class description: interface class to set up Exudyn model which can be used as model in open AI gym; see specific class functions which contain 'OVERRIDE' to integrate your model; in general, set up a model with CreateMBS(), map state to initial values, initial values to state and action to mbs;

def **__init__** (*self*, ***kwargs*)

- **classFunction**: internal function to initialize model; store self.mbs and self.simulationSettings; special arguments **kwargs are passed to CreateMBS

---

def **CreateMBS** (*self*, *SC*, *mbs*, *simulationSettings*, ***kwargs*)

- **classFunction**:

  OVERRIDE this function to create multibody system mbs and setup simulationSettings; call Assemble() at the end!

  you may also change SC.visualizationSettings() individually; kwargs may be used for special setup

---

def **SetupSpaces** (*self*)

- **classFunction**: OVERRIDE this function to set up self.action_space and self.observation_space

---

def **MapAction2MBS** (*self*, *action*)

- **classFunction**: OVERRIDE this function to map the action given by learning algorithm to the multibody system, e.g. as a load parameter

---

def **Output2StateAndDone** (*self*)

- **classFunction**: OVERRIDE this function to collect output of simulation and map to self.state tuple
- **output**: return bool done which contains information if system state is outside valid range

---

def **State2InitialValues** (*self*)

- **classFunction**: OVERRIDE this function to maps the current state to mbs initial values
- **output**: return [initialValues, initialValues_t] where initialValues[_t] are ODE2 vectors of coordinates[_t] for the mbs

---

def **TestModel** (*self*, *numberOfSteps*= 500, *seed*= 0, *model*= None, *solutionFileName*= None, *useRenderer*= True, *sleepTime*= 0.01, *stopIfDone*= False, *showTimeSpent*= True, ***kwargs*)

- **classFunction**: test model by running in simulation environment having several options
- **input**:

  numberOfSteps: number of steps to test MBS and model (with or without learned model); with renderer, press 'Q' in render window to stop simulation

  seed: seed value for reset function; this value initializes the randomizer; use e.g. time to obtain non-reproducible results

  model: either None to just test the MBS model without learned model, or containing a learned model, e.g., with A2C; use A2C.save(...) and A2C.load(...) for storing and retrieving models

  solutionFileName: if given, the MBS internal states are written to the file with given name, which can be loaded with solution viewer and visualized; solution is written every period given in simulationSettings.solutionSettings.solutionWritePeriod

useRenderer: if set True, the internal renderer is used and model updates are shown in visualization of Exudyn

return_info: internal value in reset function

sleepTime: sleep time between time steps to obtain certain frame rate for visualization

stopIfDone: if set to True, the simulation will reset as soon as the defined observation limits are reached and done is set True

showTimeSpent: if True, the total time spent is measured; this helps to check the performance of the model (e.g. how many steps can be computed per second)

---

def **SetSolver** (*self*, *solverType*)

- **classFunction**: use solverType = exudyn.DynamicSolverType.[...] to define solver (choose between implicit and explicit solvers!)

---

def **PreInitializeSolver** (*self*)

- **classFunction**: internal function which initializes dynamic solver; adapt in special cases; this function has some overhead and should not be called during reset() or step()

---

def **IntegrateStep** (*self*)

- **classFunction**: internal function which is called to solve for one step

---

def **step** (*self*, *action*)

- **classFunction**: openAI gym interface function which is called to compute one step

---

def **reset** (*self*, ***, *seed: Optional[int]*= None, *return_info: bool*= False, *options: Optional[dict]*= None)

- **classFunction**: openAI gym function which resets the system

---

def **render** (*self*, *mode*= "human")

- **classFunction**: openAI gym interface function to render the system

---

def **close** (*self*)

- **classFunction**: openAI gym interface function to close system after learning or simulation

7.4 Module: basicUtilities

Basic utility functions and constants, not depending on numpy or other python modules.

Author: Johannes Gerstmayr

Date: 2020-03-10 (created)

Notes: Additional constants are defined:

pi = 3.1415926535897932

sqrt2 = 2**0.5

g=9.81

eye2D (2x2 diagonal matrix)

eye3D (3x3 diagonal matrix)

Two variables 'gaussIntegrationPoints' and 'gaussIntegrationWeights' define integration points and weights for function GaussIntegrate(...)

def **ClearWorkspace** ()

- **function description**:

  clear all workspace variables except for system variables with '_' at beginning,

  'func' or 'module' in name; it also deletes all items in exudyn.sys and exudyn.variables,

  EXCEPT from exudyn.sys['renderState'] for pertaining the previous view of the renderer

- **notes**: Use this function with CARE! In Spyder, it is certainly safer to add the preference Run→'remove all variables before execution'. It is recommended to call ClearWorkspace() at the very beginning of your models, to avoid that variables still exist from previous computations which may destroy repeatability of results

- **example**:

```
import exudyn as exu
import exudyn.utilities
#clear workspace at the very beginning, before loading other modules and
  potentially destroying unwanted things ...
ClearWorkspace()       #cleanup
#now continue with other code
from exudyn.itemInterface import *
SC = exu.SystemContainer()
mbs = SC.AddSystem()
...
```

For examples on ClearWorkspace see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- springDamperUserFunctionNumbaJIT.py (Ex), ACFtest.py (TM), runTestExamples.py (TM)

---

def **SmartRound2String** (*x*, *prec*= 3)

- **function description**: round to max number of digits; may give more digits if this is shorter; using in general the format() with '.g' option, but keeping decimal point and using exponent where necessary

---

def **DiagonalMatrix** (*rowsColumns*, *value*= 1)

- **function description**: create a diagonal or identity matrix; used for interface.py, avoiding the need for numpy

- **input**:

  rowsColumns: provides the number of rows and columns

  value: initialization value for diagonal terms

- **output**: list of lists representing a matrix

---

def **NormL2** (*vector*)

- **function description**: compute L2 norm for vectors without switching to numpy or math module
- **input**: vector as list or in numpy format
- **output**: L2-norm of vector

For examples on NormL2 see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- bicycleIftommBenchmark.py (Ex), HydraulicActuatorStaticInitialization.py (Ex), NGsolvePistonEngine.py (Ex), reinforcementLearningRobot.py (Ex), springsDeactivateConnectors.py (Ex), ... , distanceSensor.py (TM), explicitLieGroupIntegratorTest.py (TM), fourBarMechanismIftomm.py (TM), ...

---

def **VSum** (*vector*)

- **function description**: compute sum of all values of vector
- **input**: vector as list or in numpy format
- **output**: sum of all components of vector

For examples on VSum see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- serialRobotFlexible.py (Ex), serialRobotInteractiveLimits.py (Ex), serialRobotInverseKinematics.py (Ex), serialRobotKinematicTree.py (Ex), serialRobotTSD.py (Ex), ... , movingGroundRobotTest.py (TM), serialRobotTest.py (TM)

---

def **VAdd** (*v0*, *v1*)

- **function description**: add two vectors instead using numpy
- **input**: vectors v0 and v1 as list or in numpy format
- **output**: component-wise sum of v0 and v1

For examples on VAdd see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- mobileMecanumWheelRobotWithLidar.py (Ex), NGsolvePistonEngine.py (Ex), carRollingDiscTest.py (TM), laserScannerTest.py (TM), mecanumWheelRollingDiscTest.py (TM), NGsolveCrankShaftTest.py (TM), rigidBodyCOMtest.py (TM), simulatorCouplingTwoMbs.py (TM), ...

---

def **VSub** (*v0*, *v1*)

- **function description**: subtract two vectors instead using numpy: result = v0-v1
- **input**: vectors v0 and v1 as list or in numpy format
- **output**: component-wise difference of v0 and v1

For examples on VSub see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `NGsolveCMStutorial.py` (Ex), `NGsolveGeometry.py` (Ex), `NGsolvePistonEngine.py` (Ex), `ObjectFFRFconvergenceTestHinge.py` (Ex), `NGsolveCrankShaftTest.py` (TM), `rigidBodyCOMtest.py` (TM)

---

def **VMult** (*v0*, *v1*)

- **function description**: scalar multiplication of two vectors instead using numpy: result = v0' * v1
- **input**: vectors v0 and v1 as list or in numpy format
- **output**: sum of all component wise products: c0[0]*v1[0] + v0[1]*v1[0] + ...

---

def **ScalarMult** (*scalar*, *v*)

- **function description**: multiplication vectors with scalar: result = scalar * v
- **input**: value *scalar* and vector *v* as list or in numpy format
- **output**: scalar multiplication of all components of v: [scalar*v[0], scalar*v[1], ...]

For examples on ScalarMult see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `pendulumFriction.py` (TM), `sliderCrank3Dbenchmark.py` (TM), `sliderCrank3Dtest.py` (TM)

---

def **Normalize** (*v*)

- **function description**: take a 3D vector and return a normalized 3D vector (L2Norm=1)

- **input**: vector v as list or in numpy format
- **output**: vector v multiplied with scalar such that L2-norm of vector is 1

For examples on Normalize see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ballBearningModel.py (Ex), contactCurveWithLongCurve.py (Ex), NGsolveCMStutorial.py (Ex), NGsolveGeometry.py (Ex), NGsolvePistonEngine.py (Ex), ... , NGsolveCrankShaftTest.py (TM)

---

def **Vec2Tilde** (*v*)

- **function description**: apply tilde operator (skew) to 3D-vector and return skew matrix
- **input**: 3D vector v as list or in numpy format
- **output**:

  matrix as list of lists with the skew-symmetric matrix from v:

$$\begin{bmatrix} 0 & -v[2] & v[1] \\ v[2] & 0 & -v[0] \\ -v[1] & v[0] & 0 \end{bmatrix}$$

For examples on Vec2Tilde see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- explicitLieGroupMBSTest.py (TM)

---

def **Tilde2Vec** (*m*)

- **function description**: take skew symmetric matrix and return vector (inverse of Skew(...))
- **input**: list of lists containing a skew-symmetric matrix (3x3)
- **output**: list containing the vector v (inverse function of Vec2Tilde(...))

---

def **GaussIntegrate** (*functionOfX*, *integrationOrder*, *a*, *b*)

- **function description**: compute numerical integration of functionOfX in interval [a,b] using Gaussian integration
- **input**:

functionOfX: scalar, vector or matrix-valued function with scalar argument (X or other variable)

integrationOrder: odd number in {1,3,5,7,9}; currently maximum order is 9

a: integration range start

b: integration range end

- **output**: (scalar or vectorized) integral value

---

def **LobattoIntegrate** (*functionOfX*, *integrationOrder*, *a*, *b*)

- **function description**: compute numerical integration of functionOfX in interval [a,b] using Lobatto integration
- **input**:

  functionOfX: scalar, vector or matrix-valued function with scalar argument (X or other variable)

  integrationOrder: odd number in {1,3,5}; currently maximum order is 5

  a: integration range start

  b: integration range end
- **output**: (scalar or vectorized) integral value

7.5 Module: beams

Beam utility functions, e.g. for creation of sequences of straight or curved beams.

Author: Johannes Gerstmayr

Date: 2022-01-30 (created)

Notes: For a list of plot colors useful for matplotlib, see also utilities.PlotLineCode(...)

def **GenerateStraightLineANCFCable2D** (*mbs*, *positionOfNode0*, *positionOfNode1*, *numberOfElements*, *cableTemplate*, *massProportionalLoad*= [0,0,0], *fixedConstraintsNode0*= [0,0,0,0], *fixedConstraintsNode1*= [0,0,0,0], *nodeNumber0*= -1, *nodeNumber1*= -1)

- **function description**: generate 2D ANCF cable elements along straight line given by two points; applies discretization (numberOfElements) and may apply gravity as well as nodal constraints
- **input**:

  mbs: the system where ANCF cables are added

  positionOfNode0: 3D position (list or np.array) for starting point of line

positionOfNode1: 3D position (list or np.array) for end point of line

numberOfElements: for discretization of line

cableTemplate: a ObjectANCFCable2D object, containing the desired cable properties; cable length and node numbers are set automatically

massProportionalLoad: a 3D list or np.array, containing the gravity vector or zero

fixedConstraintsNode0: a list of 4 binary values, indicating the coordinate contraints on the first node (x,y-position and x,y-slope); use None in order to apply no constraints

fixedConstraintsNode1: a list of 4 binary values, indicating the coordinate contraints on the last node (x,y-position and x,y-slope); use None in order to apply no constraints

nodeNumber0: if set other than -1, this node number defines the node that shall be used at positionOfNode0

nodeNumber1: if set other than -1, this node number defines the node that shall be used at positionOfNode1

- **output**: returns a list containing created items [cableNodeList, cableObjectList, loadList, cableNodePositionList, cableCoordinateConstraintList]
- **notes**: use GenerateStraightBeam instead
- **example**:

```
see Examples/ANCF_cantilever_test.py
```

For examples on GenerateStraightLineANCFCable2D see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ANCFALEtest.py (Ex), ANCFcantileverTest.py (Ex), ANCFrotatingCable2D.py (Ex), beltDriveALE.py (Ex), beltDriveReevingSystem.py (Ex), ... , ANCFbeltDrive.py (TM), ANCFgeneralContactCircle.py (TM), ANCFmovingRigidBodyTest.py (TM), ...

---

def **GenerateStraightLineANCFCable** (*mbs*, *positionOfNode0*, *positionOfNode1*, *numberOfElements*, *cableTemplate*, *massProportionalLoad*= [0,0,0], *fixedConstraintsNode0*= [0,0,0, 0,0,0], *fixedConstraintsNode1*= [0,0,0, 0,0,0], *nodeNumber0*= -1, *nodeNumber1*= -1)

- **function description**: generate 3D ANCF cable elements along straight line given by two points; applies discretization (numberOfElements) and may apply gravity as well as nodal constraints
- **input**:

mbs: the system where ANCF cables are added

positionOfNode0: 3D position (list or np.array) for starting point of line

positionOfNode1: 3D position (list or np.array) for end point of line

numberOfElements: for discretization of line

cableTemplate: a ObjectANCFCable object, containing the desired cable properties; cable length and node numbers are set automatically

massProportionalLoad: a 3D list or np.array, containing the gravity vector or zero

fixedConstraintsNode0: a list of binary values, indicating the coordinate contraints on the first node (position and slope); 4 coordinates for 2D and 6 coordinates for 3D node; use None in order to apply no constraints

fixedConstraintsNode1: a list of binary values, indicating the coordinate contraints on the last node (position and slope); 4 coordinates for 2D and 6 coordinates for 3D node; use None in order to apply no constraints

nodeNumber0: if set other than -1, this node number defines the node that shall be used at positionOfNode0

nodeNumber1: if set other than -1, this node number defines the node that shall be used at positionOfNode1

- **output**: returns a list containing created items [cableNodeList, cableObjectList, loadList, cableNodePositionList, cableCoordinateConstraintList]

- **example**:

```
see Examples/ANCF_cantilever_test.py
```

For examples on GenerateStraightLineANCFCable see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ANCFcableCantilevered.py (Ex), ANCFcable2DuserFunction.py (TM)

---

def **GenerateStraightBeam** (*mbs*, *positionOfNode0*, *positionOfNode1*, *numberOfElements*, *beamTemplate*, *gravity*= [0,0,0], *fixedConstraintsNode0*= None, *fixedConstraintsNode1*= None, *nodeNumber0*= -1, *nodeNumber1*= -1)

- **function description**: generic function to create beam elements along straight line given by two points; applies discretization (numberOfElements) and may apply gravity as well as nodal constraints

- **input**:

mbs: the system where beam elements are added

positionOfNode0: 3D position (list or np.array) for starting point of line

positionOfNode1: 3D position (list or np.array) for end point of line

numberOfElements: for discretization of line

beamTemplate: a Beam object (ObjectANCFCable2D, ObjectBeamGeometricallyExact2D, ObjectALEANCFCable2D, etc.), containing the desired beam type and properties; finite (beam) element length and node numbers are set automatically; for ALE element, the beamTemplate.nodeNumbers[2] must be set in the template and will not be overwritten

gravity: a 3D list or np.array, containing the gravity vector or zero

fixedConstraintsNode0: a list of binary values, indicating the coordinate contraints on the first node (position and slope); must agree with the number of coordinates in the node; use None to add no constraints

fixedConstraintsNode1: a list of binary values, indicating the coordinate contraints on the last node (position and slope); must agree with the number of coordinates in the node; use None to add no constraints

nodeNumber0: if set other than -1, this node number defines the node that shall be used at positionOfNode0

nodeNumber1: if set other than -1, this node number defines the node that shall be used at positionOfNode1

– **output**: returns a list containing created items [cableNodeList, cableObjectList, loadList, cableNodePositionList, cableCoordinateConstraintList]

– **example**:

```
import exudyn as exu
from exudyn.utilities import * #includes exudyn.beams
SC = exu.SystemContainer()
mbs = SC.AddSystem()
#example of flexible pendulum
beamTemplate = ObjectBeamGeometricallyExact2D(physicsMassPerLength=0.02,
                    physicsCrossSectionInertia=8e-9,
                    physicsBendingStiffness=8e-4,
                    physicsAxialStiffness=2000,
                    physicsShearStiffness=650,
                    visualization=VObjectBeamGeometricallyExact2D(drawHeight =
  0.002))
#create straight beam with 10 elements, apply gravity and fix (x,y) position of
  node 0 (rotation left free)
beamInfo = GenerateStraightBeam(mbs, positionOfNode0=[0,0,0], positionOfNode1
  =[0.5,0,0],
                                numberOfElements=10, beamTemplate=beamTemplate,
                                gravity=[0,-9.81,0], fixedConstraintsNode0
  =[1,1,0],)
#beamInfo contains nodes, beamObjects, loads, etc.
#Assemble and solve
```

For examples on GenerateStraightBeam see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- beamTutorial.py (Ex), pendulumGeomExactBeam2Dsimple.py (Ex)

---

def **GenerateCircularArcANCFCable2D** (*mbs*, *positionOfNode0*, *radius*, *startAngle*, *arcAngle*, *numberOfElements*, *cableTemplate*, *massProportionalLoad*= [0,0,0], *fixedConstraintsNode0*= [0,0,0,0], *fixedConstraintsNode1*= [0,0,0,0], *nodeNumber0*= -1, *nodeNumber1*= -1, *setCurvedReferenceConfiguration*= True, *verboseMode*= False)

- **function description**: generate cable elements along circular arc with given start point, radius, start angle (measured relative to x-axis, in positive rotation sense) and angle of arc
- **input**:

  mbs: the system where ANCF cables are added

  positionOfNode0: 3D position (list or np.array) for starting point of line

  radius: radius of arc

  startAngle: start angle of arc in radians ($0 \dots 2\pi$), defines the direction of the slope vector, measured relative to x-axis, in positive rotation sense

  arcAngle: total angle of arc in radians ($0 \dots 2\pi$), measured in positive rotation sense (negative angle reverts curvature and center point of circle)

  numberOfElements: for discretization of arc

  cableTemplate: a ObjectANCFCable2D object, containing the desired cable properties; cable length and node numbers are set automatically

  massProportionalLoad: a 3D list or np.array, containing the gravity vector or zero

  fixedConstraintsNode0: a list of 4 binary values, indicating the coordinate contraints on the first node (x,y-position and x,y-slope)

  fixedConstraintsNode1: a list of 4 binary values, indicating the coordinate contraints on the last node (x,y-position and x,y-slope)

  nodeNumber0: if set other than -1, this node number defines the node that shall be used at positionOfNode0

  nodeNumber1: if set other than -1, this node number defines the node that shall be used at positionOfNode1

  setCurvedReferenceConfiguration: if True, the curvature +/-(1/radius) is set as a reference configuration (sign depends on arcAngle); if False, the reference configuration is straight

  verboseMode: if True, prints out information on created nodes

- **output**: returns a list [cableNodeList, cableObjectList, loadList, cableNodePositionList, cableCoordinateConstraintList]

For examples on GenerateCircularArcANCFCable2D see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ANCFbeltDrive.py (TM), ANCFgeneralContactCircle.py (TM)

---

def **CreateReevingCurve** (*circleList*, *drawingLinesPerCircle*= 64, *numberOfANCFnodes*= -1, *removeLastLine*= False, *removeFirstLine*= False, *radialOffset*= 0., *closedCurve*= False, *graphicsElementsPerCircle*= 64, *graphicsNodeSize*= 0, *colorCircles*= [0.,0.5,1.,1.], *colorLines*= [1.,0.5,0.,1.])

- **function description**: CreateReevingCurve for creating the geometry of a reeving system based on circles with radius and left/right side of passing the circles; left/right is seen in the direction passing from one to the next circle
- **input**:

  circleList: list containing center position, radius and 'L' (left) or 'R' (right) passing of circle

  radialOffset: additional offset added to circles to account for half height of rope or beam

  closedCurve: if True, the system adds circleList[0] and circleList[1] at end of list and sets removeLastLine=True and removeFirstLine=False, in order to generate a closed curve according to given circles; furthermore, the number of nodes becomes equal to the number of elements in this case

  drawingLinesPerCircle: number of lines in lineData per one revolution

  numberOfANCFnodes: if not -1, function also generates nodes with equidistant distribution along curve!

  graphicsElementsPerCircle: number of drawing lines generated in graphicsDataLines per circle revolution (larger generates better approximation of circles)

  graphicsNodeSize: if not 0, addes graphics representation of nodes generated; for check if mesh is correct

  removeFirstLine: removes first line generated, which may be unwanted

  removeLastLine: removes last line generated, which may be unwanted

  colorCircles: RGBA color for circles

  colorLines: RGBA color for lines

- **output**: return a dictionary with 'ancfPointsSlopes':ancfPointsSlopes, 'elementLengths':elementLengths, 'elementCurvatures':elementCurvatures, 'totalLength':totalLength, 'circleData':circle2D, 'graphicsDataLines':graphicsDataLines, 'graphicsDataCircles':graphicsDataCircles ; 'ancfPointsSlopes' denotes 4-dimensional vector with (x/y) position and (x/y) slope coordinates in a row; 'elementLengths' is the list of curved lengths for elements between nodes (size is 1 smaller than number of nodes), 'elementCurvatures' is the list of scalar curvatures between nodes (according to list of elementLengths), 'totalLength' is the total length of the reeving line, 'circleData' represents the lines and arcs calculated for the reeving system, 'graphicsDataLines' is the graphicsData for the lines and 'graphicsDataCircles' represents the graphicsData for the circles

- **example**:

```
#list with circle center, radius and side at which rope runs
circleList = [[[0,0],0.2,'L'],
              [[0,1],0.2,'L'],
              [[0.8,0.8],0.4,'L'],
              [[1,0],0.2,'L'],
              [[0,0],0.2,'L'],
              [[0,1],0.2,'L'],
              ]
[] = CreateReevingCurve(circleList,
                        removeLastLine=True, #allows closed curve
                        numberOfANCFnodes=50)
```

For examples on CreateReevingCurve see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- beltDriveALE.py (Ex), beltDriveReevingSystem.py (Ex), beltDrivesComparison.py (Ex), bungeeJump.py (Ex), contactCurveWithLongCurve.py (Ex), ...

---

def **PointsAndSlopes2ANCFCable2D** (*mbs*, *ancfPointsSlopes*, *elementLengths*, *cableTemplate*, *massProportionalLoad*= [0,0,0], *fixedConstraintsNode0*= [0,0,0,0], *fixedConstraintsNode1*= [0,0,0,0], *firstNodeIsLastNode*= True, *elementCurvatures*= [], *graphicsSizeConstraints*= -1)

- **function description**: Create nodes and ANCFCable2D elements in MainSystem mbs from a given set of nodes, elements lengths and a template for the cable, based on output of function CreateReevingCurve(...); function works similar to GenerateStraightLineANCFCable2D, but for arbitrary geometry (curved elements); optionally add loads and constraints
- **input**:

  mbs: the system where ANCF elements and nodes are added

  ancfPointsSlopes: list of position and slopes for nodes, provided as 4D numpy arrays, as returned by CreateReevingCurve(...)

  elementLengths: list of element lengths per element, as returned by CreateReevingCurve(...)

  cableTemplate: a ObjectANCFCable2D object, containing the desired cable properties; cable length and node numbers are set automatically

  massProportionalLoad: a 3D list or np.array, containing the gravity vector to be applied to all elements or zero

  fixedConstraintsNode0: a list of 4 binary values, indicating the coordinate contraints on the first node (x,y-position and x,y-slope)

  fixedConstraintsNode1: a list of 4 binary values, indicating the coordinate contraints on the last node (x,y-position and x,y-slope)

firstNodeIsLastNode: if True, then the last node is using the node number of the first node and the curve is closed; otherwise, the first and last nodes are different, and the curve is open

elementCurvatures: optional list of pre-curvatures of elements, used to override the cableTemplate entry 'physicsReferenceCurvature'; use 0. for straight lines!

graphicsSizeConstraints: if set other than -1, it will be used as the size for drawing applied coordinate constraints

- **output**: returns a list [cableNodeList, cableObjectList, loadList, cableNodePositionList, cableCoordinateConstraintList]

For examples on PointsAndSlopes2ANCFCable2D see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- beltDriveALE.py (Ex), beltDriveReevingSystem.py (Ex), beltDrivesComparison.py (Ex), bungeeJump.py (Ex), reevingSystem.py (Ex), ...

---

def **GenerateSlidingJoint** (*mbs*, *cableObjectList*, *markerBodyPositionOfSlidingBody*, *localMarkerIndexOfStartCable*= 0, *slidingCoordinateStartPosition*= 0)

- **function description**: generate a sliding joint from a list of cables, marker to a sliding body, etc.
- **output**: returns the sliding joint object

For examples on GenerateSlidingJoint see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ANCFslidingAndALEjointTest.py (TM)

---

def **GenerateAleSlidingJoint** (*mbs*, *cableObjectList*, *markerBodyPositionOfSlidingBody*, *AleNode*, *localMarkerIndexOfStartCable*= 0, *AleSlidingOffset*= 0, *activeConnector*= True, *penaltyStiffness*= 0)

- **function description**: generate an ALE sliding joint from a list of cables, marker to a sliding body, etc.
- **output**: returns the sliding joint object

For examples on GenerateAleSlidingJoint see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ANCFslidingAndALEjointTest.py (TM)

7.6 Module: demos

The demos library includes basic demos which are available directly after installation; For advanced demos, see main/pythonDev/Examples and main/pythonDev/TestModels

Date: 2023-01-12

def **Demo1** (*showAll*= True)

- **function description**: very simple demo to show that exudyn is correctly installed; does not require graphics; similar to Examples/myFirstExample.py

For examples on Demo1 see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `xExudynConfigSpecial.py` (Ex)

---

def **Demo2** (*showAll*= True)

- **function description**: advanced demo, showing that graphics is available; similar to Examples/-rigid3Dexample.py

7.7 Module: FEM

Support functions and helper classes for import of meshes, finite element models (ABAQUS, ANSYS, NETGEN) and for generation of FFRF (floating frame of reference) objects. Note that since exudyn version 1.8.69 the mass and stiffness matrices in FEMinterface are either None or given in SciPy-sparse csr format (leading also to a new load/save fileVersion of FEMinterface)

Author: Johannes Gerstmayr; Stefan Holzinger (Abaqus and Ansys import utilities); Joachim Schöberl (support for Netgen and NGsolve [51, 52, 14] import and eigen computations)

Date: 2020-03-10 (created)

Notes: OLD internal CSR matrix storage format contains 3 float numbers per row: [row, column, value], can be converted to scipy csr sparse matrices with function CSRtoScipySparseCSR(...); the NEW format uses scipy's internal sparse csr format! To switch to the old format, set exudyn.FEM.useOldCSRformat=True

def **CompressedRowSparseToDenseMatrix** (*sparseData*)

- **function description**: convert zero-based sparse matrix data to dense numpy matrix
- **input**: sparseData: format (per row): [row, column, value] ==> converted into dense format

- **output**: a dense matrix as np.array

---

def **MapSparseMatrixIndices** (*matrix*, *sorting*)

- **function description**: resort a sparse matrix (internal CSR format) with given sorting for rows and columns; changes matrix directly! used for ANSYS matrix import

---

def **VectorDiadicUnitMatrix3D** (*v*)

- **function description**: compute diadic product of vector v and a 3D unit matrix = diadic(v,I_{3x3}); used for ObjectFFRF and CMS implementation

---

def **CyclicCompareReversed** (*list1*, *list2*)

- **function description**: compare cyclic two lists, reverse second list; return True, if any cyclic shifted lists are same, False otherwise

---

def **AddEntryToCompressedRowSparseArray** (*sparseData*, *row*, *column*, *value*)

- **function description**:

  add entry to compressedRowSparse matrix, avoiding duplicates

  value is either added to existing entry (avoid duplicates) or a new entry is appended

---

def **CSRtoRowsAndColumns** (*sparseMatrixCSR*)

- **function description**: compute rows and columns of a compressed sparse matrix and return as tuple: (rows,columns)

---

def **CSRtoScipySparseCSR** (*sparseMatrixCSR*)

- **function description**: DEPRECATED: convert internal compressed CSR to scipy.sparse csr matrix; should not be used and raises warning; use SparseTripletsToScipySparseCSR instead!

For examples on CSRtoScipySparseCSR see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `shapeOptimization.py` (Ex), `linearFEMgenericODE2Test.py` (TM)

---

def **SparseTripletsToScipySparseCSR** (*sparseTriplets*)

- **function description**: convert list of sparse triplets (or numpy array with one sparse triplet per row) into scipy.sparse csr_matrix

---

def **ScipySparseCSRtoCSR** (*scipyCSR*)

- **function description**: convert scipy.sparse csr matrix to internal compressed CSR

For examples on ScipySparseCSRtoCSR see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `linearFEMgenericODE2Test.py` (TM)

---

def **ResortIndicesOfCSRmatrix** (*mXXYYZZ*, *numberOfRows*)

- **function description**:

  resort indices of given NGsolve CSR matrix in XXXYYYZZZ format to XYZXYZXYZ format; numberOfRows must be equal to columns

  needed for import from NGsolve

---

def **ResortIndicesOfNGvector** (*vXXYYZZ*)

- **function description**: resort indices of given NGsolve vector in XXXYYYZZZ format to XYZXYZXYZ format

---

def **ResortIndicesExudyn2NGvector** (*vXYZXYZ*)

- **function description**: resort indices of given Exudyun vector XYZXYZXYZ to NGsolve vector in XXXYYYZZZ format

---

def **ConvertHexToTrigs** (*nodeNumbers*)

- **function description**: convert list of Hex8/C3D8 element with 8 nodes in nodeNumbers into triangle-List
- **notes**: works for Hex20 elements, but does only take the corner nodes for drawing!

For examples on ConvertHexToTrigs see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `objectFFRFTest.py` (TM)

---

def **ConvertTetToTrigs** (*nodeNumbers*)

- **function description**: convert list of Tet4/Tet10 element with 4 or 10 nodes in nodeNumbers into triangle-List
- **notes**: works for Tet10 elements, but does only take the corner nodes for drawing!

---

def **ConvertDenseToCompressedRowMatrix** (*denseMatrix*)

- **function description**: convert numpy.array dense matrix to OLD FEM internal compressed row sparse format; do not use!

---

def **ReadMatrixFromAnsysMMF** (*fileName*, *verbose*= False)

- **function description**:

  This function reads either the mass or stiffness matrix from an Ansys Matrix Market Format (MMF). The corresponding matrix can either be exported as dense matrix or sparse matrix.

- **input**: fileName of MMF file
- **output**: internal compressed row sparse matrix (as (nrows x 3) numpy array)
- **author**: Stefan Holzinger
- **notes**:

  A MMF file can be created in Ansys by placing the following APDL code inside the solution tree in Ansys Workbench:

  !!

  ! APDL code that exports sparse stiffnes and mass matrix in MMF format. If

  ! the dense matrix is needed, replace *SMAT with *DMAT in the following

  ! APDL code.

  ! Export the stiffness matrix in MMF format

  *SMAT,MatKD,D,IMPORT,FULL,file.full,STIFF

  *EXPORT,MatKD,MMF,fileNameStiffnessMatrix,,,

  ! Export the mass matrix in MMF format

  *SMAT,MatMD,D,IMPORT,FULL,file.full,MASS

  *EXPORT,MatMD,MMF,fileNameMassMatrix,,,

  !!

  In case a lumped mass matrix is needed, place the following APDL Code inside the Modal Analysis Tree:

  !!

  ! APDL code to force Ansys to use a lumped mass formulation (if available for

  ! used elements)

  LUMPM, ON, , 0

  !!

---

def **ReadMatrixDOFmappingVectorFromAnsysTxt** (*fileName*)

- **function description**:

  read sorting vector for ANSYS mass and stiffness matrices and return sorting vector as np.array

  the file contains sorting for nodes and applies this sorting to the DOF (assuming 3 DOF per node!)

  the resulting sorted vector is already converted to 0-based indices

---

def **ReadNodalCoordinatesFromAnsysTxt** (*fileName*, *verbose*= False)

- **function description**: This function reads the nodal coordinates exported from Ansys.
- **input**: fileName (file name ending must be .txt!)
- **output**: nodal coordinates as numpy array
- **author**: Stefan Holzinger
- **notes**:

  The nodal coordinates can be exported from Ansys by creating a named selection of the body whos mesh should to exported by choosing its geometry. Next, create a second named selcetion by using a worksheet. Add the named selection that was created first into the worksheet of the second named selection. Inside the working sheet, choose 'convert' and convert the first created named selection to 'mesh node' (Netzknoten in german) and click on generate to create the second named selection. Next, right click on the second named selection tha was created and choose 'export' and save the nodal coordinates as .txt file.

---

def **ReadElementsFromAnsysTxt** (*fileName*, *verbose*= False)

- **function description**: This function reads the nodal coordinates exported from Ansys.

- **input**: fileName (file name ending must be .txt!)
- **output**: element connectivity as numpy array
- **author**: Stefan Holzinger
- **notes**:

  The elements can be exported from Ansys by creating a named selection of the body whos mesh should to exported by choosing its geometry. Next, create a second named selcetion by using a worksheet. Add the named selection that was created first into the worksheet of the second named selection. Inside the worksheet, choose 'convert' and convert the first created named selection to 'mesh element' (Netzelement in german) and click on generate to create the second named selection. Next, right click on the second named selection tha was created and choose 'export' and save the elements as .txt file.

---

def **CMSObjectComputeNorm** (*mbs*, *objectNumber*, *outputVariableType*, *norm*= 'max', *nodeNumberList*= [])

- **function description**: compute current (max, min, ...) value for chosen ObjectFFRFreducedOrder object (CMSobject) with exu.OutputVariableType. The function operates on nodal values. This is a helper function, which can be used to conveniently compute output quantities of the CMSobject efficiently and to use it in sensors
- **input**:

  mbs: MainSystem of objectNumber

  objectNumber: number of ObjectFFRFreducedOrder in mbs

  outputVariableType: a exu.OutputVariableType out of [StressLocal, DisplacementLocal, VelocityLocal]

  norm: string containing chosen norm to be computed, out of 'Mises', 'maxNorm', 'min', 'max'; 'max' will return maximum of all components (component wise), 'min' does same but for minimum; 'maxNorm' computes np.linalg.norm for every node and then takes maximum of all norms; Mises computes von-Mises stress for every node and then takes maximum of all nodes

  nodeNumberList: list of mesh node numbers (from FEMinterface); if empty [], all nodes are used; otherwise, only given nodes are evaluated

– **output**: return value or list of values according to chosen norm as np.array

For examples on CMSObjectComputeNorm see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- netgenSTLtest.py (Ex), NGsolveCMStutorial.py (Ex)

7.7.1 CLASS MaterialBaseClass (in module FEM)

class description: INTERNAL material base class, e.g., for FiniteElement

7.7.2 CLASS KirchhoffMaterial(MaterialBaseClass) (in module FEM)

class description: class for representation of Kirchhoff (linear elastic, 3D and 2D) material

– **notes**: use planeStress=False for plane strain

def **__init__** (*self*, *youngsModulus*= None, *poissonsRatio*= None, *density*= 0, *materials*= None, *fes*= None, *planeStress*= True)

– **classFunction**: add according nodes, objects and constraints for FFRF object to MainSystem mbs; only implemented for Euler parameters

– **input**:

youngsModulus: Young's modulus for single domain and material; in case of multi-domain, it must be None

poissonsRatio: Poisson's ratio for single domain and material; in case of multi-domain, it must be None

density: density for for single domain and material; in case of multi-domain, it must be 0 or None

materials: dictionary of material dictionaries according to names in NGsolve mesh, containing youngsModulus, poissonsRatio and density per material, see ImportMeshFromNGsolve

fes: in case of materials dictionary, fes (as returned by ImportMeshFromNGsolve) has to be provided

planeStress: set True for 2D materials (currently not used)

---

def **Strain2Stress** (*self*, *strain*)

– **classFunction**: convert strain tensor into stress tensor using elasticity tensor

---

def **StrainVector2StressVector** (*self*, *strainVector*)

– **classFunction**: convert strain vector into stress vector

---

def **StrainVector2StressVector2D** (*self*, *strainVector2D*)

– **classFunction**: compute 2D stress vector from strain vector

---

def **LameParameters** (*self*)

– **classFunction**: compute Lame parameters from internal Young's modulus and Poisson ratio

– **output**: return vector [mu, lam] of Lame parameters

For examples on KirchhoffMaterial see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `CMSexampleCourse.py` (Ex), `netgenSTLtest.py` (Ex), `NGsolveCMStutorial.py` (Ex), `NGsolveCraigBampton.py` (Ex), `NGsolveFFRF.py` (Ex), ... , `NGsolveCMStest.py` (TM)

7.7.3 CLASS FiniteElement (in module FEM)

class description: finite element base class for lateron implementations of other finite elements

7.7.4 CLASS Tet4(FiniteElement) (in module FEM)

class description: simplistic 4-noded tetrahedral interface to compute strain/stress at nodal points

7.7.5 CLASS ObjectFFRFinterface (in module FEM)

class description: compute terms necessary for ObjectFFRF class used internally in FEMinterface to compute ObjectFFRF object this class holds all data for ObjectFFRF user functions

def **__init__** (*self*, *femInterface*)

- **classFunction**:

  initialize ObjectFFRFinterface with FEMinterface class

  initializes the ObjectFFRFinterface with nodes, modes, surface description and systemmatrices from FEMinterface

  data is then transfered to mbs object with classFunction AddObjectFFRF(...)

---

def **AddObjectFFRF** (*self*, *exu*, *mbs*, *positionRef*= [0,0,0], *eulerParametersRef*= [1,0,0,0], *initialVelocity*= [0,0,0], *initialAngularVelocity*= [0,0,0], *gravity*= [0,0,0], *constrainRigidBodyMotion*= True, *massProportionalDamping*= 0, *stiffnessProportionalDamping*= 0, *color*= [0.1,0.9,0.1,1.])

- **classFunction**: add according nodes, objects and constraints for FFRF object to MainSystem mbs; only implemented for Euler parameters
- **input**:

  exu: the exudyn module

  mbs: a MainSystem object

  positionRef: reference position of created ObjectFFRF (set in rigid body node underlying to ObjectFFRF)

  eulerParametersRef: reference euler parameters of created ObjectFFRF (set in rigid body node underlying to ObjectFFRF)

  initialVelocity: initial velocity of created ObjectFFRF (set in rigid body node underlying to ObjectFFRF)

  initialAngularVelocity: initial angular velocity of created ObjectFFRF (set in rigid body node underlying to ObjectFFRF)

  gravity: set [0,0,0] if no gravity shall be applied, or to the gravity vector otherwise

  constrainRigidBodyMotion: set True in order to add constraint (Tisserand frame) in order to suppress rigid motion of mesh nodes

  color: provided as list of 4 RGBA values

  add object to mbs as well as according nodes

---

def **UFforce** (*self*, *exu*, *mbs*, *t*, *q*, *q_t*)

- **classFunction**: optional forceUserFunction for ObjectFFRF (per default, this user function is ignored)

---

def **UFmassGenericODE2** (*self*, *exu*, *mbs*, *t*, *q*, *q_t*)

- **classFunction**: optional massMatrixUserFunction for ObjectFFRF (per default, this user function is ignored)

For examples on ObjectFFRFinterface see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `objectFFRFTest2.py` (TM)

7.7.6 CLASS ObjectFFRFreducedOrderInterface (in module FEM)

class description: compute terms necessary for ObjectFFRFreducedOrder class used internally in FEMinterface to compute ObjectFFRFreducedOrder dictionary this class holds all data for ObjectFFRFreducedOrder user functions

def **__init__** (*self*, *femInterface*= None, *rigidBodyNodeType*= 'NodeType.RotationEulerParameters', *roundMassMatrix*= 1e-13, *roundStiffnessMatrix*= 1e-13)

- **classFunction**:

  initialize ObjectFFRFreducedOrderInterface with FEMinterface class

  initializes the ObjectFFRFreducedOrderInterface with nodes, modes, surface description and reduced system matrices from FEMinterface

  data is then transfered to mbs object with classFunction AddObjectFFRFreducedOrderWithUserFunctions(...)

- **input**:

  femInterface: must provide nodes, surfaceTriangles, modeBasis, massMatrix, stiffness; if femInterface=None, an empty ObjectFFRFreducedOrderInterface instance is created which may be used to load data with LoadFromFile()

  roundMassMatrix: use this value to set entries of reduced mass matrix to zero which are below the treshold

roundStiffnessMatrix: use this value to set entries of reduced stiffness matrix to zero which are below the treshold

---

def **SaveToFile** (*self*, *fileName*, *fileVersion*= 1)

- **classFunction**: save all data to a data filename; can be used to avoid loading femInterface and FE data
- **input**:

  fileName: string for path and file name without ending ==> ".npy" will be added

  fileVersion: FOR EXPERTS: this allows to store in older format, will be recovered when loading; must be integer; version must by > 0; the default value will change in future!
- **output**: stores file

---

def **LoadFromFile** (*self*, *fileName*, *mode*= None)

- **classFunction**:

  load all data (nodes, elements, ...) from a data filename previously stored with SaveToFile(...).

  this function is much faster than the text-based import functions
- **input**:

  fileName: string for path and file name without ending ==> ".npy" will be added

  mode: choose between different file formats (NPY and NPZ); Note: NPY only works for Numpy 1.x, not for Numpy >= 2.0
- **output**: loads data into fem (note that existing values are not overwritten!)

---

def **AddObjectFFRFreducedOrderWithUserFunctions** (*self*, *exu*, *mbs*, *positionRef*= [0,0,0], *initialVelocity*= [0,0,0], *rotationMatrixRef*= [], *initialAngularVelocity*= [0,0,0], *gravity*= [0,0,0], *UFforce*= 0, *UFmassMatrix*= 0, *massProportionalDamping*= 0, *stiffnessProportionalDamping*= 0, *color*= [0.1,0.9,0.1,1.], *eulerParametersRef*= [])

- **classFunction**: add according nodes, objects and constraints for ObjectFFRFreducedOrder object to MainSystem mbs; use this function with userfunctions=0 in order to use internal C++ functionality, which is approx. 10x faster; implementation of userfunctions also available for rotation vector (Lie group formulation), which needs further testing
- **input**:

  exu: the exudyn module

  mbs: a MainSystem object

  positionRef: reference position of created ObjectFFRFreducedOrder (set in rigid body node underlying to ObjectFFRFreducedOrder)

  initialVelocity: initial velocity of created ObjectFFRFreducedOrder (set in rigid body node underlying to ObjectFFRFreducedOrder)

  rotationMatrixRef: reference rotation of created ObjectFFRFreducedOrder (set in rigid body node underlying to ObjectFFRFreducedOrder); if [], it becomes the unit matrix

  initialAngularVelocity: initial angular velocity of created ObjectFFRFreducedOrder (set in rigid body node underlying to ObjectFFRFreducedOrder)

  eulerParametersRef: DEPRECATED, use rotationParametersRef or rotationMatrixRef in future: reference euler parameters of created ObjectFFRFreducedOrder (set in rigid body node underlying to ObjectFFRFreducedOrder)

  gravity: set [0,0,0] if no gravity shall be applied, or to the gravity vector otherwise

  UFforce: (OPTIONAL, computation is slower) provide a user function, which computes the quadratic velocity vector and applied forces; see example

  UFmassMatrix: (OPTIONAL, computation is slower) provide a user function, which computes the quadratic velocity vector and applied forces; see example

  massProportionalDamping: Rayleigh damping factor for mass proportional damping (multiplied with reduced mass matrix), added to floating frame/modal coordinates only

  stiffnessProportionalDamping: Rayleigh damping factor for stiffness proportional damping, added to floating frame/modal coordinates only (multiplied with reduced stiffness matrix)

  color: provided as list of 4 RGBA values

- **example**:

```
#example of a user function for forces:
def UFforceFFRFreducedOrder(mbs, t, itemIndex, qReduced, qReduced_t):
    return cms.UFforceFFRFreducedOrder(exu, mbs, t, qReduced, qReduced_t)
#example of a user function for mass matrix:
def UFmassFFRFreducedOrder(mbs, t, itemIndex, qReduced, qReduced_t):
    return cms.UFmassFFRFreducedOrder(exu, mbs, t, qReduced, qReduced_t)
```

def **UFmassFFRFreducedOrder** (*self*, *exu*, *mbs*, *t*, *qReduced*, *qReduced_t*)

- **classFunction**: CMS mass matrix user function; qReduced and qReduced_t contain the coordiantes of the rigid body node and the modal coordinates in one vector!

---

def **UFforceFFRFreducedOrder** (*self*, *exu*, *mbs*, *t*, *qReduced*, *qReduced_t*)

- **classFunction**: CMS force matrix user function; qReduced and qReduced_t contain the coordiantes of the rigid body node and the modal coordinates in one vector!

---

def **AddObjectFFRFreducedOrder** (*self*, *mbs*, *positionRef*= [0,0,0], *initialVelocity*= [0,0,0], *rotationMatrixRef*= [], *initialAngularVelocity*= [0,0,0], *massProportionalDamping*= 0, *stiffnessProportionalDamping*= 0, *gravity*= [0,0,0], *color*= [0.1,0.9,0.1,1.])

- **classFunction**: add according nodes, objects and constraints for ObjectFFRFreducedOrder object to MainSystem mbs; use this function in order to use internal C++ functionality, which is approx. 10x faster than AddObjectFFRFreducedOrderWithUserFunctions(...)
- **input**:

  exu: the exudyn module

  mbs: a MainSystem object

  positionRef: reference position of created ObjectFFRFreducedOrder (set in rigid body node underlying to ObjectFFRFreducedOrder)

  initialVelocity: initial velocity of created ObjectFFRFreducedOrder (set in rigid body node underlying to ObjectFFRFreducedOrder)

  rotationMatrixRef: reference rotation of created ObjectFFRFreducedOrder (set in rigid body node underlying to ObjectFFRFreducedOrder); if [], it becomes the unit matrix

  initialAngularVelocity: initial angular velocity of created ObjectFFRFreducedOrder (set in rigid body node underlying to ObjectFFRFreducedOrder)

  massProportionalDamping: Rayleigh damping factor for mass proportional damping, added to floating frame/modal coordinates only

  stiffnessProportionalDamping: Rayleigh damping factor for stiffness proportional damping, added to floating frame/modal coordinates only

  gravity: set [0,0,0] if no gravity shall be applied, or to the gravity vector otherwise

  color: provided as list of 4 RGBA values

For examples on ObjectFFRFreducedOrderInterface see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- CMSexampleCourse.py (Ex), netgenSTLtest.py (Ex), NGsolveCMStutorial.py (Ex), NGsolveCraigBampton.py (Ex), NGsolveFFRF.py (Ex), ... , abaqusImportTest.py (TM), NGsolveCMStest.py (TM), NGsolveCrankShaftTest.py (TM), ...

7.7.7 CLASS HCBstaticModeSelection(Enum) (in module FEM)

class description: helper calss for function ComputeHurtyCraigBamptonModes, declaring some computation options. It offers the following options:
- allBoundaryNodes: compute a single static mode for every boundary coordinate
- RBE2: static modes only for rigid body motion at boundary nodes; using rigid boundary surfaces (additional stiffening)
- RBE3: static modes only for rigid body motion at boundary nodes; averaged rigid body motion at boundary surfaces (leads to deformation at boundaries)
- noStaticModes: do not compute static modes, only eigen modes (not recommended; usually only for tests)

7.7.8 CLASS FEMinterface (in module FEM)

class description: general interface to different FEM / mesh imports and export to EXUDYN functions use this class to import meshes from different meshing or FEM programs (NETGEN/NGsolve [14], ABAQUS, ANSYS, ..) and store it in a unique format do mesh operations, compute eigenmodes and reduced basis, etc. load/store the data efficiently with LoadFromFile(...), SaveToFile(...) if import functions are slow export to EXUDYN objects

def **__init__** (*self*)

- **classFunction**: initalize all data of the FEMinterface by, e.g., `fem = FEMinterface()`
- **example**:

```
#**** this is not an example, just a description for internal variables ****
#default values for member variables stored internally in FEMinterface fem and
  typical structure:
dictionary of different node lists:
fem.nodes = {}                  # {'Position':np.array([[x0,y0,z0],...]), '
  RigidBodyRxyz':np.array([[x0,y0,z0,alpha0,beta0,gamma0],...]),  },...]
list of elements (element connectivity):
fem.elements = []               # [{'Name':'identifier', 'Tet4':np.array([[n0,n1,n2,
  n3],...]), 'Hex8':np.array([[n0,...,n7],...]),  },...]
fem.massMatrix = None           # scipy csr_matrix
fem.stiffnessMatrix= None       # scipy csr_matrix
surface sets with faces, usually for drawing:
```

```
fem.surface = []                # [{'Name':'identifier', 'Trigs':np.array([[n0,n1,n2
  ],...]), 'Quads':np.array([[n0,...,n3],...]),  },...]
node sets for boundary conditions, etc.
fem.nodeSets = []               # [{'Name':'identifier', 'NodeNumbers':np.array([n_0
  ,...,n_ns]), 'NodeWeights':np.array([w_0,...,w_ns])},...]
element sets, e.g., for different domains, etc.
fem.elementSets = []            # [{'Name':'identifier', 'ElementNumbers':np.array([
  n_0,...,n_ns])},...]
mode basis: 'NormalModes' are eigenmodes, 'HCBmodes' are Craig-Bampton modes
  including static modes:
fem.modeBasis = {}              # {'matrix':np.array([[Psi_00,Psi_01, ..., Psi_0m
  ],...,[Psi_n0,Psi_n1, ..., Psi_nm]]),'type':'NormalModes'}
eigenvalues related to eigenvectors in mode basis:
fem.eigenValues = []            # np.array([ev0, ev1, ...])
fem.postProcessingModes = {}    # {'matrix':<matrix (np.array) containing stress
  components (xx,yy,zz,yz,xz,xy) in each column, rows are for every mesh node>,'
  outputVariableType':exudyn.OutputVariableType.StressLocal}
```

---

def **GetDictionary** (*self*)

- **classFunction**: get dictionary containing current data of FEMinterface

---

def **SetWithDictionary** (*self*, *dictData*, *warn*= True, *fromNPZ*= False)

- **classFunction**: set dictionary containing current data for FEMinterface; used internally

---

def **SaveToFile** (*self*, *fileName*, *fileVersion*= 3, *mode*= None)

- **classFunction**: save all data (nodes, elements, ...) to a data filename; this function is much faster than the text-based import functions; note that HDF5 and PKL formats lead to smaller files
- **input**:

  fileName: string for path and file name; if no ending is provided ==> ".npy" will be added and NumPy format will be used; alternatives: '.pkl' ending uses Python's pickle method (smaller files) and '.hdf5' uses the HDF5 file format, but requires the python package h5py to be installed!

fileVersion: FOR EXPERTS: this allows to store in older format, will be recovered when loading; must be integer; version must by > 0

mode: default: numpy format ('NPZ'); alternatives: 'HDF5' (requires h5py package) and 'PKL' (pickle); NPY (deprecated, under Numpy 1.x)

- **output**:

  stores file

  ***nodes*: test with 10-node tets and 86154 nodes, 50752 elements and 20 modes (incl. stress modes) gives the timings for save+load: [NPY: 2.10s, PKL: 0.76s, HDF5: 0.69s] and file sizes [NPY: 1032MB, PKL: 580MB, HDF5: 581MB]

---

def **LoadFromFile** (*self*, *fileName*, *forceVersion*= None, *mode*= None)

- **classFunction**:

  load all data (nodes, elements, ...) from a data filename previously stored with SaveToFile(...).

  this function is much faster than the text-based import functions

- **input**:

  fileName: string for path and file name; if no ending is provided ==> ".npz" will be added and NumPy format will be assumed; alternatives: '.pkl' ending uses Python's pickle method and '.hdf5' uses the HDF5 file format, but requires the python package h5py to be installed!

  forceVersion: FOR EXPERTS: this allows to store in older format, will be recovered when loading; must be integer; for old files, use forceVersion=0

- **output**: loads data into fem (note that existing values are not overwritten!); returns file version or None if version is not available

---

def **ImportFromAbaqusInputFile** (*self*, *fileName*, *typeName*= 'Part', *name*= 'Part-1', *verbose*= False, *createSurfaceTrigs*= True, *surfaceTrigsAll*= False)

- **classFunction**:

  import nodes and elements from Abaqus input file and create surface elements;

  node numbers in elements are converted from 1-based indices to python's 0-based indices;

  This function can only import one part or instance; this means that you have to merge all

  instances or parts in order to use this function for import of flexible bodies for order reduction methods

- **input**:

  fileName: file name incl. path

  typeName: this is what is searched for regarding nodes and elements, see your .inp file

  name: if there are several parts, this name should address the according part name

  verbose: use True for some debug information

  createSurfaceTrigs: if True, triangles are created for visualization (triangles both for Tet and Hex elements)

  surfaceTrigsAll: if False, visualization triangles are created at the surface; if True, surface triangles are created also for interior elements

- **output**: return node numbers as numpy array

- **notes**: only works for Hex8, Hex20, Tet4 and Tet10 (C3D4, C3D8, C3D8R, C3D10, C3D20, C3D20R) elements; some functionality is untested and works in limited cases; only works for one single part or instance

---

def **ReadMassMatrixFromAbaqus** (*self*, *fileName*, *type*= 'SparseRowColumnValue')

- **classFunction**:

  read mass matrix from compressed row text format (exported from Abaqus); in order to export system matrices, write the following lines in your Abaqus input file:

  *STEP

  *MATRIX GENERATE, STIFFNESS, MASS

  *MATRIX OUTPUT, STIFFNESS, MASS, FORMAT=COORDINATE

  *End Step

---

def **ReadStiffnessMatrixFromAbaqus** (*self*, *fileName*, *type*= 'SparseRowColumnValue')

- **classFunction**: read stiffness matrix from compressed row text format (exported from Abaqus)

---

def **GetNodesOfNGsolveBoundary** (*self*, *mesh*, *boundaryName*)

– **classFunction**: internal function to get ngsolve mesh nodes of boundary with name boundaryName

---

def **CreateNGsolveBoundaryNodeSets** (*self*, *mesh*, *boundaryNamesList*= None, *warnNodeSets*= True)

– **classFunction**: create node sets for given (or all) boundaries in NGsolve; node sets are added to existing node sets

– **input**:

mesh: a previously created `ngs.mesh` (NGsolve mesh, see examples)

boundaryNamesList: a List of boundary names used to define mesh boundaries or None; if given, node sets are only created for the given boundary names

– **output**: list of nodeSets according to FEMinterface nodeSets structure, a dictionary with 'Name', 'NodeNumbers' and 'NodeWeights'

---

def **ImportMeshFromNGsolve** (*self*, *mesh*, *density*= None, *youngsModulus*= None, *poissonsRatio*= None, *materials*= None, *createBoundaryNodeSets*= True, *boundaryNamesList*= None, *verbose*= False, *meshOrder*= 1, ***kwargs*)

– **classFunction**: import mesh from NETGEN/NGsolve and setup mechanical problem

– **input**:

mesh: a previously created `ngs.mesh` (NGsolve mesh, see examples)

youngsModulus: In case of single material: Young's modulus used for mechanical model

poissonsRatio: In case of single material: Poisson's ratio used for mechanical model

density: In case of single material: density used for mechanical model

materials: dictionary of material dictionaries according to names in NGsolve mesh, containing youngsModulus, poissonsRatio and density per material, see example

createBoundaryNodeSets: if True, during import named boundaries conditions of the mesh are transformed into node sets for further use during mode creation, etc.

boundaryNamesList: given as list of boundary names to be used for boundary node sets or None (creating node sets for all boundaries)

meshOrder: use 1 for linear elements and 2 for second order elements (recommended to use 2 for much higher accuracy!)

verbose: set True to print out some status information

- **output**: creates according nodes, elements, in FEM and returns [bfM, bfK, fes] which are the (mass matrix M, stiffness matrix K) bilinear forms and the finite element space fes
- **author**: Johannes Gerstmayr, Joachim Schöberl
- **notes**: setting ngsolve.SetNumThreads(nt) you can select the number of treads that are used for assemble or other functionality with NGsolve functionality
- **example**:

```
... #assume you have a FEMinterface fem and a NGsolve mesh
#we have to define specific materials and (if still in the mesh), the default
  material
materials = {'default':{'youngsModulus':2.1e11, 'poissonsRatio':0.3, 'density'
  :7800},
             'steel':{'youngsModulus':2.1e11, 'poissonsRatio':0.3, 'density':7800},
             'aluminum':{'youngsModulus':7e10, 'poissonsRatio':0.35, 'density'
  :2700},
             }
fem.ImportMeshFromNGsolve(self, mesh, materials)
#==> fem has now nodes, elements, node sets, etc. set according to mesh
```

---

def **ComputeEigenmodesNGsolve** (*self*, *bfM*, *bfK*, *nModes*, *maxEigensolveIterations*= 40, *excludeRigidBodyModes*= 0, *verbose*= False)

- **classFunction**: compute nModes smallest eigenvalues and eigenmodes from mass and stiffness-Matrix; store mode vectors in modeBasis, but exclude a number of 'excludeRigidBodyModes' rigid body modes from modeBasis; uses scipy for solution of generalized eigenvalue problem
- **input**:

  nModes: prescribe the number of modes to be computed; total computed modes are (nModes+excludeRigid
  but only nModes with smallest absolute eigenvalues are considered and stored

  excludeRigidBodyModes: if rigid body modes are expected (in case of free-free modes), then this number specifies the number of eigenmodes to be excluded in the stored basis (usually 6 modes in 3D)

  maxEigensolveIterations: maximum number of iterations for iterative eigensolver; default=40

  verbose: if True, output some relevant information during solving
- **output**: eigenmodes are stored internally in FEMinterface as 'modeBasis' and eigenvalues as 'eigenValues'
- **author**: Johannes Gerstmayr, Joachim Schöberl

---

def **ComputeHurtyCraigBamptonModesNGsolve** (*self*, *bfM*, *bfK*, *boundaryNodesList*, *nEigenModes*, *maxEigensolveIterations*= 40, *verbose*= False)

- **classFunction**: compute static and eigen modes based on Hurty-Craig-Bampton, for details see theory part Section 5.5. This function uses internal computational functionality of NGsolve and is often much faster than the scipy variant
- **input**:

  bfM: bilinearform for mass matrix as retured in ImportMeshFromNGsolve(...)

  bfK: bilinearform for stiffness matrix as retured in ImportMeshFromNGsolve(...)

  boundaryNodesList: [nodeList0, nodeList1, ...] a list of node lists, each of them representing a set of 'Position' nodes for which a rigid body interface (displacement/rotation and force/torque) is created; NOTE THAT boundary nodes may not overlap between the different node lists (no duplicated node indices!)

  nEigenModes: number of eigen modes in addition to static modes (may be zero for RBE2 computationMode); eigen modes are computed for the case where all rigid body motions at boundaries are fixed; only smallest nEigenModes absolute eigenvalues are considered

  maxEigensolveIterations: maximum number of iterations for iterative eigensolver; default=40

  verbose: if True, output some relevant information during solving
- **output**: stores computed modes in self.modeBasis and abs(eigenvalues) in self.eigenValues
- **author**: Johannes Gerstmayr, Joachim Schöberl

---

def **ComputePostProcessingModesNGsolve** (*self*, *fes*, *material*= 0, *outputVariableType*= 'OutputVariableType.StressLocal', *verbose*= False)

- **classFunction**: compute special stress or strain modes in order to enable visualization of stresses and strains in ObjectFFRFreducedOrder; takes a NGsolve fes as input and uses internal NGsolve methods to efficiently compute stresses or strains
- **input**:

  fes: finite element space as retured in ImportMeshFromNGsolve(...)

  material: specify material properties for computation of stresses, using a material class, e.g. material = KirchhoffMaterial(Emodulus, nu, rho); not needed for strains (material = 0)

  outputVariableType: specify either exudyn.OutputVariableType.StressLocal or exudyn.OutputVariableType as the desired output variables

- **output**: post processing modes are stored in FEMinterface in local variable postProcessingModes as a dictionary, where 'matrix' represents the modes and 'outputVariableType' stores the type of mode as a OutputVariableType
- **author**: Johannes Gerstmayr, Joachim Schöberl
- **notes**: This function is implemented in Python and rather slow for larger meshes; for NGsolve / Netgen meshes, see the according ComputePostProcessingModesNGsolve function, which is usually much faster

---

def **GetMassMatrix** (*self*, *sparse*= True)

- **classFunction**: get sparse mass matrix in according format

---

def **GetStiffnessMatrix** (*self*, *sparse*= True)

- **classFunction**: get sparse stiffness matrix in according format

---

def **NumberOfNodes** (*self*)

- **classFunction**: get total number of nodes

---

def **GetNodePositionsAsArray** (*self*)

- **classFunction**: get node points as array; only possible, if there exists only one type of Position nodes
- **notes**: in order to obtain a list of certain node positions, see example
- **example**:

```
p=GetNodePositionsAsArray(self)[42] #get node 42 position
nodeList=[1,13,42]
pArray=GetNodePositionsAsArray(self)[nodeList] #get np.array with positions of node
   indices
```

---

def **GetNodePositionsMean** (*self*, *nodeNumberList*)

- **classFunction**: get mean (average) position of nodes defined by list of node numbers

---

def **NumberOfCoordinates** (*self*)

- **classFunction**: get number of total nodal coordinates

---

def **GetNodeAtPoint** (*self*, *point*, *tolerance*= 1e-5, *raiseException*= True)

- **classFunction**:

  get node number for node at given point, e.g. p=[0.1,0.5,-0.2], using a tolerance (+/-) if coordinates are available only with reduced accuracy

  if not found, it returns an invalid index

---

def **GetNodesInPlane** (*self*, *point*, *normal*, *tolerance*= 1e-5)

- **classFunction**:

  get node numbers in plane defined by point p and (normalized) normal vector n using a tolerance for the distance to the plane

  if not found, it returns an empty list

---

def **GetNodesInCube** (*self*, *pMin*, *pMax*)

- **classFunction**: get node numbers in cube, given by pMin and pMax, containing the minimum and maximum x, y, and z coordinates

- **output**: returns list of nodes; if no nodes found, return an empty list
- **example**:

```
nList = GetNodesInCube([-1,-0.2,0],[1,0.5,0.5])
```

---

def **GetNodesOnLine** (*self*, *p1*, *p2*, *tolerance*= 1e-5)

- **classFunction**: get node numbers lying on line defined by points p1 and p2 and tolerance, which is accepted for points slightly outside the surface

---

def **GetNodesOnCylinder** (*self*, *p1*, *p2*, *radius*, *tolerance*= 1e-5)

- **classFunction**:

  get node numbers lying on cylinder surface; cylinder defined by cylinder axes (points p1 and p2),

  cylinder radius and tolerance, which is accepted for points slightly outside the surface

  if not found, it returns an empty list

---

def **GetNodesOnCircle** (*self*, *point*, *normal*, *r*, *tolerance*= 1e-5)

- **classFunction**:

  get node numbers lying on a circle, by point p, (normalized) normal vector n (which is the axis of the circle) and radius r

  using a tolerance for the distance to the plane

  if not found, it returns an empty list

---

def **GetNodeWeightsFromSurfaceAreas** (*self*, *nodeList*, *normalizeWeights*= True)

- **classFunction**:

  return list of node weights based on surface triangle areas; surface triangles are identified as such for which all nodes of a triangle are on the surface

**nodes*: requires that surface triangles have been already built during import of finite element mesh, or by calling VolumeToSurfaceElements!

- **input**:

  nodeList: list of local (Position) node numbers

  normalizeWeights: if True, weights are normalized to sum(weights)==1; otherwise, returned list contains areas according to nodes per

- **output**: numpy array with weights according to indices in node list

---

def **GetSurfaceTriangles** (*self*)

- **classFunction**: return surface trigs as node number list (for drawing in EXUDYN and for node weights)

---

def **VolumeToSurfaceElements** (*self*, *verbose*= False)

- **classFunction**:

  generate surface elements from volume elements

  stores the surface in self.surface

  only works for one element list and only for element types 'Hex8', 'Hex20', 'Tet4' and 'Tet10'

---

def **GetGyroscopicMatrix** (*self*, *rotationAxis*= 2, *sparse*= True)

- **classFunction**: get gyroscopic matrix in according format; rotationAxis=[0,1,2] = [x,y,z]

---

def **ScaleMassMatrix** (*self*, *factor*)

- **classFunction**: scale (=multiply) mass matrix with factor

---

def **ScaleStiffnessMatrix** (*self*, *factor*)

- **classFunction**: scale (=multiply) stiffness matrix with factor

---

def **AddElasticSupportAtNode** (*self*, *nodeNumber*, *springStiffness*= [1e8,1e8,1e8])

- **classFunction**:

  modify stiffness matrix to add elastic support (joint, etc.) to a node; nodeNumber zero based (as everywhere in the code...)

  springStiffness must have length according to the node size

---

def **AddNodeMass** (*self*, *nodeNumber*, *addedMass*)

- **classFunction**: modify mass matrix by adding a mass to a certain node, modifying directly the mass matrix

---

def **CreateLinearFEMObjectGenericODE2** (*self*, *mbs*, *color*= [0.9,0.4,0.4,1.])

- **classFunction**: create GenericODE2 object out of (linear) FEM model; uses always the sparse matrix mode, independent of the solver settings; this model can be directly used inside the multibody system as a static or dynamic FEM subsystem undergoing small deformations; computation is several magnitudes slower than ObjectFFRFreducedOrder
- **input**: mbs: multibody system to which the GenericODE2 is added
- **output**: return list [oGenericODE2, nodeList] containing object number of GenericODE2 as well as the list of mbs node numbers of all NodePoint nodes

---

def **CreateNonlinearFEMObjectGenericODE2NGsolve** (*self*, *mbs*, *mesh*, *density*, *youngsModulus*, *poissonsRatio*, *meshOrder*= 1, *color*= [0.9,0.4,0.4,1.])

- **classFunction**: create GenericODE2 object fully nonlinear FEM model using NGsolve; uses always the sparse matrix mode, independent of the solver settings; this model can be directly used inside the multibody system as a static or dynamic nonlinear FEM subsystem undergoing large deformations; computation is several magnitudes slower than ObjectFFRFreducedOrder
- **input**:

  mbs: multibody system to which the GenericODE2 is added

  mesh: a previously created `ngs.mesh` (NGsolve mesh, see examples)

  youngsModulus: Young's modulus used for mechanical model

  poissonsRatio: Poisson's ratio used for mechanical model

  density: density used for mechanical model

  meshOrder: use 1 for linear elements and 2 for second order elements (recommended to use 2 for much higher accuracy!)
- **output**: return list [oGenericODE2, nodeList] containing object number of GenericODE2 as well as the list of mbs node numbers of all NodePoint nodes
- **author**: Johannes Gerstmayr, Joachim Schöberl
- **notes**:

  The interface to NETGEN/NGsolve has been created together with Joachim Schöberl, main developer

  of NETGEN/NGsolve [51, 52]; Thank's a lot!

  download NGsolve at: https://ngsolve.org/

  NGsolve needs Python 3.7 (64bit) ==> use according EXUDYN version!

  note that node/element indices in the NGsolve mesh are 1-based and need to be converted to 0-base!

---

def **ComputeEigenmodes** (*self*, *nModes*, *excludeRigidBodyModes*= 0, *useSparseSolver*= True)

- **classFunction**: compute nModes smallest eigenvalues and eigenmodes from mass and stiffness-Matrix; store mode vectors in modeBasis, but exclude a number of 'excludeRigidBodyModes' rigid body modes from modeBasis; uses scipy for solution of generalized eigenvalue problem
- **input**:

nModes: prescribe the number of modes to be computed; total computed modes are (nModes+excludeRigid
but only nModes with smallest absolute eigenvalues are considered and stored

excludeRigidBodyModes: if rigid body modes are expected (in case of free-free modes), then this number specifies the number of eigenmodes to be excluded in the stored basis (usually 6 modes in 3D)

useSparseSolver: for larger systems, the sparse solver needs to be used, which iteratively solves the problem and uses a random number generator (internally in ARPACK): therefore, results are not fully repeatable!!!

- **output**: eigenmodes are stored internally in FEMinterface as 'modeBasis' and eigenvalues as 'eigenValues'
- **notes**: for NGsolve / Netgen meshes, see the according ComputeEigenmodesNGsolve function, which is usually much faster

---

def **ComputeEigenModesWithBoundaryNodes** (*self*, *boundaryNodes*, *nEigenModes*, *useSparseSolver*= True)

- **classFunction**: compute eigenmodes, using a set of boundary nodes that are all fixed; very similar to ComputeEigenmodes, but with additional definition of (fixed) boundary nodes.
- **input**:

  boundaryNodes: a list of boundary node indices, refering to 'Position' type nodes in FEMinterface; all coordinates of these nodes are fixed for the computation of the modes

  nEigenModes: prescribe the number of modes to be computed; only nEigenModes with smallest abs(eigenvalues) are considered and stored

  useSparseSolver: [yet NOT IMPLEMENTED] for larger systems, the sparse solver needs to be used, which iteratively solves the problem and uses a random number generator (internally in ARPACK): therefore, results are not fully repeatable!!!

- **output**: eigenmodes are stored internally in FEMinterface as 'modeBasis' and eigenvalues as 'eigenValues'

---

def **ComputeHurtyCraigBamptonModes** (*self*, *boundaryNodesList*, *nEigenModes*, *useSparseSolver*= True, *computationMode*= HCBstaticModeSelection.RBE2, *boundaryNodesWeights*= [], *excludeRigidBodyMotion*= True, *RBE3secondMomentOfAreaWeighting*= True, *verboseMode*= False, *timerTreshold*= 20000)

- **classFunction**: compute static and eigen modes based on Hurty-Craig-Bampton, for details see theory part Section 5.5. Note that this function may need significant time, depending on your hardware, but 50.000 nodes will require approx. 1-2 minutes and more nodes typically raise time more than linearly.

- **input**:

  boundaryNodesList: [nodeList0, nodeList1, ...] a list of node lists, each of them representing a set of 'Position' nodes for which a rigid body interface (displacement/rotation and force/torque) is created; NOTE THAT boundary nodes may not overlap between the different node lists (no duplicated node indices!)

  nEigenModes: number of eigen modes in addition to static modes (may be zero for RBE2/RBE3 computationMode); eigen modes are computed for the case where all rigid body motions at boundaries are fixed; only smallest nEigenModes absolute eigenvalues are considered

  useSparseSolver: for more than approx. 500 nodes, it is recommended to use the sparse solver; dense mode not available for RBE3

  computationMode: see class HCBstaticModeSelection for available modes; select RBE2 / RBE3 as standard, which is both efficient and accurate and which uses rigid-body-interfaces (6 independent modes) per boundary; RBE3 mode uses singular value decomposition, which requires full matrices for boundary nodes; this becomes slow in particular if the number of a single boundary node set gets larger than 500 nodes

  boundaryNodesWeights: according list of weights with same order as boundaryNodesList, as returned e.g. by FEMinterface.GetNodeWeightsFromSurfaceAreas(...)

  excludeRigidBodyMotion: if True (recommended), the first set of boundary modes is eliminated, which defines the reference conditions for the FFRF object

  RBE3secondMomentOfAreaWeighting: if True, the weighting of RBE3 boundaries is done according to second moment of area; if False, the more conventional (but less appropriate) quadratic distance to reference point weighting is used

  verboseMode: if True, some additional output is printed

  timerTreshold: for more DOF than this number, CPU times are printed even with verboseMode=False

- **output**: stores computed modes in self.modeBasis and abs(eigenvalues) in self.eigenValues

- **notes**: for NGsolve / Netgen meshes, see the according ComputeHurtyCraigBamptonModesNGsolve function, which is usually much faster - currently only implemented for RBE2 case

---

def **GetEigenFrequenciesHz** (*self*)

- **classFunction**: return list of eigenvalues in Hz of previously computed eigenmodes

---

def **ComputePostProcessingModes** (*self*, *material*= 0, *outputVariableType*= 'OutputVariableType.StressLocal', *numberOfThreads*= 1)

- **classFunction**: compute special stress or strain modes in order to enable visualization of stresses and strains in ObjectFFRFreducedOrder;
- **input**:

  material: specify material properties for computation of stresses, using a material class, e.g. material = KirchhoffMaterial(Emodulus, nu, rho); not needed for strains

  outputVariableType: specify either exudyn.OutputVariableType.StressLocal or exudyn.OutputVariableType as the desired output variables

  numberOfThreads: if numberOfThreads=1, it uses single threaded computation; if numberOfThreads>1, it uses the multiprocessing pools functionality, which requires that all code in your main file must be encapsulated within an if clause "if __name__ == '__main__':", see examples; if numberOfThreads==-1, it uses all threads/CPUs available
- **output**: post processing modes are stored in FEMinterface in local variable postProcessingModes as a dictionary, where 'matrix' represents the modes and 'outputVariableType' stores the type of mode as a OutputVariableType
- **notes**: This function is implemented in Python and rather slow for larger meshes; for NGsolve / Netgen meshes, see the according ComputePostProcessingModesNGsolve function, which is usually much faster

---

def **ComputeCampbellDiagram** (*self*, *terminalFrequency*, *nEigenfrequencies*= 10, *frequencySteps*= 25, *rotationAxis*= 2, *plotDiagram*= False, *verbose*= False, *useCorotationalFrame*= False, *useSparseSolver*= False)

- **classFunction**:

  compute Campbell diagram for given mechanical system

  create a first order system Axd + Bx = 0 with x= [q,qd]' and compute eigenvalues

  takes mass M, stiffness K and gyroscopic matrix G from FEMinterface

  currently only uses dense matrices, so it is limited to approx. 5000 unknowns!
- **input**:

terminalFrequency: frequency in Hz, up to which the campbell diagram is computed

nEigenfrequencies: gives the number of computed eigenfrequencies(modes), in addition to the rigid body mode 0

frequencySteps: gives the number of increments (gives frequencySteps+1 total points in campbell diagram)

rotationAxis:[0,1,2] = [x,y,z] provides rotation axis

plotDiagram: if True, plots diagram for nEigenfrequencies befor terminating

verbose: if True, shows progress of computation; if verbose=2, prints also eigenfrequencies

useCorotationalFrame: if False, the classic rotor dynamics formulation for rotationally-symmetric rotors is used, where the rotor can be understood in a Lagrangian-Eulerian manner: the rotation is represented by an additional (Eulerian) velocity in rotation direction; if True, the corotational frame is used, which gives a factor 2 in the gyroscopic matrix and can be used for non-symmetric rotors as well

useSparseSolver: for larger systems, the sparse solver needs to be used for creation of system matrices and for the eigenvalue solver (uses a random number generator internally in ARPACK, therefore, results are not fully repeatable!!!)

– **output**:

[listFrequencies, campbellFrequencies]

listFrequencies: list of computed frequencies

campbellFrequencies: array of campbell frequencies per eigenfrequency of system

---

def **CheckConsistency** (*self*)

– **classFunction**: perform some consistency checks

---

def **ReadMassMatrixFromAnsys** (*self*, *fileName*, *dofMappingVectorFile*, *sparse*= True, *verbose*= False)

– **classFunction**: read mass matrix from CSV format (exported from Ansys)

---

def **ReadStiffnessMatrixFromAnsys** (*self*, *fileName*, *dofMappingVectorFile*, *sparse*= True, *verbose*= False)

– **classFunction**: read stiffness matrix from CSV format (exported from Ansys)

---

def **ReadNodalCoordinatesFromAnsys** (*self*, *fileName*, *verbose*= False)

– **classFunction**: read nodal coordinates (exported from Ansys as .txt-File)

---

def **ReadElementsFromAnsys** (*self*, *fileName*, *verbose*= False)

– **classFunction**: read elements (exported from Ansys as .txt-File)

For examples on FEMinterface see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `CMSexampleCourse.py` (Ex), `netgenSTLtest.py` (Ex), `NGsolveCMStutorial.py` (Ex), `NGsolveCraigBampton.py` (Ex), `NGsolveFFRF.py` (Ex), ... , `abaqusImportTest.py` (TM), `ACFtest.py` (TM), `compareAbaqusAnsysRotorEigenfrequencies.py` (TM), ...

7.8 Module: graphics

This module newly introduces revised graphics functions, coherent with Exudyn terminology; it provides basic graphics elements like cuboid, cylinder, sphere, solid of revolution, etc.; offers also some advanced functions for STL import and mesh manipulation; for some advanced functions see graphicsDataUtilties; GraphicsData helper functions generate dictionaries which contain line, text or triangle primitives for drawing in Exudyn using OpenGL.

Author: Johannes Gerstmayr

Date: 2024-05-10 (created)

def **Sphere** (*point*= [0,0,0], *radius*= 0.1, *color*= [0.,0.,0.,1.], *nTiles*= 8, *addEdges*= False, *edgeColor*= color.black, *addFaces*= True)

– **function description**: generate graphics data for a sphere with point p and radius

– **input**:

point: center of sphere (3D list or np.array)

radius: positive value

color: provided as list of 4 RGBA values

nTiles: used to determine resolution of sphere >=3; use larger values for finer resolution

addEdges: True or number of edges along sphere shell (under development); for optimal drawing, nTiles shall be multiple of 4 or 8

edgeColor: optional color for edges

addFaces: if False, no faces are added (only edges)

- **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects

For examples on Sphere see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- bicycleIftommBenchmark.py (Ex), bungeeJump.py (Ex), chatGPTupdate.py (Ex), contactCurvePolynomial.py (Ex), graphicsDataExample.py (Ex), ... , connectorGravityTest.py (TM), contactCoordinateTest.py (TM), contactCurveExample.py (TM), ...

---

def **Lines** (*pList*, *color*= [0.,0.,0.,1.])

- **function description**: generate graphics data for lines, given by list of points and color; transforms to GraphicsData dictionary
- **input**:

  pList: list of 3D numpy arrays or lists (to achieve closed curve, set last point equal to first point)

  color: provided as list of 4 RGBA values
- **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects
- **example**:

```
#create simple 3-point lines
gLine=graphics.Lines([[0,0,0],[1,0,0],[2,0.5,0]], color=color.red)
```

For examples on Lines see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ANCFcontactCircle2.py (Ex), doublePendulum2D.py (Ex), simple4linkPendulumBing.py (Ex), doublePendulum2DControl.py (TM)

---

def **Circle** (*point*= [0,0,0], *radius*= 1, *color*= [0.,0.,0.,1.])

- **function description**: generate graphics data for a single circle; currently the plane normal = [0,0,1], just allowing to draw planar circles – this may be extended in future!
- **input**:

  point: center point of circle

radius: radius of circle

color: provided as list of 4 RGBA values

- **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects
- **notes**: the tiling (number of segments to draw circle) can be adjusted by visualizationSettings.general.circleTiling

For examples on Circle see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ANCFcontactCircle2.py (Ex)

---

def **Text** (*point*= [0,0,0], *text*= '', *color*= [0.,0.,0.,1.])

- **function description**: generate graphics data for a text drawn at a 3D position
- **input**:

  point: position of text

  text: string representing text

  color: provided as list of 4 RGBA values

  ***nodes*: text size can be adjusted with visualizationSettings.general.textSize, which affects the text size (=font size) globally
- **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects

For examples on Text see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ANCFcontactCircle2.py (Ex), NGsolveGeometry.py (Ex)

---

def **Cuboid** (*pList*, *color*= [0.,0.,0.,1.], *faces*= [1,1,1,1,1,1], *addNormals*= False, *addEdges*= False, *edgeColor*= color.black, *addFaces*= True)

- **function description**: generate graphics data for general block with endpoints, according to given vertex definition
- **input**:

  pList: is a list of points [[x0,y0,z0],[x1,y1,z1],...]

  color: provided as list of 4 RGBA values

  faces: includes the list of six binary values (0/1), denoting active faces (value=1); set index to zero to hide face

addNormals: if True, normals are added and there are separate points for every triangle

addEdges: if True, edges are added in TriangleList of GraphicsData

edgeColor: optional color for edges

addFaces: if False, no faces are added (only edges)

- **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects

---

def **BrickXYZ** (*xMin*, *yMin*, *zMin*, *xMax*, *yMax*, *zMax*, *color*= [0.,0.,0.,1.], *addNormals*= False, *addEdges*= False, *edgeColor*= color.black, *addFaces*= True)

- **function description**: generate graphics data for orthogonal 3D block with min and max dimensions
- **input**:

  x/y/z/Min/Max: minimal and maximal cartesian coordinates for orthogonal cube

  color: list of 4 RGBA values

  addNormals: add face normals to triangle information

  addEdges: if True, edges are added in TriangleList of GraphicsData

  edgeColor: optional color for edges

  addFaces: if False, no faces are added (only edges)
- **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects
- **notes**: DEPRECATED

For examples on BrickXYZ see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `geneticOptimizationSliderCrank.py` (Ex), `massSpringFrictionInteractive.py` (Ex), `mouseInteractionExample.py` (Ex), `performanceMultiThreadingNG.py` (Ex), `rigidBodyIMUtest.py` (Ex), ... , `driveTrainTest.py` (TM), `explicitLieGroupIntegratorPythonTest.py` (TM), `explicitLieGroupIntegratorTest.py` (TM), ...

---

def **Brick** (*centerPoint*= [0,0,0], *size*= [0.1,0.1,0.1], *color*= [0.,0.,0.,1.], *addNormals*= False, *addEdges*= False, *edgeColor*= color.black, *addFaces*= True, *roundness*= 0, *nTiles*= 12)

- **function description**: generate graphics data for orthogonal 3D box with center point and size; using roundness=1, it draws an ellipsoid inside the box and in case 0 < roundness < 1, it draws a body blended between box and ellipsoid

- **input**:

  centerPoint: center of box as 3D list or np.array

  size: size as 3D list or np.array

  color: list of 4 RGBA values

  addNormals: add face normals to triangle information

  addEdges: if True, edges are added in TriangleList of GraphicsData

  edgeColor: optional color for edges

  addFaces: if False, no faces are added (only edges)

  roundness: if > 0, it draws an ellipsoid, using nTiles for drawing; edges are not available if roundness > 0

  nTiles: only apply if roundness > 0; discretization of whole ellipsoid; should be multiple of 4 to avoid artifacts

- **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects; if addEdges=True, it returns a list of two dictionaries

For examples on Brick see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- addPrismaticJoint.py (Ex), addRevoluteJoint.py (Ex), ANCFrotatingCable2D.py (Ex), beltDrivesComparison.py (Ex), bicycleIftommBenchmark.py (Ex), ... , bricardMechanism.py (TM), carRollingDiscTest.py (TM), complexEigenvaluesTest.py (TM), ...

---

def **Cylinder** (*pAxis*= [0,0,0], *vAxis*= [0,0,1], *radius*= 0.1, *color*= [0.,0.,0.,1.], *nTiles*= 16, *radiusInner*= None, *angleRange*= [0,2*pi], *lastFace*= True, *cutPlain*= True, *addEdges*= False, *edgeColor*= color.black, *addFaces*= True, ***kwargs*)

- **function description**: generate graphics data for a cylinder with given axis, radius and color; nTiles gives the number of tiles (minimum=3)

- **input**:

  pAxis: axis point of one face of cylinder (3D list or np.array)

  vAxis: vector representing the cylinder's axis (3D list or np.array)

  radius: positive value representing radius of cylinder

  color: provided as list of 4 RGBA values

  nTiles: used to determine resolution of cylinder >=3; use larger values for finer resolution

  radiusInner: if not equal 0, this represents the inner radius of a hollow cylinder; some options like angleRange, lastFace, etc. do not work in this case

angleRange: given in rad, to draw only part of cylinder (halfcylinder, etc.); for full range use [0..2 * pi]

lastFace: if angleRange != [0,2*pi], then the faces of the open cylinder are shown with lastFace = True

cutPlain: only used for angleRange != [0,2*pi]; if True, a plane is cut through the part of the cylinder; if False, the cylinder becomes a cake shape ...

addEdges: if True, edges are added in TriangleList of GraphicsData; if addEdges is integer, additional int(addEdges) lines are added on the cylinder mantle

edgeColor: optional color for edges

addFaces: if False, no faces are added (only edges)

alternatingColor: if given, optionally another color in order to see rotation of solid; only works, if angleRange=[0,2*pi]

- **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects

For examples on Cylinder see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `ballBearningModel.py` (Ex), `beltDriveALE.py` (Ex), `beltDriveReevingSystem.py` (Ex), `beltDrivesComparison.py` (Ex), `bicycleIftommBenchmark.py` (Ex), ... , `ANCFbeltDrive.py` (TM), `ANCFgeneralContactCircle.py` (TM), `coordinateSpringDamperExt.py` (TM), ...

---

def **Tube** (*points*, *axes*, *radius*= 0.1, *color*= [0.,0.,0.,1.], *nTiles*= 16)

- **function description**: generate graphics data for a tube with given list of points and axes, radius and color; nTiles gives the number of tiles (minimum=3)
- **input**:

  points: list of 3D vectors (or numpy arrays) representing the center points of the tube line

  axes: list of 3D vectors (or numpy arrays) representing the axis according to the points

  radius: positive value representing radius of tube

  color: provided as list of 4 RGBA values

  nTiles: used to determine resolution of cylinder >=3; use larger values for finer resolution
- **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects

---

def **Torus** (*point*, *axis*, *radiusMajor*= 0.5, *radiusMinor*= 0.1, *color*= [0., 0., 0., 1.], *nTilesMajor*= 24, *nTilesMinor*= 12, *minorAngleStart*= 0, *minorAngleEnd*= 2*np.pi, *smoothNormals*= True, *invert*= False)

- **function description**: generate graphics data for a torus with given major and minor radius, center point and axis
- **input**:
  point: 3D vector (or numpy array) representing the center point of the torus
  axis: 3D vector (or numpy array) representing the axis of revolution of the torus
  radiusMajor: major radius of torus
  radiusMinor: minor radius of torus
  color: provided as list of 4 RGBA values
  nTilesMajor: used to for resolution of tube with major radius; use larger values for finer resolution
  nTilesMinor: used to for resolution of circle with minor radius; use larger values for finer resolution
  minorAngleStart: starting angle for minor radius; 0 is the angle at outmost radius of torus, pi is at inside
  minorAngleEnd: end angle for minor radius; use -0.5*pi / 0.5*pi to draw only the outer half of the torus
  smoothNormals: if True, the normals are added to create a smooth contour, otherwise triangles are flat
  invert: if False, the outside faces are visible; if invert=True, the inside faces are visible (influences reflections, light, etc.)
- **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects

---

def **RigidLink** (*p0*, *p1*, *axis0*= [0,0,0], *axis1*= [0,0,0], *radius*= [0.1,0.1], *thickness*= 0.05, *width*= [0.05,0.05], *color*= [0.,0.,0.,1.], *nTiles*= 16)

- **function description**: generate graphics data for a planar Link between the two joint positions, having two axes
- **input**:
  p0: joint0 center position
  p1: joint1 center position
  axis0: direction of rotation axis at p0, if drawn as a cylinder; [0,0,0] otherwise
  axis1: direction of rotation axis of p1, if drawn as a cylinder; [0,0,0] otherwise
  radius: list of two radii [radius0, radius1], being the two radii of the joints drawn by a cylinder or sphere

width: list of two widths [width0, width1], being the two widths of the joints drawn by a cylinder; ignored for sphere

thickness: the thickness of the link (shaft) between the two joint positions; thickness in z-direction or diameter (cylinder)

color: provided as list of 4 RGBA values

nTiles: used to determine resolution of cylinder >=3; use larger values for finer resolution

– **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects

For examples on RigidLink see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `fourBarMechanism3D.py` (Ex), `geneticOptimizationSliderCrank.py` (Ex), `multiMbsTest.py` (Ex), `openVRengine.py` (Ex), `pistonEngine.py` (Ex), ... , `fourBarMechanismIftomm.py` (TM), `rollingDiscTangentialForces.py` (TM), `sliderCrank3Dbenchmark.py` (TM), ...

---

def **SolidOfRevolution** (*pAxis*, *vAxis*, *contour*, *color*= [0.,0.,0.,1.], *nTiles*= 16, *smoothContour*= False, *addEdges*= False, *edgeColor*= color.black, *addFaces*= True, *smoothingAngle*= 2*np.pi, ***kwargs*)

– **function description**: generate graphics data for a solid of revolution with given 3D point and axis, 2D point list for contour, (optional)2D normals and color;

– **input**:

pAxis: axis point of one face of solid of revolution (3D list or np.array)

vAxis: vector representing the solid of revolution's axis (3D list or np.array)

contour: a list of 2D-points, specifying the contour (x=axis, y=radius), e.g.: [[0,0],[0,0.1],[1,0.1]]

color: provided as list of 4 RGBA values

nTiles: used to determine resolution of solid; use larger values for finer resolution

smoothContour: if True, the contour is made smooth by auto-computing normals to the contour

addEdges: True or number of edges along revolution mantle; for optimal drawing, nTiles shall be multiple addEdges

edgeColor: optional color for edges

addFaces: if False, no faces are added (only edges)

smoothingAngle: if angle between two edges is smaller than smoothingAngle, smoothing is applied

alternatingColor: add a second color, which enables to see the rotation of the solid

– **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects

– **example**:

```
#simple contour, using list of 2D points:
contour=[[0,0.2],[0.3,0.2],[0.5,0.3],[0.7,0.4],[1,0.4],[1,0.]]
rev1 = graphics.SolidOfRevolution(pAxis=[0,0.5,0], vAxis=[1,0,0],
                                  contour=contour, color=color.red,
                                  alternatingColor=color.grey)
#draw torus:
contour=[]
r = 0.2 #small radius of torus
R = 0.5 #big radius of torus
nc = 16 #discretization of torus
for i in range(nc+3): #+3 in order to remove boundary effects
    contour+=[[r*cos(i/nc*pi*2),R+r*sin(i/nc*pi*2)]]
#use smoothContour to make torus looking smooth
rev2 = graphics.SolidOfRevolution(pAxis=[0,0.5,0], vAxis=[1,0,0],
                                  contour=contour, color=color.red,
                                  nTiles = 64, smoothContour=True)
```

For examples on SolidOfRevolution see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ballBearningModel.py (Ex), graphicsDataExample.py (Ex), gyroStability.py (Ex), particlesSilo.py (Ex), serialRobotKinematicTreeDigging.py (Ex), ... , ConvexContactTest.py (TM)

---

def **Arrow** (*pAxis*, *vAxis*, *radius*, *color*= [0.,0.,0.,1.], *headFactor*= 2, *headStretch*= 4, *nTiles*= 12)

- **function description**: generate graphics data for an arrow with given origin, axis, shaft radius, optional size factors for head and color; nTiles gives the number of tiles (minimum=3)
- **input**:

  pAxis: axis point of the origin (base) of the arrow (3D list or np.array)

  vAxis: vector representing the vector pointing from the origin to the tip (head) of the error (3D list or np.array)

  radius: positive value representing radius of shaft cylinder

  headFactor: positive value representing the ratio between head's radius and the shaft radius

  headStretch: positive value representing the ratio between the head's radius and the head's length

  color: provided as list of 4 RGBA values

  nTiles: used to determine resolution of arrow (of revolution object) >=3; use larger values for finer resolution

- **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects

For examples on Arrow see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- beltDriveALE.py (Ex), beltDriveReevingSystem.py (Ex), beltDrivesComparison.py (Ex), graphicsDataExample.py (Ex), reevingSystem.py (Ex), ... , ACFtest.py (TM), ANCFbeltDrive.py (TM), ANCFgeneralContactCircle.py (TM), ...

---

def **Basis** (*origin*= [0,0,0], *rotationMatrix*= np.eye(3), *length*= 1, *colors*= [color.red, color.green, color.blue], *headFactor*= 2, *headStretch*= 4, *nTiles*= 12, ***kwargs*)

- **function description**: generate graphics data for three arrows representing an orthogonal basis with point of origin, shaft radius, optional size factors for head and colors; nTiles gives the number of tiles (minimum=3)
- **input**:

  origin: point of the origin of the base (3D list or np.array)

  rotationMatrix: optional transformation, which rotates the basis vectors

  length: positive value representing lengths of arrows for basis

  colors: provided as list of 3 colors (list of 4 RGBA values)

  headFactor: positive value representing the ratio between head's radius and the shaft radius

  headStretch: positive value representing the ratio between the head's radius and the head's length

  nTiles: used to determine resolution of arrows of basis (of revolution object) >=3; use larger values for finer resolution

  radius: positive value representing radius of arrows; default: radius = 0.01*length
- **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects

For examples on Basis see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ballBearningModel.py (Ex), camFollowerExample.py (Ex), fourBarMechanism3D.py (Ex), graphicsDataExample.py (Ex), gyroStability.py (Ex), ... , bricardMechanism.py (TM), contactCurveExample.py (TM), createFunctionsTest.py (TM), ...

---

def **Frame** (*HT*= np.eye(4), *length*= 1, *colors*= [color.red, color.green, color.blue], *headFactor*= 2, *headStretch*= 4, *nTiles*= 12, ***kwargs*)

- **function description**: generate graphics data for frame (similar to Basis), showing three arrows representing an orthogonal basis for the homogeneous transformation HT; optional shaft radius, optional size factors for head and colors; nTiles gives the number of tiles (minimum=3)

- **input**:

  HT: homogeneous transformation representing frame

  length: positive value representing lengths of arrows for basis

  colors: provided as list of 3 colors (list of 4 RGBA values)

  headFactor: positive value representing the ratio between head's radius and the shaft radius

  headStretch: positive value representing the ratio between the head's radius and the head's length

  nTiles: used to determine resolution of arrows of basis (of revolution object) >=3; use larger values for finer resolution

  radius: positive value representing radius of arrows; default: radius = 0.01*length

- **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects

For examples on Frame see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- serialRobotInverseKinematics.py (Ex)

---

def **Quad** (*pList*, *color*= [0.,0.,0.,1.], ***kwargs*)

- **function description**:

  generate graphics data for simple quad with option for checkerboard pattern;

  points are arranged counter-clock-wise, e.g.: p0=[0,0,0], p1=[1,0,0], p2=[1,1,0], p3=[0,1,0]

- **input**:

  pList: list of 4 quad points [[x0,y0,z0],[x1,y1,z1],...]

  color: provided as list of 4 RGBA values

  alternatingColor: second color; if defined, a checkerboard pattern (default: 10x10) is drawn with color and alternatingColor

  nTiles: number of tiles for checkerboard pattern (default: 10)

  nTilesY: if defined, use number of tiles in y-direction different from x-direction (=nTiles)

- **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects

- **example**:

```
plane = graphics.Quad([[-8, 0, -8],[ 8, 0, -8,],[ 8, 0, 8],[-8, 0, 8]],
                          color.darkgrey, nTiles=8,
                          alternatingColor=color.lightgrey)
oGround=mbs.AddObject(ObjectGround(referencePosition=[0,0,0],
                        visualization=VObjectGround(graphicsData=[plane])))
```

For examples on Quad see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- massSpringFrictionInteractive.py (Ex), nMassOscillator.py (Ex), nMassOscillatorEigenmodes.py (Ex), nMassOscillatorInteractive.py (Ex), simulateInteractively.py (Ex), ... , sphereTriangleTest.py (TM)

---

def **CheckerBoard** (*point*= [0,0,0], *normal*= [0,0,1], *size*= 1, *color*= color.lightgrey, *alternatingColor*= color.lightgrey2, *nTiles*= 10, ***kwargs*)

- **function description**:

  function to generate checkerboard background;

  points are arranged counter-clock-wise, e.g.:

- **input**:

  point: midpoint of pattern provided as list or np.array

  normal: normal to plane provided as list or np.array

  size: dimension of first side length of quad

  size2: dimension of second side length of quad

  color: provided as list of 4 RGBA values

  alternatingColor: second color; if defined, a checkerboard pattern (default: 10x10) is drawn with color and alternatingColor

  nTiles: number of tiles for checkerboard pattern in first direction

  nTiles2: number of tiles for checkerboard pattern in second direction; default: nTiles

  materialIndex: use special graphics material for both colors

- **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects

- **example**:

```
plane = graphics.CheckerBoard(normal=[0,0,1], size=5)
oGround=mbs.AddObject(ObjectGround(referencePosition=[0,0,0],
                      visualization=VObjectGround(graphicsData=[plane])))
```

For examples on CheckerBoard see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ANCFrotatingCable2D.py (Ex), ballBearningModel.py (Ex), bicycleIftommBenchmark.py (Ex), camFollowerExample.py (Ex), chainDriveExample.py (Ex), ... , ANCFoutputTest.py (TM), bricardMechanism.py (TM), connectorGravityTest.py (TM), ...

---

def **SolidExtrusion** (*vertices*, *segments*, *height*, *rot*= np.diag([1,1,1]), *pOff*= [0,0,0], *color*= [0,0,0,1], *smoothNormals*= False, *addEdges*= False, *edgeColor*= color.black, *addFaces*= True)

- **function description**:

  create graphicsData for solid extrusion based on 2D points and segments; by default, the extrusion is performed in z-direction;

  additional transformations are possible to translate and rotate the extruded body;

- **input**:

  vertices: list of pairs of coordinates of vertices in mesh [x,y], see ComputeTriangularMesh(...)

  segments: list of segments, which are pairs of node numbers [i,j], defining the boundary of the mesh;

  the ordering of the nodes is such that left triangle = inside, right triangle = outside; see ComputeTriangularMesh(...)

  height: height of extruded object

  rot: rotation matrix, which the extruded object point coordinates are multiplied with before adding offset

  pOff: 3D offset vector added to extruded coordinates; the z-coordinate of the extrusion object obtains 0 for the base plane, z=height for the top plane

  smoothNormals: if True, algorithm tries to smoothen normals at vertices and normals are added; creates more points; if False, triangle normals are used internally

  addEdges: if True or 1, edges at bottom/top are included in the GraphicsData dictionary; if 2, also mantle edges are included

  edgeColor: optional color for edges

  addFaces: if False, no faces are added (only edges)

- **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects

For examples on SolidExtrusion see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- chainDriveExample.py (Ex), graphicsDataExample.py (Ex), simulatorCouplingTwoMbs.py (TM)

---

def **FromPointsAndTrigs** (*points*, *triangles*, *color*= [0.,0.,0.,1.], *normals*= None)

- **function description**: convert triangles and points as returned from graphics.ToPointsAndTrigs(...) to GraphicsData; additionally, normals and color(s) can be provided

- **input**:

  points: list or np.array with np rows of 3 columns (floats) per point (with np points)

  triangles: list or np.array with 3 int per triangle (0-based indices to triangles), giving a matrix with nt rows and 3 columns (with nt triangles)

color: provided as list of 4 RGBA values or single list of (np)*[4 RGBA values]

normals: if not None, they have to be provided per point (as matrix, list of lists or flattened) and will be added to returned GraphicsData

- **output**: returns GraphicsData with type TriangleList

For examples on FromPointsAndTrigs see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- NGsolveGeometry.py (Ex), NGsolveOCCgeometry.py (Ex), particlesSilo.py (Ex), distanceSensor.py (TM), generalContactFrictionTests.py (TM)

---

def **ToPointsAndTrigs** (*g*)

- **function description**: convert graphics data into list of points and list of triangle indices (triplets)
- **input**: g contains a GraphicsData with type TriangleList
- **output**: returns [points, triangles], with points as list of np.array with 3 floats per point and triangles as a list of np.array with 3 int per triangle (0-based indices to points)

For examples on ToPointsAndTrigs see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- mobileMecanumWheelRobotWithLidar.py (Ex), particleClusters.py (Ex), particlesSilo.py (Ex), reinforcementLearningRobot.py (Ex), serialRobotKinematicTreeDigging.py (Ex), ... , distanceSensor.py (TM), generalContactCylinderTest.py (TM), generalContactCylinderTrigsTest.py (TM), ...

---

def **Move** (*g*, *pOff*, *Aoff*= None)

- **function description**: add rigid body transformation to GraphicsData, using position offset (global) pOff (list or np.array) and rotation Aoff (transforms local to global coordinates; list of lists or np.array); see Aoff how to scale coordinates!
- **input**:

  g: graphicsData to be transformed

  pOff: 3D offset as list or numpy.array added to rotated points

  Aoff: 3D rotation matrix as list of lists or numpy.array with shape (3,3); if A is scaled by factor, e.g. using 0.001*np.eye(3), you can also scale the coordinates; if Aoff=None, no rotation is performed

- **output**: returns new graphcsData object to be used for drawing in objects
- **notes**: transformation corresponds to HomogeneousTransformation(Aoff, pOff), transforming original coordinates v into vNew = pOff + Aoff @ v

For examples on Move see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- graphicsDataExample.py (Ex), humanRobotInteraction.py (Ex), kinematicTreeAndMBS.py (Ex), openVRengine.py (Ex), pistonEngine.py (Ex), ... , rigidBodyAsUserFunctionTest.py (TM)

---

def **MergeTriangleLists** (*g1*, *g2*)

- **function description**: merge 2 different graphics data with triangle lists
- **input**: graphicsData dictionaries g1 and g2 obtained from GraphicsData functions
- **output**: one graphicsData dictionary with single triangle lists and compatible points and normals, to be used in visualization of EXUDYN objects; edges are merged; edgeColor is taken from graphicsData g1

For examples on MergeTriangleLists see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- graphicsDataExample.py (Ex), mobileMecanumWheelRobotWithLidar.py (Ex), particleClusters.py (Ex), particlesSilo.py (Ex), serialRobotKinematicTreeDigging.py (Ex), ... , distanceSensor.py (TM), generalContactFrictionTests.py (TM), laserScannerTest.py (TM), ...

---

def **InvertTriangles** (*graphicsData*, *invertTriangles*= True, *invertVertexNormals*= True)

- **function description**: invert triangle orientation and triangle normals (or only one of these tasks); can also check consistency of normals
- **input**:

  graphicsData: graphicsData as returned e.g. from graphics.Sphere

  invertTriangles: if True, it inverts the triangle orientation (changing vertex index 0 and 1)

  invertVertexNormals: if True, the direction of normal is flipped
- **output**: returns new graphicsData (copy) with modified triangles and normals

---

def **InconsistentTriangles** (*graphicsData*)

- **function description**: check consistency of orientation of triangles and vertex (point) normals
- **input**: graphicsData: graphicsData as returned e.g. from graphics.Sphere
- **output**: returns number of cases in which triangle normals and vertex normals are inconsistent (scalar product is negative)

For examples on InconsistentTriangles see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- graphicsDataExample.py (Ex)

---

def **NGsolveMesh2PointsAndTrigs** (*mesh*= None, *ngMesh*= None, *meshOrder*= 2, *scale*= 1, *addNormals*= True, *verbose*= False)

- **function description**: convert NGsolve (surface) mesh into (surface) points and triangles; clearly, it requires to have ngsolve installed
- **input**:
  - *mesh*: a ngsolve mesh; having a geometry geo = OCCGeometry(...), mesh is returned from ngsolve.Mesh(geo.GenerateMesh(...))
  - *ngMesh*: a netgen mesh; having a geometry geo = OCCGeometry(...), ngMesh is returned from geo.GenerateMesh(...)
  - *meshOrder*: either 1 (linear, flat triangles) or 2 (quadratic, smooth triangles)
  - *scale*: additional scaling factor for geometry, as it is recommended to define netgen geometries in mm due to tolerances
  - *addNormals*: if True, it computes and adds normals
  - *verbose*: print debug information
- **output**: [points, triangles] or if addNormals=True, [points, triangles, normals] for further usage in graphics.FromPointsAndTrigs(...)
- **example**:

```
#assume having already a body of netgen OCCGeometry
geo = OCCGeometry(body)
ngMesh = geo.GenerateMesh(maxh=maxh)
#convert mesh into points, triangles and normals (with second-order elements!)
[points, triangles, normals] = graphics.NGsolveMesh2PointsAndTrigs(mesh=ngMesh)
#convert into graphicsData
gMesh = graphics.FromPointsAndTrigs( points, triangles, normals=normals,
                                     color=graphics.color.red)
#use the mesh on a ground object
mbs.CreateGround(graphicsDataList=[gMesh])
```

For examples on NGsolveMesh2PointsAndTrigs see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- NGsolveOCCgeometry.py (Ex), NGsolvePistonEngine.py (Ex)

---

def **FromSTLfileASCII** (*fileName*, *color*= [0.,0.,0.,1.], *verbose*= False, *invertNormals*= True, *invertTriangles*= True)

- **function description**: generate graphics data from STL file (text format!) and use color for visualization; this function is slow, use stl binary files with FromSTLfile(...)
- **input**:

  fileName: string containing directory and filename of STL-file (in text / SCII format) to load

  color: provided as list of 4 RGBA values

  verbose: if True, useful information is provided during reading

  invertNormals: if True, orientation of normals (usually pointing inwards in STL mesh) are inverted for compatibility in Exudyn

  invertTriangles: if True, triangle orientation (based on local indices) is inverted for compatibility in Exudyn
- **output**: creates graphicsData, inverting the STL graphics regarding normals and triangle orientations (interchanged 2nd and 3rd component of triangle index)

For examples on FromSTLfileASCII see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- stlFileImport.py (Ex)

---

def **FromSTLfile** (*fileName*, *color*= [0.,0.,0.,1.], *verbose*= False, *density*= 0., *scale*= 1., *Aoff*= [], *pOff*= [], *invertNormals*= True, *invertTriangles*= True)

- **function description**: generate graphics data from STL file, allowing text or binary format; requires numpy-stl to be installed; additionally can scale, rotate and translate
- **input**:

  fileName: string containing directory and filename of STL-file (in text / SCII format) to load

  color: provided as list of 4 RGBA values

verbose: if True, useful information is provided during reading

density: if given and if verbose, mass, volume, inertia, etc. are computed

scale: point coordinates are transformed by scaling factor

invertNormals: if True, orientation of normals (usually pointing inwards in STL mesh) are inverted for compatibility in Exudyn

invertTriangles: if True, triangle orientation (based on local indices) is inverted for compatibility in Exudyn

- **output**: creates graphicsData, inverting the STL graphics regarding normals and triangle orientations (interchanged 2nd and 3rd component of triangle index)
- **notes**: the model is first scaled, then rotated, then the offset pOff is added; finally min, max, mass, volume, inertia, com are computed!

For examples on FromSTLfile see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- humanRobotInteraction.py (Ex), ROSTurtle.py (Ex), stlFileImport.py (Ex)

---

def **AddEdgesAndSmoothenNormals** (*graphicsData*, *edgeColor*= color.black, *edgeAngle*= 0.25*pi, *pointTolerance*= 5, *addEdges*= True, *smoothNormals*= True, *roundDigits*= 5, *triangleColor*= [])

- **function description**:

  compute and return GraphicsData with edges and smoothend normals for mesh consisting of points and triangles (e.g., as returned from GraphicsData2PointsAndTrigs)

  graphicsData: single GraphicsData object of type TriangleList; existing edges are ignored

  edgeColor: optional color for edges

  edgeAngle: angle above which edges are added to geometry

  roundDigits: number of digits, relative to max dimensions of object, at which points are assumed to be equal

  smoothNormals: if True, algorithm tries to smoothen normals at vertices; otherwise, uses triangle normals

  addEdges: if True, edges are added in TriangleList of GraphicsData

  triangleColor: if triangleColor is set to a RGBA color, this color is used for the new triangle mesh throughout

- **output**: returns GraphicsData with added edges and smoothed normals

– **notes**: this function is suitable for STL import; it assumes that all colors in graphicsData are the same and only takes the first color!

For examples on AddEdgesAndSmoothenNormals see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- humanRobotInteraction.py (Ex), NGsolveGeometry.py (Ex), stlFileImport.py (Ex)

---

def **ExportSTL** (*graphicsData*, *fileName*, *solidName*= 'ExudynSolid', *invertNormals*= True, *invertTriangles*= True)

– **function description**: export given graphics data (only type TriangleList allowed!) to STL ascii file using fileName

– **input**:

graphicsData: a single GraphicsData dictionary with type='TriangleList', no list of GraphicsData

fileName: file name including (local) path to export STL file

solidName: optional name used in STL file

invertNormals: if True, orientation of normals (usually pointing inwards in STL mesh) are inverted for compatibility in Exudyn

invertTriangles: if True, triangle orientation (based on local indices) is inverted for compatibility in Exudyn

For examples on ExportSTL see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- stlFileImport.py (Ex)

7.9 Module: graphicsDataUtilities

Utility functions for visualization, which provides functions for special graphics manipulation, colors, mesh manipulation, etc.; note that specific function for GraphicsData creation now moved into the graphics submodule; includes functionality like mesh manipulation and some helper functions

Author: Johannes Gerstmayr

Date: 2020-07-26 (created) Modified: 2024-05-10 (moved primitive functions to graphics)

Notes: Some useful colors are defined, using RGBA (Red, Green, Blue and Alpha = opacity) channels in the range [0,1], e.g., red = [1,0,0,1].

Available colors are: color4red, color4green, color4blue, color4cyan, color4magenta, color4yellow, color4orange, color4pink, color4lawngreen, color4violet, color4springgreen, color4dodgerblue, color4grey, color4darkgrey, color4lightgrey, color4lightred, color4lightgreen, color4steelblue, color4brown, color4black, color4darkgrey2, color4lightgrey2, color4white

Additionally, a list of 16 colors 'color4list' is available, which is intended to be used, e.g., for creating n bodies with different colors

def **SwitchTripletOrder** (*vector*)

- **function description**: helper function to switch order of three items in a list; mostly used for reverting normals in triangles
- **input**: 3D vector as list or as np.array
- **output**: interchanged 2nd and 3rd component of list

---

def **ComputeTriangleNormal** (*p0*, *p1*, *p2*)

- **function description**: compute normalized normal for 3 triangle points
- **input**: 3D vector as list or as np.array
- **output**: normal as np.array

---

def **ComputeTriangleArea** (*p0*, *p1*, *p2*)

- **function description**: compute area of triangle given by 3 points
- **input**: 3D vector as list or as np.array
- **output**: area as float

---

def **Compute6NodeTrigsNormals** (*elementNodes*)

- **function description**:

  Internal function: compute normals to 6-node triangular surface given by elementNodes

input: elementNodes given as np.array with 6 node vectors in rows; node ordering must follow Netgen order, see the local coordinates in the function

- **output**: returns np.array with 6 normals in rows

---

def **RefineMesh** (*points*, *triangles*)

- **function description**: refine triangle mesh; every triangle is subdivided into 4 triangles
- **input**:

  points: list of np.array with 3 floats per point

  triangles: list of np.array with 3 int per triangle (0-based indices to triangles)
- **output**: returns [points2, triangles2] containing the refined mesh; if the original mesh is consistent, no points are duplicated; if the mesh is not consistent, some mesh points are duplicated!
- **notes**: becomes slow for meshes with more than 5000 points

For examples on RefineMesh see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `particleClusters.py` (Ex), `particlesSilo.py` (Ex), `tippeTop.py` (Ex), `distanceSensor.py` (TM), `generalContactCylinderTest.py` (TM), `generalContactFrictionTests.py` (TM), `generalContactImplicit1.py` (TM), `generalContactImplicit2.py` (TM), ...

---

def **ShrinkMeshNormalToSurface** (*points*, *triangles*, *distance*)

- **function description**: shrink mesh using triangle normals; every point is at least moved a distance 'distance' normal from boundary
- **input**:

  points: list of np.array with 3 floats per point

  triangles: list of np.array with 3 int per triangle (0-based indices to triangles)

  distance: float value of minimum distance
- **output**: returns [points2, triangles2] containing the refined mesh; currently the points of the subdivided triangles are duplicated!
- **notes**: ONLY works for consistent meshes (no duplicated points!)

For examples on ShrinkMeshNormalToSurface see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- generalContactFrictionTests.py (TM)

---

def **ComputeTriangularMesh** (*vertices*, *segments*)

- **function description**:

  helper function to compute triangular mesh from list of vertices (=points) and segments;

  computes triangular meshes for non-convex case. In order to make it efficient, it first computes neighbors and then defines triangles at segments to be inside/outside. Finally neighboring relations are used to define all triangles inside/outside

  finally only returns triangles that are inside the segments

- **input**:

  vertices: list of pairs of coordinates of vertices in mesh [x,y]

  segments: list of segments, which are pairs of node numbers [i,j], defining the boundary of the mesh;

  the ordering of the nodes is such that left triangle = inside, right triangle = outside, compare example with segment [V1,V2]:

  inside

  V1 V2

  O————-O

  outside

- **output**: triangulation structure of Delaunay(...), see scipy.spatial.Delaunaystructure, containing all simplices (=triangles)

- **notes**: Delauney will not work if points are duplicated; you must first create point lists without duplicated points!

- **example**:

```
points = np.array([[0, 0], [0, 2], [2, 2], [2, 1], [1, 1], [0, 1], [1, 0]])
segments = [len(points)-1,0]
for i in range(len(points)-1):
    segments += [i,i+1]
tri = ComputeTriangularMesh(points, segments)
exudyn.Print(tri.simplices)
```

---

def **SegmentsFromPoints** (*points*, *pointIndexOffset*= 0, *invert*= False, *closeCurve*= True)

- **function description**: convert point list into segments (indices to points); point indices start with pointIndexOffset
- **input**:

  invert: True: circle defines outter boundary; False: circle cuts out geometry inside a geometry

  pointIndexOffset: point indices start with pointIndexOffset
- **output**: return segments, containing list of lists of point indices for segments

---

def **CirclePointsAndSegments** (*center*= [0,0], *radius*= 0.1, *invert*= False, *pointIndexOffset*= 0, *nTiles*= 16)

- **function description**: create points and segments, used in SolidExtrusion(...) for circle with given parameters
- **input**:

  center: 2D center point (list/numpy array) for circle center

  radius: radius of circle

  invert: True: circle defines outter boundary; False: circle cuts out geometry inside a geometry

  pointIndexOffset: point indices start with pointIndexOffset

  nTiles: number of tiles/segments for circle creation (higher is finer)
- **output**: return [points, segments], both containing lists of lists
- **notes**: geometries may not intersect!

---

def **GraphicsDataRectangle** (*xMin*, *yMin*, *xMax*, *yMax*, *color*= [0.,0.,0.,1.])

- **function description**: generate graphics data for 2D rectangle
- **input**: minimal and maximal cartesian coordinates in (x/y) plane; color provided as list of 4 RGBA values
- **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects

– **notes**: DEPRECATED

For examples on GraphicsDataRectangle see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ANCFcontactCircle2.py (Ex), ANCFswitchingSlidingJoint2D.py (Ex), lavalRotor2Dtest.py (Ex), particleClusters.py (Ex), particlesTest.py (Ex), ... , ANCFcontactFrictionTest.py (TM), ANCFmovingRigidBodyTest.py (TM), ANCFslidingAndALEjointTest.py (TM), ...

---

def **GraphicsDataOrthoCubeLines** (*xMin*, *yMin*, *zMin*, *xMax*, *yMax*, *zMax*, *color*= [0.,0.,0.,1.])

– **function description**: generate graphics data for orthogonal block drawn with lines

– **input**: minimal and maximal cartesian coordinates for orthogonal cube; color provided as list of 4 RGBA values

– **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects

– **notes**: DEPRECATED

For examples on GraphicsDataOrthoCubeLines see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- rigid3Dexample.py (Ex), genericJointUserFunctionTest.py (TM), rigidBodyCOMtest.py (TM), sphericalJointTest.py (TM)

7.10 Module: GUI

Helper functions and classes for graphical interaction with Exudyn

Author: Johannes Gerstmayr

Date: 2020-01-25

Notes: This is an internal library, which is only used inside Exudyn for modifying settings.

def **GetTkRootAndNewWindow** ()

– **function description**: get new or current root and new window app; return list of [tkRoot, tkWindow, tkRuns]

---

def **TkRootExists** ()

- **function description**: this function returns True, if tkinter has already a root window (which is assumed to have already a mainloop running)

---

def **EditDictionaryWithTypeInfo** (*settingsStructure*, *exu*= None, *dictionaryName*= 'edit')

- **function description**: edit dictionaryData and return modified (new) dictionary
- **input**:

  settingsStructure: hierarchical settings structure, e.g., SC.visualizationSettings

  exu: exudyn module

  dictionaryName: name displayed in dialog
- **output**: returns modified dictionary, which can be used, e.g., for SC.visualizationSettings.SetDictionary(...)

7.11 Module: interactive

Utilities for interactive simulation and results monitoring; NOTE: does not work on MacOS!

Author: Johannes Gerstmayr

Date: 2021-01-17 (created)

def **AnimateModes** (*systemContainer*, *mainSystem*, *nodeNumber*, *period*= 0.04, *stepsPerPeriod*= 30, *showTime*= True, *renderWindowText*= '', *runOnStart*= False, *runMode*= 0, *scaleAmplitude*= 1, *title*= '', *fontSize*= 12, *checkRenderEngineStopFlag*= True, *systemEigenVectors*= None)

- **function description**: animate modes of ObjectFFRFreducedOrder, of nodal coordinates (changes periodically one nodal coordinate) or of a list of system modes provided as list of lists; for creating snapshots, press 'Static' and 'Record frames' and press 'Run' to save one figure in the image subfolder; for creating animations for one mode, use the same procedure but use 'One Cycle'. Modes may be inverted by pressing according '+' and '-' buttons next to Amplitude.
- **input**:

  systemContainer: system container (usually SC) of your model, containing visualization settings

  mainSystem: system (usually mbs) containing your model

nodeNumber: node number of which the coordinates shall be animated. In case of ObjectFFRFreducedOrder, this is the generic node, e.g., 'nGenericODE2' in the dictionary returned by the function AddObjectFFRFreducedOrderWithUserFunctions(...); if nodeNumber=None, then the systemEigenVectors list is used

period: delay for animation of every frame; the default of 0.04 results in approximately 25 frames per second

stepsPerPeriod: number of steps into which the animation of one cycle of the mode is split into

showTime: show a virtual time running from 0 to 2*pi during one mode cycle

renderWindowText: additional text written into renderwindow before 'Mode X' (use \n to add line breaks)

runOnStart: immediately go into 'Run' mode

runMode: 0=continuous run, 1=static continuous, 2=one cycle, 3=static (use slider/mouse to vary time steps)

scaleAmplitude: additional scaling for amplitude if necessary

fontSize: define font size for labels in InteractiveDialog

title: if empty, it uses default; otherwise define specific title

checkRenderEngineStopFlag: if True, stopping renderer (pressing Q or Escape) also causes stopping the interactive dialog

systemEigenVectors: may be a list of lists of system eigenvectors for ODE2 (and possibly ODE1) coordinates or a eigenvector matrix containing mode vectors in columns; if nodeNumber=None, these eigenvectors are then used to be animated

– **output**: opens interactive dialog with further settings

– **notes**:

Uses class InteractiveDialog in the background, which can be used to adjust animation creation. If meshes are large, animation artifacts may appear, which are resolved by using a larger update period.

Press 'Run' to start animation; Chose 'Mode shape', according component for contour plot; to record one cycle for animation, choose 'One cycle', run once to get the according range in the contour plot, press 'Record frames' and press 'Run', now images can be found in subfolder 'images' (for further info on animation creation see Section 2.4.13); now deactivate 'Record frames' by pressing 'Off' and chose another mode

For examples on AnimateModes see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `CMSexampleCourse.py` (Ex), `netgenSTLtest.py` (Ex), `NGsolveCMStutorial.py` (Ex), `NGsolveCraigBampton.py` (Ex), `NGsolveFFRF.py` (Ex), ... , `objectFFRFreducedOrderShowModes.py` (TM), `runTestExamples.py` (TM)

---

def **SolutionViewer**(*mainSystem*, *solution*= None, *rowIncrement*= 1, *timeout*= 0.04, *runOnStart*= True, *runMode*= 2, *fontSize*= 12, *title*= '', *checkRenderEngineStopFlag*= True)

- **NOTE**: this function is directly available in MainSystem (mbs); it should be directly called as mbs.SolutionViewer(...). For description of the interface, see the MainSystem Python extensions, Section 6.5.2.

---

def **ConvertImages2Video** (*workingDir*= 'images', *inputPattern*= 'frame

- **function description**: function to call ffmpeg in the background and convert images to video; requires ffmpeg-python to be installed
- **input**:

  workingDir: directory where images are stored and where animation is written to

  inputPattern: pattern of images; 'frame' is the name used in visualizationSettings.exportImages.saveImageF if saveImageFormat=PNG, then the ending is .png

  outputFile: filename and ending (.mp4 recommended) for generated video

  inputFrameRate: framerate for images relative to outputFrameRate: if inputFrameRate=50 and outputFrameRate=25, then only every second frame is used

  outputFrameRate: framerate for resulting video

  compressionCRF: compression rate of ffmpeg, where 0=uncompressed, 25 is medium compression, >30 is very low quality

  startNumber: start index of first frame chosen for animation

  totalFrames: total number of frames (keep field empty to select all frames after startNumber)
- **output**: None; writes animation when finished
- **example**:

```
#after successful simulation, call:
mbs.SolutionViewer() #click "Stop", "One Cycle" and "Record frames" => close window
#if images are in folder 'images', then call this to create animation:
ConvertImages2Video(workingDir='images', outputFile='test.mp4')
```

---

def **InteractiveImages2Video** (*closeAfterCreation*= False, *fontSize*= 11)

- **function description**: interactive dialog to convert generated images to videos using ffmpeg library; see also ConvertImages2Video() for meaning of values; requires ffmpeg-python to be installed

7.11.1 CLASS InteractiveDialog (in module interactive)

class description: create an interactive dialog, which allows to interact with simulations the dialog has a 'Run' button, which initiates the simulation and a 'Stop' button which stops/pauses simulation; 'Quit' closes the simulation model for examples, see `simulateInteractively.py` and `massSpringFrictionInteractive.py` use __init__ method to setup this class with certain buttons, edit boxes and sliders

- **example**:

```
#the following example is only demonstrating the structure of dialogItems and plots
#dialogItems structure:
#general items:
#    'type' can be out of:
#               'label' (simple text),
#               'button' (button with callback function),
#               'radio' (a radio button with several alternative options),
#               'slider' (with an adjustable range to choose a value)
#    'grid': (row, col, colspan) specifies the row, column and (optionally) the
  span of columns the item is placed at;
#           exception in 'radio', where grid is a list of (row, col) for every
  choice
#    'options': text options, where 'L' means flush left, 'R' means flush right
#suboptions of 'label':
#               'text': a text to be drawn
#suboptions of 'button':
#               'text': a text to be drawn on button
#               'callFunction': function which is called on button-press
#suboptions of 'radio':
#               'textValueList': [('text1',0),('text2',1)] a list of texts with
  according values
#               'value': default value (choice) of radio buttons
#               'variable': according variable in mbs.variables (or mbs.sys), which
   is set to current radio button value
#suboptions of 'slider':
#               'range': (min, max) a tuple containing minimum and maximum value of
   slider
#               'value': default value of slider
#               'steps': number of steps in slider
#               'variable': according variable in mbs.variables (or mbs.sys), which
   is set to current slider value
#example:
dialogItems = [{'type':'label', 'text':'Nonlinear oscillation simulator', 'grid'
  :(0,0,2), 'options':['L']},
               {'type':'button', 'text':'test button','callFunction':ButtonCall, '
  grid':(1,0,2)},
               {'type':'radio', 'textValueList':[('linear',0),('nonlinear',1)], '
  value':0, 'variable':'mode', 'grid': [(2,0),(2,1)]},
```

```
                {'type':'label', 'text':'excitation frequency (Hz):', 'grid':(5,0)},
                {'type':'slider', 'range':(3*f1/800, 3*f1), 'value':omegaInit/(2*pi)
    , 'steps':800, 'variable':'frequency', 'grid':(5,1)},
                {'type':'label', 'text':'damping:', 'grid':(6,0)},
                {'type':'slider', 'range': (0, 40), 'value':damper, 'steps':800, '
    variable':'damping', 'grid':(6,1)},
                {'type':'label', 'text':'stiffness:', 'grid':(7,0)},
                {'type':'slider', 'range':(0, 10000), 'value':spring, 'steps':800, '
    variable':'stiffness', 'grid':(7,1)}]
            #plots structure:
            plots={'nPoints':500,              #number of stored points in subplots (higher
    means slower drawing)
                 'subplots':(2,1),             #(rows, columns) arrangement of subplots (for
    every sensor)
                  #sensors defines per subplot (sensor, coordinate), xlabel and ylabel; if
    coordinate=0, time is used:
                 'sensors':[[(sensPos,0),(sensPos,1),'time','mass position'],
                            [(sensFreq,0),(sensFreq,1),'time','excitation frequency']],
                 'limitsX':[(0,2),(-5,5)],    #x-range per subplot; if not provided, autoscale
     is applied
                 'limitsY':[(-5,5),(0,10),], #y-range per subplot; if not provided, autoscale
     is applied
                 'fontSize':16,               #custom font size for figure
                 'subplots':False,            #if not specified, subplots are created; if
    False, all plots go into one window
                 'lineStyles':['r-','b-'],     #if not specified, uses default '-b', otherwise
     define list of line styles [string for matplotlib.pyplot.plot] per sensor
                 'sizeInches':(12,12)}        #specific x and y size of figure in inches (
    using 100 dpi)
```

def **__init__** (*self*, *mbs*, *simulationSettings*, *simulationFunction*, *dialogItems*, *plots*= None, *period*= 0.04, *realtimeFactor*= 1, *userStartSimulation*= None, *title*= '', *showTime*= False, *fontSize*= 12, *doTimeIntegration*= True, *runOnStart*= False, *addLabelStringVariables*= False, *addSliderVariables*= False, *checkRenderEngineStopFlag*= True, *userOnChange*= None, *useSysVariables*= False)

- **classFunction**: initialize an InteractiveDialog
- **input**:

  mbs: a multibody system to be simulated

  simulationSettings: exudyn.SimulationSettings() according to user settings

  simulationFunction: a user function(mbs, self) which is called before a simulation for the short period is started (e.g, assign special values, etc.); the arguments are the MainSystem mbs and the InteractiveDialog (self)

  dialogItems: a list of dictionaries, which describe the contents of the interactive items, where every dict has the structure 'type':[label, entry, button, slider, check] ... according to tkinter

widgets, 'callFunction': a function to be called, if item is changed/button pressed, 'grid': (row,col) of item to be placed, 'rowSpan': number of rows to be used, 'columnSpan': number of columns to be used; for special item options see notes

plots: list of dictionaries to specify a sensor to be plotted live, see example; otherwise use default None

period: a simulation time span in seconds which is simulated with the simulationFunction in every iteration

realtimeFactor: if 1, the simulation is nearly performed in realtime (except for computation time); if > 1, it runs faster than realtime, if < 1, than it is slower

userStartSimulation: a function F(flag) which is called every time after Run/Stop is pressed. The argument flag = False if button "Run" has been pressed, flag = True, if "Stop" has been pressed

title: title text for interactive dialog

showTime: shows current time in dialog

fontSize: adjust font size for all dialog items

doTimeIntegration: performs internal time integration with given parameters

runOnStart: immediately activate 'Run' button on start

addLabelStringVariables: True: adds a list labelStringVariables containing the (modifiable) list of string variables for label (text) widgets

addSliderVariables: True: adds a list sliderVariables containing the (modifiable) list of variables for slider (=tkinter scale) widgets; this is not necessarily needed for changing slider values, as they can also be modified with dialog.widgets[..].set(...) method

checkRenderEngineStopFlag: if True, stopping renderer (pressing Q or Escape) also causes stopping the interactive dialog

userOnChange: a user function(mbs, self) which is called after period, if widget values are different from values stored in mbs.variables; this usually occurs if buttons are pressed or sliders are moved; the arguments are the MainSystem mbs and the InteractiveDialog (self)

useSysVariables: for internal visualization functions: in this case, variables are written to mbs.sys instead of mbs.variables

– **notes**: detailed description of dialogItems and plots list/dictionary is given in commented the example below

---

def **OnQuit** (*self*, *event*= None)

- **classFunction**: function called when pressing escape or closing dialog

---

def **StartSimulation** (*self*, *event*= None)

- **classFunction**: function called on button 'Run'

---

def **ProcessWidgetStates** (*self*)

- **classFunction**: assign current values of radio buttons and sliders to mbs.variables or mbs.sys

---

def **ContinuousRunFunction** (*self*, *event*= None)

- **classFunction**: function which is repeatedly called when button 'Run' is pressed

---

def **InitializePlots** (*self*)

- **classFunction**: initialize figure and subplots for plots structure

---

def **UpdatePlots** (*self*)

- **classFunction**: update all subplots with current sensor values

---

def **InitializeSolver** (*self*)

- **classFunction**: function to initialize solver for repeated calls

---

def **FinalizeSolver** (*self*)

- **classFunction**: stop solver (finalize correctly)

---

def **RunSimulationPeriod** (*self*)

- **classFunction**: function which performs short simulation for given period

For examples on InteractiveDialog see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `massSpringFrictionInteractive.py` (Ex), `nMassOscillatorInteractive.py` (Ex), `serialRobotInteractiveLimits.py` (Ex),
  `simulateInteractively.py` (Ex), `runTestExamples.py` (TM)

7.12 Module: kinematicTree

A library for preparation of minimal coordinates (kinematic tree) formulation. This library follows mostly the algorithms of Roy Featherstone, see http://royfeatherstone.org/ His code is availble in MATLAB as well as described in the Springer Handbook of Robotics [57]. The main formalisms are based on 6x6 matrices, so-called Plücker transformations, denoted as T66, as defined by Featherstone.

Author: Johannes Gerstmayr

Date: 2021-06-22

def **MassCOMinertia2T66** (*mass*, *centerOfMass*, *inertia*)

- **function description**: convert mass, COM and inertia into 6x6 inertia matrix
- **input**:

  mass: scalar mass

  centerOfMass: 3D vector (list/array)

  inertia: 3x3 matrix (list of lists / 2D array) w.r.t. center of mass
- **output**: 6x6 numpy array for further use in minimal coordinates formulation

---

def **Inertia2T66** (*inertia*)

- **function description**: convert inertia as produced with RigidBodyInertia class into 6x6 inertia matrix (as used in KinematicTree66, Featherstone / Handbook of robotics [57])
- **output**: 6x6 numpy array for further use in minimal coordinates formulation
- **notes**: within the 6x6 matrix, the inertia tensor is defined w.r.t. the center of mass, while RigidBodyInertia defines the inertia tensor w.r.t. the reference point; however, this function correctly transforms all quantities of inertia.

For examples on Inertia2T66 see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `kinematicTreeAndMBS.py` (Ex)

---

def **Inertia66toMassCOMinertia** (*inertia66*)

- **function description**: convert 6x6 inertia matrix into mass, COM and inertia
- **input**: 6x6 numpy array containing rigid body inertia according to Featherstone / Handbook of robotics [57]
- **output**:

  [mass, centerOfMass, inertia]

  mass: scalar mass

  centerOfMass: 3D vector (list/array)

  inertia: 3x3 matrix (list of lists / 2D array) w.r.t. center of mass

---

def **JointTransformMotionSubspace66** (*jointType*, *q*)

- **function description**: return 6x6 Plücker joint transformation matrix evaluated for scalar joint coordinate q and motion subspace ('free modes' in Table 2.6 in Handbook of robotics [57])

For examples on JointTransformMotionSubspace66 see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `kinematicTreeAndMBS.py` (Ex)

---

def **JointTransformMotionSubspace** (*jointType*, *q*)

- **function description**: return list containing rotation matrix, translation vector, rotation axis and translation axis for joint transformation

For examples on JointTransformMotionSubspace see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `kinematicTreeAndMBS.py` (Ex)

---

def **CRM** (*v*)

- **function description**: computes cross product operator for motion from 6D vector v; CRM(v) @ m computes the cross product of v and motion m

---

def **CRF** (*v*)

- **function description**: computes cross product operator for force from 6D vector v; CRF(v) @ f computes the cross product of v and force f

7.12.1 CLASS KinematicTree33 (in module kinematicTree)

class description: class to define a kinematic tree in Python, which can be used for building serial or tree-structured multibody systems (or robots) with a minimal coordinates formulation, using rotation matrices and 3D offsets; for efficient computation, use the C++ ObjectKinematicTree

- **notes**: The formulation and structures widely follows the more efficient formulas (but still implemented in Python!) with 3D vectors and rotation matrices as proposed in Handbook of robotics [57], Chapter 3, but with the rotation matrices (`listOfRotations`) being transposed in the Python implementation as compared to the description in the book, being thus compliant with other Exudyn functions; the 3D vector/matrix Python implementation does not offer advantages as compared to the formulation with Plücker coordinates, BUT it reflects the formulas of the C++ implementation and is used for testing

def **__init__** (*self*, *listOfJointTypes*, *listOfRotations*, *listOfOffsets*, *listOfInertia3D*, *listOfCOM*, *listOfMass*, *listOfParents*= [], *gravity*= [0,0,-9.81])

– **classFunction**: initialize kinematic tree

– **input**:

listOfJointTypes: mandatory list of joint types 'Rx', 'Ry', 'Rz' denoting revolute joints; 'Px', 'Py', 'Pz', denoting prismatic joints

listOfRotations: per link rotation matrix, transforming coordinates of the joint coordinate system w.r.t. the previous coordinate system (this is the inverse of Plücker coordinate transforms (6x6))

listOfOffsets: per link offset vector from pervious coordinate system to the joint coordinate system

listOfInertia3D: per link 3D inertia matrix, w.r.t. reference point (not COM!)

listOfCOM: per link vector from reference point to center of mass (COM), in link coordinates

listOfMass: mass per link

listOfParents: list of parent object indices (int), according to the index in jointTypes and transformations; use empty list for kinematic chain and use -1 if no parent exists (parent=base or world frame)

gravity: a 3D list/array containing the gravity applied to the kinematic tree (in world frame)

---

def **Size** (*self*)

– **classFunction**: return number of joints, defined by size of jointTypes

---

def **XL** (*self*, *i*)

– **classFunction**: return [A, p] containing rotation matrix and offset for joint j

---

def **ForwardDynamicsCRB** (*self*, *q*= [], *q_t*= [], *torques*= [], *forces*= [])

– **classFunction**: compute forward dynamics using composite rigid body algorithm

– **input**:

q: joint space coordinates for the model at which the forward dynamics is evaluated

q_t: joint space velocity coordinates for the model at which the forward dynamics is evaluated

torques: a vector of torques applied at joint coordinates or list/array with zero length

forces: forces acting on the bodies using special format

– **output**: returns acceleration vector q_tt of joint coordinates

---

def **ComputeMassMatrixAndForceTerms** (*self*, *q*, *q_t*, *externalForces*= [])

– **classFunction**:

compute generalized mass matrix M and generalized force terms for
kinematic tree, using current state (joint) variables q and
joint velocities q_t. The generalized force terms f = fGeneralized
contain Coriolis and gravity if given in the kinematicTree.

– **input**:

q: current joint coordinates

q_t: current joint velocities

externalForces: list of torque/forces in global (world) frame per joint; may be empty list, containing 6D vectors or matrices with 6D vectors in columns that are summed up for each link

– **output**: mass matrix **M** and RHS vector $\mathbf{f}_{RHS}$ for equations of motion $M(q)\cdot q_{tt} + f(q, q_t, externalForces) = \tau$; RHS is $\mathbf{f}_{RHS} = \tau - f(q, q_t, externalForces)$; τ can be added outside of `ComputeMassMatrixAndForceTerms`

For examples on KinematicTree33 see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `kinematicTreeAndMBS.py` (Ex)

7.12.2 CLASS KinematicTree66 (in module kinematicTree)

class description: class to define a kinematic tree, which can be used for building serial or tree-structured multibody systems (or robots) with a minimal coordinates formulation, using Plücker coordinate transforms (6x6); for efficient computation, use the C++ ObjectKinematicTree

– **notes**: The formulation and structures widely follow Roy Featherstone (http://royfeatherstone.org/) / Handbook of robotics [57]

def **__init__** (*self*, *listOfJointTypes*, *listOfTransformations*, *listOfInertias*, *listOfParents*= [], *gravity*= [0,0,-9.81])

- **classFunction**: initialize kinematic tree
- **input**:

  listOfJointTypes: mandatory list of joint types 'Rx', 'Ry', 'Rz' denoting revolute joints; 'Px', 'Py', 'Pz', denoting prismatic joints

  listOfTransformations: provide a list of Plücker coordinate transforms (6x6 numpy matrices), describing the (constant) link transformation from the link coordinate system (previous/-parent joint) to this joint coordinate system

  listOfInertias: provide a list of inertias as (6x6 numpy matrices), as produced by the function MassCOMinertia2T66

  listOfParents: list of parent object indices (int), according to the index in jointTypes and transformations; use empty list for kinematic chain and use -1 if no parent exists (parent=base or world frame)

  gravity: a 3D list/array containing the gravity applied to the kinematic tree (in world frame)

---

def **Size** (*self*)

- **classFunction**: return number of joints, defined by size of jointTypes

---

def **XL** (*self*, *i*)

- **classFunction**: return 6D transformation of joint i, given by transformation

---

def **ForwardDynamicsCRB** (*self*, *q*= [], *q_t*= [], *torques*= [], *forces*= [])

- **classFunction**: compute forward dynamics using composite rigid body algorithm
- **input**:

  q: joint space coordinates for the model at which the forward dynamics is evaluated

  q_t: joint space velocity coordinates for the model at which the forward dynamics is evaluated

torques: a vector of torques applied at joint coordinates or list/array with zero length

forces: forces acting on the bodies using special format

– **output**: returns acceleration vector q_tt of joint coordinates

---

def **ComputeMassMatrixAndForceTerms** (*self*, *q*, *q_t*, *externalForces*= [])

– **classFunction**:
compute generalized mass matrix M and generalized force terms for
kinematic tree, using current state (joint) variables q and
joint velocities q_t. The generalized force terms f = fGeneralized
contain Coriolis and gravity if given in the kinematicTree.

– **input**:

q: current joint coordinates

q_t: current joint velocities

externalForces: list of torque/forces in global (world) frame per joint; may be empty list, containing 6D vectors or matrices with 6D vectors in columns that are summed up for each link

– **output**: mass matrix **M** and RHS vector $\mathbf{f}_{RHS}$ for equations of motion $M(q)\cdot q_{tt} + f(q, q_t, externalForces) = \tau$; RHS is $\mathbf{f}_{RHS} = \tau - f(q, q_t, externalForces)$; τ can be added outside of `ComputeMassMatrixAndForceTerms`

---

def **AddExternalForces** (*self*, *Xup*, *fvp*, *externalForces*= [])

– **classFunction**: add action of external forces to forces fvp and return new composed vector of forces fvp

– **input**:

Xup: 6x6 transformation matrices per joint; as computed in ComputeMassMatrixAndForceTerms

fvp: force (torque) per joint, as computed in ComputeMassMatrixAndForceTerms

externalForces: list of torque/forces in global (world) frame per joint; may be empty list, containing 6D vectors or matrices with 6D vectors in columns that are summed up for each link

For examples on KinematicTree66 see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `kinematicTreeAndMBS.py` (Ex)

7.13 Module: lieGroupBasics

Lie group methods and formulas for Lie group integration.

Author: Stefan Holzinger, Johannes Gerstmayr

Date: 2020-09-11

References:

> For details on Lie group methods used here, see the references [30, 58, 5, 60, 59, 61, 43]. Lie group methods for rotation vector are described in Holzinger and Gerstmayr [21, 32].

def **Sinc** (x)

- **function description**: compute the cardinal sine function in radians
- **input**: scalar float or int value
- **output**: float value in radians
- **author**: Stefan Holzinger

---

def **Cot** (x)

- **function description**: compute the cotangent function cot(x)=1/tan(x) in radians
- **input**: scalar float or int value
- **output**: float value in radians
- **author**: Stefan Holzinger

---

def **R3xSO3Matrix2RotationMatrix** (G)

- **function description**: computes 3x3 rotation matrix from 7x7 R3xSO(3) matrix, see [5]
- **input**: G: 7x7 matrix as np.array
- **output**: 3x3 rotation matrix as np.array
- **author**: Stefan Holzinger

---

def **R3xSO3Matrix2Translation** (*G*)

- **function description**: computes translation part of R3xSO(3) matrix, see [5]
- **input**: G: 7x7 matrix as np.array
- **output**: 3D vector as np.array containg translational part of R3xSO(3)
- **author**: Stefan Holzinger

---

def **R3xSO3Matrix** (*x*, *R*)

- **function description**: builds 7x7 matrix as element of the Lie group R3xSO(3), see [5]
- **input**:

  x: 3D vector as np.array representing the translation part corresponding to R3

  R: 3x3 rotation matrix as np.array
- **output**: 7x7 matrix as np.array
- **author**: Stefan Holzinger

---

def **ExpSO3** (*Omega*)

- **function description**: compute the matrix exponential map on the Lie group SO(3), see [43]
- **input**: 3D rotation vector as np.array
- **output**: 3x3 matrix as np.array
- **author**: Stefan Holzinger

---

def **ExpS3** (*Omega*)

- **function description**: compute the quaternion exponential map on the Lie group S(3), see [61, 43]

- **input**: 3D rotation vector as np.array
- **output**:

  4D vector as np.array containing four Euler parameters

  entry zero of output represent the scalar part of Euler parameters
- **author**: Stefan Holzinger

---

def **LogSO3** (*R*)

- **function description**: compute the matrix logarithmic map on the Lie group SO(3)
- **input**: 3x3 rotation matrix as np.array
- **output**: 3x3 skew symmetric matrix as np.array
- **author**: Johannes Gerstmayr
- **notes**: improved accuracy for very small angles as well as angles phi close to pi AS WELL AS at phi=pi

---

def **TExpSO3** (*Omega*)

- **function description**: compute the tangent operator corresponding to ExpSO3, see [5]
- **input**: 3D rotation vector as np.array
- **output**: 3x3 matrix as np.array
- **author**: Stefan Holzinger

---

def **TExpSO3Inv** (*Omega*)

- **function description**:

  compute the inverse of the tangent operator TExpSO3, see [60]

  this function was improved, see coordinateMaps.pdf by Stefan Holzinger

- **input**: 3D rotation vector as np.array
- **output**: 3x3 matrix as np.array
- **author**: Stefan Holzinger

---

def **ExpSE3** (*x*)

- **function description**: compute the matrix exponential map on the Lie group SE(3), see [5]
- **input**: 6D incremental motion vector as np.array
- **output**: 4x4 homogeneous transformation matrix as np.array
- **author**: Stefan Holzinger

For examples on ExpSE3 see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `serialRobotInverseKinematics.py` (Ex)

---

def **LogSE3** (*H*)

- **function description**: compute the matrix logarithm on the Lie group SE(3), see [60]
- **input**: 4x4 homogeneous transformation matrix as np.array
- **output**: 4x4 skew symmetric matrix as np.array
- **author**: Stefan Holzinger

For examples on LogSE3 see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `serialRobotInverseKinematics.py` (Ex)

---

def **TExpSE3** (*x*)

- **function description**: compute the tangent operator corresponding to ExpSE3, see [5]
- **input**: 6D incremental motion vector as np.array
- **output**: 6x6 matrix as np.array

- **author**: Stefan Holzinger
- **notes**: improved accuracy for very small angles as well as angles phi

---

def **TExpSE3Inv** (x)

- **function description**: compute the inverse of tangent operator TExpSE3, see [60]
- **input**: 6D incremental motion vector as np.array
- **output**: 6x6 matrix as np.array
- **author**: Stefan Holzinger
- **notes**: improved accuracy for very small angles as well as angles phi

---

def **ExpR3xSO3** (x)

- **function description**: compute the matrix exponential map on the Lie group R3xSO(3), see [5]
- **input**: 6D incremental motion vector as np.array
- **output**: 7x7 matrix as np.array
- **author**: Stefan Holzinger

---

def **TExpR3xSO3** (x)

- **function description**: compute the tangent operator corresponding to ExpR3xSO3, see [5]
- **input**: 6D incremental motion vector as np.array
- **output**: 6x6 matrix as np.array
- **author**: Stefan Holzinger

---

def **TExpR3xSO3Inv** (*x*)

- **function description**: compute the inverse of tangent operator TExpR3xSO3
- **input**: 6D incremental motion vector as np.array
- **output**: 6x6 matrix as np.array
- **author**: Stefan Holzinger

---

def **CompositionRuleDirectProductR3AndS3** (*q0*, *incrementalMotionVector*)

- **function description**: compute composition operation for pairs in the Lie group R3xS3
- **input**:

  q0: 7D vector as np.array containing position coordinates and Euler parameters

  incrementalMotionVector: 6D incremental motion vector as np.array
- **output**: 7D vector as np.array containing composed position coordinates and composed Euler parameters
- **author**: Stefan Holzinger

---

def **CompositionRuleSemiDirectProductR3AndS3** (*q0*, *incrementalMotionVector*)

- **function description**: compute composition operation for pairs in the Lie group R3 semiTimes S3 (corresponds to SE(3))
- **input**:

  q0: 7D vector as np.array containing position coordinates and Euler parameters

  incrementalMotionVector: 6D incremental motion vector as np.array
- **output**: 7D vector as np.array containing composed position coordinates and composed Euler parameters
- **author**: Stefan Holzinger

---

def **CompositionRuleDirectProductR3AndR3RotVec** (*q0*, *incrementalMotionVector*)

- **function description**:

  compute composition operation for pairs in the group obtained from the direct product of R3 and R3, see [21]

  the rotation vector is used as rotation parametrizations

  this composition operation can be used in formulations which represent the translational velocities in the global (inertial) frame

- **input**:

  q0: 6D vector as np.array containing position coordinates and rotation vector

  incrementalMotionVector: 6D incremental motion vector as np.array

- **output**: 7D vector as np.array containing composed position coordinates and composed rotation vector

- **author**: Stefan Holzinger

---

def **CompositionRuleSemiDirectProductR3AndR3RotVec** (*q0*, *incrementalMotionVector*)

- **function description**:

  compute composition operation for pairs in the group obtained from the direct product of R3 and R3.

  the rotation vector is used as rotation parametrizations

  this composition operation can be used in formulations which represent the translational velocities in the local (body-attached) frame

- **input**:

  q0: 6D vector as np.array containing position coordinates and rotation vector

  incrementalMotionVector: 6D incremental motion vector as np.array

- **output**: 6D vector as np.array containing composed position coordinates and composed rotation vector

- **author**: Stefan Holzinger

---

def **CompositionRuleDirectProductR3AndR3RotXYZAngles** (*q0*, *incrementalMotionVector*)

- **function description**:

  compute composition operation for pairs in the group obtained from the direct product of R3 and R3.

  Cardan-Tait/Bryan (CTB) angles are used as rotation parametrizations

  this composition operation can be used in formulations which represent the translational velocities in the global (inertial) frame

- **input**:

  q0: 6D vector as np.array containing position coordinates and Cardan-Tait/Bryan angles

  incrementalMotionVector: 6D incremental motion vector as np.array

- **output**: 6D vector as np.array containing composed position coordinates and composed Cardan-Tait/Bryan angles

- **author**: Stefan Holzinger

---

def **CompositionRuleSemiDirectProductR3AndR3RotXYZAngles** (*q0, incrementalMotionVector*)

- **function description**:

  compute composition operation for pairs in the group obtained from the direct product of R3 and R3.

  Cardan-Tait/Bryan (CTB) angles are used as rotation parametrizations

  this composition operation can be used in formulations which represent the translational velocities in the local (body-attached) frame

- **input**:

  q0: 6D vector as np.array containing position coordinates and Cardan-Tait/Bryan angles

  incrementalMotionVector: 6D incremental motion vector as np.array

- **output**: 6D vector as np.array containing composed position coordinates and composed Cardan-Tait/Bryan angles

- **author**: Stefan Holzinger

---

def **CompositionRuleForEulerParameters** (*q, p*)

- **function description**:

compute composition operation for Euler parameters (unit quaternions)

this composition operation is quaternion multiplication, see [61]

- **input**:

  q: 4D vector as np.array containing Euler parameters

  p: 4D vector as np.array containing Euler parameters

- **output**: 4D vector as np.array containing composed (multiplied) Euler parameters

- **author**: Stefan Holzinger

---

def **CompositionRuleForRotationVectors** (*v0*, *Omega*)

- **function description**: compute composition operation for rotation vectors v0 and Omega, see [32]

- **input**:

  v0: 3D rotation vector as np.array

  Omega: 3D (incremental) rotation vector as np.array

- **output**: 3D vector as np.array containing composed rotation vector v

- **author**: Stefan Holzinger

---

def **CompositionRuleRotXYZAnglesRotationVector** (*alpha0*, *Omega*)

- **function description**: compute composition operation for RotXYZ angles, see [32]

- **input**:

  alpha0: 3D vector as np.array containing RotXYZ angles

  Omega: 3D vector as np.array containing the (incremental) rotation vector

- **output**: 3D vector as np.array containing composed RotXYZ angles

- **author**: Stefan Holzinger

7.14 Module: mainSystemExtensions

NOTE: This module only contains links for extensions of C++ classes. The description is available in the respective descriptions of the C++ interface.

7.15 Module: particles

This module offers methods for GeneralContact, in particular particles (DEM - discrete element method)

Author: Johannes Gerstmayr

Date: 2024-10-19 (created)

def **CreateParticlesInBox** (*minPointBox*, *maxPointBox*, *minRadius*, *maxRadius*= None, *maxNumberOfParticles*= None, *offsetRadius*= 0, *verbose*= 0)

- **function description**: create set of spherical particles densly packed inside box using hexagonal closest packing (HCP); radius is randomized between minRadius and maxRadius
- **input**:

  minPointBox: [xMin,yMin,zMin] minimal cartesian coordinates for box

  maxPointBox: [xMax,yMax,zMax] maximal cartesian coordinates for box

  minRadius: minimal or nominal radius

  maxRadius: maximal radius for randomized variations of radius or None to use minRadius

  maxNumberOfParticles: if not None, this limits the amount of created particles; otherwise number of particles depends on geometry

  offsetRadius: additional space between spheres (by assuming a larger radius for packing)

  verbose: if > 0 some main parameters are printed
- **output**: [(point0, radius0), ...] a list of point-radius tuples containing the information of created particles

For examples on CreateParticlesInBox see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `simulatorCouplingTwoMbs.py` (TM)

7.16 Module: physics

The physics library includes helper functions and data related to physics models and parameters; for rigid body inertia, see rigidBodyUtilities

Date: 2021-01-20

def **StribeckFunction** (*vel*, *muDynamic*, *muStaticOffset*, *muViscous*= 0, *expVel*= 1e-3, *regVel*= 1e-3)

- **function description**:

  describes regularized Stribeck function with optial viscous part for given velocity,

$$f(v) = \begin{cases} (\mu_d + \mu_{s_{off}})v, & \text{if} \quad |v| <= v_{reg} \\ \text{Sign}(v)\left(\mu_d + \mu_{s_{off}} e^{-(|v|-v_{reg})/v_{exp}} + \mu_v(|v| - v_{reg})\right), & \text{else} \end{cases}$$

- **input**:

  vel: input velocity v

  muDynamic: dynamic friction coefficient μ_d

  muStaticOffset: $\mu_{s_{off}}$, offset to dynamic friction, which gives muStaticFriction = muDynamic + muStaticOffset

  muViscous: μ_v, viscous part, acting proportional to velocity except for regVel

  regVel: v_{reg}, small regularization velocity in which the friction is linear around zero velocity (e.g., to get Newton converged)

  expVel: v_{exp}, velocity (relative to regVel, at which the muStaticOffset decreases exponentially, at vel=expVel, the factor to muStaticOffset is exp(-1) = 36.8%)

- **output**: returns velocity dependent friction coefficient (if muDynamic and muStaticOffset are friction coefficients) or friction force (if muDynamic and muStaticOffset are on force level)

- **notes**: see Isermann (2008) and Armstrong-Helouvry (1991)

For examples on StribeckFunction see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- massSpringFrictionInteractive.py (Ex), springMassFriction.py (Ex)

---

def **RegularizedFrictionStep** (*x*, *x0*, *h0*, *x1*, *h1*)

- **function description**: helper function for RegularizedFriction(...)

---

def **RegularizedFriction** (*vel*, *muDynamic*, *muStaticOffset*, *velStatic*, *velDynamic*, *muViscous*= 0)

- **function description**: describes regularized friction function, with increased static friction, dynamic friction and optional viscous part

- **input**:

  vel: input velocity

  muDynamic: dynamic friction coefficient

  muStaticOffset: offset to dynamic friction, which gives muStaticFriction = muDynamic + muStaticOffset

muViscous: viscous part, acting proportional to velocity for velocities larger than velDynamic; extension to mentioned references

velStatic: small regularization velocity at which exactly the staticFriction is reached; for smaller velocities, the friction is smooth and zero-crossing (unphysical!) (e.g., to get Newton converged)

velDynamic: velocity at which muDynamic is reached for first time

- **output**: returns velocity dependent friction coefficient (if muDynamic and muStaticOffset are friction coefficients) or friction force (if muDynamic and muStaticOffset are on force level)
- **notes**: see references: Flores et al. [13], Qian et al. [49]

For examples on RegularizedFriction see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `massSpringFrictionInteractive.py` (Ex)

---

def **VonMisesStress** (*stress6D*)

- **function description**: compute equivalent von-Mises stress given 6 stress components or list of stress6D (or stress6D in rows of np.array)
- **input**: stress6D: 6 stress components as list or np.array, using ordering $[\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{yz}, \sigma_{xz}, \sigma_{xy}]$
- **output**: returns scalar equivalent von-Mises stress or np.array of von-Mises stresses for all stress6D

---

def **UFvonMisesStress** (*mbs*, *t*, *sensorNumbers*, *factors*, *configuration*)

- **function description**: Sensor user function to compute equivalent von-Mises stress from sensor with Stress or StressLocal OutputVariableType; if more than 1 sensor is given in sensorNumbers, then the maximum stress is computed
- **input**: arguments according to `SensorUserFunction`; factors are ignored
- **output**: returns scalar (maximum) equivalent von-Mises stress
- **example**:

```
#assuming s0, s1, s2 being sensor numbers with StressLocal components
sUser = mbs.AddSensor(SensorUserFunction(sensorNumbers=[s0,s1,s2],
                                         fileName='solution/sensorMisesStress.txt',
                                         sensorUserFunction=UFvonMisesStress))
```

7.17 Module: plot

Plot utility functions based on matplotlib, including plotting of sensors and FFT.

Author: Johannes Gerstmayr

Date: 2020-09-16 (created)

Notes: For a list of plot colors useful for matplotlib, see also advancedUtilities.PlotLineCode(...)

def **ParseOutputFileHeader** (*lines*)

- **function description**: parse header of output file (solution file, sensor file, genetic optimization output, ...) given in file.readlines() format
- **output**: return dictionary with 'type'=['sensor','solution','geneticOptimization','parameterVariation'], 'variableType' containing variable types, 'variableRanges' containing ranges for parameter variation

---

def **PlotSensorDefaults** ()

- **function description**: returns structure with default values for PlotSensor which can be modified once to be set for all later calls of PlotSensor
- **example**:

```
#change one parameter:
plot.PlotSensorDefaults().fontSize = 12
#==>now PlotSensor(...) will use fontSize=12
#==>now PlotSensor(..., fontSize=10) will use fontSize=10
#==>BUT PlotSensor(..., fontSize=16) will use fontSize=12, BECAUSE 16 is the
  original default value!!!
#see which parameters are available:
exudyn.Print(PlotSensorDefaults())
```

For examples on PlotSensorDefaults see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `serialRobotFlexible.py` (Ex), `coordinateVectorConstraintGenericODE2.py` (TM)

---

def **PlotSensor**(*mbs*, *sensorNumbers*= [], *components*= 0, *xLabel*= 'time (s)', *yLabel*= None, *labels*= [], *colorCodeOffset*= 0, *newFigure*= True, *closeAll*= False, *componentsX*= [], *title*= '', *figureName*= '', *fontSize*= 16, *colors*= [], *lineStyles*= [], *lineWidths*= [], *markerStyles*= [], *markerSizes*= [], *markerDensity*= 0.08, *rangeX*= [], *rangeY*= [], *majorTicksX*= 10, *majorTicksY*= 10, *offsets*= [], *factors*= [], *subPlot*= [], *sizeInches*= [6.4,4.8], *fileName*= '', *useXYZcomponents*= True, ***kwargs*)

- **NOTE**: this function is directly available in MainSystem (mbs); it should be directly called as mbs.PlotSensor(...). For description of the interface, see the MainSystem Python extensions, Section 6.5.2.

---

def **PlotFFT** (*frequency*, *data*, *xLabel*= 'frequency', *yLabel*= 'magnitude', *label*= '', *freqStart*= 0, *freqEnd*= -1, *logScaleX*= True, *logScaleY*= True, *majorGrid*= True, *minorGrid*= True)

- **function description**: plot fft spectrum of signal
- **input**:

  frequency: frequency vector (Hz, if time is in SECONDS)

  data: magnitude or phase as returned by ComputeFFT() in exudyn.signalProcessing

  xLabel: label for x-axis, default=frequency

  yLabel: label for y-axis, default=magnitude

  label: either empty string ('') or name used in legend

  freqStart: starting range for frequency

  freqEnd: end of range for frequency; if freqEnd==-1 (default), the total range is plotted

  logScaleX: use log scale for x-axis

  logScaleY: use log scale for y-axis

  majorGrid: if True, plot major grid with solid line

  minorGrid: if True, plot minor grid with dotted line
- **output**: creates plot and returns plot (plt) handle

For examples on PlotFFT see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `nMassOscillatorEigenmodes.py` (Ex)

---

def **FileStripSpaces** (*filename*, *outputFilename*, *fileCommentChar*= '', *removeDoubleChars*= '')

- **function description**: strip spaces at beginning / end of lines; this may be sometimes necessary when reading solutions from files that are space-separated
- **input**:

  filename: name of file to process

  outputFilename: name of file to which text without leading/trailing spaces is written

  fileCommentChar: if not equal '', lines starting with this character will not be processed

removeDoubleChars: if not equal '', this double characters (especial multiple spaces) will be removed; '1.0 3.0' will be converted into '1.0 3.0'

- **output**: new file written

---

def **DataArrayFromSensorList** (*mbs*, *sensorNumbers*, *positionList*= [], *time*= '')

- **function description**: helper function to create data array from outputs defined by sensorNumbers list [+optional positionList which must have, e.g., local arc-length of beam according to sensor numbers]; if time==', current sensor values will be used; if time!=[], evaluation will be based on loading values from file or sensor internal data and evaluate at that time
- **input**:

  mbs: a MainSystem where the sensors are given

  sensorNumbers: a list of sensor numbers, which shall be evaluated

  positionList: an optional list of positions per sensor (e.g., axial positions at beam)

  time: optional time at which the sensor values are evaluated (currently not implemented)
- **output**: returns data as numpy array, containg per row the number or position (positionList) in the first column and all sensor values in the remaining columns

For examples on DataArrayFromSensorList see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `beltDriveALE.py` (Ex), `beltDriveReevingSystem.py` (Ex)

---

def **LoadImage** (*fileName*, *trianglesAsLines*= True, *verbose*= False)

- **function description**: import image text file as exported from renderer.RedrawAndSaveImage() with exportImages.saveImageFormat='TXT'; triangles are converted to lines
- **input**: fileName includes directory
- **output**: returns dictionary with according structures

For examples on LoadImage see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `NGsolveCraigBampton.py` (Ex), `NGsolvePistonEngine.py` (Ex)

---

def **PlotImage** (*imageData*, *HT*= np.eye(4), *axesEqual*= True, *plot3D*= False, *lineWidths*= 1, *lineStyles*= '-', *triangleEdgeColors*= 'black', *triangleEdgeWidths*= 0.5, *removeAxes*= True, *orthogonalProjection*= True, *title*= '', *figureName*= '', *fileName*= '', *fontSize*= 16, *closeAll*= False, *azim*= 0., *elev*= 0.)

- **function description**: plot 2D or 3D vector image data as provided by LoadImage(...) using matplotlib
- **input**:

  imageData: dictionary as provided by LoadImage(...)

  HT: homogeneous transformation, used to transform coordinates; lines are drawn in (x,y) plane

  axesEqual: for 2D mode, axis are set equal, otherwise model is distorted

  plot3D: in this mode, a 3D visualization is used; triangles are only be displayed in this mode!

  lineWidths: width of lines

  lineStyles: matplotlib codes for lines

  triangleEdgeColors: color for triangle edges as tuple of rgb colors or matplotlib color code strings 'black', 'r', ...

  triangleEdgeWidths: width of triangle edges; set to 0 if edges shall not be shown

  removeAxes: if True, all axes and background are removed for simpler export

  orthogonalProjection: if True, projection is orthogonal with no perspective view

  title: optional string representing plot title

  figureName: optional name for figure, if newFigure=True

  fileName: if this string is non-empty, figure will be saved to given path and filename (use figName.pdf to safe as PDF or figName.png to save as PNG image); use matplotlib.use('Agg') in order not to open figures if you just want to save them

  fontSize: change general fontsize of axis, labels, etc. (matplotlib default is 12, default in PlotSensor: 16)

  closeAll: if True, close all figures before opening new one (do this only in first PlotSensor command!)

  azim, elev: for 3D plots: the initial angles for the 3D view in degrees

For examples on PlotImage see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- NGsolveCraigBampton.py (Ex), NGsolvePistonEngine.py (Ex)

7.18 Module: processing

The processing module supports multiple execution of EXUDYN models. It includes parameter variation and (genetic) optimization functionality.

Author: Johannes Gerstmayr, Stefan Holzinger

Date: 2020-11-17 (2022-02-04 modified by Stefan Holzinger)

Notes: Parallel processing, which requires multiprocessing library, can lead to considerable speedup (measured speedup factor > 50 on 80 core machine). The progess bar during multiprocessing requires the library tqdm.

def **GetVersionPlatformString** ()

- **function description**:

  internal function to return Exudyn version string, which allows to identify how results have been obtained

  writes something like 'Exudyn version = 1.2.33.dev1; Python3.9.11; Windows AVX2 FLOAT64; Windows10 V10.0.19044; AMD64; Intel64 Family 6 Model 142 Stepping 10, GenuineIntel'

- **notes**: If exudyn C++ module is not available, it outputs the Python version

---

def **ProcessParameterList** (*parameterFunction*, *parameterList*, *useMultiProcessing*, *clusterHostNames*= [], ***kwargs*)

- **function description**: processes parameterFunction for given parameters in parameterList, see ParameterVariation

- **input**:

  parameterFunction: function, which takes the form parameterFunction(parameterDict) and which returns any values that can be stored in a list (e.g., a floating point number)

  parameterList: list of parameter sets (as dictionaries) which are fed into the parameter variation, see example

  useMultiProcessing: if True, the multiprocessing lib is used for parallelized computation; WARNING: be aware that the function does not check if your function runs independently; DO NOT use GRAPHICS and DO NOT write to same output files, etc.!

  numberOfThreads: default: same as number of cpus (threads); used for multiprocessing lib;

  resultsFile: if provided, output is immediately written to resultsFile during processing

clusterHostNames: list of hostnames, e.g. clusterHostNames=['123.124.125.126','123.124.125.127'] providing a list of strings with IP addresses or host names, see dispy documentation. If list is non-empty and useMultiProcessing==True and dispy is installed, cluster computation is used; NOTE that cluster computation speedup factors shown are not fully true, as they include a significant overhead; thus, only for computations which take longer than 1-5 seconds and for sufficient network bandwith, the speedup is roughly true

useDispyWebMonitor: if given in **kwargs, a web browser is startet in case of cluster computation to manage the cluster during computation

useMPI: if given in **kwargs and set True, and if Python package mpi4py is installed, mpi parallelization is used; for hints see parameterVariationExample.py

- **output**: returns values containing the results according to parameterList
- **notes**: options are passed from Parametervariation
- **example**:

```
def PF(parameterSet):
    #in reality, value will be result of a complex exudyn simulation:
    value = sin(parameterSet['mass']) * parameterSet['stiffness']
    return value
values=ProcessParameterList(parameterFunction=PF,
                            parameterList=[{'m':1, 's':100},
                                           {'m':2, 's':100},
                                           {'m':3, 's':100},
                                           {'m':1, 's':200},
                                           {'m':2, 's':250},
                                           {'m':3, 's':300},
                                           ], useMultiProcessing=False )
```

---

def **ParameterVariation** (*parameterFunction*, *parameters*, *useLogSpace*= False, *debugMode*= False, *addComputationIndex*= False, *useMultiProcessing*= False, *showProgress*= True, *parameterFunctionData*= , *clusterHostNames*= [], *numberOfThreads*= None, *resultsFile*= '', ***kwargs*)

- **function description**:

  calls successively the function parameterFunction(parameterDict) with variation of parameters in given range; parameterDict is a dictionary, containing the current values of parameters,

  e.g., parameterDict=['mass':13, 'stiffness':12000] to be computed and returns a value or a list of values which is then stored for each parameter

- **input**:

parameterFunction: function, which takes the form parameterFunction(parameterDict) and which returns any values that can be stored in a list (e.g., a floating point number)

parameters: given as a dictionary, consist of name and tuple of (begin, end, numberOfValues) same as in np.linspace(...), e.g. 'mass':(10,50,10), for a mass varied from 10 to 50, using 10 steps OR a list of values [v0, v1, v2, ...], e.g. 'mass':[10,15,25,50]

useLogSpace: (optional) if True, the parameters are varied at a logarithmic scale, e.g., [1, 10, 100] instead linear [1, 50.5, 100]

debugMode: if True, additional print out is done

addComputationIndex: if True, key 'computationIndex' is added to every parameterDict in the call to parameterFunction(), which allows to generate independent output files for every parameter, etc.

useMultiProcessing: if True, the multiprocessing lib is used for parallelized computation; WARNING: be aware that the function does not check if your function runs independently; DO NOT use GRAPHICS and DO NOT write to same output files, etc.!

showProgress: if True, shows for every iteration the progress bar (requires tqdm library)

resultsFile: if provided, output is immediately written to resultsFile during processing

numberOfThreads: default(None): same as number of cpus (threads); used for multiprocessing lib;

parameterFunctionData: dictionary containing additional data passed to the parameterFunction inside the parameters with dict key 'functionData'; use this e.g. for passing solver parameters or other settings

clusterHostNames: list of hostnames, e.g. clusterHostNames=['123.124.125.126','123.124.125.127'] providing a list of strings with IP addresses or host names, see dispy documentation. If list is non-empty and useMultiProcessing==True and dispy is installed, cluster computation is used; NOTE that cluster computation speedup factors shown are not fully true, as they include a significant overhead; thus, only for computations which take longer than 1-5 seconds and for sufficient network bandwith, the speedup is roughly true

useDispyWebMonitor: if given in **kwargs, a web browser is started in case of cluster computation to manage the cluster during computation

useMPI: if given in **kwargs and set True, and if Python package mpi4py is installed, mpi parallelization is used; for hints see parameterVariationExample.py

– **output**:

returns [parameterList, values], containing, e.g., parameterList={'mass':[1,1,1,2,2,2,3,3,3], 'stiffness':[4,5,6, 4,5,6, 4,5,6]} and the result values of the parameter variation accoring to the parameterList,

values=[7,8,9 ,3,4,5, 6,7,8] (depends on solution of problem ..., can also contain tuples, etc.)

– **example**:

```
if __name__ == '__main__':
    ParameterVariation(parameterFunction=Test,
                       parameters={'mass':(1,10,10), 'stiffness':(1000,10000,10)},
                       useMultiProcessing=True)
```

For examples on ParameterVariation see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- dispyParameterVariationExample.py (Ex), mpi4pyExample.py (Ex), multiprocessingTest.py (Ex), parameterVariationExample.py (Ex), geneticOptimizationTest.py (TM)

---

def **GeneticOptimization** (*objectiveFunction*, *parameters*, *populationSize*= 100, *numberOfGenerations*= 10, *elitistRatio*= 0.1, *crossoverProbability*= 0.25, *crossoverAmount*= 0.5, *rangeReductionFactor*= 0.7, *distanceFactor*= 0.1, *childDistribution*= "uniform", *distanceFactorGenerations*= -1, *debugMode*= False, *addComputationIndex*= False, *useMultiProcessing*= False, *showProgress*= True, *clusterHostNames*= [], *parameterFunctionData*= , ***kwargs*)

- **function description**: compute minimum of given objectiveFunction
- **input**:

  objectiveFunction: function, which takes the form parameterFunction(parameterDict) and which returns a value or list (or numpy array) which reflects the size of the objective to be minimized

  parameters: given as a dictionary, consist of name and tuple containing the search range for this parameter (begin, end), e.g. 'mass':(10,50)

  populationSize: individuals in every generation

  initialPopulationSize: number of random initial individuals; default: population size

  numberOfGenerations: number of generations; NOTE: it is required that elitistRatio*populationSize >= 1

  elitistRatio: the number of surviving individuals in every generation is equal to the previous population times the elitistRatio

  crossoverProbability: if > 0: children are generated from two (randomly selected) parents by gene-crossover; if 0, no crossover is used

  crossoverAmount: if crossoverProbability > 0, then this amount is the probability of genes to cross; 0.1: small amount of genes cross, 0.5: 50% of genes cross

  rangeReductionFactor: reduction of mutation range (boundary) relative to range of last generation; helps algorithm to converge to more accurate values

  distanceFactor: children only survive at a certain relative distance of the current range; must be small enough (< 0.5) to allow individuals to survive; ignored if distanceFactor=0; as a rule of thumb, the distanceFactor should be zero in case that there is only one significant minimum, but if there are many local minima, the distanceFactor should be used to search at several different local minima

childDistribution: string with name of distribution for producing childs: "normal" (Gaussian, with sigma defining range), "uniform" (exactly in range of childs)

distanceFactorGenerations: number of generations (populations) at which the distance factor is active; the distance factor is used to find several local minima; finally, convergence is speed up without the distance factor

parameterFunctionData: dictionary containing additional data passed to the objectiveFunction inside the parameters with dict key 'functionData'; use this e.g. for passing solver parameters or other settings

randomizerInitialization: initialize randomizer at beginning of optimization in order to get reproducible results, provide any integer in the range between 0 and 2**32 - 1 (default: no initialization)

debugMode: if True, additional print out is done

addComputationIndex: if True, key 'computationIndex' is added to every parameterDict in the call to parameterFunction(), which allows to generate independent output files for every parameter, etc.

useMultiProcessing: if True, the multiprocessing lib is used for parallelized computation; WARNING: be aware that the function does not check if your function runs independently; DO NOT use GRAPHICS and DO NOT write to same output files, etc.!

showProgress: if True, shows for every iteration the progress bar (requires tqdm library)

numberOfThreads: default: same as number of cpus (threads); used for multiprocessing lib;

resultsFile: if provided, the results are stored columnwise into the given file and written after every generation; use resultsMonitor.py to track results in realtime

clusterHostNames: list of hostnames, e.g. clusterHostNames=['123.124.125.126','123.124.125.127'] providing a list of strings with IP addresses or host names, see dispy documentation. If list is non-empty and useMultiProcessing==True and dispy is installed, cluster computation is used; NOTE that cluster computation speedup factors shown are not fully true, as they include a significant overhead; thus, only for computations which take longer than 1-5 seconds and for sufficient network bandwith, the speedup is roughly true

useDispyWebMonitor: if given in **kwargs, a web browser is startet in case of cluster computation to manage the cluster during computation

- **output**:

  returns [optimumParameter, optimumValue, parameterList, valueList], containing the optimum parameter set 'optimumParameter', optimum value 'optimumValue', the whole list of parameters parameterList with according objective values 'valueList'

  values=[7,8,9 ,3,4,5, 6,7,8] (depends on solution of problem ..., can also contain tuples, etc.)

- **notes**: This function is still under development and shows an experimental state!

- **example**:

```
GeneticOptimization(objectiveFunction = fOpt, parameters={'mass':(1,10), 'stiffness
  ':(1000,10000)})
```

For examples on GeneticOptimization see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- geneticOptimizationSliderCrank.py (Ex), shapeOptimization.py (Ex), geneticOptimizationTest.py (TM)

---

def **Minimize** (*objectiveFunction*, *parameters*, *initialGuess*= [], *method*= 'Nelder-Mead', *tol*= 1e-4, *options*= , *enforceBounds*= True, *debugMode*= False, *showProgress*= True, *addComputationIndex*= False, *storeFunctionValues*= True, ***kwargs*)

- **function description**: Compute minimum of given objectiveFunction. This function is based on scipy.optimize.minimize() and it provides the same interface as GeneticOptimization(). Note that in special cases, you should copy this function and adapt to your needs.
- **input**:
  objectiveFunction: function, which takes the form parameterFunction(parameterDict) and which returns a value or list (or numpy array) which reflects the size of the objective to be minimized

  parameters: given as a dictionary, consist of name and tuple containing the search range for this parameter (begin, end), e.g. 'mass':(10,50)

  storeFunctionValues: if True, objectiveFunction values are computed (additional costs!) and stored in every iteration into valueList

  initialGuess: initial guess. Array of real elements of size (n,), where 'n' is the number of independent variables. If not provided by the user, initialGuess is computed from bounds provided in parameterDict.

  method: solver that should be used, e.g. 'Nelder-Mead', 'Powell', 'CG' etc. A list of available solvers can be found in the documentation of scipy.optimize.minimize().

  tol: tolerance for termination. When tol is specified, the selected minimization algorithm sets some relevant solver-specific tolerance(s) equal to tol (but this is usually not the tolerance for loss or parameters1). For detailed control, use solver-specific options using the 'options' variable.

  options: dictionary of solver options. Can be used to set absolute and relative error tolerances. Detailed information can be found in the documentation of scipy.optimize.minimize().

  enforceBounds: if True, ensures that only parameters within the bounds specified in ParameterDict are used for minimization; this may help to avoid, e.g., negative values, but may lead to non-convergence

  verbose: prints solver information into console, e.g. number of iterations 'nit', number of funcion evaluations 'nfev', status etc.

showProgress: if True, shows for every iteration objective function value, current iteration number, time needed for current iteration, maximum number of iterations and loss (current value of objective function)

addComputationIndex: if True, key 'computationIndex' is added for consistency reasons with GeneticOptimizaiton to every parameterDict in the call to parameterFunction(); however, the value is always 0, because no multi threading is used in Minimize(...)

resultsFile: if provided, the results are stored columnwise into the given file and written after every generation; use resultsMonitor.py to track results in realtime

useScipyBounds: if True, use scipy.optimize.minimize() option 'bounds' to apply bounds on variable specified in ParameterDict. Note, this option is only used by some specific methods of scipy.optimize.minimize()! method='Nelder-Mead' ignores this option for example! if False, option 'enforceBounds' will be set to False!

args: extra arguments passed to the objective function and its derivatives (fun, jac and hess functions).

- **output**: returns [optimumParameter, optimumValue, parameterList, valueList], containing the optimum parameter set 'optimumParameter', optimum value 'optimumValue', the whole list of parameters parameterList with according objective values 'valueList'
- **author**: Stefan Holzinger, Johannes Gerstmayr
- **notes**: This function is still under development and shows an experimental state! There are currently unused arguments of scipy.optimize.minimize(): Detailed information can be found in the documentation of scipy.optimize.minimize().

For examples on Minimize see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `minimizeExample.py` (Ex), `shapeOptimization.py` (Ex)

---

def **ComputeSensitivities** (*parameterFunction*, *parameters*, *scaledByReference*= False, *debugMode*= False, *addComputationIndex*= False, *useMultiProcessing*= False, *showProgress*= True, *parameterFunctionData*= dict(), ***kwargs*)

- **function description**:

  Perform a sensitivity analysis by successively calling the function parameterFunction(parameterList[i]) with a one at a time variation of parameters in the defined increments.

  e.g., parameterList[0] =['mass':13, 'stiffness':12000] to be computed and returns a value or a list of values which is then stored for each parameter

- **input**:

parameterFunction: function, which takes the form parameterFunction(parameterDict) and which returns one or more output values for which the sensitivity is calculated

parameters: given as a dictionary, consist of name and tuple of (begin, Variation steps, numberOfValues) e.g. 'mass':(10,0.01,5), for a reference mass of 10, incremented by 0.01*10 and using 5 steps in negative and positive, doing 10 steps in total

scaledByReference: if true multiplies the sensitivities with the corresponding reference parameters, so that the sensitivity resembles a change relative to the reference value

debugMode: if True, additional information is shown

addComputationIndex: if True, key 'computationIndex' is added to every parameterDict in the call to parameterFunction(), which allows to generate independent output files for every parameter etc.

useMultiProcessing: if True, the multiprocessing lib is used for parallelized computation; WARNING: be aware that the function does not check if your function runs independently; DO NOT use GRAPHICS and DO NOT write to same output files, etc.!

showProgress: if True, shows for every iteration the progress bar (requires tqdm library)

resultsFile: if provided, output is immediately written to resultsFile during processing

numberOfThreads: default: same as number of cpus (threads); used for multiprocessing lib;

parameterFunctionData: dictionary containing additional data passed to the parameterFunction inside the parameters with dict key 'functionData'; use this e.g. for passing solver parameters or other settings

- **output**: returns [parameterList, valRef, valuesSorted, sensitivity], parameterList containing the list of dictionaries processed. valRef is the Solution for the reference values paramList[0], valuesSorted contains the results sorted by the dictionary key that was varied in the simulation. The sensitivity contains the calculated sensitivity, where the rows are the corresponding outputparameters, while the columns are the input parameters, thereby the index sensitivity[1,0] is the sensitivity of output parameter 1 with respect to the input parameter 0.
- **author**: Peter Manzl
- **example**:

```
ComputeSensitivities(parameterFunction=ParameterFunction, parameters = {'mass': (
  mRef, 0.01, 3), 'spring': (1000,0.01, 10),}, multiprocessing=True)
```

For examples on ComputeSensitivities see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ComputeSensitivitiesExample.py (Ex)

---

def **PlotOptimizationResults2D** (*parameterList*, *valueList*, *xLogScale*= False, *yLogScale*= False)

- **function description**: visualize results of optimization for every parameter (2D plots)
- **input**:

  parameterList: taken from output parameterList of `GeneticOptimization`, containing a dictinary with lists of parameters

  valueList: taken from output valueList of `GeneticOptimization`; containing a list of floats that result from the objective function

  xLogScale: use log scale for x-axis

  yLogScale: use log scale for y-axis
- **output**: return [figList, axList] containing the corresponding handles; creates a figure for every parameter in parameterList

For examples on PlotOptimizationResults2D see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `geneticOptimizationSliderCrank.py` (Ex), `minimizeExample.py` (Ex), `shapeOptimization.py` (Ex), `geneticOptimizationTest.py` (TM)

---

def **PlotSensitivityResults** (*valRef*, *valuesSorted*, *sensitivity*, *fVar*= None, *strYAxis*= None)

- **function description**: visualize results of Sensitivityanalyis for every parameter (2D plots)
- **input**:

  valRef: The output values of the reference solution

  valuesSorted: The output values of the analysed function sorted by the parameter which was varied

  sensitivity: The sensitivity Matrix calculated by the function `ComputeSensitivities()`

  fVar: The list of variation stepsizes. It is assumed to be 1e-3 if not defined.

  strYAxis: A list of strings to label the plots yAxis
- **output**: return [fig, axs] containing the corresponding handles; creates a subplot for every row in the sensitivity matrix
- **author**: Peter Manzl

For examples on PlotSensitivityResults see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `ComputeSensitivitiesExample.py` (Ex)

7.19 Module: rigidBodyUtilities

Advanced utility/mathematical functions for reference frames, rigid body kinematics and dynamics. Useful Euler parameter and Tait-Bryan angle conversion functions are included. A class for rigid body inertia creating and transformation is available.

Author: Johannes Gerstmayr, Stefan Holzinger (rotation vector and Tait-Bryan angles)

Date: 2020-03-10 (created)

def **ComputeOrthonormalBasisVectors** (*vector0*)

- **function description**: compute orthogonal basis vectors (normal1, normal2) for given vector0 (non-unique solution!); the length of vector0 must not be 1; if vector0 == [0,0,0], then any normal basis is returned
- **output**: returns [vector0normalized, normal1, normal2], in which vector0normalized is the normalized vector0 (has unit length); all vectors in numpy array format

---

def **ComputeOrthonormalBasis** (*vector0*)

- **function description**: compute orthogonal basis, in which the normalized vector0 is the first column and the other columns are normals to vector0 (non-unique solution!); the length of vector0 must not be 1; if vector0 == [0,0,0], then any normal basis is returned
- **output**: returns A, a rotation matrix, in which the first column is parallel to vector0; A is a 2D numpy array

---

def **GramSchmidt** (*vector0*, *vector1*)

- **function description**: compute Gram-Schmidt projection of given 3D vector 1 on vector 0 and return normalized triad (vector0, vector1, vector0 x vector1)

For examples on GramSchmidt see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `ACFtest.py` (TM), `sliderCrank3Dbenchmark.py` (TM), `sliderCrank3Dtest.py` (TM)

---

def **Skew** (*vector*)

- **function description**: compute skew symmetric 3x3-matrix from 3x1- or 1x3-vector

For examples on Skew see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- leggedRobot.py (Ex), mobileMecanumWheelRobotWithLidar.py (Ex), carRollingDiscTest.py (TM), createRollingDiscPenaltyTest.py (TM), explicitLieGroupIntegratorPythonTest.py (TM), explicitLieGroupIntegratorTest. (TM), heavyTop.py (TM), laserScannerTest.py (TM), ...

---

def **Skew2Vec** (*skew*)

- **function description**: convert skew symmetric matrix m to vector

For examples on Skew2Vec see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- serialRobotInverseKinematics.py (Ex)

---

def **ComputeSkewMatrix** (*v*)

- **function description**: compute skew matrix from vector or matrix; used for ObjectFFRF and CMS implementation
- **input**: a vector v in np.array format, containing 3*n components or a matrix with m columns of same shape
- **output**: if v is a vector, output is (3*n x 3) skew matrix in np.array format; if v is a (n x m) matrix, the output is a (3*n x m) skew matrix in np.array format

For examples on ComputeSkewMatrix see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- objectFFRFTest.py (TM)

---

def **EulerParameters2G** (*eulerParameters*)

- **function description**: convert Euler parameters (ep) to G-matrix (=$\partial\omega/\partial\mathbf{p}_t$)

- **input**: vector of 4 eulerParameters as list or np.array
- **output**: 3x4 matrix G as np.array

---

def **EulerParameters2GLocal** (*eulerParameters*)

- **function description**: convert Euler parameters (ep) to local G-matrix ($=\partial^b \boldsymbol{\omega} / \partial \mathbf{p}_t$)
- **input**: vector of 4 eulerParameters as list or np.array
- **output**: 3x4 matrix G as np.array

For examples on EulerParameters2GLocal see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `objectFFRFTest.py` (TM), `rigidBodyAsUserFunctionTest.py` (TM)

---

def **EulerParameters2RotationMatrix** (*eulerParameters*)

- **function description**: compute rotation matrix from eulerParameters
- **input**: vector of 4 eulerParameters as list or np.array
- **output**: 3x3 rotation matrix as np.array

For examples on EulerParameters2RotationMatrix see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `ROSMobileManipulator.py` (Ex), `stiffFlyballGovernor2.py` (Ex), `stiffFlyballGovernor.py` (TM)

---

def **RotationMatrix2EulerParameters** (*rotationMatrix*)

- **function description**: compute Euler parameters from given rotation matrix
- **input**: 3x3 rotation matrix as list of lists or as np.array
- **output**: vector of 4 eulerParameters as np.array

For examples on RotationMatrix2EulerParameters see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- NGsolvePistonEngine.py (Ex), stiffFlyballGovernor2.py (Ex), perf3DRigidBodies.py (TM), rightAngleFrame.py (TM), stiffFlyballGovernor.py (TM)

---

def **AngularVelocity2EulerParameters_t** (*angularVelocity*, *eulerParameters*)

- **function description**:

  compute time derivative of Euler parameters from (global) angular velocity vector

  note that for Euler parameters $\mathbf{p}$, we have $\boldsymbol{\omega} = \mathbf{G}\dot{\mathbf{p}} \Longrightarrow \mathbf{G}^T\boldsymbol{\omega} = \mathbf{G}^T \cdot \mathbf{G} \cdot \dot{\mathbf{p}} \Longrightarrow \mathbf{G}^T\mathbf{G} = 4(\mathbf{I}_{4\times4} - \mathbf{p} \cdot \mathbf{p}^T)\dot{\mathbf{p}} = 4(\mathbf{I}_{4x4})\dot{\mathbf{p}}$

- **input**:

  angularVelocity: 3D vector of angular velocity in global frame, as lists or as np.array

  eulerParameters: vector of 4 eulerParameters as np.array or list

- **output**: vector of time derivatives of 4 eulerParameters as np.array

---

def **RotationVector2RotationMatrix** (*rotationVector*)

- **function description**: rotaton matrix from rotation vector, see appendix B in [58]
- **input**: 3D rotation vector as list or np.array
- **output**: 3x3 rotation matrix as np.array
- **notes**: gets inaccurate for very large rotations, ϕ
  $gg2 * \pi$

For examples on RotationVector2RotationMatrix see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- chatGPTupdate.py (Ex), chatGPTupdate2.py (Ex), stiffFlyballGovernor2.py (Ex), universalJoint.py (Ex), createFunctionsTest.py (TM), explicitLieGroupMBSTest.py (TM), jointArgsTest.py (TM), stiffFlyballGovernor.py (TM), ...

---

def **RotationMatrix2RotationVector** (*rotationMatrix*)

- **function description**: compute rotation vector from rotation matrix
- **input**: 3x3 rotation matrix as list of lists or as np.array
- **output**: vector of 3 components of rotation vector as np.array

For examples on RotationMatrix2RotationVector see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `explicitLieGroupMBSTest.py` (TM)

---

def **ComputeRotationAxisFromRotationVector** (*rotationVector*)

- **function description**: compute rotation axis from given rotation vector
- **input**: 3D rotation vector as np.array
- **output**: 3D vector as np.array representing the rotation axis

For examples on ComputeRotationAxisFromRotationVector see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `LieGroupIntegrationUnitTests.py` (TM)

---

def **RotationVector2G** (*rotationVector*)

- **function description**: convert rotation vector (parameters) (v) to G-matrix ($=\partial\boldsymbol{\omega}/\partial\dot{\mathbf{v}}$)
- **input**: vector of rotation vector (len=3) as list or np.array
- **output**: 3x3 matrix G as np.array

---

def **RotationVector2GLocal** (*eulerParameters*)

- **function description**: convert rotation vector (parameters) (v) to local G-matrix ($=\partial^b\boldsymbol{\omega}/\partial\mathbf{v}_t$)
- **input**: vector of rotation vector (len=3) as list or np.array
- **output**: 3x3 matrix G as np.array

---

def **RotXYZ2RotationMatrix** (*rot*)

- **function description**: compute rotation matrix from consecutive xyz rotations (Rots) (Tait-Bryan angles); A=Ax*Ay*Az; rot=[rotX, rotY, rotZ]
- **input**: 3D vector of Tait-Bryan rotation parameters [X,Y,Z] in radiant
- **output**: 3x3 rotation matrix as np.array

For examples on RotXYZ2RotationMatrix see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `InverseKinematicsNumericalExample.py` (Ex), `kinematicTreeAndMBS.py` (Ex), `stiffFlyballGovernor2.py` (Ex), `explicitLieGroupMBSTest.py` (TM), `generalContactImplicit2.py` (TM), `kinematicTreeTest.py` (TM), `stiffFlyballGovernor.py` (TM)

---

def **RotationMatrix2RotXYZ** (*rotationMatrix*)

- **function description**: convert rotation matrix to xyz Euler angles (Tait-Bryan angles); A=Ax*Ay*Az;
- **input**: 3x3 rotation matrix as list of lists or np.array
- **output**: vector of Tait-Bryan rotation parameters [X,Y,Z] (in radiant) as np.array
- **notes**:

  due to gimbal lock / singularity at rot[1] = pi/2, -pi/2, ... the reconstruction of

  `RotationMatrix2RotXYZ( RotXYZ2RotationMatrix(rot) )` may fail, but

  `RotXYZ2RotationMatrix( RotationMatrix2RotXYZ( RotXYZ2RotationMatrix(rot) ) )` works always

For examples on RotationMatrix2RotXYZ see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `serialRobotInteractiveLimits.py` (Ex)

---

def **RotXYZ2G** (*rot*)

- **function description**: compute (global-frame) G-matrix for xyz Euler angles (Tait-Bryan angles) (${}^0\mathbf{G} = \partial {}^0\boldsymbol{\omega} / \partial \dot{\boldsymbol{\theta}}$)

- **input**: 3D vector of Tait-Bryan rotation parameters [X,Y,Z] in radiant
- **output**: 3x3 matrix G as np.array

---

def **RotXYZ2G_t** (*rot*, *rot_t*)

- **function description**: compute time derivative of (global-frame) G-matrix for xyz Euler angles (Tait-Bryan angles) (${}^{0}\mathbf{G} = \partial {}^{0}\boldsymbol{\omega} / \partial \dot{\boldsymbol{\theta}}$)
- **input**:

  rot: 3D vector of Tait-Bryan rotation parameters [X,Y,Z] in radiant

  rot_t: 3D vector of time derivative of Tait-Bryan rotation parameters [X,Y,Z] in radiant/s
- **output**: 3x3 matrix G_t as np.array

---

def **RotXYZ2GLocal** (*rot*)

- **function description**: compute local (body-fixed) G-matrix for xyz Euler angles (Tait-Bryan angles) (${}^{b}\mathbf{G} = \partial {}^{b}\boldsymbol{\omega} / \partial \boldsymbol{\theta}_t$)
- **input**: 3D vector of Tait-Bryan rotation parameters [X,Y,Z] in radiant
- **output**: 3x3 matrix GLocal as np.array

---

def **RotXYZ2GLocal_t** (*rot*, *rot_t*)

- **function description**: compute time derivative of (body-fixed) G-matrix for xyz Euler angles (Tait-Bryan angles) (${}^{b}\mathbf{G} = \partial {}^{b}\boldsymbol{\omega} / \partial \boldsymbol{\theta}_t$)
- **input**:

  rot: 3D vector of Tait-Bryan rotation parameters [X,Y,Z] in radiant

  rot_t: 3D vector of time derivative of Tait-Bryan rotation parameters [X,Y,Z] in radiant/s
- **output**: 3x3 matrix GLocal_t as np.array

---

def **AngularVelocity2RotXYZ_t** (*angularVelocity*, *rotation*)

- **function description**: compute time derivatives of angles RotXYZ from (global) angular velocity vector and given rotation
- **input**:

  angularVelocity: global angular velocity vector as list or np.array

  rotation: 3D vector of Tait-Bryan rotation parameters [X,Y,Z] in radiant
- **output**: time derivative of vector of Tait-Bryan rotation parameters [X,Y,Z] (in radiant) as np.array

---

def **RotXYZ2EulerParameters** (*alpha*)

- **function description**: compute four Euler parameters from given RotXYZ angles, see [30]
- **input**: alpha: 3D vector as np.array containing RotXYZ angles
- **output**:

  4D vector as np.array containing four Euler parameters

  entry zero of output represent the scalar part of Euler parameters

---

def **RotationMatrix2RotZYZ** (*rotationMatrix*, *flip*)

- **function description**: convert rotation matrix to zyz Euler angles; A=Az*Ay*Az;
- **input**:

  rotationMatrix: 3x3 rotation matrix as list of lists or np.array

  flip: argument to choose first Euler angle to be in quadrant 2 or 3.
- **output**: vector of Euler rotation parameters [Z,Y,Z] (in radiant) as np.array
- **author**: Martin Sereinig
- **notes**: tested (compared with Robotics, Vision and Control book of P. Corke)

---

def **RotationMatrixX** (*angleRad*)

- **function description**: compute rotation matrix w.r.t. X-axis (first axis)
- **input**: angle around X-axis in radiant
- **output**: 3x3 rotation matrix as np.array

For examples on RotationMatrixX see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- addPrismaticJoint.py (Ex), addRevoluteJoint.py (Ex), graphicsDataExample.py (Ex), mobileMecanumWheelRobotWithLidar.py (Ex), NGsolveCraigBampton.py (Ex), ... , generalContactCylinderTest.py (TM), generalContactCylinderTrigsTest.py (TM), generalContactFrictionTests.py (TM), ...

---

def **RotationMatrixY** (*angleRad*)

- **function description**: compute rotation matrix w.r.t. Y-axis (second axis)
- **input**: angle around Y-axis in radiant
- **output**: 3x3 rotation matrix as np.array

For examples on RotationMatrixY see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- addPrismaticJoint.py (Ex), addRevoluteJoint.py (Ex), bicycleIftommBenchmark.py (Ex), leggedRobot.py (Ex), mobileMecanumWheelRobotWithLidar.py (Ex), ... , bricardMechanism.py (TM), complexEigenvaluesTest.py (TM), computeODE2AEeigenvaluesTest.py (TM), ...

---

def **RotationMatrixZ** (*angleRad*)

- **function description**: compute rotation matrix w.r.t. Z-axis (third axis)
- **input**: angle around Z-axis in radiant
- **output**: 3x3 rotation matrix as np.array

For examples on RotationMatrixZ see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- addPrismaticJoint.py (Ex), addRevoluteJoint.py (Ex), bicycleIftommBenchmark.py (Ex), chainDriveExample.py (Ex), fourBarMechanism3D.py (Ex), ... , bricardMechanism.py (TM), carRollingDiscTest.py (TM), complexEigenvaluesTest.py (TM), ...

---

def **HomogeneousTransformation** (A, r)

- **function description**: compute homogeneous transformation (HT) matrix from rotation matrix A and translation vector r

For examples on HomogeneousTransformation see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- humanRobotInteraction.py (Ex), InverseKinematicsNumericalExample.py (Ex), kinematicTreeAndMBS.py (Ex), NGsolveCraigBampton.py (Ex), NGsolvePistonEngine.py (Ex), ...

---

def **HTtranslate** (r)

- **function description**: HT for translation with vector r

For examples on HTtranslate see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- humanRobotInteraction.py (Ex), InverseKinematicsNumericalExample.py (Ex), kinematicTreeAndMBS.py (Ex), kinematicTreePendulum.py (Ex), openAIgymNLinkAdvanced.py (Ex), ... , createKinematicTreeTest.py (TM), kinematicTreeAndMBStest.py (TM), kinematicTreeConstraintTest.py (TM), ...

---

def **HTtranslateX** (x)

- **function description**: HT for translation along x axis with value x

For examples on HTtranslateX see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- kinematicTreeAndMBStest.py (TM)

---

def **HTtranslateY** (*y*)

– **function description**: HT for translation along y axis with value y

For examples on HTtranslateY see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- kinematicTreePendulum.py (Ex), openAIgymNLinkAdvanced.py (Ex), openAIgymNLinkContinuous.py (Ex), kinematicTreeAndMBStest.py (TM), kinematicTreeConstraintTest.py (TM)

---

def **HTtranslateZ** (*z*)

– **function description**: HT for translation along z axis with value z

---

def **HT0** ()

– **function description**: identity HT:

For examples on HT0 see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- humanRobotInteraction.py (Ex), kinematicTreeAndMBS.py (Ex), kinematicTreePendulum.py (Ex), openAIgymNLinkAdvanced.py (Ex), openAIgymNLinkContinuous.py (Ex), ... , kinematicTreeAndMBStest.py (TM), kinematicTreeConstraintTest.py (TM)

---

def **HTrotateX** (*angle*)

– **function description**: HT for rotation around axis X (first axis)

---

def **HTrotateY** (*angle*)

– **function description**: HT for rotation around axis X (first axis)

For examples on HTrotateY see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- kinematicTreeAndMBStest.py (TM)

---

def **HTrotateZ** (*angle*)

- **function description**: HT for rotation around axis X (first axis)

For examples on HTrotateZ see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ROSMobileManipulator.py (Ex), kinematicTreeAndMBStest.py (TM)

---

def **HT2translation** (*T*)

- **function description**: return translation part of HT

For examples on HT2translation see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- kinematicTreeAndMBS.py (Ex), serialRobotFlexible.py (Ex), serialRobotInteractiveLimits.py (Ex), serialRobotInverseKinematics.py (Ex), serialRobotKinematicTree.py (Ex), ... , kinematicTreeAndMBStest.py (TM),
  movingGroundRobotTest.py (TM), serialRobotTest.py (TM), ...

---

def **HT2rotationMatrix** (*T*)

- **function description**: return rotation matrix of HT

For examples on HT2rotationMatrix see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- kinematicTreeAndMBS.py (Ex), kinematicTreeAndMBStest.py (TM)

---

def **InverseHT** (*T*)

- **function description**: return inverse HT such that inv(T)*T = np.eye(4)

For examples on InverseHT see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- serialRobotKinematicTree.py (Ex)

---

def **RotationX2T66** (*angle*)

- **function description**: compute 6x6 coordinate transformation matrix for rotation around X axis; output: first 3 components for rotation, second 3 components for translation! See Featherstone / Handbook of robotics [57]

---

def **RotationY2T66** (*angle*)

- **function description**: compute 6x6 transformation matrix for rotation around Y axis; output: first 3 components for rotation, second 3 components for translation

---

def **RotationZ2T66** (*angle*)

- **function description**: compute 6x6 transformation matrix for rotation around Z axis; output: first 3 components for rotation, second 3 components for translation

---

def **Translation2T66** (*translation3D*)

- **function description**: compute 6x6 transformation matrix for translation according to 3D vector translation3D; output: first 3 components for rotation, second 3 components for translation!

---

def **TranslationX2T66** (*translation*)

- **function description**: compute 6x6 transformation matrix for translation along X axis; output: first 3 components for rotation, second 3 components for translation!

---

def **TranslationY2T66** (*translation*)

- **function description**: compute 6x6 transformation matrix for translation along Y axis; output: first 3 components for rotation, second 3 components for translation!

---

def **TranslationZ2T66** (*translation*)

- **function description**: compute 6x6 transformation matrix for translation along Z axis; output: first 3 components for rotation, second 3 components for translation!

---

def **T66toRotationTranslation** (*T66*)

- **function description**: convert 6x6 coordinate transformation (Plücker transform) into rotation and translation
- **input**: T66 given as 6x6 numpy array
- **output**: [A, v] with 3x3 rotation matrix A and 3D translation vector v

For examples on T66toRotationTranslation see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `kinematicTreeAndMBS.py` (Ex)

---

def **InverseT66toRotationTranslation** (*T66*)

- **function description**: convert inverse 6x6 coordinate transformation (Plücker transform) into rotation and translation
- **input**: inverse T66 given as 6x6 numpy array
- **output**: [A, v] with 3x3 rotation matrix A and 3D translation vector v

---

def **RotationTranslation2T66** (*A*, *v*)

- **function description**: convert rotation and translation into 6x6 coordinate transformation (Plücker transform)
- **input**:

  A: 3x3 rotation matrix A

  v: 3D translation vector v
- **output**: return 6x6 transformation matrix 'T66'

---

def **RotationTranslation2T66Inverse** (*A*, *v*)

- **function description**: convert rotation and translation into INVERSE 6x6 coordinate transformation (Plücker transform)
- **input**:

  A: 3x3 rotation matrix A

  v: 3D translation vector v
- **output**: return 6x6 transformation matrix 'T66'

For examples on RotationTranslation2T66Inverse see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `kinematicTreeAndMBS.py` (Ex)

---

def **T66toHT** (*T66*)

- **function description**: convert 6x6 coordinate transformation (Plücker transform) into 4x4 homogeneous transformation; NOTE that the homogeneous transformation is the inverse of what is computed in function pluho() of Featherstone
- **input**: T66 given as 6x6 numpy array
- **output**: homogeneous transformation (4x4 numpy array)

For examples on T66toHT see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- kinematicTreeAndMBS.py (Ex)

---

def **HT2T66Inverse** (*T*)

- **function description**: convert 4x4 homogeneous transformation into 6x6 coordinate transformation (Plücker transform); NOTE that the homogeneous transformation is the inverse of what is computed in function pluho() of Featherstone
- **output**: input: T66 (6x6 numpy array)

---

def **InertiaTensor2Inertia6D** (*inertiaTensor*)

- **function description**: convert a 3x3 matrix (list or numpy array) into a list with 6 inertia components, sorted as J00, J11, J22, J12, J02, J01

---

def **Inertia6D2InertiaTensor** (*inertia6D*)

- **function description**: convert a list or numpy array with 6 inertia components (sorted as [J00, J11, J22, J12, J02, J01]) (list or numpy array) into a 3x3 matrix (np.array)

For examples on Inertia6D2InertiaTensor see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- rigidBodyAsUserFunctionTest.py (TM)

---

def **StrNodeType2NodeType** (*sNodeType*)

- **function description**: convert string into exudyn.NodeType; call e.g. with 'NodeType.RotationEulerParamete or 'RotationEulerParameters'
- **notes**: function is not very fast, so should be avoided in time-critical situations

---

def **GetRigidBodyNode** (*nodeType*, *position*= [0,0,0], *velocity*= [0,0,0], *rotationMatrix*= [], *rotationParameters*= [], *angularVelocity*= [0,0,0])

- **function description**: get node item interface according to nodeType, using initialization with position, velocity, angularVelocity and rotationMatrix
- **input**:

  nodeType: a node type according to exudyn.NodeType, or a string of it, e.g., 'NodeType.RotationEulerParam (fastest, but additional algebraic constraint equation), 'NodeType.RotationRxyz' (Tait-Bryan angles, singularity for second angle at +/- 90 degrees), 'NodeType.RotationRotationVector' (used for Lie group integration)

  position: reference position as list or numpy array with 3 components (in global/world frame)

  velocity: initial translational velocity as list or numpy array with 3 components (in global/world frame)

  rotationMatrix: 3x3 list or numpy matrix to define reference rotation; use EITHER rotationMatrix=[[...],[...],[...]] (while rotationParameters=[]) or rotationParameters=[...] (while rotationMatrix=[])

  rotationParameters: reference rotation parameters; use EITHER rotationMatrix=[[...],[...],[...]] (while rotationParameters=[]) or rotationParameters=[...] (while rotationMatrix=[])

  angularVelocity: initial angular velocity as list or numpy array with 3 components (in global/-world frame)

- **output**: returns list containing node number and body number: [nodeNumber, bodyNumber]

---

def **AddRigidBody** (*mainSys*, *inertia*, *nodeType*= exu.NodeType.RotationEulerParameters, *position*= [0,0,0], *velocity*= [0,0,0], *rotationMatrix*= [], *rotationParameters*= [], *angularVelocity*= [0,0,0], *gravity*= [0,0,0], *graphicsDataList*= [])

- **function description**: DEPRECATED: adds a node (with str(exu.NodeType. ...)) and body for a given rigid body; all quantities (esp. velocity and angular velocity) are given in global coordinates!
- **input**:

  inertia: an inertia object as created by class RigidBodyInertia; containing mass, COM and inertia

nodeType: a node type according to exudyn.NodeType, or a string of it, e.g., 'NodeType.RotationEulerParam (fastest, but additional algebraic constraint equation), 'NodeType.RotationRxyz' (Tait-Bryan angles, singularity for second angle at +/- 90 degrees), 'NodeType.RotationRotationVector' (used for Lie group integration)

position: reference position as list or numpy array with 3 components (in global/world frame)

velocity: initial translational velocity as list or numpy array with 3 components (in global/world frame)

rotationMatrix: 3x3 list or numpy matrix to define reference rotation; use EITHER rotationMatrix=[[...],[...],[...]] (while rotationParameters=[]) or rotationParameters=[...] (while rotationMatrix=[])

rotationParameters: reference rotation parameters; use EITHER rotationMatrix=[[...],[...],[...]] (while rotationParameters=[]) or rotationParameters=[...] (while rotationMatrix=[])

angularVelocity: initial angular velocity as list or numpy array with 3 components (in global/-world frame)

gravity: if provided as list or numpy array with 3 components, it adds gravity force to the body at the COM, i.e., fAdd = m*gravity

graphicsDataList: list of graphicsData objects to define appearance of body

- **output**: returns list containing node number and body number: [nodeNumber, bodyNumber]
- **notes**: DEPRECATED and will be removed; use MainSystem.CreateRigidBody(...) instead!

---

def **AddRevoluteJoint** (*mbs*, *body0*, *body1*, *point*, *axis*, *useGlobalFrame*= True, *showJoint*= True, *axisRadius*= 0.1, *axisLength*= 0.4)

- **function description**: DEPRECATED (use MainSystem function instead): add revolute joint between two bodies; definition of joint position and axis in global coordinates (alternatively in body0 local coordinates) for reference configuration of bodies; all markers, markerRotation and other quantities are automatically computed
- **input**:

  mbs: the MainSystem to which the joint and markers shall be added

  body0: a object number for body0, must be rigid body or ground object

  body1: a object number for body1, must be rigid body or ground object

  point: a 3D vector as list or np.array containing the global center point of the joint in reference configuration

  axis: a 3D vector as list or np.array containing the global rotation axis of the joint in reference configuration

useGlobalFrame: if False, the point and axis vectors are defined in the local coordinate system of body0

- **output**: returns list [oJoint, mBody0, mBody1], containing the joint object number, and the two rigid body markers on body0/1 for the joint
- **notes**: DEPRECATED and will be removed; use MainSystem.CreateRevoluteJoint(...) instead!

For examples on AddRevoluteJoint see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- openVRengine.py (Ex)

---

def **AddPrismaticJoint** (*mbs*, *body0*, *body1*, *point*, *axis*, *useGlobalFrame*= True, *showJoint*= True, *axisRadius*= 0.1, *axisLength*= 0.4)

- **function description**: DEPRECATED (use MainSystem function instead): add prismatic joint between two bodies; definition of joint position and axis in global coordinates (alternatively in body0 local coordinates) for reference configuration of bodies; all markers, markerRotation and other quantities are automatically computed
- **input**:

  mbs: the MainSystem to which the joint and markers shall be added

  body0: a object number for body0, must be rigid body or ground object

  body1: a object number for body1, must be rigid body or ground object

  point: a 3D vector as list or np.array containing the global center point of the joint in reference configuration

  axis: a 3D vector as list or np.array containing the global translation axis of the joint in reference configuration

  useGlobalFrame: if False, the point and axis vectors are defined in the local coordinate system of body0
- **output**: returns list [oJoint, mBody0, mBody1], containing the joint object number, and the two rigid body markers on body0/1 for the joint
- **notes**: DEPRECATED and will be removed; use MainSystem.CreatePrismaticJoint(...) instead!

For examples on AddPrismaticJoint see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- openVRengine.py (Ex)

7.19.1 CLASS TreeLink (in module rigidBodyUtilities)

class description: helper class for CreateKinematicTree, representing a link on a joint within a kinematic tree

- **example**:

```
link3 = TreeLink(linkInertia = InertiaCuboid(2800, [0.25,0.08,0.08]).Translated
  ([0.125,0,0]),
                 jointType = =exu.JointType.RevoluteZ,
                 parent = 1,
                 graphicsData = graphics.Brick(centerPoint=[0.125,0,0], size
  =[0.25,0.08,0.08],
                                               color=graphics.color.blue),
                 )
```

def **__init__** (*self*, *linkInertia*, *jointType*= exu.JointType.RevoluteZ, *jointHT*= HT0(), *parent*= None, *PDcontrol*= None, *graphicsDataList*= None)

- **classFunction**: initialize inertia
- **input**:

  linkInertia: RigidBodyInertia class, containing mass, inertia, and COM

  jointHT: transformation from previous link to this link's joint

  parent: index to parent link; if parent link is ground, use -1; if all parents in a serial kinematic tree are None, parent indices are computed automatically

  PDcontrol: tuple of PD control parameters

  graphicsData: graphicsDataList link; None automatically adds a suitable graphical object from next joint to this joint; use empty list [] to add no graphics for link

For examples on TreeLink see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- createKinematicTreeTest.py (TM)

7.19.2 CLASS RigidBodyInertia (in module rigidBodyUtilities)

class description: helper class for rigid body inertia (see also derived classes Inertia...). Provides a structure to define mass, inertia and center of mass (COM) of a rigid body. The inertia tensor and center of mass must correspond when initializing the body!

- **notes**: in the default mode, inertiaTensorAtCOM = False, the inertia tensor must be provided with respect to the reference point; otherwise, it is given at COM; internally, the inertia tensor is always with respect to the reference point, not w.r.t. to COM!
- **example**:

```
i0 = RigidBodyInertia(10,np.diag([1,2,3]))
i1 = i0.Rotated(RotationMatrixX(np.pi/2))
i2 = i1.Translated([1,0,0])
```

def **__init__** (*self*, *mass*= 0, *inertiaTensor*= np.zeros([3,3]), *com*= np.zeros(3), *inertiaTensorAtCOM*= False)

- **classFunction**: initialize RigidBodyInertia with scalar mass, 3x3 inertiaTensor (w.r.t. reference point!!!) and center of mass com
- **input**:

  mass: mass of rigid body (dimensions need to be consistent, should be in SI-units)

  inertiaTensor: tensor given w.r.t. reference point, NOT w.r.t. center of mass!

  com: center of mass relative to reference point, in same coordinate system as inertiaTensor

  inertiaTensorAtCOM: bool flag: if False (default), the inertiaTensor has to be provided w.r.t. the reference point; if True, it has to be provided at the center of mass

---

def **__add__** (*self*, *otherBodyInertia*)

- **classFunction**:

  add (+) operator allows adding another inertia information with SAME local coordinate system and reference point!

  only inertias with same center of rotation can be added!
- **example**:

```
J = InertiaSphere(2,0.1) + InertiaRodX(1,2)
```

---

def **__iadd__** (*self*, *otherBodyInertia*)

- **classFunction**:

  += operator allows adding another inertia information with SAME local coordinate system and reference point!

  only inertias with same center of rotation can be added!
- **example**:

```
J = InertiaSphere(2,0.1)
J += InertiaRodX(1,2)
```

---

def **SetWithCOMinertia** (*self*, *mass*, *inertiaTensorCOM*, *com*)

- **classFunction**: set RigidBodyInertia with scalar mass, 3x3 inertiaTensor (w.r.t. com) and center of mass com
- **input**:

  mass: mass of rigid body (dimensions need to be consistent, should be in SI-units)

  inertiaTensorCOM: tensor given w.r.t. reference point, NOT w.r.t. center of mass!

  com: center of mass relative to reference point, in same coordinate system as inertiaTensor

---

def **Inertia** (*self*)

- **classFunction**: returns 3x3 inertia tensor with respect to chosen reference point (not necessarily COM)

---

def **InertiaCOM** (*self*)

- **classFunction**: returns 3x3 inertia tensor with respect to COM

---

def **COM** (*self*)

- **classFunction**: returns center of mass (COM) w.r.t. chosen reference point

---

def **Mass** (*self*)

- **classFunction**: returns mass

---

def **Translated** (*self*, *vec*)

- **classFunction**: returns a RigidBodyInertia with center of mass com shifted by vec; → transforms the returned inertiaTensor to the new center of rotation

---

def **Rotated** (*self*, *rot*)

- **classFunction**: returns a RigidBodyInertia rotated by 3x3 rotation matrix rot, such that for a given J, the new inertia tensor reads Jnew = rot*J*rot.T
- **notes**: only allowed if COM=0 !

---

def **Transformed** (*self*, *HT*)

- **classFunction**: return rigid body inertia transformed by homogeneous transformation HT

---

def **GetInertia6D** (*self*)

- **classFunction**: get vector with 6 inertia components (Jxx, Jyy, Jzz, Jyz, Jxz, Jxy) w.r.t. to reference point (not necessarily the COM), as needed in ObjectRigidBody

---

def **GetTypeName** (*self*)

- **classFunction**: which returns str of type ('InertiaCylinder', 'InertiaCuboid', ...)

---

def **GetSpecialData** (*self*)

– **classFunction**: returns dictionary with further data of inertia, like cylinder radius, etc.

---

def **GetGraphics** (*self*, *color*, *nTiles*= None, *roundness*= None)

– **classFunction**: get graphicsData object from inertia; this simplifies the rigid body creation process and allows to check for consistency; currently does not include HT-rotations!

For examples on RigidBodyInertia see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- bicycleIftommBenchmark.py (Ex), humanRobotInteraction.py (Ex), openAIgymNLinkAdvanced.py (Ex), openAIgymNLinkContinuous.py (Ex), reinforcementLearningRobot.py (Ex), ... , createRollingDiscPenaltyTest.py (TM), rigidBody2Dtest.py (TM), rigidBodyCOMtest.py (TM), ...

7.19.3 CLASS InertiaCuboid(RigidBodyInertia) (in module rigidBodyUtilities)

class description: create RigidBodyInertia with moment of inertia and mass of a cuboid with density and side lengths sideLengths along local axes 1, 2, 3; inertia w.r.t. center of mass, com=[0,0,0]

– **example**:

```
InertiaCuboid(density=1000,sideLengths=[1,0.1,0.1])
```

def **__init__** (*self*, *density*, *sideLengths*)

– **classFunction**: initialize inertia

For examples on InertiaCuboid see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- addPrismaticJoint.py (Ex), addRevoluteJoint.py (Ex), ANCFrotatingCable2D.py (Ex), bungeeJump.py (Ex), camFollowerExample.py (Ex), ... , bricardMechanism.py (TM), carRollingDiscTest.py (TM), complexEigenvaluesTest.py (TM), ...

7.19.4 CLASS InertiaRodX(RigidBodyInertia) (in module rigidBodyUtilities)

class description: create RigidBodyInertia with moment of inertia and mass of a rod with mass m and length L in local 1-direction (x-direction); inertia w.r.t. center of mass, com=[0,0,0]

def **__init__** (*self*, *mass*, *length*)

– **classFunction**: initialize inertia with mass and length of rod

For examples on InertiaRodX see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- fourBarMechanismIftomm.py (TM)

7.19.5 CLASS InertiaMassPoint(RigidBodyInertia) (in module rigidBodyUtilities)

class description: create RigidBodyInertia with moment of inertia and mass of mass point with given 'mass'; inertia w.r.t. center of mass, com=[0,0,0]; note that the inertia tensor gives zero and cannot be directly used in rigid bodies, however, it can be used to be added to another inertia tensor (e.g. to add unbalance)

def **__init__** (*self*, *mass*)

- **classFunction**: initialize inertia with mass of point

For examples on InertiaMassPoint see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `stiffFlyballGovernor2.py` (Ex), `stiffFlyballGovernorKT.py` (Ex), `stiffFlyballGovernor.py` (TM)

7.19.6 CLASS InertiaSphere(RigidBodyInertia) (in module rigidBodyUtilities)

class description: create RigidBodyInertia with moment of inertia and mass of sphere with mass and radius; inertia w.r.t. center of mass, com=[0,0,0]

def **__init__** (*self*, *mass*= None, *radius*= None, *density*= None)

- **classFunction**: initialize inertia with mass and radius of sphere

For examples on InertiaSphere see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `ballBearningModel.py` (Ex), `contactCurveWithLongCurve.py` (Ex), `graphicsDataExample.py` (Ex), `newtonsCradle.py` (Ex), `particleClusters.py` (Ex), ... , `contactSphereSphereTest.py` (TM), `contactSphereSphereTestEAPM.py` (TM), `createFunctionsTest.py` (TM), ...

7.19.7 CLASS InertiaHollowSphere(RigidBodyInertia) (in module rigidBodyUtilities)

class description: create RigidBodyInertia with moment of inertia and mass of hollow sphere with mass (concentrated at circumference) and radius; inertia w.r.t. center of mass, com=0

def **__init__** (*self*, *mass*, *radius*)

- **classFunction**: initialize inertia with mass and (inner==outer) radius of hollow sphere

7.19.8 CLASS InertiaCylinder(RigidBodyInertia) (in module rigidBodyUtilities)

class description: create RigidBodyInertia with moment of inertia and mass of cylinder with density, length and outerRadius; axis defines the orientation of the cylinder axis (0=x-axis, 1=y-axis, 2=z-axis); for hollow cylinder use innerRadius != 0; inertia w.r.t. center of mass, com=[0,0,0]

def **__init__** (*self*, *density*, *length*, *outerRadius*, *axis*, *innerRadius*= 0)

- **classFunction**: initialize inertia with density, length, outer radius, axis (0=x-axis, 1=y-axis, 2=z-axis) and optional inner radius (for hollow cylinder)

For examples on InertiaCylinder see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `ballBearningModel.py` (Ex), `camFollowerExample.py` (Ex), `chainDriveExample.py` (Ex), `gyroStability.py` (Ex), `leggedRobot.py` (Ex), ... , `ANCFbeltDrive.py` (TM), `ANCFgeneralContactCircle.py` (TM), `carRollingDiscTest.py` (TM), ...

7.20 Module: robotics

A library which includes support functions for robotics; the library is built on standard Denavit-Hartenberg Parameters and Homogeneous Transformations (HT) to describe transformations and coordinate systems; import this library e.g. with import exudyn.robotics as robotics

Author: Johannes Gerstmayr

Date: 2020-04-14

Example: New robot model uses the class Robot with class RobotLink; the old dictionary structure is defined in the example in ComputeJointHT for the definition of the 'robot' dictionary.

def **StdDH2HT** (*DHparameters*)

- **function description**: compute homogeneous transformation matrix HT from standard DHparameters=[theta, d, a, alpha]

For examples on StdDH2HT see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `humanRobotInteraction.py` (Ex), `InverseKinematicsNumericalExample.py` (Ex), `serialRobotFlexible.py` (Ex), `serialRobotInteractiveLimits.py` (Ex), `serialRobotKinematicTree.py` (Ex), ... , `movingGroundRobotTest.py` (TM), `serialRobotTest.py` (TM)

---

def **ModDHKK2HT** (*DHparameters*)

- **function description**: compute pre- and post- homogeneous transformation matrices from modified Denavit-Hartenberg DHparameters=[alpha, d, theta, r]; returns [HTpre, HTpost]; HTpre is transformation before axis rotation, HTpost includes axis rotation and everything hereafter; modified DH-Parameters according to Khalil and Kleinfinger, 1986

For examples on ModDHKK2HT see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- serialRobotKinematicTree.py (Ex)

---

def **projectAngleToPMPi** (*q0*)

- **function description**: This function projects an angle in the range $[-min_{float}, +max_{float}]$ fo the range $[-\pi, +\pi]$
- **input**: q0: An angle either as scalar, list or array
- **output**: qProj: The angle projected into the range $[-\pi to \pi]$
- **author**: Peter Manzl

7.20.1 CLASS VRobotLink (in module robotics)

class description: class to define visualization of RobotLink

def **__init__** (*self*, *jointRadius*= 0.06, *jointWidth*= 0.12, *linkWidth*= 0.1, *showMBSjoint*= True, *showCOM*= True, *linkColor*= [0.4,0.4,0.4,1], *graphicsData*= [])

- **classFunction**: initialize robot link with parameters, being self-explaining
- **input**:

  jointRadius: radius of joint to draw

  jointWidth: length or width of joint (depending on type of joint)

  showMBSjoint: if False, joint is not drawn

  linkWidth: width of link for default drawing

  linkColor: color of link for default drawing

  showCOM: if True, center of mass is marked with cube

  graphicsData: list of GraphicsData to represent link; if list is empty, link graphics will be generated from link geometry data; otherwise, drawing will be taken from graphicsData, and only showMBSjoint and showCOM flags will add additional graphics

For examples on VRobotLink see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- humanRobotInteraction.py (Ex), InverseKinematicsNumericalExample.py (Ex), kinematicTreePendulum.py (Ex), openAIgymNLinkAdvanced.py (Ex), openAIgymNLinkContinuous.py (Ex), ... , kinematicTreeAndMBStest.py (TM), kinematicTreeConstraintTest.py (TM), movingGroundRobotTest.py (TM), ...

7.20.2 CLASS RobotLink (in module robotics)

class description: class to define one link of a robot

def **__init__** (*self*, *mass*, *COM*, *inertia*, *localHT*= erb.HT0(), *jointType*= 'Rz', *parent*= -2, *preHT*= erb.HT0(), *PDcontrol*= (None,None), *visualization*= VRobotLink())

- **classFunction**: initialize robot link
- **input**:

  mass: mass of robot link

  COM: center of mass in link coordinate system

  inertia: 3x3 matrix (list of lists / numpy array) containing inertia tensor in link coordinates, with respect to center of mass

  localHT: 4x4 matrix (list of lists / numpy array) containing homogeneous transformation from local joint to link coordinates; default = identity; currently, this transformation is not available in KinematicTree, therefore the link inertia and COM must be transformed accordingly

  preHT: 4x4 matrix (list of lists / numpy array) containing homogeneous transformation from previous link to this joint; default = identity

  jointType: string containing joint type, out of: 'Rx', 'Ry', 'Rz' for revolute joints and 'Px', 'Py', 'Pz' for prismatic joints around/along the respecitive local axes

  parent: for building robots as kinematic tree; use '-2' to automatically set parents for serial robot (on fixed base), use '-1' for ground-parent and any other 0-based index for connection to parent link

  PDcontrol: tuple of P and D control values, defining position (rotation) proportional value P and velocitiy proportional value D

  visualization: VRobotLink structure containing options for drawing of link and joints; see class VRobotLink

---

def **SetPDcontrol** (*self*, *Pvalue*, *Dvalue*)

– **classFunction**: set PD control values for drive of joint related to link using position-proportional value P and differential value (velocity proportional) D

---

def **HasPDcontrol** (*self*)

– **classFunction**: check if contrl is available

---

def **GetPDcontrol** (*self*)

– **classFunction**: get PD control values

For examples on RobotLink see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- humanRobotInteraction.py (Ex), InverseKinematicsNumericalExample.py (Ex), kinematicTreeAndMBS.py (Ex), kinematicTreePendulum.py (Ex), openAIgymNLinkAdvanced.py (Ex), ... , kinematicTreeAndMBStest.py (TM), kinematicTreeConstraintTest.py (TM), movingGroundRobotTest.py (TM), ...

7.20.3 CLASS VRobotTool (in module robotics)

class description: class to define visualization of RobotTool

def **__init__** (*self*, *graphicsData*= [])

– **classFunction**: initialize robot tool with parameters; currently only graphicsData, which is a list of GraphicsData same as in mbs Objects

For examples on VRobotTool see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- humanRobotInteraction.py (Ex), InverseKinematicsNumericalExample.py (Ex), kinematicTreeAndMBS.py (Ex), kinematicTreePendulum.py (Ex), openAIgymNLinkAdvanced.py (Ex), ... , kinematicTreeAndMBStest.py (TM), kinematicTreeConstraintTest.py (TM), movingGroundRobotTest.py (TM), ...

7.20.4 CLASS RobotTool (in module robotics)

class description: define tool of robot: containing graphics and HT (may add features in future)

def **__init__** (*self*, *HT*= erb.HT0(), *visualization*= VRobotTool())

- **classFunction**: initialize robot tool
- **input**:

  HT: 4x4 matrix (list of lists / numpy array) containing homogeneous transformation to transform from last link to tool

  graphicsData: dictionary containing a list of GraphicsData, same as in exudyn Objects

For examples on RobotTool see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- humanRobotInteraction.py (Ex), InverseKinematicsNumericalExample.py (Ex), kinematicTreeAndMBS.py (Ex), kinematicTreePendulum.py (Ex), openAIgymNLinkAdvanced.py (Ex), ... , kinematicTreeAndMBStest.py (TM), kinematicTreeConstraintTest.py (TM), movingGroundRobotTest.py (TM), ...

7.20.5 CLASS VRobotBase (in module robotics)

class description: class to define visualization of RobotBase

def **__init__** (*self*, *graphicsData*= [])

- **classFunction**: initialize robot base with parameters; currently only graphicsData, which is a list of GraphicsData same as in mbs Objects

For examples on VRobotBase see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- humanRobotInteraction.py (Ex), InverseKinematicsNumericalExample.py (Ex), kinematicTreeAndMBS.py (Ex), kinematicTreePendulum.py (Ex), openAIgymNLinkAdvanced.py (Ex), ... , kinematicTreeAndMBStest.py (TM), kinematicTreeConstraintTest.py (TM), movingGroundRobotTest.py (TM), ...

7.20.6 CLASS RobotBase (in module robotics)

class description: define base of robot: containing graphics and HT (may add features in future)

def **__init__** (*self*, *HT*= erb.HT0(), *visualization*= VRobotBase())

- **classFunction**: initialize robot base
- **input**:

HT: 4x4 matrix (list of lists / numpy array) containing homogeneous transformation to transform from world coordinates to base coordinates (changes orientation and position of robot)

graphicsData: dictionary containing a list of GraphicsData, same as in exudyn Objects

For examples on RobotBase see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `humanRobotInteraction.py` (Ex), `InverseKinematicsNumericalExample.py` (Ex), `kinematicTreeAndMBS.py` (Ex), `kinematicTreePendulum.py` (Ex), `openAIgymNLinkAdvanced.py` (Ex), ... , `kinematicTreeAndMBStest.py` (TM), `kinematicTreeConstraintTest.py` (TM), `movingGroundRobotTest.py` (TM), ...

7.20.7 CLASS Robot (in module robotics)

class description: class to define a robot

def **__init__** (*self*, *gravity*= [0,0,-9.81], *base*= RobotBase(), *tool*= RobotTool(), *referenceConfiguration*= [])

- **classFunction**: initialize robot class
- **input**:

  base: definition of base using RobotBase() class

  tool: definition of tool using RobotTool() class

  gravity: a list or 3D numpy array defining gravity

  referenceConfiguration: a list of scalar quantities defining the parameters for reference configuration

---

def **AddLink** (*self*, *robotLink*)

- **classFunction**: add a link to serial robot

---

def **IsSerialRobot** (*self*)

- **classFunction**: return True, if robot is a serial robot

---

def **GetLink** (*self*, *i*)

- **classFunction**: return Link object of link i

---

def **HasParent** (*self*, *i*)

- **classFunction**: True if link has parent, False if not

---

def **GetParentIndex** (*self*, *i*)

- **classFunction**: Get index of parent link; for serial robot this is simple, but for general trees, there is a index list

---

def **NumberOfLinks** (*self*)

- **classFunction**: return number of links

---

def **GetBaseHT** (*self*)

- **classFunction**: return base as homogeneous transformation

---

def **GetToolHT** (*self*)

- **classFunction**: return base as homogeneous transformation

---

def **LinkHT** (*self*, *q*)

– **classFunction**: compute list of homogeneous transformations for every link, using current joint coordinates q; leads to different results for standard and modified DH parameters because link coordinates are different!

---

def **JointHT** (*self*, *q*)

– **classFunction**: compute list of homogeneous transformations for every joint (after rotation), using current joint coordinates q

---

def **COMHT** (*self*, *HT*)

– **classFunction**: compute list of homogeneous transformations HT from base to every COM using HT list from Robot.JointHT(...)

---

def **StaticTorques** (*self*, *HT*)

– **classFunction**: compute list of joint torques for serial robot due to gravity (gravity and mass as given in robot), taking HT from Robot.JointHT()

---

def **Jacobian** (*self*, *HT*, *toolPosition*= [], *mode*= 'all', *linkIndex*= None)

– **classFunction**: compute jacobian for translation and rotation at toolPosition using joint HT; this is using the Robot functions, but is inefficient for simulation purposes

– **input**:

HT: list of homogeneous transformations per joint , as computed by Robot.JointHT(...)

toolPosition: global position at which the jacobian is evaluated (e.g., COM); if empty [], it uses the origin of the last link

mode: 'all'...translation and rotation jacobian, 'trans'...only translation part, 'rot': only rotation part

linkIndex: link index for which the jacobian is evaluated; if linkIndex==None, it uses the last link provided in HT

– **output**: returns jacobian with translation and rotation parts in rows (3 or 6) according to mode, and one column per HT; in the kinematic tree the columns not related to linkIndex remain zero

---

def **CreateKinematicTree** (*self*, *mbs*, *name*= '', *forceUserFunction*= 0)

– **classFunction**:

Add a ObjectKinematicTree to existing mbs from the robot structure inside this robot class;

Joints defined by the kinematics as well as links (and inertia) are transferred to the kinematic tree object;

Current implementation only works for serial robots;

Control can be realized simply by adding PDcontrol to RobotLink structures, then modifying jointPositionOffsetVector and jointVelocityOffsetVector in ObjectKinematicTree; force offsets (e.g., static or dynamic torque compensation) can be added to KinematicTree jointForceVector; more general control can be added by using KinematicTree forceUserFunction;

The coordinates in KinematicTree (as well as jointPositionOffsetVector, etc.) are sorted in the order as the RobotLinks are added to the Robot class;

Note that the ObjectKinematicTree is still under development and interfaces may change.

– **input**:

mbs: the multibody system, which will be extended

name: object name in KinematicTree; transferred to KinematicTree, default = ''

forceUserFunction: defines the user function for computation of joint forces in KinematicTree; transferred to KinematicTree, default = 0

– **output**: the function returns a dictionary containing 'nodeGeneric': generic ODE2 node number ,'objectKinematicTree': the kinematic tree object, 'baseObject': the base object if created, otherwise None; further values will be added in future

---

def **CreateRedundantCoordinateMBS** (*self*, *mbs*, *baseMarker*, *jointSpringDamperUserFunctionList*= [], *jointLoadUserFunctionList*= [], *createJointTorqueLoads*= True, *rotationMarkerBase*= None, *rigidBodyNodeType*= exudyn.NodeType.RotationEulerParameters)

– **classFunction**:

Add items to existing mbs from the robot structure inside this robot class; robot is attached to baseMarker (can be ground object or moving/deformable body);

The (serial) robot is built as rigid bodies (containing rigid body nodes), where bodies represent the links which are connected by joints;

Add optional jointSpringDamperUserFunctionList for individual control of joints; otherwise use PDcontrol in RobotLink structure; additional joint torques/forces can be added via spring damper, using mbs.SetObjectParameter(...) function;

See several Python examples, e.g., `serialRobotTestTSD.py`, in Examples or TestModels;

For more efficient models, use CreateKinematicTree(...) function!

– **input**:

mbs: the multibody system, which will be extended

baseMarker: a rigid body marker, at which the robot will be placed (usually ground); note that the local coordinate system of the base must be in accordance with the DH-parameters, i.e., the z-axis must be the first rotation axis. For correction of the base coordinate system, use rotationMarkerBase

jointSpringDamperUserFunctionList: NOT IMPLEMENTED yet: jointSpringDamperUserFunctionLista list of user functions for actuation of joints with more efficient spring-damper based connector (spring-damper directly emulates PD-controller); uses torsional spring damper for revolute joints and linear spring damper for prismatic joints; can be empty list (no spring dampers); if entry of list is 0, no user function is created, just pure spring damper; parameters are taken from RobotLink PDcontrol structure, which MUST be defined using SetPDcontrol(...) in RobotLink

jointLoadUserFunctionList: DEPRECATED: a list of user functions for actuation of joints according to a LoadTorqueVector userFunction, see serialRobotTest.py as an example; can be empty list

createJointTorqueLoads: DEPRECATED: if True, independently of jointLoadUserFunctionList, joint loads are created; the load numbers are stored in lists jointTorque0List/ jointTorque1List; the loads contain zero torques and need to be updated in every computation step, e.g., using a preStepUserFunction; unitTorque0List/ unitTorque1List contain the unit torque vector for the according body(link) which needs to be applied on both bodies attached to the joint

rotationMarkerBase: add a numpy 3x3 matrix for rotation of the base, in order that the robot can be attached to any rotated base marker; the rotationMarkerBase is according to the definition in GenericJoint; note, that for moving base, the static compensation does not work (base rotation must be updated)

rigidBodyNodeType: specify node type of rigid body node, e.g., exudyn.NodeType.RotationEulerParameters etc.

– **output**: the function returns a dictionary containing per link nodes and object (body) numbers, 'nodeList', 'bodyList', the object numbers for joints, 'jointList', list of load numbers for joint torques (jointTorque0List, jointTorque1List); unit torque vectors in local coordinates of the bodies to which the torques are applied (unitTorque0List, unitTorque1List); springDamperList contains the spring dampers if defined by PDcontrol of links

---

def **GetKinematicTree66** (*self*)

– **classFunction**: export kinematicTree

---

def **GetLinkGraphicsData** (*self*, *i*, *p0*, *p1*, *axis0*, *axis1*, *linkVisualization*)

– **classFunction**: create link GraphicsData (list) for link i; internally used in CreateRedundantCoordinateMBS(...); linkVisualization contains visualization dict of link

---

def **BuildFromDictionary** (*self*, *robotDict*)

– **classFunction**: build robot structre from dictionary; this is a DEPRECATED function, which is used in older models; DO NOT USE

For examples on Robot see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `humanRobotInteraction.py` (Ex), `kinematicTreeAndMBS.py` (Ex), `kinematicTreePendulum.py` (Ex), `openAIgymNLinkAdvanced.py` (Ex), `openAIgymNLinkContinuous.py` (Ex), ... , `kinematicTreeConstraintTest.py` (TM), `movingGroundRobotTest.py` (TM), `serialRobotTest.py` (TM), ...

7.20.8 CLASS InverseKinematicsNumerical() (in module robotics)

class description: This class can be used to solve the inverse kinematics problem using a multibody system by solving the static problem of a serial robot

– **author**: Peter Manzl, Johannes Gerstmayr

– **notes**: still under development; errors in orientations of solution may occure. proviedes mtehods to calculate inverse Kinematics

def **__init__** (*self*, *robot*, *jointStiffness*= 1e0, *useRenderer*= False, *flagDebug*= False, *useAlternativeConstraints*= False)

- **classFunction**: initialize RigidBodyInertia with scalar mass, 3x3 inertiaTensor (w.r.t. reference point!!!) and center of mass com
- **input**:

  robot: robot class

  jointStiffness: the stiffness used for the robot's model joints

  useRenderer: when solving the inverse kinematics the renderer is used to show the starting/end configuration of the robot using the graphics objects definded in the robot object
- **author**: Peter Manzl

---

def **GetCurrentRobotHT** (*self*)

- **classFunction**:

  Utility function to get current Homogeneous transformation of the robot to check inverse Kinematics solution

  ** output:

  T: 4x4 homogeneous Transformation matrix of the current TCP pose

---

def **InterpolateHTs** (*self*, *T1*, *T2*, *rotStep*= np.pi/16, *minSteps*= 1)

- **classFunction**:
- **input**:

  T1: 4x4 homogeneous transformation matrix representing the first Pose

  T2: 4x4 homogeneous transformation matrix representing the second Pose

  rotStep: the max. size of steps to take for the orientation

  minSteps: minimum number of substeps to interpolate
- **output**: T: a List of homogeneous Transformations for each step between
- **author**: Peter Manzl

– **notes**: still under development; interpolation may be changed to using logSE3

---

def **SolveSafe** (*self*, *T*, *q0*= None)

– **classFunction**:

This Method can be used to solve the inverse kinematics problem by solving

the static problem of a serial robot using steps to interpolate between start and end position close to the function Solve.

This helps the function Solve() to find the correct solutions.

– **input**:

T: the 4x4 homogeneous transformation matrix representing the desired position and orientation of the Endeffector

q0: The configuration (joint angles/positions) of the robot from which the numerical methods start so calculate the solution; q0=None indicates that the stored solution (from model or previous solution) shall be used for initialization

– **output**:

[q, success]; q: The solution for the joint angles in which the robot's tool center point (TCP) reaches the desired homogeneous transformation matrix T; success=False indicates that all trials for inverse kinematics failed, leading to q=None

success: flag to indicate if method was successful

– **author**: Peter Manzl, Johannes Gerstmayr

– **notes**: still under development; errors in orientations of solution may occure. works similar to ikine_LM function of the robotics toolbox from peter corke

---

def **Solve** (*self*, *T*, *q0*= None)

– **classFunction**:

This Method can be used to solve the inverse kinematics problem by solving

the static problem of a serial robot using steps to interpolate between start and end position close to the function Solve.

T his helps the fucntion Solve to find the correct solutions.

– **input**:

T: the 4x4 homogeneous transformation matrix representing the desired position and orientation of the Endeffector

q0: The configuration (joint angles/positions) of the robot from which the numerical methods start so calculate the solution; q0=None indicates that the stored solution (from model or previous solution) shall be used for initialization

- **output**: [q, success]; q: The solution for the joint angles in which the robot's tool center point (TCP) reaches the desired homogeneous transformation matrix T; success=False indicates that all trials for inverse kinematics failed, leading to q=None
- **author**: Peter Manzl, Johannes Gerstmayr
- **notes**: still under development; errors in orientations of solution may occure. works similar to ikine_LM function of the robotics toolbox from peter corke

For examples on InverseKinematicsNumerical see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `InverseKinematicsNumericalExample.py` (Ex), `serialRobotInverseKinematics.py` (Ex)

7.20.9 Module: robotics.rosInterface

This interface collects interfaces and functionality for ROS comunication This library is under construction (2023-05); To make use of this libraries, you need to install ROS (ROS1 noetic) including rospy Please consider following workflow: make sure to have a working ROS1-NOETIC installation, ROS2 is not supported yet tested only with ROS1-NOETIC, ubuntu 20.04, and Python 3.8.10 you find all ROS1 installation steps on: http://wiki.ros.org/noetic/Installation/Ubuntu Step 1.4 we recommend to install: (sudo apt install ros-noetic-desktop) Check the installation of the turtlesim package (rosrun turtlesim turtlesim_node) if not installed: sudo apt install ros-noetic-turtlesim use a catkin workspace and build a ROS1 Package Follow instructions on: http://wiki.ros.org/ROS/Tutorials (recommend go trough step 1 to 6) Minimal example to use: create catkin workspace: mkdir -p /catkin_ws/src cd /catkin_ws catkin_make source devel/setup.bash build ROS package: cd /catkin_ws/src catkin_create_pkg my_pkg_name rospy roscpp std_msgs geometry_msgs sensor_msgs build catkin workspace and sourcing setup file cd /catkin_ws cakin_make source /catkin_ws/devel/setup.bash for more functionality see also: ROSExampleMassPoint.py, ROSExampleBringup.launch, ROSExampleControlVelocity.py

Author: Martin Sereinig, Peter Manzl

Date: 2023-05-31 (created)

7.20.9.1 CLASS ROSInterface (in module robotics.rosInterface)

class description: interface super class to establish a ROS Exudyn interface see specific class functions which can be used and extended by inheritance with class MyClass(ROSInterface)

- **author**: Martin Sereinig, Peter Manzl
- **notes**: some inspiration can be taken from

def **InitPublisher** (*self*, *pubTopicName*= '', *pubType*= Empty, *queueSize*= 10)

- **classFunction**: function to create a publisher
- **input**:

  pubTopicName: topic name to publish, actual topic will be /exudyn/pubTopicName

  pubType: data type used in topic

  queSize: length of queue to hold messages, should be as small as sending frequency (= simulation sample time)
- **author**: Martin Sereinig
- **notes**:

  find msgs types here

  http://docs.ros.org/en/melodic/api/std_msgs/html/index-msg.html
- **example**:

```
s:
        publisher for poses, pubType = PoseStamped,
        publisher for system data, pubType = Float64MultiArray,
        publisher for filtered force, pubType = WrenchStamped,
        publisher for velocities, pubType = Twist,
```

---

def **ExuCallbackGeneric** (*self*, *subTopicName*, *data*)

- **classFunction**: function to create a generic callback function for a subscriber
- **input**:

  topic: topic name generated by init Subscriber

  data: data structure for regarding individual topic
- **author**: Peter Manzl

---

def **InitSubscriber** (*self*, *subTopicNameSpace*, *subTopicName*, *subType*)

- **classFunction**: function to create a subscriber
- **input**:

  subTopicNameSpace: topic namespace: 'exudyn/'

  subTopicName: topic name to subscribe

  subType: data type for topic to subscribe
- **author**:

  Peter Manzl

  ***note*: callback function will be automatic generated for each subscriber, depending on subTopicName. Data will be found under self.subTopicName

---

def **CheckROSversion** (*self*)

- **classFunction**: check the current used ROS version
- **author**:

  Martin Sereinig

  ***note*: just supports ROS1, ROS2 support will be given in future releases

---

def **PublishPoseUpdate** (*self*, *mbs*, *tExu*, *getData*= 'node')

- **classFunction**:

  Example method to be called once per frame/control cycle in Exudyn PreStepUserFunction

  ***note*: reads sensor values, creates message, publish and subscribe to ROS
- **input**:

  mbs: mbs (exudyn.exudynCPP.MainSystem), multi-body simulation system from exudyn

  tExu: tExu (float), elapsed time since simulation start

  getData: getData (string), get pose information from 'node' or from 'sensor'
- **author**: Martin Sereinig
- **notes**:

  reads sensor values, creates message, publish and subscribe to ROS

publishing each and every step is too much, this would slow down the connection

thus: publish every few seconds, only

furthermore, as vrInterface is only updating the graphics with f=60Hz, we don't have to update

system state every 1ms, so with f=1000Hz. Instead f=60Hz equivalents to update every 1/60=17ms

timing variable to know when to send new command to robot or when to publish new mbs system state update

---

def **PublishTwistUpdate** (*self*, *mbs*, *tExu*, *getData*= 'node')

- **classFunction**:

  Example method to be called once per frame/control cycle in Exudyn PreStepUserFunction

  ***note*: reads sensor values, creates message, publish and subscribe to ROS

- **input**:

  mbs: mbs (exudyn.exudynCPP.MainSystem), multi-body simulation system from exudyn

  tExu: tExu (float), elapsed time since simulation start

  getData: getData (string), get pose information from 'node' or from 'sensor'

- **author**: Martin Sereinig

- **notes**: reads sensor values, creates message, publish and subscribe to ROS

---

def **PublishSystemStateUpdate** (*self*, *mbs*, *tExu*)

- **classFunction**: method to be send system state data once per frame/control cycle in Exudyn PreStepUserFunction

- **input**:

  mbs: mbs (exudyn.exudynCPP.MainSystem), multi-body simulation system from exudyn

  tExu: tExu (float), simulation time

  systemStateData: systemStateData (list), full Exudyn SystemState

- **author**:

  Martin Sereinig

  ***note*: collects important exudyn system data and send it to ros-topic

7.20.10 Module: robotics.future

The future module contains functionality which is currently under development and will be moved in other robotics libraries in future

Date: 2023-03-27

def **MakeCorkeRobot** (*robotDic*)

- **function description**: makeCorkeRobot, creates robot using the peter corke toolbox using standard (stdDH) or modified (modKKDH) Denavid Hartenberg parameters
- **input**:

  robotDic: robot dictionary by exudyn robotic models

  dhpara: stDH for standard DH parameter, modKKDH for modified DH parameter
- **output**: serial robot object by corke
- **author**: Martin Sereinig
- **notes**:

  DH Parameter Information:

  stdH = [theta, d, a, alpha] with Rz(theta) * Tz(d) * Tx(a) * Rx(alpha)

  modDH = [alpha, dx, theta, rz] with

  used by Corke and Lynch: Rx(alpha) * Tx(a) * Rz(theta) * Tz(d)

  used by Khali: Rx(alpha) * Tx(d) * Rz(theta) * Tz(r)

  Important note: d(khali)=a(corke) and r(khali)=d(corke)

---

def **ComputeIK3R** (*robotDic*, *HT*)

- **function description**: calculates the analytical inverse kinematics for 3R elbow type serial robot manipulator
- **input**:

  robotDic: robot dictionary

  HT: desired position and orientation for the end effector as 4x4 homogeneous transformation matrix as list of lists or np.array
- **output**:

  solutions, list of lists with posible joint angles [q1,q2,q3] (in radiant)

to achive the desired position (4 posible solutions,schoulder left/right, ellbow up/down) in following order: left/down, left/up, right/up, right/down

- **author**: Martin Sereinig
- **notes**: only applicable for standard Denavit-Hartenberg parameters
- **status**: testet with various configurations and joint angels

---

def **ComputeIKPuma560** (*robotDic*, *HT*)

- **function description**: calculates the analytical inverse kinematics for Puma560 serial 6R robotDic manipulator
- **input**:

  robotDic: robotDictionary

  HT: desired position and orientation for the end effector as 4x4 homogeneous transformation matrix as list of lists or np.array
- **output**:

  qSolutions, list of lists with posible joint angles [q1,q2,q3,q4,q5,q6] (in radiant)

  to achive the desired position and orientation (8 posible solutions,schoulder left/right, ellbow up/down, wrist flipped/notflipped (rotated by pi))

  left/down/notflipped, left/down/flipped, left/up/notflipped, left/up/flipped, right/up/notflipped, right/up/flipped, right/down/notflipped, right/down/flipped
- **author**: Martin Sereinig
- **notes**: Usage for different manipulators with sperical wrist posible, only applicable for standard Denavit-Hartenberg parameters
- **status**: tested (compared with robotDiccs, Vision and Control book of P. Corke

---

def **ComputeIKUR** (*robotDic*, *HTdes*)

- **function description**: calculates the analytical inverse kinematics for UR type serial 6R robot manipulator without sperical wrist
- **input**:

robotDic: robot dictionary

HT: desired position and orientation for the end effector as 4x4 homogeneous transformation matrix as list of lists or np.array

- **output**:

  solutions, list of lists with posible joint angles [q1,q2,q3,q4,q5,q6] (in radiant)

  to achive the desired position and orientation (8 posible solutions,schoulder left/right, ellbow up/down, wrist flipped/notflipped (rotated by pi))

  [left/down/notflipped, left/down/flipped, left/up/notflipped, left/up/flipped, right/up/notflipped, right/up/flipped, right/down/notflipped, right/down/flipped]

- **author**: Martin Sereinig

- **notes**: Usage for different manipulators without sperical wrist posible UR3,UR5,UR10, only applicable for standard Denavit-Hartenberg parameters

- **status**: under development, works for most configurations, singularities not checked -> Zero-Configuration not working

7.20.11 Module: robotics.models

This module contains robotics models; They can be imported by simply calling the functions, which return the according robot dictionary; the library is built on Denavit-Hartenberg Parameters and Homogeneous Transformations (HT) to describe transformations and coordinate systems

Date: 2021-01-10

def **Manipulator4Rsimple** ()

- **function description**: generate 4R manipulator as myRobot dictionary, settings are done in function

- **output**: myRobot dictionary

- **author**: Martin Sereinig

- **notes**: the 4th joint is used to simulate a paralell kinematics manipulator

---

def **Manipulator3RSimple** ()

- **function description**: generate 3R manipulator as myRobot dictionary, settings are done in function

- **output**: myRobot dictionary
- **author**: Martin Sereinig
- **notes**:

  DH-parameters: [theta, d, a, alpha], according to P. Corke

  Values according to WÃ¶rnle simple example with l1=0

  d=[h1 0 0];

  theta=[beta1 beta2 beta3];

  a=[l1 l2 l3];

  alpha=[pi/2 0 0];

---

def **ManipulatorPANDA** ()

- **function description**: generate Franka Emika Panda manipulator as myRobot dictionary, settings are done in function
- **output**: myRobot dictionary
- **author**: Martin Sereinig
- **notes**:

  all Parameter according to Gaz et. al [15]

  DH-parameters(std): [theta, d, a, alpha], according to P. Corke

  Standard DH Parameters, masses, inertias and com according P.Corke and Gaz et. al (they working with modified DH parameter)

  changes to standard DH Parameter checked with P.Corke toolbox

For examples on ManipulatorPANDA see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `InverseKinematicsNumericalExample.py` (Ex), `serialRobotInverseKinematics.py` (Ex)

---

def **ManipulatorUR5** ()

- **function description**: generate UR5 manipulator as myRobot dictionary, settings are done in function

- **output**: myRobot dictionary
- **author**: Martin Sereinig
- **notes**:

  define myRobot kinematics, UR5 Universal Robotics,

  Standard DH-parameters: [theta, d, a, alpha], according to P. Corke,

  Links modeld as cylindrical tubes, Inertia from Parham M. Kebria2016 / Kuefeta2014

For examples on ManipulatorUR5 see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `InverseKinematicsNumericalExample.py` (Ex), `ROSMobileManipulator.py` (Ex), `serialRobotInverseKinematics.py` (Ex)

---

def **ManipulatorPuma560** ()

- **function description**: generate puma560 manipulator as myRobot dictionary, settings are done in function
- **output**: myRobot dictionary
- **author**: Martin Sereinig
- **notes**:

  std DH-parameters: [theta, d, a, alpha], according to P. Corke page 138,

  puma p560 limits, taken from Corke Visual Control of Robots

For examples on ManipulatorPuma560 see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `humanRobotInteraction.py` (Ex), `InverseKinematicsNumericalExample.py` (Ex), `serialRobotInverseKinematics.py` (Ex), `serialRobotKinematicTreeDigging.py` (Ex)

---

def **LinkDict2Robot** (*robotLinkDict*, *robotClass*= None)

- **function description**: generate serial manipulator as robotClass object from robotLinkDict
- **input**:

robotClass: robot class object from roboticsCore; if robotClass is provided, gravity, tool and base are used from there

robotLinkDict: list of robot links generated by manipulator import for individual robot dictionary

- **output**: updated robot class
- **author**: Martin Sereinig
- **notes**:

  DH Parameter Information

  stdH = [theta, d, a, alpha] with Rz(theta) * Tz(d) * Tx(a) * Rx(alpha)

  modDH = [alpha, dx, theta, rz] with

  used by Corke and Lynch: Rx(alpha) * Tx(a) * Rz(theta) * Tz(d)

  used by Khali: Rx(alpha) * Tx(d) * Rz(theta) * Tz(r)

  Important note: d(khali)=a(corke) and r(khali)=d(corke)

For examples on LinkDict2Robot see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `ROSMobileManipulator.py` (Ex), `serialRobotInverseKinematics.py` (Ex)

---

def **LinkDictModDHKK2Robot** (*robotLinkDict*, *robotClass*= None)

- **function description**: special test function to generate serial manipulator as robotClass object from robotLinkDict using inertia parameters defined in stdDH coordinates, but creating robot from modDHKK; will be ERASED in future
- **input**:

  robotLinkDict: list of robot links generated by manipulator import for individual robot dictionary

  robotClass: robot class object from roboticsCore; if robotClass is provided, gravity, tool and base are used from there
- **output**: updated robot class
- **author**: Martin Sereinig
- **notes**: DEPRECATED; function uses modDHKK in robotLinkDict for creation, transforms inertia parameters; should only be used for testing!

7.20.12 Module: robotics.mobile

The utilities contains functionality for mobile robots based on the EXUDYN example MecanumWheel RollingDiscPenality specific friction angle of rolling disc is used to model rolls of mecanum wheels

Author: Martin Sereinig, Peter Manzl and Johannes Gerstmayr

Date: 2021-10-01 Updated: 2023-09-15

Notes: formulation is still under development

def **MobileRobot2MBS** (*mbs*, *mobileRobot*, *markerGround*, *flagGraphicsRollers*= True, **args*, ***kwargs*)

- **function description**:

  add items to existing mbs to build up a mobile robot platform,

  there are options that can be passed as args / kwargs, which can contains options as described below.

  The robot platform is built out of rigid bodies where the wheels can be modeled as rolling discs

  (mecanum wheel x/o configuration) or with a detailed mecanum wheel simulation approach

- **input**:

  mbs: the multibody system which will be extended

  markerGround: a rigid body marker, at which the robot will be placed (usually ground)

  mobileRobot: a dictionary including all information about the mobile robot platform

- **output**:

  the function returns a dictionary containing nodes, body, object and marker numbers of individual mobile robot parts

  nPlatformList, bPlatformList, oPlatformList, mPlatformList; nodes, bodies, objects and marker of the platform [nPlattform] [bPlattform] [oPlattform] []

  oAxisList, mAxlesList; objects and marker of the axles [a1, a2, a3, a4]

  nWheelsList, bWheelsList, oRollingDiscsList, mWheelsList; nodes, bodys, objects and markers of the four wheels [w1, w2, w3, w4]

- **notes**: for coordinate system, see Python function definition

For examples on MobileRobot2MBS see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ROSMobileManipulator.py (Ex)

def **Generatrix2Polynomial** (*param*, *GeneratrixFunction*, *tol*= 1e-14, *nFit*= 101, *nTest*= 1001)

- **function description**: create a polynomial describing a generatrix function
- **input**: param: list containing data (lRoll, aPoly, ...)
- **author**:

  Peter Manzl

  ***note*: create and fit a polynomial of an order high enough to approximate the given GeneratrixFunction

  with a given tolerance. The error is measured as the Chebyshev distance.

---

def **GeneratrixRoll** (*u*, *param*)

- **function description**: generatrix function for a roll of a Mecanum wheel
- **input**:

  u: parameter, max. +- pi/2

  param['r']: radius of the associated Mecanum wheel

  param['delta']: angle of the rolls rotation axis to the wheels rotation axis

  param['dRoll']: smallest distance of roll axis to the wheel axis
- **output**:

  x and y values for the function in the local frame. The rotation around the

  local x-yxis creates the surface of the roll.
- **author**: Peter Manzl
- **notes**:

  parametric equation, x,y are the generatrix of the roll in

  its local frame with the axis of rotation x, see [25].

---

def **FunDiffPoly** (*x*, *a*)

- **function description**: calculates the derivative of the polynomial $a0 * x^n + ...$
- **input**:

x: value at which the polynomial is evaluated

a: coefficients

- **output**: f:
- **author**:

  Peter Manzl

  ***note*: helper function polynomial describing a generatrix function

---

def **FunDDiffPoly** (*x*, *a*)

- **function description**: calculates the second derivative of a polynomial
- **input**:

  x: value at which the polynomial is evaluated

  a: coefficients

- **output**: f:
- **author**:

  Peter Manzl

  ***note*: helper function polynomial describing a generatrix function

7.20.12.1 CLASS MobileKinematics (in module robotics.mobile)

class description: calculate 4 wheel velocities for a mecanum wheel driven platform with given platform velocities

- **author**: Peter Manzl, Johannes Gerstmayr
- **notes**: still under development; wheel axis is mounted at y-axis; positive angVel rotates CCW in x/y plane viewed from top; for coordinate system, see Python class definition

def **__init__** (*self*, *R*, *lx*, *ly*, *flagAdjusted*= False, *lcx*= 0, *lcy*= 0, *wheeltype*= 0)

- **classFunction**: initialize mobileKinematics class
- **input**:

  R: wheel radius

lx: wheel track width

ly: wheel base

wheeltype: 1=x-config (bad), 0=o-config (good)

- **author**: Peter Manzl

---

def **GetWheelVelocities** (*self*, *vDes*)

- **classFunction**: calculate wheel velocities from Cartesian velocities
- **input**:

  vDes: desired velocity [vx, vy, omega] in the robot's local frame

  vx: platform translational velocity in local x direction

  vy: platform translational velocity in local y direction

  omega: platform rotational velocity around local z axis
- **output**: w: wheel velocities w=[w0,w1,w2,w3]
- **author**: Peter Manzl

---

def **GetCartesianVelocities** (*self*, *w*)

- **classFunction**: calculate Cartesian velocities from wheel velocities
- **input**: w: wheel velocities w=[w0,w1,w2,w3]
- **output**:

  v: Cartesian velocity [vx, vy, omega] in the robot's local frame

  vx: platform translational velocity in local x direction

  vy: platform translational velocity in local y direction

  omega: platform rotational velocity around local z axis
- **author**: Peter Manzl

For examples on MobileKinematics see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `ROSMobileManipulator.py` (Ex)

7.20.13 Module: robotics.motion

functionality for motion including generation of trajectories with acceleration profiles, path planning and motion

Author: Johannes Gerstmayr

Date: 2022-02-16

7.20.13.1 CLASS ProfileConstantAcceleration (in module robotics.motion)

class description: class to create a constant acceleration (optimal) PTP trajectory; trajectory ignores global max. velocities and accelerations

- **input**:

  finalCoordinates: list or numpy array with final coordinates for profile

  duration: duration (time) for profile

- **output**: returns profile object, which is then used to compute interpolated trajectory

def **__init__** (*self*, *finalCoordinates*, *duration*)

- **classFunction**: initialize ProfileConstantAcceleration with vector of final coordinates and duration (time span)

---

def **GetBasicProfile** (*self*, *initialTime*, *initialCoordinates*, *globalMaxVelocities*, *globalMaxAccelerations*)

- **classFunction**: return a class representing profile which is used in Trajectory

For examples on ProfileConstantAcceleration see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `humanRobotInteraction.py` (Ex), `mobileMecanumWheelRobotWithLidar.py` (Ex), `ROSMobileManipulator.py` (Ex), `serialRobotFlexible.py` (Ex), `serialRobotInteractiveLimits.py` (Ex), ... , `movingGroundRobotTest.py` (TM), `serialRobotTest.py` (TM)

7.20.13.2 CLASS ProfileLinearAccelerationsList (in module robotics.motion)

class description: class to create a linear acceleration PTP profile, using a list of accelerations to define the profile; the (joint) coordinates and velocities are computed relative to values of previous profiles; ignores global max. accelerations and velocities of Trajectory

- **input**: accelerationList: list of tuples (relativeTime, accelerationVector) in which relativeTime is the time relative to the start of the profile (first time must be zero!) and accelerationVector is the list of accelerations of this time point, which is then linearly interpolated
- **output**: returns profile object, which is then used to compute interpolated trajectory in class Trajectory
- **example**:

```
profile = ProfileLinearAccelerationsList([(0,[0.,1.,2]), (0,[1.,1.,-2])])
```

def **__init__** (*self*, *accelerationList*)

- **classFunction**: initialize ProfileLinearAccelerationsList with a list of tuples containing time and acceleration vector

---

def **GetBasicProfile** (*self*, *initialTime*, *initialCoordinates*, *globalMaxVelocities*, *globalMaxAccelerations*)

- **classFunction**: return a class representing profile which is used in Trajectory

7.20.13.3 CLASS ProfilePTP (in module robotics.motion)

class description: class to create a synchronous motion PTP trajectory, using max. accelerations and max velocities; duration automatically computed

- **input**:

  finalCoordinates: list or numpy array with final coordinates for profile

  maxVelocities: list or numpy array with maximum velocities; may be empty list []; used if smaller than globalMaxVelocities

  maxAccelerations: list or numpy array with maximum accelerations; may be empty list []; used if smaller than globalMaxAccelerations

- **output**: returns profile object, which is then used to compute interpolated trajectory

def **__init__** (*self*, *finalCoordinates*, *syncAccTimes*= True, *maxVelocities*= [], *maxAccelerations*= [])

- **classFunction**: initialize ProfilePTP with final coordinates of motion, optionally max. velocities and accelerations just for this profile (overrides global settings)

def **GetBasicProfile** (*self*, *initialTime*, *initialCoordinates*, *globalMaxVelocities*, *globalMaxAccelerations*)

- **classFunction**: return a class representing profile which is used in Trajectory

For examples on ProfilePTP see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `serialRobotFlexible.py` (Ex), `serialRobotInteractiveLimits.py` (Ex), `serialRobotInverseKinematics.py` (Ex), `serialRobotKinematicTree.py` (Ex), `serialRobotTSD.py` (Ex), ...

7.20.13.4 CLASS Trajectory (in module robotics.motion)

class description: class to define (PTP) trajectories for robots and multibody systems; trajectories are defined for a set of coordinates (e.g. joint angles or other coordinates which need to be interpolated over time)

- **example**:

```
#create simple trajectory for two joint coordinates:
traj = Trajectory(initialCoordinates=[1,1], initialTime=1)
#add optimal trajectory with max. accelerations:
traj.Add(ProfileConstantAcceleration([2.,3.],2.))
traj.Add(ProfileConstantAcceleration([3.,-1.],2.))
#add profile with limited velocities and accelerations:
traj.Add(ProfilePTP([1,1],syncAccTimes=False, maxVelocities=[1,1], maxAccelerations
  =[5,5]))
#now evaluate trajectory at certain time point (this could be now applied in a user
   function)
[s,v,a] = traj.Evaluate(t=0.5)
```

def **__init__** (*self*, *initialCoordinates*, *initialTime*= 0, *maxVelocities*= [], *maxAccelerations*= [])

- **classFunction**: initialize robot link with parameters, being self-explaining
- **input**:

  initialTime: initial time for initial coordinates

  initialCoordinates: initial coordinates for profile

  maxVelocities: list or numpy array to describe global maximum velocities per coordinate

  maxAccelerations: list or numpy array to describe global maximum accelerations per coordinate

def **GetFinalCoordinates** (*self*)

- **classFunction**: returns the coordinates at the end of the (currently) Final profile

---

def **Add** (*self*, *profile*)

- **classFunction**: add successively profiles, using MotionProfile class

---

def **GetTimes** (*self*)

- **classFunction**: return vector of times of start/end of profiles

---

def **Initialize** (*self*)

- **classFunction**: initialize some parameters for faster evaluation

---

def **Evaluate** (*self*, *t*)

- **classFunction**: return interpolation of trajectory for coordinates, velocities and accelerations at given time
- **output**: [s, v, a] as numpy arrays representing coordinates, velocities and accelerations

---

def **EvaluateCoordinate** (*self*, *t*, *coordinate*)

- **classFunction**: return interpolation of trajectory for coordinate, including velocity and acceleration coordinate at given time
- **output**: [s, v, a] being scalar position, velocity and acceleration

– **notes**: faster for single coordinate than Evaluate(...)

---

def **__iter__** (*self*)

– **classFunction**: iterator allows to use for x in trajectory: ... constructs

---

def **__getitem__** (*self*, *key*)

– **classFunction**: access to profiles via operator [], allowing trajectory[0], etc.

---

def **__len__** (*self*)

– **classFunction**: allow using len(trajectory)

---

def **__repr__** (*self*)

– **classFunction**: representation of Trajectory is given a list of profiles, allowing easy inspection of data

For examples on Trajectory see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- `humanRobotInteraction.py` (Ex), `mobileMecanumWheelRobotWithLidar.py` (Ex), `ROSMobileManipulator.py` (Ex), `serialRobotFlexible.py` (Ex), `serialRobotInteractiveLimits.py` (Ex), ... , `movingGroundRobotTest.py` (TM), `serialRobotTest.py` (TM)

7.20.14 Module: robotics.special

additional support functions for robotics; The library is built on Denavit-Hartenberg Parameters and Homogeneous Transformations (HT) to describe transformations and coordinate systems

Author: Martin Sereinig

Date: 2021-22-09

def **VelocityManipulability** (*robot*, *HT*, *mode*)

- **function description**: compute velocity manipulability measure for given pose (homogeneous transformation)
- **input**:

  robot: robot class

  HT: actual pose as homogeneous transformaton matrix

  mode: rotational or translational part of the movement
- **output**: velocity manipulability measure as scalar value, defined as $\sqrt{(det(JJ^T))}$
- **author**: Martin Sereinig
- **notes**: compute velocity dependent manipulability definded by Yoshikawa, see [65]

---

def **ForceManipulability** (*robot*, *HT*, *mode*, *singularWeight*= 100)

- **function description**: compute force manipulability measure for given pose (homogeneous transformation)
- **input**:

  robot: robot class

  HT: actual pose as hoogenious transformaton matrix

  singularWeight: Weighting of singular configurations where the value would be infinity, default value=100

  mode: rotational or translational part of the movement
- **output**: force manipulability measure as scalar value, defined as $\sqrt{((det(JJ^T))^{-1})}$
- **author**: Martin Sereinig
- **notes**: compute force dependent manipulability definded by Yoshikawa, see [65]

---

def **StiffnessManipulability** (*robot*, *JointStiffness*, *HT*, *mode*, *singularWeight*= 1000)

- **function description**: compute cartesian stiffness measure for given pose (homogeneous transformation)

- **input**:

  robot: robot class

  JointStiffness: joint stiffness matrix

  HT: actual pose as homogeneous transformaton matrix

  mode: rotational or translational part of the movement

  singularWeight: Weighting of singular configurations where the value would be infinity,default value=1000

- **output**:

  stiffness manipulability measure as scalar value, defined as minimum Eigenvalaue of the Cartesian stiffness matrix

  Cartesian stiffness matrix

- **author**: Martin Sereinig

- **notes**:

- **status**: this function is **currently under development** and under testing!

---

def **JointJacobian** (*robot*, *HTJoint*, *HTLink*)

- **function description**: compute joint jacobian for each frame for given pose (homogeneous transformation)

- **input**:

  robot: robot class

  HT: actual pose as homogeneous transformaton matrix

- **output**: Link(body)-Jacobi matrix JJ: ${}^{i}JJ_i = [{}^{i}J_{Ri}, {}^{i}J_{Ti}]$ for each link i, seperated in rotational (J_R) and translational (J_T) part of Jacobian matrix located in the i^{th} coordiante system, see [63]

- **author**: Martin Sereinig

- **notes**: runs over number of HTs given in HT (may be less than number of links), caclulations in link coordinate system located at the end of each link regarding Standard Denavid-Hartenberg parameters, see [9]

---

def **MassMatrix** (*robot*, *HT*, *jointJacobian*)

- **function description**: compute mass matrix from jointJacobian
- **input**:

  robot: robot structure

  HT: actual pose as homogeneous transformaton matrix

  jointJacobian: provide list of jacobians as provided by function JointJacobian(...)
- **output**: MM: Mass matrix
- **author**: Martin Sereinig
- **notes**:

  Mass Matrix calculation calculated in joint coordinates regarding (std) DH parameter:

  ** Dynamic equations in minimal coordinates as described in MehrkÃ¶rpersysteme by Woernle, [63], p206, eq6.90.

  ** Caclulations in link coordinate system at the end of each link

---

def **DynamicManipulability** (*robot*, *HT*, *MassMatrix*, *Tmax*, *mode*, *singularWeight*= 1000)

- **function description**: compute dynamic manipulability measure for given pose (homogeneous transformation)
- **input**:

  robot: robot structure

  HT: actual pose as homogeneous transformaton matrix

  Tmax: maximum joint torques

  mode: rotational or translational part of the movement

  MassMatrix: Mass (inertia) Maxtrix provided by the function MassMatrix

  singularWeight: Weighting of singular configurations where the value would be infinity,default value=1000
- **output**:

  dynamic manipulability measure as scalar value, defined as minimum Eigenvalaue of the dynamic manipulability matrix N

  dynamic manipulability matrix
- **author**: Martin Sereinig

- **notes**: acceleration dependent manipulability definded by Chiacchio, see [7], eq.32. The eigenvectors and eigenvalues of N ([eigenvec eigenval]=eig(N))gives the direction and value of minimal and maximal accaleration)
- **status**: this function is **currently under development** and under testing!

---

def **CalculateAllMeasures** (*robot*, *robotDic*, *q*, *mode*, *flag*= [0,0,0,0])

- **function description**: calculation of 4 different manipulability measures using a certain serial robot
- **input**:

  robot: robot class

  robotDic: robot dictionary

  q: joint position vector

  mode: trans or rot, for used parts of the manipulator Jacobi Matrix

  Tmax: maximum joint torques

  mode: rotational or translational part of the movement

  flag: flag vector to swich individual measure on and of [flagmv,flagmf,flagmst,flagma] = [1,1,1,1]
- **output**: [mv,mf,mst,mstM,ma,maM]
- **author**: Martin Sereinig
- **notes**:
- **status**: this function is **currently under development** and under testing!

7.20.15 Module: robotics.utilities

The utilities contains general helper functions for the robotics module

Date: 2023-04-15

def **AddLidar** (*mbs*, *generalContactIndex*, *positionOrMarker*, *minDistance*= 0, *maxDistance*= 1e7, *cylinderRadius*= 0, *lineLength*= 1, *numberOfSensors*= 100, *angleStart*= 0, *angleEnd*= 2*np.pi, *inclination*= 0, *rotation*= np.eye(3), *selectedTypeIndex*= exudyn.ContactTypeIndex.IndexEndOfEnumList, *storeInternal*= False, *fileName*= '', *measureVelocity*= False, *addGraphicsObject*= True, *drawDisplaced*= True, *color*= [1.0, 0.0, 0.0, 1.0])

- **function description**: Function to add many distance sensors to represent Lidar; sensors can be either placed on absolute position or attached to rigid body marker
- **input**:
  generalContactIndex: the number of the GeneralContact object in mbs; the index of the GeneralContact object which has been added with last AddGeneralContact(...) command is generalContactIndex=mbs.NumberOfGeneralContacts()-1

  positionOrMarker: either a 3D position as list or np.array, or a MarkerIndex with according rigid body marker

  minDistance: the minimum distance which is accepted; smaller distance will be ignored

  maxDistance: the maximum distance which is accepted; items being at maxDistance or futher are ignored; if no items are found, the function returns maxDistance

  cylinderRadius: in case of spheres (selectedTypeIndex=ContactTypeIndex.IndexSpheresMarkerBased), a cylinder can be used which measures the shortest distance at a certain radius (geometrically interpreted as cylinder)

  lineLength: length of line to be drawn; note that this length is drawn from obstacle towards sensor if drawDisplaced=True, but the length is always constant

  numberOfSensors: number of sensors arranged between angleStart and angleEnd; higher numbers give finer resolution (but requires more CPU time); must be larger than 1

  angleStart: starting rangle of angles to be used (in radiant); angle of lidar beam is relative to X-axis, using positive rotation sense about Z-axis

  angleEnd: end of range for angle to be used (in radiant); angle of lidar beam is relative to X-axis, using positive rotation sense about Z-axis

  inclination: angle of inclination (radiant), positive values showing upwards (Z-direction) if rotation is the identity matrix

  rotation: a 3x3 rotation matrix (numpy); the sensor is placed in the X-Y plane of the marker where it is added to; however, you can use this rotation matrix to change the orientation

  selectedTypeIndex: either this type has default value, meaning that all items in GeneralContact are measured, or there is a specific type index, which is the only type that is considered during measurement

  storeInternal: like with any SensorUserFunction, setting to True stores sensor data internally

  fileName: if defined, recorded data of SensorUserFunction is written to specified file

  measureVelocity: if True, the sensor measures additionally the velocity (component 0=distance, component 1=velocity); velocity is the velocity in direction 'dirSensor' and does not account for changes in geometry, thus it may be different from the time derivative of the distance!

  addGraphicsObject: if True, the distance sensor is also visualized graphically in a simplified manner with a red line having the length of dirSensor; NOTE that updates are ONLY performed during computation, not in visualization; for this reason, solutionSettings.sensorsWritePeriod should be accordingly small

drawDisplaced: if True, the red line is drawn backwards such that it moves along the measured surface; if False, the beam is fixed to marker or position

color: optional color for 'laser beam' to be drawn

- **output**: creates sensor and returns list of sensor numbers for all laser sensors
- **notes**: use generalContactIndex = CreateDistanceSensorGeometry(...) before to create GeneralContact module containing geometry

For examples on AddLidar see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- mobileMecanumWheelRobotWithLidar.py (Ex), laserScannerTest.py (TM)

---

def **GetRoboticsToolboxInternalModel** (*modelName*= '', *ignoreURDFerrors*= True)

- **function description**:

  Interface to roboticstoolbox (RTB) for loading internal robot models. Function retrieves internal model available from roboticstoolbox.models.URDF, usually stored in

  site-packages/rtbdata/xacro/. See the github project of roboticstoolbox-python of P. Corke and J. Haviland for more details.

  The model name is the short name used internally in the RTB. For available names, see the list roboticstoolbox.models.URDF.__all__ !

- **input**:

  modelName: string for model, such as UR5, Puma560, Panda or LBR

  ignoreURDFerrors: if set True, urdf errors are ignored and only the model is loaded

- **output**: returns dictionary with 'robot' (RTB Robot class), 'urdf' which is the RTB representation of the URDF file for loading mesh files
- **notes**: requires installation (pip install) of roboticstoolbox-python; in our tests we had problems with installers on newer Python and therefore tested with Python 3.9! Note that some models doe not include mass and inertia and therefore will not run as dynamic models!

For examples on GetRoboticsToolboxInternalModel see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- serialRobotURDF - Kopie.py (Ex), serialRobotURDF.py (Ex)

---

def **LoadURDFrobot** (*urdfFilePath*, *urdfBasePath*, *gripperLinks*= None, *manufacturer*= '')

- **function description**: Interface to roboticstoolbox (RTB) of P. Corke and J. Haviland. Use this function for loading urdf/xacro files.
- **input**:

  urdfFilePath: string relative to urdfBasePath, representing xacro or urdf file

  urdfBasePath: string representing the base path of the urdf or xacro directory for the respective robot; the urdfBasePath is used to load further files which are given internally in urdf files, such as collision or mesh files

  gripperLinks: list of link numbers representing gripper (as used internally in RTB Robot class)
- **output**: returns dictionary with 'robot' (RTB Robot class), 'urdf' which is the RTB representation of the URDF file for loading mesh files
- **notes**: requires installation (pip install) of roboticstoolbox-python; in our tests we had problems with installers on newer Python and therefore tested with Python 3.9! Note that some models doe not include mass and inertia and therefore will not run as dynamic models!

---

def **GetURDFrobotData** (*robot*, *urdf*= None, *linkColorList*= None, *staticJointValues*= None, *returnStaticGraphicsList*= False, *exportMesh*= False, *verbose*= 1)

- **function description**: Interface to roboticstoolbox (RTB) of P. Corke and J. Haviland and Pymeshlab to import robot model and visualization into a struture readable by Exudyn. NOTE that this function is to be seen as a starting point for import, while some models have to be imported differently, in particular for joints that are not revolute or prismatic (in this case, copy function into local file and modify)!
- **input**:

  robot: a RTB Robot (class) model, as returned e.g. by LoadURDFrobot

  urdf: a RTB URDF (class) representing the URDF data, as returned e.g. by LoadURDFrobot

  linkColorList: if not None, this can contain a list of RGBA color lists for each link to prescribe colors instead of using internally stored colors or general color information (.obj files); set linkColorList=[graphics.color.red]*8 to set 8 link colors red; links are counted as in the urdf file and may be different from the number of joints

  staticJointValues: if not None, has to be a list of joint angles (or displacements) for computing the static graphics list

  returnStaticGraphicsList: return a list of GraphicsData which can be put into a ground to check visualization for zero joints, using: mbs.CreateGround(graphicsDataList=staticGraphicsList)

exportMesh: if True, the returned dict also contains a meshSetList which refers to the MeshSet in pymeshlab, which can be used for debugging purposes

verbose: 0 .. no output printed (only exceptions), 1 .. warnings, 2 .. further information

- **output**: returns dictionary with items linkList, graphicsBaseList, graphicsToolList
- **notes**: requires installation (pip install) of roboticstoolbox-python and pymeshlab; if pymeshlab is not installed, a warning is raised and graphics is ignored

For examples on GetURDFrobotData see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- serialRobotURDF - Kopie.py (Ex), serialRobotURDF.py (Ex)

7.21 Module: signalProcessing

The signal library supports processing of signals for import (e.g. measurement data) and for filtering result data.

Date: 2020-12-10

Notes: This module is still under construction and should be used with care!

def **FilterSensorOutput** (*signal*, *filterWindow*= 5, *polyOrder*= 3, *derivative*= 0, *centralDifferentiate*= True)

- **function description**: filter output of sensors (using numpy savgol filter) as well as numerical differentiation to compute derivative of signal
- **input**:

  signal: numpy array (2D array with column-wise storage of signals, as exported by EXUDYN position, displacement, etc. sensors); first column = time, other columns = signals to operate on; note that it is assumed, that time devided in almost constant steps!

  derivative: 0=no derivative, 1=first derivative, 2=second derivative, etc. (>2 only possible with filter)

  polyOrder: order of polynomial for interpolation filtering

  filterWindow: if zero: produces unfiltered derivative; if positive, must be ODD integer 1,3,5,... and > polyOrder; filterWindow determines the length of the filter window (e.g., to get rid of noise)

  centralDifferentiate: if True, it uses a central differentiation for first order, unfiltered derivatives; leads to less phase shift of signal!
- **output**: numpy array containing same columns, but with filtered signal and according derivatives

For examples on FilterSensorOutput see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ANCFoutputTest.py (TM), objectFFRFreducedOrderAccelerations.py (TM)

---

def **FilterSignal** (*signal*, *samplingRate*= -1, *filterWindow*= 5, *polyOrder*= 3, *derivative*= 0, *centralDifferentiate*= True)

- **function description**: filter 1D signal (using numpy savgol filter) as well as numerical differentiation to compute derivative of signal
- **input**:

  signal: 1D numpy array

  samplingRate: (time increment) of signal values, needed for derivatives

  derivative: 0=no derivative, 1=first derivative, 2=second derivative, etc. (>2 only possible with filter)

  polyOrder: order of polynomial for interpolation filtering

  filterWindow: if zero: produces unfiltered derivative; if positive, must be ODD integer 1,3,5,... and > polyOrder; filterWindow determines the length of the filter window (e.g., to get rid of noise)

  centralDifferentiate: if True, it uses a central differentiation for first order, unfiltered derivatives; leads to less phase shift of signal!
- **output**: numpy array containing same columns, but with filtered signal and according derivatives

For examples on FilterSignal see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- objectFFRFreducedOrderAccelerations.py (TM)

---

def **ComputeFFT** (*time*, *data*)

- **function description**: computes fast-fourier-transform (FFT) resulting in frequency, magnitude and phase of signal data using numpy.fft of numpy
- **input**:

  time ... time vector in SECONDS in numpy format, having constant sampling rate (not checked!)

  data ... data vector in numpy format

- **output**:

  frequency ... frequency vector (Hz, if time is in SECONDS)

  magnitude ... magnitude vector

  phase ... phase vector (in radiant)

- **author**: Stefan Holzinger

- **date**: 02.04.2020

For examples on ComputeFFT see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- nMassOscillatorEigenmodes.py (Ex)

---

def **GetInterpolatedSignalValue** (*time*, *dataArray*, *timeArray*= [], *dataArrayIndex*= -1, *timeArrayIndex*= -1, *rangeWarning*= True, *tolerance*= 1e-6)

- **function description**: Interpolate signal having time values with constant sampling rate in timeArray and according data in dataArray

- **input**:

  time: time at which the data should be evaluated

  dataArray: 1D numpy array containing data values to be interpolated [alternatively: 2D numpy array, rows containg the data of the according time point; use dataArrayColumnIndex to specify the column of requested data]

  timeArray: 1D numpy array containing time values with CONSTANT SAMPLING RATE to be interpolated [alternatively: 2D numpy array, rows containg the time and data of the according time point; use timeArrayColumnIndex to specify the column representing time]; if timeArray is empty list [], dataArray is used instead!

  rangeWarning: print warning if resulting index gets out of range

  dataArrayColumnIndex: in case of 2D arrays, this represents the column of the requested data

  timeArrayColumnIndex: in case of 2D arrays, this represents the column of time values

  tolerance: this tolerance is used to check, if the timeArray has equidistant interpolation and if the found indices are correct; use e.g. 1e10 in order to ignore this tolerance

- **output**: interpolated value

- **notes**: for interpolation of data WITHOUT constant data rate, use numpy.interp(time, timeArray, dataArray) in case that timeArray and dataArray are 1D arrays

7.22 Module: solver

The solver module provides interfaces to static, dynamic and eigenvalue solvers. Most of the solvers are implemented inside the C++ core.

Author: Johannes Gerstmayr

Date: 2020-12-02

Notes: Solver functions are included directly in exudyn and can be used with mbs.SolveStatic(...)

def **SolverErrorMessage** (*solver*, *mbs*, *isStatic*= False, *showCausingObjects*= True, *showCausingNodes*= True, *showHints*= True)

- **function description**: (internal) helper function for unique error and helper messages

---

def **SolveStatic**(*mbs*, *simulationSettings*= exudyn.SimulationSettings(), *updateInitialValues*= False, *storeSolver*= True, *showHints*= False, *showCausingItems*= True, *autoAssemble*= True)

- **NOTE**: this function is directly available in MainSystem (mbs); it should be directly called as mbs.SolveStatic(...). For description of the interface, see the MainSystem Python extensions, Section 6.5.2.

---

def **SolveDynamic**(*mbs*, *simulationSettings*= exudyn.SimulationSettings(), *solverType*= exudyn.DynamicSolverType.GeneralizedAlpha, *updateInitialValues*= False, *storeSolver*= True, *showHints*= False, *showCausingItems*= True, *autoAssemble*= True)

- **NOTE**: this function is directly available in MainSystem (mbs); it should be directly called as mbs.SolveDynamic(...). For description of the interface, see the MainSystem Python extensions, Section 6.5.2.

---

def **SolverSuccess** (*solverStructure*)

- **function description**: return success (True/False) and error message of solver after SolveSteps(...), SolveSystem(...), SolveDynamic(...) or SolveStatic(...) have been called. May also be set if other higher level functions called e.g. SolveSystem(...)
- **input**: solverStructure: solver structure, as stored in mbs.sys or as created e.g. by exudyn.MainSolverExplicit()

- **output**: [success, errorString], returns success=True or False and in case of no success, information is provided in errorString
- **example**:

```
#assume MainSystem mbs, exu library and simulationSettings:
try:
    mbs.SolveDynamic(simulationSettings)
except:
    [success, msg] = exu.SolverSuccess(mbs.sys['dynamicSolver'])
    exu.Print('success=',success)
    exu.Print('error message=',msg)
#alternative:
solver=exu.MainSolverImplicitSecondOrder()
...
[success, msg] = exu.SolverSuccess(solver)
```

---

def **ComputeLinearizedSystem**(*mbs*, *simulationSettings*= exudyn.SimulationSettings(), *projectIntoConstraintNullspace*= False, *singularValuesTolerance*= 1e-12, *returnConstraintJacobian*= False, *returnConstraintNullspace*= False, *autoAssemble*= True)

- **NOTE**: this function is directly available in MainSystem (mbs); it should be directly called as mbs.ComputeLinearizedSystem(...). For description of the interface, see the MainSystem Python extensions, Section 6.5.2.

---

def **ComputeODE2Eigenvalues**(*mbs*, *simulationSettings*= exudyn.SimulationSettings(), *useSparseSolver*= False, *numberOfEigenvalues*= 0, *constrainedCoordinates*= [], *convert2Frequencies*= False, *useAbsoluteValues*= True, *computeComplexEigenvalues*= False, *ignoreAlgebraicEquations*= False, *singularValuesTolerance*= 1e-12, *autoAssemble*= True)

- **NOTE**: this function is directly available in MainSystem (mbs); it should be directly called as mbs.ComputeODE2Eigenvalues(...). For description of the interface, see the MainSystem Python extensions, Section 6.5.2.

---

def **ComputeSystemDegreeOfFreedom**(*mbs*, *simulationSettings*= exudyn.SimulationSettings(), *threshold*= 1e-12, *verbose*= False, *useSVD*= False, *autoAssemble*= True)

- **NOTE**: this function is directly available in MainSystem (mbs); it should be directly called as mbs.ComputeSystemDegreeOfFreedom(...). For description of the interface, see the MainSystem Python extensions, Section 6.5.2.

def **CheckSolverInfoStatistics** (*solverName*, *infoStat*, *numberOfEvaluations*)

- **function description**:

  helper function for solvers to check e.g. if high number of memory allocations happened during simulation

  This can happen, if large amount of sensors are attached and output is written in every time step

- **input**: stat=exudyn.special.InfoStat() from previous step, numberOfEvaluations is a counter which is proportional to number of RHS evaluations in method

7.23 Module: utilities

Basic support functions for simpler creation of Exudyn models. Advanced functions for loading and animating solutions and for drawing a graph of the mbs system. This library requires numpy (as well as time and copy)

Author: Johannes Gerstmayr

Date: 2019-07-26 (created)

def **GetOtherMarker** (*mbs*, *bodyNumber*, *existingMarker*, *show*= True)

- **function description**: creates a new marker for body with bodyNumber using another marker existingMarker, such that the new marker has the same reference position as the existing marker, working for MarkerBodyPosition (no rotations included); this alleviates creation of markers and calculation of localPosition

- **input**:

  mbs: multibody system where new marker is added to

  bodyNumber: body where new marker shall be attached to

  existingMarker: marker number which serves as a reference

  show: if True, marker is shown

- **output**: returns marker number of new marker

- **example**:

```
#oBody0 = mbs.CreateRigidBody(...)
#oBody1 = mbs.CreateRigidBody(...)
marker0 = mbs.AddMarker(MarkerBodyPosition(bodyNumber=oBody0,localPosition=[1,0,0])
  )
#create joint from one marker (with rotation) and other body
mbs.AddObject(SphericalJoint(markerNumbers=[marker0, GetOtherMarker(mbs, oBody1,
  marker0)]))
```

For examples on GetOtherMarker see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- NGsolveFFRF.py (Ex)

---

def **GetJointArgs** (*mbs*, *markerNumber0*= None, *markerNumber1*= None, *rotationMarker0*= None, *rotationMarker1*= None, *bodyNumber0*= None, *bodyNumber1*= None)

- **function description**: creates input args for joints, based on an exiting marker (markerNumber, may be rigid or flex body), with optional existing rotationMarker and uses another rigid body (given as bodyNumber) to create a new MarkerBodyRigid and rotationMarker; this alleviates creation of joint args, see the example; inputs are either markerNumber0 [, rotationMarker0], bodyNumber1 OR markerNumber1 [, rotationMarker1], bodyNumber0
- **input**:

  mbs: multibody system where new marker is added to

  markerNumber0: markerNumber of existing rigid body marker

  markerNumber1: markerNumber of existing rigid body marker

  rotationMarker0: joint marker rotation matrix for markerNumber0 (must be MarkerBodyRigid)

  rotationMarker1: joint marker rotation matrix for markerNumber1 (must be MarkerBodyRigid)

  bodyNumber0: existing body used to create new marker

  bodyNumber1: existing body used to create new marker
- **output**: returns dict with 'markerNumbers' list, 'rotationMarker0' and 'rotationMarker1', ready to be used as args
- **example**:

```
#oBody0 = mbs.CreateRigidBody(...)
#oBody1 = mbs.CreateRigidBody(...)
marker0 = mbs.AddMarker(MarkerBodyRigid(bodyNumber=oBody0,localPosition=[1,0,0]))
rotM0 = RotationMatrixX(0.5*pi)
#create joint from one marker (with rotation) and other body
mbs.AddObject(RevoluteJointZ(**GetJointArgs(mbs, markerNumber0=marker0,
                                            rotationMarker0=rotM0,
                                            bodyNumber1=oBody1)
```

For examples on GetJointArgs see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- jointArgsTest.py (TM)

---

def **ShowOnlyObjects** (*mbs*, *objectNumbers*= [], *showOthers*= False)

- **function description**: function to hide all objects in mbs except for those listed in objectNumbers
- **input**:

  mbs: mbs containing object

  objectNumbers: integer object number or list of object numbers to be shown; if empty list [], then all objects are shown

  showOthers: if True, then all other objects are shown again
- **output**: changes all colors in mbs, which is NOT reversible

---

def **HighlightItem** (*SC*, *mbs*, *itemNumber*, *itemType*= exudyn.ItemType.Object, *showNumbers*= True)

- **function description**: highlight a certain item with number itemNumber; set itemNumber to -1 to show again all objects
- **input**:

  mbs: mbs containing object

  itemNumbers: integer object/node/etc number to be highlighted

  itemType: type of items to be highlighted

  showNumbers: if True, then the numbers of these items are shown

---

def **__UFsensorDistance** (*mbs*, *t*, *sensorNumbers*, *factors*, *configuration*)

- **function description**: internal function used for CreateDistanceSensor

---

def **CreateDistanceSensorGeometry**(*mbs*, *meshPoints*, *meshTrigs*, *rigidBodyMarkerIndex*, *searchTreeCellSize*= [8,8,8])

- **NOTE**: this function is directly available in MainSystem (mbs); it should be directly called as mbs.CreateDistanceSensorGeometry(...). For description of the interface, see the MainSystem Python extensions, Section 6.5.2.

---

def **CreateDistanceSensor**(*mbs*, *generalContactIndex*, *positionOrMarker*, *dirSensor*, *minDistance*= -1e7, *maxDistance*= 1e7, *cylinderRadius*= 0, *selectedTypeIndex*= exudyn.ContactTypeIndex.IndexEndOfEnumList, *storeInternal*= False, *fileName*= '', *measureVelocity*= False, *addGraphicsObject*= False, *drawDisplaced*= True, *color*= exudyn.graphics.color.red)

- **NOTE**: this function is directly available in MainSystem (mbs); it should be directly called as mbs.CreateDistanceSensor(...). For description of the interface, see the MainSystem Python extensions, Section 6.5.2.

---

def **UFsensorRecord** (*mbs*, *t*, *sensorNumbers*, *factors*, *configuration*)

- **function description**: DEPRECATED: Internal SensorUserFunction, used in function AddSensorRecorder
- **notes**: Warning: this method is DEPRECATED, use storeInternal in Sensors, which is much more performant; Note, that a sensor usually just passes through values of an existing sensor, while recording the values to a numpy array row-wise (time in first column, data in remaining columns)

---

def **AddSensorRecorder** (*mbs*, *sensorNumber*, *endTime*, *sensorsWritePeriod*, *sensorOutputSize*= 3)

- **function description**: DEPRECATED: Add a SensorUserFunction object in order to record sensor output internally; this avoids creation of files for sensors, which can speedup and simplify evaluation in ParameterVariation and GeneticOptimization; values are stored internally in mbs.variables['sensorRecord'+str(sensorNumber)] where sensorNumber is the mbs sensor number
- **input**:

  mbs: mbs containing object

  sensorNumber: integer sensor number to be recorded

  endTime: end time of simulation, as given in simulationSettings.timeIntegration.endTime

sensorsWritePeriod: as given in simulationSettings.solutionSettings.sensorsWritePeriod

sensorOutputSize: size of sensor data: 3 for Displacement, Position, etc. sensors; may be larger for RotationMatrix or Coordinates sensors; check this size by calling mbs.GetSensorValues(sensorNum

- **output**: adds an according SensorUserFunction sensor to mbs; returns new sensor number; during initialization a new numpy array is allocated in mbs.variables['sensorRecord'+str(sensorNumber)] and the information is written row-wise: [time, sensorValue1, sensorValue2, ...]
- **notes**: Warning: this method is DEPRECATED, use storeInternal in Sensors, which is much more performant; Note, that a sensor usually just passes through values of an existing sensor, while recording the values to a numpy array row-wise (time in first column, data in remaining columns)

For examples on AddSensorRecorder see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- ComputeSensitivitiesExample.py (Ex)

---

def **LoadSolutionFile** (*fileName*, *safeMode*= False, *maxRows*= -1, *verbose*= True, *hasHeader*= True)

- **function description**: read coordinates solution file (exported during static or dynamic simulation with option exu.SimulationSettings().solutionSettings.coordinatesSolutionFileName='...') into dictionary:
- **input**:

  fileName: string containing directory and filename of stored coordinatesSolutionFile

  saveMode: if True, it loads lines directly to load inconsistent lines as well; use this for huge files (>2GB); is slower but needs less memory!

  verbose: if True, some information is written when importing file (use for huge files to track progress)

  maxRows: maximum number of data rows loaded, if saveMode=True; use this for huge files to reduce loading time; set -1 to load all rows

  hasHeader: set to False, if file is expected to have no header; if False, then some error checks related to file header are not performed
- **output**: dictionary with 'data': the matrix of stored solution vectors, 'columnsExported': a list with integer values showing the exported sizes [nODE2, nVel2, nAcc2, nODE1, nVel1, nAlgebraic, nData], 'nColumns': the number of data columns and 'nRows': the number of data rows

For examples on LoadSolutionFile see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- beltDriveALE.py (Ex), beltDriveReevingSystem.py (Ex), beltDrivesComparison.py (Ex), fourBarMechanism3D.py (Ex), kinematicTreeAndMBS.py (Ex), ... , ACFtest.py (TM), ANCFbeltDrive.py (TM), ANCFgeneralContactCircle.py (TM), ...

---

def **NumpyInt8ArrayToString** (*npArray*)

- **function description**: simple conversion of int8 arrays into strings (not highly efficient, so use only for short strings)

---

def **BinaryReadIndex** (*file*, *intType*)

- **function description**: read single Index from current file position in binary solution file

---

def **BinaryReadReal** (*file*, *realType*)

- **function description**: read single Real from current file position in binary solution file

---

def **BinaryReadString** (*file*, *intType*)

- **function description**: read string from current file position in binary solution file

---

def **BinaryReadArrayIndex** (*file*, *intType*)

- **function description**: read Index array from current file position in binary solution file

---

def **BinaryReadRealVector** (*file*, *intType*, *realType*)

- **function description**: read Real vector from current file position in binary solution file
- **output**: return data as numpy array, or False if no data read

---

def **LoadBinarySolutionFile** (*fileName*, *maxRows*= -1, *verbose*= True)

- **function description**: read BINARY coordinates solution file (exported during static or dynamic simulation with option exu.SimulationSettings().solutionSettings.coordinatesSolutionFileName='...') into dictionary
- **input**:

  fileName: string containing directory and filename of stored coordinatesSolutionFile

  verbose: if True, some information is written when importing file (use for huge files to track progress)

  maxRows: maximum number of data rows loaded, if saveMode=True; use this for huge files to reduce loading time; set -1 to load all rows
- **output**: dictionary with 'data': the matrix of stored solution vectors, 'columnsExported': a list with integer values showing the exported sizes [nODE2, nVel2, nAcc2, nODE1, nVel1, nAlgebraic, nData], 'nColumns': the number of data columns and 'nRows': the number of data rows

---

def **RecoverSolutionFile** (*fileName*, *newFileName*, *verbose*= 0)

- **function description**: recover solution file with last row not completely written (e.g., if crashed, interrupted or no flush file option set)
- **input**:

  fileName: string containing directory and filename of stored coordinatesSolutionFile

  newFileName: string containing directory and filename of new coordinatesSolutionFile

  verbose: 0=no information, 1=basic information, 2=information per row
- **output**: writes only consistent rows of file to file with name newFileName

---

def **InitializeFromRestartFile** (*mbs*, *simulationSettings*, *restartFileName*, *verbose*= True)

- **function description**: recover initial coordinates, time, etc. from given restart file
- **input**:

  mbs: MainSystem to be operated with

  simulationSettings: simulationSettings which is updated and shall be used afterwards for SolveDynamic(...) or SolveStatic(...)

  restartFileName: string containing directory and filename of stored restart file, as given in solutionSettings.restartFileName

  verbose: False=no information, True=basic information
- **output**: modifies simulationSettings and sets according initial conditions in mbs

---

def **SetSolutionState** (*mbs*, *solution*, *row*, *configuration*= exudyn.ConfigurationType.Current, *sendRedrawSignal*= True)

- **function description**: load selected row of solution dictionary (previously loaded with LoadSolutionFile) into specific state; flag sendRedrawSignal is only used if configuration = exudyn.ConfigurationType.Visualization

---

def **AnimateSolution** (*mbs*, *solution*, *rowIncrement*= 1, *timeout*= 0.04, *createImages*= False, *runLoop*= False)

- **function description**: This function is not further maintaned and should only be used if you do not have tkinter (like on some MacOS versions); use exudyn.interactive.SolutionViewer() instead! AnimateSolution consecutively load the rows of a solution file and visualize the result
- **input**:

  mbs: the system used for animation

  solution: solution dictionary previously loaded with LoadSolutionFile; will be played from first to last row

  rowIncrement: can be set larger than 1 in order to skip solution frames: e.g. rowIncrement=10 visualizes every 10th row (frame)

  timeout: in seconds is used between frames in order to limit the speed of animation; e.g. use timeout=0.04 to achieve approximately 25 frames per second

  createImages: creates consecutively images from the animation, which can be converted into an animation

runLoop: if True, the animation is played in a loop until 'q' is pressed in render window

- **output**: renders the scene in mbs and changes the visualization state in mbs continuously

For examples on AnimateSolution see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- NGsolvePistonEngine.py (Ex), SliderCrank.py (Ex), slidercrankWithMassSpring.py (Ex), sliderCrankFloatingTest.py (TM)

---

def **DrawSystemGraph**(*mbs*, *showLoads*= True, *showSensors*= True, *useItemNames*= False, *useItemTypes*= False, *addItemTypeNames*= True, *multiLine*= True, *fontSizeFactor*= 1., *layoutDistanceFactor*= 3., *layoutIterations*= 100, *showLegend*= True, *tightLayout*= True)

- **NOTE**: this function is directly available in MainSystem (mbs); it should be directly called as mbs.DrawSystemGraph(...). For description of the interface, see the MainSystem Python extensions, Section 6.5.2.

---

def **CreateTCPIPconnection** (*sendSize*, *receiveSize*, *IPaddress*= '127.0.0.1', *port*= 52421, *bigEndian*= False, *verbose*= False)

- **function description**:

  function which has to be called before simulation to setup TCP/IP socket (server) for sending and receiving data; can be used to communicate with other Python interpreters or for communication with MATLAB/Simulink

- **input**:

  sendSize: number of double values to be sent to TCPIP client

  receiveSize: number of double values to be received from TCPIP client

  IPaddress: string containing IP address of client (e.g., '127.0.0.1')

  port: port for communication with client

  bigEndian: if True, it uses bigEndian, otherwise littleEndian is used for byte order

- **output**: returns information (TCPIPdata class) on socket; recommended to store this in mbs.sys['TCPIPobject']

- **example**:

```
mbs.sys['TCPIPobject'] = CreateTCPIPconnection(sendSize=3, receiveSize=2,
                                               bigEndian=True, verbose=True)
sampleTime = 0.01 #sample time in MATLAB! must be same!
mbs.variables['tLast'] = 0 #in case that exudyn makes finer steps than sample time
def PreStepUserFunction(mbs, t):
    if t >= mbs.variables['tLast'] + sampleTime:
        mbs.variables['tLast'] += sampleTime
        tcp = mbs.sys['TCPIPobject']
        y = TCPIPsendReceive(tcp, np.array([t, np.sin(t), np.cos(t)])) #time,
  torque
        tau = y[1]
        exudyn.Print('tau=',tau)
    return True
try:
    mbs.SetPreStepUserFunction(PreStepUserFunction)
    #%%++++++++++++++++++++++++++++++++++++++++++++++++++
    mbs.Assemble()
    [...] #start renderer; simulate model
finally: #use this to always close connection, even in case of errors
    CloseTCPIPconnection(mbs.sys['TCPIPobject'])
#****************************************
#the following settings work between Python and MATLAB-Simulink (client), and gives
   stable results(with only delay of one step):
# TCP/IP Client Send:
#   priority = 2 (in properties)
#   blocking = false
#   Transfer Delay on (but off also works)
# TCP/IP Client Receive:
#   priority = 1 (in properties)
#   blocking = true
#   Sourec Data type = double
#   data size = number of double in packer
#   Byte order = BigEndian
#   timeout = 10
```

For examples on CreateTCPIPconnection see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- TCPIPexudynMatlab.py (Ex)

---

def **TCPIPsendReceive** (*TCPIPobject*, *sendData*)

- **function description**:

  call this function at every simulation step at which you intend to communicate with other programs via TCPIP; e.g., call this function in preStepUserFunction of a mbs model

- **input**:

  TCPIPobject: the object returned by CreateTCPIPconnection(...)

  sendData: numpy array containing data (double array) to be sent; must agree with sendSize

- **output**: returns array as received from TCPIP

- **example**:

```
mbs.sys['TCPIPobject']=CreateTCPIPconnection(sendSize=2, receiveSize=1, IPaddress='
  127.0.0.1')
y = TCPIPsendReceive(mbs.sys['TCPIPobject'], np.array([1.,2.]))
exudyn.Print(y)
```

For examples on TCPIPsendReceive see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- TCPIPexudynMatlab.py (Ex)

---

def **CloseTCPIPconnection** (*TCPIPobject*)

- **function description**: close a previously created TCPIP connection

For examples on CloseTCPIPconnection see Relevant Examples (Ex) and TestModels (TM) with weblink to github:

- TCPIPexudynMatlab.py (Ex)

7.23.1 CLASS TCPIPdata (in module utilities)

class description: helper class for CreateTCPIPconnection and for TCPIPsendReceive

Chapter 8

Objects, nodes, markers, loads and sensors reference manual

This chapter includes the reference manual for all objects (bodies/constraints), nodes, markers, loads and sensors (**= items**). For description of types (e.g., the meaning of `Vector3D` or `NumpyMatrix`), see Section 4.0.1.

8.1 Nodes

Nodes provide coordinates for objects. Loads can be applied and Markers or Sensors can be attached to Nodes. The sorting of Nodes in the system (the order they are added to mbs) defines the order of system coordinates.

8.1.1 NodePoint

A 3D point node for point masses or solid finite elements which has 3 displacement degrees of freedom for second order ordinary differential equations (ODE2).

Additional information for NodePoint:

- This `Node` has/provides the following types = `Position`
- **Short name** for Python = `Point`
- **Short name** for Python visualization object = `VPoint`

The item **NodePoint** with type = 'Point' has the following parameters:

Name	type	size	default value	description
name	String		''	node's unique name
referenceCoordinates	Vector3D	3	[0.,0.,0.]	reference coordinates of node, e.g. ref. coordinates for finite elements; global position of node without displacement
initialCoordinates	Vector3D	3	[0.,0.,0.]	initial displacement coordinate
initialVelocities	Vector3D	3	[0.,0.,0.]	initial velocity coordinate
visualization	VNodePoint			parameters for visualization of item

The item VNodePoint has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size (diameter, dimensions of underlying cube, etc.) for item; size == -1.f means that default size is used
color	Float4	4	[-1.,-1.,-1.,-1.]	Default RGBA color for nodes; 4th value is alpha-transparency; R=-1.f means, that default color is used

8.1.1.1 DESCRIPTION of NodePoint:

Information on input parameters:

input parameter	symbol	description see tables above
referenceCoordinates	$\mathbf{q}_{\text{ref}} = [q_0, q_1, q_2]^{\text{T}}_{\text{ref}} = \mathbf{p}_{\text{ref}} = [r_0, r_1, r_2]^{\text{T}}$	
initialCoordinates	$\mathbf{q}_{\text{ini}} = [q_0, q_1, q_2]^{\text{T}}_{\text{ini}} = \mathbf{u}_{\text{ini}} = [u_0, u_1, u_2]^{\text{T}}_{\text{ini}}$	
initialVelocities	$\dot{\mathbf{q}}_{\text{ini}} = \mathbf{v}_{\text{ini}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2]^{\text{T}}_{\text{ini}}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position	$\mathbf{p}_{\text{config}} = [p_0, p_1, p_2]^{\text{T}}_{\text{config}} = \mathbf{u}_{\text{config}} + \mathbf{p}_{\text{ref}}$	global 3D position vector of node; $\mathbf{u}_{\text{ref}} = 0$
Displacement	$\mathbf{u}_{\text{config}} = [q_0, q_1, q_2]^{\text{T}}_{\text{config}}$	global 3D displacement vector of node
Velocity	$\mathbf{v}_{\text{config}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2]^{\text{T}}_{\text{config}}$	global 3D velocity vector of node
Acceleration	$\mathbf{a}_{\text{config}} = \ddot{\mathbf{q}}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, \ddot{q}_2]^{\text{T}}_{\text{config}}$	global 3D acceleration vector of node
CoordinatesTotal	$\mathbf{c}_{\text{config}} = \mathbf{u}_{\text{config}} + \mathbf{p}_{\text{ref}}$	displacement plus reference coordinates of node
Coordinates	$\mathbf{c}_{\text{config}} = \mathbf{u}_{\text{config}} = [q_0, q_1, q_2]^{\text{T}}_{\text{config}}$	coordinate vector of node
Coordinates_t	$\dot{\mathbf{c}}_{\text{config}} = \mathbf{v}_{\text{config}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2]^{\text{T}}_{\text{config}}$	velocity coordinates vector of node
Coordinates_tt	$\ddot{\mathbf{c}}_{\text{config}} = \mathbf{a}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, \ddot{q}_2]^{\text{T}}_{\text{config}}$	acceleration coordinates vector of node
RotationMatrix		identity matrix (only for completeness)
Rotation	$[0, 0, 0]$	(only for completeness)
AngularVelocity	$[0, 0, 0]$	(only for completeness)
AngularVelocityLocal	$[0, 0, 0]$	(only for completeness)

Detailed information: The node provides $n_c = 3$ displacement coordinates. Equations of motion need to be provided by an according object (e.g., MassPoint, finite elements, ...). Usually, the nodal coordinates are provided in the global frame. However, the coordinate system is defined by the object (e.g. MassPoint uses global coordinates, but floating frame of reference objects use local frames). Note that for this very simple node, coordinates are identical to the nodal displacements, same for time derivatives. This is not the case, e.g. for nodes with orientation.

Example for NodePoint: see ObjectMassPoint, Section 8.2.2

For examples on NodePoint see Relevant Examples and TestModels with weblink:

- `interactiveTutorial.py` (Examples/)
- `particlesSilo.py` (Examples/)
- `particlesTest.py` (Examples/)

- particlesTest3D.py (Examples/)
- particlesTest3D2.py (Examples/)
- plotSensorExamples.py (Examples/)
- serialRobotKinematicTreeDigging.py (Examples/)
- SpringWithConstraints.py (Examples/)
- ComputeSensitivitiesExample.py (Examples/)
- coordinateSpringDamper.py (Examples/)
- massSpringFrictionInteractive.py (Examples/)
- minimizeExample.py (Examples/)
- ...
- ACFtest.py (TestModels/)
- connectorGravityTest.py (TestModels/)
- generalContactSpheresTest.py (TestModels/)
- ...

8.1.2 NodePoint2D

A 2D point node for point masses or solid finite elements which has 2 displacement degrees of freedom for second order differential equations.

Additional information for NodePoint2D:

- This `Node` has/provides the following types = `Position2D`, `Position`
- **Short name** for Python = `Point2D`
- **Short name** for Python visualization object = `VPoint2D`

The item **NodePoint2D** with type = 'Point2D' has the following parameters:

Name	type	size	default value	description
name	String		''	node's unique name
referenceCoordinates	Vector2D	2	[0.,0.]	reference coordinates of node ==> e.g. ref. coordinates for finite elements; global position of node without displacement
initialCoordinates	Vector2D	2	[0.,0.]	initial displacement coordinate
initialVelocities	Vector2D	2	[0.,0.]	initial velocity coordinate
visualization	VNodePoint2D			parameters for visualization of item

The item VNodePoint2D has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size (diameter, dimensions of underlying cube, etc.) for item; size == -1.f means that default size is used
color	Float4	4	[-1.,-1.,-1.,-1.]	Default RGBA color for nodes; 4th value is alpha-transparency; R=-1.f means, that default color is used

---

8.1.2.1 DESCRIPTION of NodePoint2D:

Information on input parameters:

input parameter	symbol	description see tables above

referenceCoordinates	$\mathbf{q}_{\text{ref}} = [q_0, q_1]^{\text{T}}_{\text{ref}} = \mathbf{p}_{\text{ref}} = [r_0, r_1]^{\text{T}}$	
initialCoordinates	$\mathbf{q}_{\text{ini}} = [q_0, q_1]^{\text{T}}_{\text{ini}} = [u_0, u_1]^{\text{T}}_{\text{ini}}$	
initialVelocities	$\dot{\mathbf{q}}_{\text{ini}} = \mathbf{v}_{\text{ini}} = [\dot{q}_0, \dot{q}_1]^{\text{T}}_{\text{ini}}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
Position	$\mathbf{p}_{\text{config}} = [p_0, p_1, 0]^{\text{T}}_{\text{config}} = \mathbf{u}_{\text{config}} + \mathbf{p}_{\text{ref}}$	global 3D position vector of node; $\mathbf{u}_{\text{ref}} = 0$
Displacement	$\mathbf{u}_{\text{config}} = [q_0, q_1, 0]^{\text{T}}_{\text{config}}$	global 3D displacement vector of node
Velocity	$\mathbf{v}_{\text{config}} = [\dot{q}_0, \dot{q}_1, 0]^{\text{T}}_{\text{config}}$	global 3D velocity vector of node
Acceleration	$\mathbf{a}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, 0]^{\text{T}}_{\text{config}}$	global 3D acceleration vector of node
CoordinatesTotal	$\mathbf{c}_{\text{config}} = \mathbf{u}_{\text{config}} + \mathbf{p}_{\text{ref}}$	displacement plus reference coordinates of node
Coordinates	$\mathbf{c}_{\text{config}} = [q_0, q_1]^{\text{T}}_{\text{config}}$	coordinate vector of node
Coordinates_t	$\dot{\mathbf{c}}_{\text{config}} = [\dot{q}_0, \dot{q}_1]^{\text{T}}_{\text{config}}$	velocity coordinates vector of node
Coordinates_tt	$\ddot{\mathbf{c}}_{\text{config}} = \mathbf{a}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1]^{\text{T}}_{\text{config}}$	acceleration coordinates vector of node
RotationMatrix		identity matrix (only for completeness)
Rotation	[0, 0, 0]	(only for completeness)
AngularVelocity	[0, 0, 0]	(only for completeness)
AngularVelocityLocal	[0, 0, 0]	(only for completeness)

Detailed information: The node provides $n_c = 2$ displacement coordinates. Equations of motion need to be provided by an according object (e.g., MassPoint2D). Coordinates are identical to the nodal displacements, except for the third coordinate u_2, which is zero, because q_2 does not exist.

Note that for this very simple node, coordinates are identical to the nodal displacements, same for time derivatives. This is not the case, e.g. for nodes with orientation.

Example for NodePoint2D: see ObjectMassPoint2D, Section 8.2.3

For examples on NodePoint2D see Relevant Examples and TestModels with weblink:

- `myFirstExample.py` (Examples/)
- `pendulum2Dconstraint.py` (Examples/)
- `pendulumIftommBenchmark.py` (Examples/)
- `sliderCrank3DwithANCFbeltDrive2.py` (Examples/)
- `SpringDamperMassUserFunction.py` (Examples/)
- `xExudynConfigSpecial.py` (Examples/)
- `ANCFslidingJoint2D.py` (Examples/)
- `ANCFslidingJoint2Drigid.py` (Examples/)

- geneticOptimizationSliderCrank.py (Examples/)
- SliderCrank.py (Examples/)
- slidercrankWithMassSpring.py (Examples/)
- switchingConstraintsPendulum.py (Examples/)
- ...
- modelUnitTests.py (TestModels/)
- sparseMatrixSpringDamperTest.py (TestModels/)
- coordinateVectorConstraint.py (TestModels/)
- ...

8.1.3 NodeRigidBodyEP

A 3D rigid body node based on Euler parameters for rigid bodies or beams; the node has 3 displacement coordinates (representing displacement of reference point ${}^0\mathbf{r}$) and four rotation coordinates (Euler parameters = unit quaternions).

Additional information for NodeRigidBodyEP:

- This `Node` has/provides the following types = `Position`, `Orientation`, `RigidBody`, `RotationEulerParameters`
- **Short name** for Python = `RigidEP`
- **Short name** for Python visualization object = `VRigidEP`

The item **NodeRigidBodyEP** with type = 'RigidBodyEP' has the following parameters:

Name	type	size	default value	description
name	String		''	node's unique name
referenceCoordinates	Vector7D	7	[0.,0.,0., 0.,0.,0.,0.]	reference coordinates (3 position coordinates and 4 Euler parameters) of node ==> e.g. ref. coordinates for finite elements or reference position of rigid body (e.g. for definition of joints)
initialCoordinates	Vector7D	7	[0.,0.,0., 0.,0.,0.,0.]	initial displacement coordinates and 4 Euler parameters relative to reference coordinates
initialVelocities	Vector7D	7	[0.,0.,0., 0.,0.,0.,0.]	initial velocity coordinates: time derivatives of initial displacements and Euler parameters
addConstraintEquation	Bool		True	True: automatically add Euler parameter constraint for node; False: Euler parameter constraint is not added, must be done manually (e.g., with CoordinateVectorConstraint)
visualization	VNodeRigidBodyEP			parameters for visualization of item

The item VNodeRigidBodyEP has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size (diameter, dimensions of underlying cube, etc.) for item; size == -1.f means that default size is used

color	Float4	4	[-1.,-1.,-1.,-1.]	Default RGBA color for nodes; 4th value is alpha-transparency; R=-1.f means, that default color is used

---

8.1.3.1 DESCRIPTION of NodeRigidBodyEP:

Information on input parameters:

input parameter	symbol	description see tables above
referenceCoordinates	$\mathbf{q}_{\text{ref}} = [q_0, q_1, q_2, \psi_0, \psi_1, \psi_2, \psi_3]^{\text{T}}_{\text{ref}} = [\mathbf{p}^{\text{T}}_{\text{ref}}, \boldsymbol{\psi}^{\text{T}}_{\text{ref}}]^{\text{T}}$	
initialCoordinates	$\mathbf{q}_{\text{ini}} = [q_0, q_1, q_2, \psi_0, \psi_1, \psi_2, \psi_3]^{\text{T}}_{\text{ini}} = [\mathbf{u}^{\text{T}}_{\text{ini}}, \boldsymbol{\psi}^{\text{T}}_{\text{ini}}]^{\text{T}}$	
initialVelocities	$\dot{\mathbf{q}}_{\text{ini}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2, \dot{\psi}_0, \dot{\psi}_1, \dot{\psi}_2, \dot{\psi}_3]^{\text{T}}_{\text{ini}} = [\dot{\mathbf{u}}^{\text{T}}_{\text{ini}}, \dot{\boldsymbol{\psi}}^{\text{T}}_{\text{ini}}]^{\text{T}}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position	${}^0\mathbf{p}_{\text{config}} = {}^0[p_0, p_1, p_2]^{\text{T}}_{\text{config}} = {}^0\mathbf{u}_{\text{config}} + {}^0\mathbf{p}_{\text{ref}}$	global 3D position vector of node; $\mathbf{u}_{\text{ref}} = 0$
Displacement	${}^0\mathbf{u}_{\text{config}} = [q_0, q_1, q_2]^{\text{T}}_{\text{config}}$	global 3D displacement vector of node
Velocity	${}^0\mathbf{v}_{\text{config}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2]^{\text{T}}_{\text{config}}$	global 3D velocity vector of node
Acceleration	${}^0\mathbf{a}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, \ddot{q}_2]^{\text{T}}_{\text{config}}$	global 3D acceleration vector of node
CoordinatesTotal		displacement/rotation coordinates of node including reference configuration
Coordinates	$\mathbf{c}_{\text{config}} = [q_0, q_1, q_2, \psi_0, \psi_1, \psi_2, \psi_3]^{\text{T}}_{\text{config}}$	coordinate vector of node, having 3 displacement coordinates and 4 Euler parameters
Coordinates_t	$\dot{\mathbf{c}}_{\text{config}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2, \dot{\psi}_0, \dot{\psi}_1, \dot{\psi}_2, \dot{\psi}_3]^{\text{T}}_{\text{config}}$	velocity coordinates vector of node
Coordinates_tt	$\ddot{\mathbf{c}}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, \ddot{q}_2, \ddot{\psi}_0, \ddot{\psi}_1, \ddot{\psi}_2, \ddot{\psi}_3]^{\text{T}}_{\text{config}}$	acceleration coordinates vector of node
RotationMatrix	$[A_{00}, A_{01}, A_{02}, A_{10}, \ldots, A_{21}, A_{22}]^{\text{T}}_{\text{config}}$	vector with 9 components of the rotation matrix ${}^{0b}\mathbf{A}_{\text{config}}$ in row-major format, in any configuration; the rotation matrix transforms local (b) to global (0) coordinates
Rotation	$[\varphi_0, \varphi_1, \varphi_2]^{\text{T}}_{\text{config}}$	vector with 3 components of the Euler/Tait-Bryan angles in xyz-sequence (${}^{0b}\mathbf{A}_{\text{config}} =: \mathbf{A}_0(\varphi_0) \cdot \mathbf{A}_1(\varphi_1) \cdot \mathbf{A}_2(\varphi_2)$), recomputed from rotation matrix
AngularVelocity	${}^0\boldsymbol{\omega}_{\text{config}} = {}^0[\omega_0, \omega_1, \omega_2]^{\text{T}}_{\text{config}}$	global 3D angular velocity vector of node

AngularVelocityLocal	${}^{b}\boldsymbol{\omega}_{\text{config}} = {}^{b}[\omega_0, \omega_1, \omega_2]^{\text{T}}_{\text{config}}$	local (body-fixed) 3D angular velocity vector of node
AngularAcceleration	${}^{0}\boldsymbol{\alpha}_{\text{config}} = {}^{0}[\alpha_0, \alpha_1, \alpha_2]^{\text{T}}_{\text{config}}$	global 3D angular acceleration vector of node

Detailed information: All coordinates $\mathbf{c}_{\text{config}}$ lead to second order differential equations. The first 3 equations are residuals of translational forces in global coordinates, while the last 4 equations are residual of local torques left-multiplied with ${}^{b}\mathbf{G}^{\text{T}}$ or global torques left-multiplied with ${}^{0}\mathbf{G}^{\text{T}}$, see Eq. (8.5), compare the equations of motion of the rigid body.

There is one additional (algebraic) constraint equation for the quaternions. The additional constraint equation, which needs to be provided by the object, reads

$$1 - \sum_{i=0}^{3} \theta_i^2 = 0. \tag{8.1}$$

The rotation matrix ${}^{0b}\mathbf{A}_{\text{config}}$ transforms a local (body-fixed) 3D position ${}^{b}\mathbf{b} = {}^{b}[b_0, b_1, b_2]^{\text{T}}$ to global 3D positions,

$${}^{0}\mathbf{b}_{\text{config}} = {}^{0b}\mathbf{A}_{\text{config}}\, {}^{b}\mathbf{b} \tag{8.2}$$

Note that the Euler parameters $\boldsymbol{\theta}_{\text{cur}}$ are computed as sum of current coordinates plus reference coordinates,

$$\boldsymbol{\theta}_{\text{cur}} = \boldsymbol{\psi}_{\text{cur}} + \boldsymbol{\psi}_{\text{ref}}. \tag{8.3}$$

The rotation matrix is defined as function of the rotation parameters $\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2, \theta_3]^{\text{T}}$

$${}^{0b}\mathbf{A} = \begin{bmatrix} -2\theta_3^2 - 2\theta_2^2 + 1 & -2\theta_3\theta_0 + 2\theta_2\theta_1 & 2*\theta_3\theta_1 + 2*\theta_2\theta_0 \\ 2\theta_3\theta_0 + 2\theta_2\theta_1 & -2\theta_3^2 - 2\theta_1^2 + 1 & 2\theta_3\theta_2 - 2\theta_1\theta_0 \\ -2\theta_2\theta_0 + 2\theta_3\theta_1 & 2\theta_3\theta_2 + 2\theta_1\theta_0 & -2\theta_2^2 - 2\theta_1^2 + 1 \end{bmatrix} \tag{8.4}$$

The derivatives of the angular velocity vectors w.r.t. the rotation velocity coordinates $\dot{\boldsymbol{\theta}} = [\dot{\theta}_0, \dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_3]^{\text{T}}$ lead to the $\mathbf{G}$ matrices, as used in the equations of motion for rigid bodies,

$${}^{0}\boldsymbol{\omega} = {}^{0}\mathbf{G}\,\dot{\boldsymbol{\theta}}, \tag{8.5}$$

$${}^{b}\boldsymbol{\omega} = {}^{b}\mathbf{G}\,\dot{\boldsymbol{\theta}}. \tag{8.6}$$

For creating a `NodeRigidBodyEP` together with a rigid body, there is a `rigidBodyUtilities` function `CreateRigidBody`, see Section 6.5.1, which simplifies the setup of a rigid body significantely!

For examples on NodeRigidBodyEP see Relevant Examples and TestModels with weblink:

- `rigid3Dexample.py` (Examples/)
- `rigidBodyIMUtest.py` (Examples/)

- rigidRotor3DbasicBehaviour.py (Examples/)
- rigidRotor3DFWBW.py (Examples/)
- rigidRotor3Dnutation.py (Examples/)
- rigidRotor3Drunup.py (Examples/)
- addPrismaticJoint.py (Examples/)
- addRevoluteJoint.py (Examples/)
- ANCFrotatingCable2D.py (Examples/)
- ballBearningModel.py (Examples/)
- bicycleIftommBenchmark.py (Examples/)
- bungeeJump.py (Examples/)
- ...
- explicitLieGroupIntegratorPythonTest.py (TestModels/)
- explicitLieGroupIntegratorTest.py (TestModels/)
- explicitLieGroupMBSTest.py (TestModels/)
- ...

8.1.4 NodeRigidBodyRxyz

A 3D rigid body node based on Euler / Tait-Bryan angles for rigid bodies or beams; all coordinates lead to second order differential equations; NOTE that this node has a singularity if the second rotation parameter reaches $\psi_1 = (2k-1)\pi/2$, with $k \in \mathbb{N}$ or $-k \in \mathbb{N}$.

Additional information for NodeRigidBodyRxyz:

- This `Node` has/provides the following types = `Position`, `Orientation`, `RigidBody`, `RotationRxyz`
- **Short name** for Python = `RigidRxyz`
- **Short name** for Python visualization object = `VRigidRxyz`

The item **NodeRigidBodyRxyz** with type = 'RigidBodyRxyz' has the following parameters:

Name	type	size	default value	description
name	String		''	node's unique name
referenceCoordinates	Vector6D	6	[0.,0.,0., 0.,0.,0.]	reference coordinates (3 position and 3 xyz Euler angles) of node ==> e.g. ref. coordinates for finite elements or reference position of rigid body (e.g. for definition of joints)
initialCoordinates	Vector6D	6	[0.,0.,0., 0.,0.,0.]	initial displacement coordinates: ux,uy,uz and 3 Euler angles (xyz) relative to reference coordinates
initialVelocities	Vector6D	6	[0.,0.,0., 0.,0.,0.]	initial velocity coordinate: time derivatives of ux,uy,uz and of 3 Euler angles (xyz)
visualization	VNodeRigidBodyRxyz			parameters for visualization of item

The item VNodeRigidBodyRxyz has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size (diameter, dimensions of underlying cube, etc.) for item; size == -1.f means that default size is used
color	Float4	4	[-1.,-1.,-1.,-1.]	Default RGBA color for nodes; 4th value is alpha-transparency; R=-1.f means, that default color is used

8.1.4.1 DESCRIPTION of NodeRigidBodyRxyz:

Information on input parameters:

input parameter	symbol	description see tables above
referenceCoordinates	$\mathbf{q}_{\text{ref}} = [q_0, q_1, q_2, \psi_0, \psi_1, \psi_2]^{\mathrm{T}}_{\text{ref}} = [\mathbf{p}^{\mathrm{T}}_{\text{ref}}, \boldsymbol{\psi}^{\mathrm{T}}_{\text{ref}}]^{\mathrm{T}}$	
initialCoordinates	$\mathbf{q}_{\text{ini}} = [q_0, q_1, q_2, \psi_0, \psi_1, \psi_2]^{\mathrm{T}}_{\text{ini}} = [\mathbf{u}^{\mathrm{T}}_{\text{ini}}, \boldsymbol{\psi}^{\mathrm{T}}_{\text{ini}}]^{\mathrm{T}}$	
initialVelocities	$\dot{\mathbf{q}}_{\text{ini}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2, \dot{\psi}_0, \dot{\psi}_1, \dot{\psi}_2]^{\mathrm{T}}_{\text{ini}} = [\dot{\mathbf{u}}^{\mathrm{T}}_{\text{ini}}, \dot{\boldsymbol{\psi}}^{\mathrm{T}}_{\text{ini}}]^{\mathrm{T}}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position	${}^0\mathbf{p}_{\text{config}} = {}^0[p_0, p_1, p_2]^{\mathrm{T}}_{\text{config}} = {}^0\mathbf{u}_{\text{config}} + {}^0\mathbf{p}_{\text{ref}}$	global 3D position vector of node; $\mathbf{u}_{\text{ref}} = 0$
Displacement	${}^0\mathbf{u}_{\text{config}} = [q_0, q_1, q_2]^{\mathrm{T}}_{\text{config}}$	global 3D displacement vector of node
Velocity	${}^0\mathbf{v}_{\text{config}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2]^{\mathrm{T}}_{\text{config}}$	global 3D velocity vector of node
Acceleration	${}^0\mathbf{a}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, \ddot{q}_2]^{\mathrm{T}}_{\text{config}}$	global 3D acceleration vector of node
CoordinatesTotal		displacement/rotation coordinates of node including reference configuration
Coordinates	$\mathbf{c}_{\text{config}} = [q_0, q_1, q_2, \psi_0, \psi_1, \psi_2]^{\mathrm{T}}_{\text{config}}$	coordinate vector of node, having 3 displacement coordinates and 3 Euler angles
Coordinates_t	$\dot{\mathbf{c}}_{\text{config}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2, \dot{\psi}_0, \dot{\psi}_1, \dot{\psi}_2]^{\mathrm{T}}_{\text{config}}$	velocity coordinates vector of node
Coordinates_tt	$\ddot{\mathbf{c}}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, \ddot{q}_2, \ddot{\psi}_0, \ddot{\psi}_1, \ddot{\psi}_2]^{\mathrm{T}}_{\text{config}}$	acceleration coordinates vector of node
RotationMatrix	$[A_{00}, A_{01}, A_{02}, A_{10}, \ldots, A_{21}, A_{22}]^{\mathrm{T}}_{\text{config}}$	vector with 9 components of the rotation matrix ${}^{0b}\mathbf{A}_{\text{config}}$ in row-major format, in any configuration; the rotation matrix transforms local (b) to global (0) coordinates
Rotation	$[\varphi_0, \varphi_1, \varphi_2]^{\mathrm{T}}_{\text{config}} = [\psi_0, \psi_1, \psi_2]^{\mathrm{T}}_{\text{ref}} + [\psi_0, \psi_1, \psi_2]^{\mathrm{T}}_{\text{config}}$	vector with 3 components of the Euler / Tait-Bryan angles in xyz-sequence (${}^{0b}\mathbf{A}_{\text{config}} =: \mathbf{A}_0(\varphi_0) \cdot \mathbf{A}_1(\varphi_1) \cdot \mathbf{A}_2(\varphi_2)$)
AngularVelocity	${}^0\boldsymbol{\omega}_{\text{config}} = {}^0[\omega_0, \omega_1, \omega_2]^{\mathrm{T}}_{\text{config}}$	global 3D angular velocity vector of node
AngularVelocityLocal	${}^b\boldsymbol{\omega}_{\text{config}} = {}^b[\omega_0, \omega_1, \omega_2]^{\mathrm{T}}_{\text{config}}$	local (body-fixed) 3D angular velocity vector of node
AngularAcceleration	${}^0\boldsymbol{\alpha}_{\text{config}} = {}^0[\alpha_0, \alpha_1, \alpha_2]^{\mathrm{T}}_{\text{config}}$	global 3D angular acceleration vector of node

Detailed information: The node has 3 displacement coordinates $[q_0, q_1, q_2]^{\mathrm{T}}$ and 3 rotation coordinates $[\psi_0, \psi_1, \psi_2]^{\mathrm{T}}$ for consecutive rotations around the 0, 1 and 2-axis (x, y and z). All coordinates $\mathbf{c}_{\text{config}}$ lead to second order differential equations. The rotation matrix ${}^{0b}\mathbf{A}_{\text{config}}$ transforms a local (body-fixed) 3D position ${}^b\mathbf{b} = {}^b[b_0, b_1, b_2]^{\mathrm{T}}$ to global 3D positions,

$$ {}^0\mathbf{b}_{\text{config}} = {}^{0b}\mathbf{A}_{\text{config}}\, {}^b\mathbf{b} \tag{8.7} $$

Note that the Euler angles $\boldsymbol{\theta}_{\text{cur}}$ are computed as sum of current coordinates plus reference coordinates,

$$\boldsymbol{\theta}_{\text{cur}} = \boldsymbol{\psi}_{\text{cur}} + \boldsymbol{\psi}_{\text{ref}}. \tag{8.8}$$

The rotation matrix is defined as function of the rotation parameters $\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2]^{\mathrm{T}}$

$$^{0b}\mathbf{A} = {}^{01}\mathbf{A}_0(\theta_0)\,{}^{12}\mathbf{A}_1(\theta_1)\,{}^{2b}\mathbf{A}_2(\theta_2) \tag{8.9}$$

see Section 4.0.3 for definition of rotation matrices $\mathbf{A}_0$, $\mathbf{A}_1$ and $\mathbf{A}_2$.

The derivatives of the angular velocity vectors w.r.t. the rotation velocity coordinates $\dot{\boldsymbol{\theta}} = [\dot{\theta}_0, \dot{\theta}_1, \dot{\theta}_2]^{\mathrm{T}}$ lead to the $\mathbf{G}$ matrices, as used in the equations of motion for rigid bodies,

$$^{0}\boldsymbol{\omega} = {}^{0}\mathbf{G}\,\dot{\boldsymbol{\theta}}, \tag{8.10}$$

$$^{b}\boldsymbol{\omega} = {}^{b}\mathbf{G}\,\dot{\boldsymbol{\theta}}. \tag{8.11}$$

For creating a `NodeRigidBodyRxyz` together with a rigid body, there is a `rigidBodyUtilities` function `CreateRigidBody`, see Section 6.5.1, which simplifies the setup of a rigid body significantely!

For examples on NodeRigidBodyRxyz see Relevant Examples and TestModels with weblink:

- `performanceMultiThreadingNG.py` (Examples/)
- `explicitLieGroupIntegratorPythonTest.py` (TestModels/)
- `explicitLieGroupIntegratorTest.py` (TestModels/)
- `explicitLieGroupMBSTest.py` (TestModels/)
- `heavyTop.py` (TestModels/)
- `connectorRigidBodySpringDamperTest.py` (TestModels/)

8.1.5 NodeRigidBodyRotVecLG

A 3D rigid body node based on rotation vector and Lie group methods for rigid bodies; the node has 3 displacement coordinates and three rotation coordinates and can be used in combination with explicit Lie Group time integration methods.

Authors: Gerstmayr Johannes, Holzinger Stefan

Additional information for NodeRigidBodyRotVecLG:

- This `Node` has/provides the following types = `Position`, `Orientation`, `RigidBody`, `RotationRotationVector`
- **Short name** for Python = `RigidRotVecLG`
- **Short name** for Python visualization object = `VRigidRotVecLG`

The item **NodeRigidBodyRotVecLG** with type = 'RigidBodyRotVecLG' has the following parameters:

Name	type	size	default value	description
name	String		''	node's unique name
referenceCoordinates	Vector6D	3	[0.,0.,0., 0.,0.,0.]	reference coordinates (position and rotation vector $\boldsymbol{\nu}$) of node ==> e.g. ref. coordinates for finite elements or reference position of rigid body (e.g. for definition of joints)
initialCoordinates	Vector6D	3	[0.,0.,0., 0.,0.,0.]	initial displacement coordinates $\mathbf{u}$ and rotation vector $\boldsymbol{\nu}$ relative to reference coordinates
initialVelocities	Vector6D	3	[0.,0.,0., 0.,0.,0.]	initial velocity coordinate: time derivatives of displacement and angular velocity vector
visualization	VNodeRigidBodyRotVecLG			parameters for visualization of item

The item VNodeRigidBodyRotVecLG has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size (diameter, dimensions of underlying cube, etc.) for item; size == -1.f means that default size is used

color	Float4	4	[-1.,-1.,-1.,-1.]	Default RGBA color for nodes; 4th value is alpha-transparency; R=-1.f means, that default color is used

---

8.1.5.1 DESCRIPTION of NodeRigidBodyRotVecLG:

Information on input parameters:

input parameter	symbol	description see tables above
referenceCoordinates	$\mathbf{q}_{\text{ref}} = [q_0, q_1, q_2, \nu_0, \nu_1, \nu_2]^{\mathrm{T}}_{\text{ref}} = [\mathbf{p}^{\mathrm{T}}_{\text{ref}}, \boldsymbol{\nu}^{\mathrm{T}}_{\text{ref}}]^{\mathrm{T}}$	
initialCoordinates	$\mathbf{q}_{\text{ini}} = [q_0, q_1, q_2, \nu_0, \nu_1, \nu_2]^{\mathrm{T}}_{\text{ini}} = [\mathbf{u}^{\mathrm{T}}_{\text{ini}}, \boldsymbol{\nu}^{\mathrm{T}}_{\text{ini}}]^{\mathrm{T}}$	
initialVelocities	$\dot{\mathbf{q}}_{\text{ini}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2, \dot{\nu}_0, \dot{\nu}_1, \dot{\nu}_2]^{\mathrm{T}}_{\text{ini}} = [\dot{\mathbf{u}}^{\mathrm{T}}_{\text{ini}}, \dot{\boldsymbol{\nu}}^{\mathrm{T}}_{\text{ini}}]^{\mathrm{T}}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position	${}^0\mathbf{p}_{\text{config}} = {}^0[p_0, p_1, p_2]^{\mathrm{T}}_{\text{config}} = {}^0\mathbf{u}_{\text{config}} + {}^0\mathbf{p}_{\text{ref}}$	global 3D position vector of node; $\mathbf{u}_{\text{ref}} = 0$
Displacement	${}^0\mathbf{u}_{\text{config}} = [q_0, q_1, q_2]^{\mathrm{T}}_{\text{config}}$	global 3D displacement vector of node
Velocity	${}^0\mathbf{v}_{\text{config}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2]^{\mathrm{T}}_{\text{config}}$	global 3D velocity vector of node
Acceleration	${}^0\mathbf{a}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, \ddot{q}_2]^{\mathrm{T}}_{\text{config}}$	global 3D acceleration vector of node
CoordinatesTotal		displacement/rotation coordinates of node including reference configuration
Coordinates	$\mathbf{c}_{\text{config}} = [q_0, q_1, q_2, \nu_0, \nu_1, \nu_2]^{\mathrm{T}}_{\text{config}}$	coordinate vector of node, having 3 displacement coordinates and 3 Euler angles
Coordinates_t	$\dot{\mathbf{c}}_{\text{config}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2, \dot{\nu}_0, \dot{\nu}_1, \dot{\nu}_2]^{\mathrm{T}}_{\text{config}}$	velocity coordinates vector of node
RotationMatrix	$[A_{00}, A_{01}, A_{02}, A_{10}, \ldots, A_{21}, A_{22}]^{\mathrm{T}}_{\text{config}}$	vector with 9 components of the rotation matrix ${}^{0b}\mathbf{A}_{\text{config}}$ in row-major format, in any configuration; the rotation matrix transforms local (b) to global (0) coordinates
Rotation	$[\varphi_0, \varphi_1, \varphi_2]^{\mathrm{T}}_{\text{config}}$	vector with 3 components of the Euler/Tait-Bryan angles in xyz-sequence (${}^{0b}\mathbf{A}_{\text{config}} =: \mathbf{A}_0(\varphi_0) \cdot \mathbf{A}_1(\varphi_1) \cdot \mathbf{A}_2(\varphi_2)$), recomputed from rotation matrix
AngularVelocity	${}^0\boldsymbol{\omega}_{\text{config}} = {}^0[\omega_0, \omega_1, \omega_2]^{\mathrm{T}}_{\text{config}}$	global 3D angular velocity vector of node
AngularVelocityLocal	${}^b\boldsymbol{\omega}_{\text{config}} = {}^b[\omega_0, \omega_1, \omega_2]^{\mathrm{T}}_{\text{config}}$	local (body-fixed) 3D angular velocity vector of node

Detailed information: For a detailed description on the rigid body dynamics formulation using this node, see Holzinger and Gerstmayr [21].

The node has 3 displacement coordinates $[q_0, q_1, q_2]^\mathrm{T}$ and three rotation coordinates, which is the rotation vector

$$\boldsymbol{\nu} = \varphi \mathbf{n} = \boldsymbol{\nu}_\mathrm{config} + \boldsymbol{\nu}_\mathrm{ref}, \tag{8.12}$$

with the rotation angle φ and the rotation axis $\mathbf{n}$. All coordinates $\mathbf{c}_\mathrm{config}$ lead to second order differential equations, However the rotation vector cannot be used as a conventional parameterization. It must be computed within a nonlinear update, using appropriate Lie group methods. The first 3 equations are residuals of translational forces in global coordinates, while the last 3 equations are residual of local (body-fixed) torques, compare the equations of motion of the rigid body.

The rotation matrix ${}^{0b}\mathbf{A}(\boldsymbol{\nu})_\mathrm{config}$ transforms a local (body-fixed) 3D position ${}^b\mathbf{b} = {}^b[b_0, b_1, b_2]^\mathrm{T}$ to global 3D positions,

$${}^0\mathbf{b}_\mathrm{config} = {}^{0b}\mathbf{A}(\boldsymbol{\nu})_\mathrm{config}\, {}^b\mathbf{b} \tag{8.13}$$

Note that $\mathbf{A}(\boldsymbol{\nu})$ is defined in function `RotationVector2RotationMatrix`, see Section 7.19.

A Lie group integrator must be used with this node, which is why the is used, the rotation parameter velocities are identical to the local angular velocity ${}^b\boldsymbol{\omega}$ and thus the matrix ${}^b\mathbf{G}$ becomes the identity matrix.

Note, that the node automatically switches to Lie group integration of its rotational coordinates, both in explicit integration as well as for implicit time integration. This node avoids typical singularities of rotations and is therefore perfectly suited for arbitrary motion. Furthermore, nonlinearities are reduced, which may improve implicit time integration performance.

For creating a `NodeRigidBodyRotVecLG` together with a rigid body, there is a `rigidBodyUtilities` function `CreateRigidBody`, see Section 6.5.1, which simplifies the setup of a rigid body significantely!

For examples on NodeRigidBodyRotVecLG see Relevant Examples and TestModels with weblink:

- `explicitLieGroupIntegratorPythonTest.py` (TestModels/)
- `explicitLieGroupIntegratorTest.py` (TestModels/)
- `explicitLieGroupMBSTest.py` (TestModels/)

8.1.6 NodeRigidBody2D

A 2D rigid body node for rigid bodies or beams; the node has 2 displacement degrees of freedom and one rotation coordinate (rotation around z-axis: uphi). All coordinates are ODE2, used for second order differetial equations.

Additional information for NodeRigidBody2D:

- This `Node` has/provides the following types = `Position2D`, `Orientation2D`, `Position`, `Orientation`, `RigidBody`
- **Short name** for Python = `Rigid2D`
- **Short name** for Python visualization object = `VRigid2D`

The item **NodeRigidBody2D** with type = 'RigidBody2D' has the following parameters:

Name	type	size	default value	description
name	String		"	node's unique name
referenceCoordinates	Vector3D	3	[0.,0.,0.]	reference coordinates (x-pos,y-pos and rotation) of node ==> e.g. ref. coordinates for finite elements; global position of node without displacement
initialCoordinates	Vector3D	3	[0.,0.,0.]	initial displacement coordinates and angle (relative to reference coordinates)
initialVelocities	Vector3D	3	[0.,0.,0.]	initial velocity coordinates
visualization	VNodeRigidBody2D			parameters for visualization of item

The item VNodeRigidBody2D has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size (diameter, dimensions of underlying cube, etc.) for item; size == -1.f means that default size is used
color	Float4	4	[-1.,-1.,-1.,-1.]	Default RGBA color for nodes; 4th value is alpha-transparency; R=-1.f means, that default color is used

8.1.6.1 DESCRIPTION of NodeRigidBody2D:

Information on input parameters:

input parameter	symbol	description see tables above
referenceCoordinates	$\mathbf{q}_{\text{ref}} = [q_0, q_1, \psi_0]^{\text{T}}_{\text{ref}}$	
initialCoordinates	$\mathbf{q}_{\text{ini}} = [q_0, q_1, \psi_0]^{\text{T}}_{\text{ini}}$	
initialVelocities	$\dot{\mathbf{q}}_{\text{ini}} = [\dot{q}_0, \dot{q}_1, \dot{\psi}_0]^{\text{T}}_{\text{ini}} = [v_0, v_1, \omega_2]^{\text{T}}_{\text{ini}}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position	${}^0\mathbf{p}_{\text{config}} = {}^0[p_0, p_1, 0]^{\text{T}}_{\text{config}} = {}^0\mathbf{u}_{\text{config}} + {}^0\mathbf{p}_{\text{ref}}$	global 3D position vector of node; $\mathbf{u}_{\text{ref}} = 0$
Displacement	${}^0\mathbf{u}_{\text{config}} = [q_0, q_1, 0]^{\text{T}}_{\text{config}}$	global 3D displacement vector of node
Velocity	${}^0\mathbf{v}_{\text{config}} = [\dot{q}_0, \dot{q}_1, 0]^{\text{T}}_{\text{config}}$	global 3D velocity vector of node
Acceleration	${}^0\mathbf{a}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, 0]^{\text{T}}_{\text{config}}$	global 3D acceleration vector of node
AngularVelocity	${}^0\boldsymbol{\omega}_{\text{config}} = {}^0[0, 0, \dot{\psi}_0]^{\text{T}}_{\text{config}}$	global 3D angular velocity vector of node
CoordinatesTotal		displacement/rotation coordinates of node including reference configuration
Coordinates	$\mathbf{c}_{\text{config}} = [q_0, q_1, \psi_0]^{\text{T}}_{\text{config}}$	coordinate vector of node, having 2 displacement coordinates and 1 angle
Coordinates_t	$\dot{\mathbf{c}}_{\text{config}} = [\dot{q}_0, \dot{q}_1, \dot{\psi}_0]^{\text{T}}_{\text{config}}$	velocity coordinates vector of node
Coordinates_tt	$\ddot{\mathbf{c}}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, \ddot{\psi}_0]^{\text{T}}_{\text{config}}$	acceleration coordinates vector of node
RotationMatrix	$[A_{00}, A_{01}, A_{02}, A_{10}, \ldots, A_{21}, A_{22}]^{\text{T}}_{\text{config}}$	vector with 9 components of the rotation matrix ${}^{0b}\mathbf{A}_{\text{config}}$ in row-major format, in any configuration; the rotation matrix transforms local (b) to global (0) coordinates
Rotation	$[0, 0, \theta_0]^{\text{T}}_{\text{config}} = [0, 0, \psi_0]^{\text{T}}_{\text{ref}} + [0, 0, \psi_0]^{\text{T}}_{\text{config}}$	vector with 3rd angle around out of plane axis
AngularVelocityLocal	${}^b\boldsymbol{\omega}_{\text{config}} = {}^b[0, 0, \dot{\psi}_0]^{\text{T}}_{\text{config}}$	local (body-fixed) 3D angular velocity vector of node
AngularAcceleration	${}^0\boldsymbol{\alpha}_{\text{config}} = {}^0[0, 0, \ddot{\psi}_0]^{\text{T}}_{\text{config}}$	global 3D angular acceleration vector of node

Detailed information: The node provides 2 displacement coordinates (displacement of center of mass (COM), (q_0, q_1)) and 1 rotation parameter (θ_0). According equations need to be provided by an according object (e.g., RigidBody2D). The node leads to 3 ODE2 equations of motions, where the first 2 equations are residuals of global translational forces, and the third equation is the residual of the torque around the Z-axis (due to planar motion, local=global).

Using the rotation parameter $\theta_{0\text{config}} = \psi_{0ref} + \psi_{0\text{config}}$, the rotation matrix is defined as

$$ {}^{0b}\mathbf{A}_{\text{config}} = \begin{bmatrix} \cos(\theta_0) & -\sin(\theta_0) & 0 \\ \sin(\theta_0) & \cos(\theta_0) & 0 \\ 0 & 0 & 1 \end{bmatrix}_{\text{config}} \tag{8.14} $$

Example for NodeRigidBody2D: see ObjectRigidBody2D

For examples on NodeRigidBody2D see Relevant Examples and TestModels with weblink:

- beltDriveALE.py (Examples/)
- beltDriveReevingSystem.py (Examples/)
- beltDrivesComparison.py (Examples/)
- doublePendulum2D.py (Examples/)
- pendulumGeomExactBeam2D.py (Examples/)
- reevingSystem.py (Examples/)
- reevingSystemOpen.py (Examples/)
- simple4linkPendulumBing.py (Examples/)
- sliderCrank3DwithANCFbeltDrive2.py (Examples/)
- ANCFmovingRigidbody.py (Examples/)
- ANCFslidingJoint2D.py (Examples/)
- ANCFslidingJoint2Drigid.py (Examples/)
- ...
- ANCFBeamEigTest.py (TestModels/)
- ANCFbeltDrive.py (TestModels/)
- ANCFgeneralContactCircle.py (TestModels/)
- ...

8.1.7 Node1D

A node with one ODE2 coordinate for one dimensional (1D) problems; use e.g. for scalar dynamic equations (Mass1D) and mass-spring-damper mechanisms, representing either translational or rotational degrees of freedom: in most cases, Node1D is equivalent to NodeGenericODE2 using one coordinate, however, it offers a transformation to 3D translational or rotational motion and allows to couple this node to 2D or 3D bodies.

Additional information for Node1D:

- This `Node` has/provides the following types = `GenericODE2`

The item **Node1D** with type = '1D' has the following parameters:

Name	type	size	default value	description
name	String		''	node's unique name
referenceCoordinates	Vector		[0.]	reference coordinate of node (in vector form)
initialCoordinates	Vector		[0.]	initial displacement coordinate (in vector form)
initialVelocities	Vector		[0.]	initial velocity coordinate (in vector form)
visualization	VNode1D			parameters for visualization of item

The item VNode1D has the following parameters:

Name	type	size	default value	description
show	Bool		False	set true, if item is shown in visualization and false if it is not shown; The node1D is represented as reference position and displacement along the global x-axis, which must not agree with the representation in the object using the Node1D

8.1.7.1 DESCRIPTION of Node1D:

Information on input parameters:

input parameter	symbol	description see tables above
referenceCoordinates	$[q_0]^T_{ref}$	
initialCoordinates	$[q_0]^T_{ini}$	

initialVelocities	$[\dot{q}_0]^T_{ini}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
CoordinatesTotal		displacement plus reference coordinates of node
Coordinates	$\mathbf{q}_{config} = [q_0]^T_{config}$	ODE2 coordinate of node (in vector form)
Coordinates_t	$\dot{\mathbf{q}}_{config} = [\dot{q}_0]^T_{config}$	ODE2 velocity coordinate of node (in vector form)
Coordinates_tt	$\ddot{\mathbf{q}}_{config} = [\ddot{q}_0]^T_{config}$	ODE2 acceleration coordinate of node (in vector form)

Detailed information: The current position/rotation coordinate of the 1D node is computed from

$$p_0 = q_{0_{ref}} + q_{0_{cur}} \tag{8.15}$$

The coordinate leads to one second order differential equation. The graphical representation and the (internal) position of the node is

$$p_{config} = \begin{bmatrix} p_{0_{config}} \\ 0 \\ 0 \end{bmatrix} \tag{8.16}$$

The (internal) velocity vector is $[p_{0_{config}}, 0, 0]^T$.

For examples on Node1D see Relevant Examples and TestModels with weblink:

- `lugreFrictionTest.py` (Examples/)
- `mpi4pyExample.py` (Examples/)
- `multiprocessingTest.py` (Examples/)
- `nMassOscillator.py` (Examples/)
- `nMassOscillatorEigenmodes.py` (Examples/)
- `nMassOscillatorInteractive.py` (Examples/)
- `coordinateSpringDamperExt.py` (TestModels/)
- `distanceSensor.py` (TestModels/)
- `driveTrainTest.py` (TestModels/)
- `mainSystemUserFunctionsTest.py` (TestModels/)

8.1.8 NodePoint2DSlope1

A 2D point/slope vector node for planar Bernoulli-Euler ANCF (absolute nodal coordinate formulation) beam elements; the node has 4 displacement degrees of freedom (2 for displacement of point node and 2 for the slope vector 'slopex'); all coordinates lead to second order differential equations; the slope vector defines the directional derivative w.r.t the local axial (x) coordinate, denoted as ()'; in straight configuration aligned at the global x-axis, the slope vector reads $\mathbf{r}' = [r'_x \;\; r'_y]^T = [1 \;\; 0]^T$.

Additional information for NodePoint2DSlope1:

- This `Node` has/provides the following types = `Position2D`, `Orientation2D`, `Point2DSlope1`, `Position`, `Orientation`
- **Short name** for Python = `Point2DS1`
- **Short name** for Python visualization object = `VPoint2DS1`

The item **NodePoint2DSlope1** with type = 'Point2DSlope1' has the following parameters:

Name	type	size	default value	description
name	String		"	node's unique name
referenceCoordinates	Vector4D	4	[0.,0.,1.,0.]	reference coordinates (x-pos,y-pos; x-slopex, y-slopex) of node; global position of node without displacement
initialCoordinates	Vector4D	4	[0.,0.,0.,0.]	initial displacement coordinates: ux, uy and x/y 'displacements' of slopex
initialVelocities	Vector4D	4	[0.,0.,0.,0.]	initial velocity coordinates
visualization	VNodePoint2DSlope1			parameters for visualization of item

The item VNodePoint2DSlope1 has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size (diameter, dimensions of underlying cube, etc.) for item; size == -1.f means that default size is used
color	Float4	4	[-1.,-1.,-1.,-1.]	Default RGBA color for nodes; 4th value is alpha-transparency; R=-1.f means, that default color is used

8.1.8.1 DESCRIPTION of NodePoint2DSlope1:

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position	${}^0\mathbf{p}_{\text{config}} = [p_0, p_1, 0]^T_{\text{config}}$	global 3D position vector of node (=displacement+reference position)
Displacement	${}^0\mathbf{u}_{\text{config}} = [q_0, q_1, 0]^T_{\text{config}}$	global 3D displacement vector of node
Velocity	${}^0\mathbf{v}_{\text{config}} = [\dot{q}_0, \dot{q}_1, 0]^T_{\text{config}}$	global 3D velocity vector of node
Acceleration	${}^0\mathbf{a}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, 0]^T_{\text{config}}$	global 3D acceleration vector of node
CoordinatesTotal		displacement plus reference coordinates of node
Coordinates		coordinates vector of node (2 displacement coordinates + 2 slope vector coordinates)
Coordinates_t		velocity coordinates vector of node (derivative of the 2 displacement coordinates + 2 slope vector coordinates)
Coordinates_tt		acceleration coordinates vector of node (derivative of the 2 displacement coordinates + 2 slope vector coordinates)

For examples on NodePoint2DSlope1 see Relevant Examples and TestModels with weblink:

- `ALEANCFpipe.py` (Examples/)
- `ANCFcantileverTestDyn.py` (Examples/)
- `ANCFcontactCircle.py` (Examples/)
- `ANCFcontactCircle2.py` (Examples/)
- `ANCFmovingRigidbody.py` (Examples/)
- `ANCFslidingJoint2D.py` (Examples/)
- `ANCFslidingJoint2Drigid.py` (Examples/)
- `ANCFswitchingSlidingJoint2D.py` (Examples/)
- `ANCFtestHalfcircle.py` (Examples/)
- `ANCFtests2.py` (Examples/)
- `sliderCrank3DwithANCFbeltDrive.py` (Examples/)
- `solverFunctionsTestEigenvalues.py` (Examples/)
- ...
- `ANCFcontactCircleTest.py` (TestModels/)
- `ANCFcontactFrictionTest.py` (TestModels/)
- `computeODE2EigenvaluesTest.py` (TestModels/)
- ...

8.1.9 NodePointSlope1

A 3D point/slope vector node for spatial Bernoulli-Euler ANCF (absolute nodal coordinate formulation) beam elements; the node has 6 displacement degrees of freedom (3 for displacement of point node and 3 for the slope vector 'slopex'); all coordinates lead to second order differential equations; the slope vector defines the directional derivative w.r.t the local axial (x) coordinate, denoted as ()'; in straight configuration aligned at the global x-axis, the slope vector reads $\mathbf{r}' = [r'_x \;\; r'_y \;\; r'_z]^T = [1 \;\; 0]^T$.

Additional information for NodePointSlope1:

- This `Node` has/provides the following types = `Position`

The item **NodePointSlope1** with type = 'PointSlope1' has the following parameters:

Name	type	size	default value	description
name	String		''	node's unique name
referenceCoordinates	Vector6D	6	[0.,0.,0.,1.,0.,0.]	reference coordinates (x-pos,y-pos,z-pos; x-slopex, y-slopex, z-slopex) of node; global position of node without displacement
initialCoordinates	Vector6D	6	[0.,0.,0.,0.,0.,0.]	initial displacement coordinates: ux, uy, uz and x/y/z 'displacements' of slopex
initialVelocities	Vector6D	6	[0.,0.,0.,0.,0.,0.]	initial velocity coordinates
visualization	VNodePointSlope1			parameters for visualization of item

The item VNodePointSlope1 has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size (diameter, dimensions of underlying cube, etc.) for item; size == -1.f means that default size is used
color	Float4	4	[-1.,-1.,-1.,-1.]	Default RGBA color for nodes; 4th value is alpha-transparency; R=-1.f means, that default color is used

8.1.9.1 DESCRIPTION of NodePointSlope1:

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position	${}^{0}\mathbf{p}_{\text{config}} = [p_0, p_1, p_2]^{\text{T}}_{\text{config}}$	global 3D position vector of node (=displacement+reference position)
Displacement	${}^{0}\mathbf{u}_{\text{config}} = [q_0, q_1, q_2]^{\text{T}}_{\text{config}}$	global 3D displacement vector of node
Velocity	${}^{0}\mathbf{a}_{\text{config}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2]^{\text{T}}_{\text{config}}$	global 3D velocity vector of node
Acceleration	${}^{0}\mathbf{a}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, \ddot{q}_2]^{\text{T}}_{\text{config}}$	global 3D acceleration vector of node
CoordinatesTotal		displacement plus reference coordinates of node
Coordinates		coordinates vector of node (3 displacement coordinates + 3 slope vector coordinates)
Coordinates_t		velocity coordinates vector of node (derivative of the 3 displacement coordinates + 3 slope vector coordinates)
Coordinates_tt		acceleration coordinates vector of node (derivative of the 3 displacement coordinates + 3 slope vector coordinates)

8.1.10 NodePointSlope12

A 3D point/slope vector node for thin ANCF (absolute nodal coordinate formulation) plate elements; the node has 9 ODE2 degrees of freedom (3 for displacement of point node and 2×3 for the slope vectors 'slopeX' and 'slopeY'); all coordinates lead to second order differential equations; the slopeX vector defines the directional derivative w.r.t the local axial (x) coordinate, etc.; in straight configuration aligned at the global x-axis, the slopeY vector reads $\mathbf{r}'_y = [0 \;\; 1 \;\; 0]^T$.

Additional information for NodePointSlope12:

- This `Node` has/provides the following types = `Position`, `Orientation`

The item **NodePointSlope12** with type = 'PointSlope12' has the following parameters:

Name	type	size	default value	description
name	String		"	node's unique name
referenceCoordinates	Vector9D	9	[0.,0.,0.,1.,0.,0.,1.,0.,0.]	reference coordinates (x-pos,y-pos,z-pos; x-slopeX, y-slopeX, z-slopeX; x-slopeY, y-slopeY, z-slopeY) of node; global position of node without displacement
initialCoordinates	Vector9D	9	[0.,0.,0.,0.,0.,0.,0.,0.,0.]	initial displacement coordinates relative to reference coordinates
initialVelocities	Vector9D	9	[0.,0.,0.,0.,0.,0.,0.,0.,0.]	initial velocity coordinates
visualization	VNodePointSlope12			parameters for visualization of item

The item VNodePointSlope12 has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size (diameter, dimensions of underlying cube, etc.) for item; size == -1.f means that default size is used
color	Float4	4	[-1.,-1.,-1.,-1.]	Default RGBA color for nodes; 4th value is alpha-transparency; R=-1.f means, that default color is used

8.1.10.1 DESCRIPTION of NodePointSlope12:

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position	${}^{0}\mathbf{p}_{\text{config}} = {}^{0}[p_0, p_1, p_2]^{\text{T}}_{\text{config}}$	global 3D position vector of node (=displacement+reference position)
Displacement	${}^{0}\mathbf{u}_{\text{config}} = {}^{0}[q_0, q_1, q_2]^{\text{T}}_{\text{config}}$	global 3D displacement vector of node
Velocity	${}^{0}\mathbf{a}_{\text{config}} = {}^{0}[\dot{q}_0, \dot{q}_1, \dot{q}_2]^{\text{T}}_{\text{config}}$	global 3D velocity vector of node
Acceleration	${}^{0}\mathbf{a}_{\text{config}} = {}^{0}[\ddot{q}_0, \ddot{q}_1, \ddot{q}_2]^{\text{T}}_{\text{config}}$	global 3D acceleration vector of node
CoordinatesTotal		displacement plus reference coordinates of node
Coordinates		coordinate vector of node (relative to reference configuration)
Coordinates_t		velocity coordinates vector of node
Coordinates_tt		acceleration coordinates vector of node
RotationMatrix	$[A_{00}, A_{01}, A_{02}, A_{10}, \ldots, A_{21}, A_{22}]^{\text{T}}_{\text{config}}$	vector with 9 components of the rotation matrix ${}^{0b}\mathbf{A}_{\text{config}}$ in row-major format, in any configuration; the rotation matrix transforms local (b) to global (0) coordinates
Rotation	$[\varphi_0, \varphi_1, \varphi_2]^{\text{T}}_{\text{config}}$	vector with 3 components of the Euler / Tait-Bryan angles in xyz-sequence
AngularVelocity	${}^{0}\boldsymbol{\omega}_{\text{config}} = {}^{0}[\omega_0, \omega_1, \omega_2]^{\text{T}}_{\text{config}}$	global 3D angular velocity vector of node
AngularVelocityLocal	${}^{b}\boldsymbol{\omega}_{\text{config}} = {}^{b}[\omega_0, \omega_1, \omega_2]^{\text{T}}_{\text{config}}$	local (body-fixed) 3D angular velocity vector of node

8.1.11 NodePointSlope23

A 3D point/slope vector node for spatial, shear and cross-section deformable ANCF (absolute nodal coordinate formulation) beam elements; the node has 9 ODE2 degrees of freedom (3 for displacement of point node and 2×3 for the slope vectors 'slopeY' and 'slopeZ'); all coordinates lead to second order differential equations; the slopeY vector defines the directional derivative w.r.t the local axial (y) coordinate, etc.; the slopeY vector reads $\mathbf{r}'_y = [0 \;\; 1 \;\; 0]^T$ and slopeZ gets $\mathbf{r}'_z = [0 \;\; 0 \;\; 1]^T$.

Additional information for NodePointSlope23:

- This `Node` has/provides the following types = `Position`, `Orientation`

The item **NodePointSlope23** with type = 'PointSlope23' has the following parameters:

Name	type	size	default value	description
name	String		"	node's unique name
referenceCoordinates	Vector9D	9	[0.,0.,0.,1.,0.,0.,1.,0.,0.]	reference coordinates (x-pos,y-pos,z-pos; x-slopey, y-slopey, z-slopey; x-slopez, y-slopez, z-slopez) of node; global position of node without displacement
initialCoordinates	Vector9D	9	[0.,0.,0.,0.,0.,0.,0.,0.,0.]	initial displacement coordinates relative to reference coordinates
initialVelocities	Vector9D	9	[0.,0.,0.,0.,0.,0.,0.,0.,0.]	initial velocity coordinates
visualization	VNodePointSlope23			parameters for visualization of item

The item VNodePointSlope23 has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size (diameter, dimensions of underlying cube, etc.) for item; size == -1.f means that default size is used
color	Float4	4	[-1.,-1.,-1.,-1.]	Default RGBA color for nodes; 4th value is alpha-transparency; R=-1.f means, that default color is used

8.1.11.1 DESCRIPTION of NodePointSlope23:

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position	${}^{0}\mathbf{p}_{\text{config}} = {}^{0}[p_0, p_1, p_2]^{\text{T}}_{\text{config}}$	global 3D position vector of node (=displacement+reference position)
Displacement	${}^{0}\mathbf{u}_{\text{config}} = {}^{0}[q_0, q_1, q_2]^{\text{T}}_{\text{config}}$	global 3D displacement vector of node
Velocity	${}^{0}\mathbf{a}_{\text{config}} = {}^{0}[\dot{q}_0, \dot{q}_1, \dot{q}_2]^{\text{T}}_{\text{config}}$	global 3D velocity vector of node
Acceleration	${}^{0}\mathbf{a}_{\text{config}} = {}^{0}[\ddot{q}_0, \ddot{q}_1, \ddot{q}_2]^{\text{T}}_{\text{config}}$	global 3D acceleration vector of node
CoordinatesTotal		displacement plus reference coordinates of node
Coordinates		coordinate vector of node (relative to reference configuration)
Coordinates_t		velocity coordinates vector of node
Coordinates_tt		acceleration coordinates vector of node
RotationMatrix	$[A_{00}, A_{01}, A_{02}, A_{10}, \ldots, A_{21}, A_{22}]^{\text{T}}_{\text{config}}$	vector with 9 components of the rotation matrix ${}^{0b}\mathbf{A}_{\text{config}}$ in row-major format, in any configuration; the rotation matrix transforms local (b) to global (0) coordinates
Rotation	$[\varphi_0, \varphi_1, \varphi_2]^{\text{T}}_{\text{config}}$	vector with 3 components of the Euler / Tait-Bryan angles in xyz-sequence
AngularVelocity	${}^{0}\boldsymbol{\omega}_{\text{config}} = {}^{0}[\omega_0, \omega_1, \omega_2]^{\text{T}}_{\text{config}}$	global 3D angular velocity vector of node
AngularVelocityLocal	${}^{b}\boldsymbol{\omega}_{\text{config}} = {}^{b}[\omega_0, \omega_1, \omega_2]^{\text{T}}_{\text{config}}$	local (body-fixed) 3D angular velocity vector of node

---

For examples on NodePointSlope23 see Relevant Examples and TestModels with weblink:

- `ANCFBeamEigTest.py` (TestModels/)
- `ANCFBeamTest.py` (TestModels/)
- `geometricallyExactBeamTest.py` (TestModels/)
- `rightAngleFrame.py` (TestModels/)

8.1.12 NodeGenericODE2

A node containing a number of ODE2 variables; use e.g. for scalar dynamic equations (Mass1D) or for the ALECable element. Note that referenceCoordinates and all initialCoordinates(_t) must be initialized, because no default values exist.

Additional information for NodeGenericODE2:

- This `Node` has/provides the following types = `GenericODE2`

The item **NodeGenericODE2** with type = 'GenericODE2' has the following parameters:

Name	type	size	default value	description
name	String		''	node's unique name
referenceCoordinates	Vector		[]	generic reference coordinates of node; must be consistent with numberOfODE2Coordinates
initialCoordinates	Vector		[]	initial displacement coordinates; must be consistent with numberOfODE2Coordinates
initialCoordinates_t	Vector		[]	initial velocity coordinates; must be consistent with numberOfODE2Coordinates
numberOfODE2Coordinates	PInt		0	number of generic ODE2 coordinates
visualization	VNodeGenericODE2			parameters for visualization of item

The item VNodeGenericODE2 has the following parameters:

Name	type	size	default value	description
show	Bool		False	set true, if item is shown in visualization and false if it is not shown

---

8.1.12.1 DESCRIPTION of NodeGenericODE2:

Information on input parameters:

input parameter	symbol	description see tables above
referenceCoordinates	$\mathbf{q}_{\text{ref}} = [q_0, \ldots, q_{nc}]^{\text{T}}_{\text{ref}}$	
initialCoordinates	$\mathbf{q}_{\text{ini}} = [q_0, \ldots, q_{nc}]^{\text{T}}_{\text{ini}}$	
initialCoordinates_t	$\dot{\mathbf{q}}_{\text{ini}} = [\dot{q}_0, \ldots, \dot{q}_{nc}]^{\text{T}}_{\text{ini}}$	
numberOfODE2Coordinates	n_c	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
CoordinatesTotal		displacement plus reference coordinates of node
Coordinates	$\mathbf{q}_{\text{config}} = [q_0, \ldots, q_{nc}]^{\mathrm{T}}_{\text{config}}$	coordinates vector of node
Coordinates_t	$\dot{\mathbf{q}}_{\text{config}} = [\dot{q}_0, \ldots, \dot{q}_{nc}]^{\mathrm{T}}_{\text{config}}$	velocity coordinates vector of node
Coordinates_tt	$\ddot{\mathbf{q}}_{\text{config}} = [\ddot{q}_0, \ldots, \ddot{q}_{nc}]^{\mathrm{T}}_{\text{config}}$	acceleration coordinates vector of node

---

For examples on NodeGenericODE2 see Relevant Examples and TestModels with weblink:

- `ALEANCFpipe.py` (Examples/)
- `ANCFALEtest.py` (Examples/)
- `ANCFmovingRigidbody.py` (Examples/)
- `beltDriveALE.py` (Examples/)
- `craneReevingSystem.py` (Examples/)
- `kinematicTreeAndMBS.py` (Examples/)
- `nMassOscillator.py` (Examples/)
- `nMassOscillatorInteractive.py` (Examples/)
- `reinforcementLearningRobot.py` (Examples/)
- `simulateInteractively.py` (Examples/)
- `stiffFlyballGovernorKT.py` (Examples/)
- `ANCFmovingRigidBodyTest.py` (TestModels/)
- `ANCFslidingAndALEjointTest.py` (TestModels/)
- `kinematicTreeTest.py` (TestModels/)
- `solverExplicitODE1ODE2test.py` (TestModels/)
- ...

8.1.13 NodeGenericODE1

A node containing a number of ODE1 variables; use e.g. linear state space systems. Note that referenceCoordinates and initialCoordinates must be initialized, because no default values exist.

The item **NodeGenericODE1** with type = 'GenericODE1' has the following parameters:

Name	type	size	default value	description
name	String		"	node's unique name
referenceCoordinates	Vector		[]	generic reference coordinates of node; must be consistent with numberOfODE1Coordinates
initialCoordinates	Vector		[]	initial displacement coordinates; must be consistent with numberOfODE1Coordinates
numberOfODE1Coordinates	PInt		0	number of generic ODE1 coordinates
visualization	VNodeGenericODE1			parameters for visualization of item

The item VNodeGenericODE1 has the following parameters:

Name	type	size	default value	description
show	Bool		False	set true, if item is shown in visualization and false if it is not shown

---

8.1.13.1 DESCRIPTION of NodeGenericODE1:

Information on input parameters:

input parameter	symbol	description see tables above
referenceCoordinates	$\mathbf{y}_{\text{ref}} = [y_0, \ldots, y_{nc}]^{\text{T}}_{\text{ref}}$	
initialCoordinates	$\mathbf{y}_{\text{ini}} = [y_0, \ldots, y_{nc}]^{\text{T}}_{\text{ini}}$	
numberOfODE1Coordinates	n_c	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
CoordinatesTotal		displacement plus reference coordinates of node
Coordinates	$\mathbf{y}_{\text{config}} = [y_0, \ldots, y_{nc}]^{\text{T}}_{\text{config}}$	ODE1 coordinates vector of node
Coordinates_t	$\dot{\mathbf{y}}_{\text{config}} = [\dot{y}_0, \ldots, \dot{y}_{nc}]^{\text{T}}_{\text{config}}$	ODE1 velocity coordinates vector of node

For examples on NodeGenericODE1 see Relevant Examples and TestModels with weblink:

- HydraulicActuator2Arms.py (Examples/)
- HydraulicActuatorStaticInitialization.py (Examples/)
- HydraulicsUserFunction.py (Examples/)
- lugreFrictionODE1.py (Examples/)
- lugreFrictionTest.py (Examples/)
- hydraulicActuatorSimpleTest.py (TestModels/)
- solverExplicitODE1ODE2test.py (TestModels/)
- taskmanagerTest.py (TestModels/)

8.1.14 NodeGenericAE

A node containing a number of AE variables; use e.g. linear state space systems. Note that referenceCoordinates and initialCoordinates must be initialized, because no default values exist.

The item **NodeGenericAE** with type = 'GenericAE' has the following parameters:

Name	type	size	default value	description
name	String		"	node's unique name
referenceCoordinates	Vector		[]	generic reference coordinates of node; must be consistent with numberOfAECoordinates
initialCoordinates	Vector		[]	initial displacement coordinates; must be consistent with numberOfAECoordinates
numberOfAECoordinates	PInt		0	number of generic AE coordinates
visualization	VNodeGenericAE			parameters for visualization of item

The item VNodeGenericAE has the following parameters:

Name	type	size	default value	description
show	Bool		False	set true, if item is shown in visualization and false if it is not shown

---

8.1.14.1 DESCRIPTION of NodeGenericAE:

Information on input parameters:

input parameter	symbol	description see tables above
referenceCoordinates	$\mathbf{y}_{\text{ref}} = [y_0, \ldots, y_{nc}]^{\text{T}}_{\text{ref}}$	
initialCoordinates	$\mathbf{y}_{\text{ini}} = [y_0, \ldots, y_{nc}]^{\text{T}}_{\text{ini}}$	
numberOfAECoordinates	n_c	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Coordinates	$\mathbf{y}_{\text{config}} = [y_0, \ldots, y_{nc}]^{\text{T}}_{\text{config}}$	AE coordinates vector of node

8.1.15 NodeGenericData

A node containing a number of data (history) variables; use e.g. for contact (active set), friction or plasticity (history variable).

Additional information for NodeGenericData:

- This `Node` has/provides the following types = `GenericData`

The item **NodeGenericData** with type = 'GenericData' has the following parameters:

Name	type	size	default value	description
name	String		"	node's unique name
initialCoordinates	Vector		[]	initial data coordinates
numberOfDataCoordinates	UInt		0	number of generic data coordinates (history variables)
visualization	VNodeGenericData			parameters for visualization of item

The item VNodeGenericData has the following parameters:

Name	type	size	default value	description
show	Bool		False	set true, if item is shown in visualization and false if it is not shown

---

8.1.15.1 DESCRIPTION of NodeGenericData:

Information on input parameters:

input parameter	symbol	description see tables above
initialCoordinates	$\mathbf{x}_{\text{ini}} = [x_0, \ldots, x_{n_c}]^{\text{T}}_{\text{ini}}$	
numberOfDataCoordinates	n_c	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Coordinates	$\mathbf{x}_{\text{config}} = [x_0, \ldots, x_{nc}]^{\text{T}}_{\text{config}}$	data coordinates (history variables) vector of node

For examples on NodeGenericData see Relevant Examples and TestModels with weblink:

- ANCFcontactCircle.py (Examples/)
- ANCFcontactCircle2.py (Examples/)
- ANCFmovingRigidbody.py (Examples/)
- ANCFslidingJoint2D.py (Examples/)
- ANCFslidingJoint2Drigid.py (Examples/)
- ANCFswitchingSlidingJoint2D.py (Examples/)
- ballBearningModel.py (Examples/)
- beltDriveALE.py (Examples/)
- beltDriveReevingSystem.py (Examples/)
- beltDrivesComparison.py (Examples/)
- bicycleIftommBenchmark.py (Examples/)
- camFollowerExample.py (Examples/)
- ...
- ANCFcontactCircleTest.py (TestModels/)
- ANCFcontactFrictionTest.py (TestModels/)
- ANCFmovingRigidBodyTest.py (TestModels/)
- ...

8.1.16 NodePointGround

A 3D point node fixed to ground. The node can be used as NodePoint, but it does not generate coordinates. Applied or reaction forces do not have any effect. This node can be used for 'blind' or 'dummy' ODE2 and ODE1 coordinates to which CoordinateSpringDamper or CoordinateConstraint objects are attached to.

Additional information for NodePointGround:

- This `Node` has/provides the following types = `Ground`, `Position2D`, `Position`, `Orientation`, `GenericODE2`
- **Short name** for Python = `PointGround`
- **Short name** for Python visualization object = `VPointGround`

The item **NodePointGround** with type = 'PointGround' has the following parameters:

Name	type	size	default value	description
name	String		''	node's unique name
referenceCoordinates	Vector3D	3	[0.,0.,0.]	reference coordinates of node ==> e.g. ref. coordinates for finite elements; global position of node without displacement
visualization	VNodePointGround			parameters for visualization of item

The item VNodePointGround has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size (diameter, dimensions of underlying cube, etc.) for item; size == -1.f means that default size is used
color	Float4	4	[-1.,-1.,-1.,-1.]	Default RGBA color for nodes; 4th value is alpha-transparency; R=-1.f means, that default color is used

8.1.16.1 DESCRIPTION of NodePointGround:

Information on input parameters:

input parameter	symbol	description see tables above
referenceCoordinates	$\mathbf{q}_{\text{ref}} = [q_0, q_1, q_2]^{\text{T}}_{\text{ref}} = \mathbf{p}_{\text{ref}} = [r_0, r_1, r_2]^{\text{T}}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position	$\mathbf{p}_{\text{config}} = [p_0, p_1, p_2]^{\text{T}}_{\text{config}} = \mathbf{p}_{\text{ref}}$	global 3D position vector of node (=reference position)
Displacement	$\mathbf{u}_{\text{config}} = [0, 0, 0]^{\text{T}}_{\text{config}}$	zero 3D vector
Velocity	$\mathbf{v}_{\text{config}} = [0, 0, 0]^{\text{T}}_{\text{config}}$	zero 3D vector
CoordinatesTotal	$\mathbf{c}_{\text{config}} = []$	vector of length zero
Coordinates	$\mathbf{c}_{\text{config}} = []$	vector of length zero
Coordinates_t	$\dot{\mathbf{c}}_{\text{config}} = []$	vector of length zero
RotationMatrix		identity matrix (only for completeness)
Rotation	$[0, 0, 0]$	(only for completeness)
AngularVelocity	$[0, 0, 0]$	(only for completeness)
AngularVelocityLocal	$[0, 0, 0]$	(only for completeness)

For examples on NodePointGround see Relevant Examples and TestModels with weblink:

- `ALEANCFpipe.py` (Examples/)
- `ANCFALEtest.py` (Examples/)
- `ANCFcableCantilevered.py` (Examples/)
- `ANCFcantileverTestDyn.py` (Examples/)
- `ANCFcontactCircle.py` (Examples/)
- `ANCFcontactCircle2.py` (Examples/)
- `ANCFmovingRigidbody.py` (Examples/)
- `ANCFrotatingCable2D.py` (Examples/)
- `ANCFslidingJoint2D.py` (Examples/)
- `ANCFslidingJoint2Drigid.py` (Examples/)
- `ANCFswitchingSlidingJoint2D.py` (Examples/)
- `ANCFtestHalfcircle.py` (Examples/)
- ...
- `ANCFBeamEigTest.py` (TestModels/)
- `ANCFBeamTest.py` (TestModels/)
- `ANCFbeltDrive.py` (TestModels/)
- ...

8.2 Objects (Body)

A Body is a special Object, which has physical properties such as mass. A localPosition can be measured w.r.t. the reference point of the body

8.2.1 ObjectGround

A ground object behaving like a rigid body, but having no degrees of freedom; used to attach body-connectors without an action. For examples see spring dampers and joints.

Additional information for ObjectGround:

- This `Object` has/provides the following types = `Ground`, `Body`

The item **ObjectGround** with type = 'Ground' has the following parameters:

Name	type	size	default value	description
name	String		''	objects's unique name
referencePosition	Vector3D	3	[0.,0.,0.]	reference point = reference position for ground object; local position is added on top of reference position for a ground object
referenceRotation	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	the constant ground rotation matrix, which transforms body-fixed (b) to global (0) coordinates
visualization	VObjectGround			parameters for visualization of item

The item VObjectGround has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
graphicsDataUserFunction	PyFunctionGraphicsData		0	A Python function which returns a body-GraphicsData object, which is a list of graphics data in a dictionary computed by the user function
graphicsData	BodyGraphicsData			Structure contains data for body visualization; data is defined in special list / dictionary structure

8.2.1.1 DESCRIPTION of ObjectGround:

Information on input parameters:

input parameter	symbol	description see tables above
referencePosition	${}^{0}\mathbf{r}$	
referenceRotation	${}^{0b}\mathbf{A} \in \mathbb{R}^{3\times3}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position	${}^{0}\mathbf{p} = {}^{0}\mathbf{r} + {}^{0b}\mathbf{A}\ {}^{b}\mathbf{b}$	global position vector of translated local position
Displacement	**0**	global displacement vector of local position
Velocity	**0**	global velocity vector of local position
AngularVelocity	**0**	angular velocity of body
RotationMatrix	${}^{0b}\mathbf{A}$	rotation matrix in vector form (stored in row-major order)

8.2.1.2 Equations

ObjectGround has no equations, as it only provides a static object, at which joints and connectors can be attached. The object does not move (in general) and forces or torques do not have an effect. However, the reference position and rotation may be changed over time. This may prescribe motion, however, with the measured velocity still being zero at each time instant. Therefore, such manipulation of reference position or rotation shall be treated with care.

In combination with markers, the `localPosition` ${}^{b}\mathbf{b}$ is transformed by the `ObjectGround` to a global point ${}^{0}\mathbf{p}$ using the reference point ${}^{0}\mathbf{r}$,

$$ {}^{0}\mathbf{p} = {}^{0}\mathbf{r} + {}^{0b}\mathbf{A}\ {}^{b}\mathbf{b} \ . \tag{8.17} $$

Userfunction: `graphicsDataUserFunction(mbs, itemNumber)`

A user function, which is called by the visualization thread in order to draw user-defined objects. The function can be used to generate any `BodyGraphicsData`, see Section 10.4. Use `exudyn.graphics` functions, see Section 7.8, to create more complicated objects. Note that `graphicsDataUserFunction` needs to copy lots of data and is therefore inefficient and only designed to enable simpler tests, but not large scale problems.

arguments / return	type or size	description

mbs	MainSystem	provides reference to mbs, which can be used in user function to access all data of the object
itemNumber	Index	integer number of the object in mbs, allowing easy access
return value	BodyGraphicsData	list of GraphicsData dictionaries, see Section 10.4

User function example:

```
import exudyn as exu
from math import sin, cos, pi
from exudyn.utilities import * #includes itemInterface and rigidBodyUtilities
import exudyn.graphics as graphics

SC = exu.SystemContainer()
mbs = SC.AddSystem()
#create simple system:
mbs.AddNode(NodePoint())
body = mbs.AddObject(MassPoint(physicsMass=1, nodeNumber=0))

#user function for moving graphics:
def UFgraphics(mbs, objectNum):
    t = mbs.systemData.GetTime(exu.ConfigurationType.Visualization) #get time if
needed
    #draw moving sphere on ground
    graphics1=graphics.Sphere(point=[sin(t*2*pi), cos(t*2*pi), 0],
                              radius=0.1, color=graphics.color.red, nTiles=32)
    return [graphics1]

#add object with graphics user function
ground = mbs.AddObject(ObjectGround(visualization=VObjectGround(
graphicsDataUserFunction=UFgraphics)))
mbs.Assemble()
sims=exu.SimulationSettings()
sims.timeIntegration.numberOfSteps = 10000000 #many steps to see graphics
SC.renderer.Start() #perform zoom all (press 'a' several times) after startup to see
 the sphere
mbs.SolveDynamic(sims)
SC.renderer.Stop()
```

For examples on ObjectGround see Relevant Examples and TestModels with weblink:

- addPrismaticJoint.py (Examples/)

- addRevoluteJoint.py (Examples/)
- ALEANCFpipe.py (Examples/)
- ANCFcableCantilevered.py (Examples/)
- ANCFcantileverTest.py (Examples/)
- ANCFcantileverTestDyn.py (Examples/)
- ANCFcontactCircle.py (Examples/)
- ANCFcontactCircle2.py (Examples/)
- ANCFmovingRigidbody.py (Examples/)
- ANCFrotatingCable2D.py (Examples/)
- ANCFslidingJoint2D.py (Examples/)
- ANCFslidingJoint2Drigid.py (Examples/)
- ...
- abaqusImportTest.py (TestModels/)
- ACFtest.py (TestModels/)
- ANCFbeltDrive.py (TestModels/)
- ...

8.2.2 ObjectMassPoint

A 3D mass point which is attached to a position-based node, usually NodePoint.

Additional information for ObjectMassPoint:

- This `Object` has/provides the following types = `Body`, `SingleNoded`
- Requested `Node` type = `Position`
- **Short name** for Python = `MassPoint`
- **Short name** for Python visualization object = `VMassPoint`

The item **ObjectMassPoint** with type = 'MassPoint' has the following parameters:

Name	type	size	default value	description
name	String		''	objects's unique name
physicsMass	UReal		0.	mass [SI:kg] of mass point
nodeNumber	NodeIndex		invalid (-1)	node number (type NodeIndex) for mass point
visualization	VObjectMassPoint			parameters for visualization of item

The item VObjectMassPoint has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
graphicsData	BodyGraphicsData			Structure contains data for body visualization; data is defined in special list / dictionary structure

---

8.2.2.1 DESCRIPTION of ObjectMassPoint:

Information on input parameters:

input parameter	symbol	description see tables above
physicsMass	m	
nodeNumber	$n0$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position	${}^0\mathbf{p}_{\text{config}}({}^b\mathbf{b}) = {}^0\mathbf{r}_{\text{config}} + {}^0\mathbf{r}_{\text{ref}} + {}^{0b}\mathbf{I}_{3\times3}\ {}^b\mathbf{b}$	global position vector of translated local position; local (body) coordinate system = global coordinate system
Displacement	${}^0\mathbf{u}_{\text{config}} = [q_0,\ q_1,\ q_2]^{\text{T}}_{\text{config}}$	global displacement vector of mass point
Velocity	${}^0\mathbf{v}_{\text{config}} = {}^0\dot{\mathbf{u}}_{\text{config}} = [\dot{q}_0,\ \dot{q}_1,\ \dot{q}_2]^{\text{T}}_{\text{config}}$	global velocity vector of mass point
Acceleration	${}^0\mathbf{a}_{\text{config}} = {}^0\ddot{\mathbf{u}}_{\text{config}} = [\ddot{q}_0,\ \ddot{q}_1,\ \ddot{q}_2]^{\text{T}}_{\text{config}}$	global acceleration vector of mass point
RotationMatrix		identity matrix (only for completeness)
Rotation	[0, 0, 0]	(only for completeness)
AngularVelocity	[0, 0, 0]	(only for completeness)
AngularVelocityLocal	[0, 0, 0]	(only for completeness)

8.2.2.2 Definition of quantities

intermediate variables	symbol	description
node position	${}^0\mathbf{r}_{\text{config}} + {}^0\mathbf{r}_{\text{ref}} = {}^0\mathbf{p}\,(n_0)_{\text{config}}$	position of mass point which is provided by node n_0 in any configuration
node displacement	${}^0\mathbf{u}_{\text{config}} = {}^0\mathbf{r}_{\text{config}} = [q_0,\ q_1,\ q_2]^{\text{T}}_{\text{config}} = {}^0\mathbf{u}\,(n_0)_{\text{config}}$	displacement of mass point which is provided by node n_0 in any configuration
node velocity	${}^0\mathbf{v}_{\text{config}} = [\dot{q}_0,\ \dot{q}_1,\ \dot{q}_2]^{\text{T}}_{\text{config}} = {}^0\mathbf{v}\,(n_0)_{\text{config}}$	velocity of mass point which is provided by node n_0 in any configuration
transformation matrix	${}^{0b}\mathbf{A} = \mathbf{I}_{3\times3}$	transformation of local body (b) coordinates to global (0) coordinates; this is the constant unit matrix, because local = global coordinates for the mass point
residual forces	${}^0\mathbf{f} = [f_0,\ f_1,\ f_2]^{\text{T}}$	residual of all forces on mass point
applied forces	${}^0\mathbf{f}_a = [f_0,\ f_1,\ f_2]^{\text{T}}$	applied forces (loads, connectors, joint reaction forces, ...)

8.2.2.3 Equations of motion

$$\begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{bmatrix} \begin{bmatrix} \ddot{q}_0 \\ \ddot{q}_1 \\ \ddot{q}_2 \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ f_2 \end{bmatrix}. \tag{8.18}$$

For example, a LoadCoordinate on coordinate 1 of the node would add a term in f_1 on the RHS.

Position-based markers can measure position $\mathbf{p}_{\text{config}}$. The **position jacobian**

$$\mathbf{J}_{pos} = \partial\mathbf{p}_{\text{cur}}/\partial\mathbf{c}_{\text{cur}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{8.19}$$

transforms the action of global applied forces ${}^0\mathbf{f}_a$ of position-based markers on the coordinates $\mathbf{c}$

$$\mathbf{Q} = \mathbf{J}_{pos}\ {}^0\mathbf{f}_a. \tag{8.20}$$

8.2.2.4 MINI EXAMPLE for ObjectMassPoint

```
node = mbs.AddNode(NodePoint(referenceCoordinates = [1,1,0],
                             initialCoordinates=[0.5,0,0],
                             initialVelocities=[0.5,0,0]))
mbs.AddObject(MassPoint(nodeNumber = node, physicsMass=1))

#assemble and solve system for default parameters
mbs.Assemble()
mbs.SolveDynamic()

#check result
exudynTestGlobals.testResult = mbs.GetNodeOutput(node, exu.OutputVariableType.Position)
[0]
#final x-coordinate of position shall be 2
```

For examples on ObjectMassPoint see Relevant Examples and TestModels with weblink:

- interactiveTutorial.py (Examples/)
- ComputeSensitivitiesExample.py (Examples/)
- coordinateSpringDamper.py (Examples/)
- massSpringFrictionInteractive.py (Examples/)
- minimizeExample.py (Examples/)
- nMassOscillator.py (Examples/)
- nMassOscillatorInteractive.py (Examples/)
- parameterVariationExample.py (Examples/)
- particleClusters.py (Examples/)
- particlesSilo.py (Examples/)
- particlesTest.py (Examples/)
- particlesTest3D.py (Examples/)
- ...
- complexEigenvaluesTest.py (TestModels/)
- connectorGravityTest.py (TestModels/)
- contactCoordinateTest.py (TestModels/)
- ...

8.2.3 ObjectMassPoint2D

A 2D mass point which is attached to a position-based 2D node.

Additional information for ObjectMassPoint2D:

- This `Object` has/provides the following types = `Body`, `SingleNoded`
- Requested `Node` type = `Position2D` + `Position`
- **Short name** for Python = `MassPoint2D`
- **Short name** for Python visualization object = `VMassPoint2D`

The item **ObjectMassPoint2D** with type = 'MassPoint2D' has the following parameters:

Name	type	size	default value	description
name	String		''	objects's unique name
physicsMass	UReal		0.	mass [SI:kg] of mass point
nodeNumber	NodeIndex		invalid (-1)	node number (type NodeIndex) for mass point
visualization	VObjectMassPoint2D			parameters for visualization of item

The item VObjectMassPoint2D has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
graphicsData	BodyGraphicsData			Structure contains data for body visualization; data is defined in special list / dictionary structure

---

8.2.3.1 DESCRIPTION of ObjectMassPoint2D:

Information on input parameters:

input parameter	symbol	description see tables above
physicsMass	m	
nodeNumber	$n0$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position	${}^0\mathbf{p}_{\text{config}}({}^b\mathbf{b}) = {}^0\mathbf{r}_{\text{config}} + {}^0\mathbf{r}_{\text{ref}} + {}^{0b}\mathbf{I}_{2\times 2}\ {}^b\mathbf{b}$	global position vector of translated local position; local (body) coordinate system = global coordinate system
Displacement	${}^0\mathbf{u}_{\text{config}} = [q_0,\ q_1,\ 0]^{\text{T}}_{\text{config}}$	global displacement vector of mass point
Velocity	${}^0\mathbf{v}_{\text{config}} = {}^0\dot{\mathbf{u}}_{\text{config}} = [\dot{q}_0,\ \dot{q}_1,\ 0]^{\text{T}}_{\text{config}}$	global velocity vector of mass point
Acceleration	${}^0\mathbf{a}_{\text{config}} = {}^0\ddot{\mathbf{u}}_{\text{config}} = [\ddot{q}_0,\ \ddot{q}_1,\ 0]^{\text{T}}_{\text{config}}$	global acceleration vector of mass point
RotationMatrix		identity matrix (only for completeness)
Rotation	[0, 0, 0]	(only for completeness)
AngularVelocity	[0, 0, 0]	(only for completeness)
AngularVelocityLocal	[0, 0, 0]	(only for completeness)

8.2.3.2 Definition of quantities

intermediate variables	symbol	description
node position	${}^0\mathbf{r}_{\text{config}} + {}^0\mathbf{r}_{\text{ref}} = {}^0\mathbf{p}\,(n_0)_{\text{config}}$	position of mass point which is provided by node n_0 in any configuration (except reference)
node displacement	${}^0\mathbf{u}_{\text{config}} = {}^0\mathbf{r}_{\text{config}} = [q_0,\ q_1,\ 0]^{\text{T}}_{\text{config}} = {}^0\mathbf{u}\,(n_0)_{\text{config}}$	displacement of mass point which is provided by node n_0 in any configuration
node velocity	${}^0\mathbf{v}_{\text{config}} = [\dot{q}_0,\ \dot{q}_1,\ 0]^{\text{T}}_{\text{config}} = {}^0\mathbf{v}\,(n_0)_{\text{config}}$	velocity of mass point which is provided by node n_0 in any configuration
transformation matrix	${}^{0b}\mathbf{A} = \mathbf{I}_{3\times 3}$	transformation of local body (b) coordinates to global (0) coordinates; this is the constant unit matrix, because local = global coordinates for the mass point
residual forces	${}^0\mathbf{f} = [f_0,\ f_1]^{\text{T}}$	residual of all forces on mass point
applied forces	${}^0\mathbf{f}_a = [f_0,\ f_1,\ f_2]^{\text{T}}$	applied forces (loads, connectors, joint reaction forces, ...)

8.2.3.3 Equations of motion

$$\begin{bmatrix} m & 0 \\ 0 & m \end{bmatrix} \begin{bmatrix} \ddot{q}_0 \\ \ddot{q}_1 \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \end{bmatrix}. \tag{8.21}$$

For example, a LoadCoordinate on coordinate 1 of the node would add a term in f_1 on the RHS.

Position-based markers can measure position $\mathbf{p}_{\text{config}}$. The **position jacobian**

$$\mathbf{J}_{pos} = \partial \mathbf{p}_{\text{cur}} / \partial \mathbf{c}_{\text{cur}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{8.22}$$

transforms the action of global applied forces ${}^0\mathbf{f}_a$ of position-based markers on the coordinates $\mathbf{c}$

$$\mathbf{Q} = \mathbf{J}_{pos}\ {}^0\mathbf{f}_a. \tag{8.23}$$

8.2.3.4 MINI EXAMPLE for ObjectMassPoint2D

```
node = mbs.AddNode(NodePoint2D(referenceCoordinates = [1,1],
                               initialCoordinates=[0.5,0],
                               initialVelocities=[0.5,0]))
mbs.AddObject(MassPoint2D(nodeNumber = node, physicsMass=1))

#assemble and solve system for default parameters
mbs.Assemble()
mbs.SolveDynamic()

#check result
exudynTestGlobals.testResult = mbs.GetNodeOutput(node, exu.OutputVariableType.Position)
[0]
#final x-coordinate of position shall be 2
```

For examples on ObjectMassPoint2D see Relevant Examples and TestModels with weblink:

- myFirstExample.py (Examples/)
- reevingSystemOpen.py (Examples/)
- xExudynConfigSpecial.py (Examples/)
- ANCFslidingJoint2D.py (Examples/)
- ANCFslidingJoint2Drigid.py (Examples/)
- geneticOptimizationSliderCrank.py (Examples/)
- pendulum2Dconstraint.py (Examples/)
- pendulumIftommBenchmark.py (Examples/)
- SliderCrank.py (Examples/)
- sliderCrank3DwithANCFbeltDrive2.py (Examples/)
- slidercrankWithMassSpring.py (Examples/)
- SpringDamperMassUserFunction.py (Examples/)
- ...
- modelUnitTests.py (TestModels/)
- coordinateVectorConstraint.py (TestModels/)
- sliderCrankFloatingTest.py (TestModels/)
- ...

8.2.4 ObjectMass1D

A 1D (translational) mass which is attached to Node1D. Note, that the mass does not need to have the interpretation as a translational mass.

Additional information for ObjectMass1D:

- This `Object` has/provides the following types = `Body`, `SingleNoded`
- Requested `Node` type = `GenericODE2`
- **Short name** for Python = `Mass1D`
- **Short name** for Python visualization object = `VMass1D`

The item **ObjectMass1D** with type = 'Mass1D' has the following parameters:

Name	type	size	default value	description
name	String		''	objects's unique name
physicsMass	UReal		0.	mass [SI:kg] of mass
nodeNumber	NodeIndex		invalid (-1)	node number (type NodeIndex) for Node1D
referencePosition	Vector3D	3	[0.,0.,0.]	a reference position, used to transform the 1D coordinate to a position
referenceRotation	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	the constant body rotation matrix, which transforms body-fixed (b) to global (0) coordinates
visualization	VObjectMass1D			parameters for visualization of item

The item VObjectMass1D has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
graphicsData	BodyGraphicsData			Structure contains data for body visualization; data is defined in special list / dictionary structure

8.2.4.1 DESCRIPTION of ObjectMass1D:

Information on input parameters:

input parameter	symbol	description see tables above
physicsMass	m	
nodeNumber	$n0$	
referencePosition	${}^{0}\mathbf{r}_0$	
referenceRotation	${}^{0b}\mathbf{A}_0 \in \mathbb{R}^{3\times3}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position	${}^{0}\mathbf{p}_{\text{config}}$	global position vector; for interpretation see intermediate variables
Displacement	${}^{0}\mathbf{u}_{\text{config}}$	global displacement vector; for interpretation see intermediate variables
Velocity	${}^{0}\mathbf{v}_{\text{config}}$	global velocity vector; for interpretation see intermediate variables
RotationMatrix	${}^{0b}\mathbf{A}$	vector with 9 components of the rotation matrix (row-major format)
Rotation		vector with 3 components of the Euler/Tait-Bryan angles in xyz-sequence (${}^{0b}\mathbf{A}_{\text{config}} =: \mathbf{A}_0(\varphi_0) \cdot \mathbf{A}_1(\varphi_1) \cdot \mathbf{A}_2(\varphi_2)$), recomputed from rotation matrix ${}^{0b}\mathbf{A}$
AngularVelocity	${}^{0}\boldsymbol{\omega}_{\text{config}}$	angular velocity of body
AngularVelocityLocal	${}^{b}\boldsymbol{\omega}_{\text{config}}$	local (body-fixed) 3D velocity vector of node

8.2.4.2 Definition of quantities

intermediate variables	symbol	description
position coordinate	$p_{0\text{config}} = c_{0\text{config}} + c_{0\text{ref}}$	position coordinate of node (nodal coordinate c_0) in any configuration
displacement coordinate	$u_{0\text{config}} = c_{0\text{config}}$	displacement coordinate of mass node in any configuration
velocity coordinate	$u_{0\text{config}}$	velocity coordinate of mass node in any configuration
Position	${}^{0}\mathbf{p}_{\text{config}} = {}^{0}\mathbf{r}_0 + {}^{0b}\mathbf{A}_0 \, {}^{b}\begin{bmatrix} p_0 \\ 0 \\ 0 \end{bmatrix}_{\text{config}}$	(translational) position of mass object in any configuration
Displacement	${}^{0}\mathbf{u}_{\text{config}} = {}^{0b}\mathbf{A}_0 \, {}^{b}\begin{bmatrix} q_0 \\ 0 \\ 0 \end{bmatrix}_{\text{config}}$	(translational) displacement of mass object in any configuration

Velocity	${}^0\mathbf{v}_{\text{config}} = {}^{0b}\mathbf{A}_0 \, {}^b\begin{bmatrix} \dot{q}_0 \\ 0 \\ 0 \end{bmatrix}_{\text{config}}$	(translational) velocity of mass object in any configuration
residual force	f	residual of all forces on mass object
applied force	${}^0\mathbf{f}_a = [f_0,\ f_1,\ f_2]^{\mathrm{T}}$	3D applied force (loads, connectors, joint reaction forces, ...)
applied torque	${}^0\boldsymbol{\tau}_a = [\tau_0,\ \tau_1,\ \tau_2]^{\mathrm{T}}$	3D applied torque (loads, connectors, joint reaction forces, ...)

A rigid body marker (e.g., MarkerBodyRigid) may be attached to this object and forces/torques can be applied. However, torques will have no effect and forces will only have effect in 'direction' of the coordinate.

8.2.4.3 Equations of motion

$$m \cdot \ddot{q}_0 = f. \tag{8.24}$$

Note that f is computed from all connectors and loads upon the object. E.g., a 3D force vector ${}^0\mathbf{f}_a$ is transformed to f as

$$f = {}^b[1,\ 0,\ 0]\ {}^{b0}\mathbf{A}_0\ {}^0\mathbf{f}_a \tag{8.25}$$

Thus, the **position jacobian** reads

$$\mathbf{J}_{pos} = \partial \mathbf{p}_{\text{cur}} / \partial q_{0_{\text{cur}}} = {}^b[1,\ 0,\ 0]\ {}^{b0}\mathbf{A}_0 \tag{8.26}$$

8.2.4.4 MINI EXAMPLE for ObjectMass1D

```
node = mbs.AddNode(Node1D(referenceCoordinates = [1],
                          initialCoordinates=[0.5],
                          initialVelocities=[0.5]))
mass = mbs.AddObject(Mass1D(nodeNumber = node, physicsMass=1))

#assemble and solve system for default parameters
mbs.Assemble()
mbs.SolveDynamic()

#check result, get current mass position at local position [0,0,0]
exudynTestGlobals.testResult = mbs.GetObjectOutputBody(mass, exu.OutputVariableType.
Position, [0,0,0])[0]
#final x-coordinate of position shall be 2
```

For examples on ObjectMass1D see Relevant Examples and TestModels with weblink:

- craneReevingSystem.py (Examples/)
- lugreFrictionTest.py (Examples/)
- mpi4pyExample.py (Examples/)
- multiprocessingTest.py (Examples/)
- nMassOscillator.py (Examples/)
- nMassOscillatorEigenmodes.py (Examples/)
- nMassOscillatorInteractive.py (Examples/)
- driveTrainTest.py (TestModels/)
- mainSystemUserFunctionsTest.py (TestModels/)

8.2.5 ObjectRotationalMass1D

A 1D rotational inertia (mass) which is attached to Node1D.

Additional information for ObjectRotationalMass1D:

- This `Object` has/provides the following types = `Body`, `SingleNoded`
- Requested `Node` type = `GenericODE2`
- **Short name** for Python = `Rotor1D`
- **Short name** for Python visualization object = `VRotor1D`

The item **ObjectRotationalMass1D** with type = 'RotationalMass1D' has the following parameters:

Name	type	size	default value	description
name	String		''	objects's unique name
physicsInertia	UReal		0.	inertia components [SI:kgm^2] of rotor / rotational mass
nodeNumber	NodeIndex		invalid (-1)	node number (type NodeIndex) of Node1D, providing rotation coordinate $\psi_0 = c_0$
referencePosition	Vector3D	3	[0.,0.,0.]	a constant reference position = reference point, used to assign joint constraints accordingly and for drawing
referenceRotation	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	an intermediate rotation matrix, which transforms the 1D coordinate into 3D, see description
visualization	VObjectRotationalMass1D			parameters for visualization of item

The item VObjectRotationalMass1D has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
graphicsData	BodyGraphicsData			Structure contains data for body visualization; data is defined in special list / dictionary structure

8.2.5.1 DESCRIPTION of ObjectRotationalMass1D:

Information on input parameters:

input parameter	symbol	description see tables above
physicsInertia	J	
nodeNumber	$n0$	
referencePosition	${}^{0}\mathbf{r}_0$	
referenceRotation	${}^{0i}\mathbf{A}_0 \in \mathbb{R}^{3\times3}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position	${}^{0}\mathbf{p}_{\text{config}} = {}^{0}\mathbf{r}$	global position vector; for interpretation see intermediate variables
Displacement	${}^{0}\mathbf{u}_{\text{config}}$	global displacement vector; for interpretation see intermediate variables
Velocity	${}^{0}\mathbf{v}_{\text{config}}$	global velocity vector; for interpretation see intermediate variables
RotationMatrix	${}^{0b}\mathbf{A}$	vector with 9 components of the rotation matrix (row-major format)
Rotation	θ	scalar rotation angle obtained from underlying node
AngularVelocity	${}^{0}\boldsymbol{\omega}_{\text{config}}$	angular velocity of body
AngularVelocityLocal	${}^{b}\boldsymbol{\omega}_{\text{config}}$	local (body-fixed) 3D velocity vector of node

8.2.5.2 Definition of quantities

intermediate variables	symbol	description
position coordinate	$\theta_{0_{\text{config}}} = c_{0_{\text{config}}} + c_{0_{\text{ref}}}$	total rotation coordinate of node (e.g., Node1D) in any configuration (nodal coordinate c_0)
displacement coordinate	$\psi_{0_{\text{config}}} = c_{0_{\text{config}}}$	change of rotation coordinate of mass node (e.g., Node1D) in any configuration (nodal coordinate c_0)
velocity coordinate	$\dot{\psi}_{0_{\text{config}}}$	rotation velocity coordinate of mass node (e.g., Node1D) in any configuration
Position	${}^{0}\mathbf{p}_{\text{config}} = {}^{0}\mathbf{r}_0$	constant (translational) position of mass object in any configuration
Displacement	${}^{0}\mathbf{u}_{\text{config}} = [0,0,0]^{\text{T}}$	(translational) displacement of mass object in any configuration
Velocity	${}^{0}\mathbf{v}_{\text{config}} = [0,0,0]^{\text{T}}$	(translational) velocity of mass object in any configuration

AngularVelocity	${}^{0}\boldsymbol{\omega}_{\text{config}} = {}^{0i}\mathbf{A}_0 \, {}^{i}\begin{bmatrix} 0 \\ 0 \\ \dot{\psi}_0 \end{bmatrix}^{\text{T}}$	
AngularVelocityLocal	${}^{b}\boldsymbol{\omega}_{\text{config}} = {}^{i}\begin{bmatrix} 0 \\ 0 \\ \dot{\psi}_0 \end{bmatrix}^{\text{T}}$	
RotationMatrix	${}^{0b}\mathbf{A} = {}^{0i}\mathbf{A}_0 \, {}^{ib}\begin{bmatrix} \cos(\theta_0) & -\sin(\theta_0) & 0 \\ \sin(\theta_0) & \cos(\theta_0) & 0 \\ 0 & 0 & 1 \end{bmatrix}$	transformation of local body (b) coordinates to global (0) coordinates
residual force	τ	residual of all forces on mass object
applied force	${}^{0}\mathbf{f}_a = [f_0, \, f_1, \, f_2]^{\text{T}}$	3D applied force (loads, connectors, joint reaction forces, ...)
applied torque	${}^{0}\boldsymbol{\tau}_a = [\tau_0, \, \tau_1, \, \tau_2]^{\text{T}}$	3D applied torque (loads, connectors, joint reaction forces, ...)

A rigid body marker (e.g., MarkerBodyRigid) may be attached to this object and forces/torques can be applied. However, forces will have no effect and torques will only have effect in 'direction' of the coordinate.

8.2.5.3 Equations of motion

$$J \cdot \ddot{\psi}_0 = \tau. \tag{8.27}$$

Note that τ is computed from all connectors and loads upon the object. E.g., a 3D torque vector ${}^{0}\boldsymbol{\tau}_a$ is transformed to τ as

$$\tau = {}^{b}[0, \, 0, \, 1] \; {}^{b0}\mathbf{A}_0 \; {}^{0}\boldsymbol{\tau}_a \tag{8.28}$$

Thus, the **rotation jacobian** reads

$$\mathbf{J}_{rot} = \partial \boldsymbol{\omega}_{\text{cur}} / \partial \dot{q}_{0,cur} = {}^{b}[0, \, 0, \, 1] \; {}^{b0}\mathbf{A}_0 \tag{8.29}$$

8.2.5.4 MINI EXAMPLE for ObjectRotationalMass1D

```
node = mbs.AddNode(Node1D(referenceCoordinates = [1], #\psi_0ref
                          initialCoordinates=[0.5],   #\psi_0ini
                          initialVelocities=[0.5]))   #\psi_t0ini
rotor = mbs.AddObject(Rotor1D(nodeNumber = node, physicsInertia=1))

#assemble and solve system for default parameters
mbs.Assemble()
mbs.SolveDynamic()

#check result, get current rotor z-rotation at local position [0,0,0]
```

```
exudynTestGlobals.testResult = mbs.GetObjectOutputBody(rotor, exu.OutputVariableType.
Rotation, [0,0,0])
#final z-angle of rotor shall be 2
```

For examples on ObjectRotationalMass1D see Relevant Examples and TestModels with weblink:

- distanceSensor.py (TestModels/)
- coordinateSpringDamperExt.py (TestModels/)
- driveTrainTest.py (TestModels/)

8.2.6 ObjectRigidBody

A 3D rigid body which is attached to a 3D rigid body node. The rotation parametrization of the rigid body follows the rotation parametrization of the node. Use Euler parameters in the general case (no singularities) in combination with implicit solvers (GeneralizedAlpha or TrapezoidalIndex2), Tait-Bryan angles for special cases, e.g., rotors where no singularities occur if you rotate about x or z axis, or use Lie-group formulation with rotation vector together with explicit solvers. REMARK: Use the class `RigidBodyInertia`, see Section 7.19.2 and `CreateRigidBody(...)`, see Section 6.5.1, of `exudyn.rigidBodyUtilities` to handle inertia, COM and mass.

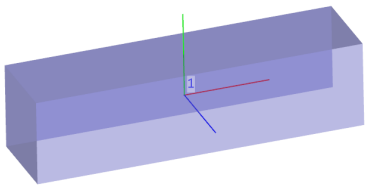


Figure 8.1: Example of ObjectRigidBody

Additional information for ObjectRigidBody:

- This `Object` has/provides the following types = `Body`, `SingleNoded`
- Requested `Node` type = `Position` + `Orientation` + `RigidBody`
- **Short name** for Python = `RigidBody`
- **Short name** for Python visualization object = `VRigidBody`

The item **ObjectRigidBody** with type = 'RigidBody' has the following parameters:

Name	type	size	default value	description
name	String		''	objects's unique name
physicsMass	UReal		0.	mass [SI:kg] of rigid body
physicsInertia	Vector6D		[0.,0.,0., 0.,0.,0.]	inertia components [SI:kgm^2]: $[J_{xx}, J_{yy}, J_{zz}, J_{yz}, J_{xz}, J_{xy}]$ in body-fixed coordinate system and w.r.t. to the reference point of the body, NOT necessarily w.r.t. to COM; use the class RigidBodyInertia of exudynRigidBodyUtilities.py and CreateRigidBody(...) of MainSystem to handle inertia, COM and mass

physicsCenterOfMass	Vector3D	3	[0.,0.,0.]	local position of COM relative to the body's reference point; if the vector of the COM is [0,0,0], the computation will not consider additional terms for the COM and it is faster
nodeNumber	NodeIndex		invalid (-1)	node number (type NodeIndex) for rigid body node
visualization	VObjectRigidBody			parameters for visualization of item

The item VObjectRigidBody has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
graphicsDataUserFunction	PyFunctionGraphicsData		0	A Python function which returns a body-GraphicsData object, which is a list of graphics data in a dictionary computed by the user function; the graphics elements need to be defined in the local body coordinates and are transformed by mbs to global coordinates
graphicsData	BodyGraphicsData			Structure contains data for body visualization; data is defined in special list / dictionary structure

---

8.2.6.1 DESCRIPTION of ObjectRigidBody:

Information on input parameters:

input parameter	symbol	description see tables above
physicsMass	m	
physicsInertia	${}^{b}\mathbf{j}_6$	
physicsCenterOfMass	${}^{b}\mathbf{b}_{COM}$	
nodeNumber	$n0$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position	${}^{0}\mathbf{p}_{\text{config}}({}^{b}\mathbf{b}) = {}^{0}\mathbf{r}_{\text{config}} + {}^{0}\mathbf{r}_{\text{ref}} + {}^{0b}\mathbf{A}\ {}^{b}\mathbf{b}$	global position vector of body-fixed point given by local position vector ${}^{b}\mathbf{b}$

Displacement	${}^0\mathbf{u}_{\text{config}} + {}^{0b}\mathbf{A}\ {}^b\mathbf{b}$	global displacement vector of body-fixed point given by local position vector ${}^b\mathbf{b}$
Velocity	${}^0\mathbf{v}_{\text{config}}({}^b\mathbf{b}) = {}^0\dot{\mathbf{u}}_{\text{config}} + {}^{0b}\mathbf{A}\,({}^b\boldsymbol{\omega} \times {}^b\mathbf{b}_{\text{config}})$	global velocity vector of body-fixed point given by local position vector ${}^b\mathbf{b}$
VelocityLocal	${}^b\mathbf{v}_{\text{config}}({}^b\mathbf{b}) = {}^{b0}\mathbf{A}\ {}^0\mathbf{v}_{\text{config}}({}^b\mathbf{b})$	local (body-fixed) velocity vector of body-fixed point given by local position vector ${}^b\mathbf{b}$
RotationMatrix	$\text{vec}({}^{0b}\mathbf{A}) = [A_{00}, A_{01}, A_{02}, A_{10}, \ldots, A_{21}, A_{22}]^{\text{T}}_{\text{config}}$	vector with 9 components of the rotation matrix (row-major format)
Rotation		vector with 3 components of the Euler angles in xyz-sequence (R=Rx*Ry*Rz), recomputed from rotation matrix
AngularVelocity	${}^0\boldsymbol{\omega}_{\text{config}}$	angular velocity of body
AngularVelocityLocal	${}^b\boldsymbol{\omega}_{\text{config}}$	local (body-fixed) 3D velocity vector of node
Acceleration	${}^0\mathbf{a}_{\text{config}}({}^b\mathbf{b}) = {}^0\ddot{\mathbf{u}} + {}^0\boldsymbol{\alpha} \times ({}^{0b}\mathbf{A}\ {}^b\mathbf{b}) + {}^0\boldsymbol{\omega} \times ({}^0\boldsymbol{\omega} \times ({}^{0b}\mathbf{A}\ {}^b\mathbf{b}))$	global acceleration vector of body-fixed point given by local position vector ${}^b\mathbf{b}$
AccelerationLocal	${}^b\mathbf{a}_{\text{config}}({}^b\mathbf{b}) = {}^{b0}\mathbf{A}\ {}^0\mathbf{a}_{\text{config}}({}^b\mathbf{b})$	local (body-fixed) acceleration vector of body-fixed point given by local position vector ${}^b\mathbf{b}$
AngularAcceleration	${}^0\boldsymbol{\alpha}_{\text{config}}$	angular acceleration vector of body
AngularAccelerationLocal	${}^b\boldsymbol{\alpha}_{\text{config}} = {}^{b0}\mathbf{A}\ {}^0\boldsymbol{\alpha}_{\text{config}}$	local angular acceleration vector of body

8.2.6.2 Definition of quantities

intermediate variables	**symbol**	**description**
inertia tensor	${}^b\mathbf{J} = {}^b\begin{bmatrix} J_{xx} & J_{xy} & J_{xz} \\ J_{xy} & J_{yy} & J_{yz} \\ J_{xz} & J_{yz} & J_{zz} \end{bmatrix}$	symmetric inertia tensor, based on components of ${}^b\mathbf{j}_6$, in body-fixed (local) coordinates and w.r.t. body's reference point
reference coordinates	$\mathbf{q}_{\text{ref}} = [\mathbf{r}^{\text{T}}_{\text{ref}}, \boldsymbol{\psi}^{\text{T}}_{\text{ref}}]^{\text{T}}$	defines reference configuration, **DIFFERENT** meaning from body's reference point!
(relative) current coordinates	$\mathbf{q}_{\text{cur}} = [\mathbf{r}^{\text{T}}_{\text{cur}}, \boldsymbol{\psi}^{\text{T}}_{\text{cur}}]^{\text{T}}$	unknowns in solver; **relative** to the reference coordinates; current coordinates at initial configuration = initial coordinates $\mathbf{q}_{\text{ini}}$
current velocity coordinates	$\dot{\mathbf{q}}_{\text{cur}} = [\mathbf{v}^{\text{T}}_{\text{cur}}, \dot{\boldsymbol{\psi}}^{\text{T}}_{\text{cur}}]^{\text{T}} = [\dot{\mathbf{p}}^{\text{T}}_{\text{cur}}, \dot{\boldsymbol{\theta}}^{\text{T}}_{\text{cur}}]^{\text{T}}$	current velocity coordinates
body's reference point	${}^0\mathbf{r}_{\text{config}} + {}^0\mathbf{r}_{\text{ref}} = {}^0\mathbf{p}\,(n_0)_{\text{config}}$	position of **body's reference point** provided by node n_0 in any configuration except for reference; if ${}^b\mathbf{b}_{COM} == [0,\ 0,\ 0]^{\text{T}}$, this position becomes equal to the COM position
reference body's reference point	${}^0\mathbf{r}_{\text{ref}} = {}^0\mathbf{p}\,(n_0)_{\text{ref}}$	position of **body's reference point** in reference configuration
body's reference point displacement	${}^0\mathbf{u}_{\text{config}} = {}^0\mathbf{r}_{\text{config}} = [q_0,\ q_1,\ q_2]^{\text{T}}_{\text{config}} = {}^0\mathbf{u}\,(n_0)_{\text{config}}$	displacement of **body's reference point** which is provided by node n_0 in any configuration

body's reference point velocity	${}^0\mathbf{v}_{\text{config}} = {}^0\dot{\mathbf{r}}_{\text{config}} = [\dot{q}_0,\ \dot{q}_1,\ \dot{q}_2]^{\text{T}}_{\text{config}} = {}^0\mathbf{v}\,(n_0)_{\text{config}}$	velocity of **body's reference point** which is provided by node n_0 in any configuration
body's reference point acceleration	${}^0\mathbf{a}_{\text{config}} = [\ddot{q}_0,\ \ddot{q}_1,\ \ddot{q}_2]^{\text{T}}_{\text{config}}$	acceleration of **body's reference point** which is provided by node n_0 in any configuration
rotation coordinates	$\boldsymbol{\theta}_{\text{config}} = \boldsymbol{\psi}(n_0)_{\text{ref}} + \boldsymbol{\psi}(n_0)_{\text{config}}$	(total) rotation parameters of body as provided by node n_0 in any configuration
rotation parameters	$\boldsymbol{\theta}_{\text{config}} = \boldsymbol{\psi}(n_0)_{\text{ref}} + \boldsymbol{\psi}(n_0)_{\text{config}}$	(total) rotation parameters of body as provided by node n_0 in any configuration
body rotation matrix	${}^{0b}\mathbf{A}_{\text{config}} = {}^{0b}\mathbf{A}\,(n_0)_{\text{config}}$	rotation matrix which transforms local to global coordinates as given by node
local position	${}^b\mathbf{b} = [{}^bb_0\,, {}^bb_1\,, {}^bb_2\,]^{\text{T}}$	local position as used by markers or sensors
angular velocity	${}^0\boldsymbol{\omega}_{\text{config}} = {}^0[\omega_0(n_0),\ \omega_1(n_0),\ \omega_2(n_0)]^{\text{T}}_{\text{config}}$	global angular velocity of body as provided by node n_0 in any configuration
local angular velocity	${}^b\boldsymbol{\omega}_{\text{config}}$	local angular velocity of body as provided by node n_0 in any configuration
body angular acceleration	${}^0\boldsymbol{\alpha}_{\text{config}} = {}^0\dot{\boldsymbol{\omega}}_{\text{config}}$	angular acceleratoin of body as provided by node n_0 in any configuration
applied forces	${}^0\mathbf{f}_a = [f_0,\ f_1,\ f_2]^{\text{T}}$	calculated from loads, connectors, ...
applied torques	${}^0\boldsymbol{\tau}_a = [\tau_0,\ \tau_1,\ \tau_2]^{\text{T}}$	calculated from loads, connectors, ...
constraint reaction forces	${}^0\mathbf{f}_\lambda = [f_{\lambda 0},\ f_{\lambda 1},\ f_{\lambda 2}]^{\text{T}}$	calculated from joints or constraint)
constraint reaction torques	${}^0\boldsymbol{\tau}_\lambda = [\tau_{\lambda 0},\ \tau_{\lambda 1},\ \tau_{\lambda 2}]^{\text{T}}$	calculated from joints or constraints

8.2.6.3 Rotation parametrization

The equations of motion of the rigid body build upon a specific parameterization of the rigid body coordinates. Rigid body coordinates are defined by the underlying node given by `nodeNumber` $n0$. Appropriate nodes are

- `NodeRigidBodyEP` (Euler parameters)
- `NodeRigidBodyRxyz` (Euler angles / Tait Bryan angles)
- `NodeRigidBodyRotVecLG` (Rotation vector with Lie group integration option)

Note that all operations for rotation parameters, such as the computation of the rotation matrix, must be performed with the rotation parameters $\boldsymbol{\theta}$, see table above, which are the sum of reference and current coordinates.

The angular velocity in body-fixed coordinates is related to the rotation parameters by means of a matrix ${}^b\mathbf{G}_{rp}$,

$$ {}^b\boldsymbol{\omega} = {}^b\mathbf{G}_{rp}\,\dot{\boldsymbol{\theta}} = {}^b\mathbf{G}_{rp}\,\dot{\boldsymbol{\psi}}\,, \tag{8.30} $$

and is specific for any rotation parametrization rp. The angular velocity in global coordinates is related to the rotation parameters by means of a matrix ${}^0\mathbf{G}_{rp}$,

$$ {}^0\boldsymbol{\omega} = {}^0\mathbf{G}_{rp}\,\dot{\boldsymbol{\theta}}\,. \tag{8.31} $$

The local angular accelerations follow as

$$ {}^b\boldsymbol{\alpha} = {}^b\dot{\boldsymbol{\omega}} = {}^b\mathbf{G}_{rp}\,\ddot{\boldsymbol{\theta}} + {}^b\dot{\mathbf{G}}_{rp}\,\dot{\boldsymbol{\theta}}\,, \tag{8.32} $$

remember that derivatives for angular velocities can also be done in the local frame. In case of Euler parameters and the Lie-group rotation vector we find that ${}^b\dot{\mathbf{G}}_{rp}\,\dot{\boldsymbol{\theta}} = \mathbf{0}$.

8.2.6.4 Equations of motion for COM

The equations of motion for a rigid body, the so-called Newton-Euler equations, can be written for the special case of the reference point = COM and split for translations and rotations, using a coordinate-free notation,

$$\begin{bmatrix} m\mathbf{I}_{3\times3} & \mathbf{0} \\ \mathbf{0} & \mathbf{J} \end{bmatrix} \begin{bmatrix} \mathbf{a}_{COM} \\ \boldsymbol{\alpha} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ -\tilde{\boldsymbol{\omega}}\mathbf{J}\boldsymbol{\omega} \end{bmatrix} + \begin{bmatrix} \mathbf{f}_a \\ \boldsymbol{\tau}_a \end{bmatrix} + \begin{bmatrix} \mathbf{f}_\lambda \\ \boldsymbol{\tau}_\lambda \end{bmatrix} \tag{8.33}$$

with the 3×3 unit matrix $\mathbf{I}_{3\times3}$ and forces $\mathbf{f}$ resp. torques $\boldsymbol{\tau}$ as discribed in the table above. A change of the reference point, using the vector $\mathbf{b}_{COM}$ from the body's reference point $\mathbf{p}$ to the COM position, is simple by replacing COM accelerations using the common relation known from Euler

$$\mathbf{a}_{COM} = \mathbf{a} + \tilde{\boldsymbol{\alpha}}\mathbf{b}_{COM} + \tilde{\boldsymbol{\omega}}\tilde{\boldsymbol{\omega}}\mathbf{b}_{COM}\,, \tag{8.34}$$

which is inserted into the first line of Eq. (8.33). Additionally, the second line of Eq. (8.33) (second Euler equation related to rate of angular momentum) is rewritten for an arbitrary reference point, $\mathbf{b}_{COM}$ denoting the vector from the body reference point to COM, using the well known relation

$$m\tilde{\mathbf{b}}_{COM}\boldsymbol{\alpha} + \mathbf{J}\boldsymbol{\alpha} + \tilde{\boldsymbol{\omega}}\mathbf{J}\boldsymbol{\omega} = \boldsymbol{\tau}_a + \boldsymbol{\tau}_\lambda \tag{8.35}$$

8.2.6.5 Equations of motion for arbitrary reference point

This immediately leads to the equations of motion for the rigid body with respect to an arbitrary reference point (≠ COM), see e.g. [63](page 258ff.), which have the general coordinate-free form

$$\begin{bmatrix} m\mathbf{I}_{3\times3} & -m\tilde{\mathbf{b}}_{COM} \\ m\tilde{\mathbf{b}}_{COM} & \mathbf{J} \end{bmatrix} \begin{bmatrix} \mathbf{a} \\ \boldsymbol{\alpha} \end{bmatrix} = \begin{bmatrix} -m\tilde{\boldsymbol{\omega}}\tilde{\boldsymbol{\omega}}\mathbf{b}_{COM} \\ -\tilde{\boldsymbol{\omega}}\mathbf{J}\boldsymbol{\omega} \end{bmatrix} + \begin{bmatrix} \mathbf{f}_a \\ \boldsymbol{\tau}_a \end{bmatrix} + \begin{bmatrix} \mathbf{f}_\lambda \\ \boldsymbol{\tau}_\lambda \end{bmatrix}, \tag{8.36}$$

in which $\mathbf{J}$ is the inertia tensor w.r.t. the chosen reference point (which has local coordinates ${}^b[0,0,0]^{\mathrm{T}}$). Eq. (8.36) can be written in the global frame (0),

$$\begin{bmatrix} m\mathbf{I}_{3\times3} & -m\,{}^0\tilde{\mathbf{b}}_{COM} \\ m\,{}^0\tilde{\mathbf{b}}_{COM} & {}^0\mathbf{J} \end{bmatrix} \begin{bmatrix} {}^0\mathbf{a} \\ {}^0\boldsymbol{\alpha} \end{bmatrix} = \begin{bmatrix} -m\,{}^0\tilde{\boldsymbol{\omega}}\,{}^0\tilde{\boldsymbol{\omega}}\,{}^0\mathbf{b}_{COM} \\ -\,{}^0\tilde{\boldsymbol{\omega}}\,{}^0\mathbf{J}\,{}^0\boldsymbol{\omega} \end{bmatrix} + \begin{bmatrix} {}^0\mathbf{f}_a \\ {}^0\boldsymbol{\tau}_a \end{bmatrix} + \begin{bmatrix} {}^0\mathbf{f}_\lambda \\ {}^0\boldsymbol{\tau}_\lambda \end{bmatrix}. \tag{8.37}$$

Expressing the translational part (first line) of Eq. (8.37) in the global frame (0), using local coordinates (b) for quantities that are constant in the body-fixed frame, ${}^b\mathbf{J}$ and ${}^b\mathbf{b}_{COM}$, thus expressing also the angular velocity ${}^b\boldsymbol{\omega}$ in the body-fixed frame, applying Eq. (8.30) and Eq. (8.32), and using the relations

$$\begin{aligned} {}^0\tilde{\boldsymbol{\omega}}\,{}^0\tilde{\boldsymbol{\omega}}\,{}^0\mathbf{b}_{COM} &= {}^{0b}\mathbf{A}\,{}^b\tilde{\boldsymbol{\omega}}\,{}^b\tilde{\boldsymbol{\omega}}\,{}^b\mathbf{b}_{COM} = -\,{}^{0b}\mathbf{A}\,{}^b\tilde{\boldsymbol{\omega}}\,{}^b\tilde{\mathbf{b}}_{COM}\,{}^b\boldsymbol{\omega} = -\,{}^{0b}\mathbf{A}\,{}^b\tilde{\boldsymbol{\omega}}\,{}^b\tilde{\mathbf{b}}_{COM}\,{}^b\mathbf{G}_{rp}\,\dot{\boldsymbol{\theta}}\,, & (8.38)\\ -m\,{}^0\tilde{\mathbf{b}}_{COM}\,{}^0\tilde{\boldsymbol{\alpha}} &= -m\,{}^{0b}\mathbf{A}\,{}^b\tilde{\mathbf{b}}_{COM}\,{}^b\tilde{\boldsymbol{\alpha}} = -m\,{}^{0b}\mathbf{A}\,{}^b\tilde{\mathbf{b}}_{COM}\left({}^b\mathbf{G}_{rp}\,\ddot{\boldsymbol{\theta}} + {}^b\dot{\mathbf{G}}_{rp}\,\dot{\boldsymbol{\theta}}\right), & (8.39) \end{aligned}$$

we obtain

$$\begin{bmatrix} m\mathbf{I}_{3\times3} & -m\,{}^{0b}\mathbf{A}\,{}^{b}\tilde{\mathbf{b}}_{COM}\,{}^{b}\mathbf{G}_{rp} \\ m\,{}^{b}\mathbf{G}_{rp}^{\mathrm{T}}\,{}^{b}\tilde{\mathbf{b}}_{COM}\,{}^{0b}\mathbf{A}^{\mathrm{T}} & {}^{b}\mathbf{G}_{rp}^{\mathrm{T}}\,{}^{b}\mathbf{J}\,{}^{b}\mathbf{G}_{rp} \end{bmatrix} \begin{bmatrix} {}^{0}\mathbf{a} \\ \ddot{\boldsymbol{\theta}} \end{bmatrix}$$
$$= \begin{bmatrix} m\,{}^{0b}\mathbf{A}\,{}^{b}\tilde{\boldsymbol{\omega}}\,{}^{b}\tilde{\mathbf{b}}_{COM}\,{}^{b}\boldsymbol{\omega} + m\,{}^{0b}\mathbf{A}\,{}^{b}\tilde{\mathbf{b}}_{COM}\,{}^{b}\dot{\mathbf{G}}_{rp}\,\dot{\boldsymbol{\theta}} \\ -{}^{b}\mathbf{G}_{rp}^{\mathrm{T}}\,{}^{b}\tilde{\boldsymbol{\omega}}\,{}^{b}\mathbf{J}\,{}^{b}\boldsymbol{\omega} - {}^{b}\mathbf{G}_{rp}^{\mathrm{T}}\,{}^{b}\mathbf{J}\,{}^{b}\dot{\mathbf{G}}_{rp}\,\dot{\boldsymbol{\theta}} \end{bmatrix} + \begin{bmatrix} {}^{0}\mathbf{f}_{a} \\ {}^{0}\mathbf{G}_{rp}^{\mathrm{T}}\,{}^{0}\boldsymbol{\tau}_{a} \end{bmatrix} + \begin{bmatrix} {}^{0}\mathbf{f}_{\lambda} \\ \mathbf{f}_{\theta,\lambda} \end{bmatrix} \tag{8.40}$$

with constraint reaction forces $\mathbf{f}_{\theta,\lambda}$ for the rotation parameters. Note that the last line has been pre-multiplied with ${}^{b}\mathbf{G}_{rp}^{\mathrm{T}}$ (in order to make the mass matrix symmetric) and that ${}^{b}\dot{\mathbf{G}}_{rp}\,\dot{\boldsymbol{\theta}} = \mathbf{0}$ in case of Euler parameters and the Lie-group rotation vector .

8.2.6.6 Euler parameters

In case of Euler parameters, a constraint equation is automatically added, reading for the index 3 case

$$g_{\theta}(\boldsymbol{\theta}) = \theta_0^2 + \theta_1^2 + \theta_2^2 + \theta_3^2 - 1 = 0 \tag{8.41}$$

and for the index 2 case

$$\dot{g}_{\theta}(\boldsymbol{\theta}) = 2\theta_0\dot{\theta}_0 + 2\theta_1\dot{\theta}_1 + 2\theta_2\dot{\theta}_2 + 2\theta_3\dot{\theta}_3 = 0 \tag{8.42}$$

Given a Lagrange parameter (algebraic variable) λ_{θ} related to the Euler parameter constraint (8.41), the constraint reaction forces in Eq. (8.40) then read

$$\mathbf{f}_{\theta,\lambda} = \frac{\partial g_{\theta}}{\boldsymbol{\theta}^{\mathrm{T}}}\lambda_{\theta} = [2\theta_0,\ 2\theta_1,\ 2\theta_2,\ 2\theta_3]^{\mathrm{T}} \tag{8.43}$$

Userfunction: `graphicsDataUserFunction(mbs, itemNumber)`

A user function, which is called by the visualization thread in order to draw user-defined objects. The function can be used to generate any `BodyGraphicsData`, see Section 10.4. Use `exudyn.graphics` functions, see Section 7.8, to create more complicated objects. Note that `graphicsDataUserFunction` needs to copy lots of data and is therefore inefficient and only designed to enable simpler tests, but not large scale problems.

For an example for `graphicsDataUserFunction` see ObjectGround, Section 8.2.1.

arguments / return	type or size	description
`mbs`	MainSystem	provides reference to mbs, which can be used in user function to access all data of the object
`itemNumber`	Index	integer number of the object in mbs, allowing easy access
`return value`	BodyGraphicsData	list of `GraphicsData` dictionaries, see Section 10.4

For creating a `ObjectRigidBody`, there is a `rigidBodyUtilities` function `CreateRigidBody`, see Section 6.5.1, which simplifies the setup of a rigid body significantely!

For examples on ObjectRigidBody see Relevant Examples and TestModels with weblink:

- `rigid3Dexample.py` (Examples/)
- `rigidBodyIMUtest.py` (Examples/)
- `addPrismaticJoint.py` (Examples/)
- `addRevoluteJoint.py` (Examples/)
- `ANCFrotatingCable2D.py` (Examples/)
- `ballBearningModel.py` (Examples/)
- `bicycleIftommBenchmark.py` (Examples/)
- `bungeeJump.py` (Examples/)
- `camFollowerExample.py` (Examples/)
- `chainDriveExample.py` (Examples/)
- `chatGPTupdate.py` (Examples/)
- `chatGPTupdate2.py` (Examples/)
- ...
- `explicitLieGroupIntegratorPythonTest.py` (TestModels/)
- `explicitLieGroupIntegratorTest.py` (TestModels/)
- `explicitLieGroupMBSTest.py` (TestModels/)
- ...

8.2.7 ObjectRigidBody2D

A 2D rigid body which is attached to a rigid body 2D node. The body obtains coordinates, position, velocity, etc. from the underlying 2D node.

Additional information for ObjectRigidBody2D:

- This `Object` has/provides the following types = `Body`, `SingleNoded`
- Requested `Node` type = `Position2D` + `Orientation2D` + `Position` + `Orientation`
- **Short name** for Python = `RigidBody2D`
- **Short name** for Python visualization object = `VRigidBody2D`

The item **ObjectRigidBody2D** with type = 'RigidBody2D' has the following parameters:

Name	type	size	default value	description
name	String		''	objects's unique name
physicsMass	UReal		0.	mass [SI:kg] of rigid body
physicsInertia	UReal		0.	inertia [SI:kgm^2] of rigid body w.r.t. reference point; this is equal to the center of mass, if physicsCenterOfMass = 0
physicsCenterOfMass	Vector2D	2	[0.,0.]	local position of COM relative to the body's reference point; if the vector of the COM is [0,0], the computation will not consider additional terms for the COM and it is faster
nodeNumber	NodeIndex		invalid (-1)	node number (type NodeIndex) for 2D rigid body node
visualization	VObjectRigidBody2D			parameters for visualization of item

The item VObjectRigidBody2D has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
graphicsDataUserFunction	PyFunctionGraphicsData		0	A Python function which returns a bodyGraphicsData object, which is a list of graphics data in a dictionary computed by the user function; the graphics elements need to be defined in the local body coordinates and are transformed by mbs to global coordinates
graphicsData	BodyGraphicsData			Structure contains data for body visualization; data is defined in special list / dictionary structure

8.2.7.1 DESCRIPTION of ObjectRigidBody2D:

Information on input parameters:

input parameter	symbol	description see tables above
physicsMass	m	
physicsInertia	J	
physicsCenterOfMass	${}^{b}\mathbf{b}_{COM}$	
nodeNumber	n_0	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position	${}^{0}\mathbf{p}_{\text{config}}({}^{b}\mathbf{b}) = {}^{0}\mathbf{r}_{\text{config}} + {}^{0}\mathbf{r}_{\text{ref}} + {}^{0b}\mathbf{A}\ {}^{b}\mathbf{b}$	global position vector of body-fixed point given by local position vector ${}^{b}\mathbf{b}$
Displacement	${}^{0}\mathbf{u}_{\text{config}} + {}^{0b}\mathbf{A}\ {}^{b}\mathbf{b}$	global displacement vector of body-fixed point given by local position vector ${}^{b}\mathbf{b}$
Velocity	${}^{0}\mathbf{v}_{\text{config}}({}^{b}\mathbf{b}) = {}^{0}\dot{\mathbf{u}}_{\text{config}} + {}^{0b}\mathbf{A}({}^{b}\boldsymbol{\omega} \times {}^{b}\mathbf{b}_{\text{config}})$	global velocity vector of body-fixed point given by local position vector ${}^{b}\mathbf{b}$
VelocityLocal	${}^{b}\mathbf{v}_{\text{config}}({}^{b}\mathbf{b}) = {}^{b0}\mathbf{A}\ {}^{0}\mathbf{v}_{\text{config}}({}^{b}\mathbf{b})$	local (body-fixed) velocity vector of body-fixed point given by local position vector ${}^{b}\mathbf{b}$
RotationMatrix	$\text{vec}({}^{0b}\mathbf{A}) = [A_{00}, A_{01}, A_{02}, A_{10}, \ldots, A_{21}, A_{22}]^{\text{T}}_{\text{config}}$	vector with 9 components of the rotation matrix (row-major format)
Rotation	$\theta_{0\text{config}}$	scalar rotation angle of body
AngularVelocity	${}^{0}\boldsymbol{\omega}_{\text{config}}$	angular velocity of body
AngularVelocityLocal	${}^{b}\boldsymbol{\omega}_{\text{config}}$	local (body-fixed) 3D velocity vector of node
Acceleration	${}^{0}\mathbf{a}_{\text{config}}({}^{b}\mathbf{b}) = {}^{0}\ddot{\mathbf{u}} + {}^{0}\boldsymbol{\alpha} \times ({}^{0b}\mathbf{A}\ {}^{b}\mathbf{b}) + {}^{0}\boldsymbol{\omega} \times ({}^{0}\boldsymbol{\omega} \times ({}^{0b}\mathbf{A}\ {}^{b}\mathbf{b}))$	global acceleration vector of body-fixed point given by local position vector ${}^{b}\mathbf{b}$
AccelerationLocal	${}^{b}\mathbf{a}_{\text{config}}({}^{b}\mathbf{b}) = {}^{b0}\mathbf{A}\ {}^{0}\mathbf{a}_{\text{config}}({}^{b}\mathbf{b})$	local (body-fixed) acceleration vector of body-fixed point given by local position vector ${}^{b}\mathbf{b}$
AngularAcceleration	${}^{0}\boldsymbol{\alpha}_{\text{config}}$	angular acceleration vector of body
AngularAccelerationLocal	${}^{b}\boldsymbol{\alpha}_{\text{config}} = {}^{b0}\mathbf{A}\ {}^{0}\boldsymbol{\alpha}_{\text{config}}$	local angular acceleration vector of body

8.2.7.2 Definition of quantities

intermediate variables	symbol	description
reference position	${}^{0}\mathbf{r}_{\text{config}} + {}^{0}\mathbf{r}_{\text{ref}} = {}^{0}\mathbf{p}\,(n_0)_{\text{config}}$	reference point, only equal to the position of COM if ${}^{b}\mathbf{b}_{COM} = \mathbf{0}$; provided by node n_0 in any configuration (except reference)
reference point displacement	${}^{0}\mathbf{u}_{\text{config}} = {}^{0}\mathbf{r}_{\text{config}} = [q_0,\ q_1,\ 0]^{\text{T}}_{\text{config}} = {}^{0}\mathbf{u}\,(n_0)_{\text{config}}$	displacement of reference point which is provided by node n_0 in any configuration; NOTE that for configurations other than reference, it is follows that ${}^{0}\mathbf{r}_{\text{ref}} - {}^{0}\mathbf{r}_{\text{config}}$
reference point velocity	${}^{0}\mathbf{v}_{\text{config}} = [\dot{q}_0,\ \dot{q}_1,\ 0]^{\text{T}}_{\text{config}} = {}^{0}\mathbf{v}\,(n_0)_{\text{config}}$	velocity of reference point which is provided by node n_0 in any configuration

body rotation	${}^0\theta_{0\text{config}} = \theta_0(n_0)_{\text{config}} = \psi_0(n_0)_{\text{ref}} + \psi_0(n_0)_{\text{config}}$	rotation of body as provided by node n_0 in any configuration
body rotation matrix	${}^{0b}\mathbf{A}_{\text{config}} = {}^{0b}\mathbf{A}(n_0)_{\text{config}}$	rotation matrix which transforms local to global coordinates as given by node
local position	${}^b\mathbf{b} = [{}^bb_0, {}^bb_1, 0]^\mathrm{T}$	local position as used by markers or sensors
body angular velocity	${}^0\boldsymbol{\omega}_{\text{config}} = {}^0[\omega_0(n_0), 0, 0]^\mathrm{T}_{\text{config}}$	rotation of body as provided by node n_0 in any configuration
(generalized) coordinates	$\mathbf{c}_{\text{config}} = [q_0, q_1, \psi_0]^\mathrm{T}$	generalized coordinates of body (= coordinates of node)
generalized forces	${}^0\mathbf{f} = [f_0, f_1, \tau_2]^\mathrm{T}$	generalized forces applied to body
applied forces	${}^0\mathbf{f}_a = [f_0, f_1, 0]^\mathrm{T}$	applied forces (loads, connectors, joint reaction forces, ...)
applied torques	${}^0\boldsymbol{\tau}_a = [0, 0, \tau_2]^\mathrm{T}$	applied torques (loads, connectors, joint reaction forces, ...)

8.2.7.3 Equations of motion

The equations of motion in case that `physicsCenterOfMass=0` read:

$$\begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & J \end{bmatrix} \begin{bmatrix} \ddot{q}_0 \\ \ddot{q}_1 \\ \ddot{\psi}_0 \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ \tau_2 \end{bmatrix} = \mathbf{f}. \tag{8.44}$$

if `physicsCenterOfMass` is nonzero, we resort to (not that J represents the moment of inertia related to the reference point!):

$$\begin{bmatrix} m & 0 & G_x \\ 0 & m & G_y \\ G_x & G_y & J \end{bmatrix} \begin{bmatrix} \ddot{q}_0 \\ \ddot{q}_1 \\ \ddot{\psi}_0 \end{bmatrix} = \begin{bmatrix} m\dot{\psi}_0^2 b_x \\ m\dot{\psi}_0^2 b_y \\ 0 \end{bmatrix} + \begin{bmatrix} f_0 \\ f_1 \\ \tau_2 \end{bmatrix} = \mathbf{f}. \tag{8.45}$$

where we use the relations caused by the non-zero center of mass

$$\begin{bmatrix} G_x \\ G_y \end{bmatrix} = m \begin{bmatrix} b_y \\ -b_x \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} b_x \\ b_y \end{bmatrix} = {}^0\mathbf{b}_{COM} \tag{8.46}$$

Position-based markers can measure position $\mathbf{p}_{\text{config}}({}^b\mathbf{b})$ depending on the local position ${}^b\mathbf{b}$. The **position jacobian** depends on the local position ${}^b\mathbf{b}$ and is defined as,

$${}^0\mathbf{J}_{pos} = \partial\, {}^0\mathbf{p}_{\text{config}}({}^b\mathbf{b})_{\text{cur}} / \partial \mathbf{c}_{\text{cur}} = \begin{bmatrix} 1 & 0 & -\sin(\theta)\,{}^bb_0 - \cos(\theta)\,{}^bb_1 \\ 0 & 1 & \cos(\theta)\,{}^bb_0 - \sin(\theta)\,{}^bb_1 \\ 0 & 0 & 0 \end{bmatrix} \tag{8.47}$$

which transforms the action of global forces ${}^0\mathbf{f}$ of position-based markers on the coordinates $\mathbf{c}$,

$$\mathbf{Q} = {}^0\mathbf{J}_{pos}^\mathrm{T}\ {}^0\mathbf{f}_a \tag{8.48}$$

Note that a LoadCoordinate on coordinate 2 of the node would add a torque τ_2 on the RHS. The **rotation jacobian**, which is computed from angular velocity, reads

$${}^0\mathbf{J}_{rot} = \partial\, {}^0\boldsymbol{\omega}_{\text{cur}} / \partial \dot{\mathbf{c}}_{\text{cur}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{8.49}$$

and transforms the action of global torques ${}^0\boldsymbol{\tau}$ of orientation-based markers on the coordinates $\mathbf{c}$,

$$\mathbf{Q} = {}^0\mathbf{J}_{rot}^{\mathrm{T}} \; {}^0\boldsymbol{\tau}_a \tag{8.50}$$

Userfunction: `graphicsDataUserFunction(mbs, itemNumber)`

A user function, which is called by the visualization thread in order to draw user-defined objects. The function can be used to generate any `BodyGraphicsData`, see Section 10.4. Use `exudyn.graphics` functions, see Section 7.8, to create more complicated objects. Note that `graphicsDataUserFunction` needs to copy lots of data and is therefore inefficient and only designed to enable simpler tests, but not large scale problems.

For an example for `graphicsDataUserFunction` see ObjectGround, Section 8.2.1.

arguments / return	type or size	description
`mbs`	MainSystem	provides reference to mbs, which can be used in user function to access all data of the object
`itemNumber`	int	integer number of the object in mbs, allowing easy access
`return value`	BodyGraphicsData	list of `GraphicsData` dictionaries, see Section 10.4

8.2.7.4 MINI EXAMPLE for ObjectRigidBody2D

```
node = mbs.AddNode(NodeRigidBody2D(referenceCoordinates = [1,1,0.25*np.pi],
                                   initialCoordinates=[0.5,0,0],
                                   initialVelocities=[0.5,0,0.75*np.pi]))
mbs.AddObject(RigidBody2D(nodeNumber = node, physicsMass=1, physicsInertia=2))

#assemble and solve system for default parameters
mbs.Assemble()
mbs.SolveDynamic()

#check result
exudynTestGlobals.testResult = mbs.GetNodeOutput(node, exu.OutputVariableType.Position)
[0]
exudynTestGlobals.testResult+= mbs.GetNodeOutput(node, exu.OutputVariableType.
Coordinates)[2]
#final x-coordinate of position shall be 2, angle theta shall be np.pi
```

For examples on ObjectRigidBody2D see Relevant Examples and TestModels with weblink:

- `beltDriveALE.py` (Examples/)

- beltDriveReevingSystem.py (Examples/)
- beltDrivesComparison.py (Examples/)
- reevingSystem.py (Examples/)
- reevingSystemOpen.py (Examples/)
- sliderCrank3DwithANCFbeltDrive2.py (Examples/)
- ANCFmovingRigidbody.py (Examples/)
- ANCFslidingJoint2D.py (Examples/)
- ANCFslidingJoint2Drigid.py (Examples/)
- ANCFswitchingSlidingJoint2D.py (Examples/)
- doublePendulum2D.py (Examples/)
- finiteSegmentMethod.py (Examples/)
- ...
- ANCFbeltDrive.py (TestModels/)
- ANCFgeneralContactCircle.py (TestModels/)
- ANCFcontactFrictionTest.py (TestModels/)
- ...

8.3 Objects (SuperElement)

A SuperElement is a special Object which acts on a set of nodes. Essentially, SuperElements can be linked with special SuperElement markers. SuperElements may represent complex flexible bodies, based on finite element formulations.

8.3.1 ObjectGenericODE2

A system of n second order ordinary differential equations (ODE2), having a mass matrix, damping/-gyroscopic matrix, stiffness matrix and generalized forces. It can combine generic nodes, or node points. User functions can be used to compute mass matrix and generalized forces depending on given coordinates. NOTE that all matrices, vectors, etc. must have the same dimensions n or ($n \times n$), or they must be empty (0×0), except for the mass matrix which always needs to have dimensions ($n \times n$).

Additional information for ObjectGenericODE2:

- This `Object` has/provides the following types = `Body`, `MultiNoded`, `SuperElement`
- Requested `Node` type: read detailed information of item

The item **ObjectGenericODE2** with type = 'GenericODE2' has the following parameters:

Name	type	size	default value	description
name	String		''	objects's unique name
nodeNumbers	ArrayNodeIndex		[]	node numbers which provide the coordinates for the object (consecutively as provided in this list)
massMatrix	PyMatrixContainer		PyMatrixContainer[]	mass matrix of object as MatrixContainer (or numpy array / list of lists)
stiffnessMatrix	PyMatrixContainer		PyMatrixContainer[]	stiffness matrix of object as MatrixContainer (or numpy array / list of lists); NOTE that (dense/sparse triplets) format must agree with dampingMatrix and jacobianUserFunction
dampingMatrix	PyMatrixContainer		PyMatrixContainer[]	damping matrix of object as MatrixContainer (or numpy array / list of lists); NOTE that (dense/sparse triplets) format must agree with stiffnessMatrix and jacobianUserFunction
forceVector	NumpyVector		[]	generalized force vector added to RHS
forceUserFunction	PyFunctionVectorMbsScalarIndex2Vector		0	A Python user function which computes the generalized user force vector for the ODE2 equations; see description below

massMatrixUserFunction	PyFunctionMatrixContainerMbsScalarIndex2Vector		0	A Python user function which computes the mass matrix instead of the constant mass matrix given in **M**; return numpy array or MatrixContainer; see description below
jacobianUserFunction	PyFunctionMatrixContainerMbsScalarIndex2Vector2Scalar		0	A Python user function which computes the jacobian, i.e., the derivative of the left-hand-side object equation w.r.t. the coordinates (times f_{ODE2}) and w.r.t. the velocities (times f_{ODE2_t}). Terms on the RHS must be subtracted from the LHS equation; the respective terms for the stiffness matrix and damping matrix are automatically added; see description below
coordinateIndexPerNode	ArrayIndex		[]	this list contains the local coordinate index for every node, which is needed, e.g., for markers; the list is generated automatically every time parameters have been changed
tempCoordinates	NumpyVector		[]	temporary vector containing coordinates
tempCoordinates_t	NumpyVector		[]	temporary vector containing velocity coordinates
tempCoordinates_tt	NumpyVector		[]	temporary vector containing acceleration coordinates
visualization	VObjectGenericODE2			parameters for visualization of item

The item VObjectGenericODE2 has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
color	Float4	4	[-1.,-1.,-1.,-1.]	RGBA color for object; 4th value is alpha-transparency; R=-1.f means, that default color is used
triangleMesh	NumpyMatrixI		MatrixI[]	a matrix, containg node number triples in every row, referring to the node numbers of the GenericODE2 object; the mesh uses the nodes to visualize the underlying object; contour plot colors are still computed in the local frame!
showNodes	Bool		False	set true, nodes are drawn uniquely via the mesh, eventually using the floating reference frame, even in the visualization of the node is show=False; node numbers are shown with indicator 'NF'

graphicsDataUserFunction	PyFunctionGraphicsData		0	A Python function which returns a bodyGraphicsData object, which is a list of graphics data in a dictionary computed by the user function; the graphics data is draw in global coordinates; it can be used to implement user element visualization, e.g., beam elements or simple mechanical systems; note that this user function may significantly slow down visualization

---

8.3.1.1 DESCRIPTION of ObjectGenericODE2:

Information on input parameters:

input parameter	symbol	description see tables above
nodeNumbers	$\mathbf{n}_n = [n_0, \ldots, n_n]^\mathrm{T}$	
massMatrix	$\mathbf{M} \in \mathbb{R}^{n \times n}$	
stiffnessMatrix	$\mathbf{K} \in \mathbb{R}^{n \times n}$	
dampingMatrix	$\mathbf{D} \in \mathbb{R}^{n \times n}$	
forceVector	$\mathbf{f} \in \mathbb{R}^n$	
forceUserFunction	$\mathbf{f}_{user} \in \mathbb{R}^n$	
massMatrixUserFunction	$\mathbf{M}_{user} \in \mathbb{R}^{n \times n}$	
jacobianUserFunction	$\mathbf{J}_{user} \in \mathbb{R}^{n \times n}$	
tempCoordinates	$\mathbf{c}_{temp} \in \mathbb{R}^n$	
tempCoordinates_t	$\dot{\mathbf{c}}_{temp} \in \mathbb{R}^n$	
tempCoordinates_tt	$\ddot{\mathbf{c}}_{temp} \in \mathbb{R}^n$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
CoordinatesTotal		all ODE2 displacement plus reference coordinates of object
Coordinates		all ODE2 (displacement) coordinates
Coordinates_t		all ODE2 velocity coordinates
Coordinates_tt		all ODE2 acceleration coordinates
Force		generalized forces for all coordinates (residual of all forces except mass*accleration; corresponds to ComputeODE2LHS)

8.3.1.2 Additional output variables for superelement node access

Functions like `GetObjectOutputSuperElement(...)`, see Section 6.5.4, or `SensorSuperElement`, see Section 6.5.7, directly access special output variables (`OutputVariableType`) of the mesh nodes of the superelement. Additionally, the contour drawing of the object can make use the `OutputVariableType` of the meshnodes.

For this object, all nodes of `ObjectGenericODE2` map their `OutputVariableType` to the meshnode → see at the according node for the list of `OutputVariableType`.

8.3.1.3 Equations of motion

An object with node numbers $[n_0, \ldots, n_n]$ and according numbers of nodal coordinates $[n_{c_0}, \ldots, n_{c_n}]$, the total number of equations (=coordinates) of the object is

$$n = \sum_i n_{c_i}, \tag{8.51}$$

which is used throughout the description of this object.

8.3.1.4 Equations of motion

The equations of motion read,

$$\mathbf{M}\ddot{\mathbf{q}} + \mathbf{D}\dot{\mathbf{q}} + \mathbf{K}\mathbf{q} = \mathbf{f} + \mathbf{f}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}}) \tag{8.52}$$

Note that the user function $\mathbf{f}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}})$ may be empty (=0), and `iN` represents the itemNumber (=objectNumber).

In case that a user mass matrix is specified, Eq. (8.52) is replaced with

$$\mathbf{M}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}})\ddot{\mathbf{q}} + \mathbf{D}\dot{\mathbf{q}} + \mathbf{K}\mathbf{q} = \mathbf{f} + \mathbf{f}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}}) \tag{8.53}$$

The (internal) Jacobian $\mathbf{J}$ of Eq. (8.52) (assuming $\mathbf{f}$ to be constant!) reads

$$\mathbf{J} = f_{ODE2}\left(\mathbf{K} - \frac{\partial \mathbf{f}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}})}{\partial \mathbf{q}}\right) + f_{ODE2_t}\left(\mathbf{D} - \frac{\partial \mathbf{f}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}})}{\partial \dot{\mathbf{q}}}\right) + \tag{8.54}$$

Chosing $f_{ODE2} = 1$ and $f_{ODE2_t} = 0$ would immediately give the jacobian of position quantities.

If no `jacobianUserFunction` is specified, the jacobian is – as with many objects in `Exudyn` – computed by means of numerical differentiation. In case that a `jacobianUserFunction` is specified, it must represent the jacobian of the LHS of Eq. (8.52) without $\mathbf{K}$ and $\mathbf{D}$ (these matrices are added internally),

$$\mathbf{J}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}}, f_{ODE2}, f_{ODE2_t}) = -f_{ODE2}\left(\frac{\partial \mathbf{f}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}})}{\partial \mathbf{q}}\right) - f_{ODE2_t}\left(\frac{\partial \mathbf{f}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}})}{\partial \dot{\mathbf{q}}}\right) \tag{8.55}$$

For clarification also see the **example** in `TestModels/linearFEMgenericODE2.py`.

CoordinateLoads are added for the respective ODE2 coordinate on the RHS of the latter equation.

Userfunction: `forceUserFunction(mbs, t, itemNumber, q, q_t)`

A user function, which computes a force vector depending on current time and states of object. Can be used to create any kind of mechanical system by using the object states. Note that itemNumber represents the index of the ObjectGenericODE2 object in mbs, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs to which object belongs
`t`	Real	current time in mbs
`itemNumber`	Index	integer number i_N of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
`q`	Vector $\in \mathbb{R}^n$	object coordinates (e.g., nodal displacement coordinates) in current configuration, without reference values
`q_t`	Vector $\in \mathbb{R}^n$	object velocity coordinates (time derivative of `q`) in current configuration
`return value`	Vector $\in \mathbb{R}^n$	returns force vector for object

---

Userfunction: `massMatrixUserFunction(mbs, t, itemNumber, q, q_t)`

A user function, which computes a mass matrix depending on current time and states of object. Can be used to create any kind of mechanical system by using the object states.

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs to which object belongs to
`t`	Real	current time in mbs
`itemNumber`	Index	integer number i_N of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
`q`	Vector $\in \mathbb{R}^n$	object coordinates (e.g., nodal displacement coordinates) in current configuration, without reference values
`q_t`	Vector $\in \mathbb{R}^n$	object velocity coordinates (time derivative of `q`) in current configuration
`return value`	MatrixContainer $\in \mathbb{R}^{n \times n}$	returns mass matrix for object, as exu.MatrixContainer, numpy array or list of lists; use MatrixContainer sparse format for larger matrices to speed up computations.

Userfunction: `jacobianUserFunction(mbs, t, itemNumber, q, q_t, fODE2, fODE2_t)`

A user function, which computes the jacobian of the LHS of the equations of motion, depending on current time, states of object and two factors which are used to distinguish between position level and velocity level derivatives. Can be used to create any kind of mechanical system by using the object states.

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs to which object belongs to
`t`	Real	current time in mbs
`itemNumber`	Index	integer number i_N of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
`q`	Vector $\in \mathbb{R}^n$	object coordinates (e.g., nodal displacement coordinates) in current configuration, without reference values
`q_t`	Vector $\in \mathbb{R}^n$	object velocity coordinates (time derivative of `q`) in current configuration
`fODE2`	Real	factor to be multiplied with the position level jacobian, see Eq. (8.55)
`fODE2_t`	Real	factor to be multiplied with the velocity level jacobian, see Eq. (8.55)
`return value`	MatrixContainer $\in \mathbb{R}^{n \times n}$	returns special jacobian for object, as exu.MatrixContainer, numpy array or list of lists; use MatrixContainer sparse format for larger matrices to speed up computations; NOTE that the format of returnValue must AGREE with (dense/sparse triplet) format of stiffnessMatrix and dampingMatrix; sparse triplets MAY NOT contain zero values!

Userfunction: `graphicsDataUserFunction(mbs, itemNumber)`

A user function, which is called by the visualization thread in order to draw user-defined objects. The function can be used to generate any `BodyGraphicsData`, see Section 10.4. Use `exudyn.graphics` functions, see Section 7.8, to create more complicated objects. Note that `graphicsDataUserFunction` needs to copy lots of data and is therefore inefficient and only designed to enable simpler tests, but not large scale problems.

For an example for `graphicsDataUserFunction` see ObjectGround, Section 8.2.1.

arguments / return	type or size	description
`mbs`	MainSystem	provides reference to mbs, which can be used in user function to access all data of the object

itemNumber	Index	integer number of the object in mbs, allowing easy access
return value	BodyGraphicsData	list of GraphicsData dictionaries, see Section 10.4

User function example:

```
    #user function, using variables M, K, ... from mini example, replacing
ObjectGenericODE2(...)
    KD = numpy.diag([200,100])
    #nonlinear force example; this force is added to right-hand-side ==> negative sign!
    def UFforce(mbs, t, itemNumber, q, q_t):
        return -np.dot(KD, q_t*q) #add nonlinear term for q_t and q, q_t*q gives vector

    #non-constant mass matrix:
    def UFmass(mbs, t, itemNumber, q, q_t):
        return (q[0]+1)*M #uses mass matrix from mini example

    #non-constant mass matrix:
    def UFgraphics(mbs, itemNumber):
        t = mbs.systemData.GetTime(exu.ConfigurationType.Visualization) #get time if
needed
        p = mbs.GetObjectOutputSuperElement(objectNumber=itemNumber, variableType = exu.
OutputVariableType.Position,
                                            meshNodeNumber = 0, #get first node's
position
                                            configuration = exu.ConfigurationType.
Visualization)
        graphics1=graphics.Sphere(point=p,radius=0.1, color=graphics.color.red)
            graphics2 = {'type':'Line', 'data': list(p)+[0,0,0], 'color':graphics.color.
blue}
        return [graphics1, graphics2]

    #now add object instead of object in mini-example:
    oGenericODE2 = mbs.AddObject(ObjectGenericODE2(nodeNumbers=[nMass0,nMass1],
                        massMatrix=M, stiffnessMatrix=K, dampingMatrix=D,
                        forceUserFunction=UFforce, massMatrixUserFunction=UFmass,
                        visualization=VObjectGenericODE2(graphicsDataUserFunction=
UFgraphics)))
```

8.3.1.5 MINI EXAMPLE for ObjectGenericODE2

```
#set up a mechanical system with two nodes; it has the structure: |~~M0~~M1
nMass0 = mbs.AddNode(NodePoint(referenceCoordinates=[0,0,0]))
nMass1 = mbs.AddNode(NodePoint(referenceCoordinates=[1,0,0]))

mass = 0.5 * np.eye(3)      #mass of nodes
stif = 5000 * np.eye(3)     #stiffness of nodes
damp = 50 * np.eye(3)      #damping of nodes
Z = 0. * np.eye(3)          #matrix with zeros
#build mass, stiffness and damping matrices (:
M = np.block([[mass,         0.*np.eye(3)],
              [0.*np.eye(3), mass        ] ])
K = np.block([[2*stif, -stif],
              [ -stif,  stif] ])
D = np.block([[2*damp, -damp],
              [ -damp,  damp] ])

oGenericODE2 = mbs.AddObject(ObjectGenericODE2(nodeNumbers=[nMass0,nMass1],
                                               massMatrix=M,
                                               stiffnessMatrix=K,
                                               dampingMatrix=D))

mNode1 = mbs.AddMarker(MarkerNodePosition(nodeNumber=nMass1))
mbs.AddLoad(Force(markerNumber = mNode1, loadVector = [10, 0, 0])) #static solution
=10*(1/5000+1/5000)=0.0004

#assemble and solve system for default parameters
mbs.Assemble()

mbs.SolveDynamic(solverType = exudyn.DynamicSolverType.TrapezoidalIndex2)

#check result at default integration time
exudynTestGlobals.testResult = mbs.GetNodeOutput(nMass1, exu.OutputVariableType.Position
)[0]
```

For examples on ObjectGenericODE2 see Relevant Examples and TestModels with weblink:

- kinematicTreeAndMBS.py (Examples/)
- nMassOscillator.py (Examples/)
- nMassOscillatorInteractive.py (Examples/)
- simulateInteractively.py (Examples/)
- ACFtest.py (TestModels/)
- coordinateVectorConstraintGenericODE2.py (TestModels/)
- genericODE2test.py (TestModels/)
- linearFEMgenericODE2.py (TestModels/)

- linearFEMgenericODE2Test.py (TestModels/)
- objectFFRFTest.py (TestModels/)
- objectGenericODE2Test.py (TestModels/)
- rigidBodyAsUserFunctionTest.py (TestModels/)
- solverExplicitODE1ODE2test.py (TestModels/)

8.3.2 ObjectKinematicTree

A special object to represent open kinematic trees using minimal coordinate formulation. The kinematic tree is defined by lists of joint types, parents, inertia parameters (w.r.t. COM), etc. per link (body) and given joint (pre) transformations from the previous joint. Every joint / link is defined by the position and orientation of the previous joint and a coordinate transformation (incl. translation) from the previous link's to this link's joint coordinates. The joint can be combined with a marker, which allows to attach connectors as well as joints to represent closed loop mechanisms. Efficient models can be created by using tree structures in combination with constraints and very long chains should be avoided and replaced by (smaller) jointed chains if possible. The class Robot from exudyn.robotics can also be used to create kinematic trees, which are then exported as KinematicTree or as redundant multibody system. Use specialized settings in VisualizationSettings.bodies.kinematicTree for showing joint frames and other properties.

Additional information for ObjectKinematicTree:

- This `Object` has/provides the following types = `Body`, `MultiNoded`, `SuperElement`
- Requested `Node` type = `GenericODE2`
- **Short name** for Python = `KinematicTree`
- **Short name** for Python visualization object = `VKinematicTree`

The item **ObjectKinematicTree** with type = 'KinematicTree' has the following parameters:

Name	type	size	default value	description
name	String		''	objects's unique name
nodeNumber	NodeIndex		invalid (-1)	node number (type NodeIndex) of GenericODE2 node containing the coordinates for the kinematic tree; n being the number of minimal coordinates
gravity	Vector3D		[0.,0.,0.]	gravity vector in inertial coordinates; used to simply apply gravity as LoadMassProportional is not available for KinematicTree
baseOffset	Vector3D		[0.,0.,0.]	offset vector for base, in global coordinates
jointTypes	JointTypeList		[]	joint types of kinematic Tree joints, using exu.JointType, like exu.JointType.RevoluteZ; must be always set
linkParents	ArrayIndex		[]	index of parent joint/link; if no parent exists, the value is -1; by default, $p_0 = -1$ because the ith parent index must always fulfill $p_i < i$; must be always set

jointTransformations	Matrix3DList		[]	list of constant joint transformations from parent joint coordinates p_0 to this joint coordinates j_0; this allows to adjust the orientation of the joint axes (but it does not affect the joint offset); if no parent exists (−1), the base coordinate system 0 is used; must be always set
jointOffsets	Vector3DList		[]	list of constant joint offsets from parent joint to this joint; p_0, p_1, ... denote the parent coordinate systems; this means that the joint offset is added prior to performing the joint transformation; if no parent exists (−1), the base coordinate system 0 is used; must be always set
linkInertiasCOM	Matrix3DList		[]	list of link inertia tensors w.r.t. COM in joint/link j_i coordinates; must be always set
linkCOMs	Vector3DList		[]	list of vectors for center of mass (COM) in joint/link j_i coordinates; must be always set
linkMasses	Vector		[]	masses of links; must be always set
linkForces	Vector3DList		[]	list of 3D force vectors per link in global coordinates acting on joint frame origin; use force-torque couple to realize off-origin forces; defaults to empty list [], adding no forces
linkTorques	Vector3DList		[]	list of 3D torque vectors per link in global coordinates; defaults to empty list [], adding no torques
jointForceVector	Vector		[]	generalized force vector per coordinate added to RHS of EOM; represents a torque around the axis of rotation in revolute joints and a force in prismatic joints; for a revolute joint i, the torque $f[i]$ acts positive (w.r.t. rotation axis) on link i and negative on parent link p_i; must be either empty list/array [] (default) or have size n
jointPositionOffsetVector	Vector		[]	offset for joint coordinates used in P(D) control; acts in positive joint direction similar to jointForceVector; should be modified, e.g., in preStepUserFunction; must be either empty list/array [] (default) or have size n
jointVelocityOffsetVector	Vector		[]	velocity offset for joint coordinates used in (P)D control; acts in positive joint direction similar to jointForceVector; should be modified, e.g., in preStepUserFunction; must be either empty list/array [] (default) or have size n

jointPControlVector	Vector		[]	proportional (P) control values per joint (multiplied with position error between joint value and offset $\mathbf{u}_o$); note that more complicated control laws must be implemented with user functions; must be either empty list/array [] (default) or have size n
jointDControlVector	Vector		[]	derivative (D) control values per joint (multiplied with velocity error between joint velocity and velocity offset $\mathbf{v}_o$); note that more complicated control laws must be implemented with user functions; must be either empty list/array [] (default) or have size n
forceUserFunction	PyFunctionVectorMbsScalarIndex2Vector		0	A Python user function which computes the generalized force vector on RHS with identical action as jointForceVector; see description below
visualization	VObjectKinematicTree			parameters for visualization of item

The item VObjectKinematicTree has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
showLinks	Bool		True	set true, if links shall be shown; if graphicsDataList is empty, a standard drawing for links is used (drawing a cylinder from previous joint or base to next joint; size relative to frame size in KinematicTree visualization settings); else graphicsDataList are used per link; NOTE visualization of joint and COM frames can be modified via visualizationSettings.bodies.kinematicTree
showJoints	Bool		True	set true, if joints shall be shown; if graphicsDataList is empty, a standard drawing for joints is used (drawing a cylinder for revolute joints; size relative to frame size in KinematicTree visualization settings)
color	Float4	4	[-1.,-1.,-1.,-1.]	RGBA color for object; 4th value is alpha-transparency; R=-1.f means, that default color is used

graphicsDataList	BodyGraphicsDataList			Structure contains data for link/joint visualization; data is defined as list of BodyGraphicsData where every BodyGraphicsData corresponds to one link/joint; must either be emtpy list or length must agree with number of links

---

8.3.2.1 DESCRIPTION of ObjectKinematicTree:

Information on input parameters:

input parameter	symbol	description see tables above
nodeNumber	$n_0 \in \mathbb{N}^n$	
gravity	${}^0\mathbf{g} \in \mathbb{R}^3$	
baseOffset	${}^0\mathbf{p}_b \in \mathbb{R}^3$	
jointTypes	$\mathbf{j}_T \in \mathbb{N}^n$	
linkParents	$\mathbf{i}_p = [p_0, p_1, \ldots] \in \mathbb{N}^n$	
jointTransformations	$\mathbf{T} = [{}^{p_0,j_0}\mathbf{T}_0\,, {}^{p_1,j_1}\mathbf{T}_1\,, \ldots] \in [\mathbb{R}^{3\times3}, ...]$	
jointOffsets	$\mathbf{V} = [{}^{p_0}o_0\,, {}^{p_1}o_1\,, \ldots] \in [\mathbb{R}^3, ...]$	
linkInertiasCOM	$\mathbf{J}_{COM} = [{}^{j_0}\mathbf{J}_0\,, {}^{j_1}\mathbf{J}_1\,, \ldots] \in [\mathbb{R}^{3\times3}, ...]$	
linkCOMs	$\mathbf{C} = [{}^{j_0}\mathbf{c}_0\,, {}^{j_1}\mathbf{c}_1\,, \ldots] \in [\mathbb{R}^3, ...]$	
linkMasses	$\mathbf{m} \in \mathbb{R}^n$	
linkForces	${}^0\mathbf{F} \in [\mathbb{R}^3, ...]$	
linkTorques	${}^0\mathbf{F}_\tau \in [\mathbb{R}^3, ...]$	
jointForceVector	$\mathbf{f} \in \mathbb{R}^n$	
jointPositionOffsetVector	$\mathbf{u}_o \in \mathbb{R}^n$	
jointVelocityOffsetVector	$\mathbf{v}_o \in \mathbb{R}^n$	
jointPControlVector	$\mathbf{P} \in \mathbb{R}^n$	
jointDControlVector	$\mathbf{D} \in \mathbb{R}^n$	
forceUserFunction	$\mathbf{f}_{user} \in \mathbb{R}^n$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Coordinates		all ODE2 joint coordinates, including reference values (which is slightly inconsistent with CoordinatesTotal used in nodes); if you need values without reference part, read out the node; these are the minimal coordinates of the object
Coordinates_t		all ODE2 velocity coordinates
Coordinates_tt		all ODE2 acceleration coordinates

Force		generalized forces for all coordinates (residual of all forces except mass*accleration; corresponds to ComputeODE2LHS)

8.3.2.2 SensorKinematicTree output variables

The following output variables are available with `SensorKinematicTree` for a specific link. Within the link n_i, a local position ${}^{n_i}\mathbf{p}_{n_i}$ is required. All output variables are available for different configurations. Furthermore, ${}^{0,n_i}\mathbf{T}$ is the homogeneous transformation from link n_i coordinates to global coordinates.

Kinematic tree output variables	**symbol**	**description**
Position	${}^{0}\mathbf{p}_{n_i} = {}^{0,n_i}\mathbf{T}\ {}^{n_i}\mathbf{p}_{n_i}$	global position of local position at link n_i
Displacement	${}^{0}\mathbf{u}_{n_i} = {}^{0,n_i}\mathbf{T}\ {}^{n_i}\mathbf{p}_{n_i} - {}^{0}\mathbf{p}_{n_{i,\mathrm{ref}}}$	global displacement of local position at link n_i
Rotation	$\boldsymbol{\phi}_{n_i}$	Tait-Bryan angles of link n_i
RotationMatrix	${}^{0,n_i}\mathbf{A}_{n_i}$	rotation matrix of link n_i
VelocityLocal	${}^{n_i}\mathbf{v}_{n_i}$	local velocity of local position at link n_i
Velocity	${}^{0}\mathbf{v}_{n_i} = {}^{0,n_i}\dot{\mathbf{T}}\ {}^{n_i}\mathbf{p}_{n_i}$	global velocity of local position at link n_i
VelocityLocal	${}^{n_i}\mathbf{v}_{n_i}$	local velocity of local position at link n_i
Acceleration	${}^{0}\mathbf{a}_{n_i} = {}^{0,n_i}\dot{\mathbf{T}}\ {}^{n_i}\mathbf{p}_{n_i}$	global acceleration of local position at link n_i
AccelerationLocal	${}^{n_i}\mathbf{a}_{n_i}$	local acceleration of local position at link n_i
AngularVelocity	${}^{0}\boldsymbol{\omega}_{n_i}$	global angular velocity of local position at link n_i
AngularVelocityLocal	${}^{n_i}\boldsymbol{\omega}_{n_i}$	local angular velocity of local position at link n_i
AngularAcceleration	${}^{0}\boldsymbol{\alpha}_{n_i}$	global angular acceleration of local position at link n_i
AngularAccelerationLocal	${}^{n_i}\boldsymbol{\alpha}_{n_i}$	local angular acceleration of local position at link n_i

8.3.2.3 General notes

The `KinematicTree` object is used to represent the equations of motion of a (open) tree-structured multibody system using a minimal set of coordinates. Even though that `Exudyn` is based on redundant coordinates, the `KinematicTree` allows to efficiently model standard multibody models based on revolute and prismatic joints. Especially, a chain with 3 links leads to only 3 equations of motion, while a redundant formulation would lead to 3×7 coordinates using Euler Parameters and 3×6 constraints for joints and Euler parameters, which gives a set of 39 equations. However this set of equations is very sparse and the evaluation is much faster than the kinematic tree.

The question, which formulation to chose cannot be answered uniquely. However, `KinematicTree` objects do not include constraints, so they can be solved with explicit solvers. Furthermore, the joint values (angels) can be addressed directly – controllers or sensors are generally simpler.

8.3.2.4 General

The equations follow the description given in Chapters 2 and 3 in the handbook of robotics, 2016 edition [57].

Functions like `GetObjectOutputSuperElement(...)`, see Section 6.5.4, or `SensorSuperElement`, see Section 6.5.7, directly access special output variables (`OutputVariableType`) of the (mesh) nodes of the superelement. The mesh nodes are the links of the `KinematicTree`.

Note, however, that some functionality is considerably different for `ObjectGenericODE2`.

8.3.2.5 Equations of motion

The `KinematicTree` has one node of type `NodeGenericODE2` with n coordinates. The equations of motion are built by special multibody algorithms, following Featherstone [12]. For a short introduction into this topic, see Chapter 3 of [57].

The kinematic tree defines a set of rigid bodies connected by joints, having no loops. In this way, every body i, also denoted as link, has either a previous body $p(i) \neq -1$ or not. The previous body for body i is $p(i)$. The coordinates of joint i are defined as q_i.

The following joint transformations are considered (as homogeneous transformations):

- $\mathbf{X}_J(i)$ ... joint transformation due to rotation or translation
- $\mathbf{X}_L(i)$ ... link transformation (e.g. given by kinematics of mechanism)
- ${}^{i,-1}\mathbf{X}$ ... transformation from global (-1) to local joint i coordinates
- ${}^{i,p(i)}\mathbf{X}$ ... transformation from previous joint to joint i coordinates

Furthermore, we use

$\mathbf{\Phi}_i$ ... motion subspace for joint i

which denotes the transformation from joint coordinate (scalar) to rotations and translations. We can compute the local joint angular velocity $\boldsymbol{\omega}_i$ and translational velocity $\mathbf{w}_i$, as a 6D vector $\mathbf{v}_i^J$, from

$$\mathbf{v}_i^J = \begin{bmatrix} \boldsymbol{\omega}_i \\ \mathbf{w}_i \end{bmatrix} = \mathbf{\Phi}_i \, \dot{q}_i \tag{8.56}$$

The joint coordinates, which can be rotational or translational, are stored in the vector

$$\mathbf{q} = [q_0, \ldots, q_{N_B-1}]^{\mathrm{T}} , \tag{8.57}$$

and the vector of joint velocity coordinates reads

$$\dot{\mathbf{q}} = [\dot{q}_0, \ldots, \dot{q}_{N_B-1}]^{\mathrm{T}} . \tag{8.58}$$

Knowing the motion subspace $\mathbf{\Phi}_i$ for joint i, the velocity of joint i reads

$$\mathbf{v}_i = \mathbf{v}_{p(i)} + \mathbf{\Phi}_i \, \dot{q}_i , \tag{8.59}$$

and accelerations follow as

$$\mathbf{a}_i = \mathbf{a}_{p(i)} + \mathbf{\Phi}_i \, \ddot{q}_i + \dot{\mathbf{\Phi}}_i \, \dot{q}_i . \tag{8.60}$$

Note that the previous formulas can be interpreted coordinate free, but they are usually implemented in joint coordinates.

The local forces due to applied forces and inertia forces are computed, for now independently, for every link,

$$\mathbf{f}_i = \mathbf{I}_i \mathbf{a}_i + \mathbf{v}_i \times \mathbf{I}_i \mathbf{v}_i - {}^{i,-1}\mathbf{X}^{\mathrm{T}} \cdot {}^{-1}\mathbf{f}^a \tag{8.61}$$

The total forces can be computed from inverse dynamics. At every free end of the tree, the forces are added up for the previous link, which needs to be done recursively starting at the leaves of the tree,

$$\mathbf{f}_{p(i)} \mathrel{+}= {}^{i,p(i)}\mathbf{X}^{\mathrm{T}} \cdot \mathbf{f}_i \tag{8.62}$$

The mass matrix is then built by recursively computing the intertia of the links and adding the joint contributions by projecting the local inertia into the joint motion space, see the composite-rigid-body algorithm.

Note that $\cdot$ for multiplication of matrices and vectors is added for clarity, especially in case of left and right indices. The whole algorithm for forward and inverse dynamics is given in the following figures.

8.3.2.6 Implementation and user functions

Currently, there is only the so-called Composite-Rigid-Body (CRB) algorithm implemented. This algorithm does not show the highest performance, but creates the mass matrix $\mathbf{M}_{CRB}$ and forces $\mathbf{f}_{CRB}$ in a conventional form. The equations read

$$\mathbf{M}_{CRB}(\mathbf{q})\ddot{\mathbf{q}} = \mathbf{f}_{CRB}(\mathbf{q}, \dot{\mathbf{q}}) + \mathbf{f} + \mathbf{f}_{PD} + \mathbf{f}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}}) \tag{8.63}$$

The term $\mathbf{f}_{CRB}(\mathbf{q}, \dot{\mathbf{q}})$ represents inertial terms, which are due to accelerations and quadratic velocities and is computed by `ComputeODE2LHS`. Note that the user function $\mathbf{f}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}})$ may be empty (=0), and `iN` represents the itemNumber (=objectNumber). The force $\mathbf{f}$ is given by the `jointForceVector`, which also may have zero length, causing it to be ignored. While $\mathbf{f}$ is constant, it may be varied using a `mbs.preStepUserFunction`, which can then represent any force over time. Note that such changes are not considered in the object's jacobian.

The user force $\mathbf{f}_{user}$ is described below and may represent any force over time. Note that this force is considered in the object's jacobian, but it does not include external dependencies – if a control law is feeds back measured quantities and couples them to forces. This leads to worse performance (up to non-convergence) of implicit solvers.

The control force $\mathbf{f}_{PD}$ realizes a simple linear control law

$$\mathbf{f}_{PD} = \mathbf{P} \cdot (\mathbf{u}_o - \mathbf{q}) + \mathbf{D} \cdot (\mathbf{v}_o - \dot{\mathbf{q}}) \tag{8.64}$$

Here, the '.' operator represents an element-wise multiplication of two vectors, resulting in a vector. The force $\mathbf{f}_{PD}$ at the RHS acts in direction of prescribed joint motion $\mathbf{u}_o$ and prescribed joint velocities $\mathbf{v}_o$ multiplied with proportional and 'derivative' factors P and D. Omitting $\mathbf{u}_o$ and $\mathbf{v}_o$ and putting $\mathbf{f}_{PD}$ on the LHS, we immediately can interpret these terms as stiffness and damping on the single coordinates. The control force is also considered in the object's jacobian, which is currently computed by numerical differentiation.

Algorithm 1 Recursive Newton-Euler algorithm (acc. to Featherstone). The symbol '#' represents comments.

function RNEA($\mathbf{q}$, $\dot{\mathbf{q}}$, MotionSubspace(i), $\mathbf{X}_L$, ${}^{-1}\mathbf{f}_i^a$, assume $\dot{\mathbf{\Phi}}_i = 0$)

$\mathbf{v}_{-1} = \mathbf{0}$
$\mathbf{a}_{-1} = -\mathbf{g}$ # gravity vector
# loop over N_B bodies:
for $i = 0$ **to** $N_B - 1$ **do**
  # compute forward transformations:
  $\mathbf{X}_J(i) = \mathbf{X}_{JT}(i, q_i)$
  ${}^{i,p(i)}\mathbf{X} = \mathbf{X}_J(i)\,\mathbf{X}_L(i)$
  $\mathbf{\Phi}_i$ = MotionSubspace(i)
  if $p(i) \neq -1$ **then**
    ${}^{i,-1}\mathbf{X} = {}^{i,p(i)}\mathbf{X} \cdot {}^{p(i),-1}\mathbf{X}$
  end if
  # compute forward kinematics:
  $\mathbf{v}_i = \mathbf{v}_{p(i)} + \mathbf{\Phi}_i\,\dot{q}_i$,
  $\mathbf{a}_i = \mathbf{a}_{p(i)} + \mathbf{v}_i \times \mathbf{\Phi}_i\,\dot{q}_i$ #$\mathbf{\Phi}_i\,\ddot{q}_i$ put on RHS
  $\mathbf{f}_i = \mathbf{I}_i\mathbf{a}_i + \mathbf{v}_i \times \mathbf{I}_i\mathbf{v}_i - {}^{i,-1}\mathbf{X}^{\mathrm{T}} \cdot {}^{-1}\mathbf{f}_i^a$
end for
#compute inverse dynamics
for $i = N_B - 1$ **to** 0 **do**
  $\tau_i = \mathbf{\Phi}_i^{\mathrm{T}} \cdot \mathbf{f}_i$ # joint i force (torque)
  if $p(i) \neq -1$ **then**
    $\mathbf{f}_{p(i)}$ += ${}^{i,p(i)}\mathbf{X}^{\mathrm{T}}\,\mathbf{f}_i$
  end if
end for
return τ
end function

Algorithm 2 Composite-rigid-body algorithm (acc. to Featherstone). The symbol '#' represents comments.

function MassMatrix($\boldsymbol{\Phi}_i$, ${}^{i,p(i)}\mathbf{X}$, $\mathbf{I}_i$)

mass matrix:
$\mathbf{M}_0 = \mathbf{0}$
for $i = 0$ **to** $N_B - 1$ **do**
  # initialise 6D inertia tensors:
  $\mathbf{I}_i^C = \mathbf{I}_i$
end for
#recursively update inertias
for $i = N_B - 1$ **to** 0 **do**
  # project inertia into motion subspace:
  $\mathbf{F} = \mathbf{I}_i^C \, \boldsymbol{\Phi}_i$
  $\mathbf{M}_{ii} = \boldsymbol{\Phi}_i^T \, \mathbf{F}$
  if $p(i) \neq -1$ **then**
    $\mathbf{I}_{p(i)}^C$ += ${}^{i,p(i)}\mathbf{X}^T \cdot \mathbf{I}_i^C \cdot {}^{i,p(i)}\mathbf{X}$
  end if
  $j = i$
  # compute mass matrix terms:
  while $p(j) \neq -1$ **do**
    $\mathbf{F} = {}^{j,p(j)}\mathbf{X}^T \cdot \mathbf{F}$
    $j = p(j)$
    $\mathbf{M}_{ij} = \mathbf{F}^T \, \boldsymbol{\Phi}_i$
    $\mathbf{M}_{ji} = \mathbf{M}_{ij}$
  end while
end for
return $\mathbf{M}$

end function

More detailed equations will be added later on. Follow exactly the description (and coordinate systems) of the object parameters, especially for describing the kinematic chain as well as the inertial parameters.

Userfunction: `forceUserFunction(mbs, t, itemNumber, q, q_t)`

A user function, which computes a force vector applied to the joint coordinates depending on current time and states of object. Note that itemNumber represents the index of the ObjectKinematicTree object in mbs, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs to which object belongs
`t`	Real	current time in mbs
`itemNumber`	Index	integer number i_N of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
`q`	Vector $\in \mathbb{R}^n$	object coordinates (e.g., nodal displacement coordinates) in current configuration, without reference values
`q_t`	Vector $\in \mathbb{R}^n$	object velocity coordinates (time derivative of `q`) in current configuration
`return value`	Vector $\in \mathbb{R}^n$	returns force vector for object

---

8.3.2.7 MINI EXAMPLE for ObjectKinematicTree

```
#build 1R mechanism (pendulum)
L = 1 #length of link
RBinertia = InertiaCuboid(1000, [L,0.1*L,0.1*L])
inertiaLinkCOM = RBinertia.InertiaCOM() #KinematicTree requires COM inertia
linkCOM = np.array([0.5*L,0.,0.]) #if COM=0, gravity does not act on pendulum!

offsetsList = exu.Vector3DList([[0,0,0]])
rotList = exu.Matrix3DList([np.eye(3)])
linkCOMs=exu.Vector3DList([linkCOM])
linkInertiasCOM=exu.Matrix3DList([inertiaLinkCOM])


nGeneric = mbs.AddNode(NodeGenericODE2(referenceCoordinates=[0.],initialCoordinates
=[0.],
                                       initialCoordinates_t=[0.],numberOfODE2Coordinates
=1))
```

```
oKT = mbs.AddObject(ObjectKinematicTree(nodeNumber=nGeneric, jointTypes=[exu.JointType.
RevoluteZ], linkParents=[-1],
                                    jointTransformations=rotList, jointOffsets=offsetsList
, linkInertiasCOM=linkInertiasCOM,
                                    linkCOMs=linkCOMs, linkMasses=[RBinertia.mass],
                                    baseOffset = [0.5,0.,0.], gravity=[0.,-9.81,0.]))

#assemble and solve system for default parameters
mbs.Assemble()

simulationSettings = exu.SimulationSettings() #takes currently set values or default
values
simulationSettings.timeIntegration.numberOfSteps = 1000 #gives very accurate results
mbs.SolveDynamic(simulationSettings , solverType=exu.DynamicSolverType.RK67) #highly
accurate!

#check final value of angle:
q0 = mbs.GetNodeOutput(nGeneric, exu.OutputVariableType.Coordinates)
#exu.Print(q0)
exudynTestGlobals.testResult = q0 #-3.134018551808591; RigidBody2D with 2e6 time steps
gives: -3.134018551809384
```

For examples on ObjectKinematicTree see Relevant Examples and TestModels with weblink:

- kinematicTreeAndMBS.py (Examples/)
- reinforcementLearningRobot.py (Examples/)
- stiffFlyballGovernorKT.py (Examples/)
- kinematicTreeTest.py (TestModels/)
- createKinematicTreeTest.py (TestModels/)

8.3.3 ObjectFFRF

This object is used to represent equations modelled by the FFRF. It contains a RigidBodyNode (always node 0) and a list of other nodes representing the finite element nodes used in the FFRF. Note that temporary matrices and vectors are subject of change in future. Usually you SHOULD NOT USE THIS OBJECT - use the much more efficient ObjectFFRFreducedOrder object with modal reduction instead.

Authors: Gerstmayr Johannes, Zwölfer Andreas

Additional information for ObjectFFRF:

- This `Object` has/provides the following types = `Body`, `MultiNoded`, `SuperElement`
- Requested `Node` type: read detailed information of item

The item **ObjectFFRF** with type = 'FFRF' has the following parameters:

Name	type	size	default value	description
name	String		"	objects's unique name
nodeNumbers	ArrayNodeIndex		[]	node numbers which provide the coordinates for the object (consecutively as provided in this list); the (n_{nf} + 1) nodes represent the nodes of the FE mesh (except for node 0); the global nodal position needs to be reconstructed from the rigid-body motion of the reference frame
massMatrixFF	PyMatrixContainer		PyMatrixContainer[]	body-fixed and ONLY flexible coordinates part of mass matrix of object given in Python numpy format (sparse (CSR) or dense, converted to sparse matrix); internally data is stored in triplet format
stiffnessMatrixFF	PyMatrixContainer		PyMatrixContainer[]	body-fixed and ONLY flexible coordinates part of stiffness matrix of object in Python numpy format (sparse (CSR) or dense, converted to sparse matrix); internally data is stored in triplet format
dampingMatrixFF	PyMatrixContainer		PyMatrixContainer[]	body-fixed and ONLY flexible coordinates part of damping matrix of object in Python numpy format (sparse (CSR) or dense, converted to sparse matrix); internally data is stored in triplet format

forceVector	NumpyVector		[]	generalized, force vector added to RHS; the rigid body part $\mathbf{f}_r$ is directly applied to rigid body coordinates while the flexible part $\mathbf{f}_f$ is transformed from global to local coordinates; note that this force vector only allows to add gravity forces for bodies with COM at the origin of the reference frame
forceUserFunction	PyFunctionVectorMbsScalarIndex2Vector		0	A Python user function which computes the generalized user force vector for the ODE2 equations; note the different coordinate systems for rigid body and flexible part; The function args are mbs, time, objectNumber, coordinates q (without reference values) and coordinate velocities q_t; see description below
massMatrixUserFunction	PyFunctionMatrixMbsScalarIndex2Vector		0	A Python user function which computes the TOTAL mass matrix (including reference node) and adds the local constant mass matrix; note the different coordinate systems as described in the FFRF mass matrix; see description below
computeFFRFterms	Bool		True	flag decides whether the standard FFRF terms are computed; use this flag for user-defined definition of FFRF terms in mass matrix and quadratic velocity vector
coordinateIndexPerNode	ArrayIndex		[]	this list contains the local coordinate index for every node, which is needed, e.g., for markers; the list is generated automatically every time parameters have been changed
objectIsInitialized	Bool		False	ALWAYS set to False! flag used to correctly initialize all FFRF matrices; as soon as this flag is False, internal (constant) FFRF matrices are recomputed during Assemble()
physicsMass	UReal		0.	total mass [SI:kg] of FFRF object, auto-computed from mass matrix ${}^b\mathbf{M}$
physicsInertia	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	inertia tensor [SI:kgm^2] of rigid body w.r.t. to the reference point of the body, auto-computed from the mass matrix ${}^b\mathbf{M}$
physicsCenterOfMass	Vector3D	3	[0.,0.,0.]	local position of center of mass (COM); auto-computed from mass matrix ${}^b\mathbf{M}$
PHItTM	NumpyMatrix		Matrix[]	projector matrix; may be removed in future
referencePositions	NumpyVector		[]	vector containing the reference positions of all flexible nodes
tempVector	NumpyVector		[]	temporary vector
tempCoordinates	NumpyVector		[]	temporary vector containing coordinates

tempCoordinates_t	NumpyVector		[]	temporary vector containing velocity coordinates
tempRefPosSkew	NumpyMatrix		Matrix[]	temporary matrix with skew symmetric local (deformed) node positions
tempVelSkew	NumpyMatrix		Matrix[]	temporary matrix with skew symmetric local node velocities
visualization	VObjectFFRF			parameters for visualization of item

The item VObjectFFRF has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown; use visualizationSettings.bodies.deformationScaleFactor to draw scaled (local) deformations; the reference frame node is shown with additional letters RF
color	Float4	4	[-1.,-1.,-1.,-1.]	RGBA color for object; 4th value is alpha-transparency; R=-1.f means, that default color is used
triangleMesh	NumpyMatrixI		MatrixI[]	a matrix, containg node number triples in every row, referring to the node numbers of the GenericODE2 object; the mesh uses the nodes to visualize the underlying object; contour plot colors are still computed in the local frame!
showNodes	Bool		False	set true, nodes are drawn uniquely via the mesh, eventually using the floating reference frame, even in the visualization of the node is show=False; node numbers are shown with indicator 'NF'

8.3.3.1 DESCRIPTION of ObjectFFRF:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
nodeNumbers	$\mathbf{n}_\mathrm{f} = [n_0, \ldots, n_{n_\mathrm{nf}}]^\mathrm{T}$	
massMatrixFF	${}^b\mathbf{M} \in \mathbb{R}^{n_\mathrm{f} \times n_\mathrm{f}}$	
stiffnessMatrixFF	${}^b\mathbf{K} \in \mathbb{R}^{n_\mathrm{f} \times n_\mathrm{f}}$	
dampingMatrixFF	${}^b\mathbf{D} \in \mathbb{R}^{n_\mathrm{f} \times n_\mathrm{f}}$	

forceVector	${}^0\mathbf{f} = [{}^0\mathbf{f}_r, {}^0\mathbf{f}_f]^T \in \mathbb{R}^{n_c}$	
forceUserFunction	$\mathbf{f}_{user} = [{}^0\mathbf{f}_{r,user}, {}^b\mathbf{f}_{f,user}]^T \in \mathbb{R}^{n_c}$	
massMatrixUserFunction	$\mathbf{M}_{user} \in \mathbb{R}^{n_c \times n_c}$	
physicsMass	m	
physicsInertia	$J_r \in \mathbb{R}^{3\times3}$	
physicsCenterOfMass	${}^b\mathbf{b}_{COM}$	
PHItTM	$\mathbf{\Phi}_t^T \in \mathbb{R}^{n_f \times 3}$	
referencePositions	$\mathbf{x}_{ref} \in \mathbb{R}^{n_f}$	
tempVector	$\mathbf{v}_{temp} \in \mathbb{R}^{n_f}$	
tempCoordinates	$\mathbf{c}_{temp} \in \mathbb{R}^{n_f}$	
tempCoordinates_t	$\dot{\mathbf{c}}_{temp} \in \mathbb{R}^{n_f}$	
tempRefPosSkew	$\tilde{\mathbf{p}}_f \in \mathbb{R}^{n_f \times 3}$	
tempVelSkew	$\tilde{\dot{\mathbf{c}}}_f \in \mathbb{R}^{n_f \times 3}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Coordinates		all ODE2 coordinates
Coordinates_t		all ODE2 velocity coordinates
Coordinates_tt		all ODE2 acceleration coordinates
Force		generalized forces for all coordinates (residual of all forces except mass*accleration; corresponds to ComputeODE2LHS)

8.3.3.2 Additional output variables for superelement node access

Functions like `GetObjectOutputSuperElement(...)`, see Section 6.5.4, or `SensorSuperElement`, see Section 6.5.7, directly access special output variables (`OutputVariableType`) of the mesh nodes n_i of the superelement. Additionally, the contour drawing of the object can make use the `OutputVariableType` of the meshnodes.

8.3.3.3 Super element output variables

super element output variables	symbol	description
Position	${}^0\mathbf{p}_{\text{config}}(n_i) = {}^0\mathbf{r}_{\text{config}} + {}^{0b}\mathbf{A}_{\text{config}} {}^b\mathbf{p}_{\text{config}}(n_i)$	global position of mesh node n_i including rigid body motion and flexible deformation
Displacement	${}^0\mathbf{c}_{\text{config}}(n_i) = {}^0\mathbf{p}_{\text{config}}(n_i) - {}^0\mathbf{p}_{\text{ref}}(n_i)$	global displacement of mesh node n_i including rigid body motion and flexible deformation
Velocity	${}^0\mathbf{v}_{\text{config}}(n_i) = {}^0\dot{\mathbf{r}}_{\text{config}} + {}^{0b}\mathbf{A}_{\text{config}}({}^b\dot{\mathbf{q}}_{\text{f config}}(n_i) + {}^b\boldsymbol{\omega}_{\text{config}} \times {}^b\mathbf{p}_{\text{config}}(n_i))$	global velocity of mesh node n_i including rigid body motion and flexible deformation

Acceleration	${}^0\mathbf{a}_{\text{config}}(n_i) = {}^0\ddot{\mathbf{r}}_{\text{config config}}$ $+ {}^{0b}\mathbf{A}_{\text{config}}\, {}^b\ddot{\mathbf{q}}_{\text{f config}}(n_i)$ $+2\, {}^0\boldsymbol{\omega}_{\text{config}} \times {}^{0b}\mathbf{A}_{\text{config}}\, {}^b\dot{\mathbf{q}}_{\text{f config}}(n_i)$ $+ {}^0\boldsymbol{\alpha}_{\text{config}} \times {}^0\mathbf{p}_{\text{config}}(n_i)$ $+ {}^0\boldsymbol{\omega}_{\text{config}} \times ({}^0\boldsymbol{\omega}_{\text{config}} \times {}^0\mathbf{p}_{\text{config}}(n_i))$	global acceleration of mesh node n_i including rigid body motion and flexible deformation; note that ${}^0\mathbf{p}_{\text{config}}(n_i) = {}^{0b}\mathbf{A}\, {}^b\mathbf{p}_{\text{config}}(n_i)$
DisplacementLocal	${}^b\mathbf{d}_{\text{config}}(n_i) = {}^b\mathbf{p}_{\text{config}}(n_i) - {}^b\mathbf{x}_{\text{ref}}(n_i)$	local displacement of mesh node n_i, representing the flexible deformation within the body frame; note that ${}^0\mathbf{u}_{\text{config}} \neq {}^{0b}\mathbf{A}\, {}^b\mathbf{d}_{\text{config}}$!
VelocityLocal	${}^b\dot{\mathbf{q}}_{\text{f config}}(n_i)$	local velocity of mesh node n_i, representing the rate of flexible deformation within the body frame

8.3.3.4 Definition of quantities

intermediate variables	symbol	description
object coordinates	$\mathbf{q} = [\mathbf{q}_\text{t}^\text{T},\ \mathbf{q}_\text{r}^\text{T},\ \mathbf{q}_\text{f}^\text{T}]^\text{T}$	object coordinates
rigid body coordinates	$\mathbf{q}_\text{rigid} = [\mathbf{q}_\text{t}^\text{T},\ \mathbf{q}_\text{r}^\text{T}]^\text{T} = [q_0,\ q_1,\ q_2,\ \psi_0,\ \psi_1,\ \psi_2,\ \psi_3]^\text{T}$	rigid body coordinates in case of Euler parameters
reference frame (rigid body) position	${}^0\mathbf{r}_\text{config} = {}^0\mathbf{q}_\text{t,config} + {}^0\mathbf{q}_\text{t,ref}$	global position of underlying rigid body node n_0 which defines the reference frame origin
reference frame (rigid body) orientation	${}^{0b}\mathbf{A}(\boldsymbol{\theta})_\text{config}$	transformation matrix for transformation of local (reference frame) to global coordinates, given by underlying rigid body node n_0
local nodal position	${}^b\mathbf{p}^{(i)} = {}^b\mathbf{x}^{(i)}_\text{ref} + {}^b\mathbf{q}_\text{f}^{(i)}$	vector of body-fixed (local) position of node (i), including flexible part
local nodal positions	${}^b\mathbf{p} = {}^b\mathbf{x}_\text{ref} + {}^b\mathbf{q}_\text{f}$	vector of all body-fixed (local) nodal positions including flexible part
rotation coordinates	$\boldsymbol{\theta}_\text{cur} = [\psi_0,\ \psi_1,\ \psi_2,\ \psi_3]^\text{T}_\text{ref} + [\psi_0,\ \psi_1,\ \psi_2,\ \psi_3]^\text{T}_\text{cur}$	rigid body coordinates in case of Euler parameters
flexible coordinates	${}^b\mathbf{q}_\text{f}$	flexible, body-fixed coordinates
transformation of flexible coordinates	${}^{0b}\mathbf{A}_{bd} = \text{diag}([{}^{0b}\mathbf{A},\ \ldots,\ {}^{0b}\mathbf{A})$	block diagonal transformation matrix, which transforms all flexible coordinates from local to global coordinates

The derivations follow Zwölfer and Gerstmayr [67] with only small modifications in the notation.

8.3.3.5 Nodal coordinates

Consider an object with $n = 1 + n_\text{nf}$ nodes, n_nf being the number of 'flexible' nodes and one additional node is the rigid body node for the reference frame. The list if node numbers is $[n_0,\ \ldots,\ n_{n_\text{nf}}]$ and the according numbers of nodal coordinates are $[n_{c_0},\ \ldots,\ n_{c_n}]$, where n_0 denotes the rigid body node. This gives n_c total nodal coordinates,

$$n_c = \sum_{i=0}^{n_\text{nf}} n_{c_i}\ , \tag{8.65}$$

whereof the number of flexible coordinates is

$$n_{\mathrm{f}} = 3 \cdot n_{\mathrm{nf}} \ . \tag{8.66}$$

The total number of equations (=coordinates) of the object is n_c. The first node n_0 represents the rigid body motion of the underlying reference frame with $n_{c_{\mathrm{r}}} = n_{c_0}$ coordinates [1].

8.3.3.6 Kinematics

We assume a finite element mesh with The kinematics of the FFRF is based on a splitting of translational ($\mathbf{c}_t \in \mathbb{R}^{n_{\mathrm{f}}}$), rotational ($\mathbf{c}_{\mathrm{r}} \in \mathbb{R}^{n_{\mathrm{f}}}$) and flexible ($\mathbf{c}_{\mathrm{f}} \in \mathbb{R}^{n_{\mathrm{f}}}$) nodal displacements,

$$^0\mathbf{c} = {}^0\mathbf{c}_{\mathrm{t}} + {}^0\mathbf{c}_{\mathrm{r}} + {}^0\mathbf{c}_{\mathrm{f}} \ . \tag{8.67}$$

which are written in global coordinates in Eq. (8.67) but will be transformed to other coordinates later on.

In the present formulation of `ObjectFFRF`, we use the following set of object coordinates (unknowns)

$$\mathbf{q} = \begin{bmatrix} {}^0\mathbf{q}_{\mathrm{t}}^{\mathrm{T}} & \boldsymbol{\theta}^{\mathrm{T}} & {}^b\mathbf{q}_{\mathrm{f}}^{\mathrm{T}} \end{bmatrix}^{\mathrm{T}} \in \mathbb{R}^{n_c} \tag{8.68}$$

with ${}^0\mathbf{q}_{\mathrm{t}} \in \mathbb{R}^3$, $\boldsymbol{\theta} \in \mathbb{R}^4$ and ${}^b\mathbf{q}_{\mathrm{f}} \in \mathbb{R}^{n_{\mathrm{f}}}$. Note that parts of the coordinates $\mathbf{q}$ can be already interpreted in specific coordinate systems, which is therefore added.

With the relations

$$\begin{aligned}
\boldsymbol{\Phi}_{\mathrm{t}} &= [\mathbf{I}_{3\times3}, \ \ldots, \ \mathbf{I}_{3\times3}]^{\mathrm{T}} \in \mathbb{R}^{n_{\mathrm{f}}\times3} \ , && (8.69)\\
{}^0\mathbf{c}_{\mathrm{t}} &= \boldsymbol{\Phi}_{\mathrm{t}}\, {}^0\mathbf{q}_{\mathrm{t}} \ , && (8.70)\\
{}^0\mathbf{c}_{\mathrm{r}} &= \left({}^{0b}\mathbf{A}_{bd} - \mathbf{I}_{bd}\right) {}^b\mathbf{x}_{\mathrm{ref}} \ , && (8.71)\\
{}^0\mathbf{c}_{\mathrm{f}} &= {}^{0b}\mathbf{A}_{bd}\, {}^b\mathbf{q}_{\mathrm{f}} \ , \text{and} && (8.72)\\
\mathbf{I}_{bd} &= \mathrm{diag}(\mathbf{I}_{3\times3}, \ \ldots, \ \mathbf{I}_{3\times3}) \in \mathbb{R}^{n_{\mathrm{f}}\times n_{\mathrm{f}}} \ , && (8.73)
\end{aligned}$$

we obtain the total relation of (global) nodal displacements to the object coordinates

$$^0\mathbf{c} = \boldsymbol{\Phi}_{\mathrm{t}}\, {}^0\mathbf{q}_{\mathrm{t}} + \left({}^{0b}\mathbf{A}_{bd} - \mathbf{I}_{bd}\right) {}^b\mathbf{x}_{\mathrm{ref}} + {}^{0b}\mathbf{A}_{bd}\, {}^b\mathbf{q}_{\mathrm{f}} \ . \tag{8.74}$$

On velocity level, we have

$$^0\dot{\mathbf{c}} = \mathbf{L}\dot{\mathbf{q}} \ , \tag{8.75}$$

with the matrix $\mathbf{L} \in \mathbb{R}^{n_{\mathrm{f}}\times n_c}$

$$\mathbf{L} = \left[\boldsymbol{\Phi}_{\mathrm{t}}, \ -{}^{0b}\mathbf{A}_{bd}\, {}^b\tilde{\mathbf{p}}\, {}^b\mathbf{G}, \ {}^{0b}\mathbf{A}_{bd}\right] \tag{8.76}$$

with the rotation parameters specific matrix ${}^b\mathbf{G}$, implicitly defined in the rigid body node by the relation ${}^b\boldsymbol{\omega} = {}^b\mathbf{G}\,\dot{\boldsymbol{\theta}}$ and the body-fixed nodal position vector (for node i)

$$^b\mathbf{p} = {}^b\mathbf{x}_{\mathrm{ref}} + {}^b\mathbf{q}_{\mathrm{f}} \ , \quad {}^b\mathbf{p}^{(i)} = {}^b\mathbf{x}_{\mathrm{ref}}^{(i)} + {}^b\mathbf{q}_{\mathrm{f},i}^{(i)} \tag{8.77}$$

[1] e.g., $n_{c_{\mathrm{r}}} = 6$ coordinates for Euler angles and $n_{c_{\mathrm{r}}} = 7$ coordinates in case of Euler parameters; currently only the Euler parameter case is implemented.

and the special tilde matrix for vectors $\mathbf{p} \in \mathbb{R}^{3n_{\text{nf}}}$,

$$ {}^{b}\tilde{\mathbf{p}} = \begin{bmatrix} {}^{b}\tilde{\mathbf{p}}^{(i)} \\ \vdots \\ {}^{b}\tilde{\mathbf{p}}^{(i)} \end{bmatrix} \in \mathbb{R}^{3n_{\text{nf}} \times 3} . \tag{8.78} $$

with the tilde operator for a $\mathbf{p}^{(i)} \in \mathbb{R}^3$ defined in the common notations section.

8.3.3.7 Equations of motion

We use the Lagrange equations extended for constraint $\mathbf{g}$,

$$ \frac{d}{dt}\left(\frac{\partial T}{\partial \dot{\mathbf{q}}^{\mathrm{T}}}\right) - \frac{\partial T}{\partial \mathbf{q}^{\mathrm{T}}} + \frac{\partial V}{\partial \mathbf{q}^{\mathrm{T}}} + \frac{\partial \boldsymbol{\lambda}^{\mathrm{T}}\mathbf{g}}{\partial \mathbf{q}^{\mathrm{T}}} = \frac{\partial W}{\partial \mathbf{q}^{\mathrm{T}}} \tag{8.79} $$

with the quantities

$$ T({}^{0}\dot{\mathbf{c}}(\mathbf{q},\dot{\mathbf{q}})) = \frac{1}{2}\,{}^{0}\dot{\mathbf{c}}^{\mathrm{T}}\,{}^{0}\mathbf{M}\,{}^{0}\dot{\mathbf{c}} = \frac{1}{2}\,{}^{0}\dot{\mathbf{c}}^{\mathrm{T}}\,{}^{0b}\mathbf{A}_{bd}\,{}^{b}\mathbf{M}\,{}^{0b}\mathbf{A}_{bd}{}^{\mathrm{T}}\,{}^{0}\dot{\mathbf{c}} = \frac{1}{2}\,{}^{0}\dot{\mathbf{c}}^{\mathrm{T}}\,{}^{b}\mathbf{M}\,{}^{0}\dot{\mathbf{c}} \tag{8.80} $$

$$ V({}^{0}\mathbf{q}_{\mathrm{f}}) = \frac{1}{2}\,{}^{b}\mathbf{q}_{\mathrm{f}}^{\mathrm{T}}\,{}^{b}\mathbf{K}\,{}^{b}\mathbf{q}_{\mathrm{f}} \tag{8.81} $$

$$ \delta W({}^{0}\mathbf{c}(\mathbf{q}),t) = {}^{b}\delta\mathbf{c}^{\mathrm{T}}\,\mathbf{f} \tag{8.82} $$

$$ \mathbf{g}(\mathbf{q},t) = \mathbf{0} \tag{8.83} $$

$$ \tag{8.84} $$

Note that ${}^{b}\mathbf{M}$ and ${}^{b}\mathbf{K}$ are the conventional finite element mass an stiffness matrices defined in the body frame.

Elementary differentiation rules of the Lagrange equations lead to

$$ \mathbf{L}^{\mathrm{T}}\mathbf{M}\mathbf{L}\ddot{\mathbf{q}} + \mathbf{L}^{\mathrm{T}}\mathbf{M}\dot{\mathbf{L}}\dot{\mathbf{q}} + \hat{\mathbf{K}}\mathbf{q} + \frac{\partial \mathbf{g}}{\partial \mathbf{q}^{\mathrm{T}}}\boldsymbol{\lambda} = \mathbf{L}^{\mathrm{T}}\mathbf{f} \tag{8.85} $$

with $\mathbf{M} = {}^{b}\mathbf{M}$ and $\hat{\mathbf{K}}$ becoming obvious in Eq. (8.86). Note that Eq. (8.85) is given in global coordinates for the translational part, in terms of rotation parameters for the rotation part and in body-fixed coordinates for the flexible part of the equations.

In case that `computeFFRFterms = True`, the equations 8.85 can be transformed into the equations of motion,

$$ \left(\mathbf{M}_{user}(mbs,t,i_N,\mathbf{q},\dot{\mathbf{q}}) + \begin{bmatrix} \mathbf{M}_{\mathrm{tt}} & \mathbf{M}_{\mathrm{tr}} & \mathbf{M}_{\mathrm{tf}} \\ & \mathbf{M}_{\mathrm{rr}} & \mathbf{M}_{\mathrm{rf}} \\ \text{sym.} & & {}^{b}\mathbf{M} \end{bmatrix}\right)\ddot{\mathbf{q}} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & {}^{b}\mathbf{D} \end{bmatrix}\dot{\mathbf{q}} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & {}^{b}\mathbf{K} \end{bmatrix}\mathbf{q} = \mathbf{f}_v(\mathbf{q},\dot{\mathbf{q}}) + \begin{bmatrix} \mathbf{f}_{\mathrm{r}} \\ {}^{0b}\mathbf{A}_{bd}^{\mathrm{T}}\,\mathbf{f}_{\mathrm{f}} \end{bmatrix} + \mathbf{f}_{user}(mbs,t,i_N,\mathbf{q} \tag{8.86} $$

in which `iN` represents the itemNumber (=objectNumber of ObjectFFRF in mbs) in the user function.

The mass terms are given as

$$\mathbf{M}_{tt} = \mathbf{\Phi}_t^T {}^b\mathbf{M}\,\mathbf{\Phi}_t, \tag{8.87}$$

$$\mathbf{M}_{tr} = -{}^{0b}\mathbf{A}\,\mathbf{\Phi}_t^T {}^b\mathbf{M}\,{}^b\tilde{\mathbf{p}}\,{}^b\mathbf{G}, \tag{8.88}$$

$$\mathbf{M}_{tf} = {}^{0b}\mathbf{A}\,\mathbf{\Phi}_t^T {}^b\mathbf{M}, \tag{8.89}$$

$$\mathbf{M}_{rr} = {}^b\mathbf{G}^T {}^b\tilde{\mathbf{p}}^T {}^b\mathbf{M}\,{}^b\tilde{\mathbf{p}}\,{}^b\mathbf{G}, \tag{8.90}$$

$$\mathbf{M}_{rf} = -{}^b\mathbf{G}^T {}^b\tilde{\mathbf{p}}^T {}^b\mathbf{M}\ . \tag{8.91}$$

In case that `computeFFRFterms = False`, the mass terms $\mathbf{M}_{tt}, \mathbf{M}_{tr}, \mathbf{M}_{tf}, \mathbf{M}_{rr}, \mathbf{M}_{rf}, {}^b\mathbf{M}$ in Eq. (8.86) are set to zero (and not computed) and the quadratic velocity vector $\mathbf{f}_v = \mathbf{0}$. Note that the user functions $\mathbf{f}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}})$ and $\mathbf{M}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}})$ may be empty (=0). The detailed equations of motion for this element can be found in [66].

The quadratic velocity vector follows as

$$\mathbf{f}_v(\mathbf{q}, \dot{\mathbf{q}}) = \begin{bmatrix} -{}^{0b}\mathbf{A}\,\mathbf{\Phi}_t^T {}^b\mathbf{M}\left({}^b\tilde{\boldsymbol{\omega}}_{bd}\,{}^b\tilde{\boldsymbol{\omega}}_{bd}\,{}^b\mathbf{p} + 2\,{}^b\tilde{\boldsymbol{\omega}}_{bd}\,{}^b\dot{\mathbf{q}}_f - {}^b\tilde{\mathbf{p}}\,{}^b\dot{\mathbf{G}}\,\dot{\boldsymbol{\theta}}\right) \\ {}^b\mathbf{G}^T {}^b\tilde{\mathbf{p}}^T {}^b\mathbf{M}\left({}^b\tilde{\boldsymbol{\omega}}_{bd}\,{}^b\tilde{\boldsymbol{\omega}}_{bd}\,{}^b\mathbf{p} + 2\,{}^b\tilde{\boldsymbol{\omega}}_{bd}\,{}^b\dot{\mathbf{q}}_f - {}^b\tilde{\mathbf{p}}\,{}^b\dot{\mathbf{G}}\,\dot{\boldsymbol{\theta}}\right) \\ -{}^b\mathbf{M}\left({}^b\tilde{\boldsymbol{\omega}}_{bd}\,{}^b\tilde{\boldsymbol{\omega}}_{bd}\,{}^b\mathbf{p} + 2\,{}^b\tilde{\boldsymbol{\omega}}_{bd}\,{}^b\dot{\mathbf{q}}_f - {}^b\tilde{\mathbf{p}}\,{}^b\dot{\mathbf{G}}\,\dot{\boldsymbol{\theta}}\right) \end{bmatrix} \tag{8.92}$$

with the special matrix

$${}^b\tilde{\boldsymbol{\omega}}_{bd} = \mathrm{diag}\left({}^b\tilde{\boldsymbol{\omega}}_{bd}, \ldots, {}^b\tilde{\boldsymbol{\omega}}_{bd}\right) \in \mathbb{R}^{n_f \times n_f} \tag{8.93}$$

CoordinateLoads are added for each ODE2 coordinate on the RHS of the latter equation.
If the rigid body node is using Euler parameters $\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2, \theta_3]^T$, an **additional constraint** (constraint nr. 0) is added automatically for the Euler parameter norm, reading

$$1 - \sum_{i=0}^{3} \theta_i^2 = 0. \tag{8.94}$$

In order to suppress the rigid body motion of the mesh nodes, you should apply a ObjectConnectorCoordinateVector object with the following constraint equations which impose constraints of a so-called Tisserand frame, giving 3 constraints for the position of the center of mass

$$\Phi_t^T {}^b\mathbf{M}\,\mathbf{q}_f = 0 \tag{8.95}$$

and 3 constraints for the rotation,

$$\tilde{\mathbf{x}}_f^T {}^b\mathbf{M}\,\mathbf{q}_f = 0 \tag{8.96}$$

Userfunction: `forceUserFunction(mbs, t, itemNumber, q, q_t)`

A user function, which computes a force vector depending on current time and states of object. Can be used to create any kind of mechanical system by using the object states.

arguments / return	type or size	description

`mbs`	MainSystem	provides MainSystem mbs to which object belongs
`t`	Real	current time in mbs
`itemNumber`	Index	integer number of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
`q`	Vector $\in \mathbb{R}_c^n$	object coordinates (nodal displacement coordinates of rigid body and mesh nodes) in current configuration, without reference values
`q_t`	Vector $\in \mathbb{R}_c^n$	object velocity coordinates (time derivative of `q`) in current configuration
`return value`	Vector $\in \mathbb{R}^{n_c}$	returns force vector for object

---

Userfunction: `massMatrixUserFunction(mbs, t, itemNumber, q, q_t)`

A user function, which computes a mass matrix depending on current time and states of object. Can be used to create any kind of mechanical system by using the object states.

arguments / return	**type or size**	**description**
`mbs`	MainSystem	provides MainSystem mbs to which object belongs
`t`	Real	current time in mbs
`itemNumber`	Index	integer number of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
`q`	Vector $\in \mathbb{R}_c^n$	object coordinates (nodal displacement coordinates of rigid body and mesh nodes) in current configuration, without reference values
`q_t`	Vector $\in \mathbb{R}_c^n$	object velocity coordinates (time derivative of `q`) in current configuration
`return value`	NumpyMatrix $\in \mathbb{R}^{n_c \times n_c}$	returns mass matrix for object

---

For examples on ObjectFFRF see Relevant Examples and TestModels with weblink:

- `objectFFRFTest.py` (TestModels/)
- `objectFFRFTest2.py` (TestModels/)

8.3.4 ObjectFFRFreducedOrder

This object is used to represent modally reduced flexible bodies using the FFRF and the CMS. It can be used to model real-life mechanical systems imported from finite element codes or Python tools such as NETGEN/NGsolve, see the `FEMinterface` in Section 7.7.8. It contains a RigidBodyNode (always node 0) and a NodeGenericODE2 representing the modal coordinates. Currently, equations must be defined within user functions, which are available in the FEM module, see class `ObjectFFRFreducedOrderInterface`, especially the user functions `UFmassFFRFreducedOrder` and `UFforceFFRFreducedOrder`, Section 7.7.6.

Authors: Gerstmayr Johannes, Zwölfer Andreas

Additional information for ObjectFFRFreducedOrder:

- This `Object` has/provides the following types = `Body`, `MultiNoded`, `SuperElement`
- Requested `Node` type: read detailed information of item
- **Short name** for Python = `CMSobject`
- **Short name** for Python visualization object = `VCMSobject`

The item **ObjectFFRFreducedOrder** with type = 'FFRFreducedOrder' has the following parameters:

Name	type	size	default value	description
name	String		''	objects's unique name
nodeNumbers	ArrayNodeIndex		[]	node numbers of rigid body node and NodeGenericODE2 for modal coordinates; the global nodal position needs to be reconstructed from the rigid-body motion of the reference frame, the modal coordinates and the mode basis
massMatrixReduced	PyMatrixContainer		PyMatrixContainer[]	body-fixed and ONLY flexible coordinates part of reduced mass matrix; provided as MatrixContainer(sparse/dense matrix)
stiffnessMatrixReduced	PyMatrixContainer		PyMatrixContainer[]	body-fixed and ONLY flexible coordinates part of reduced stiffness matrix; provided as MatrixContainer(sparse/dense matrix)
dampingMatrixReduced	PyMatrixContainer		PyMatrixContainer[]	body-fixed and ONLY flexible coordinates part of reduced damping matrix; provided as MatrixContainer(sparse/dense matrix)

forceUserFunction	PyFunctionVectorMbsScalarIndex2Vector		0	A Python user function which computes the generalized user force vector for the ODE2 equations; see description below
massMatrixUserFunction	PyFunctionMatrixMbsScalarIndex2Vector		0	A Python user function which computes the TOTAL mass matrix (including reference node) and adds the local constant mass matrix; see description below
computeFFRFterms	Bool		True	flag decides whether the standard FFRF/CMS terms are computed; use this flag for user-defined definition of FFRF terms in mass matrix and quadratic velocity vector
modeBasis	NumpyMatrix		Matrix[]	mode basis, which transforms reduced coordinates to (full) nodal coordinates, written as a single vector $[u_{x,n_0}, u_{y,n_0}, u_{z,n_0}, \ldots, u_{x,n_n}, u_{y,n_n}, u_{z,n_n}]^T$
outputVariableModeBasis	NumpyMatrix		Matrix[]	mode basis, which transforms reduced coordinates to output variables per mode and per node; s_{OV} is the size of the output variable, e.g., 6 for stress modes (S_{xx}, ..., S_{xy})
outputVariableTypeModeBasis	OutputVariableType		OutputVariableType::_None	this must be the output variable type of the outputVariableModeBasis, e.g. exu.OutputVariableType.Stress
referencePositions	NumpyVector		[]	vector containing the reference positions of all flexible nodes, needed for graphics
objectIsInitialized	Bool		False	ALWAYS set to False! flag used to correctly initialize all FFRF matrices; as soon as this flag is False, some internal (constant) FFRF matrices are recomputed during Assemble()
physicsMass	UReal		0.	total mass [SI:kg] of FFRFreducedOrder object
physicsInertia	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	inertia tensor [SI:kgm^2] of rigid body w.r.t. to the reference point of the body
physicsCenterOfMass	Vector3D	3	[0.,0.,0.]	local position of center of mass (COM)
mPsiTildePsi	NumpyMatrix		Matrix[]	special FFRFreducedOrder matrix, computed in ObjectFFRFreducedOrderInterface
mPsiTildePsiTilde	NumpyMatrix		Matrix[]	special FFRFreducedOrder matrix, computed in ObjectFFRFreducedOrderInterface
mPhitTPsi	NumpyMatrix		Matrix[]	special FFRFreducedOrder matrix, computed in ObjectFFRFreducedOrderInterface

mPhitTPsiTilde	NumpyMatrix		Matrix[]	special FFRFreducedOrder matrix, computed in ObjectFFRFreducedOrderInterface
mXRefTildePsi	NumpyMatrix		Matrix[]	special FFRFreducedOrder matrix, computed in ObjectFFRFreducedOrderInterface
mXRefTildePsiTilde	NumpyMatrix		Matrix[]	special FFRFreducedOrder matrix, computed in ObjectFFRFreducedOrderInterface
physicsCenterOfMassTilde	Matrix3D		[[0,0,0], [0,0,0], [0,0,0]]	tilde matrix from local position of COM; autocomputed during initialization
tempUserFunctionForce	NumpyVector		[]	temporary vector for UF force
visualization	VObjectFFRFreducedOrder			parameters for visualization of item

The item VObjectFFRFreducedOrder has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown; use visualizationSettings.bodies.deformationScaleFactor to draw scaled (local) deformations; the reference frame node is shown with additional letters RF
color	Float4	4	[-1.,-1.,-1.,-1.]	RGBA color for object; 4th value is alpha-transparency; R=-1.f means, that default color is used
triangleMesh	NumpyMatrixI		MatrixI[]	a matrix, containg node number triples in every row, referring to the node numbers of the GenericODE2 object; the mesh uses the nodes to visualize the underlying object; contour plot colors are still computed in the local frame!
showNodes	Bool		False	set true, nodes are drawn uniquely via the mesh, eventually using the floating reference frame, even in the visualization of the node is show=False; node numbers are shown with indicator 'NF'

8.3.4.1 DESCRIPTION of ObjectFFRFreducedOrder:

Information on input parameters:

input parameter	symbol	description see tables above
nodeNumbers	$\mathbf{n} = [n_0, n_1]^T$	
massMatrixReduced	$\mathbf{M}_{red} \in \mathbb{R}^{n_m \times n_m}$	
stiffnessMatrixReduced	$\mathbf{K}_{red} \in \mathbb{R}^{n_m \times n_m}$	
dampingMatrixReduced	$\mathbf{D}_{red} \in \mathbb{R}^{n_m \times n_m}$	
forceUserFunction	$\mathbf{f}_{user} \in \mathbb{R}^{n_{ODE2}}$	
massMatrixUserFunction	$\mathbf{M}_{user} \in \mathbb{R}^{n_{ODE2} \times n_{ODE2}}$	
modeBasis	${}^b\boldsymbol{\Psi} \in \mathbb{R}^{n_f \times n_m}$	
outputVariableModeBasis	${}^b\boldsymbol{\Psi}_{OV} \in \mathbb{R}^{n_n \times (n_m \cdot s_{OV})}$	
referencePositions	${}^b\mathbf{x}_{ref} \in \mathbb{R}^{n_f}$	
physicsMass	m	
physicsInertia	$\mathbf{J}_r \in \mathbb{R}^{3\times3}$	
physicsCenterOfMass	${}^b\mathbf{b}_{COM}$	
physicsCenterOfMassTilde	${}^b\tilde{\mathbf{b}}_{COM}$	
tempUserFunctionForce	$\mathbf{f}_{temp} \in \mathbb{R}^{n_{ODE2}}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Coordinates		all ODE2 coordinates
Coordinates_t		all ODE2 velocity coordinates
Force		generalized forces for all coordinates (residual of all forces except mass*accleration; corresponds to ComputeODE2LHS)

8.3.4.2 Super element output variables

Functions like `GetObjectOutputSuperElement(...)`, see Section 6.5.4, or `SensorSuperElement`, see Section 6.5.7, directly access special output variables (`OutputVariableType`) of the mesh nodes of the superelement. Additionally, the contour drawing of the object can make use the `OutputVariableType` of the meshnodes.

super element output variables	symbol	description
DisplacementLocal (mesh node i)	${}^b\mathbf{u}_f^{(i)} = \left({}^b\boldsymbol{\Psi}\boldsymbol{\zeta}\right)_{3\cdot i\ldots3\cdot i+2} = \begin{bmatrix} {}^b\mathbf{q}_{f,i\cdot3} \\ {}^b\mathbf{q}_{f,i\cdot3+1} \\ {}^b\mathbf{q}_{f,i\cdot3+2} \end{bmatrix}$	local nodal mesh displacement in reference (body) frame, measuring only flexible part of displacement
VelocityLocal (mesh node (i))	${}^b\dot{\mathbf{u}}_f^{(i)} = \left({}^b\boldsymbol{\Psi}\dot{\boldsymbol{\zeta}}\right)_{3\cdot i\ldots3\cdot i+2}$	local nodal mesh velocity in reference (body) frame, only for flexible part of displacement
Displacement (mesh node (i))	${}^0\mathbf{u}_{config}^{(i)} = {}^0\mathbf{q}_{t,config} + {}^{0b}\mathbf{A}_{config}\ {}^b\mathbf{p}_{f,config}^{(i)} - ({}^0\mathbf{q}_{t,ref} + {}^{0b}\mathbf{A}_{ref}\ {}^b\mathbf{x}_{ref}^{(i)})$	nodal mesh displacement in global coordinates
Position (mesh node (i))	${}^0\mathbf{p}^{(i)} = {}^0\mathbf{r} + {}^{0b}\mathbf{A}\ {}^b\mathbf{p}_f^{(i)}$	nodal mesh position in global coordinates

Velocity (mesh node (i))	${}^0\dot{\mathbf{u}}^{(i)} = {}^0\dot{\mathbf{q}}_\mathrm{t} + {}^{0b}\mathbf{A}\,({}^b\dot{\mathbf{u}}_\mathrm{f}^{(i)} + {}^b\tilde{\boldsymbol{\omega}}\,{}^b\mathbf{p}_\mathrm{f}^{(i)})$	nodal mesh velocity in global coordinates
Acceleration (mesh node (i))	${}^0\mathbf{a}^{(i)} = {}^0\ddot{\mathbf{q}}_\mathrm{t} + {}^{0b}\mathbf{A}\,{}^b\ddot{\mathbf{u}}_\mathrm{f}^{(i)} + 2\,{}^0\boldsymbol{\omega} \times {}^{0b}\mathbf{A}\,{}^b\dot{\mathbf{u}}_\mathrm{f}^{(i)} + {}^0\boldsymbol{\alpha} \times {}^0\mathbf{p}_\mathrm{f}^{(i)} + {}^0\boldsymbol{\omega} \times ({}^0\boldsymbol{\omega} \times {}^0\mathbf{p}_\mathrm{f}^{(i)})$	global acceleration of mesh node n_i including rigid body motion and flexible deformation; note that ${}^0\mathbf{x}(n_i) = {}^{0b}\mathbf{A}\,{}^b\mathbf{x}(n_i)$
StressLocal (mesh node (i))	${}^b\boldsymbol{\sigma}^{(i)} = ({}^b\boldsymbol{\Psi}_{OV}\,\boldsymbol{\zeta})_{3\cdot i \ldots 3\cdot i+5}$	linearized stress components of mesh node (i) in reference frame; $\boldsymbol{\sigma} = [\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{yz}, \sigma_{xz}, \sigma_{xy}]^\mathrm{T}$; ONLY available, if ${}^b\boldsymbol{\Psi}_{OV}$ is provided and `outputVariableTypeModeBasis== exu.OutputVariableType.StressLocal`
StrainLocal (mesh node (i))	${}^b\boldsymbol{\varepsilon}^{(i)} = ({}^b\boldsymbol{\Psi}_{OV}\boldsymbol{\zeta})_{3\cdot i \ldots 3\cdot i+5}$	linearized strain components of mesh node (i) in reference frame; $\boldsymbol{\varepsilon} = [\varepsilon_{xx}, \varepsilon_{yy}, \varepsilon_{zz}, \varepsilon_{yz}, \varepsilon_{xz}, \varepsilon_{xy}]^\mathrm{T}$; ONLY available, if ${}^b\boldsymbol{\Psi}_{OV}$ is provided and `outputVariableTypeModeBasis== exu.OutputVariableType.StrainLocal`

intermediate variables	**symbol**	**description**
reference frame	b	the body-fixed / local frame is always denoted by b
number of rigid body coordinates	n_rigid	number of rigid body node coordinates: 6 in case of Euler angles (not fully available for ObjectFFRFreducedOrder) and 7 in case of Euler parameters
number of flexible / mesh coordinates	$n_\mathrm{f} = 3 \cdot n_n$	with number of nodes n_n; relevant for visualization
number of modal coordinates	$n_m \ll n_\mathrm{f}$	the number of reduced or modal coordinates, computed from number of columns given in `modeBasis`
total number object coordinates	$n_{ODE2} = n_m + n_{rigid}$	
reference frame origin	${}^0\mathbf{r} = {}^0\mathbf{q}_\mathrm{t} + {}^0\mathbf{q}_\mathrm{t,ref}$	reference frame position (origin)
reference frame rotation	$\boldsymbol{\theta}_\mathrm{config} = \boldsymbol{\theta}_\mathrm{config} + \boldsymbol{\theta}_{ref}$	reference frame rotation parameters in any configuration except reference
reference frame orientation	${}^{0b}\mathbf{A}_\mathrm{config} = {}^{0b}\mathbf{A}_\mathrm{config}(\boldsymbol{\theta}_\mathrm{config})$	transformation matrix for transformation of local (reference frame) to global coordinates, given by underlying rigid body node n_0
local vector of flexible coordinates	${}^b\mathbf{q}_\mathrm{f} = {}^b\boldsymbol{\Psi}\boldsymbol{\zeta}$	represents mesh displacements; vector of alternating x,y, an z coordinates of local (in body frame) mesh displacements reconstructed from modal coordinates $\boldsymbol{\zeta}$; only evaluated for selected node points (e.g., sensors) during computation; corresponds to same vector in `ObjectFFRF`
local nodal positions	${}^b\mathbf{p}_\mathrm{f} = {}^b\mathbf{q}_\mathrm{f} + {}^b\mathbf{x}_\mathrm{ref}$	vector of all body-fixed nodal positions including flexible part; only evaluated for selected node points during computation

local position of node (i)	${}^b\mathbf{p}_\mathrm{f}^{(i)} = {}^b\mathbf{u}_\mathrm{f}^{(i)} + {}^b\mathbf{x}_\mathrm{ref}^{(i)} = \begin{bmatrix} {}^b\mathbf{q}_{\mathrm{f},i\cdot3} \\ {}^b\mathbf{q}_{\mathrm{f},i\cdot3+1} \\ {}^b\mathbf{q}_{\mathrm{f},i\cdot3+2} \end{bmatrix} + \begin{bmatrix} {}^b\mathbf{x}_{\mathrm{ref},i\cdot3} \\ {}^b\mathbf{x}_{\mathrm{ref},i\cdot3+1} \\ {}^b\mathbf{x}_{\mathrm{ref},i\cdot3+2} \end{bmatrix}$	body-fixed, deformed nodal mesh position (including flexible part)
vector of modal coordinates	$\boldsymbol{\zeta} = [\zeta_0, \ldots, \zeta_{n_m-1}]^\mathrm{T}$	vector of modal or reduced coordinates; these coordinates can either represent amplitudes of eigenmodes, static modes or general modes, depending on your mode basis
coordinate vector	$\mathbf{q} = [{}^0\mathbf{q}_\mathrm{t}, \boldsymbol{\psi}, \boldsymbol{\zeta}]$	vector of object coordinates; $\mathbf{q}_\mathrm{t}$ and $\boldsymbol{\psi}$ are the translation and rotation part of displacements of the reference frame, provided by the rigid body node (node number 0)
flexible coordinates transformation matrix	${}^{0b}\mathbf{A}_{bd} = \mathrm{diag}([{}^{0b}\mathbf{A}, \ldots, {}^{0b}\mathbf{A}])$	block diagonal transformation matrix, which transforms all flexible coordinates from local to global coordinates

8.3.4.3 Modal reduction and reduced inertia matrices

The formulation is based on the EOM of `ObjectFFRF`, **also regarding parts of notation** and some input parameters, Section 8.3.3, and can be found in Zwölfer and Gerstmayr [67] with only small modifications in the notation. The notation of kinematics quantities follows the floating frame of reference idea with quantities given in the tables above and sketched in Fig. 8.2.

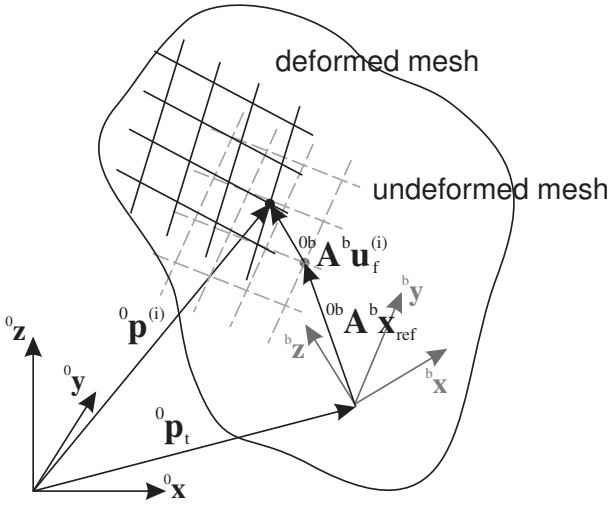


Figure 8.2: Floating frame of reference with exemplary position of a mesh node i.

The reduced order FFRF formulation is based on an approximation of flexible coordinates ${}^b\mathbf{q}_\mathrm{f}$ by means of a reduction or mode basis ${}^b\boldsymbol{\Psi}$ (`modeBasis`) and the the modal coordinates $\boldsymbol{\zeta}$,

$$ {}^b\mathbf{q}_\mathrm{f} \approx {}^b\boldsymbol{\Psi}\boldsymbol{\zeta} \tag{8.97} $$

The mode basis ${}^b\boldsymbol{\Psi}$ contains so-called mode shape vectors in its columns, which may be computed from eigen analysis, static computation or more advanced techniques, see the helper functions in module `exudyn.FEM`, within the class FEMinterface. To compute eigen modes, use `FEMinterface.ComputeEigenmodes(...)` or `FEMinterface.ComputeHurtyCraigBamptonModes(...)`. For details on model order reduction and component mode synthesis, see Section 5.5. In many applications, n_m typically ranges between 10 and 50, but also beyond – depending on the desired accuracy of the model.

The `ObjectFFRF` coordinates and Eqs. (8.86)[2] can be reduced by the matrix $\mathbf{H} \in \mathbb{R}^{(n_t+n_{\text{rigid}})\times n_{ODE2}}$,

$$\mathbf{q}_{FFRF} = \begin{bmatrix} \mathbf{q}_t \\ \boldsymbol{\theta} \\ {}^b\mathbf{q}_f \end{bmatrix} = \begin{bmatrix} \mathbf{I}_{3\times3} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_r & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & {}^b\boldsymbol{\Psi} \end{bmatrix} \begin{bmatrix} \mathbf{q}_t \\ \boldsymbol{\theta} \\ \boldsymbol{\zeta} \end{bmatrix} = \mathbf{H}\,\mathbf{q} \tag{8.98}$$

with the 4×4 identity matrix $\mathbf{I}_r$ in case of Euler parameters and the reduced coordinates $\mathbf{q}$.

The reduced equations follow from the reduction of system matrices in Eqs. (8.86),

$$\mathbf{K}_{\text{red}} = {}^b\boldsymbol{\Psi}^T\,{}^b\mathbf{K}\,{}^b\boldsymbol{\Psi}\,, \tag{8.99}$$

$$\mathbf{M}_{\text{red}} = {}^b\boldsymbol{\Psi}^T\,{}^b\mathbf{M}\,{}^b\boldsymbol{\Psi}\,, \tag{8.100}$$

$$\tag{8.101}$$

the computation of rigid body inertia

$${}^b\boldsymbol{\Theta}_u = {}^b\tilde{\mathbf{x}}_{\text{ref}}^T\,{}^b\mathbf{M}\,{}^b\tilde{\mathbf{x}}_{\text{ref}} \tag{8.102}$$

$$\tag{8.103}$$

the center of mass (and according tilde matrix), using $\boldsymbol{\Phi}_t$ from Eq. (8.69),

$${}^b\boldsymbol{\chi}_u = \frac{1}{m}\boldsymbol{\Phi}_t^T\,{}^b\mathbf{M}\,{}^b\mathbf{x}_{\text{ref}} \tag{8.104}$$

$${}^b\tilde{\boldsymbol{\chi}}_u = \frac{1}{m}\boldsymbol{\Phi}_t^T\,{}^b\mathbf{M}\,{}^b\tilde{\mathbf{x}}_{\text{ref}} \tag{8.105}$$

$$\tag{8.106}$$

and seven inertia-like matrices [67],

$$\mathbf{M}_{AB} = \mathbf{A}^T\,{}^b\mathbf{M}\,\mathbf{B}, \quad \text{using} \quad \mathbf{AB} \in \left[\boldsymbol{\Psi}\boldsymbol{\Psi},\ \widetilde{\boldsymbol{\Psi}}\boldsymbol{\Psi},\ \widetilde{\boldsymbol{\Psi}}\widetilde{\boldsymbol{\Psi}},\ \boldsymbol{\Phi}_t\boldsymbol{\Psi},\ \boldsymbol{\Phi}_t\widetilde{\boldsymbol{\Psi}},\ \tilde{\mathbf{x}}_{\text{ref}}\boldsymbol{\Psi},\ \tilde{\mathbf{x}}_{\text{ref}}\widetilde{\boldsymbol{\Psi}}\right] \tag{8.107}$$

Note that the special tilde operator for vectors $\mathbf{p} \in \mathbb{R}^{n_f}$ of Eq. (8.78) is frequently used.

8.3.4.4 Equations of motion

Equations of motion, in case that `computeFFRFterms = True`:

$$\left(\mathbf{M}_{user}(mbs,t,\mathbf{q},\dot{\mathbf{q}}) + \begin{bmatrix} \mathbf{M}_{tt} & \mathbf{M}_{tr} & \mathbf{M}_{tf} \\ & \mathbf{M}_{rr} & \mathbf{M}_{rf} \\ \text{sym.} & & \mathbf{M}_{ff} \end{bmatrix}\right)\ddot{\mathbf{q}} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathbf{D}_{ff} \end{bmatrix}\dot{\mathbf{q}} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathbf{K}_{ff} \end{bmatrix}\mathbf{q} = \mathbf{f}_v(\mathbf{q},\dot{\mathbf{q}}) + \mathbf{f}_{user}(mbs,t,\mathbf{q},\dot{\mathbf{q}}) \tag{8.108}$$

[2] this is not done for user functions and `forceVector`

[3] Note that in case of Euler parameters for the parameterization of rotations for the reference frame, the Euler parameter constraint equation is added automatically by this object. The single terms of the mass matrix are defined as[67]

$$\mathbf{M}_{tt} = m\mathbf{I}_{3\times 3} \tag{8.110}$$

$$\mathbf{M}_{tr} = -{}^{0b}\mathbf{A}\left[m\,{}^{b}\tilde{\boldsymbol{\chi}}_{u} + \mathbf{M}_{\Phi_t\widetilde{\Psi}}\left(\boldsymbol{\zeta}\otimes\mathbf{I}\right)\right]{}^{b}\mathbf{G} \tag{8.111}$$

$$\mathbf{M}_{tf} = {}^{0b}\mathbf{A}\,\mathbf{M}_{\Phi_t\Psi} \tag{8.112}$$

$$\mathbf{M}_{rr} = {}^{b}\mathbf{G}^{\mathrm{T}}\left[{}^{b}\boldsymbol{\Theta}_{u} + \mathbf{M}_{\tilde{\mathbf{x}}_{\mathrm{ref}}\widetilde{\Psi}}\left(\boldsymbol{\zeta}\otimes\mathbf{I}\right) + \left(\boldsymbol{\zeta}\otimes\mathbf{I}\right)^{\mathrm{T}}\mathbf{M}^{\mathrm{T}}_{\tilde{\mathbf{x}}_{\mathrm{ref}}\widetilde{\Psi}} + \left(\boldsymbol{\zeta}\otimes\mathbf{I}\right)^{\mathrm{T}}\mathbf{M}_{\widetilde{\Psi}\widetilde{\Psi}}\left(\boldsymbol{\zeta}\otimes\mathbf{I}\right)\right]{}^{b}\mathbf{G} \tag{8.113}$$

$$\mathbf{M}_{rf} = -{}^{b}\mathbf{G}^{\mathrm{T}}\left[\mathbf{M}_{\tilde{\mathbf{x}}_{\mathrm{ref}}\Psi} + \left(\boldsymbol{\zeta}\otimes\mathbf{I}\right)^{\mathrm{T}}\mathbf{M}_{\widetilde{\Psi}\Psi}\right] \tag{8.114}$$

$$\mathbf{M}_{ff} = \mathbf{M}_{\Psi\Psi} \tag{8.115}$$

with the Kronecker product[4],

$$\boldsymbol{\zeta}\otimes\mathbf{I} = \begin{bmatrix} \zeta_0\mathbf{I} \\ \vdots \\ \zeta_{m-1}\mathbf{I} \end{bmatrix} \tag{8.116}$$

The quadratic velocity vector $\mathbf{f}_v(\mathbf{q},\dot{\mathbf{q}}) = \left[\mathbf{f}_{vt}^{\mathrm{T}},\ \mathbf{f}_{vr}^{\mathrm{T}},\ \mathbf{f}_{vf}^{\mathrm{T}}\right]^{\mathrm{T}}$ reads

$$\begin{aligned}\mathbf{f}_{vt} &= {}^{0b}\mathbf{A}\,{}^{b}\tilde{\boldsymbol{\omega}}\left[m\,{}^{b}\tilde{\boldsymbol{\chi}}_{u} + \mathbf{M}_{\Phi_t\widetilde{\Psi}}\left(\boldsymbol{\zeta}\otimes\mathbf{I}\right)\right]{}^{b}\boldsymbol{\omega} + 2\,{}^{0b}\mathbf{A}\,\mathbf{M}_{\Phi_t\widetilde{\Psi}}\left(\dot{\boldsymbol{\zeta}}\otimes\mathbf{I}\right){}^{b}\boldsymbol{\omega} \\ &\quad + {}^{0b}\mathbf{A}\left[m\,{}^{b}\tilde{\boldsymbol{\chi}}_{u} + \mathbf{M}_{\Phi_t\widetilde{\Psi}}\left(\boldsymbol{\zeta}\otimes\mathbf{I}\right)\right]{}^{b}\dot{\mathbf{G}}\,\dot{\boldsymbol{\theta}}\,,\end{aligned} \tag{8.117}$$

$$\begin{aligned}\mathbf{f}_{vr} &= -{}^{b}\mathbf{G}^{\mathrm{T}}\,{}^{b}\tilde{\boldsymbol{\omega}}\left[{}^{b}\boldsymbol{\Theta}_{u} + \mathbf{M}_{\tilde{\mathbf{x}}_{\mathrm{ref}}\widetilde{\Psi}}\left(\boldsymbol{\zeta}\otimes\mathbf{I}\right) + \left(\boldsymbol{\zeta}\otimes\mathbf{I}\right)^{\mathrm{T}}\mathbf{M}^{\mathrm{T}}_{\tilde{\mathbf{x}}_{\mathrm{ref}}\widetilde{\Psi}} + \left(\boldsymbol{\zeta}\otimes\mathbf{I}\right)^{\mathrm{T}}\mathbf{M}_{\widetilde{\Psi}\widetilde{\Psi}}\left(\boldsymbol{\zeta}\otimes\mathbf{I}\right)\right]{}^{b}\boldsymbol{\omega} \\ &\quad -2\,{}^{b}\mathbf{G}^{\mathrm{T}}\left[\mathbf{M}_{\tilde{\mathbf{x}}_{\mathrm{ref}}\widetilde{\Psi}}\left(\dot{\boldsymbol{\zeta}}\otimes\mathbf{I}\right) + \left(\boldsymbol{\zeta}\otimes\mathbf{I}\right)^{\mathrm{T}}\mathbf{M}_{\widetilde{\Psi}\widetilde{\Psi}}\left(\dot{\boldsymbol{\zeta}}\otimes\mathbf{I}\right)\right]{}^{b}\boldsymbol{\omega} \\ &\quad -{}^{b}\mathbf{G}^{\mathrm{T}}\left[{}^{b}\boldsymbol{\Theta}_{u} + \mathbf{M}_{\tilde{\mathbf{x}}_{\mathrm{ref}}\widetilde{\Psi}}\left(\boldsymbol{\zeta}\otimes\mathbf{I}\right) + \left(\boldsymbol{\zeta}\otimes\mathbf{I}\right)^{\mathrm{T}}\mathbf{M}^{\mathrm{T}}_{\tilde{\mathbf{x}}_{\mathrm{ref}}\widetilde{\Psi}} + \left(\boldsymbol{\zeta}\otimes\mathbf{I}\right)^{\mathrm{T}}\mathbf{M}_{\widetilde{\Psi}\widetilde{\Psi}}\left(\boldsymbol{\zeta}\otimes\mathbf{I}\right)\right]{}^{b}\dot{\mathbf{G}}\,\dot{\boldsymbol{\theta}}\,,\end{aligned} \tag{8.118}$$

$$\begin{aligned}\mathbf{f}_{vf} &= \left(\mathbf{I}_{\zeta}\otimes{}^{b}\boldsymbol{\omega}\right)^{\mathrm{T}}\left[\mathbf{M}^{\mathrm{T}}_{\tilde{\mathbf{x}}_{\mathrm{ref}}\widetilde{\Psi}} + \mathbf{M}_{\widetilde{\Psi}\widetilde{\Psi}}\left(\boldsymbol{\zeta}\otimes\mathbf{I}\right)\right]{}^{b}\boldsymbol{\omega} + 2\mathbf{M}^{\mathrm{T}}_{\widetilde{\Psi}\Psi}\left(\dot{\boldsymbol{\zeta}}\otimes\mathbf{I}\right){}^{b}\boldsymbol{\omega} \\ &\quad + \left[\mathbf{M}^{\mathrm{T}}_{\tilde{\mathbf{x}}_{\mathrm{ref}}\Psi} + \mathbf{M}^{\mathrm{T}}_{\widetilde{\Psi}\Psi}\left(\boldsymbol{\zeta}\otimes\mathbf{I}\right)\right]{}^{b}\dot{\mathbf{G}}\,\dot{\boldsymbol{\theta}}\,.\end{aligned} \tag{8.119}$$

Note that terms including ${}^{b}\dot{\mathbf{G}}\,\dot{\boldsymbol{\theta}}$ vanish in case of Euler parameters or in case that ${}^{b}\dot{\mathbf{G}} = \mathbf{0}$, and we use another Kronecker product with the unit matrix $\mathbf{I}_{\zeta}\in\mathbb{R}^{n_m\times n_m}$,

$$\mathbf{I}_{\zeta}\otimes{}^{b}\boldsymbol{\omega} = \begin{bmatrix} {}^{b}\boldsymbol{\omega} & & \\ & \ddots & \\ & & {}^{b}\boldsymbol{\omega} \end{bmatrix} \in \mathbb{R}^{3n_m\times n_m} \tag{8.120}$$

[3] NOTE that currently the internal (C++) computed terms are zero,

$$\begin{bmatrix} \mathbf{M}_{tt} & \mathbf{M}_{tr} & \mathbf{M}_{tf} \\ & \mathbf{M}_{rr} & \mathbf{M}_{rf} \\ \text{sym.} & & \mathbf{M}_{ff} \end{bmatrix} = \mathbf{0} \quad \text{and} \quad \mathbf{f}_v(\mathbf{q},\dot{\mathbf{q}}) = \mathbf{0}\,, \tag{8.109}$$

but they are implemented in predefined user functions, see `FEM.py`, Section 7.7.6. In near future, these terms will be implemented in C++ and replace the user functions.

[4] In Python numpy module this is computed by `numpy.kron(zeta, Im).T`

In case that `computeFFRFterms = False`, the mass terms $\mathbf{M}_{\text{tt}} \dots \mathbf{M}_{\text{ff}}$ are zero (not computed) and the quadratic velocity vector $\mathbf{f}_Q = \mathbf{0}$. Note that the user functions $\mathbf{f}_{user}(mbs, t, \mathbf{q}, \dot{\mathbf{q}})$ and $\mathbf{M}_{user}(mbs, t, \mathbf{q}, \dot{\mathbf{q}})$ may be empty (=0). The detailed equations of motion for this element can be found in [67].

8.3.4.5 Position Jacobian

For joints and loads, the position jacobian of a node is needed in order to compute forces applied to averaged displacements and rotations at nodes. Recall that the modal coordinates ζ are transformed to node coordinates by means of the mode basis ${}^b\boldsymbol{\Psi}$,

$$ {}^b\mathbf{q}_{\text{f}} = {}^b\boldsymbol{\Psi}\zeta \,. \tag{8.121} $$

The local displacements ${}^b\mathbf{u}_{\text{f}}^{(i)}$ of a specific node i can be reconstructed in this way by means of

$$ {}^b\mathbf{u}_{\text{f}}^{(i)} = \begin{bmatrix} {}^b\mathbf{q}_{\text{f},i\cdot 3} \\ {}^b\mathbf{q}_{\text{f},i\cdot 3+1} \\ {}^b\mathbf{q}_{\text{f},i\cdot 3+2} \end{bmatrix} , \tag{8.122} $$

and the global position of a node, see tables above, reads

$$ {}^0\mathbf{p}^{(i)} = {}^0\mathbf{p}_{\text{t}} + {}^{0b}\mathbf{A} \left({}^b\mathbf{u}_{\text{f}}^{(i)} + {}^b\mathbf{x}_{\text{ref}}^{(i)} \right) \tag{8.123} $$

Thus, the jacobian of the global position reads

$$ {}^0\mathbf{J}_{\text{pos}}^{(i)} = \frac{\partial\, {}^0\mathbf{p}^{(i)}}{\partial[\mathbf{q}_{\text{t}},\ \theta,\ \zeta]} = \left[\mathbf{I}_{3\times 3},\ -{}^{0b}\mathbf{A} \left({}^b\tilde{\mathbf{u}}_{\text{f}}^{(i)} + {}^b\tilde{\mathbf{x}}_{\text{ref}}^{(i)} \right) {}^b\mathbf{G} ,\ {}^{0b}\mathbf{A} \begin{bmatrix} {}^b\boldsymbol{\Psi}_{r=3i}^{\text{T}} \\ {}^b\boldsymbol{\Psi}_{r=3i+1}^{\text{T}} \\ {}^b\boldsymbol{\Psi}_{r=3i+2}^{\text{T}} \end{bmatrix} \right] , \tag{8.124} $$

in which ${}^b\boldsymbol{\Psi}_{r=\dots}$ represents the row r of the mode basis (matrix) ${}^b\Psi$, and the matrix

$$ \begin{bmatrix} {}^b\boldsymbol{\Psi}_{r=3i}^{\text{T}} \\ {}^b\boldsymbol{\Psi}_{r=3i+1}^{\text{T}} \\ {}^b\boldsymbol{\Psi}_{r=3i+2}^{\text{T}} \end{bmatrix} \in \mathbb{R}^{3\times n_m} \tag{8.125} $$

Furthermore, the jacobian of the local position reads

$$ {}^b\mathbf{J}_{\text{pos}}^{(i)} = \frac{\partial\, {}^b\mathbf{p}_{\text{f}}^{(i)}}{\partial[\mathbf{q}_{\text{t}},\ \theta,\ \zeta]} = \left[\mathbf{0},\ \mathbf{0},\ \begin{bmatrix} {}^b\boldsymbol{\Psi}_{r=3i}^{\text{T}} \\ {}^b\boldsymbol{\Psi}_{r=3i+1}^{\text{T}} \\ {}^b\boldsymbol{\Psi}_{r=3i+2}^{\text{T}} \end{bmatrix} \right] , \tag{8.126} $$

which is used in `MarkerSuperElementRigid`.

8.3.4.6 Joints and Loads

Use special `MarkerSuperElementPosition` to apply forces, SpringDampers or spherical joints. This marker can be attached to a single node of the underlying mesh or to a set of nodes, which is then averaged, see the according marker description.

Use special `MarkerSuperElementRigid` to apply torques or special joints (e.g., `JointGeneric`). This marker must be attached to a set of nodes which can represent rigid body motion. The rigid body motion is then averaged for all of these nodes, see the according marker description.

For application of mass proportional loads (gravity), you can use conventional MarkerBodyMass. However, **do not use** `MarkerBodyPosition` or `MarkerBodyRigid` for ObjectFFRFreducedOrder, unless wanted, because it only attaches to the floating frame. This means, that a force to a `MarkerBodyPosition` would only be applied to the (rigid) floating frame, but not onto the deformable body and results depend strongly on the choice of the reference frame (or the underlying mode shapes).

CoordinateLoads are added for each ODE2 coordinate on the RHS of the equations of motion.

Userfunction: `forceUserFunction(mbs, t, itemNumber, q, q_t)`

A user function, which computes a force vector depending on current time and states of object. Can be used to create any kind of mechanical system by using the object states. Note that itemNumber represents the index of the ObjectFFRFreducedOrder object in mbs, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs to which object belongs
`t`	Real	current time in mbs
`itemNumber`	Index	integer number of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
`q`	Vector $\in \mathbb{R}^{n}_{ODE2}$	FFRF object coordinates (rigid body coordinates and reduced coordinates in a list) in current configuration, without reference values
`q_t`	Vector $\in \mathbb{R}^{n}_{ODE2}$	object velocity coordinates (time derivatives of q) in current configuration
`return value`	Vector $\in \mathbb{R}^{n_{ODE2}}$	returns force vector for object

---

Userfunction: `massMatrixUserFunction(mbs, t, itemNumber, q, q_t)`

A user function, which computes a mass matrix depending on current time and states of object. Can be used to create any kind of mechanical system by using the object states.

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs to which object belongs
`t`	Real	current time in mbs

`itemNumber`	Index	integer number of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
`q`	Vector $\in \mathbb{R}^n_{ODE2}$	FFRF object coordinates (rigid body coordinates and reduced coordinates in a list) in current configuration, without reference values
`q_t`	Vector $\in \mathbb{R}^n_{ODE2}$	object velocity coordinates (time derivatives of `q`) in current configuration
`return value`	NumpyMatrix $\in \mathbb{R}^{n_{ODE2} \times n_{ODE2}}$	returns mass matrix for object

---

For examples on ObjectFFRFreducedOrder see Relevant Examples and TestModels with weblink:

- `NGsolvePistonEngine.py` (Examples/)
- `objectFFRFreducedOrderNetgen.py` (Examples/)
- `NGsolveCrankShaftTest.py` (TestModels/)
- `objectFFRFreducedOrderAccelerations.py` (TestModels/)
- `objectFFRFreducedOrderShowModes.py` (TestModels/)
- `objectFFRFreducedOrderStressModesTest.py` (TestModels/)
- `superElementRigidJointTest.py` (TestModels/)

8.4 Objects (FiniteElement)

A FiniteElement is a special Object and Body, which is used to define deformable bodies, such as beams or solid finite elements. FiniteElements are usually linked to two or more nodes.

8.4.1 ObjectANCFCable

A 3D cable finite element using 2 nodes of type NodePointSlope1. The localPosition of the beam with length L=physicsLength and height h ranges in X-direction in range $[0, L]$ and in Y-direction in range $[-h/2, h/2]$ (which is in fact not needed in the equations of motion (EOM)). For description see ObjectANCFCable2D, which is almost identical to 3D case. Note that this element does not include torsion, therfore a torque cannot be applied along the local x-axis.

Additional information for ObjectANCFCable:

- This `Object` has/provides the following types = `Body`, `MultiNoded`
- Requested `Node` type = `Position`
- **Short name** for Python = `Cable`
- **Short name** for Python visualization object = `VCable`

The item **ObjectANCFCable** with type = 'ANCFCable' has the following parameters:

Name	type	size	default value	description
name	String		''	objects's unique name
physicsLength	UReal		0.	[SI:m] reference length of beam; such that the total volume (e.g. for volume load) gives ρAL; must be positive
physicsMassPerLength	UReal		0.	[SI:kg/m] mass per length of beam
physicsBendingStiffness	UReal		0.	[SI:Nm^2] bending stiffness of beam; the bending moment is $m = EI(\kappa - \kappa_0)$, in which κ is the material measure of curvature
physicsAxialStiffness	UReal		0.	[SI:N] axial stiffness of beam; the axial force is $f_{ax} = EA(\varepsilon - \varepsilon_0)$, in which $\varepsilon = \lvert\mathbf{r}'\rvert - 1$ is the axial strain
physicsBendingDamping	UReal		0.	[SI:Nm^2/s] bending damping of beam ; the additional virtual work due to damping is $\delta W_{\dot\kappa} = \int_0^L \dot\kappa \delta\kappa dx$
physicsAxialDamping	UReal		0.	[SI:N/s] axial damping of beam; the additional virtual work due to damping is $\delta W_{\dot\varepsilon} = \int_0^L \dot\varepsilon \delta\varepsilon dx$
physicsReferenceAxialStrain	Real		0.	[SI:1] reference axial strain of beam (pre-deformation) of beam; without external loading the beam will statically keep the reference axial strain value

strainIsRelativeToReference	Real		0.	if set to 1., a pre-deformed reference configuration is considered as the stressless state; if set to 0., the straight configuration plus the values of ε_0 and κ_0 serve as a reference geometry; allows also values between 0. and 1.
nodeNumbers	NodeIndex2		[invalid [-1], invalid [-1]]	two node numbers ANCF cable element
useReducedOrderIntegration	Index		0	0/false: use Gauss order 9 integration for virtual work of axial forces, order 5 for virtual work of bending moments; 1/true: use Gauss order 7 integration for virtual work of axial forces, order 3 for virtual work of bending moments
visualization	VObjectANCFCable			parameters for visualization of item

The item VObjectANCFCable has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown; note that all quantities are computed at the beam centerline, even if drawn on surface of cylinder of beam; this effects, e.g., Displacement or Velocity, which is drawn constant over cross section
radius	float		0.	if radius==0, only the centerline is drawn; else, a cylinder with radius is drawn; circumferential tiling follows general.cylinderTiling and beam axis tiling follows bodies.beams.axialTiling
color	Float4		[-1.,-1.,-1.,-1.]	RGBA color of the object; if R==-1, use default color

---

8.4.1.1 DESCRIPTION of ObjectANCFCable:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
physicsLength	L	
physicsMassPerLength	ρA	
physicsBendingStiffness	EI	

physicsAxialStiffness	EA	
physicsBendingDamping	d_K	
physicsAxialDamping	d_ε	
physicsReferenceAxialStrain	ε_0	
strainIsRelativeToReference	f_{ref}	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
Position	${}^0\mathbf{p}_{\text{config}}(x,0,0) = \mathbf{r}_{\text{config}}(x) + y \cdot \mathbf{n}_{\text{config}}(x)$	global position vector of local position $[x,0,0]$
Displacement	${}^0\mathbf{u}_{\text{config}}(x,0,0) = {}^0\mathbf{p}_{\text{config}}(x,0,0) - {}^0\mathbf{p}_{\text{ref}}(x,0,0)$	global displacement vector of local position
Velocity	${}^0\mathbf{v}(x,0,0) = {}^0\dot{\mathbf{r}}(x)$	global velocity vector of local position
Director1	$\mathbf{r}'(x)$	(axial) slope vector of local axis position (at y=0)
StrainLocal	ε	axial strain (scalar) of local axis position (at Y=Z=0)
CurvatureLocal	$[K_x, K_y, K_z]^{\text{T}}$	local curvature vector
ForceLocal	N	(local) section normal force (scalar, including reference strains) (at y=z=0); note that strains are highly inaccurate when coupled to bending, thus consider useReducedOrderIntegration=2 and evaluate axial strain at nodes or at midpoint
TorqueLocal	M	(local) bending moment (scalar) (at y=z=0), which are bending moments as there is no torque
Acceleration	${}^0\mathbf{a}(x,0,0) = {}^0\ddot{\mathbf{r}}(x)$	global acceleration vector of local position

---

8.4.1.2 MINI EXAMPLE for ObjectANCFCable

```
from exudyn.beams import GenerateStraightLineANCFCable
rhoA = 78.
EA = 1000000.
EI = 833.3333333333333
cable = Cable(physicsMassPerLength=rhoA,
              physicsBendingStiffness=EI,
              physicsAxialStiffness=EA,
              )

ancf=GenerateStraightLineANCFCable(mbs=mbs,
```

```
                positionOfNode0=[0,0,0], positionOfNode1=[2,0,0],
                numberOfElements=32, #converged to 4 digits
                cableTemplate=cable, #this defines the beam element properties
                massProportionalLoad = [0,-9.81,0],
                fixedConstraintsNode0 = [1,1,1, 0,1,1], #add constraints for pos and rot (
r'_y,r'_z)
                )
lastNode = ancf[0][-1]

#assemble and solve system for default parameters
mbs.Assemble()
mbs.SolveStatic()

#check result
exudynTestGlobals.testResult = mbs.GetNodeOutput(lastNode, exu.OutputVariableType.
Displacement)[0]
#ux=-0.5013058140308901
```

8.4.2 ObjectANCFCable2D

A 2D cable finite element using 2 nodes of type NodePoint2DSlope1. The localPosition of the beam with length L=physicsLength and height h ranges in X-direction in range $[0, L]$ and in Y-direction in range $[-h/2, h/2]$ (which is in fact not needed in the EOM).

Additional information for ObjectANCFCable2D:

- Requested `Node` type = `Position2D` + `Orientation2D` + `Point2DSlope1` + `Position` + `Orientation`
- **Short name** for Python = `Cable2D`
- **Short name** for Python visualization object = `VCable2D`

The item **ObjectANCFCable2D** with type = 'ANCFCable2D' has the following parameters:

Name	type	size	default value	description
name	String		''	objects's unique name
physicsLength	UReal		0.	[SI:m] reference length of beam; such that the total volume (e.g. for volume load) gives ρAL; must be positive
physicsMassPerLength	UReal		0.	[SI:kg/m] mass per length of beam
physicsBendingStiffness	UReal		0.	[SI:Nm^2] bending stiffness of beam; the bending moment is $m = EI(\kappa - \kappa_0)$, in which κ is the material measure of curvature
physicsAxialStiffness	UReal		0.	[SI:N] axial stiffness of beam; the axial force is $f_{ax} = EA(\varepsilon - \varepsilon_0)$, in which $\varepsilon = \|\mathbf{r}'\| - 1$ is the axial strain
physicsBendingDamping	UReal		0.	[SI:Nm^2/s] bending damping of beam ; the additional virtual work due to damping is $\delta W_{\dot\kappa} = \int_0^L \dot\kappa \delta\kappa dx$
physicsAxialDamping	UReal		0.	[SI:N/s] axial damping of beam; the additional virtual work due to damping is $\delta W_{\dot\varepsilon} = \int_0^L \dot\varepsilon \delta\varepsilon dx$
physicsReferenceAxialStrain	Real		0.	[SI:1] reference axial strain of beam (pre-deformation) of beam; without external loading the beam will statically keep the reference axial strain value
physicsReferenceCurvature	Real		0.	[SI:1/m] reference curvature of beam (pre-deformation) of beam; without external loading the beam will statically keep the reference curvature value
strainIsRelativeToReference	Real		0.	if set to 1., a pre-deformed reference configuration is considered as the stressless state; if set to 0., the straight configuration plus the values of ε_0 and κ_0 serve as a reference geometry; allows also values between 0. and 1.

nodeNumbers	NodeIndex2		[invalid [-1], invalid [-1]]	two node numbers ANCF cable element
useReducedOrderIntegration	Index		0	0/false: use Gauss order 9 integration for virtual work of axial forces, order 5 for virtual work of bending moments; 1/True: use Gauss order 7 integration for virtual work of axial forces, order 3 for virtual work of bending moments; 2: use mixed Lobatto/-Gauss integration with exceptional quality of axial strain, however, spurious (hourglass) modes may occur!
axialForceUserFunction	PyFunctionMbsScalarIndexScalar9		0	A Python function which defines the (nonlinear relations) of local strains (including axial strain and bending strain) as well as time derivatives to the local axial force; see description below
bendingMomentUserFunction	PyFunctionMbsScalarIndexScalar9		0	A Python function which defines the (nonlinear relations) of local strains (including axial strain and bending strain) as well as time derivatives to the local bending moment; see description below
visualization	VObjectANCFCable2D			parameters for visualization of item

The item VObjectANCFCable2D has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawHeight	float		0.	if beam is drawn with rectangular shape, this is the drawing height
color	Float4		[-1.,-1.,-1.,-1.]	RGBA color of the object; if R==-1, use default color

---

8.4.2.1 DESCRIPTION of ObjectANCFCable2D:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
physicsLength	L	
physicsMassPerLength	ρA	
physicsBendingStiffness	EI	
physicsAxialStiffness	EA	

physicsBendingDamping	d_K	
physicsAxialDamping	d_ε	
physicsReferenceAxialStrain	ε_0	
physicsReferenceCurvature	κ_0	
strainIsRelativeToReference	f_{ref}	
axialForceUserFunction	UF $\in \mathbb{R}$	
bendingMomentUserFunction	UF $\in \mathbb{R}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
Position	${}^0\mathbf{p}_{\text{config}}(x,y,0) = \mathbf{r}_{\text{config}}(x) + y \cdot \mathbf{n}_{\text{config}}(x)$	global position vector of local position $[x,y,0]$
Displacement	${}^0\mathbf{u}_{\text{config}}(x,y,0) = {}^0\mathbf{p}_{\text{config}}(x,y,0) - {}^0\mathbf{p}_{\text{ref}}(x,y,0)$	global displacement vector of local position
Velocity	${}^0\mathbf{v}(x,y,0) = {}^0\dot{\mathbf{r}}(x) - y \cdot \omega_2 \cdot {}^0\mathbf{t}(x)$	global velocity vector of local position
VelocityLocal	${}^b\mathbf{v}(x,y,0) = {}^{b0}\mathbf{A}\ {}^0\mathbf{v}(x,y,0)$	local velocity vector of local position
Rotation	$\varphi = \text{atan2}(r'_y, r'_x)$	(scalar) rotation angle of axial slope vector (relative to global x-axis)
Director1	$\mathbf{r}'(x)$	(axial) slope vector of local axis position (at y=0)
StrainLocal	ε	axial strain (scalar) of local axis position (at Y=0)
CurvatureLocal	K	axial strain (scalar)
ForceLocal	N	(local) section normal force (scalar, including reference strains) (at y=0); note that strains are highly inaccurate when coupled to bending, thus consider useReducedOrderIntegration=2 and evaluate axial strain at nodes or at midpoint
TorqueLocal	M	(local) bending moment (scalar) (at y=0)
AngularVelocity	$\boldsymbol{\omega} = [0, , 0, \omega_2]$	angular velocity of local axis position (at y=0)
Acceleration	${}^0\mathbf{a}(x,y,0) = {}^0\ddot{\mathbf{r}}(x) - y \cdot \dot{\omega}_2 \cdot {}^0\mathbf{t}(x) - y \cdot \omega_2 \cdot {}^0\dot{\mathbf{t}}(x)$	global acceleration vector of local position
AngularAcceleration	$\boldsymbol{\alpha} = [0, , 0, \dot{\omega}_2]$	angular acceleration of local axis position

8.4.2.2 Definition of quantities

intermediate variables	**symbol**	**description**
beam height	h	beam height used in several definitions, but effectively undefined. The geometry of the cross section has no influence except for drawing or contact.

local beam position	${}^{b}\mathbf{b} = [x,\, y,\, 0]^{\mathrm{T}}$	local position at axial coordinate $x \in [0, L]$ and cross section coordinate $y \in [-h/2, h/2]$.
beam axis position	${}^{0}\mathbf{r}(x) = \mathbf{r}(x)$	
beam axis slope	${}^{0}\mathbf{r}'(x) = \mathbf{r}'(x)$	
beam axis tangent	${}^{0}\mathbf{t}(x) = \frac{\mathbf{r}'(x)}{\|\mathbf{r}(x)'\|}$	this (normalized) vector is normal to cross section
beam axis normal	${}^{0}\mathbf{n}(x) = [n_x,\, n_y]^{\mathrm{T}} = [-t_y,\, t_x]^{\mathrm{T}}$	this (normalized) vector lies within the cross section and defines positive y-direction.
angular velocity	$\omega_2 = (-r'_y \cdot \dot{r}'_x + r'_x \cdot \dot{r}'_y)/\|\mathbf{r}(x)'\|^2$	
rotation matrix	${}^{0b}\mathbf{A}$	

The Bernoulli-Euler beam is capable of large axial and bendig deformation as it employs the material measure of curvature for the bending.

8.4.2.3 Kinematics and interpolation

Note that in this section, expressions are written in 2D, while output variables are in general 3D quantities, adding a zero for the z-coordinate. ANCF elements follow the original concept proposed by Shabana [55]. The present 2D element is based on the interpolation used by Berzeri and Shabana [4], but the formulation (especially of the elastic forces) is according to Gerstmayr and Irschik [22]. Slight improvements for the integration of elastic forces and additional terms for off-axis forces and constraints are mentioned here.

The current position of an arbitrary element at local axial position $x \in [0, L]$, where L is the beam length, reads

$$\mathbf{r} = \mathbf{r}(x, t), \tag{8.127}$$

The derivative of the position w.r.t. the axial reference coordinate is denoted as slope vector,

$$\mathbf{r}' = \frac{\partial \mathbf{r}(x, t)}{\partial x} \tag{8.128}$$

The interpolation is based on cubic (spline) interpolation of position, displacements and velocities. The generalized coordinates $\mathbf{q} \in \mathbb{R}^8$ of the beam element is defined by

$$\mathbf{q} = \begin{bmatrix} \mathbf{r}_0^T & \mathbf{r}_0'^T & \mathbf{r}_1^T & \mathbf{r}_1'^T \end{bmatrix}^T. \tag{8.129}$$

in which $\mathbf{r}_0$ is the position of node 0 and $\mathbf{r}_1$ is the position of node 1, $\mathbf{r}'_0$ the slope at node 0 and $\mathbf{r}'_1$ the slope at node 1. Note that ANCF coordinates in the present notation are computed as sum of reference and current coordinates

$$\mathbf{q} = \mathbf{q}_{\mathrm{cur}} + \mathbf{q}_{\mathrm{ref}} \tag{8.130}$$

which is used throughout here. For time derivatives, it follows that $\dot{\mathbf{q}} = \dot{\mathbf{q}}_{\mathrm{cur}}$.

Position and slope are interpolated with shape functions. The position and slope along the beam are interpolated by means of

$$\mathbf{r} = \mathbf{S}\mathbf{q} \qquad \text{and} \qquad \mathbf{r}' = \mathbf{S}'\mathbf{q}. \tag{8.131}$$

in which $\mathbf{S}$ is the shape function matrix,

$$\mathbf{S}(x) = [\, S_1(x)\,\mathbf{I}_{2\times2} \;\; S_2(x)\,\mathbf{I}_{2\times2} \;\; S_3(x)\,\mathbf{I}_{2\times2} \;\; S_4(x)\,\mathbf{I}_{2\times2} \,]\,. \tag{8.132}$$

with identity matrix $\mathbf{I}_{2\times2} \in \mathbb{R}^{2\times2}$ and the shape functions

$$\begin{aligned} S_1(x) &= 1 - 3\frac{x^2}{L^2} + 2\frac{x^3}{L^3}, \quad S_2(x) = x - 2\frac{x^2}{L} + \frac{x^3}{L^2} \\ S_3(x) &= 3\frac{x^2}{L^2} - 2\frac{x^3}{L^3}, \qquad S_4(x) = -\frac{x^2}{L} + \frac{x^3}{L^2} \end{aligned} \tag{8.133}$$

Velocity simply follows as

$$\frac{\partial \mathbf{r}}{\partial t} = \dot{\mathbf{r}} = \mathbf{S}\dot{\mathbf{q}}. \tag{8.134}$$

8.4.2.4 Mass matrix

The mass matrix is constant and therefore precomputed at the first time it is needed (e.g., during computation of initial accelerations). The analytical form of the mass matrix reads

$$\mathbf{M}_{analytic} = \int_0^L \rho A \mathbf{S}(x)^T \mathbf{S}(x) dx \tag{8.135}$$

which is approximated using

$$\mathbf{M} = \sum_{ip=0}^{n_{ip}-1} w(x_{ip}) \frac{L}{2} \rho A \mathbf{S}(x_{ip})^T \mathbf{S}(x_{ip}) \tag{8.136}$$

with integration weights $w(x_{ip})$, $\sum w(x_{ip}) = 2$, and integration points x_{ip}, given as,

$$x_{ip} = \frac{L}{2}\xi_{ip} + \frac{L}{2}\,. \tag{8.137}$$

Here, we use the Gauss integration rule with order 7, having $n_{ip} = 4$ Gauss points, see Section 5.4. Due to the third order polynomials, the integration is exact up to round-off errors.

8.4.2.5 Elastic forces

The elastic forces $\mathbf{Q}_e$ are implicitly defined by the relation to the virtual work of elastic forces, δW_e, of applied forces, δW_a and of viscous forces, δW_v,

$$\mathbf{Q}_e^T \delta\mathbf{q} = \delta W_e + \delta W_a + \delta W_v. \tag{8.138}$$

The virtual work of elastic forces reads [22],

$$\delta W_e = \int_0^L (N\delta\varepsilon + M\delta K)\, dx, \tag{8.139}$$

in which the axial strain is defined as [22]

$$\varepsilon = \|\mathbf{r}'\| - 1. \tag{8.140}$$

and the material measure of curvature (bending strain) is given as

$$K = \mathbf{e}_3^T \frac{\mathbf{r}' \times \mathbf{r}''}{\|\mathbf{r}'\|^2}. \tag{8.141}$$

in which $\mathbf{e}_3$ is the unit vector which is perpendicular to the plane of the planar beam element.

By derivation, we obtain the variation of axial strain

$$\delta\varepsilon = \frac{\partial \varepsilon}{\partial q_i}\delta q_i = \frac{1}{\|\mathbf{r}'\|}\mathbf{r}'^T\mathbf{S}_i'\delta q_i. \tag{8.142}$$

and the variation of K

$$\begin{aligned} \delta K &= \frac{\partial}{\partial q_i}\left(\frac{(\mathbf{r}'^T \times \mathbf{r}'')^T\mathbf{e}_3}{\|\mathbf{r}'\|^2}\right)\delta q_i \\ &= \frac{1}{\|\mathbf{r}'\|^4}\left[\|\mathbf{r}'\|^2(\mathbf{S}_i' \times \mathbf{r}'' + \mathbf{r}' \times \mathbf{S}_i'') - 2(\mathbf{r}' \times \mathbf{r}'')(\mathbf{r}'^T\mathbf{S}_i')\right]^T \mathbf{e}_3\delta q_i \end{aligned} \tag{8.143}$$

The normal force (axial force) N in the beam is defined as function of the current strain ε,

$$N = EA\,(\varepsilon - \varepsilon_0 - f_{\text{ref}} \cdot \varepsilon_{\text{ref}}). \tag{8.144}$$

in which ε_0 includes the (pre-)stretch of the beam, e.g., due to temperature or plastic deformation and ε_{ref} includes the strain of the reference configuration. As can be seen, the reference strain is only considered, if $f_{\text{ref}} = 1$, which allows to consider the reference configuration to be completely stress-free (but the default value is $f_{\text{ref}} = 0$!). Note that – due to the inherent nonlinearity of ε – a combination of ε_0 and $f_{\text{ref}} = 1$ is physically only meaningful for small strains. A factor $f_{\text{ref}} < 1$ allows to realize a smooth transition between deformed and straight reference configuration, e.g. for initial configurations.

The bending moment M in the beam is defined as function of the current material measure of curvature K,

$$M = EI\,(K - K_0 - f_{\text{ref}} \cdot K_{\text{ref}}). \tag{8.145}$$

in which K_0 includes the (pre-)curvature of the undeformed beam and K_{ref} includes the curvature of the reference configuration, multiplied with the factor $f_{\text{ref}} = 1$, see the axial strain above.

Using the latter definitions, the elastic forces follow from Eq. (8.138).

The virtual work of viscous damping forces, assuming viscous effects proportial to axial streching and bending, is defined as

$$\delta W_v = \int_0^L \left(d_\varepsilon \dot{\varepsilon}\delta\varepsilon + d_K\dot{K}\delta K\right) dx. \tag{8.146}$$

with material coefficients d_ε and d_K. The time derivatives of axial strain $\dot{\varepsilon}_p$ follows by elementary differentiation

$$\dot{\varepsilon} = \frac{\partial}{\partial t}\left(\|\mathbf{r}'\| - 1\right) = \frac{1}{\|\mathbf{r}'\|}\mathbf{r}'^T\mathbf{S}'\dot{\mathbf{q}} \tag{8.147}$$

as well as the derivative of the curvature,

$$\begin{aligned} \dot{K} &= \frac{\partial}{\partial t}\left(\mathbf{e}_3^T\frac{\mathbf{r}' \times \mathbf{r}''}{\|\mathbf{r}'\|^2}\right) \\ &= \frac{\mathbf{e}_3^T}{(\mathbf{r}'^T\mathbf{r}')^2}\left((\mathbf{r}'^T\mathbf{r}')\frac{\partial\,(\mathbf{r}' \times \mathbf{r}'')^T}{\partial t} - (\mathbf{r}' \times \mathbf{r}'')^T\frac{\partial(\mathbf{r}'^T\mathbf{r}')}{\partial t}\right) \\ &= \frac{\mathbf{e}_3^T}{(\mathbf{r}'^T\mathbf{r}')^2}\left((\mathbf{r}'^T\mathbf{r}')\left((\mathbf{S}'\dot{\mathbf{q}}) \times \mathbf{r}'' + (\mathbf{S}''\dot{\mathbf{q}}) \times \mathbf{r}'\right) - (\mathbf{r}' \times \mathbf{r}'')\,(2\mathbf{r}'^T(\mathbf{S}'\dot{\mathbf{q}}))\right). \end{aligned} \tag{8.148}$$

The virtual work of applied forces reads

$$\delta W_a = \sum_i \mathbf{f}_i^T \delta \mathbf{r}_i(x_f) + \int_0^L \mathbf{b}^T \delta \mathbf{r}(x)\, dx\,, \tag{8.149}$$

in which $\mathbf{f}_i$ are forces applied to a certain position x_f at the beam centerline. The second term contains a load per length $\mathbf{b}$, which is case of gravity vector $\mathbf{g}$ reads

$$\mathbf{b} = \rho \mathbf{g}. \tag{8.150}$$

Note that the variation of $\mathbf{r}$ simply follows as

$$\delta \mathbf{r} = \mathbf{S}\, \delta \mathbf{q} \tag{8.151}$$

8.4.2.6 Numerical integration of Elastic Forces

The numerical integration of elastic forces $\mathbf{Q}_e$ is split into terms due to $\delta\varepsilon$ and δK,

$$\mathbf{Q}_e = \int_0^L \left(\bullet(x) \frac{\partial \delta \varepsilon}{\partial \delta \mathbf{q}} + \bullet(x) \frac{\partial \delta K}{\partial \delta \mathbf{q}} \right) dx \tag{8.152}$$

using different integration rules

$$\mathbf{Q}_e \approx \sum_{ip=0}^{n_{ip}^{\varepsilon}-1} \left(\frac{L}{2} \bullet (x_{ip}) \frac{\partial \delta \varepsilon}{\partial \delta \mathbf{q}} \right) + \sum_{ip=0}^{n_{ip}^{K}-1} \left(\frac{L}{2} \bullet (x_{ip}) \frac{\partial \delta K}{\partial \delta \mathbf{q}} \right) dx \tag{8.153}$$

with the integration points x_{ip} as defined in Eq. (8.137) and integration rules from Section 5.4. There are 3 different options for integration rules depending on the flag `useReducedOrderIntegration`:

1. `useReducedOrderIntegration` = 0: $n_{ip}^{\varepsilon} = 5$ (Gauss order 9), $n_{ip}^{K} = 3$ (Gauss order 5) – this is considered as full integration, leading to very small approximations; certainly, due to the high nonlinearity of expressions, this is only an approximation.
2. `useReducedOrderIntegration` = 1: $n_{ip}^{\varepsilon} = 4$ (Gauss order 7), $n_{ip}^{K} = 2$ (Gauss order 3) – this is considered as reduced integration, which is usually sufficiently accurate but leads to slightly less computational efforts, especially for bending terms.
3. `useReducedOrderIntegration` = 2: $n_{ip}^{\varepsilon} = 3$ (Lobatto order 3), $n_{ip}^{K} = 2$ (Gauss order 3) – this is a further reduced integration, with the exceptional property that axial strain and bending strain terms are computed at completely disjointed locations: axial strain terms are evaluated at 0, $L/2$ and L, while bending terms are evaluated at $\frac{L}{2} \pm \frac{L}{2}\sqrt{1/3}$. This allows axial strains to freely follow the bending terms at $\frac{L}{2} \pm \frac{L}{2}\sqrt{1/3}$, while axial strains are almost independent from bending terms at 0, $L/2$ and L. However, due to the highly reduced integration, spurious (hourglass) modes may occur in certain applications!

Note that the Jacobian of elastic forces is computed using automatic differentiation.

8.4.2.7 Access functions

For application of forces and constraints at any local beam position ${}^{b}\mathbf{b} = [x, y, 0]^{\mathrm{T}}$, the position / velocity Jacobian reads

$$\frac{\partial\, {}^{0}\mathbf{v}(x)}{\partial \dot{\mathbf{q}}} = \mathbf{S}(x) + \left[-y \cdot n_x S_1'(x) \frac{1}{\|\mathbf{r}'\|}\, {}^{0}\mathbf{t},\ -y \cdot n_y S_1'(x) \frac{1}{\|\mathbf{r}'\|}\, {}^{0}\mathbf{t},\ -y \cdot n_x S_2'(x) \frac{1}{\|\mathbf{r}'\|}\, {}^{0}\mathbf{t},\ \dots \right] \tag{8.154}$$

with the normalized beam axis normal ${}^{0}\mathbf{n} = [n_x, n_y]^{\mathrm{T}}$, see table above.

For application of torques at any axis point x, the rotation / angular velocity Jacobian $\frac{\partial\, {}^{0}\omega(x)}{\partial \dot{\mathbf{q}}} \in \mathbb{R}^{3\times 8}$ reads

$$\frac{\partial\, {}^{0}\omega(x)}{\partial \dot{\mathbf{q}}} = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ -r_y' \cdot S_1'(x) \frac{1}{\mathbf{r}'^2} & r_x' \cdot S_1'(x) \frac{1}{\mathbf{r}'^2} & -r_y' \cdot S_2'(x) \frac{1}{\mathbf{r}'^2} & \cdots & r_x' \cdot S_4'(x) \frac{1}{\mathbf{r}'^2} \end{bmatrix} \tag{8.155}$$

Userfunction: `axialForceUserFunction(mbs, t, itemNumber, axialPositionNormalized, axialStrain, axialStrain_t, axialStrainRef, physicsAxialStiffness, physicsAxialDamping, curvature, curvature_t, curvatureRef)`

A user function, which computes the axial force depending on time, strains and curvatures and object parameters (stiffness, damping). The object variables are provided to the function using the current values of the ANCFCable2D object. Note that itemNumber represents the index of the object in mbs, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`. **NOTE:** this function has a different interface as compared to the bending moment function.

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs to which object belongs
`t`	Real	current time in mbs
`itemNumber`	Index	integer number i_N of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
`axialPositionNormalized`	Real	axial position at the cable where the user function is evaluated; range is [0,1]
`axialStrain`	Real	ε
`axialStrain_t`	Real	ε_t
`axialStrainRef`	Real	$\varepsilon_0 + f_{\text{ref}} \cdot \varepsilon_{\text{ref}}$
`physicsAxialStiffness`	Real	as given in object parameters
`physicsAxialDamping`	Real	as given in object parameters
`curvature`	Real	K
`curvature_t`	Real	$\dot{K}$
`curvatureRef`	Real	$K_0 + f_{\text{ref}} \cdot K_{\text{ref}}$
`return value`	Real	scalar value of computed axial force

Userfunction: bendingMomentUserFunction(mbs, t, itemNumber, axialPositionNormalized, curvature, curvature_t, curvatureRef, physicsBendingStiffness, physicsBendingDamping, axialStrain, axialStrain_t, axialStrainRef)

A user function, which computes the bending moment depending on time, strains and curvatures and object parameters (stiffness, damping). The object variables are provided to the function using the current values of the ANCFCable2D object. Note that itemNumber represents the index of the object in mbs, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`. **NOTE:** this function has a different interface as compared to the axial force function.

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs to which object belongs
`t`	Real	current time in mbs
`itemNumber`	Index	integer number i_N of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
`axialPositionNormalized`	Real	axial position at the cable where the user function is evaluated; range is [0,1]
`curvature`	Real	K
`curvature_t`	Real	$\dot{K}$
`curvatureRef`	Real	$K_0 + f_{\text{ref}} \cdot K_{\text{ref}}$
`physicsBendingStiffness`	Real	as given in object parameters
`physicsBendingDamping`	Real	as given in object parameters
`axialStrain`	Real	ε
`axialStrain_t`	Real	ε_t
`axialStrainRef`	Real	$\varepsilon_0 + f_{\text{ref}} \cdot \varepsilon_{\text{ref}}$
`return value`	Real	scalar value of computed bending moment

User function example:

```
#define some material parameters
rhoA = 100.
EA =   1e7.
EI =   1e5

#example of bending moment user function
def bendingMomentUserFunction(mbs, t, itemNumber, axialPositionNormalized,
            curvature, curvature_t, curvatureRef, physicsBendingStiffness,
            physicsBendingDamping, axialStrain, axialStrain_t, axialStrainRef):
    fact = min(1,t) #runs from 0 to 1
    #change reference curvature of beam over time:
    kappa=(curvature-curvatureRef*fact)
    return physicsBendingStiffness*(kappa) + physicsBendingDamping*curvature_t

def axialForceUserFunction(mbs, t, itemNumber, axialPositionNormalized,
```

```
                axialStrain, axialStrain_t, axialStrainRef, physicsAxialStiffness,
                physicsAxialDamping, curvature, curvature_t, curvatureRef):
        fact = min(1,t) #runs from 0 to 1
        return (physicsAxialStiffness*(axialStrain-fact*axialStrainRef) +
                physicsAxialDamping*axialStrain_t)

    cable = ObjectANCFCable2D(physicsMassPerLength=rhoA,
                    physicsBendingStiffness=EI,
                    physicsBendingDamping = EI*0.1,
                    physicsAxialStiffness=EA,
                    physicsAxialDamping=EA*0.05,
                    physicsReferenceAxialStrain=0.1, #10% stretch
                    physicsReferenceCurvature=1,     #radius=1
                    bendingMomentUserFunction=bendingMomentUserFunction,
                    axialForceUserFunction=axialForceUserFunction,
                    )
    #use  cable with GenerateStraightLineANCFCable(...)
```

8.4.2.8 MINI EXAMPLE for ObjectANCFCable2D

```
rhoA = 78.
EA = 1000000.
EI = 833.3333333333333
cable = Cable2D(physicsMassPerLength=rhoA,
                physicsBendingStiffness=EI,
                physicsAxialStiffness=EA,
                )

ancf=GenerateStraightLineANCFCable2D(mbs=mbs,
                positionOfNode0=[0,0,0], positionOfNode1=[2,0,0],
                numberOfElements=32, #converged to 4 digits
                cableTemplate=cable, #this defines the beam element properties
                massProportionalLoad = [0,-9.81,0],
                fixedConstraintsNode0 = [1,1,0,1], #add constraints for pos and rot (r'
_y)
                )
lastNode = ancf[0][-1]

#assemble and solve system for default parameters
mbs.Assemble()
mbs.SolveStatic()

#check result
```

```
exudynTestGlobals.testResult = mbs.GetNodeOutput(lastNode, exu.OutputVariableType.
Displacement)[0]
#ux=-0.5013058140308901
```

For examples on ObjectANCFCable2D see Relevant Examples and TestModels with weblink:

- ALEANCFpipe.py (Examples/)
- ANCFcantileverTest.py (Examples/)
- ANCFcantileverTestDyn.py (Examples/)
- ANCFcontactCircle.py (Examples/)
- ANCFcontactCircle2.py (Examples/)
- ANCFmovingRigidbody.py (Examples/)
- ANCFrotatingCable2D.py (Examples/)
- ANCFslidingJoint2D.py (Examples/)
- ANCFslidingJoint2Drigid.py (Examples/)
- ANCFswitchingSlidingJoint2D.py (Examples/)
- ANCFtestHalfcircle.py (Examples/)
- ANCFtests2.py (Examples/)
- ...
- ANCFcontactCircleTest.py (TestModels/)
- ANCFcontactFrictionTest.py (TestModels/)
- computeODE2EigenvaluesTest.py (TestModels/)
- ...

8.4.3 ObjectALEANCFCable2D

A 2D cable finite element using 2 nodes of type NodePoint2DSlope1 and a axially moving coordinate of type NodeGenericODE2, which adds additional (redundant) motion in axial direction of the beam. This allows modeling pipes but also axially moving beams. The localPosition of the beam with length L=physicsLength and height h ranges in X-direction in range $[0, L]$ and in Y-direction in range $[-h/2, h/2]$ (which is in fact not needed in the EOM).

Additional information for ObjectALEANCFCable2D:

- Requested `Node` type: read detailed information of item
- **Short name** for Python = `ALECable2D`
- **Short name** for Python visualization object = `VALECable2D`

The item **ObjectALEANCFCable2D** with type = 'ALEANCFCable2D' has the following parameters:

Name	type	size	default value	description
name	String		''	objects's unique name
physicsLength	UReal		0.	[SI:m] reference length of beam; such that the total volume (e.g. for volume load) gives ρAL; must be positive
physicsMassPerLength	UReal		0.	[SI:kg/m] total mass per length of beam (including axially moving parts / fluid)
physicsMovingMassFactor	UReal		1.	this factor denotes the amount of ρA which is moving; physicsMovingMassFactor=1 means, that all mass is moving; physicsMovingMassFactor=0 means, that no mass is moving; factor can be used to simulate e.g. pipe conveying fluid, in which ρA is the mass of the pipe+fluid, while $physicsMovingMassFactor \cdot \rho A$ is the mass per unit length of the fluid
physicsBendingStiffness	UReal		0.	[SI:Nm^2] bending stiffness of beam; the bending moment is $m = EI(\kappa - \kappa_0)$, in which κ is the material measure of curvature
physicsAxialStiffness	UReal		0.	[SI:N] axial stiffness of beam; the axial force is $f_{ax} = EA(\varepsilon - \varepsilon_0)$, in which $\varepsilon = \|\mathbf{r}'\| - 1$ is the axial strain
physicsBendingDamping	UReal		0.	[SI:Nm^2/s] bending damping of beam ; the additional virtual work due to damping is $\delta W_{\dot\kappa} = \int_0^L \dot\kappa \delta\kappa dx$
physicsAxialDamping	UReal		0.	[SI:N/s] axial damping of beam; the additional virtual work due to damping is $\delta W_{\dot\varepsilon} = \int_0^L \dot\varepsilon \delta\varepsilon dx$

physicsReferenceAxialStrain	Real		0.	[SI:1] reference axial strain of beam (pre-deformation) of beam; without external loading the beam will statically keep the reference axial strain value
physicsReferenceCurvature	Real		0.	[SI:1/m] reference curvature of beam (pre-deformation) of beam; without external loading the beam will statically keep the reference curvature value
physicsUseCouplingTerms	Bool		True	true: correct case, where all coupling terms due to moving mass are respected; false: only include constant mass for ALE node coordinate, but deactivate other coupling terms (behaves like ANCFCable2D then)
physicsAddALEvariation	Bool		True	true: correct case, where additional terms related to variation of strain and curvature are added
nodeNumbers	NodeIndex3		[invalid [-1], invalid [-1], invalid [-1]]	two node numbers ANCF cable element, third node=ALE GenericODE2 node
useReducedOrderIntegration	Index		0	0/false: use Gauss order 9 integration for virtual work of axial forces, order 5 for virtual work of bending moments; 1/true: use Gauss order 7 integration for virtual work of axial forces, order 3 for virtual work of bending moments
strainIsRelativeToReference	Real		0.	if set to 1., a pre-deformed reference configuration is considered as the stressless state; if set to 0., the straight configuration plus the values of ε_0 and κ_0 serve as a reference geometry; allows also values between 0. and 1.
visualization	VObjectALEANCFCable2D			parameters for visualization of item

The item VObjectALEANCFCable2D has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawHeight	float		0.	if beam is drawn with rectangular shape, this is the drawing height
color	Float4		[-1.,-1.,-1.,-1.]	RGBA color of the object; if R==-1, use default color

8.4.3.1 DESCRIPTION of ObjectALEANCFCable2D:

Information on input parameters:

input parameter	symbol	description see tables above
physicsLength	L	
physicsMassPerLength	ρA	
physicsBendingStiffness	EI	
physicsAxialStiffness	EA	
physicsBendingDamping	d_K	
physicsAxialDamping	d_ε	
physicsReferenceAxialStrain	ε_0	
physicsReferenceCurvature	κ_0	
strainIsRelativeToReference	f_{ref}	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position		global position vector of local position (in X/Y beam coordinates)
Displacement		global displacement vector of local position
Velocity		global velocity vector of local position
VelocityLocal		local velocity vector of local position
Rotation		(scalar) rotation angle of axial slope vector (relative to global x-axis)
Director1		(axial) slope vector of local axis position (at Y=0)
StrainLocal	ε	axial strain (scalar) of local axis position (at Y=0)
CurvatureLocal	K	axial strain (scalar)
ForceLocal	N	(local) section normal force (scalar, including reference strains) (at Y=0); note that strains are highly inaccurate when coupled to bending, thus consider useReducedOrderIntegration=2 and evaluate axial strain at nodes or at midpoint
TorqueLocal	M	(local) bending moment (scalar) (at Y=0)

A 2D cable finite element using 2 nodes of type NodePoint2DSlope1 and an axially moving coordinate of type NodeGenericODE2. The element has 8+1 coordinates and uses cubic polynomials for position interpolation. In addition to ANCFCable2D the element adds an Eulerian axial velocity by the GenericODE2 coordiante. The parameter `physicsMovingMassFactor` allows to control the amount of mass, which moves with the Eulerian velocity (e.g., the fluid), and which is not moving (the pipe). A factor of `physicsMovingMassFactor=1` gives an axially moving beam.

The Bernoulli-Euler beam is capable of large deformation as it employs the material measure of curvature for the bending. Note that damping (physicsBendingDamping, physicsAxialDamping) only acts on the non-moving part of the beam, as it is the case for the pipe.

Note that most functions act on the underlying cable finite element, which is not co-moving axially. E.g., if you apply constraints to the nodal coordinates, the cable can be fixed, while still the axial component is freely moving. If you apply a LoadForce using a MarkerPosition, the force is acting on the beam finite element, but not on the axially moving coordinate. In contrast to the latter, the ObjectJointALEMoving2D and the MarkerBodyMass are acting on the moving coordinate as well.

A detailed paper on this element is yet under submission, but a similar formulation can be found in [47] and the underlying beam element is identical to ObjectANCFCable2D.

For examples on ObjectALEANCFCable2D see Relevant Examples and TestModels with weblink:

- `ALEANCFpipe.py` (Examples/)
- `ANCFALEtest.py` (Examples/)
- `ANCFmovingRigidbody.py` (Examples/)
- `flexiblePendulumANCF.py` (Examples/)
- `ANCFoutputTest.py` (TestModels/)

8.4.4 ObjectANCFBeam

OBJECT UNDER CONSTRUCTION: A 3D beam finite element based on the absolute nodal coordinate formulation, using two nodes. The localPosition x of the beam ranges from $-L/2$ (at node 0) to $L/2$ (at node 1). The axial coordinate is x (first coordinate) and the cross section is spanned by local y/z axes; assuming dimensions w_y and w_z in cross section, the local position range is $\in [[-L/2, L/2], [-wy/2, wy/2], [-wz/2, wz/2]]$.

Additional information for ObjectANCFBeam:

- This `Object` has/provides the following types = `Body`, `MultiNoded`
- Requested `Node` type = `Position` + `Orientation`
- **Short name** for Python = `ANCFBeam`
- **Short name** for Python visualization object = `VANCFBeam`

The item **ObjectANCFBeam** with type = 'ANCFBeam' has the following parameters:

Name	type	size	default value	description
name	String		''	objects's unique name
nodeNumbers	NodeIndex2		[invalid [-1], invalid [-1]]	two node numbers for beam element
physicsLength	PReal		0.	[SI:m] reference length of beam; such that the total volume (e.g. for volume load) gives ρAL; must be positive
sectionData	BeamSection		BeamSection()	data as given by exudyn.BeamSection(), defining inertial, stiffness and damping parameters of beam section.
crossSectionPenaltyFactor	Vector3D		[1.,1.,1.]	[SI:1] additional penalty factors for cross section deformation, which are in total $k_{cs} = [f_{yy} \cdot k_{yy}, f_{zz} \cdot k_{zz}, f_{yz} \cdot k_{yz}]^T$
visualization	VObjectANCFBeam			parameters for visualization of item

The item VObjectANCFBeam has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown; geometry is defined by sectionGeometry
sectionGeometry	BeamSectionGeometry		BeamSectionGeometry()	defines cross section shape used for visualization and contact

color	Float4		[-1.,-1.,-1.,-1.]	RGBA color of the object; if R==-1, use default color

---

8.4.4.1 DESCRIPTION of ObjectANCFBeam:

Information on input parameters:

input parameter	symbol	description see tables above
physicsLength	L	
crossSectionPenaltyFactor	$f_{cs} = [f_{yy}, f_{zz}, f_{yz}]^T$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position		global position vector of local position vector
Displacement		global displacement vector of local position vector
Velocity		global velocity vector of local position vector
VelocityLocal		global velocity vector of local position vector
AngularVelocity		global angular velocity vector of local (axis) position vector
AngularVelocityLocal		local angular velocity vector of local (axis) position vector
Acceleration		global acceleration vector of local position vector
Rotation		3D Tait-Bryan rotation components of cross section rotation
RotationMatrix		rotation matrix of cross section rotation as 9D vector

Detailed description coming later.

For examples on ObjectANCFBeam see Relevant Examples and TestModels with weblink:

- `ANCFBeamEigTest.py` (TestModels/)
- `ANCFBeamTest.py` (TestModels/)
- `geometricallyExactBeamTest.py` (TestModels/)
- `rightAngleFrame.py` (TestModels/)

8.4.5 ObjectBeamGeometricallyExact2D

A 2D geometrically exact beam finite element, currently using 2 nodes of type NodeRigidBody2D; FURTHER TESTS REQUIRED. Note that the orientation of the nodes need to follow the cross section orientation in case that includeReferenceRotations=True; e.g., an angle 0 represents the cross section aligned with the y-axis, while and angle $\pi/2$ means that the cross section points in negative x-direction. Pre-curvature can be included with physicsReferenceCurvature and axial pre-stress can be considered by using a physicsLength different from the reference configuration of the nodes. The localPosition of the beam with length L=physicsLength and height h ranges in X-direction in range $[-L/2, L/2]$ and in Y-direction in range $[-h/2, h/2]$ (which is in fact not needed in the EOM).

Additional information for ObjectBeamGeometricallyExact2D:

- This `Object` has/provides the following types = `Body`, `MultiNoded`
- Requested `Node` type = `Position2D` + `Orientation2D` + `Position` + `Orientation`
- **Short name** for Python = `Beam2D`
- **Short name** for Python visualization object = `VBeam2D`

The item **ObjectBeamGeometricallyExact2D** with type = 'BeamGeometricallyExact2D' has the following parameters:

Name	type	size	default value	description
name	String		''	objects's unique name
nodeNumbers	NodeIndex2		[invalid [-1], invalid [-1]]	two node numbers for beam element
physicsLength	UReal		0.	[SI:m] reference length of beam; such that the total volume (e.g. for volume load) gives ρAL; must be positive
physicsMassPerLength	UReal		0.	[SI:kg/m] mass per length of beam
physicsCrossSectionInertia	UReal		0.	[SI:kg m] cross section mass moment of inertia; inertia acting against rotation of cross section
physicsBendingStiffness	UReal		0.	[SI:Nm^2] bending stiffness of beam; the bending moment is $m = EI(\kappa - \kappa_0)$, in which κ is the material measure of curvature
physicsAxialStiffness	UReal		0.	[SI:N] axial stiffness of beam; the axial force is $f_{ax} = EA(\varepsilon - \varepsilon_0)$, in which ε is the axial strain
physicsShearStiffness	UReal		0.	[SI:N] effective shear stiffness of beam, including stiffness correction
physicsBendingDamping	UReal		0.	[SI:Nm^2/s] viscous damping of bending deformation; the additional virtual work due to damping is $\delta W_\kappa = \int_0^L \dot{\kappa}\delta\kappa dx$
physicsAxialDamping	UReal		0.	[SI:N/s] viscous damping of axial deformation

physicsShearDamping	UReal		0.	[SI:N/s] viscous damping of shear deformation
physicsReferenceCurvature	Real		0.	[SI:1/m] reference curvature of beam (predeformation) of beam
includeReferenceRotations	bool		False	if True, rotations at nodes consider reference rotations, which are used for the computation of bending strains (this means that a pre-curved beam is stress-free); if False, the reference rotation of the cross section is orthogonal to the direction between the reference position of the end nodes. This allows to easily share nodes among several beams with different cross section orientation.
visualization	VObjectBeamGeometricallyExact2D			parameters for visualization of item

The item VObjectBeamGeometricallyExact2D has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawHeight	float		0.	if beam is drawn with rectangular shape, this is the drawing height
color	Float4		[-1.,-1.,-1.,-1.]	RGBA color of the object; if R==-1, use default color

---

8.4.5.1 DESCRIPTION of ObjectBeamGeometricallyExact2D:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
physicsLength	L	
physicsMassPerLength	ρA	
physicsCrossSectionInertia	ρJ	
physicsBendingStiffness	EI	
physicsAxialStiffness	EA	
physicsShearStiffness	GA	
physicsBendingDamping	d_K	
physicsAxialDamping	d_ε	
physicsShearDamping	d_γ	
physicsReferenceCurvature	κ_0	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position		global position vector of local axis (X) and cross section (Y) position
Displacement		global displacement vector of local axis (X) and cross section (Y) position
Velocity		global velocity vector of local axis (X) and cross section (Y) position
Rotation		3D Tait-Bryan rotation components, containing rotation around Z-axis only
StrainLocal		6 (local) strain components, containing only axial (XX, index 0) and shear strain (XY, index 5); evaluated at beam axis ONLY
CurvatureLocal		3D vector of (local) curvature, only Z component is non-zero
ForceLocal		3D vector of (local) section normal force, containing axial (X) and shear force (Y)
TorqueLocal		3D vector of (local) torques, containing only bending moment (Z)

See paper of Simo and Vu-Quoc (1986). Detailed description coming later.

For examples on ObjectBeamGeometricallyExact2D see Relevant Examples and TestModels with weblink:

- `pendulumGeomExactBeam2D.py` (Examples/)
- `ANCFBeamEigTest.py` (TestModels/)
- `geometricallyExactBeam2Dtest.py` (TestModels/)
- `gridGeomExactBeam2D.py` (TestModels/)
- `LShapeGeomExactBeam2D.py` (TestModels/)

8.4.6 ObjectBeamGeometricallyExact

OBJECT UNDER CONSTRUCTION: A 3D geometrically exact beam finite element, currently using two 3D rigid body nodes. The localPosition x of the beam ranges from $-L/2$ (at node 0) to $L/2$ (at node 1). The axial coordinate is x (first coordinate) and the cross section is spanned by local y/z axes.

Additional information for ObjectBeamGeometricallyExact:

- This `Object` has/provides the following types = `Body`, `MultiNoded`
- Requested `Node` type = `Position` + `Orientation`
- **Short name** for Python = `Beam3D`
- **Short name** for Python visualization object = `VBeam3D`

The item **ObjectBeamGeometricallyExact** with type = 'BeamGeometricallyExact' has the following parameters:

Name	type	size	default value	description
name	String		''	objects's unique name
nodeNumbers	NodeIndex2		[invalid [-1], invalid [-1]]	two node numbers for beam element
physicsLength	PReal		0.	[SI:m] reference length of beam; such that the total volume (e.g. for volume load) gives ρAL; must be positive
sectionData	BeamSection		BeamSection()	data as given by exudyn.BeamSection(), defining inertial, stiffness and damping parameters of beam section.
visualization	VObjectBeamGeometricallyExact			parameters for visualization of item

The item VObjectBeamGeometricallyExact has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown; geometry is defined by sectionGeometry
sectionGeometry	BeamSectionGeometry		BeamSectionGeometry()	defines cross section shape used for visualization and contact
color	Float4		[-1.,-1.,-1.,-1.]	RGBA color of the object; if R==-1, use default color

8.4.6.1 DESCRIPTION of ObjectBeamGeometricallyExact:

Information on input parameters:

input parameter	symbol	description see tables above
physicsLength	L	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position		global position vector of local axis (1) and cross section (2) position
Displacement		global displacement vector of local axis (1) and cross section (2) position
Velocity		global velocity vector of local axis (1) and cross section (2) position
Rotation		3D Tait-Bryan rotation components, containing rotation around z-axis only
StrainLocal		6 strain components, containing only axial (xx) and shear strain (xy)
CurvatureLocal		3D vector of curvature, containing only curvature w.r.t. z-axis

Detailed description coming later.

For examples on ObjectBeamGeometricallyExact see Relevant Examples and TestModels with weblink:

- `geometricallyExactBeamTest.py` (TestModels/)
- `rightAngleFrame.py` (TestModels/)

8.4.7 ObjectANCFThinPlate

A 3D thin Kirchhoff plate finite element based on the absolute nodal coordinate formulation, using 4 nodes of type NodePointSlope12. The geometry as well as (deformed and distorted) reference configuration is given by the nodes. The localPosition follows unit-coordinates in the range [-1,1] for X, Y and Z coordinates; the thickness of the plate is h; This element is under construction.

Additional information for ObjectANCFThinPlate:

- This `Object` has/provides the following types = `Body`, `MultiNoded`
- Requested `Node` type = `Position`

The item **ObjectANCFThinPlate** with type = 'ANCFThinPlate' has the following parameters:

Name	type	size	default value	description
name	String		''	objects's unique name
physicsThickness	UReal		0.	[SI:m] thickness of plate
physicsDensity	UReal		0.	[SI:kg/m^3] density of the plate, possibly averaged over thickness
physicsStrainCoefficients	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	[SI:N/m] stiffness coefficients related to in-plane normal and shear strains, integrated over height of the plate
physicsCurvatureCoefficients	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	[SI:Nm] stiffness coefficients related to curvatures, integrated over height of the plate
strainIsRelativeToReference	Real		1.	if set to 1., a pre-deformed reference configuration is considered as the stressless state; if set to 0., the straight configuration serves as a reference geometry; allows also values between 0. and 1. to perform a transition during static computation
nodeNumbers	NodeIndex4		[invalid [-1], invalid [-1], invalid [-1], invalid [-1]]	4 NodePointSlope12 node numbers
useReducedOrderIntegration	Index		0	0/false: use highest Gauss integration for virtual work of strains
visualization	VObjectANCFThinPlate			parameters for visualization of item

The item VObjectANCFThinPlate has the following parameters:

Name	type	size	default value	description

show	Bool		True	set true, if item is shown in visualization and false if it is not shown; note that all quantities are computed at the beam centerline, even if drawn on surface of cylinder of beam; this effects, e.g., Displacement or Velocity, which is drawn constant over cross section
color	Float4		[-1.,-1.,-1.,-1.]	RGBA color of the object; if R==-1, use default color

---

8.4.7.1 DESCRIPTION of ObjectANCFThinPlate:

Information on input parameters:

input parameter	symbol	description see tables above
physicsThickness	h	
physicsDensity	ρ	
physicsStrainCoefficients	$\mathbf{D}_\varepsilon$	
physicsCurvatureCoefficients	$\mathbf{D}_\kappa$	
strainIsRelativeToReference	f_{ref}	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position	${}^0\mathbf{p}_{\text{config}}(x,0,0) = \mathbf{r}_{\text{config}}(x) + y \cdot \mathbf{n}_{\text{config}}(x)$	global position vector of local position $[x,0,0]$
Displacement	${}^0\mathbf{u}_{\text{config}}(x,0,0) = {}^0\mathbf{p}_{\text{config}}(x,0,0) - {}^0\mathbf{p}_{\text{ref}}(x,0,0)$	global displacement vector of local position
Velocity	${}^0\mathbf{v}(x,0,0) = {}^0\dot{\mathbf{r}}(x)$	global velocity vector of local position
Director1	$\mathbf{r}'(x)$	(axial) slope vector of local axis position (at y=0)
StrainLocal	ε	axial strain (scalar) of local axis position (at Y=Z=0)
CurvatureLocal	$[K_x, K_y, K_z]^{\text{T}}$	local curvature vector
ForceLocal	N	(local) section normal force (scalar, including reference strains) (at y=z=0); note that strains are highly inaccurate when coupled to bending, thus consider useReducedOrderIntegration=2 and evaluate axial strain at nodes or at midpoint
TorqueLocal	M	(local) bending moment (scalar) (at y=z=0), which are bending moments as there is no torque
Acceleration	${}^0\mathbf{a}(x,0,0) = {}^0\ddot{\mathbf{r}}(x)$	global acceleration vector of local position

8.4.7.2 MINI EXAMPLE for ObjectANCFThinPlate

```
#to be done

#check result
exudynTestGlobals.testResult = 0
#ux=-0.5013058140308901
```

8.5 Objects (Joint)

A Joint is a special Object, Connector and Constraint, which is attached to position or rigid body markers. The joint results in special algebraic equations and requires implicit time integration. Joints represent special constraints, as described in multibody system dynamics literature.

8.5.1 ObjectJointGeneric

A generic joint in 3D; constrains components of the absolute position and rotations of two points given by PointMarkers or RigidMarkers. An additional local rotation (rotationMarker) can be used to adjust the three rotation axes and/or sliding axes.

Additional information for ObjectJointGeneric:

- This `Object` has/provides the following types = `Connector`, `Constraint`
- Requested `Marker` type = `Position` + `Orientation`
- **Short name** for Python = `GenericJoint`
- **Short name** for Python visualization object = `VGenericJoint`

The item **ObjectJointGeneric** with type = 'JointGeneric' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex	2	[invalid [-1], invalid [-1]]	list of markers used in connector
constrainedAxes	ArrayIndex	6	[1,1,1,1,1,1]	flag, which determines which translation (0,1,2) and rotation (3,4,5) axes are constrained; for j_i, two values are possible: 0=free axis, 1=constrained axis
rotationMarker0	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	local rotation matrix for marker $m0$; translation and rotation axes for marker $m0$ are defined in the local body coordinate system and additionally transformed by rotationMarker0
rotationMarker1	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	local rotation matrix for marker $m1$; translation and rotation axes for marker $m1$ are defined in the local body coordinate system and additionally transformed by rotationMarker1
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint

offsetUserFunctionParameters	Vector6D		[0.,0.,0.,0.,0.,0.]	vector of 6 parameters for joint's offsetUserFunction
offsetUserFunction	PyFunctionVector6DmbsScalarIndexVector6D 0			A Python function which defines the time-dependent (fixed) offset of translation (indices 0,1,2) and rotation (indices 3,4,5) joint coordinates with parameters (mbs, t, offsetUserFunctionParameters)
offsetUserFunction_t	PyFunctionVector6DmbsScalarIndexVector6D 0			(NOT IMPLEMENTED YET)time derivative of offsetUserFunction using the same parameters
alternativeConstraints	Bool		False	this is an experimental flag, may change in future: if uses alternative contraint equations for rotations, currently in case of 3 locked rotations: ${}^0\mathbf{t}_{x0}^{\mathrm{T}}({}^0\mathbf{t}_{y1} \times {}^0\mathbf{t}_{z0})$, ${}^0\mathbf{t}_{y0}^{\mathrm{T}}({}^0\mathbf{t}_{z1} \times {}^0\mathbf{t}_{x0})$, ${}^0\mathbf{t}_{z0}^{\mathrm{T}}({}^0\mathbf{t}_{x1} \times {}^0\mathbf{t}_{y0})$; this avoids 180°flips of the standard configuration in static computations, but leads to different values in Lagrange multipliers
visualization	VObjectJointGeneric			parameters for visualization of item

The item VObjectJointGeneric has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
axesRadius	float		0.1	radius of joint axes to draw
axesLength	float		0.4	length of joint axes to draw
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

---

8.5.1.1 DESCRIPTION of ObjectJointGeneric:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
markerNumbers	$[m0, m1]^{\mathrm{T}}$	
constrainedAxes	$\mathbf{j} = [j_0, \ldots, j_5]$	
rotationMarker0	${}^{m0,J0}\mathbf{A}$	
rotationMarker1	${}^{m1,J1}\mathbf{A}$	

offsetUserFunctionParameters	$\mathbf{p}_{par}$	
offsetUserFunction	UF $\in \mathbb{R}^6$	
offsetUserFunction_t	UF $\in \mathbb{R}^6$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
Position	${}^0\mathbf{p}_{m0}$	current global position of position marker $m0$
Velocity	${}^0\mathbf{v}_{m0}$	current global velocity of position marker $m0$
DisplacementLocal	${}^{J0}\Delta\mathbf{p}$	relative displacement in local joint0 coordinates; uses local J0 coordinates even for spherical joint configuration
VelocityLocal	${}^{J0}\Delta\mathbf{v}$	relative translational velocity in local joint0 coordinates
Rotation	${}^{J0}\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2]^{\mathrm{T}}$	relative rotation parameters (Tait Bryan Rxyz); if all axes are fixed, this output represents the rotational drift; for a revolute joint with free Z-axis, it contains the rotation in the Z-component
AngularVelocityLocal	${}^{J0}\Delta\boldsymbol{\omega}$	relative angular velocity in local joint0 coordinates; if all axes are fixed, this output represents the angular velocity constraint error; for a revolute joint, it contains the angular velocity of this axis
ForceLocal	${}^{J0}\mathbf{f}$	joint force in local $J0$ coordinates
TorqueLocal	${}^{J0}\mathbf{m}$	joint torque in local $J0$ coordinates; depending on joint configuration, the result may not be the according torque vector

8.5.1.2 Definition of quantities

intermediate variables	**symbol**	**description**
marker m0 position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker m0
marker m0 orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker m0
joint J0 orientation	${}^{0,J0}\mathbf{A} = {}^{0,m0}\mathbf{A}\ {}^{m0,J0}\mathbf{A}$	joint $J0$ rotation matrix
joint J0 orientation vectors	${}^{0,J0}\mathbf{A} = [{}^0\mathbf{t}_{x0}, {}^0\mathbf{t}_{y0}, {}^0\mathbf{t}_{z0}]^{\mathrm{T}}$	orientation vectors (represent local x, y, and z axes) in global coordinates, used for definition of constraint equations
marker m1 position	${}^0\mathbf{p}_{m1}$	accordingly
marker m1 orientation	${}^{0,m1}\mathbf{A}$	current rotation matrix provided by marker m1

joint J1 orientation	${}^{0,J1}\mathbf{A} = {}^{0,m1}\mathbf{A}\ {}^{m1,J1}\mathbf{A}$	joint $J1$ rotation matrix
joint J1 orientation vectors	${}^{0,J1}\mathbf{A} = [{}^{0}\mathbf{t}_{x1}, {}^{0}\mathbf{t}_{y1}, {}^{v}\mathbf{t}_{z1}]^{\mathrm{T}}$	orientation vectors (represent local x, y, and z axes) in global coordinates, used for definition of constraint equations
marker m0 velocity	${}^{0}\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
marker m1 velocity	${}^{0}\mathbf{v}_{m1}$	accordingly
marker m0 velocity	${}^{b}\boldsymbol{\omega}_{m0}$	current local angular velocity vector provided by marker m0
marker m1 velocity	${}^{b}\boldsymbol{\omega}_{m1}$	current local angular velocity vector provided by marker m1
Displacement	${}^{0}\Delta\mathbf{p} = {}^{0}\mathbf{p}_{m1} - {}^{0}\mathbf{p}_{m0}$	used, if all translational axes are constrained
Velocity	${}^{0}\Delta\mathbf{v} = {}^{0}\mathbf{v}_{m1} - {}^{0}\mathbf{v}_{m0}$	used, if all translational axes are constrained (velocity level)
DisplacementLocal	${}^{J0}\Delta\mathbf{p}$	$\left({}^{0,m0}\mathbf{A}\ {}^{m0,J0}\mathbf{A}\right)^{\mathrm{T}}\ {}^{0}\Delta\mathbf{p}$
VelocityLocal	${}^{J0}\Delta\mathbf{v}$	$\left({}^{0,m0}\mathbf{A}\ {}^{m0,J0}\mathbf{A}\right)^{\mathrm{T}}\ {}^{0}\Delta\mathbf{v}$ ... note that this is the global relative velocity projected into the local $J0$ coordinate system
AngularVelocityLocal	${}^{J0}\Delta\omega$	$\left({}^{0,m0}\mathbf{A}\ {}^{m0,J0}\mathbf{A}\right)^{\mathrm{T}}\left({}^{0,m1}\mathbf{A}\ {}^{m1}\omega - {}^{0,m0}\mathbf{A}\ {}^{m0}\omega\right)$
algebraic variables	$\mathbf{z} = [\lambda_0, \ldots, \lambda_5]^{\mathrm{T}}$	vector of algebraic variables (Lagrange multipliers) according to the algebraic equations

8.5.1.3 Connector constraint equations

Equations for translational part (`activeConnector = True`) :
If $[j_0, \ldots, j_2] = [1,1,1]^{\mathrm{T}}$, meaning that all translational coordinates are fixed, the translational index 3 constraints read ($UF_{0,1,2}(mbs, t, \mathbf{p}_{par})$ is the translational part of the user function UF),

$$ {}^{0}\mathbf{p}_{m1} - {}^{0}\mathbf{p}_{m0} - UF_{0,1,2}(mbs, t, i_N, \mathbf{p}_{par}) = \mathbf{0} \tag{8.156} $$

and the translational index 2 constraints read

$$ {}^{0}\mathbf{v}_{m1} - {}^{0}\mathbf{v}_{m0} - UF_{t;0,1,2}(mbs, t, i_N, \mathbf{p}_{par}) = \mathbf{0} \tag{8.157} $$

and `iN` represents the itemNumber (=objectNumber). If $[j_0, \ldots, j_2] \neq [1,1,1]^{\mathrm{T}}$, meaning that at least one translational coordinate is free, the translational index 3 constraints read for every component $k \in [0,1,2]$ of the vector ${}^{J0}\Delta\mathbf{p}$

$$ {}^{J0}\Delta p_k - UF_k(mbs, t, i_N, \mathbf{p}_{par}) = 0 \quad \text{if} \quad j_k = 1 \quad \text{and} \tag{8.158} $$

$$ \lambda_k = 0 \quad \text{if} \quad j_k = 0 \tag{8.159} $$

$$ \tag{8.160} $$

and the translational index 2 constraints read for every component $k \in [0,1,2]$ of the vector ${}^{J0}\Delta\mathbf{v}$

$$ {}^{J0}\Delta v_k - UF\_t_k(mbs, t, i_N, \mathbf{p}_{par}) = 0 \quad \text{if} \quad j_k = 1 \quad \text{and} \tag{8.161} $$

$$ \lambda_k = 0 \quad \text{if} \quad j_k = 0 \tag{8.162} $$

$$ \tag{8.163} $$

Equations for rotational part (`activeConnector = True`) :
The following equations are exemplarily for certain constrained rotation axes configurations, which shall represent all other possibilities. Note that the axes are always given in global coordinates, compare the table in Section 8.5.1.2.

Equations are only given for the index 3 case; the index 2 case can be derived from these equations easily (see C++ code...). In case of user functions, the additional rotation matrix ${}^{J0,J0U}\mathbf{A}\,(UF_{3,4,5}(mbs, t, \mathbf{p}_{par}))$, in which the three components of $UF_{3,4,5}$ are interpreted as Tait-Bryan angles that are added to the joint frame.

If **3 rotation axes are constrained** (e.g., translational or planar joint), $[j_3, \ldots, j_5] = [1,1,1]^{\mathrm{T}}$, the index 3 constraint equations read

$$
\begin{aligned}
{}^{0}\mathbf{t}_{z0}^{\mathrm{T}}\,{}^{0}\mathbf{t}_{y1} &= 0 && (8.164)\\
{}^{0}\mathbf{t}_{z0}^{\mathrm{T}}\,{}^{0}\mathbf{t}_{x1} &= 0 && (8.165)\\
{}^{0}\mathbf{t}_{x0}^{\mathrm{T}}\,{}^{0}\mathbf{t}_{y1} &= 0 && (8.166)
\end{aligned}
$$

If **2 rotation axes are constrained** (revolute joint), e.g., $[j_3, \ldots, j_5] = [0,1,1]^{\mathrm{T}}$, the index 3 constraint equations read

$$
\begin{aligned}
\lambda_3 &= 0 && (8.167)\\
{}^{0}\mathbf{t}_{x0}^{\mathrm{T}}\,{}^{0}\mathbf{t}_{y1} &= 0 && (8.168)\\
{}^{0}\mathbf{t}_{x0}^{\mathrm{T}}\,{}^{0}\mathbf{t}_{z1} &= 0 && (8.169)
\end{aligned}
$$

If **1 rotation axis is constrained** (universal joint), e.g., $[j_3, \ldots, j_5] = [1,0,0]^{\mathrm{T}}$, the index 3 constraint equations read

$$
\begin{aligned}
{}^{0}\mathbf{t}_{y0}^{\mathrm{T}}\,{}^{0}\mathbf{t}_{z1} &= 0 && (8.170)\\
\lambda_4 &= 0 && (8.171)\\
\lambda_5 &= 0 && (8.172)
\end{aligned}
$$

if `activeConnector = False`,

$$\mathbf{z} = \mathbf{0} \qquad (8.173)$$

Userfunction: `offsetUserFunction(mbs, t, itemNumber, offsetUserFunctionParameters)`

A user function, which computes scalar offset for relative joint translation and joint rotation for the GenericJoint, e.g., in order to move or rotate a body on a prescribed trajectory. It is NECESSARY to use sufficiently smooth functions, having **initial offsets** consistent with **initial configuration** of bodies, either zero or compatible initial offset-velocity, and no initial accelerations. The `offsetUserFunction` is **ONLY used** in case of static computation or index3 (generalizedAlpha) time integration. In order to be on the safe side, provide both `offsetUserFunction` and `offsetUserFunction_t`.

Note that itemNumber represents the index of the object in mbs, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

The user function gets time and the offsetUserFunctionParameters as an input and returns the computed offset vector for all relative translational and rotational joint coordinates:

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs in which underlying item is defined
`t`	Real	current time in mbs
`itemNumber`	Index	integer number of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
`offsetUserFunctionParameters`	Real	$\mathbf{p}_{par}$, set of parameters which can be freely used in user function
`return value`	Real	computed offset vector for given time

Userfunction: `offsetUserFunction_t(mbs, t, itemNumber, offsetUserFunctionParameters)`

A user function, which computes an offset **velocity** vector for the GenericJoint. It is NECESSARY to use sufficiently smooth functions, having **initial offset velocities** consistent with **initial velocities** of bodies. The `offsetUserFunction_t` is used instead of `offsetUserFunction` in case of `velocityLevel = True`, or for index2 time integration and needed for computation of initial accelerations in second order implicit time integrators.

Note that itemNumber represents the index of the object in mbs, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

The user function gets time and the offsetUserFunctionParameters as an input and returns the computed offset velocity vector for all relative translational and rotational joint coordinates:

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs in which underlying item is defined
`t`	Real	current time in mbs
`itemNumber`	Index	integer number of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
`offsetUserFunctionParameters`	Real	$\mathbf{p}_{par}$, set of parameters which can be freely used in user function
`return value`	Real	computed offset velocity vector for given time

User function example:

```
#simple example, computing only the translational offset for x-coordinate
from math import sin, cos, pi
def UFoffset(mbs, t, itemNumber, offsetUserFunctionParameters):
    return [offsetUserFunctionParameters[0]*(1 - cos(t*10*2*pi)), 0,0,0,0,0]
```

For examples on ObjectJointGeneric see Relevant Examples and TestModels with weblink:

- ballBearningModel.py (Examples/)
- bungeeJump.py (Examples/)
- pistonEngine.py (Examples/)
- universalJoint.py (Examples/)
- ANCFrotatingCable2D.py (Examples/)
- beamTutorial.py (Examples/)
- CMSexampleCourse.py (Examples/)
- craneReevingSystem.py (Examples/)
- fourBarMechanism3D.py (Examples/)
- humanRobotInteraction.py (Examples/)
- leggedRobot.py (Examples/)
- mobileMecanumWheelRobotWithLidar.py (Examples/)
- ...
- driveTrainTest.py (TestModels/)
- revoluteJointPrismaticJointTest.py (TestModels/)
- bricardMechanism.py (TestModels/)
- ...

8.5.2 ObjectJointRevoluteZ

A revolute joint in 3D; constrains the position of two rigid body markers and the rotation about two axes, while the joint z-rotation axis (defined in local coordinates of marker 0 / joint J0 coordinates) can freely rotate. An additional local rotation (rotationMarker) can be used to transform the markers' coordinate systems into the joint coordinate system. For easier definition of the joint, use the exudyn.rigidbodyUtilities function AddRevoluteJoint(...), Section 7.19, for two rigid bodies (or ground).

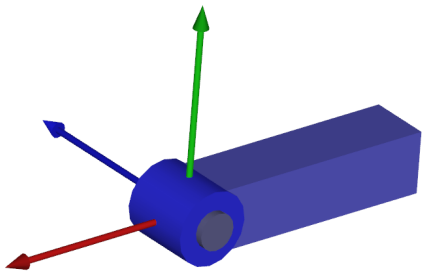


Figure 8.3: Example of RevoluteJointZ

Additional information for ObjectJointRevoluteZ:

- This `Object` has/provides the following types = `Connector`, `Constraint`
- Requested `Marker` type = `Position` + `Orientation`
- **Short name** for Python = `RevoluteJointZ`
- **Short name** for Python visualization object = `VRevoluteJointZ`

The item **ObjectJointRevoluteZ** with type = 'JointRevoluteZ' has the following parameters:

Name	type	size	default value	description
name	String		''	constraints's unique name
markerNumbers	ArrayMarkerIndex	2	[invalid [-1], invalid [-1]]	list of markers used in connector
rotationMarker0	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	local rotation matrix for marker $m0$; translation and rotation axes for marker $m0$ are defined in the local body coordinate system and additionally transformed by rotationMarker0

rotationMarker1	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	local rotation matrix for marker $m1$; translation and rotation axes for marker $m1$ are defined in the local body coordinate system and additionally transformed by rotationMarker1
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectJointRevoluteZ			parameters for visualization of item

The item VObjectJointRevoluteZ has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
axisRadius	float		0.1	radius of joint axis to draw
axisLength	float		0.4	length of joint axis to draw
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

---

8.5.2.1 DESCRIPTION of ObjectJointRevoluteZ:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
markerNumbers	$[m0, m1]^{\mathrm{T}}$	
rotationMarker0	${}^{m0,J0}\mathbf{A}$	
rotationMarker1	${}^{m1,J1}\mathbf{A}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
Position	${}^{0}\mathbf{p}_{m0}$	current global position of position marker $m0$
Velocity	${}^{0}\mathbf{v}_{m0}$	current global velocity of position marker $m0$
DisplacementLocal	${}^{J0}\Delta\mathbf{p}$	relative displacement in local joint0 coordinates; uses local J0 coordinates even for spherical joint configuration

VelocityLocal	${}^{J0}\Delta\mathbf{v}$	relative translational velocity in local joint0 coordinates
Rotation	${}^{J0}\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2]^{\mathrm{T}}$	relative rotation parameters (Tait Bryan Rxyz); Z component represents rotation of joint, other components represent constraint drift
AngularVelocityLocal	${}^{J0}\Delta\boldsymbol{\omega}$	relative angular velocity in joint J0 coordinates, giving a vector with Z-component only
ForceLocal	${}^{J0}\mathbf{f}$	joint force in local $J0$ coordinates
TorqueLocal	${}^{J0}\mathbf{m}$	joint torques in local $J0$ coordinates; torque around Z is zero

8.5.2.2 Definition of quantities

intermediate variables	**symbol**	**description**
marker m0 position	${}^{0}\mathbf{p}_{m0}$	current global position which is provided by marker m0
marker m0 orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker m0
joint J0 orientation	${}^{0,J0}\mathbf{A} = {}^{0,m0}\mathbf{A}\ {}^{m0,J0}\mathbf{A}$	joint $J0$ rotation matrix
joint J0 orientation vectors	${}^{0,J0}\mathbf{A} = [{}^{0}\mathbf{t}_{x0}, {}^{0}\mathbf{t}_{y0}, {}^{0}\mathbf{t}_{z0}]^{\mathrm{T}}$	orientation vectors (represent local x, y, and z axes) in global coordinates, used for definition of constraint equations
marker m1 position	${}^{0}\mathbf{p}_{m1}$	accordingly
marker m1 orientation	${}^{0,m1}\mathbf{A}$	current rotation matrix provided by marker m1
joint J1 orientation	${}^{0,J1}\mathbf{A} = {}^{0,m1}\mathbf{A}\ {}^{m1,J1}\mathbf{A}$	joint $J1$ rotation matrix
joint J1 orientation vectors	${}^{0,J1}\mathbf{A} = [{}^{0}\mathbf{t}_{x1}, {}^{0}\mathbf{t}_{y1}, {}^{0}\mathbf{t}_{z1}]^{\mathrm{T}}$	orientation vectors (represent local x, y, and z axes) in global coordinates, used for definition of constraint equations
marker m0 velocity	${}^{0}\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
marker m1 velocity	${}^{0}\mathbf{v}_{m1}$	accordingly
marker m0 velocity	${}^{b}\boldsymbol{\omega}_{m0}$	current local angular velocity vector provided by marker m0
marker m1 velocity	${}^{b}\boldsymbol{\omega}_{m1}$	current local angular velocity vector provided by marker m1
Displacement	${}^{0}\Delta\mathbf{p} = {}^{0}\mathbf{p}_{m1} - {}^{0}\mathbf{p}_{m0}$	used, if all translational axes are constrained
Velocity	${}^{0}\Delta\mathbf{v} = {}^{0}\mathbf{v}_{m1} - {}^{0}\mathbf{v}_{m0}$	used, if all translational axes are constrained (velocity level)
DisplacementLocal	${}^{J0}\Delta\mathbf{p}$	$\left({}^{0,m0}\mathbf{A}\ {}^{m0,J0}\mathbf{A}\right)^{\mathrm{T}}\ {}^{0}\Delta\mathbf{p}$
VelocityLocal	${}^{J0}\Delta\mathbf{v}$	$\left({}^{0,m0}\mathbf{A}\ {}^{m0,J0}\mathbf{A}\right)^{\mathrm{T}}\ {}^{0}\Delta\mathbf{v}$ ... note that this is the global relative velocity projected into the local $J0$ coordinate system

AngularVelocityLocal	${}^{J0}\Delta\omega$	$\left({}^{0,m0}\mathbf{A}\ {}^{m0,J0}\mathbf{A}\right)^{\mathrm{T}}\left({}^{0,m1}\mathbf{A}\ {}^{m1}\omega - {}^{0,m0}\mathbf{A}\ {}^{m0}\omega\right)$
algebraic variables	$\mathbf{z} = [\lambda_0, \ldots, \lambda_5]^{\mathrm{T}}$	vector of algebraic variables (Lagrange multipliers) according to the algebraic equations

8.5.2.3 Connector constraint equations

Equations for translational part (`activeConnector = True`) :
The translational index 3 constraints read,

$$ {}^{0}\Delta\mathbf{p} = \mathbf{0} \tag{8.174}$$

and the translational index 2 constraints read

$$ {}^{0}\Delta\mathbf{v} = \mathbf{0} \tag{8.175}$$

Equations for rotational part (`activeConnector = True`) :
Note that the axes are always given in global coordinates, compare the table in Section 8.5.2.2, and they include the transformations by ${}^{m0,J0}\mathbf{A}$ and ${}^{m1,J1}\mathbf{A}$. The index 3 constraint equations read

$$ {}^{0}\mathbf{t}_{z0}^{\mathrm{T}}\ {}^{0}\mathbf{t}_{x1} = 0 \tag{8.176}$$

$$ {}^{0}\mathbf{t}_{z0}^{\mathrm{T}}\ {}^{0}\mathbf{t}_{y1} = 0 \tag{8.177}$$

The index 2 constraints follow from the derivative of Eq. (8.176) w.r.t. time, and are given in the C++ code. if `activeConnector = False`,

$$ \mathbf{z} = \mathbf{0} \tag{8.178}$$

8.5.2.4 MINI EXAMPLE for ObjectJointRevoluteZ

```
#example with rigid body at [0,0,0], with torsional load
nBody = mbs.AddNode(RigidRxyz())
oBody = mbs.AddObject(RigidBody(physicsMass=1, physicsInertia=[1,1,1,0,0,0],
                                nodeNumber=nBody))

mBody = mbs.AddMarker(MarkerNodeRigid(nodeNumber=nBody))
mGround = mbs.AddMarker(MarkerBodyRigid(bodyNumber=oGround,
                                        localPosition = [0,0,0]))
mbs.AddObject(RevoluteJointZ(markerNumbers = [mGround, mBody])) #rotation around ground
Z-axis

#torque around z-axis;
mbs.AddLoad(Torque(markerNumber = mBody, loadVector=[0,0,1]))
```

```
#assemble and solve system for default parameters
mbs.Assemble()
mbs.SolveDynamic(exu.SimulationSettings())

#check result at default integration time
exudynTestGlobals.testResult = mbs.GetNodeOutput(nBody, exu.OutputVariableType.Rotation)
[2]
```

For examples on ObjectJointRevoluteZ see Relevant Examples and TestModels with weblink:

- addRevoluteJoint.py (Examples/)
- rigidBodyTutorial3withMarkers.py (Examples/)
- ballBearningModel.py (Examples/)
- bicycleIftommBenchmark.py (Examples/)
- chatGPTupdate.py (Examples/)
- chatGPTupdate2.py (Examples/)
- multiMbsTest.py (Examples/)
- openVRengine.py (Examples/)
- pistonEngine.py (Examples/)
- rigidBodyTutorial3.py (Examples/)
- rollerBearningModel.py (Examples/)
- solutionViewerTest.py (Examples/)
- ...
- plotSensorTest.py (TestModels/)
- revoluteJointPrismaticJointTest.py (TestModels/)
- bricardMechanism.py (TestModels/)
- ...

8.5.3 ObjectJointPrismaticX

A prismatic joint in 3D; constrains the relative rotation of two rigid body markers and relative motion w.r.t. the joint y and z axes, allowing a relative motion along the joint x axis (defined in local coordinates of marker 0 / joint J0 coordinates). An additional local rotation (rotationMarker) can be used to transform the markers' coordinate systems into the joint coordinate system. For easier definition of the joint, use the exudyn.rigidbodyUtilities function AddPrismaticJoint(...), Section 7.19, for two rigid bodies (or ground).

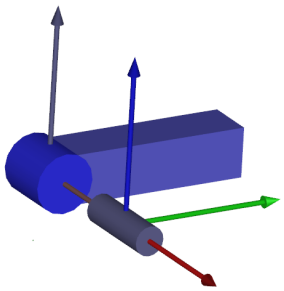


Figure 8.4: Example of PrismaticJointX

Additional information for ObjectJointPrismaticX:

- This `Object` has/provides the following types = `Connector`, `Constraint`
- Requested `Marker` type = `Position` + `Orientation`
- **Short name** for Python = `PrismaticJointX`
- **Short name** for Python visualization object = `VPrismaticJointX`

The item **ObjectJointPrismaticX** with type = 'JointPrismaticX' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex	2	[invalid [-1], invalid [-1]]	list of markers used in connector
rotationMarker0	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	local rotation matrix for marker $m0$; translation and rotation axes for marker $m0$ are defined in the local body coordinate system and additionally transformed by rotationMarker0

rotationMarker1	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	local rotation matrix for marker $m1$; translation and rotation axes for marker $m1$ are defined in the local body coordinate system and additionally transformed by rotationMarker1
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectJointPrismaticX			parameters for visualization of item

The item VObjectJointPrismaticX has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
axisRadius	float		0.1	radius of joint axis to draw
axisLength	float		0.4	length of joint axis to draw
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

---

8.5.3.1 DESCRIPTION of ObjectJointPrismaticX:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
markerNumbers	$[m0, m1]^{\mathrm{T}}$	
rotationMarker0	${}^{m0,J0}\mathbf{A}$	
rotationMarker1	${}^{m1,J1}\mathbf{A}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
Position	${}^{0}\mathbf{p}_{m0}$	current global position of position marker $m0$
Velocity	${}^{0}\mathbf{v}_{m0}$	current global velocity of position marker $m0$
DisplacementLocal	${}^{J0}\Delta\mathbf{p}$	relative displacement in local joint0 coordinates; uses local J0 coordinates even for spherical joint configuration

VelocityLocal	${}^{J0}\Delta\mathbf{v}$	relative translational velocity in local joint0 coordinates
Rotation	${}^{J0}\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2]^{\mathrm{T}}$	relative rotation parameters (Tait Bryan Rxyz); if all axes are fixed, this output represents the rotational drift; for a revolute joint, it contains the rotation of this axis
AngularVelocityLocal	${}^{J0}\Delta\boldsymbol{\omega}$	relative angular velocity in local joint0 coordinates; if all axes are fixed, this output represents the angular velocity constraint error; for a revolute joint, it contains the angular velocity of this axis
ForceLocal	${}^{J0}\mathbf{f}$	joint force in local $J0$ coordinates
TorqueLocal	${}^{J0}\mathbf{m}$	joint torque in local $J0$ coordinates; depending on joint configuration, the result may not be the according torque vector

8.5.3.2 Definition of quantities

intermediate variables	**symbol**	**description**
marker m0 position	${}^{0}\mathbf{p}_{m0}$	current global position which is provided by marker m0
marker m0 orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker m0
joint J0 orientation	${}^{0,J0}\mathbf{A} = {}^{0,m0}\mathbf{A}\ {}^{m0,J0}\mathbf{A}$	joint $J0$ rotation matrix
joint J0 orientation vectors	${}^{0,J0}\mathbf{A} = [{}^{0}\mathbf{t}_{x0}, {}^{0}\mathbf{t}_{y0}, {}^{0}\mathbf{t}_{z0}]^{\mathrm{T}}$	orientation vectors (represent local x, y, and z axes) in global coordinates, used for definition of constraint equations
marker m1 position	${}^{0}\mathbf{p}_{m1}$	accordingly
marker m1 orientation	${}^{0,m1}\mathbf{A}$	current rotation matrix provided by marker m1
joint J1 orientation	${}^{0,J1}\mathbf{A} = {}^{0,m1}\mathbf{A}\ {}^{m1,J1}\mathbf{A}$	joint $J1$ rotation matrix
joint J1 orientation vectors	${}^{0,J1}\mathbf{A} = [{}^{0}\mathbf{t}_{x1}, {}^{0}\mathbf{t}_{y1}, v\mathbf{t}_{z1}]^{\mathrm{T}}$	orientation vectors (represent local x, y, and z axes) in global coordinates, used for definition of constraint equations
marker m0 velocity	${}^{0}\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
marker m1 velocity	${}^{0}\mathbf{v}_{m1}$	accordingly
marker m0 velocity	${}^{b}\boldsymbol{\omega}_{m0}$	current local angular velocity vector provided by marker m0
marker m1 velocity	${}^{b}\boldsymbol{\omega}_{m1}$	current local angular velocity vector provided by marker m1
Displacement	${}^{0}\Delta\mathbf{p} = {}^{0}\mathbf{p}_{m1} - {}^{0}\mathbf{p}_{m0}$	used, if all translational axes are constrained
DisplacementLocal	${}^{J0}\Delta\mathbf{p}$	$\left({}^{0,m0}\mathbf{A}\ {}^{m0,J0}\mathbf{A}\right)^{\mathrm{T}}\ {}^{0}\Delta\mathbf{p}$

VelocityLocal	${}^{J0}\Delta\mathbf{v}$	$\left({}^{0,m0}\mathbf{A}\ {}^{m0,J0}\mathbf{A}\right)^{\mathrm{T}}\ {}^{0}\Delta\mathbf{v}$ ... note that this is the global relative velocity projected into the local $J0$ coordinate system
AngularVelocityLocal	${}^{J0}\Delta\omega$	$\left({}^{0,m0}\mathbf{A}\ {}^{m0,J0}\mathbf{A}\right)^{\mathrm{T}}\left({}^{0,m1}\mathbf{A}\ {}^{m1}\omega - {}^{0,m0}\mathbf{A}\ {}^{m0}\omega\right)$
algebraic variables	$\mathbf{z} = [\lambda_0, \ldots, \lambda_5]^{\mathrm{T}}$	vector of algebraic variables (Lagrange multipliers) according to the algebraic equations

8.5.3.3 Connector constraint equations

Equations for translational part (`activeConnector = True`) :
The two translational index 3 constraints for a free motion along the local x-axis read (in the coordinate system $J0$),

$$\begin{aligned} {}^{J0}\mathbf{p}_{y,m1} - {}^{J0}\mathbf{p}_{y,m0} &= \mathbf{0} \\ {}^{J0}\mathbf{p}_{z,m1} - {}^{J0}\mathbf{p}_{z,m0} &= \mathbf{0} \end{aligned} \tag{8.179}$$

and the translational index 2 constraints read

$$\begin{aligned} {}^{J0}\mathbf{v}_{y,m1} - {}^{J0}\mathbf{v}_{y,m0} &= \mathbf{0} \\ {}^{J0}\mathbf{v}_{z,m1} - {}^{J0}\mathbf{v}_{z,m0} &= \mathbf{0} \end{aligned} \tag{8.180}$$

Equations for rotational part (`activeConnector = True`) :
Note that the axes are always given in global coordinates, compare the table in Section 8.5.3.2. The index 3 constraint equations read

$${}^{0}\mathbf{t}_{z0}^{\mathrm{T}}\ {}^{0}\mathbf{t}_{y1} = 0 \tag{8.181}$$

$${}^{0}\mathbf{t}_{z0}^{\mathrm{T}}\ {}^{0}\mathbf{t}_{x1} = 0 \tag{8.182}$$

$${}^{0}\mathbf{t}_{x0}^{\mathrm{T}}\ {}^{0}\mathbf{t}_{y1} = 0 \tag{8.183}$$

The index 2 constraints follow from the derivative of Eq. (8.181) w.r.t., and are given in the C++ code.
if `activeConnector = False`,

$$\mathbf{z} = \mathbf{0} \tag{8.184}$$

For examples on ObjectJointPrismaticX see Relevant Examples and TestModels with weblink:

- `revoluteJointPrismaticJointTest.py` (TestModels/)

8.5.4 ObjectJointSpherical

A spherical joint, which constrains the relative translation between two position based markers.

Additional information for ObjectJointSpherical:

- This `Object` has/provides the following types = `Connector`, `Constraint`
- Requested `Marker` type = `Position`
- **Short name** for Python = `SphericalJoint`
- **Short name** for Python visualization object = `VSphericalJoint`

The item **ObjectJointSpherical** with type = 'JointSpherical' has the following parameters:

Name	type	size	default value	description
name	String		''	constraints's unique name
markerNumbers	ArrayMarkerIndex	2	[invalid [-1], invalid [-1]]	list of markers used in connector; $m1$ is the moving coin rigid body and $m0$ is the marker for the ground body, which use the localPosition=[0,0,0] for this marker!
constrainedAxes	ArrayIndex	3	[1,1,1]	flag, which determines which translation (0,1,2) and rotation (3,4,5) axes are constrained; for j_i, two values are possible: 0=free axis, 1=constrained axis
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectJointSpherical			parameters for visualization of item

The item VObjectJointSpherical has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
jointRadius	float		0.1	radius of joint to draw
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

8.5.4.1 DESCRIPTION of ObjectJointSpherical:

Information on input parameters:

input parameter	symbol	description see tables above
markerNumbers	$[m0, m1]^{\mathrm{T}}$	
constrainedAxes	$\mathbf{j} = [j_0, \ldots, j_2]$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position	${}^0\mathbf{p}_{m0}$	current global position of position marker $m0$
Velocity	${}^0\mathbf{v}_{m0}$	current global velocity of position marker $m0$
Displacement	${}^0\Delta\mathbf{p} = {}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0}$	constraint drift or relative motion, if not all axes fixed
Force	${}^0\mathbf{f}$	joint force in global coordinates

8.5.4.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker $m0$
marker m1 position	${}^0\mathbf{p}_{m1}$	current global position which is provided by marker $m1$
marker m0 velocity	${}^0\mathbf{v}_{m0}$	current global velocity which is provided by marker $m0$
marker m1 velocity	${}^0\mathbf{v}_{m1}$	current global velocity which is provided by marker $m1$
relative velocity	${}^0\Delta\mathbf{v} = {}^0\mathbf{v}_{m1} - {}^0\mathbf{v}_{m0}$	constraint velocity error, or relative velocity if not all axes fixed
algebraic variables	$\mathbf{z} = [\lambda_0, \ldots, \lambda_2]^{\mathrm{T}}$	vector of algebraic variables (Lagrange multipliers) according to the algebraic equations

8.5.4.3 Connector constraint equations

`activeConnector = True`: If $[j_0, \ldots, j_2] = [1, 1, 1]^{\mathrm{T}}$, meaning that all translational coordinates are fixed, the translational index 3 constraints read

$$ {}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0} = \mathbf{0} \tag{8.185} $$

and the translational index 2 constraints read

$$ {}^0\mathbf{v}_{m1} - {}^0\mathbf{v}_{m0} = \mathbf{0} \tag{8.186} $$

If $[j_0, \ldots, j_2] \neq [1,1,1]^T$, meaning that at least one translational coordinate is free, the translational index 3 constraints read for every component $k \in [0,1,2]$ of the vector ${}^0\Delta\mathbf{p}$

$$
\begin{aligned}
{}^0\Delta p_k &= 0 \quad \text{if} \quad j_k = 1 \quad \text{and} && (8.187)\\
\lambda_k &= 0 \quad \text{if} \quad j_k = 0 && (8.188)\\
& && (8.189)
\end{aligned}
$$

and the translational index 2 constraints read for every component $k \in [0,1,2]$ of the vector ${}^0\Delta\mathbf{v}$

$$
\begin{aligned}
{}^0\Delta v_k &= 0 \quad \text{if} \quad j_k = 1 \quad \text{and} && (8.190)\\
\lambda_k &= 0 \quad \text{if} \quad j_k = 0 && (8.191)\\
& && (8.192)
\end{aligned}
$$

`activeConnector = False`:

$$\mathbf{z} = \mathbf{0} \tag{8.193}$$

8.5.4.4 Example for body position marker

In this example, we study the constraint equations for two body position marker, see Section 8.9.2, based on rigid bodies, see Section 8.2.6. The markers m_0 and m_1 have the positions

$${}^0\mathbf{p}_0\,({}^b\mathbf{b}_0) = {}^0\mathbf{r}_{\text{ref},0} + {}^0\mathbf{u}_0 + {}^{0b}\mathbf{A}_0\,{}^b\mathbf{b}_0, \quad {}^0\mathbf{p}_1\,({}^b\mathbf{b}_1) = {}^0\mathbf{r}_{\text{ref},1} + {}^0\mathbf{u}_1 + {}^{0b}\mathbf{A}_1\,{}^b\mathbf{b}_1\,. \tag{8.194}$$

From there, we can derive the 3 constraint equation

$${}^0\mathbf{r}_{\text{ref},1} + {}^0\mathbf{u}_1 + {}^{0b}\mathbf{A}_1\,{}^b\mathbf{b}_1 - \left({}^0\mathbf{r}_{\text{ref},0} + {}^0\mathbf{u}_0 + {}^{0b}\mathbf{A}_0\,{}^b\mathbf{b}_0\right) = \mathbf{0}\,. \tag{8.195}$$

The constraint jacobians simply follow from the position jacobians of the respective markers ${}^0\mathbf{J}_{\text{pos},0}$ and ${}^0\mathbf{J}_{\text{pos},1}$. The position jacobians are added to the system jacobian at rows according to the global indices of the constraint equations and the columns are determined by the coordinate indices of the bodies' coordinates.

For examples on ObjectJointSpherical see Relevant Examples and TestModels with weblink:

- `NGsolveLinearFEM.py` (Examples/)
- `newtonsCradle.py` (Examples/)
- `bungeeJump.py` (Examples/)
- `chainDriveExample.py` (Examples/)
- `humanRobotInteraction.py` (Examples/)
- `NGsolvePostProcessingStresses.py` (Examples/)
- `objectFFRFreducedOrderNetgen.py` (Examples/)
- `sliderCrank3DwithANCFbeltDrive.py` (Examples/)
- `ACFtest.py` (TestModels/)

- createFunctionsTest.py (TestModels/)
- driveTrainTest.py (TestModels/)
- mainSystemExtensionsTests.py (TestModels/)
- genericJointUserFunctionTest.py (TestModels/)
- kinematicTreeConstraintTest.py (TestModels/)
- objectFFRFreducedOrderAccelerations.py (TestModels/)
- ...

8.5.5 ObjectJointRollingDisc

A joint representing a rolling rigid disc (marker 1) on a flat surface (marker 0, ground body) in global x-y plane. The contraint is based on an idealized rolling formulation with no slip. The contraints works for discs as long as the disc axis and the plane normal vector are not parallel. It must be assured that the disc has contact to ground in the initial configuration (adjust z-position of body accordingly). The ground body can be a rigid body which is moving. In this case, the flat surface is assumed to be in the x-y-plane at $z = 0$. Note that the rolling body must have the reference point at the center of the disc. NOTE: the cases of normal other than z-direction, wheel axis other than x-axis and moving ground body needs to be tested further, check your results!

Additional information for ObjectJointRollingDisc:

- This `Object` has/provides the following types = `Connector`, `Constraint`
- Requested `Marker` type = `Position` + `Orientation`
- **Short name** for Python = `RollingDiscJoint`
- **Short name** for Python visualization object = `VRollingDiscJoint`

The item **ObjectJointRollingDisc** with type = 'JointRollingDisc' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex	2	[invalid [-1], invalid [-1]]	list of markers used in connector; $m0$ represents the ground and $m1$ represents the rolling body, which has its reference point (=local position [0,0,0]) at the disc center point
constrainedAxes	ArrayIndex	3	[1,1,1]	flags, which determine which constraints are active, in which j_0 represents lateral motion, j_1 longitudinal (forward/backward) motion and j_2 represents the normal (contact) direction
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
discRadius	PReal		0	defines the disc radius
discAxis	Vector3D		[1,0,0]	axis of disc defined in marker $m1$ frame
planeNormal	Vector3D		[0,0,1]	normal to the contact / rolling plane defined in marker $m0$ coordinates
visualization	VObjectJointRollingDisc			parameters for visualization of item

The item VObjectJointRollingDisc has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
discWidth	float		0.1	width of disc for drawing
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

---

8.5.5.1 DESCRIPTION of ObjectJointRollingDisc:

Information on input parameters:

input parameter	symbol	description see tables above
markerNumbers	$[m0, m1]^{\mathrm{T}}$	
constrainedAxes	$\mathbf{j} = [j_0, \ldots, j_2]$	
discAxis	${}^{m1}\mathbf{w}_1$, $\lvert {}^{m1}\mathbf{w}_1 \rvert = 1$	
planeNormal	${}^{m0}\mathbf{v}_{PN}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position	${}^{0}\mathbf{p}_{G}$	current global position of contact point between rolling disc and ground
Velocity	${}^{0}\mathbf{v}_{trail}$	current velocity of the trail (according to motion of the contact point along the trail!) in global coordinates; this is not the velocity of the contact point; needs further testing for general case of relative moving bodies
ForceLocal	${}^{J1}\mathbf{f} = {}^{0}[f_0, f_1, f_2]^{\mathrm{T}} = [-\mathbf{z}^T\,{}^{0}\mathbf{w}_{lat}\,, -\mathbf{z}^T\,{}^{0}\mathbf{w}_2\,, -\mathbf{z}^T\,{}^{0}\mathbf{v}_{PN}\,]^{\mathrm{T}}$	contact forces acting on disc, in special $J1$ joint coordinates, f_0 being the lateral force (parallel to ground plane), f_1 being the longitudinal force and f_2 being the normal force
RotationMatrix	${}^{0,J1}\mathbf{A} = [{}^{0}\mathbf{w}_{lat}\,, {}^{0}\mathbf{w}_2, {}^{0}\mathbf{v}_{PN}\,]$	transformation matrix of special joint coordinates $J1$ to global coordinates

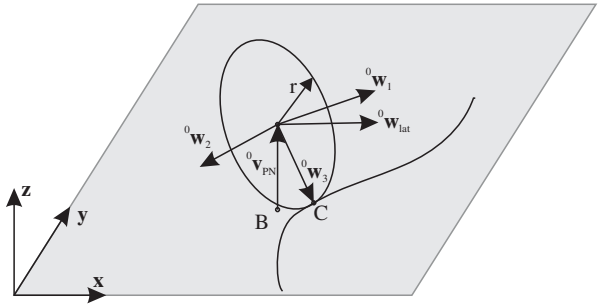
8.5.5.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^{0}\mathbf{p}_{m0}$	current global position of marker $m0$; needed only if body $m0$ is not a ground body

marker m0 orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker m0 (assumed to be rigid body)
marker m0 velocity	${}^{0}\mathbf{v}_{m0}$	current global velocity which is provided by marker m0 (assumed to be rigid body)
marker m0 angular velocity	${}^{0}\boldsymbol{\omega}_{m0}$	current angular velocity vector provided by marker m0 (assumed to be rigid body)
marker m1 position	${}^{0}\mathbf{p}_{m1}$	center of disc
marker m1 orientation	${}^{0,m1}\mathbf{A}$	current rotation matrix provided by marker m1
marker m1 velocity	${}^{0}\mathbf{v}_{m1}$	accordingly
marker m1 angular velocity	${}^{0}\boldsymbol{\omega}_{m1}$	current angular velocity vector provided by marker m1
ground normal vector	${}^{0}\mathbf{v}_{PN} = {}^{0,m0}\mathbf{A}\ {}^{m0}\mathbf{v}_{PN}$	normalized normal vector to the ground plane, moving with marker $m0$
ground position B	${}^{0}\mathbf{p}_{B}$	disc center point projected on ground in plane normal (z-direction, $z = 0$)
ground position C	${}^{0}\mathbf{p}_{C}$	contact point of disc with ground in global coordinates
ground velocity C	${}^{0}\mathbf{v}_{Cm1}$	velocity of disc (marker 1) at ground contact point (must be zero if ground does not move)
ground velocity C	${}^{0}\mathbf{v}_{Cm2}$	velocity of ground (marker 0) at ground contact point (is always zero if ground does not move)
wheel axis vector	${}^{0}\mathbf{w}_1 = {}^{0,m1}\mathbf{A}\ {}^{m1}\mathbf{w}_1$	normalized disc axis vector
longitudinal vector	${}^{0}\mathbf{w}_2$	vector in longitudinal (motion) direction
lateral vector	${}^{0}\mathbf{w}_{lat} = {}^{0}\mathbf{v}_{PN} \times {}^{0}\mathbf{w}_2$	vector in lateral direction, parallel to ground plane
contact point vector	${}^{0}\mathbf{w}_3$	normalized vector from disc center point in direction of contact point C
$D1$ transformation matrix	${}^{0,D1}\mathbf{A} = [{}^{0}\mathbf{w}_1, {}^{0}\mathbf{w}_2, {}^{0}\mathbf{w}_3]$	transformation of special disc coordinates $D1$ to global coordinates
$J1$ transformation matrix	${}^{0,J1}\mathbf{A} = [{}^{0}\mathbf{w}_{lat}, {}^{0}\mathbf{w}_2, {}^{0}\mathbf{v}_{PN}]$	transformation of special joint $J1$ coordinates to global coordinates
algebraic variables	$\mathbf{z} = [\lambda_0, \lambda_1, \lambda_2]^{\mathrm{T}}$	vector of algebraic variables (Lagrange multipliers) according to the algebraic equations

8.5.5.3 Geometric relations

The main geometrical setup is shown in the following figure:



First, the contact point ${}^0\mathbf{p}_C$ must be computed. With the helper vector,

$$ {}^0\mathbf{x} = {}^0\mathbf{w}_1 \times {}^0\mathbf{v}_{PN} \tag{8.196} $$

we obtain a disc coordinate system, representing the longitudinal direction,

$$ {}^0\mathbf{w}_2 = \frac{1}{|{}^0\mathbf{x}|} {}^0\mathbf{x} \tag{8.197} $$

and the vector to the contact point,

$$ {}^0\mathbf{w}_3 = {}^0\mathbf{w}_1 \times {}^0\mathbf{w}_2 \tag{8.198} $$

The contact point C can be computed from

$$ {}^0\mathbf{p}_C = {}^0\mathbf{p}_{m1} + r \cdot {}^0\mathbf{w}_3 \tag{8.199} $$

The velocity of the contact point at the disc is computed from,

$$ {}^0\mathbf{v}_{Cm1} = {}^0\mathbf{v}_{m1} + {}^0\boldsymbol{\omega}_{m1} \times (r \cdot {}^0\mathbf{w}_3) \tag{8.200} $$

If marker 0 body is (moving) rigid body instead of a ground body, the contact point C is reconstructed in body of marker 0,

$$ {}^{m0}\mathbf{p}_C = {}^{m0,0}\mathbf{A} ({}^0\mathbf{p}_C - {}^0\mathbf{p}_{m0}) \tag{8.201} $$

The velocity of the contact point at the marker 0 body reads

$$ {}^0\mathbf{v}_{Cm0} = {}^0\mathbf{v}_{m0} + {}^0\boldsymbol{\omega}_{m0} \times \left({}^{0,m0}\mathbf{A} \; {}^{m0}\mathbf{p}_C \right) \tag{8.202} $$

8.5.5.4 Connector constraint equations

Constraints for `activeConnector = True`:
The non-holonomic, index 2 constraints for the tangential and normal contact follow from (an index 3 formulation would be possible, but is not implemented yet because of mixing different jacobians)

$$ {}^{J1,0}\mathbf{A} \left(\begin{bmatrix} {}^0\mathbf{v}_{Cm1,x} \\ {}^0\mathbf{v}_{Cm1,y} \\ {}^0\mathbf{v}_{Cm1,z} \end{bmatrix} - \begin{bmatrix} {}^0\mathbf{v}_{Cm0,x} \\ {}^0\mathbf{v}_{Cm0,y} \\ {}^0\mathbf{v}_{Cm0,z} \end{bmatrix} \right) = \mathbf{0} \tag{8.203} $$

In case that `activeConnector = False`, the Lagrange multipliers are set to zero:

$$ \mathbf{z} = \mathbf{0} \tag{8.204} $$

Note that since version 1.8.27 the constraints can be turned on/off separately with `constrainedAxes=[b0,b1,b2]`, in which `b0` represents the flag for lateral motion, `b1` switches the constraint for forward motion and `b2` for motion in plane normal direction.

For examples on ObjectJointRollingDisc see Relevant Examples and TestModels with weblink:

- `bicycleIftommBenchmark.py` (Examples/)

- reinforcementLearningRobot.py (Examples/)
- rollingCoinTest.py (TestModels/)
- rollingDiscTangentialForces.py (TestModels/)
- rotatingTableTest.py (TestModels/)
- createFunctionsTest.py (TestModels/)
- createRollingDiscTest.py (TestModels/)

8.5.6 ObjectJointRevolute2D

A revolute joint in 2D; constrains the absolute 2D position of two points given by PointMarkers or RigidMarkers

Additional information for ObjectJointRevolute2D:

- This `Object` has/provides the following types = `Connector`, `Constraint`
- Requested `Marker` type = `Position`
- **Short name** for Python = `RevoluteJoint2D`
- **Short name** for Python visualization object = `VRevoluteJoint2D`

The item **ObjectJointRevolute2D** with type = 'JointRevolute2D' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex		[invalid [-1], invalid [-1]]	list of markers used in connector
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectJointRevolute2D			parameters for visualization of item

The item VObjectJointRevolute2D has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = radius of revolute joint; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

---

8.5.6.1 DESCRIPTION of ObjectJointRevolute2D:

For examples on ObjectJointRevolute2D see Relevant Examples and TestModels with weblink:

- pendulumGeomExactBeam2D.py (Examples/)
- sliderCrank3DwithANCFbeltDrive2.py (Examples/)
- ANCFrotatingCable2D.py (Examples/)
- ANCFslidingJoint2D.py (Examples/)
- ANCFslidingJoint2Drigid.py (Examples/)
- beltDriveALE.py (Examples/)
- beltDriveReevingSystem.py (Examples/)
- beltDrivesComparison.py (Examples/)
- chainDriveExample.py (Examples/)
- doublePendulum2D.py (Examples/)
- finiteSegmentMethod.py (Examples/)
- geneticOptimizationSliderCrank.py (Examples/)
- ...
- ANCFbeltDrive.py (TestModels/)
- ANCFgeneralContactCircle.py (TestModels/)
- ANCFoutputTest.py (TestModels/)
- ...

8.5.7 ObjectJointPrismatic2D

A prismatic joint in 2D; allows the relative motion of two bodies, using two RigidMarkers; the vector $\mathbf{t}_0$ = axisMarker0 is given in local coordinates of the first marker's (body) frame and defines the prismatic axis; the vector $\mathbf{n}_1$ = normalMarker1 is given in the second marker's (body) frame and is the normal vector to the prismatic axis; using the global position vector $\mathbf{p}_0$ and rotation matrix $\mathbf{A}_0$ of marker0 and the global position vector $\mathbf{p}_1$ rotation matrix $\mathbf{A}_1$ of marker1, the equations for the prismatic joint follow as

$$(\mathbf{p}_1 - \mathbf{p}_0)^T \cdot \mathbf{A}_1 \cdot \mathbf{n}_1 = 0 \tag{8.205}$$

$$(\mathbf{A}_0 \cdot \mathbf{t}_0)^T \cdot \mathbf{A}_1 \cdot \mathbf{n}_1 = 0 \tag{8.206}$$

The lagrange multipliers follow for these two equations $[\lambda_0, \lambda_1]$, in which λ_0 is the transverse force and λ_1 is the torque in the joint.

Additional information for ObjectJointPrismatic2D:

- This `Object` has/provides the following types = `Connector`, `Constraint`
- Requested `Marker` type = `Position` + `Orientation`
- **Short name** for Python = `PrismaticJoint2D`
- **Short name** for Python visualization object = `VPrismaticJoint2D`

The item **ObjectJointPrismatic2D** with type = 'JointPrismatic2D' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex		[invalid [-1], invalid [-1]]	list of markers used in connector
axisMarker0	Vector3D		[1.,0.,0.]	direction of prismatic axis, given as a 3D vector in Marker0 frame
normalMarker1	Vector3D		[0.,1.,0.]	direction of normal to prismatic axis, given as a 3D vector in Marker1 frame
constrainRotation	Bool		True	flag, which determines, if the connector also constrains the relative rotation of the two objects; if set to false, the constraint will keep an algebraic equation set equal zero
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectJointPrismatic2D			parameters for visualization of item

The item VObjectJointPrismatic2D has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = radius of revolute joint; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

---

8.5.7.1 DESCRIPTION of ObjectJointPrismatic2D:

For examples on ObjectJointPrismatic2D see Relevant Examples and TestModels with weblink:

- `sliderCrank3DwithANCFbeltDrive2.py` (Examples/)
- `geneticOptimizationSliderCrank.py` (Examples/)
- `PARTS_ATEs_moving.py` (TestModels/)
- `scissorPrismaticRevolute2D.py` (TestModels/)
- `sliderCrankFloatingTest.py` (TestModels/)

8.5.8 ObjectJointSliding2D

A specialized sliding joint (without rotation) in 2D between a Cable2D (marker1) and a position-based marker (marker0); the data coordinate x[0] provides the current index in slidingMarkerNumbers, and x[1] the local position in the cable element at the beginning of the timestep.

Additional information for ObjectJointSliding2D:

- This `Object` has/provides the following types = `Connector`, `Constraint`
- Requested `Marker` type = `_None`
- Requested `Node` type = `GenericData`
- **Short name** for Python = `SlidingJoint2D`
- **Short name** for Python visualization object = `VSlidingJoint2D`

The item **ObjectJointSliding2D** with type = 'JointSliding2D' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex		[invalid [-1], invalid [-1]]	marker m0: position or rigid body marker of mass point or rigid body; marker m1: updated marker to Cable2D element, where the sliding joint currently is attached to; must be initialized with an appropriate (global) marker number according to the starting position of the sliding object; this marker changes with time (PostNewtonStep)
slidingMarkerNumbers	ArrayMarkerIndex		[]	these markers are used to update marker m1, if the sliding position exceeds the current cable's range; the markers must be sorted such that marker m_{si} at x=cable(i).length is equal to marker(i+1) at x=0 of cable(i+1)
slidingMarkerOffsets	Vector		[]	this list contains the offsets of every sliding object (given by slidingMarkerNumbers) w.r.t. to the initial position (0): marker m0: offset=0, marker m1: offset=Length(cable0), marker m2: offset=Length(cable0)+Length(cable1), ...
nodeNumber	NodeIndex		invalid (-1)	node number of a NodeGenericData for 1 dataCoordinate showing the according marker number which is currently active and the start-of-step (global) sliding position

classicalFormulation	Bool		True	True: uses a formulation with 3 (+1) equations, including the force in sliding direction to be zero; forces in global coordinates, only index 3; False: use local formulation, which only needs 2 (+1) equations and can be used with index 2 formulation
constrainRotation	Bool		False	True: add constraint on rotation of marker m0 relative to slope (if True, marker m0 must be a rigid body marker); False: marker m0 body can rotate freely
axialForce	Real		0	ONLY APPLIES if classicalFormulation==True; axialForce represents an additional sliding force acting between beam and marker m0 body in axial (beam) direction; this force can be used to drive a body on a beam, but can only be changed with user functions.
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectJointSliding2D			parameters for visualization of item

The item VObjectJointSliding2D has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = radius of revolute joint; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

---

8.5.8.1 DESCRIPTION of ObjectJointSliding2D:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
markerNumbers	$[m0, m1]^T$	
slidingMarkerNumbers	$[m_{s0}, \ldots, m_{sn}]^T$	
slidingMarkerOffsets	$[d_{s0}, \ldots, d_{sn}]$	
nodeNumber	n_{GD}	

axialForce	f_{ax}	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position		position vector of joint given by marker0
Velocity		velocity vector of joint given by marker0
SlidingCoordinate		global sliding coordinate along all elements; the maximum sliding coordinate is equivalent to the reference lengths of all sliding elements
Force		joint force vector (3D)

8.5.8.2 Definition of quantities

intermediate variables	symbol	description
data node	$\mathbf{x} = [x_{data0}, x_{data1}]^T$	coordinates of node with node number n_{GD}
data coordinate 0	x_{data0}	the current index in slidingMarkerNumbers
data coordinate 1	x_{data1}	the global sliding coordinate (ranging from 0 to the total length of all sliding elements) at **start-of-step** - beginning of the timestep
marker m0 position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker m0
marker m0 velocity	${}^0\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
marker m0 orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker m0 (assumed to be rigid body)
marker m0 angular velocity	${}^0\boldsymbol{\omega}_{m0}$	current angular velocity vector provided by marker m0 (assumed to be rigid body)
cable coordinates	$\mathbf{q}_{ANCF,m1}$	current coordiantes of the ANCF cable element with the current marker $m1$ is referring to
sliding position	${}^0\mathbf{r}_{ANCF} = \mathbf{S}(s_{el})\mathbf{q}_{ANCF,m1}$	current global position at the ANCF cable element, evaluated at local sliding position s_{el}
sliding position slope	${}^0\mathbf{r}'_{ANCF} = \mathbf{S}'(s_{el})\mathbf{q}_{ANCF,m1} = [r'_0, r'_1]^T$	current global slope vector of the ANCF cable element, evaluated at local sliding position s_{el}
sliding velocity	${}^0\mathbf{v}_{ANCF} = \mathbf{S}(s_{el})\dot{\mathbf{q}}_{ANCF,m1}$	current global velocity at the ANCF cable element, evaluated at local sliding position s_{el} (s_{el} not differentiated!!!)
sliding velocity slope	${}^0\mathbf{v}'_{ANCF} = \mathbf{S}'(s_{el})\dot{\mathbf{q}}_{ANCF,m1}$	current global slope velocity vector of the ANCF cable element, evaluated at local sliding position s_{el}

sliding normal vector	${}^0\mathbf{n} = [-r'_1, r'_0]$	2D normal vector computed from slope $\mathbf{r}' = {}^0\mathbf{r}'_{ANCF}$
sliding normal velocity vector	${}^0\dot{\mathbf{n}} = [-\dot{r}'_1, \dot{r}'_0]$	time derivative of 2D normal vector computed from slope velocity $\dot{\mathbf{r}}' = {}^0\dot{\mathbf{r}}'_{ANCF}$
algebraic coordinates	$\mathbf{z} = [\lambda_0, \lambda_1, s]^T$	algebraic coordinates composed of Lagrange multipliers λ_0 and λ_1 (in local cable coordinates: λ_0 is in axis direction) and the current sliding coordinate s, which is local in the current cable element.
local sliding coordinate	s	local incremental sliding coordinate s: the (algebraic) sliding coordinate **relative to the start-of-step value**. Thus, s only contains small local increments.

output variables	**symbol**	**formula**
Position	${}^0\mathbf{p}_{m0}$	current global position of position marker $m0$
Velocity	${}^0\mathbf{v}_{m0}$	current global velocity of position marker $m0$
SlidingCoordinate	$s_g = s + x_{data1}$	current value of the global sliding coordinate
Force	$\mathbf{f}$	see below

8.5.8.3 Geometric relations

Assume we have given the sliding coordinate s (e.g., as a guess of the Newton method or beginning of the time step). The element sliding coordinate (in the local coordinates of the current sliding element) is computed as

$$s_{el} = s + x_{data1} - d_{m1} = s_g - d_{m1}. \tag{8.207}$$

The vector (=difference; error) between the marker $m0$ and the marker $m1$ (=$\mathbf{r}_{ANCF}$) positions reads

$${}^0\Delta\mathbf{p} = {}^0\mathbf{r}_{ANCF} - {}^0\mathbf{p}_{m0} \tag{8.208}$$

The vector (=difference; error) between the marker $m0$ and the marker $m1$ velocities reads

$${}^0\Delta\mathbf{v} = {}^0\dot{\mathbf{r}}_{ANCF} - {}^0\mathbf{v}_{m0} \tag{8.209}$$

8.5.8.4 Connector constraint equations (classicalFormulation=True)

The 2D sliding joint is implemented having 3 equations (4 if constrainRotation==True, see below), using the special algebraic coordinates $\mathbf{z}$. The algebraic equations read

$${}^0\Delta\mathbf{p} = \mathbf{0}, \quad \text{... two index 3 eqs, ensure sliding body stays at cable} \tag{8.210}$$

$$[\lambda_0, \lambda_1] \cdot {}^0\mathbf{r}'_{ANCF} - |{}^0\mathbf{r}'_{ANCF}| \cdot f_{\text{ax}} = 0, \quad \text{... one index 1 equ., ensure force in sliding dir. = 0} \tag{8.211}$$

$$\tag{8.212}$$

No index 2 case exists, because no time derivative exists for s_{el}. The jacobian matrices for algebraic and ODE2 coordinates read

$$\mathbf{J}_{AE} = \begin{bmatrix} 0 & 0 & r'_0 \\ 0 & 0 & r'_1 \\ r'_0 & r'_1 & r''_0\lambda_0 + r''_1\lambda_1 \end{bmatrix} \tag{8.213}$$

$$\mathbf{J}_{ODE2} = \begin{bmatrix} -J_{pos,m0} & \mathbf{S}(s_{el}) \\ \mathbf{0}^{\mathrm{T}} & [\lambda_0,\, \lambda_1] \cdot \mathbf{S}'(s_{el}) \end{bmatrix} \tag{8.214}$$

if `activeConnector = False`, the algebraic equations are changed to:

$$\lambda_0 = 0, \tag{8.215}$$

$$\lambda_1 = 0, \tag{8.216}$$

$$s = 0 \tag{8.217}$$

8.5.8.5 Connector constraint equations (classicalFormulation=False)

The 2D sliding joint is implemented having 3 equations (first equation is dummy and could be eliminated; 4 equations if constrainRotation==True, see below), using the special algebraic coordinates $\mathbf{z}$. The algebraic equations read

$$\lambda_0 = 0, \quad \text{... equation not necessary, but can be used for switching to other modes} \tag{8.218}$$

$${}^0\Delta\mathbf{p}^{\mathrm{T}}\,{}^0\mathbf{n} = 0, \quad \text{... equation ensures that sliding body stays at cable centerline; index3} \tag{8.219}$$

$${}^0\Delta\mathbf{p}^{\mathrm{T}}\,{}^0\mathbf{r}'_{ANCF} = 0. \quad \text{... resolves the sliding coordinate } s\text{; index1 equation!} \tag{8.220}$$

In the index 2 case, the second equation reads

$${}^0\Delta\mathbf{v}^{\mathrm{T}}\,{}^0\mathbf{n} + {}^0\Delta\mathbf{p}^{\mathrm{T}}\,{}^0\dot{\mathbf{n}} = 0 \tag{8.221}$$

if `activeConnector = False`, the algebraic equations are changed to:

$$\lambda_0 = 0, \tag{8.222}$$

$$\lambda_1 = 0, \tag{8.223}$$

$$s = 0 \tag{8.224}$$

In case that `constrainRotation = True`, an additional constraint is added for the relative rotation between the slope of the cable and the orientation of marker m0 body. Assuming that the orientation of marker m0 is a 2D matrix (taking only x and y coordinates), the constraint reads

$${}^0\mathbf{r}'^{\mathrm{T}}_{ANCF}\,{}^{0,m0}\mathbf{A}\begin{bmatrix}0\\1\end{bmatrix} = 0 \tag{8.225}$$

The index 2 case follows straightforward to

$${}^0\dot{\mathbf{r}}'^{\mathrm{T}}_{ANCF}\,{}^{0,m0}\mathbf{A}\begin{bmatrix}0\\1\end{bmatrix} + {}^0\mathbf{r}'^{\mathrm{T}}_{ANCF}\,{}^{0,m0}\mathbf{A}\,{}^0\tilde{\boldsymbol{\omega}}_{m0}\begin{bmatrix}0\\1\end{bmatrix} = 0 \tag{8.226}$$

again assuming, that ${}^0\tilde{\boldsymbol{\omega}}_{m0}$ is only a 2×2 matrix.

8.5.8.6 Post Newton Step

After the Newton solver has converged, a PostNewtonStep is performed for the element, which updates the marker $m1$ index if necessary.

$$\begin{aligned} s_{el} < 0 \quad &\rightarrow \quad x_{data0} \mathrel{-}= 1 \\ s_{el} > L \quad &\rightarrow \quad x_{data0} \mathrel{+}= 1 \end{aligned} \tag{8.227}$$

Furthermore, it is checked, if x_{data0} becomes smaller than zero, which raises a warning and keeps $x_{data0} = 0$. The same results if $x_{data0} \geq sn$, then $x_{data0} = sn$. Finally, the data coordinate is updated in order to provide the starting value for the next step,

$$x_{data1} \mathrel{+}= s. \tag{8.228}$$

For examples on ObjectJointSliding2D see Relevant Examples and TestModels with weblink:

- `ANCFmovingRigidbody.py` (Examples/)
- `ANCFslidingJoint2D.py` (Examples/)
- `ANCFslidingJoint2Drigid.py` (Examples/)
- `ANCFswitchingSlidingJoint2D.py` (Examples/)
- `modelUnitTests.py` (TestModels/)

8.5.9 ObjectJointALEMoving2D

A specialized axially moving joint (without rotation) in 2D between a ALE Cable2D (marker1) and a position-based marker (marker0); ALE=Arbitrary Lagrangian Eulerian; the data coordinate x[0] provides the current index in slidingMarkerNumbers, and the ODE2 coordinate q[0] provides the (given) moving coordinate in the cable element.

Additional information for ObjectJointALEMoving2D:

- This `Object` has/provides the following types = `Connector`, `Constraint`
- Requested `Marker` type = `_None`
- Requested `Node` type: read detailed information of item
- **Short name** for Python = `ALEMovingJoint2D`
- **Short name** for Python visualization object = `VALEMovingJoint2D`

The item **ObjectJointALEMoving2D** with type = 'JointALEMoving2D' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex		[invalid [-1], invalid [-1]]	marker m0: position-marker of mass point or rigid body; marker m1: updated marker to ANCF Cable2D element, where the sliding joint currently is attached to; must be initialized with an appropriate (global) marker number according to the starting position of the sliding object; this marker changes with time (PostNewtonStep)
slidingMarkerNumbers	ArrayMarkerIndex		[]	a list of sn (global) marker numbers which are are used to update marker1
slidingMarkerOffsets	Vector		[]	this list contains the offsets of every sliding object (given by slidingMarkerNumbers) w.r.t. to the initial position (0): marker0: offset=0, marker1: offset=Length(cable0), marker2: offset=Length(cable0)+Length(cable1), ...
slidingOffset	Real		0.	sliding offset list [SI:m]: a list of sn scalar offsets, which represent the (reference arc) length of all previous sliding cable elements
nodeNumbers	ArrayNodeIndex		[invalid [-1], invalid [-1]]	node number of NodeGenericData (GD) with one data coordinate and of NodeGenericODE2 (ALE) with one ODE2 coordinate

usePenaltyFormulation	Bool		False	flag, which determines, if the connector is formulated with penalty, but still using algebraic equations (IsPenaltyConnector() still false)
penaltyStiffness	Real		0.	penalty stiffness [SI:N/m] used if usePenaltyFormulation=True
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectJointALEMoving2D			parameters for visualization of item

The item VObjectJointALEMoving2D has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = radius of revolute joint; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

---

8.5.9.1 DESCRIPTION of ObjectJointALEMoving2D:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
markerNumbers	$[m0,\ m1]^{\mathrm{T}}$	
slidingMarkerNumbers	$[m_{s0}, \ldots, m_{sn}]^{\mathrm{T}}$	
slidingMarkerOffsets	$[d_{s0}, \ldots, d_{sn}]$	
slidingOffset	s_{off}	
nodeNumbers	$[n_{GD}, n_{ALE}]$	
penaltyStiffness	k	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
Position	${}^{0}\mathbf{p}_{m0}$	current global position of position marker $m0$
Velocity	${}^{0}\mathbf{v}_{m0}$	current global velocity of position marker $m0$

SlidingCoordinate	$s_g = q_{ALE} + s_{off}$	current value of the global sliding ALE coordinate, including offset; note that reference coordinate of q_{ALE} is ignored!
Coordinates	$[x_{data0}, q_{ALE}]^T$	provides two values: [0] = current sliding marker index, [1] = ALE sliding coordinate
Coordinates_t	$[\dot{q}_{ALE}]^T$	provides ALE sliding velocity
Force	$\mathbf{f}$	joint force vector (3D)

8.5.9.2 Definition of quantities

intermediate variables	**symbol**	**description**
generic data node	$\mathbf{x} = [x_{data0}]^T$	coordinates of node with node number n_{GD}
generic ODE2 node	$\mathbf{q} = [q_0]^T$	coordinates of node with node number n_{ALE}, which is shared with all ALE-ANCF and ALE sliding joint objects
data coordinate	x_{data0}	the current index in slidingMarkerNumbers
ALE coordinate	$q_{ALE} = q_0$	current ALE coordinate (in fact this is the Eulerian coordinate in the ALE formulation); note that reference coordinate of q_{ALE} is ignored!
marker m0 position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker m0
marker m0 velocity	${}^0\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
cable coordinates	$\mathbf{q}_{ANCF,m1}$	current coordiantes of the ANCF cable element with the current marker $m1$ is referring to
sliding position	${}^0\mathbf{r}_{ANCF} = \mathbf{S}(s_{el})\mathbf{q}_{ANCF,m1}$	current global position at the ANCF cable element, evaluated at local sliding position s_{el}
sliding position slope	${}^0\mathbf{r}'_{ANCF} = \mathbf{S}'(s_{el})\mathbf{q}_{ANCF,m1}$	current global slope vector of the ANCF cable element, evaluated at local sliding position s_{el}
sliding velocity	${}^0\mathbf{v}_{ANCF} = \mathbf{S}(s_{el})\dot{\mathbf{q}}_{ANCF,m1} + \dot{q}_{ALE}\,{}^0\mathbf{r}'_{ANCF}$	current global velocity at the ANCF cable element, evaluated at local sliding position s_{el}, including convective term
sliding normal vector	${}^0\mathbf{n} = [-r'_1, r'_0]$	2D normal vector computed from slope $\mathbf{r}' = {}^0\mathbf{r}'_{ANCF}$
algebraic variables	$\mathbf{z} = [\lambda_0, \lambda_1]^T$	algebraic variables (Lagrange multipliers) according to the algebraic equations

8.5.9.3 Geometric relations

The element sliding coordinate (in the local coordinates of the current sliding element) is computed from the ALE coordinate

$$s_{el} = q_{ALE} + s_{off} - d_{m1} = s_g - d_{m1}. \tag{8.229}$$

For the description of the according quantities, see the description above. The distance d_{m1} is obtained from the `slidingMarkerOffsets` list, using the current (local) index x_{data0}. The vector (=difference; error) between the marker $m0$ and the marker $m1$ (=$\mathbf{r}_{ANCF}$) positions reads

$$^0\Delta\mathbf{p} = {}^0\mathbf{r}_{ANCF} - {}^0\mathbf{p}_{m0} \tag{8.230}$$

Note that ${}^0\mathbf{p}_{m0}$ represents the current position of the marker $m0$, which could represent the midpoint of a mass sliding along the beam. The position ${}^0\mathbf{r}_{ANCF}$ is computed from the beam represented by marker $m1$, using the local beam coordinate $x = s_{el}$. The marker and the according beam finite element changes during movement using the list `slidingMarkerNumbers` and the index is updated in the PostNewtonStep. The vector (=difference; error) between the marker $m0$ and the marker $m1$ velocities reads

$$^0\Delta\mathbf{v} = {}^0\mathbf{v}_{ANCF} - {}^0\mathbf{v}_{m0} \tag{8.231}$$

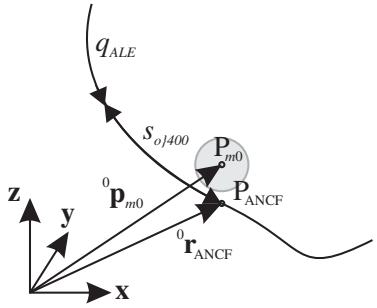


Figure 8.5: Geometrical relations for ALE sliding joint.

8.5.9.4 Connector constraint equations

The 2D sliding joint is implemented having 2 equations, using the Lagrange multipliers $\mathbf{z}$. The algebraic (index 3) equations read

$$^0\Delta\mathbf{p} = 0 \tag{8.232}$$

Note that the Lagrange multipliers $[\lambda_0, \lambda_1]^T$are the global forces in the joint. In the index 2 case the algebraic equations read

$$^0\Delta\mathbf{v} = 0 \tag{8.233}$$

If `usePenalty = True`, the algebraic equations are changed to:

$$^0\Delta\mathbf{p} - \frac{1}{k}\mathbf{z} = 0. \tag{8.234}$$

If `activeConnector = False`, the algebraic equations are changed to:

$$\lambda_0 = 0, \tag{8.235}$$

$$\lambda_1 = 0. \tag{8.236}$$

8.5.9.5 Post Newton Step

After the Newton solver has converged, a PostNewtonStep is performed for the element, which updates the marker $m1$ index if necessary.

$$\begin{aligned} s_{el} < 0 \quad &\rightarrow \quad x_{data0} \mathrel{-}= 1 \\ s_{el} > L \quad &\rightarrow \quad x_{data0} \mathrel{+}= 1 \end{aligned} \tag{8.237}$$

Furthermore, it is checked, if x_{data0} becomes smaller than zero, which raises a warning and keeps $x_{data0} = 0$. The same results if $x_{data0} \geq sn$, then $x_{data0} = sn$. Finally, the data coordinate is updated in order to provide the starting value for the next step,

$$x_{data1} \mathrel{+}= s. \tag{8.238}$$

For examples on ObjectJointALEMoving2D see Relevant Examples and TestModels with weblink:

- `ANCFmovingRigidbody.py` (Examples/)
- `ANCFmovingRigidBodyTest.py` (TestModels/)

8.6 Objects (Connector)

A Connector is a special Object, which links two or more markers. A Connector which is not a Constraint, is a force element (e.g., spring-damper) or a penalty based joint.

8.6.1 ObjectConnectorSpringDamper

An simple spring-damper element with additional force; connects to position-based markers.

Additional information for ObjectConnectorSpringDamper:

- This `Object` has/provides the following types = `Connector`
- Requested `Marker` type = `Position`
- **Short name** for Python = `SpringDamper`
- **Short name** for Python visualization object = `VSpringDamper`

The item **ObjectConnectorSpringDamper** with type = 'ConnectorSpringDamper' has the following parameters:

Name	type	size	default value	description
name	String		"	connector's unique name
markerNumbers	ArrayMarkerIndex		[invalid [-1], invalid [-1]]	list of markers used in connector
referenceLength	UReal		0.	reference length [SI:m] of spring
stiffness	UReal		0.	stiffness [SI:N/m] of spring; force acts against (length-initialLength)
damping	UReal		0.	damping [SI:N/(m s)] of damper; force acts against d/dt(length)
force	Real		0.	added constant force [SI:N] of spring; scalar force; f=1 is equivalent to reducing initialLength by 1/stiffness; f > 0: tension; f < 0: compression; can be used to model actuator force
velocityOffset	Real		0.	velocity offset [SI:m/s] of damper, being equivalent to time change of reference length
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
springForceUserFunction	PyFunctionMbsScalarIndexScalar5 0			A Python function which defines the spring force with parameters; the Python function will only be evaluated, if activeConnector is true, otherwise the SpringDamper is inactive; see description below
visualization	VObjectConnectorSpringDamper			parameters for visualization of item

The item VObjectConnectorSpringDamper has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = diameter of spring; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

8.6.1.1 DESCRIPTION of ObjectConnectorSpringDamper:

Information on input parameters:

input parameter	symbol	description see tables above
markerNumbers	$[m0, m1]^T$	
referenceLength	L_0	
stiffness	k	
damping	d	
force	f_a	
velocityOffset	$\dot{L}_0$	
springForceUserFunction	UF $\in \mathbb{R}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Distance		distance between both points
Displacement		relative displacement between both points
Velocity		relative velocity between both points
Force	**f**	3D spring-damper force vector
ForceLocal	f_{SD}	scalar spring-damper force

8.6.1.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker m0
marker m1 position	${}^0\mathbf{p}_{m1}$	
marker m0 velocity	${}^0\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
marker m1 velocity	${}^0\mathbf{v}_{m1}$	
time derivative of distance	$\dot{L}$	$\Delta^0\mathbf{v}^{\mathrm{T}}\mathbf{v}_f$
output variables	**symbol**	**formula**
Displacement	$\Delta^0\mathbf{p}$	${}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0}$
Velocity	$\Delta^0\mathbf{v}$	${}^0\mathbf{v}_{m1} - {}^0\mathbf{v}_{m0}$
Distance	L	$\lvert\Delta^0\mathbf{p}\rvert$
Force	$\mathbf{f}$	see below

8.6.1.3 Connector forces

The unit vector in force direction reads (raises SysError if $L = 0$),

$$\mathbf{v}_f = \frac{1}{L}\Delta^0\mathbf{p} \tag{8.239}$$

If `activeConnector = True`, the scalar spring force is computed as

$$f_{SD} = k \cdot (L - L_0) + d \cdot (\dot{L} - \dot{L}_0) + f_a \tag{8.240}$$

If the springForceUserFunction UF is defined, $\mathbf{f}$ instead becomes (t is current time)

$$f_{SD} = \mathrm{UF}(mbs, t, i_N, L - L_0, \dot{L} - \dot{L}_0, k, d, f_a) \tag{8.241}$$

and `iN` represents the itemNumber (=objectNumber). Note that, if `activeConnector = False`, f_{SD} is set to zero.

The vector of the spring-damper force applied at both markers finally reads

$$\mathbf{f} = f_{SD}\mathbf{v}_f \tag{8.242}$$

The virtual work of the connector force is computed from the virtual displacement

$$\delta\Delta^0\mathbf{p} = \delta\,{}^0\mathbf{p}_{m1} - \delta\,{}^0\mathbf{p}_{m0}\ , \tag{8.243}$$

and the virtual work (note the transposed version here, because the resulting generalized forces shall be a column vector),

$$\delta W_{SD} = \mathbf{f}\delta\Delta^0\mathbf{p} = \Big(k \cdot (L - L_0) + d \cdot (\dot{L} - \dot{L}_0) + f_a\Big)\Big(\delta\,{}^0\mathbf{p}_{m1} - \delta\,{}^0\mathbf{p}_{m0}\Big)^{\mathrm{T}}\mathbf{v}_f\ . \tag{8.244}$$

The generalized (elastic) forces thus result from

$$\mathbf{Q}_{SD} = \frac{\partial\,{}^0\mathbf{p}}{\partial\mathbf{q}_{SD}^{\mathrm{T}}}\mathbf{f}\ , \tag{8.245}$$

and read for the markers $m0$ and $m1$,

$$\mathbf{Q}_{SD,m0} = -\Big(k \cdot (L - L_0) + d \cdot (\dot{L} - \dot{L}_0) + f_a\Big)\mathbf{J}_{pos,m0}^{\mathrm{T}}\mathbf{v}_f, \quad \mathbf{Q}_{SD,m1} = \Big(k \cdot (L - L_0) + d \cdot (\dot{L} - \dot{L}_0) + f_a\Big)\mathbf{J}_{pos,m1}^{\mathrm{T}}\mathbf{v}_f\ , \tag{8.246}$$

where $\mathbf{J}_{pos,m1}$ represents the derivative of marker $m1$ w.r.t. its associated coordinates $\mathbf{q}_{m1}$, analogously $\mathbf{J}_{pos,m0}$.

8.6.1.4 Connector Jacobian

The position-level jacobian for the connector, involving all coordinates associated with markers $m0$ and $m1$, follows from

$$\mathbf{J}_{SD} = \begin{bmatrix} \frac{\partial \mathbf{Q}_{SD,m0}}{\partial \mathbf{q}_{m0}} & \frac{\partial \mathbf{Q}_{SD,m0}}{\partial \mathbf{q}_{m1}} \\ \frac{\partial \mathbf{Q}_{SD,m0}}{\partial \mathbf{q}_{m1}} & \frac{\partial \mathbf{Q}_{SD,m1}}{\partial \mathbf{q}_{m1}} \end{bmatrix} \tag{8.247}$$

and the velocity level jacobian reads

$$\mathbf{J}_{SD,t} = \begin{bmatrix} \frac{\partial \mathbf{Q}_{SD,m0}}{\partial \dot{\mathbf{q}}_{m0}} & \frac{\partial \mathbf{Q}_{SD,m0}}{\partial \dot{\mathbf{q}}_{m1}} \\ \frac{\partial \mathbf{Q}_{SD,m0}}{\partial \dot{\mathbf{q}}_{m1}} & \frac{\partial \mathbf{Q}_{SD,m1}}{\partial \dot{\mathbf{q}}_{m1}} \end{bmatrix} \tag{8.248}$$

The sub-Jacobians follow from

$$\frac{\partial \mathbf{Q}_{SD,m0}}{\partial \mathbf{q}_{m0}} = -\frac{\partial \mathbf{J}_{pos,m0}^{\mathrm{T}}}{\partial \mathbf{q}_{m0}} \mathbf{v}_f \left(k \cdot (L - L_0) + d \cdot (\dot{L} - \dot{L}_0) + f_a \right) - \mathbf{J}_{pos,m0}^{\mathrm{T}} \frac{\partial \mathbf{v}_f \left(k \cdot (L - L_0) + d \cdot (\dot{L} - \dot{L}_0) + f_a \right)}{\partial \mathbf{q}_{m0}} \tag{8.249}$$

in which the term $\frac{\partial \mathbf{J}_{pos,m0}^{\mathrm{T}}}{\partial \mathbf{q}_{m0}}$ is computed from a special function provided by markers, that compute the derivative of the marker jacobian times a constant vector, in this case the spring force $\mathbf{f}$; this jacobian term is usually less dominant, but is included in the numerical as well as the analytical derivatives, see the general jacobian computation information.

The other term, which is the dominant term, is computed as (dependence of velocity term on position coordinates and $\dot{L}_0$ term neglected),

$$\begin{aligned} \frac{\partial \mathbf{Q}_{SD,m0}}{\partial \mathbf{q}_{m0}} &= -\mathbf{J}_{pos,m0}^{\mathrm{T}} \frac{\partial \mathbf{v}_f \left(k \cdot (L - L_0) + d \cdot (\dot{L} - \dot{L}_0) + f_a \right)}{\partial \mathbf{q}_{m0}} \\ &= -\mathbf{J}_{pos,m0}^{\mathrm{T}} \frac{\partial \left(k \cdot \left(\Delta^0 \mathbf{p} - L_0 \mathbf{v}_f \right) + \mathbf{v}_f \left(d \cdot \mathbf{v}_f^{\mathrm{T}} \Delta^0 \mathbf{v} + f_a \right) \right)}{\partial \mathbf{q}_{m0}} \\ &\approx \mathbf{J}_{pos,m0}^{\mathrm{T}} \left(k \cdot \mathbf{I} - k \frac{L_0}{L} \left(\mathbf{I} - {}^0\mathbf{v}_f \otimes {}^0\mathbf{v}_f \right) + \frac{1}{L} \left(\mathbf{I} - {}^0\mathbf{v}_f \otimes {}^0\mathbf{v}_f \right) \left(d \cdot \mathbf{v}_f^{\mathrm{T}} \Delta^0 \mathbf{v} + f_a \right) \right. \\ &\quad \left. + d \, {}^0\mathbf{v}_f \otimes \left(\frac{1}{L} \left(\mathbf{I} - {}^0\mathbf{v}_f \otimes {}^0\mathbf{v}_f \right) {}^0\mathbf{v}_f \right) \right) {}^0\mathbf{J}_{pos,m0} \end{aligned} \tag{8.250}$$

Alternatively (again $\dot{L}_0$ term neglected):

$$\begin{aligned} \frac{\partial \mathbf{Q}_{SD,m0}}{\partial \mathbf{q}_{m0}} &= -\mathbf{J}_{pos,m0}^{\mathrm{T}} \frac{\partial \mathbf{v}_f \left(k \cdot (L - L_0) + d \cdot (\dot{L} - \dot{L}_0) + f_a \right)}{\partial \mathbf{q}_{m0}} \\ &= \mathbf{J}_{pos,m0}^{\mathrm{T}} \frac{1}{L} \left(\mathbf{I} - {}^0\mathbf{v}_f \otimes {}^0\mathbf{v}_f \right) \left(k \cdot (L - L_0) + d \cdot (\dot{L} - \dot{L}_0) + f_a \right) \mathbf{J}_{pos,m0} \\ &\quad + \mathbf{J}_{pos,m0}^{\mathrm{T}} \, {}^0\mathbf{v}_f \otimes \left(k \cdot {}^0\mathbf{v}_f + d \cdot \Delta^0 \mathbf{v} \frac{1}{L} \left(\mathbf{I} - {}^0\mathbf{v}_f \otimes {}^0\mathbf{v}_f \right) \right) \mathbf{J}_{pos,m0} - d \mathbf{J}_{pos,m0}^{\mathrm{T}} \, {}^0\mathbf{v}_f \otimes {}^0\mathbf{v}_f \frac{\partial \Delta^0 \mathbf{v}}{\partial \mathbf{q}_{m0}} \\ &= \mathbf{J}_{pos,m0}^{\mathrm{T}} \left(\frac{f_{SD}}{L} \left(\mathbf{I} - {}^0\mathbf{v}_f \otimes {}^0\mathbf{v}_f \right) + k \, {}^0\mathbf{v}_f \otimes {}^0\mathbf{v}_f + \frac{d}{L} \left({}^0\mathbf{v}_f \otimes \Delta^0 \mathbf{v} \right) \cdot \left(\mathbf{I} - {}^0\mathbf{v}_f \otimes {}^0\mathbf{v}_f \right) + ...! \right) \mathbf{J}_{pos,m0} \end{aligned} \tag{8.251}$$

Noting that $\frac{\partial \mathbf{v}_f}{\partial \mathbf{q}_{m0}} = -\frac{1}{L} \left(\mathbf{I} - {}^0\mathbf{v}_f \otimes {}^0\mathbf{v}_f \right) {}^0\mathbf{J}_{pos,m0}$ and $\frac{\partial \mathbf{v}_f}{\partial \mathbf{q}_{m1}} = \frac{1}{L} \left(\mathbf{I} - {}^0\mathbf{v}_f \otimes {}^0\mathbf{v}_f \right) {}^0\mathbf{J}_{pos,m1}$. The Jacobian w.r.t.

velocity coordinates follows as

$$\begin{aligned}\frac{\partial \mathbf{Q}_{SD,m0}}{\partial \dot{\mathbf{q}}_{m0}} &= -\mathbf{J}_{pos,m0}^{\mathrm{T}} \frac{\partial \mathbf{v}_f \left(k \cdot (L - L_0) + d \cdot (\dot{L} - \dot{L}_0) + f_a\right)}{\partial \dot{\mathbf{q}}_{m0}} \\ &= \mathbf{J}_{pos,m0}^{\mathrm{T}} \left(d \mathbf{v}_f \otimes \mathbf{v}_f\right) {}^0\mathbf{J}_{pos,m0} \end{aligned} \tag{8.252}$$

Note that in case that $L = 0$, the term $\frac{1}{L}\left(\mathbf{I} - {}^0\mathbf{v}_f \otimes {}^0\mathbf{v}_f\right)$ is replaced by the unit matrix, in order to avoid zero (singular) jacobian; this is a workaround and should only occur in exceptional cases.

The term $\frac{\partial \Delta^0 \mathbf{v}}{\partial \mathbf{q}_{m0}}$, which is important for large damping, yields

$$\frac{\partial \Delta^0 \mathbf{v}}{\partial \mathbf{q}_{m0}} = \frac{\partial \mathbf{J}_{pos,m0} \dot{\mathbf{q}}_{m0}}{\partial \mathbf{q}_{m0}} = \frac{\partial \mathbf{J}_{pos,m0}}{\partial \mathbf{q}_{m0}} \dot{\mathbf{q}}_{m0} \tag{8.253}$$

The latter term is currently neglected.

Jacobians for markers $m1$ and mixed $m0/m1$ terms follow analogously.

Userfunction: `springForceUserFunction(mbs, t, itemNumber, deltaL, deltaL_t, stiffness, damping, force)`

A user function, which computes the spring force depending on time, object variables (deltaL, deltaL_t) and object parameters (stiffness, damping, force). The object variables are provided to the function using the current values of the SpringDamper object. Note that itemNumber represents the index of the object in mbs, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs to which object belongs
`t`	Real	current time in mbs
`itemNumber`	Index	integer number i_N of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
`deltaL`	Real	$L - L_0$, spring elongation
`deltaL_t`	Real	$(\dot{L} - \dot{L}_0)$, spring velocity, including offset
`stiffness`	Real	copied from object
`damping`	Real	copied from object
`force`	Real	copied from object; constant force
`return value`	Real	scalar value of computed spring force

User function example:

```
#define nonlinear force
def UFforce(mbs, t, itemNumber, u, v, k, d, F0):
    return k*u + d*v + F0
```

```
#markerNumbers taken from mini example
mbs.AddObject(ObjectConnectorSpringDamper(markerNumbers=[m0,m1],
                                          referenceLength = 1,
                                          stiffness = 100, damping = 1,
                                          springForceUserFunction = UFforce))
```

8.6.1.5 MINI EXAMPLE for ObjectConnectorSpringDamper

```
node = mbs.AddNode(NodePoint(referenceCoordinates = [1.05,0,0]))
oMassPoint = mbs.AddObject(MassPoint(nodeNumber = node, physicsMass=1))

m0 = mbs.AddMarker(MarkerBodyPosition(bodyNumber=oGround, localPosition=[0,0,0]))
m1 = mbs.AddMarker(MarkerBodyPosition(bodyNumber=oMassPoint, localPosition=[0,0,0]))

mbs.AddObject(ObjectConnectorSpringDamper(markerNumbers=[m0,m1],
                                          referenceLength = 1, #shorter than initial
distance
                                          stiffness = 100,
                                          damping = 1))

#assemble and solve system for default parameters
mbs.Assemble()
mbs.SolveDynamic()

#check result at default integration time
exudynTestGlobals.testResult = mbs.GetNodeOutput(node, exu.OutputVariableType.Position)
[0]
```

For examples on ObjectConnectorSpringDamper see Relevant Examples and TestModels with weblink:

- HydraulicsUserFunction.py (Examples/)
- SpringDamperMassUserFunction.py (Examples/)
- ballBearningModel.py (Examples/)
- basicTutorial2024.py (Examples/)
- camFollowerExample.py (Examples/)
- chatGPTupdate.py (Examples/)
- contactCurveWithLongCurve.py (Examples/)
- springDamperTutorialNew.py (Examples/)
- springMassFriction.py (Examples/)

- symbolicUserFunctionMasses.py (Examples/)
- tutorialNeuralNetwork.py (Examples/)
- ANCFcontactCircle.py (Examples/)
- ...
- createFunctionsTest.py (TestModels/)
- loadUserFunctionTest.py (TestModels/)
- mainSystemExtensionsTests.py (TestModels/)
- ...

8.6.2 ObjectConnectorCartesianSpringDamper

An 3D spring-damper element, providing springs and dampers in three (global) directions (x,y,z); the connector can be attached to position-based markers.

Additional information for ObjectConnectorCartesianSpringDamper:

- This `Object` has/provides the following types = `Connector`
- Requested `Marker` type = `Position`
- **Short name** for Python = `CartesianSpringDamper`
- **Short name** for Python visualization object = `VCartesianSpringDamper`

The item **ObjectConnectorCartesianSpringDamper** with type = 'ConnectorCartesianSpringDamper' has the following parameters:

Name	type	size	default value	description
name	String		''	connector's unique name
markerNumbers	ArrayMarkerIndex		[invalid [-1], invalid [-1]]	list of markers used in connector
stiffness	Vector3D		[0.,0.,0.]	stiffness [SI:N/m] of springs; act against relative displacements in 0, 1, and 2-direction
damping	Vector3D		[0.,0.,0.]	damping [SI:N/(m s)] of dampers; act against relative velocities in 0, 1, and 2-direction
offset	Vector3D		[0.,0.,0.]	offset between two springs
springForceUserFunction	PyFunctionVector3DmbsScalarIndexScalar4Vector3D		0	A Python function which computes the 3D force vector between the two marker points, if activeConnector=True; see description below
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectConnectorCartesianSpringDamper			parameters for visualization of item

The item VObjectConnectorCartesianSpringDamper has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

drawSize	float		-1.	drawing size = diameter of spring; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

---

8.6.2.1 DESCRIPTION of ObjectConnectorCartesianSpringDamper:

Information on input parameters:

input parameter	symbol	description see tables above
markerNumbers	$[m0, m1]^\mathrm{T}$	
stiffness	$\mathbf{k}$	
damping	$\mathbf{d}$	
offset	$\mathbf{v}_{\mathrm{off}}$	
springForceUserFunction	UF $\in \mathbb{R}^3$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Displacement	$\Delta^0\mathbf{p} = {}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0}$	relative displacement in global coordinates
Distance	$L = \lvert\Delta^0\mathbf{p}\rvert$	scalar distance between both marker points
Velocity	$\Delta^0\mathbf{v} = {}^0\mathbf{v}_{m1} - {}^0\mathbf{v}_{m0}$	relative translational velocity in global coordinates
Force	$\mathbf{f}_{SD}$	joint force in global coordinates, see equations

8.6.2.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker m0
marker m1 position	${}^0\mathbf{p}_{m1}$	
marker m0 velocity	${}^0\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
marker m1 velocity	${}^0\mathbf{v}_{m1}$	

8.6.2.3 Connector forces

Connector forces are based on relative displacements and relative veolocities in global coordinates. Relative displacement between marker m0 to marker m1 positions is given by

$$\Delta^0\mathbf{p} = {}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0} , \tag{8.254}$$

and relative velocity reads

$$\Delta^0\mathbf{v} = {}^0\mathbf{v}_{m1} - {}^0\mathbf{v}_{m0} . \tag{8.255}$$

If `activeConnector = True`, the spring force vector is computed as

$${}^0\mathbf{f}_{SD} = \text{diag}(\mathbf{k}) \cdot (\Delta^0\mathbf{p} - {}^0\mathbf{v}_{\text{off}}) + \text{diag}(\mathbf{d}) \cdot \Delta^0\mathbf{v} . \tag{8.256}$$

If the springForceUserFunction UF is defined, $\mathbf{f}_{SD}$ instead becomes (t is current time)

$${}^0\mathbf{f}_{SD} = \text{UF}(mbs, t, i_N, \Delta^0\mathbf{p}, \Delta^0\mathbf{v}, \mathbf{k}, \mathbf{d}, \mathbf{v}_{\text{off}}) , \tag{8.257}$$

and `iN` represents the itemNumber (=objectNumber). If `activeConnector = False`, $\mathbf{f}_{SD}$ is set to zero.

The force $\mathbf{f}_{SD}$ acts via the markers' position jacobians $\mathbf{J}_{pos,m0}$ and $\mathbf{J}_{pos,m1}$. The generalized forces added to the LHS equations read for marker $m0$,

$$\mathbf{f}_{LHS,m0} = -{}^0\mathbf{J}^{\text{T}}_{pos,m0}\ {}^0\mathbf{f}_{SD} , \tag{8.258}$$

and for marker $m1$,

$$\mathbf{f}_{LHS,m1} = {}^0\mathbf{J}^{\text{T}}_{pos,m1}\ {}^0\mathbf{f}_{SD} . \tag{8.259}$$

The LHS equation parts are added accordingly using the LTG mapping. Note that the different signs result from the signs in Eq. (8.254).

The connector also provides an analytic jacobian, which is used if `newton.numericalDifferentiation.forODE` = `False` and if there is no springForceUserFunction (otherwise numerical differentiation is used).

The anayltic jacobian for the coupled equation parts $\mathbf{f}_{LHS,m0}$ and $\mathbf{f}_{LHS,m1}$ is based on the local jacobians

$$\begin{aligned}
\mathbf{J}_{loc0} &= f_{ODE2}\frac{\partial\,{}^0\mathbf{f}_{SD}}{\partial\,{}^0\mathbf{p}_{m0}} + f_{ODE2_t}\frac{\partial\,{}^0\mathbf{f}_{SD}}{\partial\,{}^0\mathbf{v}_{m0}} = -f_{ODE2}\cdot\text{diag}(\mathbf{k}) - f_{ODE2_t}\cdot\text{diag}(\mathbf{d}) , \\
\mathbf{J}_{loc1} &= f_{ODE2}\frac{\partial\,{}^0\mathbf{f}_{SD}}{\partial\,{}^0\mathbf{p}_{m1}} + f_{ODE2_t}\frac{\partial\,{}^0\mathbf{f}_{SD}}{\partial\,{}^0\mathbf{v}_{m1}} = f_{ODE2}\cdot\text{diag}(\mathbf{k}) + f_{ODE2_t}\cdot\text{diag}(\mathbf{d}) .
\end{aligned} \tag{8.260}$$

Here, f_{ODE2} is the factor for the position derivative and f_{ODE2_t} is the factor for the velocity derivative, which allows a computation of the computation for both the position as well as the velocity part at the same time.

The complete jacobian for the LHS equations then reads,

$$\begin{aligned}
\mathbf{J}_{CSD} &= \begin{bmatrix} \frac{\partial \mathbf{f}_{LHS,m0}}{\partial \mathbf{q}_{m0}} & \frac{\partial \mathbf{f}_{LHS,m0}}{\partial \mathbf{q}_{m1}} \\ \frac{\partial \mathbf{f}_{LHS,m1}}{\partial \mathbf{q}_{m0}} & \frac{\partial \mathbf{f}_{LHS,m1}}{\partial \mathbf{q}_{m1}} \end{bmatrix} + \mathbf{J}_{CSD'} \\
&= \begin{bmatrix} -{}^0\mathbf{J}_{pos,m0}^{\mathrm{T}} \mathbf{J}_{loc0} \mathbf{J}_{pos,m0} & -{}^0\mathbf{J}_{pos,m0}^{\mathrm{T}} \mathbf{J}_{loc1} \mathbf{J}_{pos,m1} \\ {}^0\mathbf{J}_{pos,m1}^{\mathrm{T}} \mathbf{J}_{loc0} \mathbf{J}_{pos,m0} & {}^0\mathbf{J}_{pos,m1}^{\mathrm{T}} \mathbf{J}_{loc1} \mathbf{J}_{pos,m1} \end{bmatrix} + \mathbf{J}_{CSD'} \\
&= \begin{bmatrix} {}^0\mathbf{J}_{pos,m0}^{\mathrm{T}} \mathbf{J}_{loc1} \mathbf{J}_{pos,m0} & -{}^0\mathbf{J}_{pos,m0}^{\mathrm{T}} \mathbf{J}_{loc1} \mathbf{J}_{pos,m1} \\ -{}^0\mathbf{J}_{pos,m1}^{\mathrm{T}} \mathbf{J}_{loc1} \mathbf{J}_{pos,m0} & {}^0\mathbf{J}_{pos,m1}^{\mathrm{T}} \mathbf{J}_{loc1} \mathbf{J}_{pos,m1} \end{bmatrix} + \mathbf{J}_{CSD'}
\end{aligned} \tag{8.261}$$

Here, $\mathbf{q}_{m0}$ are the coordinates associated with marker $m0$ and $\mathbf{q}_{m1}$ of marker $m1$.

The second term $\mathbf{J}_{CSD'}$ is only non-zero if $\frac{\partial\, {}^0\mathbf{J}_{pos,i}^{\mathrm{T}}}{\partial \mathbf{q}_i}$ is non-zero, using $i \in \{m0, m1\}$. As the latter terms would require to compute a 3-dimensional array, the second jacobian term is computed as

$$\mathbf{J}_{CSD'} = \begin{bmatrix} -f_{ODE2} \frac{\partial \left({}^0\mathbf{J}_{pos,m0}^{\mathrm{T}} \mathbf{f}' \right)}{\partial \mathbf{q}_{m0}} & \mathbf{0} \\ \mathbf{0} & f_{ODE2} \frac{\partial \left({}^0\mathbf{J}_{pos,m1}^{\mathrm{T}} \mathbf{f}' \right)}{\partial \mathbf{q}_{m1}} \end{bmatrix} \tag{8.262}$$

in which we set $\mathbf{f}' = {}^0\mathbf{f}_{SD}$, but the derivatives in Eq. (8.262) are evaluated by setting $\mathbf{f}' = const$.

Userfunction: `springForceUserFunction(mbs, t, itemNumber, displacement, velocity, stiffness, damping, offset)`

A user function, which computes the 3D spring force vector depending on time, object variables (deltaL, deltaL_t) and object parameters (stiffness, damping, force). The object variables are provided to the function using the current values of the SpringDamper object. Note that itemNumber represents the index of the object in mbs, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs in which underlying item is defined
`t`	Real	current time in mbs
`itemNumber`	Index	integer number i_N of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
`displacement`	Vector3D	$\Delta^0\mathbf{p}$
`velocity`	Vector3D	$\Delta^0\mathbf{v}$
`stiffness`	Vector3D	copied from object
`damping`	Vector3D	copied from object
`offset`	Vector3D	copied from object

return value	Vector3D	list or numpy array of computed spring force

User function example:

```
#define simple force for spring-damper:
def UFforce(mbs, t, itemNumber, u, v, k, d, offset):
    return [u[0]*k[0],u[1]*k[1],u[2]*k[2]]

#markerNumbers and parameters taken from mini example
mbs.AddObject(CartesianSpringDamper(markerNumbers = [mGround, mMass],
                                    stiffness = [k,k,k],
                                    damping = [0,k*0.05,0], offset = [0,0,0],
                                    springForceUserFunction = UFforce))
```

8.6.2.4 MINI EXAMPLE for ObjectConnectorCartesianSpringDamper

```
#example with mass at [1,1,0], 5kg under load 5N in -y direction
k=5000
nMass = mbs.AddNode(NodePoint(referenceCoordinates=[1,1,0]))
oMass = mbs.AddObject(MassPoint(physicsMass = 5, nodeNumber = nMass))

mMass = mbs.AddMarker(MarkerNodePosition(nodeNumber=nMass))
mGround = mbs.AddMarker(MarkerBodyPosition(bodyNumber=oGround, localPosition = [1,1,0]))
mbs.AddObject(CartesianSpringDamper(markerNumbers = [mGround, mMass],
                                    stiffness = [k,k,k],
                                    damping = [0,k*0.05,0], offset = [0,0,0]))
mbs.AddLoad(Force(markerNumber = mMass, loadVector = [0, -5, 0])) #static solution
=-5/5000=-0.001m

#assemble and solve system for default parameters
mbs.Assemble()
mbs.SolveDynamic()

#check result at default integration time
exudynTestGlobals.testResult = mbs.GetNodeOutput(nMass, exu.OutputVariableType.
Displacement)[1]
```

For examples on ObjectConnectorCartesianSpringDamper see Relevant Examples and TestModels with weblink:

- mouseInteractionExample.py (Examples/)

- rigid3Dexample.py (Examples/)
- sliderCrank3DwithANCFbeltDrive2.py (Examples/)
- cartesianSpringDamper.py (Examples/)
- cartesianSpringDamperUserFunction.py (Examples/)
- chatGPTupdate.py (Examples/)
- ANCFcontactCircle.py (Examples/)
- ANCFcontactCircle2.py (Examples/)
- ANCFslidingJoint2D.py (Examples/)
- flexibleRotor3Dtest.py (Examples/)
- lavalRotor2Dtest.py (Examples/)
- NGsolvePistonEngine.py (Examples/)
- ...
- scissorPrismaticRevolute2D.py (TestModels/)
- sphericalJointTest.py (TestModels/)
- complexEigenvaluesTest.py (TestModels/)
- ...

8.6.3 ObjectConnectorRigidBodySpringDamper

An 3D spring-damper element acting on relative displacements and relative rotations of two rigid body (position+orientation) markers; connects to (position+orientation)-based markers and represents a penalty-based rigid joint (or prismatic, revolute, etc.)

Additional information for ObjectConnectorRigidBodySpringDamper:

- This `Object` has/provides the following types = `Connector`
- Requested `Marker` type = `Position` + `Orientation`
- Requested `Node` type = `GenericData`
- **Short name** for Python = `RigidBodySpringDamper`
- **Short name** for Python visualization object = `VRigidBodySpringDamper`

The item **ObjectConnectorRigidBodySpringDamper** with type = 'ConnectorRigidBodySpringDamper' has the following parameters:

Name	type	size	default value	description
name	String		''	connector's unique name
markerNumbers	ArrayMarkerIndex		[invalid [-1], invalid [-1]]	list of markers used in connector
nodeNumber	NodeIndex		invalid (-1)	node number of a NodeGenericData (size depends on application) for dataCoordinates for user functions (e.g., implementing contact/friction user function)
stiffness	Matrix6D		np.zeros((6,6))	stiffness [SI:N/m or Nm/rad] of translational, torsional and coupled springs; act against relative displacements in x, y, and z-direction as well as the relative angles (calculated as Euler angles); in the simplest case, the first 3 diagonal values correspond to the local stiffness in x,y,z direction and the last 3 diagonal values correspond to the rotational stiffness around x,y and z axis
damping	Matrix6D		np.zeros((6,6))	damping [SI:N/(m/s) or Nm/(rad/s)] of translational, torsional and coupled dampers; very similar to stiffness, however, the rotational velocity is computed from the angular velocity vector
rotationMarker0	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	local rotation matrix for marker 0; stiffness, damping, etc. components are measured in local coordinates relative to rotationMarker0

rotationMarker1	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	local rotation matrix for marker 1; stiffness, damping, etc. components are measured in local coordinates relative to rotationMarker1
offset	Vector6D		[0.,0.,0.,0.,0.,0.]	translational and rotational offset considered in the spring force calculation
intrinsicFormulation	Bool		False	if True, the joint uses the intrinsic formulation, which is independent on order of markers, using a mid-point and mid-rotation for evaluation and application of connector forces and torques; this uses a Lie group formulation; in this case, the force/-torque vector is computed from the stiffness matrix times the 6-vector of the SE3 matrix logarithm between the two marker positions/rotations, see the equations
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
springForceTorqueUserFunction	PyFunctionVector6DmbsScalarIndex4Vector3D2Matrix6D2Matrix3DVector6D		0	A Python function which computes the 6D force-torque vector (3D force + 3D torque) between the two rigid body markers, if activeConnector=True; see description below
postNewtonStepUserFunction	PyFunctionVectorMbsScalarIndex4VectorVector3D2Matrix6D2Matrix3DVector6D		0	A Python function which computes the error of the PostNewtonStep; see description below
visualization	VObjectConnectorRigidBodySpringDamper			parameters for visualization of item

The item VObjectConnectorRigidBodySpringDamper has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = diameter of spring; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

8.6.3.1 DESCRIPTION of ObjectConnectorRigidBodySpringDamper:

Information on input parameters:

input parameter	symbol	description see tables above
nodeNumber	n_d	
springForceTorqueUserFunction	$\mathrm{UF} \in \mathbb{R}^6$	
postNewtonStepUserFunction	$\mathrm{UF}_{PN} \in \mathbb{R}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
DisplacementLocal	${}^{J0}\Delta\mathbf{p}$	relative displacement in local joint0 coordinates
VelocityLocal	${}^{J0}\Delta\mathbf{v}$	relative translational velocity in local joint0 coordinates
Rotation	${}^{J0}\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2]^{\mathrm{T}}$	relative rotation parameters (Tait Bryan Rxyz); these are the angles used for calculation of joint torques (e.g. if cX is the diagonal rotational stiffness, the moment for axis X reads mX=cX*phiX, etc.)
AngularVelocityLocal	${}^{J0}\Delta\boldsymbol{\omega}$	relative angular velocity in local joint0 coordinates
ForceLocal	${}^{J0}\mathbf{f}$	joint force in local joint0 coordinates
TorqueLocal	${}^{J0}\mathbf{m}$	joint torque in in local joint0 coordinates

8.6.3.2 Definition of quantities

input parameter	symbol	description
stiffness	$\mathbf{k} \in \mathbb{R}^{6\times6}$	stiffness in $J0$ coordinates
damping	$\mathbf{d} \in \mathbb{R}^{6\times6}$	damping in $J0$ coordinates
offset	${}^{J0}\mathbf{v}_{\mathrm{off}} \in \mathbb{R}^6$	offset in $J0$ coordinates
rotationMarker0	${}^{m0,J0}\mathbf{A}$	rotation matrix which transforms from joint 0 into marker 0 coordinates
rotationMarker1	${}^{m1,J1}\mathbf{A}$	rotation matrix which transforms from joint 1 into marker 1 coordinates
markerNumbers[0]	$m0$	global marker number m0
markerNumbers[1]	$m1$	global marker number m1

intermediate variables	symbol	description
marker m0 position	${}^{0}\mathbf{p}_{m0}$	current global position which is provided by marker m0
marker m0 orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker m0
marker m1 position	${}^{0}\mathbf{p}_{m1}$	accordingly

marker m1 orientation	${}^{0,m1}\mathbf{A}$	current rotation matrix provided by marker m1
marker m0 velocity	${}^{0}\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
marker m1 velocity	${}^{0}\mathbf{v}_{m1}$	accordingly
marker m0 velocity	${}^{m0}\boldsymbol{\omega}_{m0}$	current local angular velocity vector provided by marker m0
marker m1 velocity	${}^{m1}\boldsymbol{\omega}_{m1}$	current local angular velocity vector provided by marker m1
Displacement	${}^{0}\Delta\mathbf{p}$	${}^{0}\mathbf{p}_{m1} - {}^{0}\mathbf{p}_{m0}$
Velocity	${}^{0}\Delta\mathbf{v}$	${}^{0}\mathbf{v}_{m1} - {}^{0}\mathbf{v}_{m0}$
DisplacementLocal	${}^{J0}\Delta\mathbf{p}$	$\left({}^{0,m0}\mathbf{A}\ {}^{m0,J0}\mathbf{A}\right)^{\mathrm{T}}\ {}^{0}\Delta\mathbf{p}$
VelocityLocal	${}^{J0}\Delta\mathbf{v}$	$\left({}^{0,m0}\mathbf{A}\ {}^{m0,J0}\mathbf{A}\right)^{\mathrm{T}}\ {}^{0}\Delta\mathbf{v}$
AngularVelocityLocal	${}^{J0}\Delta\boldsymbol{\omega}$	$\left({}^{0,m0}\mathbf{A}\ {}^{m0,J0}\mathbf{A}\right)^{\mathrm{T}}\left({}^{0,m1}\mathbf{A}\ {}^{m1}\boldsymbol{\omega} - {}^{0,m0}\mathbf{A}\ {}^{m0}\boldsymbol{\omega}\right)$

8.6.3.3 Connector forces

If `activeConnector = True`, the vector spring force is computed as

$$\begin{bmatrix} {}^{J0}\mathbf{f}_{SD} \\ {}^{J0}\mathbf{m}_{SD} \end{bmatrix} = \mathbf{k}\left(\begin{bmatrix} {}^{J0}\Delta\mathbf{p} \\ {}^{J0}\boldsymbol{\theta} \end{bmatrix} - {}^{J0}\mathbf{v}_{\text{off}}\right) + \mathbf{d}\begin{bmatrix} {}^{J0}\Delta\mathbf{v} \\ {}^{J0}\Delta\omega \end{bmatrix} \tag{8.263}$$

For the application of joint forces to markers, $[{}^{J0}\mathbf{f}_{SD}, {}^{J0}\mathbf{m}_{SD}]^{\mathrm{T}}$ is transformed into global coordinates. if `activeConnector = False`, ${}^{J0}\mathbf{f}_{SD}$ and ${}^{J0}\mathbf{m}_{SD}$ are set to zero.

If the springForceTorqueUserFunction UF is defined and `activeConnector = True`, $\mathbf{f}_{SD}$ instead becomes (t is current time)

$$\mathbf{f}_{SD} = \text{UF}(mbs, t, i_N, {}^{J0}\Delta\mathbf{p}, {}^{J0}\boldsymbol{\theta}, {}^{J0}\Delta\mathbf{v}, {}^{J0}\Delta\boldsymbol{\omega}, \text{stiffness}, \text{damping}, \text{rotationMarker0}, \text{rotationMarker1}, \text{offset}) \tag{8.264}$$

and `iN` represents the itemNumber (=objectNumber).

Userfunction: `springForceTorqueUserFunction(mbs, t, itemNumber, displacement, rotation, velocity, angularVelocity, stiffness, damping, rotJ0, rotJ1, offset)`

A user function, which computes the 6D spring-damper force-torque vector depending on mbs, time, local quantities (displacement, rotation, velocity, angularVelocity, stiffness), which are evaluated at current time, which are relative quantities between both markers and which are defined in joint J0 coordinates. As relative rotations are defined by Tait-Bryan rotation parameters, it is recommended to use this connector for small relative rotations only (except for rotations about one axis). Furthermore, the user function contains object parameters (stiffness, damping, rotationMarker0/1, offset). Note that itemNumber represents the index of the object in mbs, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

Detailed description of the arguments and local quantities:

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs in which underlying item is defined
`t`	Real	current time in mbs
`itemNumber`	Index	integer number i_N of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
`displacement`	Vector3D	${}^{J0}\Delta\mathbf{p}$
`rotation`	Vector3D	${}^{J0}\boldsymbol{\theta}$
`velocity`	Vector3D	${}^{J0}\Delta\mathbf{v}$
`angularVelocity`	Vector3D	${}^{J0}\Delta\boldsymbol{\omega}$
`stiffness`	Vector6D	copied from object
`damping`	Vector6D	copied from object
`rotJ0`	Matrix3D	rotationMarker0 copied from object
`rotJ1`	Matrix3D	rotationMarker1 copied from object
`offset`	Vector6D	copied from object
`return value`	Vector6D	list or numpy array of computed spring force-torque

---

Userfunction: `postNewtonStepUserFunction(mbs, t, Index itemIndex, dataCoordinates, displacement, rotation, velocity, angularVelocity, stiffness, damping, rotJ0, rotJ1, offset)`

A user function which computes the error of the PostNewtonStep ε_{PN}, a recommended for stepsize reduction t_{recom} (use values > 0 to recommend step size or values < 0 else; 0 gives minimum step size) and the updated dataCoordinates $\mathbf{d}^k$ of `NodeGenericData` n_d. Except from `dataCoordinates`, the arguments are the same as in `springForceTorqueUserFunction`. The `postNewtonStepUserFunction` should be used together with the dataCoordinates in order to implement a active set or switching strategy for discontinuous events, such as in contact, friction, plasticity, fracture or similar.

Detailed description of the arguments and local quantities:

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs in which underlying item is defined
`t`	Real	current time in mbs
`itemNumber`	Index	integer number of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
`dataCoordinates`	Vector	$\mathbf{d}^{k-1} = [d_0^{k-1}, d_1^{k-1}, \ldots]$ for previous post Newton step $k-1$
...	...	other arguements see `springForceTorqueUserFunction`
`return value`	Vector	$\left[\varepsilon_{PN}, t_{recom}, d_0^k, d_1^k, ...\right]$ where k indicates the current step

---

User function example:

```
#define simple force for spring-damper:
def UFforce(mbs, t, itemNumber, displacement, rotation, velocity, angularVelocity,
            stiffness, damping, rotJ0, rotJ1, offset):
    k = stiffness #passed as list
    u = displacement
    return [u[0]*k[0][0],u[1]*k[1][1],u[2]*k[2][2], 0,0,0]

#markerNumbers and parameters taken from mini example
mbs.AddObject(RigidBodySpringDamper(markerNumbers = [mGround, mBody],
                                    stiffness = np.diag([k,k,k, 0,0,0]),
                                    damping = np.diag([0,k*0.01,0, 0,0,0]),
                                    offset = [0,0,0, 0,0,0],
                                    springForceTorqueUserFunction = UFforce))
```

8.6.3.4 MINI EXAMPLE for ObjectConnectorRigidBodySpringDamper

```
#example with rigid body at [0,0,0], 1kg under initial velocity
k=500
nBody = mbs.AddNode(RigidRxyz(initialVelocities=[0,1e3,0, 0,0,0]))
oBody = mbs.AddObject(RigidBody(physicsMass=1, physicsInertia=[1,1,1,0,0,0],
                                nodeNumber=nBody))

mBody = mbs.AddMarker(MarkerNodeRigid(nodeNumber=nBody))
mGround = mbs.AddMarker(MarkerBodyRigid(bodyNumber=oGround,
                                        localPosition = [0,0,0]))
mbs.AddObject(RigidBodySpringDamper(markerNumbers = [mGround, mBody],
                                    stiffness = np.diag([k,k,k, 0,0,0]),
                                    damping = np.diag([0,k*0.01,0, 0,0,0]),
                                    offset = [0,0,0, 0,0,0]))

#assemble and solve system for default parameters
mbs.Assemble()
mbs.SolveDynamic(exu.SimulationSettings())

#check result at default integration time
exudynTestGlobals.testResult = mbs.GetNodeOutput(nBody, exu.OutputVariableType.
Displacement)[1]
```

For examples on ObjectConnectorRigidBodySpringDamper see Relevant Examples and TestModels with weblink:

- ROSMassPoint.py (Examples/)

- ROSMobileManipulator.py (Examples/)
- stiffFlyballGovernor2.py (Examples/)
- bricardMechanism.py (TestModels/)
- rigidBodySpringDamperIntrinsic.py (TestModels/)
- connectorRigidBodySpringDamperTest.py (TestModels/)
- rotatingTableTest.py (TestModels/)
- stiffFlyballGovernor.py (TestModels/)
- superElementRigidJointTest.py (TestModels/)

8.6.4 ObjectConnectorLinearSpringDamper

An linear spring-damper element acting on relative translations along given axis of local joint0 coordinate system; connects to position and orientation-based markers; the linear spring-damper is intended to act within prismatic joints or in situations where only one translational axis is free; if the two markers rotate relative to each other, the spring-damper will always act in the local joint0 coordinate system.

Additional information for ObjectConnectorLinearSpringDamper:

- This `Object` has/provides the following types = `Connector`
- Requested `Marker` type = `Position` + `Orientation`
- **Short name** for Python = `LinearSpringDamper`
- **Short name** for Python visualization object = `VLinearSpringDamper`

The item **ObjectConnectorLinearSpringDamper** with type = 'ConnectorLinearSpringDamper' has the following parameters:

Name	type	size	default value	description
name	String		''	connector's unique name
markerNumbers	ArrayMarkerIndex		[invalid [-1], invalid [-1]]	list of markers used in connector
stiffness	Real		0.	torsional stiffness [SI:Nm/rad] against relative rotation
damping	Real		0.	torsional damping [SI:Nm/(rad/s)]
axisMarker0	Vector3D		[1,0,0]	local axis of spring-damper in marker 0 coordinates; this axis will co-move with marker $m0$; if marker m0 is attached to ground, the spring-damper represents linear equations
offset	Real		0.	translational offset considered in the spring force calculation (this can be used as position control input!)
velocityOffset	Real		0.	velocity offset considered in the damper force calculation (this can be used as velocity control input!)
force	Real		0.	additional constant force [SI:Nm] added to spring-damper; this can be used to prescribe a force between the two attached bodies (e.g., for actuation and control)
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint

springForceUserFunction	PyFunctionMbsScalarIndexScalar5		0	A Python function which computes the scalar force between the two rigid body markers along axisMarker0 in $m0$ coordinates, if activeConnector=True; see description below
visualization	VObjectConnectorLinearSpringDamper			parameters for visualization of item

The item VObjectConnectorLinearSpringDamper has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = diameter of spring; size == -1.f means that default connector size is used
drawAsCylinder	Bool		False	if this flag is True, the spring-damper is represented as cylinder; this may fit better if the spring-damper represents an actuator
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

---

8.6.4.1 DESCRIPTION of ObjectConnectorLinearSpringDamper:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
markerNumbers	$[m0, m1]$	
stiffness	k	
damping	d	
axisMarker0	${}^{m0}\mathbf{d}$	
offset	x_{off}	
velocityOffset	v_{off}	
force	f_c	
springForceUserFunction	UF $\in \mathbb{R}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
DisplacementLocal	Δx	(scalar) relative displacement of the spring-damper
VelocityLocal	Δv	(scalar) relative velocity of spring-damper
ForceLocal	f_{SD}	(scalar) spring-damper force

8.6.4.2 Definition of quantities

input parameter	symbol	description
markerNumbers[0]	$m0$	global marker number m0
markerNumbers[1]	$m1$	global marker number m1

intermediate variables	symbol	description
marker m0 orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker m0
marker m0 position	${}^{0}\mathbf{p}_{m0}$	current position matrix provided by marker m0
marker m1 position	${}^{0}\mathbf{p}_{m1}$	current position matrix provided by marker m1
marker m0 velocity	${}^{0}\mathbf{v}_{m0}$	current global velocity vector provided by marker m0
marker m1 velocity	${}^{0}\mathbf{v}_{m1}$	current global velocity vector provided by marker m1
relative displacement	$\Delta x = ({}^{0,m0}\mathbf{A}\ {}^{m0}\mathbf{d})^{\mathrm{T}}({}^{0}\mathbf{p}_{m1} - {}^{0}\mathbf{p}_{m0})$	scalar relative displacement
relative velocity	$\Delta v = ({}^{0,m0}\mathbf{A}\ {}^{m0}\mathbf{d})^{\mathrm{T}}({}^{0}\mathbf{v}_{m1} - {}^{0}\mathbf{v}_{m0})$	scalar relative velocity; note that this only corresponds to the time derivative of Δx if the markers only move along the axis (in a prismatic joint)

8.6.4.3 Connector forces

If `activeConnector = True`, the vector spring force is computed as

$$f_{SD} = k\,(\Delta x - x_{\text{off}}) + d\,(\Delta v - v_{\text{off}}) + f_c \tag{8.265}$$

if `activeConnector = False`, f_{SD} is set zero.

If the springForceUserFunction UF is defined and `activeConnector = True`, f_{SD} instead becomes (t is current time)

$$f_{SD} = \text{UF}(mbs, t, i_N, \Delta x, \Delta v, \text{stiffness}, \text{damping}, \text{offset}) \tag{8.266}$$

and `iN` represents the itemNumber (=objectNumber).

Userfunction: `springForceUserFunction(mbs, t, itemNumber, displacement, velocity, stiffness, damping, offset)`

A user function, which computes the scalar torque depending on mbs, time, local quantities (relative displacement, relative velocity), which are evaluated at current time. Furthermore, the user function contains object parameters (stiffness, damping, offset). Note that itemNumber represents the index of the object in mbs, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

Detailed description of the arguments and local quantities:

---

arguments / return	type or size	description
mbs	MainSystem	provides MainSystem mbs in which underlying item is defined
t	Real	current time in mbs
itemNumber	Index	integer number i_N of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
displacement	Real	Δx
velocity	Real	Δv
stiffness	Real	copied from object
damping	Real	copied from object
offset	Real	copied from object
return value	Real	computed force

User function example:

```
#define simple cubic force for spring-damper:
def UFforce(mbs, t, itemNumber, displacement, velocity, stiffness, damping, offset):
    k = stiffness #passed as list
    return k*displacement + 0.1*k* displacement**3

#markerNumbers and parameters taken from mini example
mbs.AddObject(LinearSpringDamper(markerNumbers = [mGround, mBody],
                                 stiffness = k,
                                 damping = k*0.01,
                                 offset = 0,
                                 springForceUserFunction = UFforce))
```

8.6.4.4 MINI EXAMPLE for ObjectConnectorLinearSpringDamper

```
#example with rigid body at [0,0,0], with torsional load
k=2e3
nBody = mbs.AddNode(RigidRxyz())
oBody = mbs.AddObject(RigidBody(physicsMass=1, physicsInertia=[1,1,1,0,0,0],
                                nodeNumber=nBody))

mBody = mbs.AddMarker(MarkerNodeRigid(nodeNumber=nBody))
mGround = mbs.AddMarker(MarkerBodyRigid(bodyNumber=oGround,
                                        localPosition = [0,0,0]))
mbs.AddObject(PrismaticJointX(markerNumbers = [mGround, mBody])) #motion along ground X-axis
mbs.AddObject(LinearSpringDamper(markerNumbers = [mGround, mBody], axisMarker0=[1,0,0],
                                 stiffness = k, damping = k*0.01, offset = 0))
```

```
#force along x-axis; expect approx. Delta x = 1/k=0.0005
mbs.AddLoad(Force(markerNumber = mBody, loadVector=[1,0,0]))

#assemble and solve system for default parameters
mbs.Assemble()
mbs.SolveDynamic(exu.SimulationSettings())

#check result at default integration time
exudynTestGlobals.testResult = mbs.GetNodeOutput(nBody, exu.OutputVariableType.
Displacement)[0]
```

For examples on ObjectConnectorLinearSpringDamper see Relevant Examples and TestModels with weblink:

- chainDriveExample.py (Examples/)

8.6.5 ObjectConnectorTorsionalSpringDamper

An torsional spring-damper element acting on relative rotations around Z-axis of local joint0 coordinate system; connects to orientation-based markers; if other rotation axis than the local joint0 Z axis shall be used, the joint rotationMarker0 / rotationMarker1 may be used. The joint perfectly extends a RevoluteJoint with a spring-damper, which can also be used to represent feedback control in an elegant and efficient way, by chosing appropriate user functions. It also allows to measure continuous / infinite rotations by making use of a NodeGeneric which compensates $\pm\pi$ jumps in the measured rotation (OutputVariableType.Rotation).

Additional information for ObjectConnectorTorsionalSpringDamper:

- This `Object` has/provides the following types = `Connector`
- Requested `Marker` type = `Orientation`
- Requested `Node` type = `GenericData`
- **Short name** for Python = `TorsionalSpringDamper`
- **Short name** for Python visualization object = `VTorsionalSpringDamper`

The item **ObjectConnectorTorsionalSpringDamper** with type = 'ConnectorTorsionalSpringDamper' has the following parameters:

Name	type	size	default value	description
name	String		''	connector's unique name
markerNumbers	ArrayMarkerIndex		[invalid [-1], invalid [-1]]	list of markers used in connector
nodeNumber	NodeIndex		invalid (-1)	node number of a NodeGenericData with 1 dataCoordinate for continuous rotation reconstruction; if this node is left to invalid index, it will not be used
stiffness	Real		0.	torsional stiffness [SI:Nm/rad] against relative rotation
damping	Real		0.	torsional damping [SI:Nm/(rad/s)]
rotationMarker0	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	local rotation matrix for marker 0; transforms joint into marker coordinates
rotationMarker1	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	local rotation matrix for marker 1; transforms joint into marker coordinates
offset	Real		0.	rotational offset considered in the spring torque calculation (this can be used as rotation control input!)
velocityOffset	Real		0.	angular velocity offset considered in the damper torque calculation (this can be used as angular velocity control input!)

torque	Real		0.	additional constant torque [SI:Nm] added to spring-damper; this can be used to prescribe a torque between the two attached bodies (e.g., for actuation and control)
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
springTorqueUserFunction	PyFunctionMbsScalarIndexScalar5 0			A Python function which computes the scalar torque between the two rigid body markers in local joint0 coordinates, if activeConnector=True; see description below
visualization	VObjectConnectorTorsionalSpringDamper			parameters for visualization of item

The item VObjectConnectorTorsionalSpringDamper has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = diameter of spring; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

---

8.6.5.1 DESCRIPTION of ObjectConnectorTorsionalSpringDamper:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
nodeNumber	n_d	
stiffness	k	
damping	d	
offset	θ_{off}	
velocityOffset	ω_{off}	
torque	τ_c	
springTorqueUserFunction	$\text{UF} \in \mathbb{R}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**

Rotation	$\Delta\theta$	relative rotation around the spring-damper Z-coordinate, enhanced to a continuous rotation (infinite rotations > $+\pi$ and < $-\pi$) if a NodeGeneric with 1 coordinate as added
AngularVelocityLocal	$\Delta\omega$	scalar relative angular velocity around joint0 Z-axis
TorqueLocal	τ_{SD}	scalar spring-damper torque around the local joint0 Z-axis

8.6.5.2 Definition of quantities

input parameter	symbol	description
rotationMarker0	${}^{m0,J0}\mathbf{A}$	rotation matrix which transforms from joint 0 into marker 0 coordinates
rotationMarker1	${}^{m1,J1}\mathbf{A}$	rotation matrix which transforms from joint 1 into marker 1 coordinates
markerNumbers[0]	$m0$	global marker number m0
markerNumbers[1]	$m1$	global marker number m1
nodeNumber	$n0$	optional node number of a generic node (otherwise exu.InvalidIndex())

intermediate variables	symbol	description
marker m0 orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker m0
marker m1 orientation	${}^{0,m1}\mathbf{A}$	current rotation matrix provided by marker m1
marker m0 ang. velocity	${}^{m0}\boldsymbol{\omega}_{m0}$	current local angular velocity vector provided by marker m0
marker m1 ang. velocity	${}^{m1}\boldsymbol{\omega}_{m1}$	current local angular velocity vector provided by marker m1
AngularVelocityLocal	$\Delta\omega = \left({}^{J0,m1}\mathbf{A}\ {}^{m1}\boldsymbol{\omega} - {}^{J0,m0}\mathbf{A}\ {}^{m0}\boldsymbol{\omega}\right)_Z$	angular velocity around joint0 Z-axis

8.6.5.3 Connector forces

If `activeConnector = True`, the vector spring force is computed as

$$\tau_{SD} = k\,(\Delta\theta - \theta_{\text{off}}) + d\,(\Delta\omega - \omega_{\text{off}}) + \tau_c \tag{8.267}$$

if `activeConnector = False`, τ_{SD} is set zero.

If the springTorqueUserFunction UF is defined and `activeConnector = True`, τ_{SD} instead becomes (t is current time)

$$\tau_{SD} = \text{UF}(mbs, t, i_N, \Delta\theta, \Delta\omega, \text{stiffness}, \text{damping}, \text{offset}) \tag{8.268}$$

and `iN` represents the itemNumber (=objectNumber).

Userfunction: springTorqueUserFunction(mbs, t, itemNumber, rotation, angularVelocity, stiffness, damping, offset)

A user function, which computes the scalar torque depending on mbs, time, local quantities (relative rotation, relative angularVelocity), which are evaluated at current time. Furthermore, the user function contains object parameters (stiffness, damping, offset). Note that itemNumber represents the index of the object in mbs, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

Detailed description of the arguments and local quantities:

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs in which underlying item is defined
`t`	Real	current time in mbs
`itemNumber`	Index	integer number i_N of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
`rotation`	Real	$\Delta\theta$
`angularVelocity`	Real	$\Delta\omega$
`stiffness`	Real	copied from object
`damping`	Real	copied from object
`offset`	Real	copied from object
`return value`	Real	computed torque

User function example:

```
    #define simple cubic force for spring-damper:
    def UFforce(mbs, t, itemNumber, rotation, angularVelocity, stiffness, damping,
offset):
        k = stiffness #passed as list
        u = rotation
        return k*u + 0.1*k*u**3

    #markerNumbers and parameters taken from mini example
    mbs.AddObject(TorsionalSpringDamper(markerNumbers = [mGround, mBody],
                                        stiffness = k,
                                        damping = k*0.01,
                                        offset = 0,
                                        springTorqueUserFunction = UFforce))
```

8.6.5.4 MINI EXAMPLE for ObjectConnectorTorsionalSpringDamper

```
#example with rigid body at [0,0,0], with torsional load
k=2e3
nBody = mbs.AddNode(RigidRxyz())
oBody = mbs.AddObject(RigidBody(physicsMass=1, physicsInertia=[1,1,1,0,0,0],
                                nodeNumber=nBody))

mBody = mbs.AddMarker(MarkerNodeRigid(nodeNumber=nBody))
mGround = mbs.AddMarker(MarkerBodyRigid(bodyNumber=oGround,
                                        localPosition = [0,0,0]))
mbs.AddObject(RevoluteJointZ(markerNumbers = [mGround, mBody])) #rotation around ground
Z-axis
mbs.AddObject(TorsionalSpringDamper(markerNumbers = [mGround, mBody],
                                    stiffness = k, damping = k*0.01, offset = 0))

#torque around z-axis; expect approx. phiZ = 1/k=0.0005
mbs.AddLoad(Torque(markerNumber = mBody, loadVector=[0,0,1]))

#assemble and solve system for default parameters
mbs.Assemble()
mbs.SolveDynamic(exu.SimulationSettings())

#check result at default integration time
exudynTestGlobals.testResult = mbs.GetNodeOutput(nBody, exu.OutputVariableType.Rotation)
[2]
```

For examples on ObjectConnectorTorsionalSpringDamper see Relevant Examples and TestModels with weblink:

- chainDriveExample.py (Examples/)
- mobileMecanumWheelRobotWithLidar.py (Examples/)
- ROSMobileManipulator.py (Examples/)
- ROSTurtle.py (Examples/)
- serialRobotInteractiveLimits.py (Examples/)
- serialRobotTSD.py (Examples/)
- createFunctionsTest.py (TestModels/)
- rotatingTableTest.py (TestModels/)
- sliderCrank3Dbenchmark.py (TestModels/)

8.6.6 ObjectConnectorCoordinateSpringDamper

A 1D (scalar) spring-damper element acting on single ODE2 coordinates; connects to coordinate-based markers; NOTE that the coordinate markers only measure the coordinate (=displacement), but the reference position is not included as compared to position-based markers!; the spring-damper can also act on rotational coordinates.

Additional information for ObjectConnectorCoordinateSpringDamper:

- This `Object` has/provides the following types = `Connector`
- Requested `Marker` type = `Coordinate`
- **Short name** for Python = `CoordinateSpringDamper`
- **Short name** for Python visualization object = `VCoordinateSpringDamper`

The item **ObjectConnectorCoordinateSpringDamper** with type = 'ConnectorCoordinateSpring-Damper' has the following parameters:

Name	type	size	default value	description
name	String		''	connector's unique name
markerNumbers	ArrayMarkerIndex		[invalid [-1], invalid [-1]]	list of markers used in connector
stiffness	Real		0.	stiffness [SI:N/m] of spring; acts against relative value of coordinates
damping	Real		0.	damping [SI:N/(m s)] of damper; acts against relative velocity of coordinates
offset	Real		0.	offset between two coordinates (reference length of springs), see equation
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
springForceUserFunction	PyFunctionMbsScalarIndexScalar5		0	A Python function which defines the spring force with 8 parameters, see equations section / see description below
visualization	VObjectConnectorCoordinateSpringDamper			parameters for visualization of item

The item VObjectConnectorCoordinateSpringDamper has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = diameter of spring; size == -1.f means that default connector size is used

color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

---

8.6.6.1 DESCRIPTION of ObjectConnectorCoordinateSpringDamper:

Information on input parameters:

input parameter	symbol	description see tables above
stiffness	k	
damping	d	
offset	l_{off}	
springForceUserFunction	UF $\in \mathbb{R}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Displacement	Δq	relative scalar displacement of marker coordinates
Velocity	Δv	difference of scalar marker velocity coordinates
Force	f_{SD}	scalar force in connector

8.6.6.2 Definition of quantities

intermediate variables	symbol	description
marker m0 coordinate	q_{m0}	current displacement coordinate which is provided by marker m0; does NOT include reference coordinate!
marker m1 coordinate	q_{m1}	
marker m0 velocity coordinate	v_{m0}	current velocity coordinate which is provided by marker m0
marker m1 velocity coordinate	v_{m1}	

8.6.6.3 Connector forces

Displacement between marker m0 to marker m1 coordinates (does NOT include reference coordinates),

$$\Delta q = q_{m1} - q_{m0} \tag{8.269}$$

and relative velocity,

$$\Delta v = v_{m1} - v_{m0} \tag{8.270}$$

If `activeConnector = True`, the scalar spring force vector is computed as

$$f_{SD} = k\,(\Delta q - l_{\text{off}}) + d \cdot \Delta v \tag{8.271}$$

If the springForceUserFunction UF is defined, $\mathbf{f}_{SD}$ instead becomes (t is current time)

$$f_{SD} = \text{UF}(mbs, t, i_N, \Delta q, \Delta v, k, d, l_{\text{off}}) \tag{8.272}$$

and `iN` represents the itemNumber (=objectNumber).

If `activeConnector = False`, f_{SD} is set to zero.

NOTE that until 2023-01-21 (exudyn V1.5.76), the CoordinateSpringDamper included the parameters dryFriction and dryFrictionProportionalZone. These parameters have been removed and they are only available in CoordinateSpringDamperExt, HOWEVER, with different names. In order to use CoordinateSpringDamperExt instead of the old CoordinateSpringDamper with the same friction behavior, we recoomend:

- USE CoordinateSpringDamperExt.fDynamicFriction INSTEAD of CoordinateSpringDamper.dryFriction
- USE CoordinateSpringDamperExt.frictionProportionalZone INSTEAD of CoordinateSpringDamper.dryFrictionProporti
- CoordinateSpringDamperExt.frictionProportionalZone has a different behavior in case that it is zero; thus use 1e-16 in this case, to get as close as possible to previous behaviour
- the variables stiffness, damping and offset have the same interpretation in both objects
- keep every other friction, sticking and contact variables in CoordinateSpringDamperExt as default values
- user functions obtained a new interface in CoordinateSpringDamperExt, which just needs to be adapted

---

Userfunction: `springForceUserFunction(mbs, t, itemNumber, displacement, velocity, stiffness, damping, offset, dryFriction, dryFrictionProportionalZone)`

A user function, which computes the scalar spring force depending on time, object variables (displacement, velocity) and object parameters . The object variables are passed to the function using the current values of the CoordinateSpringDamper object. Note that itemNumber represents the index of the object in mbs, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs in which underlying item is defined
`t`	Real	current time in mbs
`itemNumber`	Index	integer number i_N of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
`displacement`	Real	Δq
`velocity`	Real	Δv
`stiffness`	Real	copied from object
`damping`	Real	copied from object
`offset`	Real	copied from object
`return value`	Real	scalar value of computed force

User function example:

```
#see also mini example! NOTE changes above since 2023-01-23
def UFforce(mbs, t, itemNumber, u, v, k, d, offset):
    return k*(u-offset) + d*v
```

8.6.6.4 MINI EXAMPLE for ObjectConnectorCoordinateSpringDamper

```
#define user function:
#NOTE: removed 2023-01-21: dryFriction, dryFrictionProportionalZone
def springForce(mbs, t, itemNumber, u, v, k, d, offset):
    return 0.1*k*u+k*u**3+v*d

nMass=mbs.AddNode(Point(referenceCoordinates = [2,0,0]))
massPoint = mbs.AddObject(MassPoint(physicsMass = 5, nodeNumber = nMass))

groundMarker=mbs.AddMarker(MarkerNodeCoordinate(nodeNumber= nGround, coordinate = 0))
nodeMarker  =mbs.AddMarker(MarkerNodeCoordinate(nodeNumber= nMass, coordinate = 0))

#Spring-Damper between two marker coordinates
mbs.AddObject(CoordinateSpringDamper(markerNumbers = [groundMarker, nodeMarker],
                                     stiffness = 5000, damping = 80,
                                     springForceUserFunction = springForce))
loadCoord = mbs.AddLoad(LoadCoordinate(markerNumber = nodeMarker, load = 1)) #static
linear solution:0.002

#assemble and solve system for default parameters
mbs.Assemble()
mbs.SolveDynamic()

#check result at default integration time
exudynTestGlobals.testResult = mbs.GetNodeOutput(nMass,
                                                 exu.OutputVariableType.Displacement)[0]
```

For examples on ObjectConnectorCoordinateSpringDamper see Relevant Examples and TestModels with weblink:

- slidercrankWithMassSpring.py (Examples/)
- ANCFALEtest.py (Examples/)
- ANCFswitchingSlidingJoint2D.py (Examples/)
- beltDriveALE.py (Examples/)
- beltDriveReevingSystem.py (Examples/)

- ComputeSensitivitiesExample.py (Examples/)
- coordinateSpringDamper.py (Examples/)
- finiteSegmentMethod.py (Examples/)
- geneticOptimizationSliderCrank.py (Examples/)
- lugreFrictionTest.py (Examples/)
- massSpringFrictionInteractive.py (Examples/)
- minimizeExample.py (Examples/)
- ...
- scissorPrismaticRevolute2D.py (TestModels/)
- ANCFbeltDrive.py (TestModels/)
- ANCFcontactFrictionTest.py (TestModels/)
- ...

8.6.7 ObjectConnectorCoordinateSpringDamperExt

A 1D (scalar) spring-damper element acting on single ODE2 coordinates; same as ObjectConnectorCoordinateSpringDamper but with extended features, such as limit stop and improved friction; has different user function interface and additional data node as compared to ObjectConnectorCoordinateSpringDamper, but otherwise behaves very similar. The CoordinateSpringDamperExt is very useful for a single axis of a robot or similar machine modelled with a KinematicTree, as it can add friction and limits based on physical properties. It is highly recommended, to use the bristle model for friction with frictionProportionalZone=0 in case of implicit integrators (GeneralizedAlpha) as it converges better.

Additional information for ObjectConnectorCoordinateSpringDamperExt:

- This `Object` has/provides the following types = `Connector`
- Requested `Marker` type = `Coordinate`
- Requested `Node` type = `GenericData`
- **Short name** for Python = `CoordinateSpringDamperExt`
- **Short name** for Python visualization object = `VCoordinateSpringDamperExt`

The item **ObjectConnectorCoordinateSpringDamperExt** with type = 'ConnectorCoordinateSpringDamperExt' has the following parameters:

Name	type	size	default value	description
name	String		''	connector's unique name
markerNumbers	ArrayMarkerIndex		[invalid [-1], invalid [-1]]	list of markers used in connector
nodeNumber	NodeIndex		invalid (-1)	node number of a NodeGenericData for 3 data coordinates (friction mode, last sticking position, limit stop state), see description for details; must exist in case of bristle friction model or limit stops
stiffness	Real		0.	stiffness [SI:N/m] of spring; acts against relative value of coordinates
damping	Real		0.	damping [SI:N/(m s)] of damper; acts against relative velocity of coordinates
offset	Real		0.	offset between two coordinates (reference length of springs), see equation; it can be used to represent the pre-scribed drive coordinate
velocityOffset	Real		0.	offset between two coordinates; used to model D-control of a drive, where damping is not acting against prescribed velocity

factor0	Real		1.	marker 0 coordinate is multiplied with factor0
factor1	Real		1.	marker 1 coordinate is multiplied with factor1
fDynamicFriction	UReal		0.	dynamic (viscous) friction force [SI:N] against relative velocity when sliding; assuming a normal force f_N, the friction force can be interpreted as $f_\mu = \mu f_N$
fStaticFrictionOffset	UReal		0.	static (dry) friction offset force [SI:N]; assuming a normal force f_N, the friction force is limited by $f_\mu \leq (\mu_{so} + \mu_d) f_N = f_{\mu_d} + f_{\mu_{so}}$
stickingStiffness	UReal		0.	stiffness of bristles in sticking case [SI:N/m]
stickingDamping	UReal		0.	damping of bristles in sticking case [SI:N/(m/s)]
exponentialDecayStatic	PReal		1.e-3	relative velocity for exponential decay of static friction offset force [SI:m/s] against relative velocity; at $\Delta v = v_{\text{exp}}$, the static friction offset force is reduced to 36.8%
fViscousFriction	Real		0.	viscous friction force part [SI:N/(m s)], acting against relative velocity in sliding case
frictionProportionalZone	UReal		0.	if non-zero, a regularized Stribeck model is used, regularizing friction force around zero velocity - leading to zero friction force in case of zero velocity; this does not require a data node at all; if zero, the bristle model is used, which requires a data node which contains previous friction state and last sticking position
limitStopsUpper	Real		0.	upper (maximum) value [SI:m] of coordinate before limit is activated; defined relative to the two marker coordinates
limitStopsLower	Real		0.	lower (minimum) value [SI:m] of coordinate before limit is activated; defined relative to the two marker coordinates
limitStopsStiffness	UReal		0.	stiffness [SI:N/m] of limit stop (contact stiffness); following a linear contact model
limitStopsDamping	UReal		0.	damping [SI:N/(m/s)] of limit stop (contact damping); following a linear contact model
useLimitStops	bool		False	if True, limit stops are considered and parameters must be set accordingly; furthermore, the NodeGenericData must have 3 data coordinates
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
springForceUserFunction	PyFunctionMbsScalarIndexScalar11		0	A Python function which defines the spring force with 8 parameters, see equations section / see description below
visualization	VObjectConnectorCoordinateSpringDamperExt			parameters for visualization of item

The item VObjectConnectorCoordinateSpringDamperExt has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = diameter of spring; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

---

8.6.7.1 DESCRIPTION of ObjectConnectorCoordinateSpringDamperExt:

Information on input parameters:

input parameter	symbol	description see tables above
stiffness	k	
damping	d	
offset	x_{off}	
velocityOffset	v_{off}	
factor0	f_0	
factor1	f_1	
fDynamicFriction	$f_{\mu,\text{d}}$	
fStaticFrictionOffset	$f_{\mu,\text{so}}$	
stickingStiffness	k_μ	
stickingDamping	d_μ	
exponentialDecayStatic	v_{exp}	
fViscousFriction	$f_{\mu,\text{v}}$	
frictionProportionalZone	v_{reg}	
limitStopsUpper	s_{upper}	
limitStopsLower	s_{lower}	
limitStopsStiffness	k_{limits}	
limitStopsDamping	d_{limits}	
springForceUserFunction	UF $\in \mathbb{R}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Displacement	Δq	relative scalar displacement of marker coordinates
Velocity	Δv	difference of scalar marker velocity coordinates
Force	f_{SD}	scalar spring force

8.6.7.2 Definition of quantities

intermediate variables	symbol	description
marker m0 coordinate	q_{m0}	current displacement coordinate which is provided by marker m0; does NOT include reference coordinate!
marker m1 coordinate	q_{m1}	
marker m0 velocity coordinate	v_{m0}	current velocity coordinate which is provided by marker m0
marker m1 velocity coordinate	v_{m1}	

8.6.7.3 Connector forces

Displacement between marker m0 to marker m1 coordinates (does NOT include reference coordinates),

$$q = f_1 \cdot q_{m1} - f_0 \cdot q_{m0} \tag{8.273}$$

and relative velocity,

$$v = f_1 \cdot v_{m1} - f_0 \cdot v_{m0} \tag{8.274}$$

The friction force is computed from given friction 'force' parameters, as there is no normal force in this model. This means, that `fDynamicFriction` represents $\mu_d \cdot F_N$ in which μ_d is the friction parameter and F_N is an according normal force.

The friction force is computed for different cases:

- CASE 1: `frictionProportionalZone != 0` ($v_{\text{reg}} \neq 0$):
  This case works well for explicit integrators and represents simplified friction. It is suited best, e.g., for drives if considered for a specific velocity, but not for the velocity=0 (at which no friction force is produced). If $f_{\mu,\text{d}} > 0$ or $f_{\mu,\text{so}} > 0$ or $f_{\mu,\text{v}}! = 0$, the Stribeck friction model is used, with

$$f_{\text{friction}} = \begin{cases} (f_{\mu,\text{d}} + f_{\mu,\text{so}})\frac{\Delta v}{v_{\text{reg}}}, & \text{if} \quad |v| <= v_{\text{reg}} \quad \text{and} \quad v_{\text{reg}} \neq 0 \\ \text{Sign}(v)\left(f_{\mu,\text{d}} + f_{\mu,\text{so}} e^{-(|v|-v_{reg})/v_{exp}} + f_{\mu,\text{v}}(|v| - v_{\text{reg}})\right), & \text{else} \end{cases} \tag{8.275}$$

  This case does not use a PostNewton iteration (which may be advantageous in constant step size explicit integration, but may be problematic in implicit integration).

- CASE 2: `frictionProportionalZone != 0` (or `useLimitStops=True`):
  This case is perfectly suited for implicit integration, as it includes special switching variables that help to avoid numerical problems due to switching (e.g., between stick and slip) during a Newton iteration. In this case, a so-called bristle model is used, which requires the nodeNumber (data node) to be defined by a GenericDataNode, which must contain 3 data variables. In case of sticking, the sticking force results from a spring-damper model with parameters k_{limits} and d_{limits}, which resolves sticking very well. The last sticking position is tracked, which allows to change between stick and slip; however, transition means a reduction of accuracy and requires additional computation of system Jacobians and Newton or discontinuous iterations. This case includes a PostNewton iteration to switch between stick and slip.

In CASE 2, the GenericDataNode has the 3 data variables (friction mode, last sticking position, limit stop state):

0: friction mode d_μ:
   $d_\mu = 0$: stick,
   $d_\mu = \pm f_{\text{slip}}$: slip (in according positive or negative direction); f_{slip} representing the slipping force
1: last sticking position x_{lsp}: contains relative coordinate q at last sticking position; in the sticking case, any deviation from that position leads to an additional bristle force

$$f^*_{\text{friction}} = (q - x_{lsp}) \cdot k_\mu + v \cdot d_\mu \tag{8.276}$$

2: limit stop state d_{ls}: $d_{ls} = 0$: no limit reached (no contact, $d_{ls} < 0$: limitStopsLower surpassed, $d_{ls} > 0$: limitStopsUpper surpassed; $|d_{ls}|$ contains the penetration value of the soft contact model

Initialization of the GenericDataNode should be done such that the initial state (e.g. stick) is already set within this variable. Not doing so may change results (as the solver assumes that the model is already slipping) and requires additional iterations. NOTE, that in particular, if d_μ is initilized with 0 (stick) and x_{lsp} (last sticking position) differs largely from the current q, a large initial force may result.

The contact force f_{contact} is computed if limit stops are reached. The contact is represented by a spring-damper, which is activated as soon as the limit is reached and deactivated, if the limit is left again. Contact forces are computed from stiffness k_{limits} and damping d_{limits}, penetration into stop and velocity,

$$f_{\text{contact}} = \begin{cases} k_{\text{limits}} \cdot (q - s_{\text{upper}}) + d_{\text{limits}} \cdot v & \text{if} \quad q > s_{\text{upper}} \\ k_{\text{limits}} \cdot (q - s_{\text{lower}}) + d_{\text{limits}} \cdot v & \text{if} \quad q < s_{\text{lower}} \end{cases} \tag{8.277}$$

NOTE: while a combination of friction and limit stop is possible, it may be wanted to put a friction with `frictionProportionalZone != 0` and a limit stop into two different objects, as the combined behavior would switch to a PostNewton method for the regularized friction model.

If `activeConnector = True`, the scalar spring force vector is computed as

$$f_{SD} = k \cdot (q - x_{\text{off}}) + d \cdot (v - v_{\text{off}}) + f_{\text{friction}} + f_{\text{contact}} \tag{8.278}$$

If the springForceUserFunction UF is defined, $\mathbf{f}_{SD}$ instead becomes (t is current time)

$$f_{SD} = \text{UF}(mbs, t, i_N, q, v, k, d, x_{\text{off}}, v_{\text{off}}, f_{\mu,\text{d}}, f_{\mu,\text{so}}, v_{\text{exp}}, f_{\mu,\text{v}}, v_{\text{reg}}) \tag{8.279}$$

and `iN` represents the itemNumber (=objectNumber).

The virtual work of the connector force is computed from the virtual displacement

$$\delta q = f_1 \cdot \delta q_{m1} - f_0 \cdot \delta q_{m0} \ , \tag{8.280}$$

and the virtual work results as

$$\delta W_{SD} = f_{SD} \cdot \delta q = f_{SD} \cdot (f_1 \cdot \delta q_{m1} - f_0 \cdot \delta q_{m0}) \ . \tag{8.281}$$

The generalized (elastic) forces thus read for the markers $m0$ and $m1$,

$$\mathbf{Q}_{SD,m0} = -f_{SD} \cdot f_0 \cdot \mathbf{J}_{coord,m0} \ , \quad \mathbf{Q}_{SD,m1} = f_{SD} \cdot f_1 \cdot \mathbf{J}_{coord,m1} \ , \tag{8.282}$$

in which $\mathbf{J}_{coord,m0}$ and $\mathbf{J}_{coord,m1}$ represent the coordinate Jacobians of the respective markers. As can be seen in generalized force $\mathbf{Q}$, the factors f_0 and f_1 are added accordingly which increase the force on 'slower' coordinates for certain gear ratios.

If `activeConnector = False`, f_{SD} is set to zero.

Userfunction: springForceUserFunction(mbs, t, itemNumber, displacement, velocity, stiffness, damping, offset, velocityOffset, fDynamicFriction, fStaticFrictionOffset, exponentialDecayStat fViscousFriction, frictionProportionalZone)

A user function, which computes the scalar spring force depending on time, object variables (displacement, velocity) and several object parameters. Note that itemNumber represents the index of the object in mbs, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

Only a subset of object variables is passed to the function using the current values of the CoordinateSpringDamperExt object. For parameters that are not passed via the user function interface, use mbs.GetObject(itemNumber) or, e.g., mbs.GetObjectParameter(itemNumber, 'limitStopsUpper') to obtain these parameters inside the user function.

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs in which underlying item is defined
`t`	Real	current time in mbs
`itemNumber`	Index	integer number i_N of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
`displacement`	Real	Δq
`velocity`	Real	Δv
`stiffness`	Real	copied from object
`damping`	Real	copied from object
`offset`	Real	copied from object
`velocityOffset`	Real	copied from object
`fDynamicFriction`	Real	copied from object
`fStaticFrictionOffset`	Real	copied from object
`exponentialDecayStatic`	Real	copied from object
`fViscousFriction`	Real	copied from object
`frictionProportionalZone`	Real	copied from object, also called regularization velocity or regVel
`return value`	Real	scalar value of computed force

User function example:

```
    #see also mini example!
    #For further parameters, use mbs.GetObject(itemNumber) or
    #  e.g. mbs.GetObjectParameter(itemNumber, 'limitStopsUpper')
    def UFforce(mbs, t, itemNumber, u, v, k, d, offset, vOffset, muDynamic,
myStaticOffset, muExpVel, muViscous, muRegVel):
```

```
        return k*(u-offset) + d*v
```

For examples on ObjectConnectorCoordinateSpringDamperExt see Relevant Examples and TestModels with weblink:

- coordinateSpringDamper.py (Examples/)
- lugreFrictionTest.py (Examples/)
- massSpringFrictionInteractive.py (Examples/)
- coordinateSpringDamperExt.py (TestModels/)

8.6.8 ObjectConnectorGravity

A connector for additing forces due to gravitational fields beween two bodies, which can be used for aerospace and small-scale astronomical problems; DO NOT USE this connector for adding gravitational forces (loads), which should be using LoadMassProportional, which is acting global and always in the same direction.

Additional information for ObjectConnectorGravity:

- This `Object` has/provides the following types = `Connector`
- Requested `Marker` type = `Position`
- **Short name** for Python = `ConnectorGravity`
- **Short name** for Python visualization object = `VConnectorGravity`

The item **ObjectConnectorGravity** with type = 'ConnectorGravity' has the following parameters:

Name	type	size	default value	description
name	String		"	connector's unique name
markerNumbers	ArrayMarkerIndex		[invalid [-1], invalid [-1]]	list of markers used in connector
gravitationalConstant	Real		6.67430e-11	gravitational constant [SI:$m^3kg^{-1}s^{-2}$)]; while not recommended, a negative constant gan represent a repulsive force
mass0	UReal		0.	mass [SI:kg] of object attached to marker $m0$
mass1	UReal		0.	mass [SI:kg] of object attached to marker $m1$
minDistanceRegularization	UReal		0.	distance [SI:m] at which a regularization is added in order to avoid singularities, if objects come close
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectConnectorGravity			parameters for visualization of item

The item VObjectConnectorGravity has the following parameters:

Name	type	size	default value	description
show	Bool		False	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = diameter of spring; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

8.6.8.1 DESCRIPTION of ObjectConnectorGravity:

Information on input parameters:

input parameter	symbol	description see tables above
markerNumbers	$[m0, m1]^\mathrm{T}$	
gravitationalConstant	G	
mass0	$mass_0$	
mass1	$mass_1$	
minDistanceRegularization	d_{min}	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Distance	L	distance between both points
Displacement	$\Delta^0\mathbf{p}$	relative displacement between both points
Force	$\mathbf{f}$	gravity force vector, pointing from marker $m0$ to marker $m1$

8.6.8.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker m0
marker m1 position	${}^0\mathbf{p}_{m1}$	
marker m0 velocity	${}^0\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
marker m1 velocity	${}^0\mathbf{v}_{m1}$	

output variables	symbol	formula
Displacement	$\Delta^0\mathbf{p}$	${}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0}$
Velocity	$\Delta^0\mathbf{v}$	${}^0\mathbf{v}_{m1} - {}^0\mathbf{v}_{m0}$
Distance	L	$\lvert\Delta^0\mathbf{p}\rvert$
Force	$\mathbf{f}$	see below

8.6.8.3 Connector forces

The unit vector in force direction reads (if $L = 0$, singularity can be avoided using regularization),

$$\mathbf{v}_f = \frac{1}{L}\Delta^0\mathbf{p} \tag{8.283}$$

If `activeConnector = True`, and $L >= d_{min}$ the gravitational force is computed as

$$f_G = -G\frac{mass_0 \cdot mass_1}{L^2} \tag{8.284}$$

If `activeConnector = True`, and $L < d_{min}$ the gravitational force is computed as

$$f_G = -G\frac{mass_0 \cdot mass_1}{L^2 + (L - d_{min})^2} \tag{8.285}$$

which results in a regularization for small distances, which is helpful if there are no restrictions in objects to keep apart. If $d_{min} = 0$ and $L = 0$, there a system error is raised.

The vector of the gravitational force applied at both markers, pointing from marker $m0$ to marker $m1$, finally reads

$$\mathbf{f} = f_G \mathbf{v}_f \tag{8.286}$$

The virtual work of the connector force is computed from the virtual displacement

$$\delta\Delta^0\mathbf{p} = \delta\,^0\mathbf{p}_{m1} - \delta\,^0\mathbf{p}_{m0}\,, \tag{8.287}$$

and the virtual work (not the transposed version here, because the resulting generalized forces shall be a column vector,

$$\delta W_G = \mathbf{f}\delta\Delta^0\mathbf{p} = -\left(-G\frac{mass_0 \cdot mass_1}{L^2}\right)\left(\delta\,^0\mathbf{p}_{m1} - \delta\,^0\mathbf{p}_{m0}\right)^{\mathrm{T}}\mathbf{v}_f\,. \tag{8.288}$$

The generalized (elastic) forces thus result from

$$\mathbf{Q}_G = \frac{\partial\,^0\mathbf{p}}{\partial\mathbf{q}_G^{\mathrm{T}}}\mathbf{f}\,, \tag{8.289}$$

and read for the markers $m0$ and $m1$,

$$\mathbf{Q}_{G,m0} = -\left(-G\frac{mass_0 \cdot mass_1}{L^2}\right)\mathbf{J}_{pos,m0}^{\mathrm{T}}\mathbf{v}_f, \quad \mathbf{Q}_{G,m1} = \left(-G\frac{mass_0 \cdot mass_1}{L^2}\right)\mathbf{J}_{pos,m1}^{\mathrm{T}}\mathbf{v}_f\,, \tag{8.290}$$

where $\mathbf{J}_{pos,m1}$ represents the derivative of marker $m1$ w.r.t. its associated coordinates $\mathbf{q}_{m1}$, analogously $\mathbf{J}_{pos,m0}$.

8.6.8.4 MINI EXAMPLE for ObjectConnectorGravity

```
mass0 = 1e25
mass1 = 1e3
r = 1e5
G = 6.6743e-11
vInit = np.sqrt(G*mass0/r)
tEnd = (r*0.5*np.pi)/vInit #quarter period
node0 = mbs.AddNode(NodePoint(referenceCoordinates = [0,0,0])) #star
node1 = mbs.AddNode(NodePoint(referenceCoordinates = [r,0,0],
                              initialVelocities=[0,vInit,0])) #satellite
oMassPoint0 = mbs.AddObject(MassPoint(nodeNumber = node0, physicsMass=mass0))
oMassPoint1 = mbs.AddObject(MassPoint(nodeNumber = node1, physicsMass=mass1))

m0 = mbs.AddMarker(MarkerNodePosition(nodeNumber=node0))
m1 = mbs.AddMarker(MarkerNodePosition(nodeNumber=node1))
```

```
mbs.AddObject(ObjectConnectorGravity(markerNumbers=[m0,m1],
                                     mass0 = mass0, mass1=mass1))

#assemble and solve system for default parameters
mbs.Assemble()
sims = exu.SimulationSettings()
sims.timeIntegration.endTime = tEnd
mbs.SolveDynamic(sims, solverType=exu.DynamicSolverType.RK67)

#check result at default integration time
#expect y=x after one period of orbiting (got: 100000.00000000479)
exudynTestGlobals.testResult = mbs.GetNodeOutput(node1, exu.OutputVariableType.Position)
[1]/100000
```

For examples on ObjectConnectorGravity see Relevant Examples and TestModels with weblink:

- connectorGravityTest.py (TestModels/)

8.6.9 ObjectConnectorHydraulicActuatorSimple

A basic hydraulic actuator with pressure build up equations. The actuator follows a valve input value, which results in a in- or outflow of fluid depending on the pressure difference. Valve values can be prescribed by user functions (not yet available) or with the MainSystem PreStepUserFunction(...).

Additional information for ObjectConnectorHydraulicActuatorSimple:

- This `Object` has/provides the following types = `Connector`
- Requested `Marker` type = `Position`
- Requested `Node` type: read detailed information of item
- **Short name** for Python = `HydraulicActuatorSimple`
- **Short name** for Python visualization object = `VHydraulicActuatorSimple`

The item **ObjectConnectorHydraulicActuatorSimple** with type = 'ConnectorHydraulicActuatorSimple' has the following parameters:

Name	type	size	default value	description
name	String		"	connector's unique name
markerNumbers	ArrayMarkerIndex		[invalid [-1], invalid [-1]]	list of markers used in connector
nodeNumbers	ArrayNodeIndex		[]	currently a list with one node number of NodeGenericODE1 for 2 hydraulic pressures (reference values for this node must be zero); data node may be added in future for switching
offsetLength	UReal		0.	offset length [SI:m] of cylinder, representing minimal distance between the two bushings at stroke=0
strokeLength	PReal		0.	stroke length [SI:m] of cylinder, representing maximum extension relative to L_o; the measured distance between the markers is $L_s + L_o$
chamberCrossSection0	PReal		0.	cross section [SI:m^2] of chamber (inner cylinder) at piston head (nut) side (0)
chamberCrossSection1	PReal		0.	cross section [SI:m^2] of chamber at piston rod side (1); usually smaller than chamberCrossSection0
hoseVolume0	PReal		0.	hose volume [SI:m^3] at piston head (nut) side (0); as the effective bulk modulus would go to infinity at stroke length zero, the hose volume must be greater than zero
hoseVolume1	PReal		0.	hose volume [SI:m^3] at piston rod side (1); as the effective bulk modulus would go to infinity at max. stroke length, the hose volume must be greater than zero

valveOpening0	Real		0.	relative opening of valve [−1 . . . 1] [SI:1] at piston head (nut) side (0); positive value is valve opening towards system pressure, negative value is valve opening towards tank pressure; zero means closed valve
valveOpening1	Real		0.	relative opening of valve [−1 . . . 1] [SI:1] at piston rod side (1); positive value is valve opening towards system pressure, negative value is valve opening towards tank pressure; zero means closed valve
actuatorDamping	UReal		0.	damping [SI:N/(m s)] of hydraulic actuator (against actuator axial velocity)
oilBulkModulus	PReal		0.	bulk modulus of oil [SI:N/(m^2)]
cylinderBulkModulus	UReal		0.	bulk modulus of cylinder [SI:N/(m^2)]; in fact, this is value represents the effect of the cylinder stiffness on the effective bulk modulus
hoseBulkModulus	UReal		0.	bulk modulus of hose [SI:N/(m^2)]; in fact, this is value represents the effect of the hose stiffness on the effective bulk modulus
nominalFlow	PReal		0.	nominal flow of oil through valve [SI:m^3/s]
systemPressure	Real		0.	system pressure [SI:N/(m^2)]
tankPressure	Real		0.	tank pressure [SI:N/(m^2)]
useChamberVolumeChange	Bool		False	if True, the pressure build up equations include the change of oil stiffness due to change of chamber volume
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectConnectorHydraulicActuatorSimple			parameters for visualization of item

The item VObjectConnectorHydraulicActuatorSimple has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
cylinderRadius	float		0.05	radius for drawing of cylinder
rodRadius	float		0.03	radius for drawing of rod
pistonRadius	float		0.04	radius for drawing of piston (if drawn transparent)
pistonLength	float		0.001	radius for drawing of piston (if drawn transparent)
rodMountRadius	float		0.0	radius for drawing of rod mount sphere
baseMountRadius	float		0.0	radius for drawing of base mount sphere
baseMountLength	float		0.0	radius for drawing of base mount sphere

colorCylinder	Float4		[-1.,-1.,-1.,-1.]	RGBA cylinder color; if R==-1, use default connector color
colorPiston	Float4		[0.8,0.8,0.8,1.]	RGBA piston color

8.6.9.1 DESCRIPTION of ObjectConnectorHydraulicActuatorSimple:

Information on input parameters:

input parameter	symbol	description see tables above
markerNumbers	$[m0, m1]^\mathrm{T}$	
nodeNumbers	$\mathbf{n}_n = [n_{ODE1}]^\mathrm{T}$	
offsetLength	L_o	
strokeLength	L_s	
chamberCrossSection0	A_0	
chamberCrossSection1	A_1	
hoseVolume0	$V_{h,0}$	
hoseVolume1	$V_{h,1}$	
valveOpening0	$A_{v,0}$	
valveOpening1	$A_{v,1}$	
actuatorDamping	d_{HA}	
oilBulkModulus	K_{oil}	
cylinderBulkModulus	K_{cyl}	
hoseBulkModulus	K_{hose}	
nominalFlow	Q_n	
systemPressure	p_s	
tankPressure	p_t	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Distance	$L = \lvert\Delta^0\mathbf{p}\rvert$	distance between both marker points (usually the actuator bushings); current actuator length
Displacement		relative displacement between both marker points
Velocity	$\Delta^0\mathbf{v}$	relative velocity between both points
VelocityLocal	$\dot{L}$	actuator velocity, the derivative of actuator length
Force		force in actuator resulting as the difference of both pressures times according cross sections

8.6.9.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker m0
marker m1 position	${}^0\mathbf{p}_{m1}$	
marker m0 velocity	${}^0\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
marker m1 velocity	${}^0\mathbf{v}_{m1}$	
Displacement	$\Delta^0\mathbf{p} = {}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0}$	The relative vector between marker points, stored as Displacement in output variables
current actuator length	$L = \lvert\Delta^0\mathbf{p}\rvert$	stored as Distance in output variables
time derivative of actuator length	$\dot{L} = \Delta^0\mathbf{v}^{\mathrm{T}}\mathbf{v}_f$	
Velocity	$\Delta^0\mathbf{v} = {}^0\mathbf{v}_{m1} - {}^0\mathbf{v}_{m0}$	The vectorial relative velocity
Force	$\mathbf{f}$	see below

8.6.9.3 Connector forces

The unit vector in force direction reads (raises SysError if $L = 0$),

$$\mathbf{v}_f = \frac{1}{L}\Delta^0\mathbf{p} \tag{8.291}$$

The simple double-acting hydraulic actuator has two pressure chambers, one being denoted with 0 at the piston head (nut) and the other at the piston rod side denoted with 1. The pressure p_0 acts at the piston head at area A_0, while the pressure p_1 counteracts on the opposite side with (usually smaller) area A_1. If `activeConnector = True`, the scalar actuator force (tension = positive) is computed as

$$f_{HA} = -p_0 \cdot A_0 + p_1 \cdot A_1 + v \cdot d_H A \tag{8.292}$$

where v represents the actuator velocitiy and $d_H A$ is the viscous damping coefficient.

The vector of the actuator force applied at both markers finally reads

$$\mathbf{f} = f_{HA}\mathbf{v}_f \tag{8.293}$$

The virtual work of the connector force is computed from the virtual displacement

$$\delta\Delta^0\mathbf{p} = \delta\,{}^0\mathbf{p}_{m1} - \delta\,{}^0\mathbf{p}_{m0}\ , \tag{8.294}$$

and the virtual work (not the transposed version here, because the resulting generalized forces shall be a column vector),

$$\delta W_{HA} = \mathbf{f}\delta\Delta^0\mathbf{p}\ . \tag{8.295}$$

8.6.9.4 Pressure build up equations

The hydraulics model consists of a double-acting piston. It follows the paper of [50] except for the friction and the additional valve, which are not available here.

The hydraulic actuator contains internal states, namely pressures p_0 and p_1. The ODE1 for pressures follows for the the case of laminar flow, based on system and tank pressure, valve positions as well as the actuator velocity and position (only for change of volume).

The distance between the two marker points, which are usually the bushings or clevis mounts of the hydraulic cylinder, is denoted as L. The stroke length $s \in [0, L_s]$ is defined as

$$s = L - L_o \tag{8.296}$$

such that at zero stroke, the actuator length is L_o. The stroke velocity (positive value means extension) reads

$$\dot{s} = {}^{0}\Delta^{\mathrm{T}} \mathbf{v}_f \tag{8.297}$$

If `useChamberVolumeChange == True`, the volume change due to stroke change will be considered for the volume related to the stiffness of the fluid. The cylinder volumes in chambers 0 and 1 are then

$$V_{0,cur} = V_{h,0} + A_0 \cdot s, \quad V_{1,cur} = V_{h,1} + A_1 \cdot (L_s - s) \tag{8.298}$$

The effective bulk modulus for chamber $k \in 0, 1$ is computed as follows,

$$K_{k,eff} = \frac{1}{\frac{1}{K_{oil}} + \frac{V_{k,cur} - V_{h,k}}{V_{k,cur} \cdot K_{cyl}} + \frac{V_{h,k}}{V_{k,cur} \cdot K_{hose}}}, \tag{8.299}$$

where we use a slightly different approach from [50] when computing the volume for the cylinder bulk modulus term for $k = 1$.

Note that in case of $K_{cyl} = 0$ and/or $K_{hose} = 0$, the according fractions in Eq. (8.299) are set to zero (which other wise would give infinity).

Otherwise, if `useChamberVolumeChange == False`, $V_{0,cur} = V_{h,0}$, $V_{1,cur} = V_{h,1}$ and $K_{k,eff} = K_{oil}$ for chambers $k \in 0, 1$.

The pressure equations (explicit ODE1) have the structure

$$\begin{bmatrix} \dot{p}_0 \\ \dot{p}_1 \end{bmatrix} = \begin{bmatrix} f_0(p_0, s, \dot{s}) \\ f_1(p_1, s, \dot{s}) \end{bmatrix} \tag{8.300}$$

and follow for different cases and chambers / valves $k = \{0, 1\}$, based on the simple model where

- $A_{v,k} = 0$: valve k closed
- $A_{v,k} > 0$: valve k opened towards system pressure (pump)
- $A_{v,k} < 0$: valve k opened towards tank pressure

Thus, the following equations are used[5]:

$$\dot{p}_0 = \frac{K_{0,eff}}{V_{0,cur}} \left(-A_0 \cdot \dot{s} + A_{v,0} \cdot Q_n \cdot \mathrm{sqrts}(p_s - p_0)\right) \quad \text{if} \quad \mathrm{A}_{v,0} \geq 0 \tag{8.301}$$

$$\dot{p}_0 = \frac{K_{0,eff}}{V_{0,cur}} \left(-A_0 \cdot \dot{s} + A_{v,0} \cdot Q_n \cdot \mathrm{sqrts}(p_0 - p_t)\right) \quad \text{if} \quad \mathrm{A}_{v,0} < 0 \tag{8.302}$$

$$\dot{p}_1 = \frac{K_{1,eff}}{V_{1,cur}} \left(A_1 \cdot \dot{s} + A_{v,1} \cdot Q_n \cdot \mathrm{sqrts}(p_s - p_1)\right) \quad \text{if} \quad \mathrm{A}_{v,1} \geq 0 \tag{8.303}$$

[5] while it would happen rarely in regular operation, the arguments of the square roots could become negative; thus, in the implementation we use $\mathrm{sqrts}(x) = \mathrm{sign}(x) \cdot \sqrt{\mathrm{abs}(x)}$.

$$\dot{p}_1 = \frac{K_{1,eff}}{V_{1,cur}} \left(A_1 \cdot \dot{s} + A_{v,1} \cdot Q_n \cdot \text{sqrts}(p_1 - p_t) \right) \quad \text{if} \quad \text{A}_{v,1} < 0 \tag{8.304}$$

For examples on ObjectConnectorHydraulicActuatorSimple see Relevant Examples and TestModels with weblink:

- `HydraulicActuator2Arms.py` (Examples/)
- `HydraulicActuatorStaticInitialization.py` (Examples/)
- `hydraulicActuatorSimpleTest.py` (TestModels/)

8.6.10 ObjectConnectorReevingSystemSprings

A rD reeving system defined by a list of torque-free and friction-free sheaves or points that are connected with one rope (modelled as massless spring). NOTE that the spring can undergo tension AND compression (in order to avoid compression, use a PreStepUserFunction to turn off stiffness and damping in this case!). The force is assumed to be constant all over the rope. The sheaves or connection points are defined by nr rigid body markers $[m_0, m_1, \ldots, m_{nr-1}]$. At both ends of the rope there may be a prescribed motion coupled to a coordinate marker each, given by m_{c0} and m_{c1} .

Additional information for ObjectConnectorReevingSystemSprings:

- This `Object` has/provides the following types = `Connector`
- Requested `Marker` type = `_None`
- **Short name** for Python = `ReevingSystemSprings`
- **Short name** for Python visualization object = `VReevingSystemSprings`

The item **ObjectConnectorReevingSystemSprings** with type = 'ConnectorReevingSystemSprings' has the following parameters:

Name	type	size	default value	description
name	String		''	connector's unique name
markerNumbers	ArrayMarkerIndex		[invalid [-1], invalid [-1]]	list of position or rigid body markers used in reeving system and optional two coordinate markers (m_{c0}, m_{c1}); the first marker m_0 and the last rigid body marker m_{nr-1} represent the ends of the rope and are directly connected to a position; the markers $m_1, \ldots, m_{nr-2}$ can be connected to sheaves, for which a radius and an axis can be prescribed. The coordinate markers are optional and represent prescribed length at the rope ends (marker m_{c0} is added length at start, marker m_{c1} is added length at end of the rope in the reeving system)
hasCoordinateMarkers	Bool		False	flag, which determines, the list of markers (markerNumbers) contains two coordinate markers at the end of the list, representing the prescribed change of length at both ends
coordinateFactors	Vector2D		[1,1]	factors which are multiplied with the values of coordinate markers; this can be used, e.g., to change directions or to transform rotations (revolutions of a sheave) into change of length

stiffnessPerLength	UReal		0.	stiffness per length [SI:N/m/m] of rope; in case of cross section A and Young's modulus E, this parameter results in $E \cdot A$; the effective stiffness of the reeving system is computed as EA/L in which L is the current length of the rope
dampingPerLength	UReal		0.	axial damping per length [SI:N/(m/s)/m] of rope; the effective damping coefficient of the reeving system is computed as DA/L in which L is the current length of the rope
dampingTorsional	UReal		0.	torsional damping [SI:Nms] between sheaves; this effect can damp rotations around the rope axis, pairwise between sheaves; this parameter is experimental
dampingShear	UReal		0.	damping of shear motion [SI:Ns] between sheaves; this effect can damp motion perpendicular to the rope between each pair of sheaves; this parameter is experimental
regularizationForce	Real		0.1	small regularization force [SI:N] in order to avoid large compressive forces; this regularization force can either be < 0 (using a linear tension/compression spring model) or > 0, which restricts forces in the rope to be always $\geq -F_{reg}$. Note that smaller forces lead to problems in implicit integrators and smaller time steps. For explicit integrators, this force can be chosen close to zero.
referenceLength	Real		0.	reference length for computation of roped force
sheavesAxes	Vector3DList		[]	list of local vectors axes of sheaves; vectors refer to rigid body markers given in list of markerNumbers; first and last axes are ignored, as they represent the attachment of the rope ends
sheavesRadii	Vector		[]	radius for each sheave, related to list of markerNumbers and list of sheaveAxes; first and last radii must always be zero.
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectConnectorReevingSystemSprings			parameters for visualization of item

The item VObjectConnectorReevingSystemSprings has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

ropeRadius	float		0.001	radius of rope
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

---

8.6.10.1 DESCRIPTION of ObjectConnectorReevingSystemSprings:

Information on input parameters:

input parameter	symbol	description see tables above
markerNumbers	$[m_0, m_1, \ldots, m_{nr-1}, m_{c0}, m_{c1}]^T$	
coordinateFactors	$[f_0, f_1]^T$	
stiffnessPerLength	EA	
dampingPerLength	DA	
dampingTorsional	DT	
dampingShear	DS	
regularizationForce	F_{reg}	
referenceLength	L_{ref}	
sheavesAxes	$\mathbf{l}_a = [{}^{m0}\mathbf{a}_0, {}^{m1}\mathbf{a}_1, \ldots] in [\mathbb{R}^3, ...]$	
sheavesRadii	$\mathbf{l}_r = [r_0, r_1, \ldots]^T \in \mathbb{R}^n$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Distance	L	current total length of rope
VelocityLocal	$\dot{L}$	scalar time derivative of current total length of rope
ForceLocal	F	scalar force in reeving system (constant over length of rope)

8.6.10.2 General model assumptions

The `ConnectorReevingSystemSprings` model is based on a linear elastic, visco-elastic, and mass-less spring which is tangent to a list of rolls. The contact with the rolls is friction-less, causing no torque w.r.t. the rolling axis of the sheave. The force in the rope results from the difference of the total length L compared to the reference length or the rope, which may be changed by adding or subtracting rope length at the end points. All geometric operations are performed in 3D, allowing to model simple reeving systems in 3D.

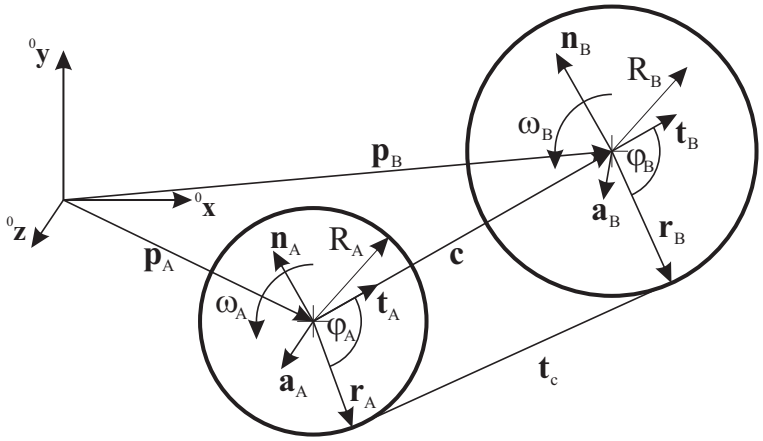


Figure 8.6: Geometry of common tangent for two spatial circles defined by radii R_A and R_B as well as by the normalized axis vectors $\mathbf{a}_A$ and $\mathbf{a}_B$. The tangent is undefined, if one of the axis vectors is parallel to the vector $\mathbf{c}$, which connects the two center points. The positive rotation sense is indicated by means of the angular velocities ω_A and ω_B.

8.6.10.3 Common tangent of two circles in 3D

In order to compute the total length of the rope of the reeving system, the tangent of two arbitrary circles in space needs to be computed. Considering Fig. 8.6, the relations are based on the center points of the circles $\mathbf{p}_A$ and $\mathbf{p}_B$, the radii R_A and R_B as well as the axis vectors $\mathbf{a}_A$ and $\mathbf{a}_B$, the latter vectors also defining the side at which the tangent contacts. For the definition of the tangent, the vectors $\mathbf{r}_A$ and $\mathbf{r}_B$ need to be computed.

For the special case of $R_A = R_B = 0$, it follows that $\mathbf{r}_A = \mathbf{p}_A$ and $\mathbf{r}_B = \mathbf{p}_B$. Otherwise, we first compute the vector between circle centers,

$$\mathbf{c} = \mathbf{p}_B - \mathbf{p}_A, \quad \text{and} \quad \mathbf{c}_0 = \frac{\mathbf{c}}{|\mathbf{c}|} , \tag{8.305}$$

and obtain the tangent vectors

$$\mathbf{t}_A = \mathbf{t}_B = \mathbf{c}_0 , \tag{8.306}$$

as well as the normal vectors

$$\mathbf{n}_A = \mathbf{a}_A \times \mathbf{c}_0, \quad \text{and} \quad \mathbf{n}_B = \mathbf{a}_B \times \mathbf{c}_0 . \tag{8.307}$$

Note that the orientation of the axis vectors $\mathbf{a}_A$ and $\mathbf{a}_B$ defines the orientation of the normals. By definition, we assume the following conditions,

$$\mathbf{n}_A^{\mathrm{T}} \mathbf{r}_A < 0, \quad \text{and} \quad \mathbf{n}_B^{\mathrm{T}} \mathbf{r}_B < 0 . \tag{8.308}$$

For two circles with equal radius and axes orientations, the angles result in $\varphi_A = \varphi_B = \pi$. In general, the unknown vectors $\mathbf{r}_A$ and $\mathbf{r}_B$ are computed by means of Newton's method. The unknown tangent vector is given as

$$\mathbf{t}_c = \mathbf{p}_B + \mathbf{r}_B - \mathbf{p}_A - \mathbf{r}_A = \mathbf{c} + \mathbf{r}_B - \mathbf{r}_A . \tag{8.309}$$

We now parameterize the two unknown vectors by means of unknown angles φ_A and φ_B,

$$\mathbf{r}_A = -R_A\left(\cos(\varphi_A)\mathbf{t}_A - \sin(\varphi_A)\mathbf{n}_A\right), \quad \text{and} \quad \mathbf{r}_B = -R_B\left(\cos(\varphi_B)\mathbf{t}_B - \sin(\varphi_B)\mathbf{n}_B\right) . \tag{8.310}$$

As vectors $\mathbf{r}_A$ and $\mathbf{r}_B$ must be perpendicular to $\mathbf{t}_c$, it follows that

$$\mathbf{r}_A^{\mathrm{T}}(\mathbf{c} + \mathbf{r}_B - \mathbf{r}_A) = 0, \quad \text{and} \quad \mathbf{r}_B^{\mathrm{T}}(\mathbf{c} + \mathbf{r}_B - \mathbf{r}_A) = 0, \tag{8.311}$$

or

$$\mathbf{r}_A^{\mathrm{T}}\mathbf{c} + \mathbf{r}_A^{\mathrm{T}}\mathbf{r}_B - R_A^2 = 0, \quad \text{and} \quad \mathbf{r}_B^{\mathrm{T}}\mathbf{c} - \mathbf{r}_B^{\mathrm{T}}\mathbf{r}_A + R_B^2 = 0 . \tag{8.312}$$

The relations Eq. (8.312) reduce to only one equation, if either $R_A = 0$ or $R_B = 0$. The equations can be solved by Newton's method by computing the jacobian of $\mathbf{J}_{CT}$ of Eq. (8.312) w.r.t. the unknown angles φ_A and φ_B. The iterations are started with

$$\varphi_A = \pi \quad \text{and} \quad \varphi_B = \pi, \tag{8.313}$$

and iterate until the error is below a certain tolerance, for details see the implementation in `Geometry.h`.

8.6.10.4 Connector forces

The current rope length results from the configuration of sheaves, including start and end position:

$$L = d_{m_0-m_1} + C_{m_1} + d_{m_1-m_2} + C_{m_2} + \ldots + d_{m_{nr-2}-m_{nr-1}} \tag{8.314}$$

in which $d_{\ldots}$ represents the free spans between two sheaves as computed from the common tangent in the previous section, and $C_{\ldots}$ represents the length along the circumference of the according marker if the according radius r is non-zero. The quantity $C_{\ldots}$ can be computed easily as soon as the radius vectors to the tangents $\mathbf{r}_A$ and $\mathbf{r}_B$ are known. Within a series of tangents, the previous to the current tangent will always enclose an angle between 0 and $2 \cdot \pi$.

In case that `hasCoordinateMarkers=True`, the total reference length and its derivative result as

$$L_0 = L_{ref} + f_0 \cdot q_{m_{c0}} + f_1 \cdot q_{m_{c1}}, \quad \dot{L}_0 = f_0 \cdot \dot{q}_{m_{c0}} + f_1 \cdot \dot{q}_{m_{c1}}, \tag{8.315}$$

while we set $L_0 = L_{ref}$ and $\dot{L}_0 = 0$ otherwise. The linear force in the reeving system (assumed to be constant all over the rope) is computed as

$$F_{lin} = (L - L_0)\frac{EA}{L_0} + (\dot{L} - \dot{L}_0)\frac{DA}{L_0} \tag{8.316}$$

The rope force is computed from

$$F = \begin{cases} F_{lin} & \text{if} \quad F_{lin} > 0 \\ F_{reg} \cdot \tanh(F_{lin}/F_{reg}) & \text{else} \end{cases} \tag{8.317}$$

Which allows small compressive forces F_{reg}. In case that $F_{reg} < 0$, compressive forces are not regularized (linear spring). The case $F_{reg} = 0$ will be used in future only in combination with a data node, which allows switching similar as in friction and contact elements.

Note that in case of $L_0 = 0$, the term $\frac{1}{L_0}$ is replaced by 1000. However, this case must be avoided by the user by choosing appropriate parameters for the system.

Additional damping may be added via the parameters *DT* and *DS*, which have to be treated carefully. The shearing parameter may be helpful to damp undesired oscillatory shearing motion, however, it may also damp rigid body motion of the overall mechanism.

Further details are given in the implementation and examples are provided in the `Examples` and `TestModels` folders.

For examples on ObjectConnectorReevingSystemSprings see Relevant Examples and TestModels with weblink:

- `craneReevingSystem.py` (Examples/)
- `reevingSystemSpringsTest.py` (TestModels/)

8.6.11 ObjectConnectorRollingDiscPenalty

A (flexible) connector representing a rolling rigid disc (marker 1) on a flat surface (marker 0, ground body, not moving) in global x-y plane. The connector is based on a penalty formulation and adds friction and slipping. The contraints works for discs as long as the disc axis and the plane normal vector are not parallel. Parameters may need to be adjusted for better convergence (e.g., dryFrictionProportionalZone). The formulation for the arbitrary disc axis is still under development and needs further testing. Note that the rolling body must have the reference point at the center of the disc.

Additional information for ObjectConnectorRollingDiscPenalty:

- This `Object` has/provides the following types = `Connector`
- Requested `Marker` type = `Position` + `Orientation`
- Requested `Node` type = `GenericData`
- **Short name** for Python = `RollingDiscPenalty`
- **Short name** for Python visualization object = `VRollingDiscPenalty`

The item **ObjectConnectorRollingDiscPenalty** with type = 'ConnectorRollingDiscPenalty' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex	2	[invalid [-1], invalid [-1]]	list of markers used in connector; $m0$ represents a point at the plane surface (normal of surface plane defined by planeNormal); the ground can also be a moving rigid body; $m1$ represents the rolling body, which has its reference point (=local position [0,0,0]) at the disc center point
nodeNumber	NodeIndex		invalid (-1)	node number of a NodeGenericData (size=3) for 3 dataCoordinates, needed for discontinuous iteration (friction and contact)
discRadius	PReal		0.	defines the disc radius
discAxis	Vector3D		[1,0,0]	axis of disc defined in marker $m1$ frame
planeNormal	Vector3D		[0,0,1]	normal to the contact / rolling plane (ground); note that the plane reference point can be arbitrarily chosen by the location of the marker $m0$
dryFrictionAngle	Real		0.	angle [SI:1 (rad)] which defines a rotation of the local tangential coordinates dry friction; this allows to model Mecanum wheels with specified roll angle

contactStiffness	UReal		0.	normal contact stiffness [SI:N/m]
contactDamping	UReal		0.	normal contact damping [SI:N/(m s)]
dryFriction	Vector2D		[0,0]	dry friction coefficients [SI:1] in local marker 1 joint $J1$ coordinates; if $\alpha_t == 0$, lateral direction $l = x$ and forward direction $f = y$; assuming a normal force f_n, the local friction force can be computed as ${}^{J1}\begin{bmatrix} f_{t,x} \\ f_{t,y} \end{bmatrix} = \begin{bmatrix} \mu_x f_n \\ \mu_y f_n \end{bmatrix}$
dryFrictionProportionalZone	Real		0.	limit velocity [m/s] up to which the friction is proportional to velocity (for regularization / avoid numerical oscillations)
viscousFriction	Vector2D		[0,0]	viscous friction coefficients [SI:1/(m/s)] in local marker 1 joint $J1$ coordinates; proportional to slipping velocity, leading to increasing slipping friction force for increasing slipping velocity
rollingFrictionViscous	Real		0.	rolling friction [SI:1], which acts against the velocity of the trail on ground and leads to a force proportional to the contact normal force; currently, only implemented for disc axis parallel to ground!
useLinearProportionalZone	Bool		False	if True, a linear proportional zone is used; the linear zone performs better in implicit time integration as the Jacobian has a constant tangent in the sticking case
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectConnectorRollingDiscPenalty			parameters for visualization of item

The item VObjectConnectorRollingDiscPenalty has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
discWidth	float		0.1	width of disc for drawing
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

---

8.6.11.1 DESCRIPTION of ObjectConnectorRollingDiscPenalty:

Information on input parameters:

input parameter	symbol	description see tables above
markerNumbers	$[m0, m1]^{\mathrm{T}}$	
nodeNumber	n_d	
discAxis	${}^{m1}\mathbf{w}_1$, $\lvert {}^{m1}\mathbf{w}_1 \rvert = 1$	
planeNormal	${}^{m0}\mathbf{v}_{PN}$, $\lvert {}^{m0}\mathbf{v}_{PN} \rvert = 1$	
dryFrictionAngle	α_t	
contactStiffness	k_c	
contactDamping	d_c	
dryFriction	$[\mu_x, \mu_y]^{\mathrm{T}}$	
dryFrictionProportionalZone	v_μ	
viscousFriction	$[d_x, d_y]^{\mathrm{T}}$	
rollingFrictionViscous	μ_r	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position	${}^{0}\mathbf{p}_{G}$	current global position of contact point between rolling disc and ground
Velocity	${}^{0}\mathbf{v}_{trail}$	current velocity of the trail (according to motion of the contact point along the trail!) in global coordinates; this is not the velocity of the contact point!
VelocityLocal	${}^{J1}\mathbf{v}$	relative slip velocity at contact point in special $J1$ joint coordinates
ForceLocal	${}^{J1}\mathbf{f} = {}^{0}[f_{t,x}, f_{t,y}, f_n]^{\mathrm{T}}$	contact forces acting on disc, in special $J1$ joint coordinates, see section Connector Forces, $f_{t,x}$ being the lateral force (parallel to ground plane), $f_{t,y}$ being the longitudinal force and f_n being the contact normal force
RotationMatrix	${}^{0,J1}\mathbf{A} = [{}^{0}\mathbf{w}_{lat}, {}^{0}\mathbf{w}_2, {}^{0}\mathbf{v}_{PN}]$	transformation matrix of special joint coordinates $J1$ to global coordinates

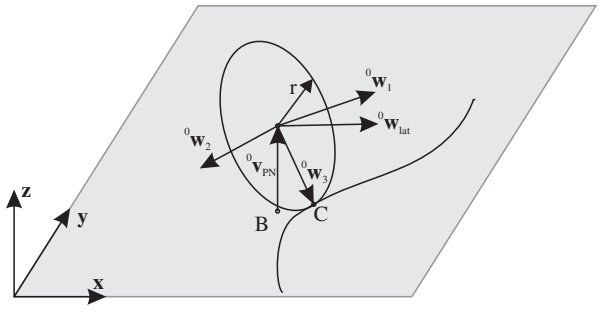
8.6.11.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^{0}\mathbf{p}_{m0}$	current global position which is provided by marker m0, any ground reference point; currently unused
marker m0 orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker m0; currently unused
marker m1 position	${}^{0}\mathbf{p}_{m1}$	center of disc
marker m1 orientation	${}^{0,m1}\mathbf{A}$	current rotation matrix provided by marker m1

data coordinates	$\mathbf{x} = [x_0, x_1, x_2]^{\mathrm{T}}$	data coordinates for $[x_0, x_1]$: hold the sliding velocity in lateral and longitudinal direction of last discontinuous iteration; x_2: represents gap of last discontinuous iteration (in contact normal direction)
marker m1 velocity	${}^0\mathbf{v}_{m1}$	accordingly
marker m1 angular velocity	${}^0\boldsymbol{\omega}_{m1}$	current angular velocity vector provided by marker m1
ground normal vector	${}^0\mathbf{v}_{PN} = {}^{0,m0}\mathbf{A}\ {}^{m0}\mathbf{v}_{PN}$	normalized normal vector to the ground body (rotates with marker $m0$ if not fixed to ground)
ground position B	${}^0\mathbf{p}_B$	disc center point projected on ground (normal projection)
ground position C	${}^0\mathbf{p}_C$	contact point of disc with ground
ground velocity C	${}^0\mathbf{v}_C$	velocity of disc at ground contact point (must be zero at end of iteration)
wheel axis vector	${}^0\mathbf{w}_1 = {}^{0,m1}\mathbf{A}\ {}^{m1}\mathbf{w}_1$	normalized disc axis vector in global coordinates
longitudinal vector	${}^0\mathbf{w}_2$	vector in longitudinal (motion) direction
contact point vector	${}^0\mathbf{w}_3$	normalized vector from disc center point in direction of contact point C
lateral vector	${}^0\mathbf{w}_{lat} = {}^0\mathbf{v}_{PN} \times {}^0\mathbf{w}_2$	vector in lateral direction, parallel to ground plane
$D1$ transformation matrix	${}^{0,D1}\mathbf{A} = [{}^0\mathbf{w}_1, {}^0\mathbf{w}_2, {}^0\mathbf{w}_3]$	transformation of special disc coordinates $D1$ to global coordinates
connector forces	${}^{J1}\mathbf{f} = [f_{t,x}, f_{t,y}, f_n]^{\mathrm{T}}$	joint force vector at contact point in joint 1 coordinates: x=lateral direction, y=longitudinal direction, z=plane normal (contact normal)

8.6.11.3 Geometric relations

The main geometrical setup is shown in the following figure:



First, the contact point ${}^0\mathbf{p}_C$ must be computed. With the helper vector,

$$ {}^0\mathbf{x} = {}^0\mathbf{w}_1 \times {}^0\mathbf{v}_{PN} \tag{8.318} $$

we create a disc coordinate system $D1$ (${}^0\mathbf{w}_1$, ${}^0\mathbf{w}_2$, ${}^0\mathbf{w}_3$), with the longitudinal direction,

$$ {}^0\mathbf{w}_2 = \frac{1}{|{}^0\mathbf{x}|}\,{}^0\mathbf{x} \tag{8.319} $$

and the vector to the contact point,

$$ {}^0\mathbf{w}_3 = {}^0\mathbf{w}_1 \times {}^0\mathbf{w}_2 \tag{8.320} $$

The vector from marker $m0$ position to the contact point can be computed from

$$ {}^0\mathbf{p}_C = {}^0\mathbf{p}_{m1} + r \cdot {}^0\mathbf{w}_3 - {}^0\mathbf{p}_{m0} \tag{8.321} $$

The velocity of the contact point at the disc is computed from,

$$ {}^0\mathbf{v}_C = {}^0\mathbf{v}_{m1} + {}^0\boldsymbol{\omega}_{m1} \times (r \cdot {}^0\mathbf{w}_3) - \left({}^0\mathbf{v}_{m0} + {}^0\boldsymbol{\omega}_{m0} \times {}^0\mathbf{p}_C\right) \tag{8.322} $$

A second coordinate system, denoted as $J1$, is defined by vectors $({}^0\mathbf{w}_{lat},\ {}^0\mathbf{w}_2,\ {}^0\mathbf{v}_{PN})$, using

$$ {}^0\mathbf{w}_{lat} = {}^0\mathbf{v}_{PN} \times {}^0\mathbf{w}_2 \tag{8.323} $$

Note that **in the case that** the rolling axis ${}^0\mathbf{w}_1$ lies in the rolling plane, we obtain the special case ${}^0\mathbf{w}_{lat} = {}^0\mathbf{w}_1$ and ${}^0\mathbf{w}_3 = -{}^0\mathbf{v}_{PN}$.

8.6.11.4 Computation of normal and tangential forces

The connector forces at the contact point C are computed as follows. The normal contact force reads

$$ f_n = \left(k_c \cdot {}^0\mathbf{p}_C + d_c \cdot {}^0\mathbf{v}_C\right)^{\mathrm{T}} {}^0\mathbf{v}_{PN} \ . \tag{8.324} $$

Note that due to the projection onto ${}^0\mathbf{v}_{PN}$, this equation also works for inclined planes and reference points, that are not at $[0,0,0]^{\mathrm{T}}$. The inplane velocity in joint coordinates,

$$ {}^{J1}\mathbf{v}_t = [{}^0\mathbf{v}_C^{\mathrm{T}}\, {}^0\mathbf{w}_{lat},\ {}^0\mathbf{v}_C^{\mathrm{T}}\, {}^0\mathbf{w}_2]^{\mathrm{T}} \ , \tag{8.325} $$

is used for the computation of tangential forces,

$$ {}^{J1}\mathbf{f}_t = [f_{t,x},\ f_{t,y}]^{\mathrm{T}} = {}^{J1}\boldsymbol{\mu} \cdot \left(\phi(|\mathbf{v}_t|, v_\mu) \cdot f_n \cdot {}^{J1}\mathbf{e}_t\right) \ , \tag{8.326} $$

with the regularization function, see Geradin and Cardona [16] (Sec. 7.9.3), if `useLinearProportionalZone=False`,

$$ \phi(v, v_\mu) = \begin{cases} \left(2 - \dfrac{v}{v_\mu}\right)\dfrac{v}{v_\mu} & \text{if} \quad v \le v_\mu \\ 1 & \text{if} \quad v > v_\mu \end{cases} \tag{8.327} $$

and the linear regularization function, if `useLinearProportionalZone=True`,

$$ \phi(v, v_\mu) = \begin{cases} \dfrac{v}{v_\mu} & \text{if} \quad v \le v_\mu \\ 1 & \text{if} \quad v > v_\mu \end{cases} \tag{8.328} $$

The direction of tangential slip is given as

$$ {}^{J1}\mathbf{e}_t = \begin{cases} \dfrac{{}^{J1}\mathbf{v}_t}{|\mathbf{v}_t|} & \text{if} \quad |\mathbf{v}_t| > 0 \\ \begin{bmatrix} 0 \\ 0 \end{bmatrix} & \text{else} \end{cases} \tag{8.329} $$

The friction coefficient matrix ${}^{J1}\boldsymbol{\mu}$ is given in joint coordinates and computed from

$$ {}^{J1}\boldsymbol{\mu} = \begin{bmatrix} \mu_x + d_x \cdot |\mathbf{v}_t| & 0 \\ 0 & \mu_y + d_y \cdot |\mathbf{v}_t| \end{bmatrix} \tag{8.330} $$

where for isotropic behaviour of surface and wheel, it will give a diagonal matrix with the friction coefficient in the diagonal. In case that the dry friction angle α_t is not zero, the $\boldsymbol{\mu}$ changes to

$$ {}^{J1}\boldsymbol{\mu} = \begin{bmatrix} \cos(\alpha_t) & \sin(\alpha_t) \\ -\sin(\alpha_t) & \cos(\alpha_t) \end{bmatrix} \begin{bmatrix} \mu_x + d_x \cdot |\mathbf{v}_t| & 0 \\ 0 & \mu_y + d_y \cdot |\mathbf{v}_t| \end{bmatrix} \begin{bmatrix} \cos(\alpha_t) & -\sin(\alpha_t) \\ \sin(\alpha_t) & \cos(\alpha_t) \end{bmatrix} \tag{8.331} $$

8.6.11.5 Connector forces

Finally, the connector forces read in joint coordinates

$$ {}^{J1}\mathbf{f} = \begin{bmatrix} f_{t,x} \\ f_{t,y} \\ f_n \end{bmatrix} \tag{8.332} $$

and in global coordinates, they are computed from

$$ {}^{0}\mathbf{f} = f_{t,x}\, {}^{0}\mathbf{w}_{lat} + f_{t,y}\, {}^{0}\mathbf{w}_{2} + f_n\, {}^{0}\mathbf{v}_{PN} \tag{8.333} $$

Due to the fact that the marker positions are not collocated with the contact point, there are additional torques that need to be considered in the action on the body. The torque onto the disc (marker $m1$) is computed as

$$ {}^{0}\boldsymbol{\tau}_{m1} = (r \cdot {}^{0}\mathbf{w}_{3}) \times {}^{0}\mathbf{f} \tag{8.334} $$

The torque onto the ground (marker $m0$) is computed as

$$ {}^{0}\boldsymbol{\tau}_{m0} = {}^{0}\mathbf{p}_{C} \times {}^{0}\mathbf{f} \tag{8.335} $$

Note that if `activeConnector = False`, we replace Eq. (8.332) with

$$ {}^{J1}\mathbf{f} = \mathbf{0} \tag{8.336} $$

For examples on ObjectConnectorRollingDiscPenalty see Relevant Examples and TestModels with weblink:

- `bicycleIftommBenchmark.py` (Examples/)
- `leggedRobot.py` (Examples/)
- `mobileMecanumWheelRobotWithLidar.py` (Examples/)
- `reinforcementLearningRobot.py` (Examples/)
- `carRollingDiscTest.py` (TestModels/)
- `laserScannerTest.py` (TestModels/)

- mecanumWheelRollingDiscTest.py (TestModels/)
- rollingCoinPenaltyTest.py (TestModels/)
- rollingDiscTangentialForces.py (TestModels/)
- rotatingTableTest.py (TestModels/)
- createFunctionsTest.py (TestModels/)
- createRollingDiscPenaltyTest.py (TestModels/)

8.6.12 ObjectContactConvexRoll

A contact connector representing a convex roll (marker 1) on a flat surface (marker 0, ground body, not moving) in global x-y plane. The connector is similar to ObjectConnectorRollingDiscPenalty, but includes a (strictly) convex shape of the roll defined by a polynomial. It is based on a penalty formulation and adds friction and slipping. The formulation is still under development and needs further testing. Note that the rolling body must have the reference point at the center of the disc.

Author: Manzl Peter

Additional information for ObjectContactConvexRoll:

- This `Object` has/provides the following types = `Connector`
- Requested `Marker` type = `Position` + `Orientation`
- Requested `Node` type = `GenericData`

The item **ObjectContactConvexRoll** with type = 'ContactConvexRoll' has the following parameters:

Name	type	size	default value	description
name	String		''	constraints's unique name
markerNumbers	ArrayMarkerIndex	2	[invalid [-1], invalid [-1]]	list of markers used in connector; $m0$ represents the ground, which can undergo translations but not rotations, and $m1$ represents the rolling body, which has its reference point (=local position [0,0,0]) at the roll's center point
nodeNumber	NodeIndex		invalid (-1)	node number of a NodeGenericData (size=3) for 3 dataCoordinates, needed for discontinuous iteration (friction and contact)
contactStiffness	Real		0.	normal contact stiffness [SI:N/m]
contactDamping	Real		0.	normal contact damping [SI:N/(m s)]
dynamicFriction	UReal		0.	dynamic friction coefficient for friction model, see StribeckFunction in exudyn.physics, Section 7.16
staticFrictionOffset	UReal		0.	static friction offset for friction model (static friction = dynamic friction + static offset), see StribeckFunction in exudyn.physics, Section 7.16
viscousFriction	UReal		0.	viscous friction coefficient (velocity dependent part) for friction model, see StribeckFunction in exudyn.physics, Section 7.16

exponentialDecayStatic	PReal		1e-3	exponential decay of static friction offset (must not be zero!), see StribeckFunction in exudyn.physics (named expVel there!), Section 7.16
frictionProportionalZone	UReal		1e-3	limit velocity [m/s] up to which the friction is proportional to velocity (for regularization / avoid numerical oscillations), see StribeckFunction in exudyn.physics (named regVel there!), Section 7.16
rollLength	UReal		0.	roll length [m], symmetric w.r.t. centerpoint
coefficientsHull	NumpyVector		[]	a vector of polynomial coefficients, which provides the polynomial of the CONVEX hull of the roll; hull(x) = $k_0 x^{n_p-1} + kx^{n_p-2} + \ldots + k_{n_p-2}x + k_{n_p-1}$
coefficientsHullDerivative	NumpyVector		[]	polynomial coefficients of the polynomial hull'(x)
coefficientsHullDDerivative	NumpyVector		[]	second derivative of the hull polynomial.
rBoundingSphere	UReal		0	The radius of the bounding sphere for the contact pre-check, calculated from the polynomial coefficients of the hull
pContact	Vector3D		[0,0,0]	The current potential contact point. Contact occures if pContact[2] < 0.
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectContactConvexRoll			parameters for visualization of item

The item VObjectContactConvexRoll has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

---

8.6.12.1 DESCRIPTION of ObjectContactConvexRoll:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
markerNumbers	$[m0, m1]^T$	
nodeNumber	n_d	

contactStiffness	k_c	
contactDamping	d_c	
dynamicFriction	μ_d	
staticFrictionOffset	μ_{s_off}	
viscousFriction	μ_v	
exponentialDecayStatic	v_{exp}	
frictionProportionalZone	v_{reg}	
rollLength	L	
coefficientsHull	$\mathbf{k} \in \mathbb{R}^{n_p}$	
coefficientsHullDerivative	$\mathbf{k}' \in \mathbb{R}^{n_p}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
Position	${}^0\mathbf{p}_C$	current global position of contact point between roller and ground
Velocity	${}^0\mathbf{v}_C$	current velocity of the trail (contact) point in global coordinates; this is the velocity with which the contact moves over the ground plane
Force	${}^0\mathbf{f}$	Roll-ground force in ground coordinates
Torque	${}^0\mathbf{m}$	Roll-ground torque in ground coordinates

8.6.12.2 Definition of quantities

intermediate variables	**symbol**	**description**
marker m0 position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker m0, any ground reference point; currently unused
marker m0 orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker m0; currently unused
marker m1 position	${}^0\mathbf{p}_{m1}$	center of roll
Contact position	${}^0\mathbf{p}_C$	Position of the Contact point C in the global frame 0
Position marker m1 to contact	${}^0\mathbf{p}_{\text{m1,C}}$	Position of the contact point C relative to the marker m1 in global frame
marker m1 orientation	${}^{0,m1}\mathbf{A}$	current rotation matrix provided by marker m1
data coordinates	$\mathbf{x} = [x_0, x_1, x_2]^{\mathrm{T}}$	data coordinates for $[x_0, x_1]$: hold the sliding velocity in lateral and longitudinal direction of last discontinuous iteration; x_2: represents gap of last discontinuous iteration (in contact normal direction)
marker m1 velocity	${}^0\mathbf{v}_{m1}$	current global velocity which is provided by marker m1

marker m1 angular velocity	${}^0\boldsymbol{\omega}_{m1}$	current angular velocity vector provided by marker m1
ground normal vector	${}^0\mathbf{n}$	normalized normal vector to the (moving, but not rotating) ground, by default [0,0,1]

8.6.12.3 Geometric relations

The geometrical setup is shown in Fig. 8.7. To calculate the contact point of the convex body of revolution the contact (ground) plane is rotated into the local frame of the body. In this local frame in which the generatrix of the body of revolution is described by the polynomial function

$$\mathrm{r}({}^b x) = \sum_{i=0}^{n} k_i \, x^{n-i} \tag{8.337}$$

with the coefficients of the hull a_i. As a pre-Check for the contact two spheres are put into both ends of the object with the maximum radius and only if one of these is in contact. The contact point ${}^{\mathrm{b}}\mathbf{p}_{\mathrm{m1,C}}$ is calculated relative to the bodies marker `m1` in the bodies local frame and transformed accordingly. The contact point C can for be calculated convex bodies by matching the derivative of the polynomial $r({}^b x)$ with the gradient of the contact plane, shown in Fig. 8.7, explained in detail in [40]. At the contact point a normal force $\mathbf{f}_{\mathrm{N}} = [0\ 0\ \mathrm{f_N}]^{\mathrm{T}}$ with

$$\mathrm{f_N} = \begin{cases} -(k_c \, z_{\mathrm{pen}} + d_c \, \dot{z}_{\mathrm{pen}}) & z_{\mathrm{pen}} > 0 \\ 0 & \text{else} \end{cases} \tag{8.338}$$

acts against the penetration of the ground. The penetration depth z_{pen} is the z-component of the position vector of the contact point relative to the ground frame ${}^0\mathbf{p}_{\mathrm{C}}$.
The revolution results in a velocity of

$${}^0\mathbf{v}_{\mathrm{C}} = {}^0\boldsymbol{\omega}_{\mathrm{m1}} \times {}^0\mathbf{p}_{\mathrm{m1,C}} \tag{8.339}$$

in the contact point, while the tangential component of the velocity of the body itself with the normal Vector to the contact plane $\mathbf{n}$ follows to

$${}^0\mathbf{v}_{\mathrm{m1,t}} = {}^0\mathbf{v}_{\mathrm{m1}} - {}^0\mathbf{n} \left({}^0\mathbf{n}^T \, {}^0\mathbf{v}_{\mathrm{m1}} \right). \tag{8.340}$$

Therefore the slip velocity of the body can be calculated with

$${}^0\mathbf{v}_{\mathrm{s}} = {}^0\mathbf{v}_{\mathrm{C}} - {}^0\mathbf{v}_{\mathrm{m1,t}} \tag{8.341}$$

and points in the direction

$${}^0\mathbf{r}_{s} = \frac{1}{\left\| {}^0\mathbf{v}_{\mathrm{s}} \right\|} {}^0\mathbf{v}_{\mathrm{s}}. \tag{8.342}$$

The slip force is then calculated

$${}^0\mathbf{f}_{\mathrm{s}} = \mu(\left\| {}^0\mathbf{v}_{\mathrm{s}} \right\|) \, \mathrm{f_N} \, {}^0\mathbf{r}_{\mathrm{s}} \tag{8.343}$$

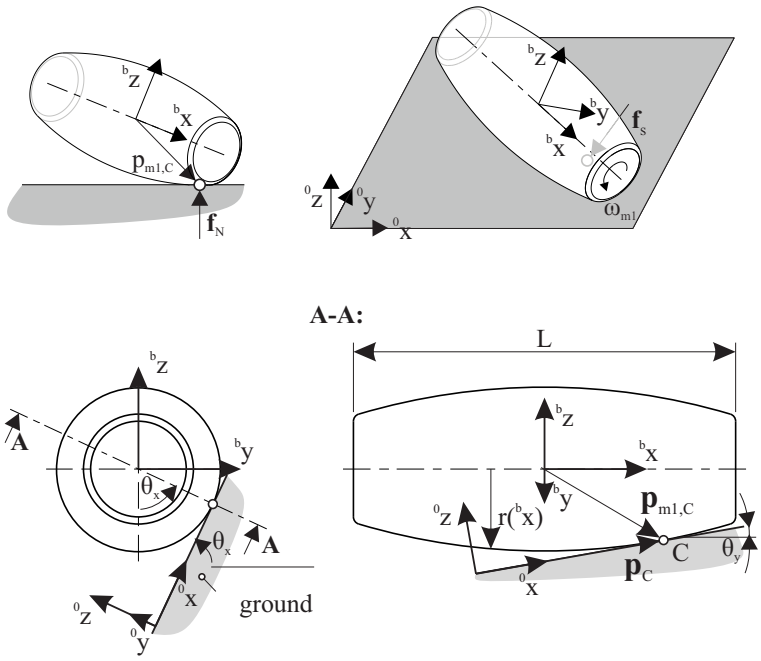


Figure 8.7: Sketch of the roller Dimensions. The rollers radius $r({}^b x)$ is described by the polynomial `coefficientsHull`.

and uses for the friction coefficient μ the regularized friction approach from the StribeckFunction, see Section 7.16. The torque

$$ {}^0\boldsymbol{\tau} = {}^0\mathbf{p}_{\mathrm{m1,C}} \times ({}^0\mathbf{f}_{\mathrm{N}} + {}^0\mathbf{f}_{\mathrm{s}}) \tag{8.344} $$

acts onto the body, resulting from the slip force acting not in the bodies center.

For examples on ObjectContactConvexRoll see Relevant Examples and TestModels with weblink:

- `ConvexContactTest.py` (TestModels/)

8.6.13 ObjectContactCoordinate

A penalty-based contact condition for one coordinate; the contact gap g is defined as $g = marker.value[1] - marker.value[0] - offset$; the contact force f_c is zero for $gap > 0$ and otherwise computed from $f_c = g * contactStiffness + \dot{g} * contactDamping$; during Newton iterations, the contact force is actived only, if $dataCoordinate[0] <= 0$; dataCoordinate is set equal to gap in nonlinear iterations, but not modified in Newton iterations.

Additional information for ObjectContactCoordinate:

- This `Object` has/provides the following types = `Connector`
- Requested `Marker` type = `Coordinate`
- Requested `Node` type = `GenericData`

The item **ObjectContactCoordinate** with type = 'ContactCoordinate' has the following parameters:

Name	type	size	default value	description
name	String		"	connector's unique name
markerNumbers	ArrayMarkerIndex		[invalid [-1], invalid [-1]]	markers define contact gap
nodeNumber	NodeIndex		invalid (-1)	node number of a NodeGenericData for 1 dataCoordinate (used for active set strategy ==> holds the gap of the last discontinuous iteration)
contactStiffness	UReal		0.	contact (penalty) stiffness [SI:N/m]; acts only upon penetration
contactDamping	UReal		0.	contact damping [SI:N/(m s)]; acts only upon penetration
offset	Real		0.	offset [SI:m] of contact
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectContactCoordinate			parameters for visualization of item

The item VObjectContactCoordinate has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = diameter of spring; size == -1.f means that default connector size is used

color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

---

8.6.13.1 DESCRIPTION of ObjectContactCoordinate:

For examples on ObjectContactCoordinate see Relevant Examples and TestModels with weblink:

- ANCFcontactCircle.py (Examples/)
- ANCFcontactCircle2.py (Examples/)
- ANCFcontactCircleTest.py (TestModels/)
- contactCoordinateTest.py (TestModels/)

8.6.14 ObjectContactCircleCable2D

A very specialized penalty-based contact condition between a 2D circle (=marker0, any Position-marker) on a body and an ANCFCable2DShape (=marker1, Marker: BodyCable2DShape), in xy-plane; a node NodeGenericData is required with the number of cordinates according to the number of contact segments; the contact gap g is integrated (piecewise linear) along the cable and circle; the contact force f_c is zero for $gap > 0$ and otherwise computed from $f_c = g * contactStiffness + \dot{g} * contactDamping$; during Newton iterations, the contact force is actived only, if $dataCoordinate[0] <= 0$; dataCoordinate is set equal to gap in nonlinear iterations, but not modified in Newton iterations.

Additional information for ObjectContactCircleCable2D:

- This `Object` has/provides the following types = `Connector`
- Requested `Marker` type = `_None`
- Requested `Node` type = `GenericData`

The item **ObjectContactCircleCable2D** with type = 'ContactCircleCable2D' has the following parameters:

Name	type	size	default value	description
name	String		"	connector's unique name
markerNumbers	ArrayMarkerIndex		[invalid [-1], invalid [-1]]	markers define contact gap
nodeNumber	NodeIndex		invalid (-1)	node number of a NodeGenericData for nSegments dataCoordinates (used for active set strategy ==> hold the gap of the last discontinuous iteration and the friction state)
numberOfContactSegments	Index		3	number of linear contact segments to determine contact; each segment is a line and is associated to a data (history) variable; must be same as in according marker
contactStiffness	UReal		0.	contact (penalty) stiffness [SI:N/m/(contact segment)]; the stiffness is per contact segment; specific contact forces (per length) f_N act in contact normal direction only upon penetration
contactDamping	UReal		0.	contact damping [SI:N/(m s)/(contact segment)]; the damping is per contact segment; acts in contact normal direction only upon penetration
circleRadius	UReal		0.	radius [SI:m] of contact circle
offset	Real		0.	offset [SI:m] of contact, e.g. to include thickness of cable element

activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectContactCircleCable2D			parameters for visualization of item

The item VObjectContactCircleCable2D has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
showContactCircle	Bool		True	if True and show=True, the underlying contact circle is shown; uses circleTiling*4 for tiling (from VisualizationSettings.general)
drawSize	float		-1.	drawing size = diameter of spring; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

---

8.6.14.1 DESCRIPTION of ObjectContactCircleCable2D:

8.6.14.2 Connector equations

Geometry and equations are very similar to `ObjectContactFrictionCircleCable2D`, while friction is not used and no torque is transferred to the circle object.

For examples on ObjectContactCircleCable2D see Relevant Examples and TestModels with weblink:

- `ANCFcontactCircle.py` (Examples/)
- `ANCFcontactCircle2.py` (Examples/)
- `ANCFmovingRigidbody.py` (Examples/)
- `ANCFslidingJoint2D.py` (Examples/)
- `ANCFcontactCircleTest.py` (TestModels/)
- `ANCFmovingRigidBodyTest.py` (TestModels/)
- `ANCFslidingAndALEjointTest.py` (TestModels/)

8.6.15 ObjectContactFrictionCircleCable2D

A very specialized penalty-based contact/friction condition between a 2D circle in the local x/y plane (=marker0, a RigidBody Marker, from node or object) on a body and an ANCFCable2DShape (=marker1, Marker: BodyCable2DShape), in xy-plane; a node NodeGenericData is required with 3×(number of contact segments) – containing per segment: [contact gap, stick/slip (stick=0, slip=+-1, undefined=-2), last friction position]. The connector works with Cable2D and ALECable2D, HOWEVER, due to conceptual differences the (tangential) frictionStiffness cannot be used with ALECable2D; if using, it gives wrong tangential stresses, even though it may work in general.

Additional information for ObjectContactFrictionCircleCable2D:

- This `Object` has/provides the following types = `Connector`
- Requested `Marker` type = `_None`
- Requested `Node` type = `GenericData`

The item **ObjectContactFrictionCircleCable2D** with type = 'ContactFrictionCircleCable2D' has the following parameters:

Name	type	size	default value	description
name	String		"	connector's unique name
markerNumbers	ArrayMarkerIndex		[invalid [-1], invalid [-1]]	a marker $m0$ with position and orientation and a marker $m1$ of type BodyCable2DShape; together defining the contact geometry
nodeNumber	NodeIndex		invalid (-1)	node number of a NodeGenericData with 3 $\times n_{cs}$ dataCoordinates (used for active set strategy $\rightarrow$ hold the gap of the last discontinuous iteration, friction state (+-1=slip, 0=stick, -2=undefined) and the last sticking position; initialize coordinates with list [0.1]*n_{cs}+[-2]*n_{cs}+[0.]*n_{cs}, meaning that there is no initial contact with undefined slip/stick
numberOfContactSegments	PInt		3	number of linear contact segments to determine contact; each segment is a line and is associated to a data (history) variable; must be same as in according marker
contactStiffness	UReal		0.	contact (penalty) stiffness [SI:N/m/(contact segment)]; the stiffness is per contact segment; specific contact forces (per length) f_n act in contact normal direction only upon penetration

contactDamping	UReal		0.	contact damping [SI:N/(m s)/(contact segment)]; the damping is per contact segment; acts in contact normal direction only upon penetration
frictionVelocityPenalty	UReal		0.	tangential velocity dependent penalty coefficient for friction [SI:N/(m s)/(contact segment)]; the coefficient causes tangential (contact) forces against relative tangential velocities in the contact area
frictionStiffness	UReal		0.	tangential displacement dependent penalty/stiffness coefficient for friction [SI:N/m/(contact segment)]; the coefficient causes tangential (contact) forces against relative tangential displacements in the contact area
frictionCoefficient	UReal		0.	friction coefficient [SI: 1]; tangential specific friction forces (per length) f_t must fulfill the condition $f_t \leq \mu f_n$
circleRadius	UReal		0.	radius [SI:m] of contact circle
useSegmentNormals	Bool		True	True: use normal and tangent according to linear segment; this is appropriate for very long (compared to circle) segments; False: use normals at segment points according to vector to circle center; this is more consistent for short segments, as forces are only applied in beam tangent and normal direction
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectContactFrictionCircleCable2D			parameters for visualization of item

The item VObjectContactFrictionCircleCable2D has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set True, if item is shown in visualization and false if it is not shown; note that only normal contact forces can be drawn, which are approximated by $k_c \cdot g$ (neglecting damping term)
showContactCircle	Bool		True	if True and show=True, the underlying contact circle is shown; uses circleTiling*4 for tiling (from VisualizationSettings.general)
drawSize	float		-1.	drawing size = diameter of spring; size == -1.f means that default connector size is used

color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

---

8.6.15.1 DESCRIPTION of ObjectContactFrictionCircleCable2D:

Information on input parameters:

input parameter	symbol	description see tables above
markerNumbers	$[m0, m1]^T$	
nodeNumber	n_g	
numberOfContactSegments	n_{cs}	
contactStiffness	k_c	
contactDamping	d_c	
frictionVelocityPenalty	μ_v	
frictionStiffness	μ_k	
frictionCoefficient	μ	
circleRadius	r	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Coordinates	$[u_{t,0}, g_0, u_{t,1}, g_1, \ldots, u_{t,n_{cs}}, g_{n_{cs}}]^T$	local (relative) displacement in tangential (**t**) and normal (**n**) direction per segment (n_{cs}); values are only provided in case of contact, otherwise zero; tangential displacement is only non-zero in case of sticking!
Coordinates_t	$[v_{t,0}, v_{n,0}, v_{t,1}, v_{n,1}, \ldots, v_{t,n_{cs}}, v_{n,n_{cs}}]^T$	local (relative) velocity in tangential (**t**) and normal (**n**) direction per segment (n_{cs}); values are only provided in case of contact, otherwise zero
ForceLocal	$[f_{t,0}, f_{n,0}, f_{t,1}, f_{n,1}, \ldots, f_{t,n_{cs}}, f_{n,n_{cs}}]^T$	local contact forces in tangential (**t**) and normal (**n**) direction per segment (n_{cs})

8.6.15.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^0\mathbf{p}_{m0}$	represents current global position of the circle's centerpoint
marker m0 velocity	${}^0\mathbf{v}_{m0}$	current global velocity which is provided by marker m0

marker m1		represents the 2D ANCF cable
data node	$\mathbf{x} = [x_i, \ldots, x_{3n_{cs}-1}]^T$	coordinates of node with node number n_{GD}
data coordinates for segment i	$[x_i, x_{n_{cs}+i}, x_{2\cdot n_{cs}+i}]^T$ = $[x_{gap}, x_{isSlipStick}, x_{lastStick}]^T$, with $i \in [0, n_{cs}-1]$	The data coordinates include the gap x_{gap}, the stick-slip state $x_{isSlipStick}$ and the previous sticking position $x_{lastStick}$ as computed in the PostNewtonStep, see description below.
shortest distance to segment s_i	$\mathbf{d}_{g,i}$	shortest distance of center of circle to contact segment, considering the endpoint of the segment

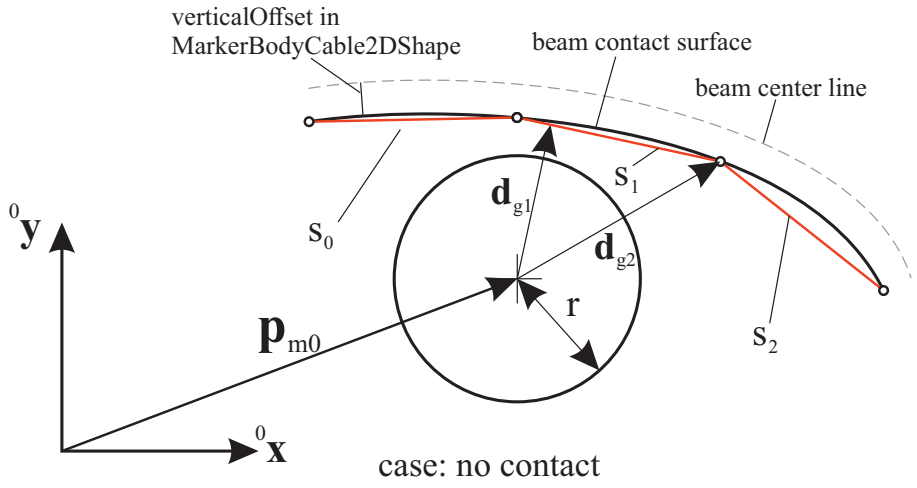


Figure 8.8: Sketch of cable, contact segments and circle; showing case without contact, $|\mathbf{d}_{g1}| > r$, while contact occurs with $|\mathbf{d}_{g1}| \leq r$; the shortest distance vector $\mathbf{d}_{g1}$ is related to segment s_1 (which is perpendicular to the the segment line) and $\mathbf{d}_{g2}$ is the shortest distance to the end point of segment s_2, not being perpendicular.

8.6.15.3 Connector forces: contact geometry

The connector represents a force element between a 'circle' (or cylinder) represented by a marker $m0$, which has position and orientation, and an `ANCFCable2D` beam element (denoted as 'cable') represented by a `MarkerBodyCable2DShape` $m1$. The cable with reference length L is discretized by splitting into n_{cs} straight segments s_i, located between points p_i and p_{i+1}. Note that these points can be placed with an offset from the cable centerline, see `verticalOffset` defined in `MarkerBodyCable2DShape`. In order to compute the gap function for a line segment, the shortest distance of one line segment with points $\mathbf{p}_i$, $\mathbf{p}_{i+1}$ to the circle's centerpoint given by the marker $\mathbf{p}_{m0}$ is computed. All computations here are performed in the global coordinates system (0), including edge points of every segment.

With the intermediate quantities (all of them related to segment s_i)[6],

$$\mathbf{v}_s = \mathbf{p}_{i+1} - \mathbf{p}_i, \quad \mathbf{v}_p = \mathbf{p}_{m0} - \mathbf{p}_i, \quad n = \mathbf{v}_s^T \mathbf{v}_p, \quad d = \mathbf{v}_s^T \mathbf{v}_s \tag{8.345}$$

[6] we omit s_i in some terms for brevity!

and assuming that $d \neq 0$ (otherwise the two segment points would be identical and the shortest distance would be $d_g = |\mathbf{v}_p|$), we find the relative position ρ of the shortest (projected) point on the segment, which runs from 0 to 1 if lying on the segment, as

$$\rho = \frac{n}{d} \tag{8.346}$$

We distinguish 3 cases (see also Fig. 8.8 for cases 1 and 2):

1. If $\rho \leq 0$, the shortest distance would be the distance to point $\mathbf{p}_p = \mathbf{p}_i$, reading

$$d_g = |\mathbf{p}_{m0} - \mathbf{p}_i| \quad (\rho \leq 0) \tag{8.347}$$

2. If $\rho \geq 1$, the shortest distance would be the distance to point $\mathbf{p}_p = \mathbf{p}_{i+1}$, reading

$$d_g = |\mathbf{p}_{m0} - \mathbf{p}_{i+1}| \quad (\rho \geq 1) \tag{8.348}$$

3. Finally, if $0 < \rho < 1$, then the shortest distance has a projected point somewhere on the segment with the point (projected on the segment)

$$\mathbf{p}_p = \mathbf{p}_i + \rho \cdot \mathbf{v}_s \tag{8.349}$$

and the distance

$$d_g = |\mathbf{d}_g| = \sqrt{\mathbf{v}_p^{\mathrm{T}} \mathbf{v}_p - (n^2)/d} \tag{8.350}$$

Here, the shortest distance vector for every segment results from the projected point $\mathbf{p}_p$ of the above mentioned cases, see also Fig. 8.8, with the relation

$$\mathbf{d}_g = \mathbf{d}_{g,s_i} = \mathbf{p}_{m0} - \mathbf{p}_p \ . \tag{8.351}$$

The contact gap for a specific point for segment s_i is in general defined as

$$g = g_{s_i} = d_g - r \ . \tag{8.352}$$

using $d_g = |\mathbf{d}_g|$.

8.6.15.4 Contact frame and relative motion

Irrespective of the choice of `useSegmentNormals`, the contact normal vector $\mathbf{n}_{s_i}$ and tangential vector $\mathbf{t}_{s_i}$ are defined per segment as

$$\mathbf{n}_{s_i} = \mathbf{n} = [n_0, n_1]^{\mathrm{T}} = \frac{1}{|\mathbf{d}_{g,s_i}|} \mathbf{d}_{g,s_i}, \quad \mathbf{t}_{s_i} = \mathbf{t} = [-n_1, n_0]^{\mathrm{T}} \tag{8.353}$$

The vectors $\mathbf{t}_{s_i}$ and $\mathbf{n}_{s_i}$ define the local (contact) frame for further computations.

The velocity at the closest point of the segment s_i is interpolated using ρ and computed as

$$\dot{\mathbf{p}}_p = (1 - \rho) \cdot \mathbf{v}_i + \rho \cdot \mathbf{v}_{i+1} \tag{8.354}$$

Alternatively, $\dot{\mathbf{p}}_p$ could be computed from the cable element by evaluating the velocity at the contact points, but we feel that this choice is more consistent with the computations at position level.

The gap velocity v_n ($\neq \dot{g}$) thus reads

$$v_n = \left(\dot{\mathbf{p}}_p - \dot{\mathbf{p}}_{m0}\right)\mathbf{n} \tag{8.355}$$

In a similar, the tangential velocity reads

$$v_t = \left(\dot{\mathbf{p}}_p - \dot{\mathbf{p}}_{m0}\right)\mathbf{t} \tag{8.356}$$

In case of `frictionStiffness != 0`, we continuously track the sticking position at which the cable element (or segment) and the circle previously sticked together, similar as proposed by Lugrís et al. [39]. The difference here to the latter reference, is that we explicitly exclude switching from Newton's method and that Lugrís et al. used contact points, while we use linear segments. For a simple 1D example using this position based approach for friction, see `Examples/lugreFrictionText.py`, which compares the traditional LuGre friction model [10] with the position based model with tangential stiffness.

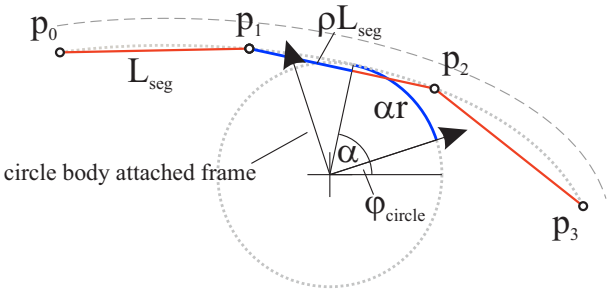


Figure 8.9: Calculation of last sticking position; blue parts mark the sticking position calculated as $x^*_{curStick}$.

Because there is the chance to wind/unwind relative to the (last) sticking position without slipping, the following strategy is used. In case of sliding (which could be the last time sliding before sticking), we compute the **current sticking position**, see Fig. 8.9, as the sum of the relative position at the segment s

$$x_{s,curStick} = \rho \cdot L_{seg} \tag{8.357}$$

in which $\rho \in [0,1]$ denotes the relative position of contact at the segment with reference length $L_{seg} = \frac{L}{n_{cs}}$. The relative position at the circle c is

$$x_{c,curStick} = \alpha \cdot r \tag{8.358}$$

We immediately see, that under pure rolling[7],

$$x_{s,curStick} + x_{c,curStick} = \text{const.} \tag{8.359}$$

Note that the `verticalOffset` from the cable center line, as defined in the related `MarkerBodyCable2DShape`, influences the behavior significantly, which is why we recommend to use `verticalOffset=0` whenever this is an appropriate assumption. Thus, the current sticking position $x_{curStick}$ is computed per

[7] neglecting the effects of small penetration, usually much smaller than shown for visibility in Fig. 8.9.

segment as

$$x^*_{curStick} = x_{s,curStick} + x_{c,curStick}, \tag{8.360}$$

Due to the possibility of switching of $\alpha + \phi$ between $-\pi$ and π, the result is normalized to

$$x_{curStick} = x^*_{curStick} - \text{floor}\left(\frac{x^*_{curStick}}{2\pi \cdot r} + \frac{1}{2}\right) \cdot 2\pi \cdot r, \tag{8.361}$$

which gives $\bar{x}_{curStick} \in [-\pi \cdot r, \pi \cdot r]$, which is stored in the 3rd data variable (per segment). The function floor() is a standardized version of rounding, available in C and Python programming languages. In the `PostNewtonStep`, the last sticking position is computed, $x_{lastStick} = x_{curStick}$, and it is also available in the `startOfStep` state.

8.6.15.5 Contact forces: definition

The contact force f_n is zero for $g > 0$ and otherwise computed from

$$f_n = k_c \cdot g + d_c \cdot v_n \tag{8.362}$$

NOTE that currently, there is only a linear spring-damper model available, assuming that the impact dynamics is not dominating (such as in belt drives or reeving systems).

Friction forces are primarily based on relative (tangential) velocity at each segment. The 'linear' friction force, based on the velocity penalty parameter μ_v reads

$$f_t^{(lin)} = \mu_v \cdot v_t \ , \tag{8.363}$$

8.6.15.6 PostNewtonStep

In general, see the solver flow chart for the `DiscontinuousIteration`, see Fig. 11.5, should be considered when reading this description. Every step is started with values `startOfStep`, while current values are iterated and updated in the Newton or `DiscontinuousIteration`.

The `PostNewtonStep` computes 3 values per segment, which are used for computation of contact forces, irrespectively of the current geometryof the contact. The `PostNewtonStep` is called after every full Newton method and evaluates the current state w.r.t. the assumed data variables. If the assumptions do not fit, new data variables are computed. This is necessary in order to avoid discontinuities in the equations, while otherwise the Newton iterations would not (or only slowly) converge.

The data variables per segment are

$$[x_{gap}, x_{isSlipStick}, x_{lastStick}] \tag{8.364}$$

Here, x_{gap} contains the gap of the segment (≤ 0 means contact), $x_{lastStick}$ is described in Eq. (8.361), and $x_{isSlipStick}$ defines the stick or slip case,

- $x_{isSlipStick} = -2$: undefined, used for initialization
- $x_{isSlipStick} = 0$: sticking
- $x_{isSlipStick} = \pm 1$: slipping, sign defines slipping direction

The basic algorithm in the `PostNewtonStep`, with all operations given for any segment s_i, can be summarized as follows:

I. Evaluate gap per segment g using Eq. (8.352) and store in data variable: $x_{gap} = g$

II. If $x_{gap} < 0$ and ($\mu_v \neq 0$ or $\mu_k \neq 0$):

1. Compute contact force f_n according to Eq. (8.362)
2. Compute current sticking position $x_{curStick}$ according to Eq. (8.360)[8]
3. Retrieve `startOfStep` sticking position[9] in $x_{lastStick}^{startOfStep}$ and compute and normalize difference in sticking position[10]:

$$\Delta x_{stick}^{*} = x_{curStick} - x_{lastStick}^{startOfStep}, \quad \Delta x_{stick} = \Delta x_{stick}^{*} - \text{floor}\left(\frac{\Delta x_{stick}^{*}}{2\pi \cdot r} + \frac{1}{2}\right) \cdot 2\pi \cdot r \tag{8.365}$$

4. Compute linear tangential force for friction stiffness and velocity penalty:

$$f_{t,lin} = \mu_v \cdot v_t + \mu_k \Delta x_{stick} \tag{8.366}$$

5. Compute tangential force according to Coulomb friction model [11]:

$$f_t = \begin{cases} f_t^{(lin)}, & \text{if} \quad |f_t^{(lin)}| \leq \mu \cdot |f_n| \\ \mu \cdot |f_n| \cdot \text{Sign}(\Delta x_{stick}), & \text{else} \end{cases} \tag{8.367}$$

6. In the case of slipping, given by $|f_t^{(lin)}| > \mu \cdot |f_n|$, we update the last sticking position in the data variable, such that the spring is pre-tensioned already,

$$x_{lastStick} = x_{curStick} - \text{Sign}(\Delta x_{stick}) \frac{\mu \cdot |f_n|}{\mu_k}, \quad x_{isSlipStick} = \text{Sign}(\Delta x_{stick}) \tag{8.368}$$

7. In the case of sticking, given by $|f_t^{(lin)}| \leq \mu \cdot |f_n|$: Set $x_{isSlipStick} = 0$ and, if $x_{isSlipStick}^{startOfStep} = -2$ (undefined), we update $x_{lastStick} = x_{curStick}$, while otherwise, $x_{lastStick}$ is unchanged.

III. If $x_{gap} > 0$ or ($\mu_v == 0$ and $\mu_k == 0$), we set $x_{isSlipStick} = -2$ (undefined); this means that in the next step (if this step is accepted), there is no stored sticking position.

IV. Compute an error $\varepsilon_{PNS} = \varepsilon_{PNS}^{n} + \varepsilon_{PNS}^{t}$, with physical units forces (per segment point), for `PostNewtonStep`:

1. if gap $x_{gap,lastPNS}$ of previous `PostNewtonStep` had different sign to current gap, set

$$\varepsilon_{PNS}^{n} = k_c \cdot \|x_{gap} - x_{gap,lastPNS}\| \tag{8.369}$$

while otherwise $\varepsilon_{PNS}^{n} = 0$.

[8] terms are only evaluated if $\mu_k \neq 0$

[9] Importantly, the `PostNewtonStep` always refers to the `startOfStep` state in the sticking position, because in the discontinuous iterations, the algorithm could switch to slipping in between and override the last sticking position in the current step

[10] in case that $x_{isSlipStick} = -2$, meaning that there is no stored sticking position, we set $\Delta x_{stick} = 0$

[11] note that the sign of Δx_{stick} is used here, but alternatively we may also use the sign of $f_{t,lin}$

2. if stick-slip-state $x_{isSlipStick,lastPNS}$ of previous `PostNewtonStep` is different from current $x_{isSlipStick}$, set

$$\varepsilon^t_{PNS} = \| \left(\|f_t^{(lin)}\| - \mu \cdot |f_n| \right) \| \tag{8.370}$$

while otherwise $\varepsilon^t_{PNS} = 0$.

Note that the `PostNewtonStep` is iterated and the data variables are updated continuously until convergence, or until a max. number of iterations is reached. If `ignoreMaxIterations` == 0, computation will continue even if no convergence is reached after the given number of iterations. This will lead so larger errors in such steps, but may have less influence on the overall solution if such cases are rare.

8.6.15.7 Computation of connector forces in Newton

The computation of LHS terms, the action of forces produced by the contact-friction element, is done during Newton iterations and may not have discontinuous behavior, thus relating computations to data variables computed in the `PostNewtonStep`. For efficiency, the LHS computation is only performed, if the `PostNewtonStep` determined contact in any segment.

The operations are similar to the `PostNewtonStep`, but without switching. The following operations are performed for each segment s_i, if $x_{gap,s_i} <= 0$:

I. Compute contact force f_n, Eq. (8.362).

II. In case of sticking ($|x_{isSlipStick}| \neq 1$):

II.1 the current sticking position $x_{curStick}$ is computed from Eq. (8.360), and the difference of current and last sticking position reads[12]:

$$\Delta x^*_{stick} = x_{curStick} - x_{lastStick}, \quad \Delta x_{stick} = x^*_{stick} - \text{floor}\left(\frac{\Delta x^*_{stick}}{2\pi \cdot r} + \frac{1}{2} \right) \cdot 2\pi \cdot r \tag{8.371}$$

II.2 if the friction stiffness is μ_k == 0 or if $x_{isSlipStick}$ == −2, we set $\Delta x_{stick} = 0$

II.3 using the tangential velocity from Eq. (8.356), the tangent force follows as (even if it is larger than the sticking limit)

$$f_t = \mu_v \cdot v_t + \mu_k \Delta x_{stick} \tag{8.372}$$

III. In case of slipping ($|x_{isSlipStick}| = 1$), the tangential firction force is set as[13],

$$f_t = \mu \cdot |f_n| \cdot x_{isSlipStick}, \quad \text{else} \tag{8.373}$$

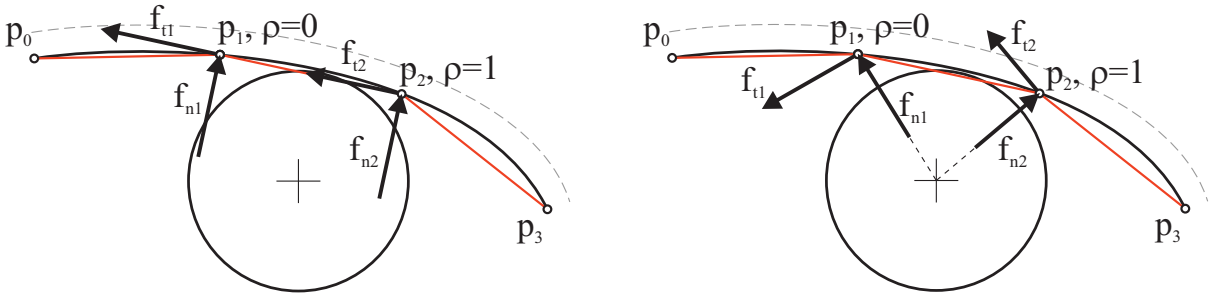
Note that in the Newton method, the tangential force may be inconsistent with the Kuhn-Tucker conditions. However, the `PostNewtonStep` resolves this inconsistency.

[12] see the difference to the `PostNewtonStep`: we use $x_{lastStick}$ here, not the `startOfStep` variant.

[13] see again difference to `PostNewtonStep`!

8.6.15.8 Computation of LHS terms for circle and ANCF cable element

If `activeConnector = True`, contact forces $\mathbf{f}_i$ with $i \in [0, n_{cs}]$ – these are $(n_{cs}+1)$ forces – are applied at the points p_i, and they are computed for every contact segments (i.e., two segments may contribute to contact forces of one point). For every contact computation, first all contact forces at segment points are set to zero. We distinguish two cases SN and PWN. If `useSegmentNormals==True`, we use the SN case, while otherwise the PWN case is used, compare Fig. 8.10.



case: contact, usePointWiseNormals=False

case: contact, usePointWiseNormals=True

Figure 8.10: Choice of normals and tangent vectors for calculation of normal contact forces and tangential (friction) forces; note that the `useSegmentNormals=False` is not appropriate for this setup and would produce highly erroneous forces.

Segment normals (=SN) lead to always good approximations for normal directions, irrespectively of short or extremely long segments as compared to the circle. However, in case of segments that are short as compared to the circle radius, normals computed from the center of the circle to the segment points (=PWN) are more consistent and produce tangents only in circumferential direction, which may improve behavior in some applications. The equations for the two cases read:

CASE SN: use **S**egment **N**ormals
If there is contact in a segment s_i, i.e., gap state $x_{gap} \leq 0$, see Fig. 8.8(right), contact forces $\mathbf{f}_{s_i}$ are computed per segment,

$$\mathbf{f}_{s_i} = f_n \cdot \mathbf{n}_{s_i} + f_t \mathbf{t}_{s_i} \tag{8.374}$$

and added to every force at segment points according to

$$\begin{aligned} \mathbf{f}_i \quad &+= \quad (1-\rho) \cdot \mathbf{f}_{s_i} \\ \mathbf{f}_{i+1} \quad &+= \quad \rho \cdot \mathbf{f}_{s_i} \end{aligned} \tag{8.375}$$

while in case $x_{gap} > 0$ nothing is added.

CASE PWN: use **P**oint **W**ise **N**ormals (at segment points)
If there is contact in a segment s_i, i.e., gap $x_{gap} \leq 0$, see Fig. 8.8(right), intermediate contact forces $\mathbf{f}_i^{l,r}$ are computed per segment point,

$$\mathbf{f}^l = f_n \cdot \mathbf{n}_{l,s_i} + f_t \mathbf{t}_{l,s_i}, \quad \mathbf{f}^r = f_n \cdot \mathbf{n}_{r,s_i} + f_t \mathbf{t}_{r,s_i} \tag{8.376}$$

in which $\mathbf{n}_{l,s_i}$ is the vector from circle center to the left point (i) of the segment s_i, and $\mathbf{n}_{l,s_i}$ to the right point ($i+1$). The tangent vectors are perpendicular to the normals. The forces are then applied to the contact forces $\mathbf{f}_i$ using the parameter ρ, which takes into account the distance of contact to the left or right side of the segment,

$$\begin{aligned} \mathbf{f}_i &\mathrel{+}= (1-\rho)\cdot\mathbf{f}^l \\ \mathbf{f}_{i+1} &\mathrel{+}= \rho\cdot\mathbf{f}^r \end{aligned} \tag{8.377}$$

while in case $x_{gap} > 0$ nothing is added.

The forces $\mathbf{f}_i$ are then applied through the marker to the `ObjectANCFCable2D` element as point loads via a position jacobian (using the according access function), for details see the C++ implementation.

The forces on the circle marker $m0$ are computed as the total sum of all segment contact forces,

$$\mathbf{f}_{m0} = -\sum_{s_i} \mathbf{f}_{s_i} \tag{8.378}$$

and additional torques on the circle's rotation simply follow from

$$\tau_{m0} = -\sum_{s_i} r\cdot f_{t_{s_i}} \ . \tag{8.379}$$

During Newton iterations, the contact forces for segment s_i are considered only, if x_i <= 0. The dataCoordinate x_i is not modified during Newton iterations, but computed during the DiscontinuousIteration, see Fig. 11.5 in the solver description.

If `activeConnector = False`, all contact and friction forces on the cable and the force and torque on the circle's marker are set to zero.

For examples on ObjectContactFrictionCircleCable2D see Relevant Examples and TestModels with weblink:

- `beltDriveALE.py` (Examples/)
- `beltDriveReevingSystem.py` (Examples/)
- `beltDrivesComparison.py` (Examples/)
- `sliderCrank3DwithANCFbeltDrive.py` (Examples/)
- `sliderCrank3DwithANCFbeltDrive2.py` (Examples/)
- `ANCFcontactFrictionTest.py` (TestModels/)
- `ANCFmovingRigidBodyTest.py` (TestModels/)
- `ANCFslidingAndALEjointTest.py` (TestModels/)

8.6.16 ObjectContactSphereSphere

A simple contact connector between two spheres, using various contact models and the option for contact of sphere inside hollow sphere (marker1). The connector implements at least the same functionality as in GeneralContact and is intended for simple setups and for testing, while GeneralContact is much more efficient due to parallelization approaches and efficient contact search.

Authors: Gerstmayr Johannes, Weyrer Sebastian

Additional information for ObjectContactSphereSphere:

- This `Object` has/provides the following types = `Connector`
- Requested `Marker` type = `Position` + `Orientation`
- Requested `Node` type = `GenericData`

The item **ObjectContactSphereSphere** with type = 'ContactSphereSphere' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex	2	[invalid [-1], invalid [-1]]	list of markers representing centers of spheres, used in connector
nodeNumber	NodeIndex		invalid (-1)	node number of a NodeGenericData with numberOfDataCoordinates = 4 dataCoordinates, needed for discontinuous iteration (friction and contact); data variables contain values from last PostNewton iteration: data[0] is the gap, data[1] is the norm of the tangential velocity (and thus contains information if it is stick or slip); data[2] is the impact velocity; data[3] is the plastic overlap of the Edinburgh Adhesive Elasto-Plastic Model, initialized usually with 0 and set back to 0 in case that spheres have been separated.
spheresRadii	Vector2D	2	[-1.,-1.]	Vector containing radius of sphere 0 and radius of sphere 1 [SI:m].
isHollowSphere1	Bool		False	flag, which determines, if sphere attached to marker 1 (radius 1) is a hollow sphere.
dynamicFriction	UReal		0.	dynamic friction coefficient for friction model, see StribeckFunction in exudyn.physics, Section 7.16

frictionProportionalZone	UReal		1e-3	limit velocity [m/s] up to which the friction is proportional to velocity (for regularization / avoid numerical oscillations), see StribeckFunction in exudyn.physics (named regVel there!), Section 7.16
contactStiffness	UReal		0.	normal contact stiffness [SI:N/m] (units in case that $n_{exp} = 1$)
contactDamping	UReal		0.	linear normal contact damping [SI:N/(m s)]; this damping should be used (!=0) if the restitution coefficient is < 1, as it changes its behavior.
contactStiffnessExponent	PReal		1.	exponent in normal contact model [SI:1]
constantPullOffForce	UReal		0.	constant adhesion force [SI:N]; Edinburgh Adhesive Elasto-Plastic Model
contactPlasticityRatio	UReal		0.	ratio of contact stiffness for first loading and unloading/reloading [SI:1]; Edinburgh Adhesive Elasto-Plastic Model; $\lambda_P = 1 - k_c/K2$, which gives the contact stiffness for unloading/reloading $K2 = k_c/(1-\lambda_P)$; set to 0 in order to fully deactivate Edinburgh Adhesive Elasto-Plastic Model model
adhesionCoefficient	UReal		0.	coefficient for adhesion [SI:N/m] (units in case that $n_{adh} = 1$); Edinburgh Adhesive Elasto-Plastic Model; set to 0 to deactivate adhesion model
adhesionExponent	UReal		1.	exponent for adhesion coefficient [SI:1]; Edinburgh Adhesive Elasto-Plastic Model
restitutionCoefficient	PReal		1.	coefficient of restitution [SI:1]; used in particular for impact mechanics; different models available within parameter impactModel; the coefficient must be > 0, but can become arbitrarily small to emulate plastic impact (however very small values may lead to numerical problems)
minimumImpactVelocity	UReal		0.	minimal impact velocity for coefficient of restitution [SI:1]; this value adds a lower bound for impact velocities for calculation of viscous impact force; it can be used to apply a larger damping behavior for low impact velocities (or permanent contact)
impactModel	UInt		0	number of impact model: 0) linear model (only linear damping is used); 1) Hunt-Crossley model; 2) Gonthier/EtAl-Carvalho/Martins mixed model; model 2 is much more accurate regarding the coefficient of restitution, in the full range [0,1] except for 0; NOTE: in all models, the linear contactDamping is added, if not set to zero!

activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectContactSphereSphere			parameters for visualization of item

The item VObjectContactSphereSphere has the following parameters:

Name	type	size	default value	description
show	Bool		False	set true, if item is shown in visualization and false if it is not shown; draws spheres by given radii
color	Float4		[0.7,0.7,0.7,1.]	RGBA connector color; if R==-1, use default color

---

8.6.16.1 DESCRIPTION of ObjectContactSphereSphere:

Information on input parameters:

input parameter	symbol	description see tables above
markerNumbers	$[m0, m1]^T$	
nodeNumber	n_d	
spheresRadii	$[r_0, r_1]^T$	
dynamicFriction	μ_d	
frictionProportionalZone	v_{reg}	
contactStiffness	k_c	
contactDamping	d_c	
contactStiffnessExponent	n_{exp}	
constantPullOffForce	f_{adh}	
contactPlasticityRatio	λ_{P}	
adhesionCoefficient	k_{adh}	
adhesionExponent	n_{adh}	
restitutionCoefficient	e_{res}	
minimumImpactVelocity	$\dot{\delta}_{-,\text{min}}$	
impactModel	m_{impact}	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position		contact center point (also given for positive gap, when no contact occurs)

Displacement		global displacement vector between the two spheres midpoints
DisplacementLocal		1D Vector, containing only gap
Velocity		global relative velocity between the two spheres midpoints
Force		global contact force vector
Director1		contains normalized vector from marker 0 to marker 1
Torque		global torque due to friction on marker 0; to obetain torque on marker 1, multiply the torque with the factor $\frac{r_1+g/2}{r_0+g/2}$

8.6.16.2 Definition of quantities

intermediate variables	**symbol**	**description**
marker m0 position	${}^{0}\mathbf{p}_{m0}$	global position of sphere 0 center as provided by marker m0
marker m0 orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker m0
marker m1 position	${}^{0}\mathbf{p}_{m1}$	global position of sphere 1 center as provided by marker m1
marker m1 orientation	${}^{0,m1}\mathbf{A}$	current rotation matrix provided by marker m1
data coordinates	$\mathbf{x} = [x_0, x_1, x_2, x_3]^{\mathrm{T}}$	hold the current gap (0), the (norm of the) tangential velocity (1), the impact velocity (2), and the plastic deformation (3) of the adhesion model
marker m0 velocity	${}^{0}\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
marker m1 velocity	${}^{0}\mathbf{v}_{m1}$	current global velocity which is provided by marker m1
marker m0 angular velocity	${}^{0}\boldsymbol{\omega}_{m0}$	current angular velocity vector provided by marker m0
marker m1 angular velocity	${}^{0}\boldsymbol{\omega}_{m1}$	current angular velocity vector provided by marker m1

8.6.16.3 Connector Forces

This section outlines the computation of the forces acting on the two spheres when they are in contact with each other. Two types of forces can act on the spheres due to the connector:

- normal force computed according to the chosen impact model m_{impact} and with contact damping if $d_c \neq 0$; this type of force does not create a torque acting on the spheres.
- tangential force due to a regularized friction law to model dry friction between the spheres; this type of force creates a torque acting on the spheres and is computed independently of the chosen

impact model if $\mu_d \neq 0$ is set. Note that in the implemented model, rolling deformations are not considered, i.e. the friction is only a function of the relative tangential velocity between the spheres at the contact point.

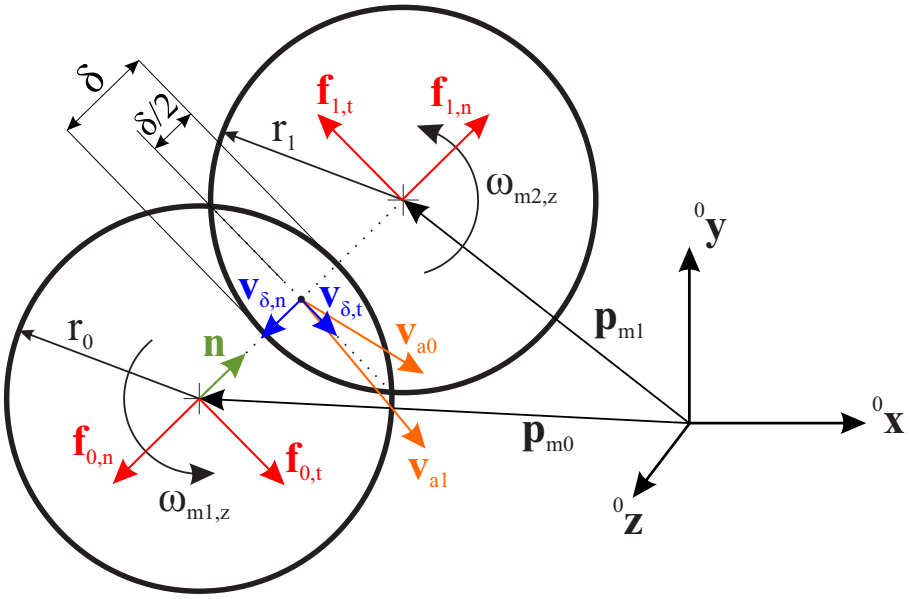


Figure 8.11: Two spheres that are in contact. For illustration, a force due to the overlap δ acting in the direction of $\mathbf{n}$ for marker 1 is shown, as well as a force due to friction acting against the tangential (gap) velocity. The respective opposing forces are imprinted on marker 0.

Calculations reflect the case for outer contact of two spheres using $h_1 = 1$. In case that isHollowSphere1=True, we set $h_1 = -1$ while the remaining formulas remain unchanged.

For the following, the gap g between the two spheres is computed as

$$g = h_1 \| {}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0} \| - (r_0 + h_1 r_1) \tag{8.380}$$

and the overlap δ is the negated gap: $\delta = -g$. In the contact case, the overlap δ is positive. The normal vector ${}^0\mathbf{n}$ points from marker 0 to the contact point,

$${}^0\mathbf{n} = h_1 \frac{{}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0}}{\| {}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0} \|} . \tag{8.381}$$

which in case of sphere-sphere contact, ${}^0\mathbf{n}$ points from marker 0 to marker 1, and in case of sphere-hollowsphere contact, ${}^0\mathbf{n}$ points from marker 1 to marker 0.

The scalar normal (gap) velocity $v_{\delta,\mathrm{n}}$ is computed with the velocities ${}^0\mathbf{v}_{m0} = {}^0\dot{\mathbf{p}}_{m0}$ and ${}^0\mathbf{v}_{m1} = {}^0\dot{\mathbf{p}}_{m1}$

$$v_{\delta,\mathrm{n}} = \left({}^0\mathbf{v}_{m1} - {}^0\mathbf{v}_{m0} \right) \cdot {}^0\mathbf{n} \tag{8.382}$$

and the tangential (gap) velocity ${}^0\mathbf{v}_{\delta,\mathrm{t}}$ at the contact point, that is needed for the friction model, reads

$${}^0\mathbf{v}_{\delta,\mathrm{t}} = \left({}^0\mathbf{v}_{a1} - {}^0\mathbf{v}_{a0} \right) - v_{\delta,\mathrm{n}} \cdot {}^0\mathbf{n} , \qquad v_{\mathrm{rel}} = \| {}^0\mathbf{v}_{\delta,\mathrm{t}} \| . \tag{8.383}$$

To take the angular velocity of the spheres into account, the velocities ${}^0\mathbf{v}_{a0}$ and ${}^0\mathbf{v}_{a1}$ at the contact point are computed using Euler's theorem for kinematics:

$$ {}^0\mathbf{v}_{a0} = {}^0\mathbf{v}_{m0} + {}^0\boldsymbol{\omega}_{m0} \times \left({}^0\mathbf{n} \cdot \left(r_0 - \frac{\delta}{2} \right) \right), \qquad {}^0\mathbf{v}_{a1} = {}^0\mathbf{v}_{m1} + h_1 \, {}^0\boldsymbol{\omega}_{m1} \times \left(- {}^0\mathbf{n} \cdot \left(r_1 - h_1 \frac{\delta}{2} \right) \right) . \tag{8.384} $$

The normal force acting on marker 1 is generally written as

$$ {}^0\mathbf{f}_{1,\mathrm{n}} = \underbrace{(f_c + f_d)}_{f_{1,\mathrm{n}}} \cdot {}^0\mathbf{n} \ , \tag{8.385} $$

where f_c is the elastic and f_d the damping part. The damping f_d is always computed the same, independent of the chosen impact model:

$$ f_d = -d_c v_{\delta,\mathrm{n}} \ . \tag{8.386} $$

The negative sign is because of the damping acting against the gap velocity: in the case of a positive normal (gap) velocity, the damping acts against ${}^0\mathbf{n}$ for marker 1. The elastic force f_c is computed depending on the chosen impact model.

CASE $m_{\mathrm{impact}} = 0$: the Adhesive Elasto-Plastic model described in [42] is used. This model captures the key bulk behavior of cohesive powders and granular soils. For the impact model, the plastic overlap δ_p is needed. It is computed with

$$ \delta_p = \lambda_{\mathrm{p}}^{\frac{1}{n_{\mathrm{exp}}}} \delta \ . \tag{8.387} $$

The Adhesive Elasto-Plastic model distinguishes three different cases, modeling the loading and unloading behavior of the spheres:

$$ f_c = \begin{cases} -f_{\mathrm{adh}} + k_c \delta^{n_{\mathrm{exp}}} & \text{if } k_2 \left(\delta^{n_{\mathrm{exp}}} - \delta_p^{n_{\mathrm{exp}}} \right) \geq k_c \delta^{n_{\mathrm{exp}}} \\ -f_{\mathrm{adh}} + k_2 \left(\delta^{n_{\mathrm{exp}}} - \delta_p^{n_{\mathrm{exp}}} \right) & \text{if } k_c \delta^{n_{\mathrm{exp}}} > k_2 \left(\delta^{n_{\mathrm{exp}}} - \delta_p^{n_{\mathrm{exp}}} \right) > -k_{\mathrm{adh}} \delta^{n_{\mathrm{adh}}} \\ -f_{\mathrm{adh}} - k_{\mathrm{adh}} \delta^{n_{\mathrm{adh}}} & \text{if } -k_{\mathrm{adh}} \delta^{n_{\mathrm{adh}}} > k_2 \left(\delta^{n_{\mathrm{exp}}} - \delta_p^{n_{\mathrm{exp}}} \right) \end{cases} . \tag{8.388} $$

Note that k_2 is computed with $k_2 = k_c/(1 - \lambda_{\mathrm{P}})$. The terms with the stiffness k_c and k_2 have a positive sign, since they act in the direction of ${}^0\mathbf{n}$ for marker 1. The constant adhesion force f_{adh} and the stiffness k_{adh} act against ${}^0\mathbf{n}$, which corresponds to a force sticking the spheres together.

CASE $m_{\mathrm{impact}} = 1$: the restitution model proposed by Hunt and Crossley in [35] is used to simulate the energy loss of the spheres during contact:

$$ f_c = k_c \delta^{n_{\mathrm{exp}}} + \lambda \delta^{n_{\mathrm{exp}}} v_{\delta,\mathrm{n}} \tag{8.389} $$

with

$$ \lambda = \frac{k_c}{\dot{\delta}_-} \frac{3}{2} (e_{\mathrm{res}} - 1) \ . \tag{8.390} $$

The restitution coefficient e_{res} describes the ration of the normal (gap) velocity before and after the impact of the spheres. In the case of $e_{\mathrm{res}} < 1$, the impact has a plastic portion, resulting in a force

acting against ${}^0\mathbf{n}$ for marker 1, which is why λ must be negative in that case. $\dot{\delta}_-$ is the initial relative velocity, which is either the minimum impact velocity or the normal (negated gap) velocity:

$$\dot{\delta}_- = \max\left(\dot{\delta}_{-,\min}; -v_{\delta,\mathrm{n}}\right) \tag{8.391}$$

Note that the Hunt-Crossley restitution is valid for a very small energy loss ($e_{\mathrm{res}} \approx 1$) [6].

CASE $m_{\mathrm{impact}} = 2$: a generalization of the Hunt-Crossley restitution proposed by Carvalho and Martins in [6] is used for $e_{\mathrm{res}} > \frac{1}{3}$ and a model proposed by Gonthier et al. in [27] is used for impacts with a high plastic proportion, $e_{\mathrm{res}} < \frac{1}{3}$. Note that the two models are identical at $e_{\mathrm{res}} = \frac{1}{3}$. λ is therefore computed as follows:

$$\lambda = \begin{cases} \frac{k_c}{\dot{\delta}_-}\frac{3}{2}(e_{\mathrm{res}}-1)\frac{11-e_{\mathrm{res}}}{1+9e_{\mathrm{res}}} & \text{if } e_{\mathrm{res}} > \frac{1}{3} \\ \frac{k_c}{\dot{\delta}_-}\frac{e_{\mathrm{rep}}^2-1}{e_{\mathrm{rep}}} & \text{if } e_{\mathrm{res}} > 0 \end{cases} . \tag{8.392}$$

The tangential force acting on marker 1 due to the friction model acts against the tangential velocity $\mathbf{v}_{\delta,\mathrm{t}}$, see the computation of $\mathbf{v}_{\delta,\mathrm{t}}$ in Equation (8.383). Thus, the tangential force for marker 1 is computed as

$${}^0\mathbf{f}_{1,\mathrm{t}} = -{}^0\mathbf{v}_{\delta,\mathrm{t}} \cdot \begin{cases} \frac{\mu_d f_{1,\mathrm{n}}}{v_{reg}} & \text{if } v_{rel} < v_{reg} \\ \frac{\mu_d f_{1,\mathrm{n}}}{v_{rel}} & \text{else} \end{cases} . \tag{8.393}$$

Note that the case distinction above is made to ensure that for very small relative velocities the friction force does not become implausibly high. Taken together, the force acting on marker 1 due to the connector is computed as

$${}^0\mathbf{f}_{m1} = {}^0\mathbf{f}_{1,\mathrm{n}} + {}^0\mathbf{f}_{1,\mathrm{t}} , \tag{8.394}$$

the force acting on marker 0 is ${}^0\mathbf{f}_{m0} = -{}^0\mathbf{f}_{m1}$. The global torque ${}^0\boldsymbol{\tau}_{m1}$ acting on marker 1 due to the connector is computed as

$${}^0\boldsymbol{\tau}_{m1} = -h_1\,{}^0\mathbf{n}\left(r_1 - h_1\frac{1}{2}\delta\right) \times {}^0\mathbf{f}_{m1} , \tag{8.395}$$

and on marker 0 as

$${}^0\boldsymbol{\tau}_{m0} = {}^0\mathbf{n}\left(r_0 - \frac{1}{2}\delta\right) \times {}^0\mathbf{f}_{m0} = {}^0\mathbf{n}\left(r_0 - \frac{1}{2}\delta\right) \times \left(-{}^0\mathbf{f}_{m1}\right) . \tag{8.396}$$

It can be seen that the torque due to the connector is the same for both spheres, if $r_0 = r_1$ applies.

For examples on ObjectContactSphereSphere see Relevant Examples and TestModels with weblink:

- `newtonsCradle.py` (Examples/)
- `rollerBearningModel.py` (Examples/)
- `contactSphereSphereTest.py` (TestModels/)
- `contactSphereSphereTestEAPM.py` (TestModels/)
- `sphereTriangleTest2.py` (TestModels/)

8.6.17 ObjectContactSphereTorus

[UNDER DEVELOPMENT] A simple contact connector between a sphere (marker0) and a torus (marker1). The sphere is assumed to be placed inside of the torus (outer contact of sphere with torus currently not implemented!).

Author: Gerstmayr Johannes

Additional information for ObjectContactSphereTorus:

- This `Object` has/provides the following types = `Connector`
- Requested `Marker` type = `Position` + `Orientation`
- Requested `Node` type = `GenericData`

The item **ObjectContactSphereTorus** with type = 'ContactSphereTorus' has the following parameters:

Name	type	size	default value	description
name	String		''	constraints's unique name
markerNumbers	ArrayMarkerIndex	2	[invalid [-1], invalid [-1]]	list of markers representing centers of sphere (marker 0) and center of torus (marker 1)
nodeNumber	NodeIndex		invalid (-1)	node number of a NodeGenericData with numberOfDataCoordinates = 4 dataCoordinates, needed for discontinuous iteration (friction and contact); data variables contain values from last PostNewton iteration: data[0] is the gap, data[1] is the norm of the tangential velocity (and thus contains information if it is stick or slip); data[2] is the impact velocity; data[3] is unused.
radiusSphere	PReal		0.	radius of sphere [SI:m]
torusMajorRadius	PReal		0.	major radius of torus [SI:m], representing center of rotated circle
torusMinorRadius	PReal		0.	minor radius of torus [SI:m], representing radius of circle of ring
torusAxis	Vector3D	3	[0,0,0]	Vector containing rotation axis of torus; must be a unit vector.
dynamicFriction	UReal		0.	dynamic friction coefficient for friction model, see StribeckFunction in exudyn.physics, Section 7.16

frictionProportionalZone	UReal		1e-3	limit velocity [m/s] up to which the friction is proportional to velocity (for regularization / avoid numerical oscillations), see StribeckFunction in exudyn.physics (named regVel there!), Section 7.16
contactStiffness	UReal		0.	normal contact stiffness [SI:N/m] (units in case that $n_{exp} = 1$)
contactDamping	UReal		0.	linear normal contact damping [SI:N/(m s)]; this damping should be used (!=0) if the restitution coefficient is < 1, as it changes its behavior.
contactStiffnessExponent	PReal		1.	exponent in normal contact model [SI:1]
restitutionCoefficient	PReal		1.	coefficient of restitution [SI:1]; used in particular for impact mechanics; different models available within parameter impactModel; the coefficient must be > 0, but can become arbitrarily small to emulate plastic impact (however very small values may lead to numerical problems)
minimumImpactVelocity	UReal		0.	minimal impact velocity for coefficient of restitution [SI:1]; this value adds a lower bound for impact velocities for calculation of viscous impact force; it can be used to apply a larger damping behavior for low impact velocities (or permanent contact)
impactModel	UInt		0	number of impact model: 0) linear model (only linear damping is used); 1) Hunt-Crossley model; 2) Gonthier/EtAl-Carvalho/Martins mixed model; model 2 is much more accurate regarding the coefficient of restitution, in the full range [0,1] except for 0; NOTE: in all models, the linear contactDamping is added, if not set to zero!
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectContactSphereTorus			parameters for visualization of item

The item VObjectContactSphereTorus has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		False	set true, if item is shown in visualization and false if it is not shown; draws spheres by given radii
color	Float4		[0.7,0.7,0.7,1.]	RGBA connector color; if R==-1, use default color

8.6.17.1 DESCRIPTION of ObjectContactSphereTorus:

Information on input parameters:

input parameter	symbol	description see tables above
markerNumbers	$[m0, m1]^{\mathrm{T}}$	
nodeNumber	n_d	
radiusSphere	r_S	
torusMajorRadius	r_M	
torusMinorRadius	r_m	
torusAxis	$\mathbf{v}_{axis}$	
dynamicFriction	μ_d	
frictionProportionalZone	v_{reg}	
contactStiffness	k_c	
contactDamping	d_c	
contactStiffnessExponent	n_{exp}	
restitutionCoefficient	e_{res}	
minimumImpactVelocity	$\dot{\delta}_{-,\mathrm{min}}$	
impactModel	m_{impact}	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position		contact center point (also given for positive gap, when no contact occurs)
Displacement		global displacement vector between the two spheres midpoints
DisplacementLocal		1D Vector, containing only gap
Director1		normalized vector from marker 0 to marker 1
Director2		the normalized vector from marker 0 to marker 1 projected into the plane of the torus major circle
Director3		normalized vector from the projected point on the major circle (center of the minor circle) to marker 1, being in direction of the contact and normal to the surface
Force		global contact force vector
Torque		global torque due to friction on marker 0

8.6.17.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^{0}\mathbf{p}_{m0}$	global position of torus 0 center as provided by marker m0
marker m0 orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker m0
marker m1 position	${}^{0}\mathbf{p}_{m1}$	global position of sphere 1 center as provided by marker m1
marker m1 orientation	${}^{0,m1}\mathbf{A}$	current rotation matrix provided by marker m1
data coordinates	$\mathbf{x} = [x_0, x_1, x_2, x_3]^T$	hold the current gap (0), the (norm of the) tangential velocity (1), the impact velocity (2), and (3) which is undefined
marker m0 velocity	${}^{0}\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
marker m1 velocity	${}^{0}\mathbf{v}_{m1}$	current global velocity which is provided by marker m1
marker m0 angular velocity	${}^{0}\boldsymbol{\omega}_{m0}$	current angular velocity vector provided by marker m0
marker m1 angular velocity	${}^{0}\boldsymbol{\omega}_{m1}$	current angular velocity vector provided by marker m1

8.6.17.3 Connector Forces

TBD

For examples on ObjectContactSphereTorus see Relevant Examples and TestModels with weblink:

- `ballBearningModel.py` (Examples/)

8.6.18 ObjectContactSphereTriangle

[UNDER DEVELOPMENT] A simple contact connector between a sphere (marker0) and a triangle (marker1). Penalty-based contact is computed from penetration of the sphere with the triangle, including contact with edges if desired.

Author: Gerstmayr Johannes

Additional information for ObjectContactSphereTriangle:

- This `Object` has/provides the following types = `Connector`
- Requested `Marker` type = `Position` + `Orientation`
- Requested `Node` type = `GenericData`

The item **ObjectContactSphereTriangle** with type = 'ContactSphereTriangle' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex	2	[invalid [-1], invalid [-1]]	list of markers representing the center of the sphere (marker 0) and the reference point of the triangle (marker 1), where triangle nodal positions are defined in the local coordinates of marker 1.
nodeNumber	NodeIndex		invalid (-1)	node number of a NodeGenericData with numberOfDataCoordinates = 4 dataCoordinates, needed for discontinuous iteration (friction and contact); data variables contain values from last PostNewton iteration: data[0] is the gap, data[1] is the norm of the tangential velocity (and thus contains information if it is stick or slip); data[2] is the impact velocity; data[3] is unused.
radiusSphere	PReal		0.	radius of sphere [SI:m]
trianglePoints	Vector3DList		[]	triangle points, defined in marker 1 local coordinates
includeEdges	UInt		7	Binary flag, where 1 defines contact with edges 0, 2 with edge 1 and 4 with edge 2; 7 means that contact with all edges is included; edge 0 is the edge between node 0 and node 1

dynamicFriction	UReal		0.	dynamic friction coefficient for friction model, see StribeckFunction in exudyn.physics, Section 7.16
frictionProportionalZone	UReal		1e-3	limit velocity [m/s] up to which the friction is proportional to velocity (for regularization / avoid numerical oscillations), see StribeckFunction in exudyn.physics (named regVel there!), Section 7.16
contactStiffness	UReal		0.	normal contact stiffness [SI:N/m] (units in case that $n_{exp} = 1$)
contactDamping	UReal		0.	linear normal contact damping [SI:N/(m s)]; this damping should be used (!=0) if the restitution coefficient is < 1, as it changes its behavior.
contactStiffnessExponent	PReal		1.	exponent in normal contact model [SI:1]
restitutionCoefficient	PReal		1.	coefficient of restitution [SI:1]; used in particular for impact mechanics; different models available within parameter impactModel; the coefficient must be > 0, but can become arbitrarily small to emulate plastic impact (however very small values may lead to numerical problems)
minimumImpactVelocity	UReal		0.	minimal impact velocity for coefficient of restitution [SI:1]; this value adds a lower bound for impact velocities for calculation of viscous impact force; it can be used to apply a larger damping behavior for low impact velocities (or permanent contact)
impactModel	UInt		0	number of impact model: 0) linear model (only linear damping is used); 1) Hunt-Crossley model; 2) Gonthier/EtAl-Carvalho/Martins mixed model; model 2 is much more accurate regarding the coefficient of restitution, in the full range [0,1] except for 0; NOTE: in all models, the linear contactDamping is added, if not set to zero!
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectContactSphereTriangle			parameters for visualization of item

The item VObjectContactSphereTriangle has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		False	set true, if item is shown in visualization and false if it is not shown; draws spheres by given radii

color	Float4		[0.7,0.7,0.7,1.]	RGBA connector color; if R==-1, use default color

---

8.6.18.1 DESCRIPTION of ObjectContactSphereTriangle:

Information on input parameters:

input parameter	symbol	description see tables above
markerNumbers	$[m0, m1]^{\mathrm{T}}$	
nodeNumber	n_d	
radiusSphere	r_S	
trianglePoints	$[{}^{m_1}\mathbf{p}_0, {}^{m_1}\mathbf{p}_1, {}^{m_1}\mathbf{p}_2]$	
dynamicFriction	μ_d	
frictionProportionalZone	v_{reg}	
contactStiffness	k_c	
contactDamping	d_c	
contactStiffnessExponent	n_{exp}	
restitutionCoefficient	e_{res}	
minimumImpactVelocity	$\dot{\delta}_{-,\text{min}}$	
impactModel	m_{impact}	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position		contact center point (also given for positive gap, when no contact occurs)
Displacement		global displacement vector between the two spheres midpoints
DisplacementLocal		1D Vector, containing only gap
Director1		normalized vector from sphere midpoint (marker 0) to triangle contact point
Force		global contact force vector
Torque		global torque due to friction on marker 0

8.6.18.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^0\mathbf{p}_{m0}$	global position of torus 0 center as provided by marker m0

marker m0 orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker m0
marker m1 position	${}^{0}\mathbf{p}_{m1}$	global position of sphere 1 center as provided by marker m1
marker m1 orientation	${}^{0,m1}\mathbf{A}$	current rotation matrix provided by marker m1
data coordinates	$\mathbf{x} = [x_0, x_1, x_2, x_3]^T$	hold the current gap (0), the (norm of the) tangential velocity (1), the impact velocity (2), and (3) which is undefined
marker m0 velocity	${}^{0}\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
marker m1 velocity	${}^{0}\mathbf{v}_{m1}$	current global velocity which is provided by marker m1
marker m0 angular velocity	${}^{0}\boldsymbol{\omega}_{m0}$	current angular velocity vector provided by marker m0
marker m1 angular velocity	${}^{0}\boldsymbol{\omega}_{m1}$	current angular velocity vector provided by marker m1

8.6.18.3 Connector Forces

TBD

For examples on ObjectContactSphereTriangle see Relevant Examples and TestModels with weblink:

- `sphereTriangleTest.py` (TestModels/)
- `sphereTriangleTest2.py` (TestModels/)

8.6.19 ObjectContactCurveCircles

A contact model between a curve defined by piecewise segments and a set of circles. The 2D curve may corotate in 3D with the underlying marker and also defines the plane of action for the circles. [REQUIRES FURTHER TESTING; friction not yet available]

Additional information for ObjectContactCurveCircles:

- This `Object` has/provides the following types = `Connector`
- Requested `Marker` type = `Position` + `Orientation`
- Requested `Node` type = `GenericData`
- **Short name** for Python = `CamFollowerContactPlanar`
- **Short name** for Python visualization object = `VCamFollowerContactPlanar`

The item **ObjectContactCurveCircles** with type = 'ContactCurveCircles' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex	2	[invalid [-1], invalid [-1]]	list of $n_c + 1$ markers; marker $m0$ represents the marker carrying the curve; all other markers represent centers of n_c circles, used in connector
nodeNumber	NodeIndex		invalid (-1)	node number of a NodeGenericData with nDataVariablesPerSegment dataCoordinates per segment, needed for discontinuous iteration; data variables contain values from last PostNewton iteration: data[0+3*i] is the circle number, data[1+3*i] is the gap, data[2+3*i] is the tangential velocity (and thus contains information if it is stick or slip)
circlesRadii	NumpyVector		[]	Vector containing radii of n_c circles [SI:m]; number according to size of markerNumbers-1

segmentsData	PyMatrixContainer		PyMatrixContainer[]	matrix containing a set of two planar point coordinates in each row, representing segments attached to marker $m0$ and undergoing contact with the circles; for segment $s0$ row 0 reads [$p_{0x,s0}$, $p_{0y,s0}$, $p_{1x,s0}$, $p_{1y,s0}$]; note that the segments must be ordered such that going from $\mathbf{p}_0$ to $\mathbf{p}_1$, the exterior lies on the right (positive) side. MatrixContainer has to be provided in dense mode!
polynomialData	PyMatrixContainer		PyMatrixContainer[]	matrix containing coefficients for special polynomial enhancements of the linear segments; each row contains coefficients for polynomials for the according segment, prescribing slopes at beginning and end of segment as well as curvature at beginning and end of segment; slopes and curvatures are defined in a local x/y coordinate system where x is the segment axis (start: x=0; x-axis points towards end point) and the segment normal is in y-direction; MatrixContainer has to be provided in dense mode!
rotationMarker0	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	local rotation matrix for marker 0; used to rotate marker coordinates such that the curve lies in the $x-y$-plane
dynamicFriction	UReal		0.	dynamic friction coefficient for friction model, see StribeckFunction in exudyn.physics, Section 7.16
frictionProportionalZone	UReal		1e-3	limit velocity [m/s] up to which the friction is proportional to velocity (for regularization / avoid numerical oscillations), see StribeckFunction in exudyn.physics (named regVel there!), Section 7.16
contactStiffness	Real		0.	normal contact stiffness [SI:N/(m*m)]
contactDamping	Real		0.	linear normal contact damping [SI:N/(m s)]; this damping is a simplification of real contact dissipation and should be used with care.
contactModel	UInt		0	number of contact model: 0) linear model for stiffness and damping, only proportional to penetration; contact force is computed from $l_{\text{seg}}\,(p \cdot k_c + \dot{p} \cdot d_c)$ as long as $p > 0$; while this is numerically more stable, it gives jumps in forces when sliding over contact geometry 1) contact force proportional to integral over penetration area of circle with segments, giving a smoother contact force when sliding over geometry;

activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
gapPerSegment	NumpyVector		[]	temporary vector for computed gap
gapPerSegment_t	NumpyVector		[]	temporary vector for computed gap velocity
segmentsForceLocalX	NumpyVector		[]	temporary vector for contact force per segment in local X-direction
segmentsForceLocalY	NumpyVector		[]	temporary vector for contact force per segment in local Y-direction
visualization	VObjectContactCurveCircles			parameters for visualization of item

The item VObjectContactCurveCircles has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown; draws curve and circles with given radii; uses visualizationSettings circleTiling for circles and circleTiling/2 for tiling of non-straight segments
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

---

8.6.19.1 DESCRIPTION of ObjectContactCurveCircles:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
markerNumbers	$[m0, m_{c0}, m_{c1}, \ldots]^\mathrm{T}$	
nodeNumber	n_d	
circlesRadii	$[r_{c0}, r_{c1}, \ldots]^\mathrm{T} \in \mathbb{R}^{n_c}$	
segmentsData	$\mathbf{D} \in \mathbb{R}^{n_s \times 4}$	
polynomialData	$\mathbf{P} \in \mathbb{R}^{n_s \times n_p}$	
dynamicFriction	μ_d	
frictionProportionalZone	v_{reg}	
contactStiffness	k_c	
contactDamping	d_c	
contactModel	m_{contact}	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
DisplacementLocal		vector containing the minimum distance to segments per circle midpoint (< 0 in case of contact, and -1 if not computed: if not in according vicinity in search tree)
VelocityLocal		vector containing relative (normal) velocity per circle midpoint (or NaN if not computed)
ForceLocal		pairs of normal and tangential forces per circle or (Nan,Nan) if not computed

8.6.19.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^{0}\mathbf{p}_{m0}$	global position of sphere 0 center as provided by marker m0
marker m0 orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker m0
marker m0 velocity	${}^{0}\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
marker m0 angular velocity	${}^{0}\boldsymbol{\omega}_{m0}$	current angular velocity vector provided by marker m0
data coordinates	$\mathbf{x} = [x_0, x_1, \ldots]^T$	data coordinates per number of circle markers

8.6.19.3 Geometric relations

tbd

For examples on ObjectContactCurveCircles see Relevant Examples and TestModels with weblink:

- `camFollowerExample.py` (Examples/)
- `chainDriveExample.py` (Examples/)
- `contactCurvePolynomial.py` (Examples/)
- `contactCurveWithLongCurve.py` (Examples/)
- `contactCurveExample.py` (TestModels/)

8.7 Objects (Constraint)

A Constraint is a special Object and Connector, which links two or more markers. A Constraint leads to algebraic equations, which exactly fulfill special constraints on the kinematic behavior of the multibody syste, such as a constraint on a coordinate or a distance constraint.

8.7.1 ObjectConnectorDistance

Connector which enforces constant or prescribed distance between two bodies/nodes.

Additional information for ObjectConnectorDistance:

- This `Object` has/provides the following types = `Connector`, `Constraint`
- Requested `Marker` type = `Position`
- **Short name** for Python = `DistanceConstraint`
- **Short name** for Python visualization object = `VDistanceConstraint`

The item **ObjectConnectorDistance** with type = 'ConnectorDistance' has the following parameters:

Name	type	size	default value	description
name	String		''	constraints's unique name
markerNumbers	ArrayMarkerIndex		[invalid [-1], invalid [-1]]	list of markers used in connector
distance	PReal		0.	prescribed distance [SI:m] of the used markers; must by greater than zero
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectConnectorDistance			parameters for visualization of item

The item VObjectConnectorDistance has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = link size; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

8.7.1.1 DESCRIPTION of ObjectConnectorDistance:

Information on input parameters:

input parameter	symbol	description see tables above
markerNumbers	$[m0, m1]^{\mathrm{T}}$	
distance	d_0	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Displacement	${}^0\Delta\mathbf{p}$	relative displacement in global coordinates
Velocity	${}^0\Delta\mathbf{v}$	relative translational velocity in global coordinates
Distance	$\lvert {}^0\Delta\mathbf{p} \rvert$	distance between markers (should stay constant; shows constraint deviation)
Force	λ_0	joint force (=scalar Lagrange multiplier)

8.7.1.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker m0
marker m1 position	${}^0\mathbf{p}_{m1}$	accordingly
marker m0 velocity	${}^0\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
marker m1 velocity	${}^0\mathbf{v}_{m1}$	accordingly
relative displacement	${}^0\Delta\mathbf{p}$	${}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0}$
relative velocity	${}^0\Delta\mathbf{v}$	${}^0\mathbf{v}_{m1} - {}^0\mathbf{v}_{m0}$
algebraicVariable	λ_0	Lagrange multiplier = force in constraint

8.7.1.3 Connector forces constraint equations

If `activeConnector = True`, the index 3 algebraic equation reads

$$\left|{}^0\Delta\mathbf{p}\right| - d_0 = 0 \tag{8.397}$$

Due to the fact that the force direction is given by

$$\frac{1}{\lvert {}^0\Delta\mathbf{p} \rvert}\, {}^0\Delta\mathbf{p} \ , \tag{8.398}$$

the prescribed distance d_0 may not be zero. This would, otherwise, result in a change of the number of constraints. The index 2 (velocity level) algebraic equation reads

$$\left(\frac{{}^0\Delta\mathbf{p}}{\left|{}^0\Delta\mathbf{p}\right|}\right)^{\mathrm{T}} \Delta\mathbf{v} = 0 \tag{8.399}$$

if `activeConnector = False`, the algebraic equation reads

$$\lambda_0 = 0 \tag{8.400}$$

8.7.1.4 MINI EXAMPLE for ObjectConnectorDistance

```
#example with 1m pendulum, 50kg under gravity
nMass = mbs.AddNode(NodePoint2D(referenceCoordinates=[1,0]))
oMass = mbs.AddObject(MassPoint2D(physicsMass = 50, nodeNumber = nMass))

mMass = mbs.AddMarker(MarkerNodePosition(nodeNumber=nMass))
mGround = mbs.AddMarker(MarkerBodyPosition(bodyNumber=oGround, localPosition = [0,0,0]))
oDistance = mbs.AddObject(DistanceConstraint(markerNumbers = [mGround, mMass], distance
= 1))

mbs.AddLoad(Force(markerNumber = mMass, loadVector = [0, -50*9.81, 0]))

#assemble and solve system for default parameters
mbs.Assemble()

sims=exu.SimulationSettings()
sims.timeIntegration.generalizedAlpha.spectralRadius=0.7
mbs.SolveDynamic(sims)

#check result at default integration time
exudynTestGlobals.testResult = mbs.GetNodeOutput(nMass, exu.OutputVariableType.Position)
[0]
```

For examples on ObjectConnectorDistance see Relevant Examples and TestModels with weblink:

- chatGPTupdate.py (Examples/)
- chatGPTupdate2.py (Examples/)
- newtonsCradle.py (Examples/)
- HydraulicActuatorStaticInitialization.py (Examples/)
- pendulum2Dconstraint.py (Examples/)
- pendulumIftommBenchmark.py (Examples/)
- fourBarMechanismTest.py (TestModels/)
- createFunctionsTest.py (TestModels/)
- deleteItemsTest.py (TestModels/)
- mainSystemExtensionsTests.py (TestModels/)

- taskmanagerTest.py (TestModels/)
- coordinateVectorConstraint.py (TestModels/)
- coordinateVectorConstraintGenericODE2.py (TestModels/)
- modelUnitTests.py (TestModels/)
- PARTS_ATEs_moving.py (TestModels/)
- ...

8.7.2 ObjectConnectorCoordinate

A coordinate constraint which constrains two (scalar) coordinates of Marker[Node|Body]Coordinates attached to nodes or bodies. The constraint acts directly on coordinates, but does not include reference values, e.g., of nodal values. This constraint is computationally efficient and should be used to constrain nodal coordinates.

Additional information for ObjectConnectorCoordinate:

- This `Object` has/provides the following types = `Connector`, `Constraint`
- Requested `Marker` type = `Coordinate`
- **Short name** for Python = `CoordinateConstraint`
- **Short name** for Python visualization object = `VCoordinateConstraint`

The item **ObjectConnectorCoordinate** with type = 'ConnectorCoordinate' has the following parameters:

Name	type	size	default value	description
name	String		''	constraints's unique name
markerNumbers	ArrayMarkerIndex		[invalid [-1], invalid [-1]]	list of markers used in connector
offset	Real		0.	An offset between the two values
factorValue1	Real		1.	An additional factor multiplied with value1 used in algebraic equation
velocityLevel	Bool		False	If true: connector constrains velocities (only works for ODE2 coordinates!); offset is used between velocities; in this case, the offsetUserFunction_t is considered and offsetUserFunction is ignored
offsetUserFunction	PyFunctionMbsScalarIndexScalar		0	A Python function which defines the time-dependent offset; see description below
offsetUserFunction_t	PyFunctionMbsScalarIndexScalar		0	time derivative of offsetUserFunction; needed for velocity level constraints; see description below
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectConnectorCoordinate			parameters for visualization of item

The item VObjectConnectorCoordinate has the following parameters:

Name	type	size	default value	description

show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = link size; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

---

8.7.2.1 DESCRIPTION of ObjectConnectorCoordinate:

Information on input parameters:

input parameter	symbol	description see tables above
markerNumbers	$[m0, m1]^T$	
offset	l_{off}	
factorValue1	k_{m1}	
offsetUserFunction	$UF \in \mathbb{R}$	
offsetUserFunction_t	$UF_t \in \mathbb{R}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Displacement	Δq	relative scalar displacement of marker coordinates, not including factorValue1
Velocity	Δv	difference of scalar marker velocity coordinates, not including factorValue1
ConstraintEquation	$\mathbf{c}$	(residuum of) constraint equation
Force	λ_0	scalar constraint force (Lagrange multiplier)

8.7.2.2 Definition of quantities

intermediate variables	symbol	description
marker m0 coordinate	q_{m0}	current displacement coordinate which is provided by marker m0; does NOT include reference coordinate!
marker m1 coordinate	q_{m1}	
marker m0 velocity coordinate	v_{m0}	current velocity coordinate which is provided by marker m0
marker m1 velocity coordinate	v_{m1}	

difference of coordinates	$\Delta q = q_{m1} - q_{m0}$	Displacement between marker m0 to marker m1 coordinates (does NOT include reference coordinates)
difference of velocity coordinates	$\Delta v = v_{m1} - v_{m0}$	

8.7.2.3 Connector constraint equations

If `activeConnector = True`, the index 3 algebraic equation reads

$$\mathbf{c}(q_{m0}, q_{m1}) = k_{m1} \cdot q_{m1} - q_{m0} - l_{\text{off}} = 0 \tag{8.401}$$

If the offsetUserFunction UF is defined, **c** instead becomes (t is current time)

$$\mathbf{c}(q_{m0}, q_{m1}) = k_{m1} \cdot q_{m1} - q_{m0} - \text{UF}(mbs, t, i_N, l_{\text{off}}) = 0 \tag{8.402}$$

The `activeConnector = True`, index 2 (velocity level) algebraic equation reads

$$\dot{\mathbf{c}}(\dot{q}_{m0}, \dot{q}_{m1}) = k_{m1} \cdot \dot{q}_{m1} - \dot{q}_{m0} - d = 0 \tag{8.403}$$

The factor d in velocity level equations is zero, except if parameters.velocityLevel = True, then $d = l_{\text{off}}$. If velocity level constraints are active and the velocity level offsetUserFunction_t UF_t is defined, $\dot{\mathbf{c}}$ instead becomes (t is current time)

$$\dot{\mathbf{c}}(\dot{q}_{m0}, \dot{q}_{m1}) = k_{m1} \cdot \dot{q}_{m1} - \dot{q}_{m0} - \text{UF}_t(mbs, t, i_N, l_{\text{off}}) = 0 \tag{8.404}$$

and `iN` represents the itemNumber (=objectNumber). Note that the index 2 equations are used, if the solver uses index 2 formulation OR if the flag parameters.velocityLevel = True (or both). The user functions include dependency on time t, but this time dependency is not respected in the computation of initial accelerations. Therefore, it is recommended that UF and UF_t does not include initial accelerations.

If `activeConnector = False`, the (index 1) algebraic equation reads for ALL cases:

$$\mathbf{c}(\lambda_0) = \lambda_0 = 0 \tag{8.405}$$

Userfunction: `offsetUserFunction(mbs, t, itemNumber, lOffset)`

A user function, which computes scalar offset for the coordinate constraint, e.g., in order to move a node on a prescribed trajectory. It is NECESSARY to use sufficiently smooth functions, having **initial offsets** consistent with **initial configuration** of bodies, either zero or compatible initial offset-velocity, and no initial accelerations. The `offsetUserFunction` is **ONLY used** in case of static computation or index3 (generalizedAlpha) time integration. In order to be on the safe side, provide both `offsetUserFunction` and `offsetUserFunction_t`.

Note that itemNumber represents the index of the object in mbs, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

The user function gets time and the offset parameter as an input and returns the computed offset:

arguments / return	type or size	description
mbs	MainSystem	provides MainSystem mbs in which underlying item is defined
t	Real	current time in mbs
itemNumber	Index	integer number i_N of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
lOffset	Real	l_{off}
return value	Real	computed offset for given time

Userfunction: offsetUserFunction_t(mbs, t, itemNumber, lOffset)

A user function, which computes scalar offset **velocity** for the coordinate constraint. It is NECESSARY to use sufficiently smooth functions, having **initial offset velocities** consistent with **initial velocities** of bodies. The offsetUserFunction_t is used instead of offsetUserFunction in case of velocityLevel = True, or for index2 time integration and needed for computation of initial accelerations in second order implicit time integrators.

Note that itemNumber represents the index of the object in mbs, which can be used to retrieve additional data from the object through mbs.GetObjectParameter(itemNumber, ...), see the according description of GetObjectParameter.

The user function gets time and the offset parameter as an input and returns the computed offset velocity:

arguments / return	type or size	description
mbs	MainSystem	provides MainSystem mbs in which underlying item is defined
t	Real	current time in mbs
itemNumber	Index	integer number of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
lOffset	Real	l_{off}
return value	Real	computed offset velocity for given time

User function example:

```
#see also mini example!
from math import sin, cos, pi
def UFoffset(mbs, t, itemNumber, lOffset):
    return 0.5*lOffset*(1-cos(0.5*pi*t))

def UFoffset_t(mbs, t, itemNumber, lOffset): #time derivative of UFoffset
    return 0.5*lOffset*0.5*pi*sin(0.5*pi*t)

nMass=mbs.AddNode(Point(referenceCoordinates = [2,0,0]))
massPoint = mbs.AddObject(MassPoint(physicsMass = 5, nodeNumber = nMass))
```

```
    groundMarker=mbs.AddMarker(MarkerNodeCoordinate(nodeNumber= nGround, coordinate = 0)
)
    nodeMarker  =mbs.AddMarker(MarkerNodeCoordinate(nodeNumber= nMass, coordinate = 0))

    #Spring-Damper between two marker coordinates
    mbs.AddObject(CoordinateConstraint(markerNumbers = [groundMarker, nodeMarker],
                                       offset = 0.1,
                                       offsetUserFunction = UFoffset,
                                       offsetUserFunction_t = UFoffset_t))
```

8.7.2.4 MINI EXAMPLE for ObjectConnectorCoordinate

```
def OffsetUF(mbs, t, itemNumber, lOffset): #gives 0.05 at t=1
    return 0.5*(1-np.cos(2*3.141592653589793*0.25*t))*lOffset

nMass=mbs.AddNode(Point(referenceCoordinates = [2,0,0]))
massPoint = mbs.AddObject(MassPoint(physicsMass = 5, nodeNumber = nMass))

groundMarker=mbs.AddMarker(MarkerNodeCoordinate(nodeNumber= nGround, coordinate = 0))
nodeMarker  =mbs.AddMarker(MarkerNodeCoordinate(nodeNumber= nMass, coordinate = 0))

#Spring-Damper between two marker coordinates
mbs.AddObject(CoordinateConstraint(markerNumbers = [groundMarker, nodeMarker],
                                   offset = 0.1, offsetUserFunction = OffsetUF))

#assemble and solve system for default parameters
mbs.Assemble()
mbs.SolveDynamic()

#check result at default integration time
exudynTestGlobals.testResult  = mbs.GetNodeOutput(nMass, exu.OutputVariableType.
Displacement)[0]
```

For examples on ObjectConnectorCoordinate see Relevant Examples and TestModels with weblink:

- NGsolveModalAnalysis.py (Examples/)
- sliderCrank3DwithANCFbeltDrive2.py (Examples/)
- ballBearningModel.py (Examples/)
- camFollowerExample.py (Examples/)
- ALEANCFpipe.py (Examples/)
- ANCFALEtest.py (Examples/)

- ANCFcantileverTestDyn.py (Examples/)
- ANCFcontactCircle.py (Examples/)
- ANCFcontactCircle2.py (Examples/)
- ANCFmovingRigidbody.py (Examples/)
- ANCFrotatingCable2D.py (Examples/)
- ANCFslidingJoint2D.py (Examples/)
- ...
- contactCurveExample.py (TestModels/)
- createFunctionsTest.py (TestModels/)
- ANCFBeamTest.py (TestModels/)
- ...

8.7.3 ObjectConnectorCoordinateVector

A constraint which constrains the coordinate vectors of two markers Marker[Node|Object|Body]Coordinates attached to nodes or bodies. The marker uses the objects LTG-lists to build the according coordinate mappings.

Additional information for ObjectConnectorCoordinateVector:

- This `Object` has/provides the following types = `Connector`, `Constraint`
- Requested `Marker` type = `Coordinate`
- **Short name** for Python = `CoordinateVectorConstraint`
- **Short name** for Python visualization object = `VCoordinateVectorConstraint`

The item **ObjectConnectorCoordinateVector** with type = 'ConnectorCoordinateVector' has the following parameters:

Name	type	size	default value	description
name	String		''	constraints's unique name
markerNumbers	ArrayMarkerIndex		[invalid [-1], invalid [-1]]	list of markers used in connector
scalingMarker0	NumpyMatrix		Matrix[]	linear scaling matrix for coordinate vector of marker 0; matrix provided in Python numpy format
scalingMarker1	NumpyMatrix		Matrix[]	linear scaling matrix for coordinate vector of marker 1; matrix provided in Python numpy format
quadraticTermMarker0	NumpyMatrix		Matrix[]	quadratic scaling matrix for coordinate vector of marker 0; matrix provided in Python numpy format
quadraticTermMarker1	NumpyMatrix		Matrix[]	quadratic scaling matrix for coordinate vector of marker 1; matrix provided in Python numpy format
offset	NumpyVector		[]	offset added to constraint equation; only active, if no userFunction is defined
velocityLevel	Bool		False	If true: connector constrains velocities (only works for ODE2 coordinates!); offset is used between velocities; in this case, the offsetUserFunction_t is considered and offsetUserFunction is ignored
constraintUserFunction	PyFunctionVectorMbsScalarIndex2VectorBool		0	A Python user function which computes the constraint equations; to define the number of algebraic equations, set scalingMarker0 as a numpy.zeros((nAE,1)) array with nAE being the number algebraic equations; see description below

jacobianUserFunction	PyFunctionMatrixContainer		MbsScalarIndex2VectorBool 0	A Python user function which computes the jacobian, i.e., the derivative of the left-hand-side object equation w.r.t. the coordinates (times f_{ODE2}) and w.r.t. the velocities (times f_{ODE2_t}). Terms on the RHS must be subtracted from the LHS equation; the respective terms for the stiffness matrix and damping matrix are automatically added; see description below
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectConnectorCoordinateVector			parameters for visualization of item

The item VObjectConnectorCoordinateVector has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==-1, use default color

---

8.7.3.1 DESCRIPTION of ObjectConnectorCoordinateVector:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
markerNumbers	$[m0, m1]^{\mathrm{T}}$	
scalingMarker0	$\mathbf{X}_{m0} \in \mathbb{R}^{n_{ae} \times n_{q_{m0}}}$	
scalingMarker1	$\mathbf{X}_{m1} \in \mathbb{R}^{n_{ae} \times n_{q_{m1}}}$	
quadraticTermMarker0	$\mathbf{Y}_{m0} \in \mathbb{R}^{n_{ae} \times n_{q_{m0}}}$	
quadraticTermMarker1	$\mathbf{Y}_{m0} \in \mathbb{R}^{n_{ae} \times n_{q_{m0}}}$	
offset	$\mathbf{v}_{\mathrm{off}} \in \mathbb{R}^{n_{ae}}$	
constraintUserFunction	$\mathbf{c}_{user} \in \mathbb{R}^{n_{ae}}$	
jacobianUserFunction	$\mathbf{J}_{user} \in \mathbb{R}^{(n_{q_{m0}} + n_{q_{m1}}) \times n_{ae}}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**

Displacement	$\Delta\mathbf{q}$	relative scalar displacement of marker coordinates, not including scaling matrices
Velocity	$\Delta\mathbf{v}$	difference of scalar marker velocity coordinates, not including scaling matrices
ConstraintEquation	$\mathbf{c}$	(residuum of) constraint equations
Force	$\boldsymbol{\lambda}$	constraint force vector (vector of Lagrange multipliers), resulting from action of constraint equations

8.7.3.2 Definition of quantities

intermediate variables	**symbol**	**description**
marker m0 coordinate vector	$\mathbf{q}_{m0} \in \mathbb{R}^{n_{q_{m0}}}$	coordinate vector provided by marker $m0$; depending on the marker, the coordinates may or may not include reference coordinates
marker m1 coordinate vector	$\mathbf{q}_{m1} \in \mathbb{R}^{n_{q_{m1}}}$	coordinate vector provided by marker $m1$; depending on the marker, the coordinates may or may not include reference coordinates
marker m0 velocity coordinate vector	$\dot{\mathbf{q}}_{m0} \in \mathbb{R}^{n_{q_{m0}}}$	velocity coordinate vector provided by marker $m0$
marker m1 velocity coordinate vector	$\dot{\mathbf{q}}_{m1} \in \mathbb{R}^{n_{q_{m1}}}$	velocity coordinate vector provided by marker $m1$
number of algebraic equations	n_{ae}	number of algebraic equations must be same as number of rows in $\mathbf{X}_{m0}$ and $\mathbf{X}_{m1}$
difference of coordinates	$\Delta\mathbf{q} = \mathbf{q}_{m1} - \mathbf{q}_{m0}$	Displacement between marker m0 to marker m1 coordinates
difference of velocity coordinates	$\Delta\mathbf{v} = \dot{\mathbf{q}}_{m1} - \dot{\mathbf{q}}_{m0}$	

8.7.3.3 Remarks

The number of algebraic equations depends on the maximum number of rows in $\mathbf{X}_{m0}$, $\mathbf{Y}_{m0}$, $\mathbf{X}_{m1}$ and $\mathbf{Y}_{m1}$. The number of rows of the latter matrices must either be zero or the maximum of these rows.

The number of columns in $\mathbf{X}_{m0}$ (or $\mathbf{Y}_{m0}$) must agree with the length of the coordinate vector $\mathbf{q}_{m0}$ and the number of columns in $\mathbf{X}_{m1}$ (or $\mathbf{Y}_{m1}$) must agree with the length of the coordinate vector $\mathbf{q}_{m1}$, if these matrices are not empty matrices. If one marker k is a ground marker (node/object), the length of $\mathbf{q}_{m,k}$ is zero and also the according matrices $\mathbf{X}_{m,k}$, $\mathbf{Y}_{m,k}$ have zero size and will not be considered in the computation of the constraint equations.

8.7.3.4 Connector constraint equations

If `activeConnector = True` and no `constraintUserFunction` is defined, the index 3 algebraic equations

$$\mathbf{c}(\mathbf{q}_{m0}, \mathbf{q}_{m1}) = \mathbf{X}_{m1} \cdot \mathbf{q}_{m1} + \mathbf{Y}_{m1} \cdot \mathbf{q}_{m1}^2 - \mathbf{X}_{m0} \cdot \mathbf{q}_{m0} - \mathbf{Y}_{m0} \cdot \mathbf{q}_{m0}^2 - \mathbf{v}_{\text{off}} = 0 \tag{8.406}$$

Note that the squared coordinates are understood as $\mathbf{q}_{m0}^2 = [q_{0,m0}^2,\ q_{1,m0}^2,\ \ldots]^\mathrm{T}$, same for $\mathbf{q}_{m1}^2$.

The index 2 (velocity level) algebraic equation accordingly reads

$$\dot{\mathbf{c}}(\dot{\mathbf{q}}_{m0}, \dot{\mathbf{q}}_{m1}) = \mathbf{X}_{m1} \cdot \dot{\mathbf{q}}_{m1} + \mathbf{Y}_{m1} \cdot \dot{\mathbf{q}}_{m1}^2 - \mathbf{X}_{m0} \cdot \dot{\mathbf{q}}_{m0} - \mathbf{Y}_{m0} \cdot \dot{\mathbf{q}}_{m0}^2 - \mathbf{d}_{\text{off}} = 0 \tag{8.407}$$

The vector $\mathbf{d}$ in velocity level equations is zero, except if `parameters.velocityLevel = True`, then $\mathbf{d} = \mathbf{v}_{\text{off}}$.

Note that the index 2 equations are used, if the solver uses index 2 formulation OR if the flag `parameters.velocityLevel = True` (or both). However, the `constraintUserFunction` has to be chosen accordingly by the user, either as position or as velocity level. The user functions include dependency on time t, but this time dependency is not respected in the computation of initial accelerations. Therefore,

If `activeConnector = False`, the (index 1) algebraic equation reads for ALL cases:

$$\mathbf{c}(\lambda) = \lambda = 0 \tag{8.408}$$

If a `constraintUserFunction` is defined, it also requires an according `jacobianUserFunction` (and vice versa).

Userfunction: `constraintUserFunction(mbs, t, itemNumber, q, q_t, velocityLevel)`

A user function, which computes algebraic equations for the connector based on the marker coordinates stored in `q` and `q_t`. Depending on `velocityLevel`, the user function needs to compute either the position-level (`velocityLevel=False`) or the velocity level (`velocityLevel=True`) constraint equations. Note that for Index 2 solvers, the `constraintUserFunction` may be called with `velocityLevel=True` but `jacobianUserFunction` is called with `velocityLevel=False`. To define the number of algebraic equations, set `scalingMarker0` as a `numpy.zeros((nAE,1))` array with `nAE` being the number algebraic equations. The returned vector of `constraintUserFunction` must have size `nAE`.

Note that itemNumber represents the index of the ObjectGenericODE2 object in mbs, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs to which object belongs to
`t`	Real	current time in mbs
`itemNumber`	Index	integer number i_N of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
`q`	Vector $\in \mathbb{R}^{(n_{q_{m0}}+n_{q_{m1}})}$	connector coordinates, subsequently for marker $m0$ and marker $m1$, in current configuration
`q_t`	Vector $\in \mathbb{R}^{(n_{q_{m0}}+n_{q_{m1}})}$	connector velocity coordinates in current configuration

`velocityLevel`	Bool	velocityLevel as currently stored in connector
return value	Vector $\in \mathbb{R}^{n_{ae}}$	returns vector (numpy array or list) of evaluated constraint equations for connector

---

Userfunction: `jacobianUserFunction(mbs, t, itemNumber, q, q_t, velocityLevel)`

A user function, which computes the jacobian of the algebraic equations w.r.t. the ODE2 coordiantes (ODE2_t velocity coordinates if `velocityLevel=True`). The jacobian needs to exactly represent the derivative of the constraintUserFunction. The returned matrix of `jacobianUserFunction` must have `nAE` rows and `len(q)` columns.

arguments / return	**type or size**	**description**
`mbs`	MainSystem	provides MainSystem mbs to which object belongs to
`t`	Real	current time in mbs
`itemNumber`	Index	integer number i_N of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
`q`	Vector $\in \mathbb{R}^{(n_{q_{m0}}+n_{q_{m1}})}$	connector coordinates, subsequently for marker $m0$ and marker $m1$, in current configuration
`q_t`	Vector $\in \mathbb{R}^{(n_{q_{m0}}+n_{q_{m1}})}$	connector velocity coordinates in current configuration
`velocityLevel`	Bool	velocityLevel as currently stored in connector
return value	MatrixContainer $\in \mathbb{R}^{(n_{q_{m0}}+n_{q_{m1}})\times n_{ae}}$	returns special jacobian for connector, as exu.MatrixContainer, numpy array or list of lists; use MatrixContainer sparse format for larger matrices to speed up computations; sparse triplets MAY NOT contain zero values!

For examples on ObjectConnectorCoordinateVector see Relevant Examples and TestModels with weblink:

- `coordinateVectorConstraint.py` (TestModels/)
- `coordinateVectorConstraintGenericODE2.py` (TestModels/)
- `rigidBodyAsUserFunctionTest.py` (TestModels/)

8.8 Objects (Object)

A Object provides equations, using coordinates from Nodes. General objects lead to system equations, that do not represent physical Bodies or Connectors.

8.8.1 ObjectGenericODE1

A system of n first order ordinary differential equations (ODE1), having a system matrix, a rhs vector, but mostly it will use a user function to describe special ODE1 systems. It is based on NodeGenericODE1 nodes. NOTE that all matrices, vectors, etc. must have the same dimensions n or $(n \times n)$, or they must be empty (0×0), using [] in Python.

Additional information for ObjectGenericODE1:

- This `Object` has/provides the following types = `MultiNoded`
- Requested `Node` type: read detailed information of item

The item **ObjectGenericODE1** with type = 'GenericODE1' has the following parameters:

Name	type	size	default value	description
name	String		''	objects's unique name
nodeNumbers	ArrayNodeIndex		[]	node numbers which provide the coordinates for the object (consecutively as provided in this list)
systemMatrix	NumpyMatrix		Matrix[]	system matrix (state space matrix) of first order ODE
rhsVector	NumpyVector		[]	a constant rhs vector (e.g., for constant input)
rhsUserFunction	PyFunctionVectorMbsScalarIndexVector		0	A Python user function which computes the right-hand-side (rhs) of the first order ODE; see description below
coordinateIndexPerNode	ArrayIndex		[]	this list contains the local coordinate index for every node, which is needed, e.g., for markers; the list is generated automatically every time parameters have been changed
tempCoordinates	NumpyVector		[]	temporary vector containing coordinates
tempCoordinates_t	NumpyVector		[]	temporary vector containing velocity coordinates
visualization	VObjectGenericODE1			parameters for visualization of item

The item VObjectGenericODE1 has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

8.8.1.1 DESCRIPTION of ObjectGenericODE1:

Information on input parameters:

input parameter	symbol	description see tables above
nodeNumbers	$\mathbf{n}_n = [n_0, \ldots, n_n]^\mathrm{T}$	
systemMatrix	$\mathbf{A} \in \mathbb{R}^{n \times n}$	
rhsVector	$\mathbf{f} \in \mathbb{R}^n$	
rhsUserFunction	$\mathbf{f}_{user} \in \mathbb{R}^n$	
tempCoordinates	$\mathbf{c}_{temp} \in \mathbb{R}^n$	
tempCoordinates_t	$\dot{\mathbf{c}}_{temp} \in \mathbb{R}^n$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
CoordinatesTotal		all ODE2 displacement plus reference coordinates of object
Coordinates		all ODE1 coordinates
Coordinates_t		all ODE1 velocity coordinates

8.8.1.2 Equations of motion

An object with node numbers $[n_0, \ldots, n_n]$ and according numbers of nodal coordinates $[n_{c_0}, \ldots, n_{c_n}]$, the total number of equations (=coordinates) of the object is

$$n = \sum_i n_{c_i}, \tag{8.409}$$

which is used throughout the description of this object.

8.8.1.3 Equations of motion

$$\dot{\mathbf{q}} = \mathbf{f} + \mathbf{f}_{user}(mbs, t, i_N, \mathbf{q}) \tag{8.410}$$

Note that the user function $\mathbf{f}_{user}(mbs, t, i_N, \mathbf{q})$ may be empty (=0), and that `iN` represents the itemNumber (=objectNumber).

CoordinateLoads are added for the respective ODE1 coordinate on the RHS of the latter equation.

Userfunction: `rhsUserFunction(mbs, t, itemNumber, q)`

A user function, which computes a RHS vector depending on current time and states of the object. Can be used to create any kind of first order system, especially state space equations (inputs are added via CoordinateLoads to every node). Note that itemNumber represents the index of the ObjectGenericODE1 object in mbs, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs to which object belongs
`t`	Real	current time in mbs
`itemNumber`	Index	integer number i_N of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
`q`	Vector $\in \mathbb{R}^n$	object coordinates (composed from ODE1 nodal coordinates) in current configuration, without reference values
`return value`	Vector $\in \mathbb{R}^n$	returns force vector for object

---

User function example:

```
A = numpy.diag([200,100])
#simple linear user function returning A*q + const
def UFrhs(mbs, t, itemNumber, q):
    return np.dot(A, q) + np.array([0,2])

nODE1 = mbs.AddNode(NodeGenericODE1(referenceCoordinates=[0,0],
                                    initialCoordinates=[1,0],
numberOfODE1Coordinates=2))

#now add object instead of object in mini-example:
oGenericODE1 = mbs.AddObject(ObjectGenericODE1(nodeNumbers=[nODE1],
                     rhsUserFunction=UFrhs))
```

8.8.1.4 MINI EXAMPLE for ObjectGenericODE1

```
#set up a 2-DOF system
nODE1 = mbs.AddNode(NodeGenericODE1(referenceCoordinates=[0,0],
                                    initialCoordinates=[1,0],
                                    numberOfODE1Coordinates=2))
```

```
#build system matrix and force vector
#undamped mechanical system with m=1, K=100, f=1
A = np.array([[0,1],
              [-100,0]])
b = np.array([0,1])

oGenericODE1 = mbs.AddObject(ObjectGenericODE1(nodeNumbers=[nODE1],
                                               systemMatrix=A,
                                               rhsVector=b))

#assemble and solve system for default parameters
mbs.Assemble()

sims=exu.SimulationSettings()
solverType = exu.DynamicSolverType.RK44
mbs.SolveDynamic(solverType=solverType, simulationSettings=sims)

#check result at default integration time
exudynTestGlobals.testResult = mbs.GetNodeOutput(nODE1, exu.OutputVariableType.
Coordinates)[0]
```

For examples on ObjectGenericODE1 see Relevant Examples and TestModels with weblink:

- HydraulicsUserFunction.py (Examples/)
- lugreFrictionODE1.py (Examples/)
- lugreFrictionTest.py (Examples/)
- solverExplicitODE1ODE2test.py (TestModels/)
- taskmanagerTest.py (TestModels/)

8.9 Markers

A Marker provides an interface BETWEEN a large variety of Nodes / Bodies / Objects AND Connectors / Loads. To understand which markers are needed, see first the requested `Marker` type of the connector, constraint or joint. Hereafter, chose a `Marker` – attached to a node, body or object – with the according properties. The `Marker` may provide more information (e.g., position and orientation) than needed.

8.9.1 MarkerBodyMass

A marker attached to the body mass; use this marker to apply a body-load (e.g. gravitational force).

Additional information for MarkerBodyMass:

- This `Marker` has/provides the following types = `Object`, `Body`, `BodyMass`

The item **MarkerBodyMass** with type = 'BodyMass' has the following parameters:

Name	type	size	default value	description
name	String		"	marker's unique name
bodyNumber	ObjectIndex		invalid (-1)	body number to which marker is attached to
visualization	VMarkerBodyMass			parameters for visualization of item

The item VMarkerBodyMass has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

8.9.1.1 DESCRIPTION of MarkerBodyMass:

For examples on MarkerBodyMass see Relevant Examples and TestModels with weblink:

- `ALEANCFpipe.py` (Examples/)
- `ANCFmovingRigidbody.py` (Examples/)
- `ANCFslidingJoint2D.py` (Examples/)
- `ANCFslidingJoint2Drigid.py` (Examples/)

- ANCFswitchingSlidingJoint2D.py (Examples/)
- CMSexampleCourse.py (Examples/)
- finiteSegmentMethod.py (Examples/)
- NGsolveCMStutorial.py (Examples/)
- NGsolvePostProcessingStresses.py (Examples/)
- ObjectFFRFconvergenceTestBeam.py (Examples/)
- ObjectFFRFconvergenceTestHinge.py (Examples/)
- pendulumGeomExactBeam2D.py (Examples/)
- ...
- fourBarMechanismIftomm.py (TestModels/)
- genericJointUserFunctionTest.py (TestModels/)
- modelUnitTests.py (TestModels/)
- ...

8.9.2 MarkerBodyPosition

A position body-marker attached to a local (body-fixed) position ${}^{b}\mathbf{b} = [b_0,\ b_1,\ b_2]$ (x, y, and z coordinates) of the body. It provides position information as well as the according derivatives (=velocity and derivative of position w.r.t. body coordinates). It can be used for connectors, joints or loads where position is required. If connectors also require orientation information, use a MarkerBodyRigid.

Additional information for MarkerBodyPosition:

- This `Marker` has/provides the following types = `Object`, `Body`, `Position`

The item **MarkerBodyPosition** with type = 'BodyPosition' has the following parameters:

Name	type	size	default value	description
name	String		"	marker's unique name
bodyNumber	ObjectIndex		invalid (-1)	body number to which marker is attached to
localPosition	Vector3D	3	[0.,0.,0.]	local body position of marker; e.g. local (body-fixed) position where force is applied to
visualization	VMarkerBodyPosition			parameters for visualization of item

The item VMarkerBodyPosition has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

8.9.2.1 DESCRIPTION of MarkerBodyPosition:

Information on input parameters:

input parameter	symbol	description see tables above
localPosition	${}^{b}\mathbf{b}$	

The body position marker provides an interface to a object of type body (`ObjectGround`, `ObjectMassPoint`, `ObjectRigidBody`, ...) and provides access to kinematic quantities such as **position** and **velocity** and to the **position jacobian**, using a `localPosition` ${}^{b}\mathbf{b}$ which is defined within the local coordinates of the body (b). The kinematic quantities are computed according to the definition of output variables in the respective bodies.

The position jacobian represents the derivative of the node position $\mathbf{p}_n$ with all nodal coordinates,

$$ {}^{0}\mathbf{J}_{\text{pos}} = \frac{\partial\, {}^{0}\mathbf{p}_n}{\partial \mathbf{q}_n} \tag{8.411} $$

and it is usually computed as the derivative of the (global) translational velocity w.r.t. velocity coordinates,

$$ {}^{0}\mathbf{J}_{\text{pos}} = \frac{\partial\, {}^{0}\mathbf{v}_n}{\partial \dot{\mathbf{q}}_n} \tag{8.412} $$

As an example of the `ObjectRigidBody2D`, see Section 8.2.7, the position and velocity are computed as

$$ {}^{0}\mathbf{p}_{\,\text{config}}({}^{b}\mathbf{b}) = {}^{0}\mathbf{r}_{\,\text{config}} + {}^{0}\mathbf{r}_{\,\text{ref}} + {}^{0b}\mathbf{A}\ {}^{b}\mathbf{b}\ , \tag{8.413} $$

$$ {}^{0}\mathbf{v}_{\,\text{config}}({}^{b}\mathbf{b}) = {}^{0}\dot{\mathbf{u}}_{\,\text{config}} + {}^{0b}\mathbf{A}\,({}^{b}\boldsymbol{\omega} \times {}^{b}\mathbf{b}_{\,\text{config}})\ . \tag{8.414} $$

Thus, the position jacobian for `ObjectRigidBody2D` reads

$$ {}^{0}\mathbf{J}_{\text{pos}}^{\text{NodeRigidBody2D}} = \begin{bmatrix} 1 & 0 & -\sin\theta_0\, {}^{b}b_0 & -\cos\theta_0\, {}^{b}b_1 \\ 0 & 1 & \cos\theta_0\, {}^{b}b_0 & -\sin\theta_0\, {}^{b}b_1 \\ 0 & 0 & 0 & \end{bmatrix} \tag{8.415} $$

For details, see the respective definition of the body and the C++ implementation.

For examples on MarkerBodyPosition see Relevant Examples and TestModels with weblink:

- `ANCFcontactCircle.py` (Examples/)
- `ANCFcontactCircle2.py` (Examples/)
- `ANCFmovingRigidbody.py` (Examples/)
- `ANCFslidingJoint2D.py` (Examples/)
- `ANCFslidingJoint2Drigid.py` (Examples/)
- `ANCFswitchingSlidingJoint2D.py` (Examples/)
- `beltDrivesComparison.py` (Examples/)
- `bungeeJump.py` (Examples/)
- `coordinateSpringDamper.py` (Examples/)
- `finiteSegmentMethod.py` (Examples/)
- `flexibleRotor3Dtest.py` (Examples/)
- `geneticOptimizationSliderCrank.py` (Examples/)
- ...

- ANCFcontactCircleTest.py (TestModels/)
- ANCFcontactFrictionTest.py (TestModels/)
- ANCFmovingRigidBodyTest.py (TestModels/)
- ...

8.9.3 MarkerBodyRigid

A rigid-body (position+orientation) body-marker attached to a local (body-fixed) position ${}^{b}\mathbf{b} = [b_0,\ b_1,\ b_2]$ (x, y, and z coordinates) of the body. It provides position and orientation (rotation), as well as the according derivatives. It can be used for most connectors, joints or loads where either position, position and orientation, or orientation are required.

Additional information for MarkerBodyRigid:

- This `Marker` has/provides the following types = `Object`, `Body`, `Position`, `Orientation`

The item **MarkerBodyRigid** with type = 'BodyRigid' has the following parameters:

Name	type	size	default value	description
name	String		"	marker's unique name
bodyNumber	ObjectIndex		invalid (-1)	body number to which marker is attached to
localPosition	Vector3D	3	[0.,0.,0.]	local body position of marker; e.g. local (body-fixed) position where force is applied to
visualization	VMarkerBodyRigid			parameters for visualization of item

The item VMarkerBodyRigid has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

8.9.3.1 DESCRIPTION of MarkerBodyRigid:

Information on input parameters:

input parameter	symbol	description see tables above
localPosition	${}^{b}\mathbf{b}$	

For examples on MarkerBodyRigid see Relevant Examples and TestModels with weblink:

- `addPrismaticJoint.py` (Examples/)

- addRevoluteJoint.py (Examples/)
- ANCFcontactCircle.py (Examples/)
- ANCFcontactCircle2.py (Examples/)
- ANCFrotatingCable2D.py (Examples/)
- ANCFslidingJoint2D.py (Examples/)
- ANCFtestHalfcircle.py (Examples/)
- ANCFtests2.py (Examples/)
- ballBearningModel.py (Examples/)
- beamTutorial.py (Examples/)
- beltDriveALE.py (Examples/)
- beltDriveReevingSystem.py (Examples/)
- ...
- abaqusImportTest.py (TestModels/)
- ACFtest.py (TestModels/)
- ANCFbeltDrive.py (TestModels/)
- ...

8.9.4 MarkerNodePosition

A node-Marker attached to a position-based node. It can be used for connectors, joints or loads where position is required. If connectors also require orientation information, use a MarkerNodeRigid.

Additional information for MarkerNodePosition:

- This `Marker` has/provides the following types = `Node`, `Position`

The item **MarkerNodePosition** with type = 'NodePosition' has the following parameters:

Name	type	size	default value	description
name	String		"	marker's unique name
nodeNumber	NodeIndex		invalid (-1)	node number to which marker is attached to
visualization	VMarkerNodePosition			parameters for visualization of item

The item VMarkerNodePosition has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

8.9.4.1 DESCRIPTION of MarkerNodePosition:

The node position marker provides an interface to a node which contains a position (`NodePoint`, `NodePoint2D`, `NodeRigidBodyEP`, `NodePointSlope`, ...) and accesses **position**, **velocity** and the **position jacobian**. The position and velocity are computed according to the definition of output variables in the respective nodes.

The position jacobian represents the derivative of the node position $\mathbf{p}_n$ with all nodal coordinates,

$$ {}^0\mathbf{J}_{pos} = \frac{\partial\, {}^0\mathbf{p}_n}{\partial \mathbf{q}_n} \tag{8.416} $$

For details, see the respective definition of the node and the C++ implementation.

In examplary case of a `NodeRigidBody2D`, see Section 8.1.6, its coordinates are $\mathbf{q}_n = [q_0,\ q_1,\ \psi_0,\ ]^T$, where q_0 represents the x-displacement and q_1 represents the y-displacement, such that the jacobian for the 3D position vector reads

$$ {}^0\mathbf{J}_{pos}^{\text{NodeRigidBody2D}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{8.417} $$

For examples on MarkerNodePosition see Relevant Examples and TestModels with weblink:

- ALEANCFpipe.py (Examples/)
- ANCFcantileverTest.py (Examples/)
- ANCFcantileverTestDyn.py (Examples/)
- ANCFcontactCircle.py (Examples/)
- ANCFcontactCircle2.py (Examples/)
- ANCFslidingJoint2D.py (Examples/)
- ANCFslidingJoint2Drigid.py (Examples/)
- ANCFtestHalfcircle.py (Examples/)
- ANCFtests2.py (Examples/)
- bungeeJump.py (Examples/)
- doublePendulum2D.py (Examples/)
- flexibleRotor3Dtest.py (Examples/)
- ...
- ACFtest.py (TestModels/)
- ANCFcontactCircleTest.py (TestModels/)
- ANCFcontactFrictionTest.py (TestModels/)
- ...

8.9.5 MarkerNodeRigid

A rigid-body (position+orientation) node-marker attached to a rigid-body node. It provides position and orientation (rotation), as well as the according derivatives. It can be used for most connectors, joints or loads where either position, position and orientation, or orientation are required.

Additional information for MarkerNodeRigid:

- This `Marker` has/provides the following types = `Node`, `Position`, `Orientation`

The item **MarkerNodeRigid** with type = 'NodeRigid' has the following parameters:

Name	type	size	default value	description
name	String		"	marker's unique name
nodeNumber	NodeIndex		invalid (-1)	node number to which marker is attached to
visualization	VMarkerNodeRigid			parameters for visualization of item

The item VMarkerNodeRigid has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

8.9.5.1 DESCRIPTION of MarkerNodeRigid:

The node rigid body marker provides an interface to a node which contains a position and an orientation (`NodeRigidBodyEP`, `NodeRigidBody2D`, ...) and provides access to kinematic quantities such as **position**, **velocity**, **orientation** (rotation matrix), **angular velocity**. It also provides the **position jacobian** and the **rotation jacobian**. The kinematic quantities are computed according to the definition of output variables in the respective nodes.

The position jacobian represents the derivative of the node position $\mathbf{p}_n$ with all nodal coordinates,

$$ {}^0\mathbf{J}_{pos} = \frac{\partial\, {}^0\mathbf{p}_n}{\partial \mathbf{q}_n} \tag{8.418} $$

and it is usually computed as the derivative of the (global) translational velocity w.r.t. velocity coordinates,

$$ {}^0\mathbf{J}_{pos} = \frac{\partial\, {}^0\mathbf{v}_n}{\partial \dot{\mathbf{q}}_n} \tag{8.419} $$

The rotation jacobian is computed as the derivative of the (global) angular velocity w.r.t. velocity coordinates,

$$ {}^{0}\mathbf{J}_{\text{rot}} = \frac{\partial\, {}^{0}\boldsymbol{\omega}_{\text{n}}}{\partial \dot{\mathbf{q}}_{\text{n}}} \tag{8.420} $$

This usually results in the velocity transformation matrix. For details, see the respective definition of the node and the C++ implementation.

For examples on MarkerNodeRigid see Relevant Examples and TestModels with weblink:

- `ANCFcontactCircle.py` (Examples/)
- `ANCFcontactCircle2.py` (Examples/)
- `ANCFrotatingCable2D.py` (Examples/)
- `ANCFslidingJoint2D.py` (Examples/)
- `ANCFtestHalfcircle.py` (Examples/)
- `ANCFtests2.py` (Examples/)
- `beamTutorial.py` (Examples/)
- `beltDriveALE.py` (Examples/)
- `beltDriveReevingSystem.py` (Examples/)
- `beltDrivesComparison.py` (Examples/)
- `CMSexampleCourse.py` (Examples/)
- `newtonsCradle.py` (Examples/)
- ...
- `abaqusImportTest.py` (TestModels/)
- `ANCFBeamTest.py` (TestModels/)
- `ANCFbeltDrive.py` (TestModels/)
- ...

8.9.6 MarkerNodeCoordinate

A node-Marker attached to a ODE2 coordinate of a node; this marker allows to connect a coordinate-based constraint or connector to a nodal coordinate (also NodeGround); for ODE1 coordinates use MarkerNodeODE1Coordinate.

Additional information for MarkerNodeCoordinate:

- This `Marker` has/provides the following types = `Node`, `Coordinate`

The item **MarkerNodeCoordinate** with type = 'NodeCoordinate' has the following parameters:

Name	type	size	default value	description
name	String		"	marker's unique name
nodeNumber	NodeIndex		invalid (-1)	node number to which marker is attached to
coordinate	UInt		invalid (-1)	coordinate of node to which marker is attached to
visualization	VMarkerNodeCoordinate			parameters for visualization of item

The item VMarkerNodeCoordinate has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

8.9.6.1 DESCRIPTION of MarkerNodeCoordinate:

For examples on MarkerNodeCoordinate see Relevant Examples and TestModels with weblink:

- `ALEANCFpipe.py` (Examples/)
- `ANCFALEtest.py` (Examples/)
- `ANCFcantileverTestDyn.py` (Examples/)
- `ANCFcontactCircle.py` (Examples/)
- `ANCFcontactCircle2.py` (Examples/)
- `ANCFmovingRigidbody.py` (Examples/)

- ANCFrotatingCable2D.py (Examples/)
- ANCFslidingJoint2D.py (Examples/)
- ANCFslidingJoint2Drigid.py (Examples/)
- ANCFswitchingSlidingJoint2D.py (Examples/)
- ANCFtestHalfcircle.py (Examples/)
- ANCFtests2.py (Examples/)
- ...
- ANCFBeamEigTest.py (TestModels/)
- ANCFBeamTest.py (TestModels/)
- ANCFbeltDrive.py (TestModels/)
- ...

8.9.7 MarkerNodeCoordinates

A node-Marker attached to all ODE2 coordinates of a node; IN CONTRAST to MarkerNodeCoordinate, the marker coordinates INCLUDE the reference values! for ODE1 coordinates use MarkerNodeODE1Coordinates (under development).

Additional information for MarkerNodeCoordinates:

- This `Marker` has/provides the following types = `Node`, `Coordinate`

The item **MarkerNodeCoordinates** with type = 'NodeCoordinates' has the following parameters:

Name	type	size	default value	description
name	String		''	marker's unique name
nodeNumber	NodeIndex		invalid (-1)	node number to which marker is attached to
visualization	VMarkerNodeCoordinates			parameters for visualization of item

The item VMarkerNodeCoordinates has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

8.9.7.1 DESCRIPTION of MarkerNodeCoordinates:

For examples on MarkerNodeCoordinates see Relevant Examples and TestModels with weblink:

- `coordinateVectorConstraint.py` (TestModels/)
- `coordinateVectorConstraintGenericODE2.py` (TestModels/)
- `rigidBodyAsUserFunctionTest.py` (TestModels/)

8.9.8 MarkerNodeODE1Coordinate

A node-Marker attached to a ODE1 coordinate of a node.

Additional information for MarkerNodeODE1Coordinate:

- This `Marker` has/provides the following types = `Node`, `Coordinate`

The item **MarkerNodeODE1Coordinate** with type = 'NodeODE1Coordinate' has the following parameters:

Name	type	size	default value	description
name	String		"	marker's unique name
nodeNumber	NodeIndex		invalid (-1)	node number to which marker is attached to
coordinate	UInt		invalid (-1)	coordinate of node to which marker is attached to
visualization	VMarkerNodeODE1Coordinate			parameters for visualization of item

The item VMarkerNodeODE1Coordinate has the following parameters:

Name	type	size	default value	description
show	Bool		False	currently not available; set true, if item is shown in visualization and false if it is not shown

8.9.9 MarkerNodeRotationCoordinate

A node-Marker attached to a a node containing rotation; the Marker measures a rotation coordinate (Tait-Bryan angles) or angular velocities on the velocity level.

Additional information for MarkerNodeRotationCoordinate:

- This `Marker` has/provides the following types = `Node`, `Coordinate`

The item **MarkerNodeRotationCoordinate** with type = 'NodeRotationCoordinate' has the following parameters:

Name	type	size	default value	description
name	String		"	marker's unique name
nodeNumber	NodeIndex		invalid (-1)	node number to which marker is attached to
rotationCoordinate	UInt		invalid (-1)	rotation coordinate: 0=x, 1=y, 2=z
visualization	VMarkerNodeRotationCoordinate			parameters for visualization of item

The item VMarkerNodeRotationCoordinate has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

8.9.9.1 DESCRIPTION of MarkerNodeRotationCoordinate:

For examples on MarkerNodeRotationCoordinate see Relevant Examples and TestModels with weblink:

- `openVRengine.py` (Examples/)
- `pistonEngine.py` (Examples/)
- `rigidRotor3DbasicBehaviour.py` (Examples/)
- `sliderCrank3DwithANCFbeltDrive2.py` (Examples/)
- `driveTrainTest.py` (TestModels/)
- `sliderCrank3Dbenchmark.py` (TestModels/)

8.9.10 MarkerSuperElementPosition

A position marker attached to a SuperElement, such as ObjectFFRF, ObjectGenericODE2 and ObjectFFRFreducedOrder (for which it is in its current implementation inefficient for large number of meshNodeNumbers). The marker acts on the mesh (interface) nodes, not on the underlying nodes of the object.

Additional information for MarkerSuperElementPosition:

- This `Marker` has/provides the following types = `Object`, `Body`, `Position`

The item **MarkerSuperElementPosition** with type = 'SuperElementPosition' has the following parameters:

Name	type	size	default value	description
name	String		"	marker's unique name
bodyNumber	ObjectIndex		invalid (-1)	body number to which marker is attached to
meshNodeNumbers	ArrayIndex		[]	a list of n_m mesh node numbers of superelement (=interface nodes) which are used to compute the body-fixed marker position; the related nodes must provide 3D position information, such as NodePoint, NodePoint2D, NodeRigidBody[..]; in order to retrieve the global node number, the generic body needs to convert local into global node numbers
weightingFactors	Vector		[]	a list of n_m weighting factors per node to compute the final local position; the sum of these weights shall be 1, such that a summation of all nodal positions times weights gives the average position of the marker
visualization	VMarkerSuperElementPosition			parameters for visualization of item

The item VMarkerSuperElementPosition has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
showMarkerNodes	Bool		True	set true, if all nodes are shown (similar to marker, but with less intensity)

8.9.10.1 DESCRIPTION of MarkerSuperElementPosition:

Information on input parameters:

input parameter	symbol	description see tables above
bodyNumber	n_b	
meshNodeNumbers	$[k_0, \ldots, k_{n_m-1}]^T$	
weightingFactors	$[w_0, \ldots, w_{n_m-1}]^T$	

Definition of marker quantities:

intermediate variables	symbol	description
number of mesh nodes	n_m	size of `meshNodeNumbers` and `weightingFactors` which are marked; this must not be the number of mesh nodes in the marked object
mesh node number	$i = k_i$	abbreviation
mesh node points	${}^0\mathbf{p}_i$	position of mesh node k_i in object n_b
mesh node velocities	${}^0\mathbf{v}_i$	velocity of mesh node i in object n_b
marker position	${}^0\mathbf{p}_m = \sum_i w_i \cdot {}^0\mathbf{p}_i$	current global position which is provided by marker
marker velocity	${}^0\mathbf{v}_m = \sum_i w_i \cdot {}^0\mathbf{v}_i$	current global velocity which is provided by marker

8.9.10.2 Marker quantities

The marker provides a 'position' jacobian, which is the derivative of the marker velocity w.r.t. the object velocity coordinates $\dot{\mathbf{q}}_{n_b}$,

$$\mathbf{J}_{m,pos} = \frac{\partial\, {}^0\mathbf{v}_m}{\partial \dot{\mathbf{q}}_{n_b}} = \sum_i w_i \cdot \mathbf{J}_{i,pos} \tag{8.421}$$

in which $\mathbf{J}_{i,pos}$ denotes the position jacobian of mesh node i,

$$\mathbf{J}_{i,pos} = \frac{\partial\, {}^0\mathbf{v}_i}{\partial \dot{\mathbf{q}}_{n_b}} \tag{8.422}$$

The jacobian $\mathbf{J}_{i,pos}$ usually contains mostly zeros for `ObjectGenericODE2`, because the jacobian only affects one single node. In `ObjectFFRFreducedOrder`, the jacobian may affect all reduced coordinates.

Note that $\mathbf{J}_{m,pos}$ is actually computed by the `ObjectSuperElement` within the function `GetAccessFunctionSuper`

8.9.10.3 MINI EXAMPLE for MarkerSuperElementPosition

```
#set up a mechanical system with two nodes; it has the structure: |~~M0~~M1
#==>further examples see objectGenericODE2Test.py, objectFFRFTest2.py, etc.
nMass0 = mbs.AddNode(NodePoint(referenceCoordinates=[0,0,0]))
nMass1 = mbs.AddNode(NodePoint(referenceCoordinates=[1,0,0]))
mGround = mbs.AddMarker(MarkerBodyPosition(bodyNumber=oGround, localPosition = [1,0,0]))
```

```
mass = 0.5 * np.eye(3)      #mass of nodes
stif = 5000 * np.eye(3)     #stiffness of nodes
damp = 50 * np.eye(3)      #damping of nodes
Z = 0. * np.eye(3)          #matrix with zeros
#build mass, stiffness and damping matrices (:
M = np.block([[mass,         0.*np.eye(3)],
              [0.*np.eye(3), mass        ] ])
K = np.block([[2*stif, -stif],
              [ -stif,  stif] ])
D = np.block([[2*damp, -damp],
              [ -damp,  damp] ])

oGenericODE2 = mbs.AddObject(ObjectGenericODE2(nodeNumbers=[nMass0,nMass1],
                                               massMatrix=M,
                                               stiffnessMatrix=K,
                                               dampingMatrix=D))

#EXAMPLE for single node marker on super element body, mesh node 1; compare results to
ObjectGenericODE2 example!!!
mSuperElement = mbs.AddMarker(MarkerSuperElementPosition(bodyNumber=oGenericODE2,
meshNodeNumbers=[1], weightingFactors=[1]))
mbs.AddLoad(Force(markerNumber = mSuperElement, loadVector = [10, 0, 0]))

#assemble and solve system for default parameters
mbs.Assemble()

mbs.SolveDynamic(solverType = exudyn.DynamicSolverType.TrapezoidalIndex2)

#check result at default integration time
exudynTestGlobals.testResult = mbs.GetNodeOutput(nMass1, exu.OutputVariableType.Position
)[0]
```

For examples on MarkerSuperElementPosition see Relevant Examples and TestModels with weblink:

- NGsolvePistonEngine.py (Examples/)
- NGsolvePostProcessingStresses.py (Examples/)
- objectFFRFreducedOrderNetgen.py (Examples/)
- objectFFRFreducedOrderAccelerations.py (TestModels/)
- objectFFRFreducedOrderStressModesTest.py (TestModels/)
- objectFFRFreducedOrderTest.py (TestModels/)
- objectFFRFTest.py (TestModels/)
- objectFFRFTest2.py (TestModels/)
- objectGenericODE2Test.py (TestModels/)

- perfObjectFFRFreducedOrder.py (TestModels/)
- superElementRigidJointTest.py (TestModels/)

8.9.11 MarkerSuperElementRigid

A position and orientation (rigid-body) marker attached to a SuperElement, such as ObjectFFRF, ObjectGenericODE2 and ObjectFFRFreducedOrder (for which it may be inefficient). The marker acts on the mesh nodes, not on the underlying nodes of the object. Note that in contrast to the MarkerSuperElementPosition, this marker needs a set of interface nodes which are not aligned at one line, such that these node points can represent a rigid body motion. Note that definitions of marker positions are slightly different from MarkerSuperElementPosition.

Additional information for MarkerSuperElementRigid:

- This `Marker` has/provides the following types = `Object`, `Body`, `Position`, `Orientation`

The item **MarkerSuperElementRigid** with type = 'SuperElementRigid' has the following parameters:

Name	type	size	default value	description
name	String		"	marker's unique name
bodyNumber	ObjectIndex		invalid (-1)	body number to which marker is attached to
offset	Vector3D	3	[0.,0.,0.]	local marker SuperElement reference position offset used to correct the center point of the marker, which is computed from the weighted average of reference node positions (which may have some offset to the desired joint position). Note that this offset shall be small and larger offsets can cause instability in simulation models (better to have symmetric meshes at joints).
meshNodeNumbers	ArrayIndex		[]	a list of n_m mesh node numbers of superelement (=interface nodes) which are used to compute the body-fixed marker position and orientation; the related nodes must provide 3D position information, such as NodePoint, NodePoint2D, NodeRigidBody[..]; in order to retrieve the global node number, the generic body needs to convert local into global node numbers
weightingFactors	Vector		[]	a list of n_m weighting factors per node to compute the final local position and orientation; these factors could be based on surface integrals of the constrained mesh faces

useAlternativeApproach	Bool		True	this flag switches between two versions for the computation of the rotation and angular velocity of the marker; alternative approach uses skew symmetric matrix of reference position; follows the inertia concept
rotationsExponentialMap	Index		2	Experimental flag (2 is the correct value and will be used in future, removing this flag): This value switches different behavior for computation of rotations and angular velocities: 0 uses linearized rotations and angular velocities, 1 uses the exponential map for rotations but linear angular velocities, 2 uses the exponential map for rotations and the according tangent map for angular velocities
visualization	VMarkerSuperElementRigid			parameters for visualization of item

The item VMarkerSuperElementRigid has the following parameters:

Name	**type**	**size**	**default value**	**description**
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
showMarkerNodes	Bool		True	set true, if all nodes are shown (similar to marker, but with less intensity)

8.9.11.1 DESCRIPTION of MarkerSuperElementRigid:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
bodyNumber	n_b	
offset	${}^r\mathbf{o}_{ref}$	
meshNodeNumbers	$[k_0, \ldots, k_{n_m-1}]^\mathrm{T}$	
weightingFactors	$[w_0, \ldots, w_{n_m-1}]^\mathrm{T}$	

Definition of marker quantities:

intermediate variables	**symbol**	**description**
number of mesh nodes	n_m	size of `meshNodeNumbers` and `weightingFactors` which are marked; this must not be the number of mesh nodes in the marked object
mesh node number	$i = k_i$	abbreviation, runs over all marker mesh nodes

mesh node local displacement	${}^{r}\mathbf{u}^{(i)}$	current local (within reference frame r) displacement of mesh node k_i in object n_b
mesh node local position	${}^{r}\mathbf{p}^{(i)} = {}^{r}\mathbf{x}^{(i)}_{\text{ref}} + {}^{r}\mathbf{u}^{(i)}$	current local (within reference frame r, which is the body frame b ,e.g., in `ObjectFFRFreducedOrder`) position of mesh node k_i in object n_b
mesh node local reference position	${}^{r}\mathbf{x}^{(i)}_{\text{ref}}$	local (within reference frame r) reference position of mesh node k_i in object n_b, see e.g. `ObjectFFRFreducedOrder`
averaged local reference position	${}^{r}\mathbf{x}^{\text{avg}}_{\text{ref}} = \sum_i w_i \, {}^{r}\mathbf{x}^{(i)}_{\text{ref}}$	midpoint reference position of marker; averaged local reference positions of all mesh nodes k_i, using weighting for averaging; may not coincide with center point of your idealized joint surface (e.g., midpoint of cylinder), see Fig. 8.12
marker centered mesh node local reference position	${}^{r}\mathbf{p}^{(i)}_{\text{ref}} = {}^{r}\mathbf{x}^{(i)}_{\text{ref}} - {}^{r}\mathbf{x}^{\text{avg}}_{\text{ref}}$	local reference position of mesh node k_i relative to the center position of marker
mesh node local velocity	${}^{r}\mathbf{v}^{(i)}$	current local (within reference frame r) velocity of mesh node k_i in object n_b
super element reference point	${}^{0}\mathbf{p}_{r}$ (= ${}^{0}\mathbf{p}_{\text{t}}$ in `ObjectFFRFreducedOrder`)	current position (origin) of super element's floating frame (r), which is zero, if the object does not provide a reference frame (such as GenericODE2)
super element rotation matrix	${}^{0r}\mathbf{A}$	current rigid body transformation matrix of super element's floating frame (r), which is the identity matrix, if the object does not provide a reference frame (such as GenericODE2)
super element angular velocity	${}^{r}\boldsymbol{\omega}_{r}$	current local angular velocity of super element's floating frame (r), which is zero, if the object does not provide a reference frame (such as GenericODE2)
marker position	${}^{0}\mathbf{p}_{m} = {}^{0}\mathbf{p}_{r} + {}^{0r}\mathbf{A}\left({}^{r}\mathbf{o}_{\text{ref}} + \sum_i w_i \cdot {}^{r}\mathbf{p}^{(i)}\right)$	current global position which is provided by marker; note offset ${}^{r}\mathbf{o}_{\text{ref}}$ added, if used as a correction of marker mesh nodes
marker velocity	${}^{0}\mathbf{v}_{m} = {}^{0}\dot{\mathbf{p}}_{r} + {}^{0r}\mathbf{A}\left({}^{r}\tilde{\boldsymbol{\omega}}_{r}\left({}^{r}\mathbf{o}_{\text{ref}} + \sum_i w_i \cdot {}^{r}\mathbf{p}^{(i)}\right) + \sum_i (w_i \cdot {}^{r}\dot{\mathbf{u}}^{(i)})\right)$	current global velocity which is provided by marker
marker rotation matrix	${}^{0r}\mathbf{A}_{m} = {}^{0r}\mathbf{A} \cdot \mathbf{exp}({}^{r}\boldsymbol{\theta}_{m})$	current rotation matrix, which transforms the local marker coordinates and adds the rigid body transformation of floating frames ${}^{0r}\mathbf{A}$; uses exponential map for SO3, assumes that $\boldsymbol{\theta}$ represents a rotation vector
marker local rotation	${}^{r}\boldsymbol{\theta}_{m}$	current local linearized rotations (rotation vector); for the computation, see below for the standard and alternative approach
marker local angular velocity	${}^{r}\boldsymbol{\omega}_{m}$	local angular velocity due to mesh node velocity only; for the computation, see below for the standard and alternative approach

marker global angular velocity	${}^0\boldsymbol{\omega}_m = {}^0\boldsymbol{\omega}_r + {}^{0r}\mathbf{A}\ {}^r\boldsymbol{\omega}_m$	current global angular velocity

8.9.11.2 Marker background

The marker allows to realize a multi-point constraint (assuming that the marker is used in a joint constraint), connecting to averaged nodal displacements and rotations (also known as RBE3 in NASTRAN), see e.g. [29]. However, using Craig-Bampton RBE2 modes, will create RBE2 multi-point constraints for `ObjectFFRFreducedOrder` objects.

For more information on the various quantities and their coordinate systems, see table above and Fig. 8.12.

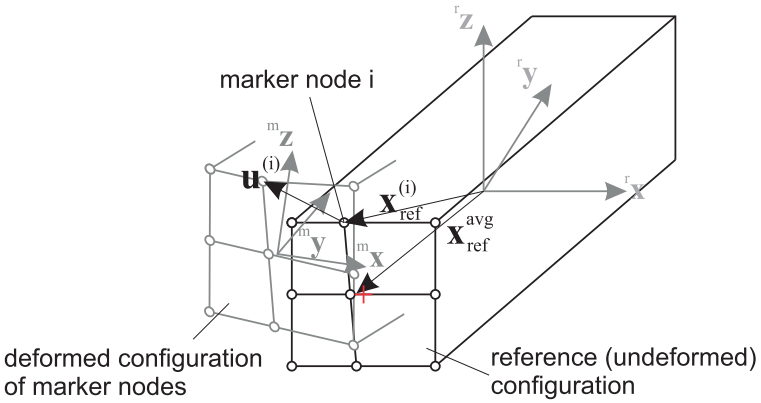


Figure 8.12: Sketch of marker nodes, exemplary node i, reference coordinates and marker coordinate system; note the difference of the center of the marker 'surface' (rectangle) marked with the red cross, and the averaged of the averaged local reference position.

8.9.11.3 Marker quantities

The marker provides a 'position' jacobian, which is the derivative of the global marker velocity w.r.t. the object velocity coordinates $\dot{\mathbf{q}}_{n_b}$,

$$ {}^0\mathbf{J}_{m,pos} = \frac{\partial\, {}^0\mathbf{v}_m}{\dot{\mathbf{q}}_{n_b}} \,. \tag{8.423} $$

In case of `ObjectGenericODE2`, assuming pure displacement based nodes, the jacobian will consist of zeros and unit matrices $\mathbf{I}$,

$$ {}^0\mathbf{J}^{GenericODE2}_{m,pos} = \frac{\partial\, {}^0\mathbf{v}_m}{\dot{\mathbf{q}}_{n_b}} = [\mathbf{0},\ \ldots,\ \mathbf{0},\ w_0\mathbf{I},\ \mathbf{0},\ \ldots,\ \mathbf{0},\ w_1\mathbf{I},\ \mathbf{0},\ \ldots,\ \mathbf{0}] \ , \tag{8.424} $$

in which the $\mathbf{I}$ matrices are placed at the according indices of marker nodes.

In case of `ObjectFFRFreducedOrder`, this jacobian is computed as weighted sum of the position jacobians, see `ObjectFFRFreducedOrder`,

$$ {}^0\mathbf{J}^{FFRFreduced}_{m,pos} = \frac{\partial\, {}^0\mathbf{v}_m}{\dot{\mathbf{q}}_{n_b}} = \sum_i w_i\, {}^0\mathbf{J}^{(i)}_{\text{pos}} = \left[\mathbf{I},\ -{}^{0r}\mathbf{A} \left({}^r\mathbf{o}_{\text{ref}} + \sum_i {}^r\mathbf{p}^{(i)} \right) {}^r\mathbf{G} \,,\ \sum_i w_i\, {}^{0r}\mathbf{A} \begin{bmatrix} {}^r\boldsymbol{\Psi}^{\mathrm{T}}_{r=3i} \\ {}^r\boldsymbol{\Psi}^{\mathrm{T}}_{r=3i+1} \\ {}^r\boldsymbol{\Psi}^{\mathrm{T}}_{r=3i+2} \end{bmatrix} \right] . \tag{8.425} $$

In `ObjectFFRFreducedOrder`, the jacobian usually affects all reduced coordinates.

8.9.11.4 Standard approach for computation of rotation (`useAlternativeApproach = False`)

As compared to `MarkerSuperElementPosition`, `MarkerSuperElementRigid` also links the marker to the orientation of the set of nodes provided. For this reason, the check performed in `mbs.assemble()` will take care that the nodes are capable to describe rotations. The first approach, here called as a standard, follows the idea that displacements contribute to rotation are weighted by their quadratic distance, cf. [29], and gives the (small rotation) rotation vector

$$ {}^r\boldsymbol{\theta}_m = \frac{\sum_i w_i \, {}^r\tilde{\mathbf{p}}^{(i)}_{ref} \times {}^r\mathbf{u}^{(i)}}{\sum_i w_i |{}^r\mathbf{p}^{(i)}_{ref}|^2} \tag{8.426} $$

Note that $\mathbf{p}^{(i)}_{ref}$ is not the reference position in the `ObjectFFRFreducedOrder` object, but it is relative to the midpoint reference position all marker nodes, given in ${}^r\mathbf{x}^{\text{avg}}_{\text{ref}}$. Accordingly, the marker local angular velocity can be calculated as

$$ {}^r\boldsymbol{\omega}_m = {}^r\dot{\boldsymbol{\theta}}_m = \frac{\sum_i w_i \, {}^r\tilde{\mathbf{p}}^{(i)}_{ref} \, {}^r\mathbf{v}_i}{\sum_i w_i |{}^r\mathbf{p}^{(i)}_{ref}|^2} \tag{8.427} $$

The marker also provides a 'rotation' jacobian, which is the derivative of the marker angular velocity ${}^0\boldsymbol{\omega}_m$ w.r.t. the object velocity coordinates $\dot{\mathbf{q}}_{n_b}$,

$$ {}^0\mathbf{J}_{m,rot} = \frac{\partial \, {}^0\boldsymbol{\omega}_m}{\partial \dot{\mathbf{q}}_{n_b}} = \frac{\partial \, {}^{0r}\mathbf{A} \, ({}^r\boldsymbol{\omega}_r + {}^r\boldsymbol{\omega}_m)}{\partial \dot{\mathbf{q}}_{n_b}} = {}^{0r}\mathbf{A} \left(\frac{\partial \, {}^r\boldsymbol{\omega}_r}{\partial \dot{\mathbf{q}}_{n_b}} + \frac{\sum_i w_i \, {}^r\tilde{\mathbf{p}}^{(i)}_{ref} \, {}^r\mathbf{J}^{(i)}_{pos}}{\sum_i w_i |{}^r\mathbf{p}^{(i)}_{ref}|^2} \right) \tag{8.428} $$

In case of `ObjectFFRFreducedOrder`, this jacobian is computed as

$$ {}^0\mathbf{J}^{FFRFreduced}_{m,rot} = \left[\mathbf{0}, \; {}^{0r}\mathbf{A} \, {}^r\mathbf{G}_{local}, \; {}^{0r}\mathbf{A} \, \frac{\sum_i w_i \, {}^r\tilde{\mathbf{p}}^{(i)}_{ref} \, {}^r\mathbf{J}^{(i)}_{pos,f}}{\sum_i w_i |{}^r\mathbf{p}^{(i)}_{ref}|^2} \right] \tag{8.429} $$

in which you should know that

- we used $\frac{\partial \, {}^r\boldsymbol{\omega}_r}{\partial \dot{\boldsymbol{\theta}}_r} = {}^r\mathbf{G}_{local}$,
- $\boldsymbol{\theta}_r$ represent the rotation parameters for the rigid body node of `ObjectFFRFreducedOrder`,
- ${}^r\mathbf{J}^{(i)}_{pos,f}$ is the **local** jacobian, which only includes the flexible part of the local jacobian for a single mesh node, ${}^r\mathbf{J}^{(i)}_{pos}$ (note the small *r* on the upper left), as defined in `ObjectFFRFreducedOrder`.

For further quantities also consult the according description in `ObjectFFRFreducedOrder`.

8.9.11.5 Alternative computation of rotation (`useAlternativeApproach = True`)

Note that this approach is **still under development** and needs further validation. However, tests show that this model is superior to the standard approach, as it improves the averaging of motion w.r.t. rotations at the marker nodes.

In the alternative approach, the weighting matrix $\mathbf{W}$ has the interpretation of an inertia tensor built from nodes using weights equal to node masses. In such an interpretation, the 'local angular momentum' w.r.t. the marker (averaged) position can be computed as

$$\mathbf{W}\,{}^r\boldsymbol{\omega}_m = \sum_i w_i\,{}^r\tilde{\mathbf{p}}_{ref}^{(i)}\left({}^r\mathbf{v}^{(i)} - {}^r\mathbf{v}^{\text{avg}}\right) = -\sum_i \left(w_i\,{}^r\tilde{\mathbf{p}}_{ref}^{(i)}\,{}^r\tilde{\mathbf{p}}_{ref}^{(i)}\right){}^r\boldsymbol{\omega}_m \tag{8.430}$$

which implicitly defines the weighting matrix $\mathbf{W}$, which must be invertable (but it is only a 3×3 matrix!),

$$\mathbf{W} = -\sum_i w_i\,{}^r\tilde{\mathbf{p}}_{ref}^{(i)}\,{}^r\tilde{\mathbf{p}}_{ref}^{(i)} \tag{8.431}$$

Furthermore, we need to introduce the averaged velocity of the marker averaged reference position, using ${}^r\dot{\mathbf{u}}^{(i)} = {}^r\mathbf{v}^{(i)}$, which is defined as

$${}^r\mathbf{v}^{\text{avg}} = \sum_i w_i\,{}^r\mathbf{v}^{(i)}\ , \tag{8.432}$$

similar to the averaged local reference position ${}^r\mathbf{x}_{\text{ref}}^{\text{avg}}$ given in the table above, see also Fig. 8.12.

In the alternative approach, thus the marker local rotations read

$${}^r\boldsymbol{\theta}_{m,alt} = \mathbf{W}^{-1}\sum_i w_i\,{}^r\tilde{\mathbf{p}}_{ref}^{(i)}\left({}^r\mathbf{u}^{(i)} - {}^r\mathbf{x}_{\text{ref}}^{\text{avg}}\right)\ , \tag{8.433}$$

and the marker local angular velocity is defined as

$${}^r\boldsymbol{\omega}_{m,alt} = \mathbf{W}^{-1}\sum_i w_i\,{}^r\tilde{\mathbf{p}}_{ref}^{(i)}\left({}^r\mathbf{v}^{(i)} - {}^r\mathbf{v}^{\text{avg}}\right)\ . \tag{8.434}$$

Note that, the average velocity ${}^r\mathbf{v}^{\text{avg}}$ would cancel out in a symmetric mesh, but would cause spurious angular velocities in unsymmetric (w.r.t. the axis of rotation) distribition of mesh nodes. This could even lead to spurious rotations or angular velocities in pure translatoric motion.

In the alternative mode, the Jacobian for the rotation / angular velocity is defined as

$${}^0\mathbf{J}_{m,rot,alt} = \frac{\partial\,{}^0\boldsymbol{\omega}_m}{\partial\dot{\mathbf{q}}_{n_b}} = \frac{\partial\,{}^{0r}\mathbf{A}\,({}^r\boldsymbol{\omega}_r + {}^r\boldsymbol{\omega}_m)}{\partial\dot{\mathbf{q}}_{n_b}} = {}^{0r}\mathbf{A}\left(\frac{\partial\,{}^r\boldsymbol{\omega}_r}{\partial\dot{\mathbf{q}}_{n_b}} + \mathbf{W}^{-1}\sum_i w_i\,{}^r\tilde{\mathbf{p}}_{ref}^{(i)}\,{}^r\mathbf{J}_{pos}^{(i)}\right) \tag{8.435}$$

In case of `ObjectFFRFreducedOrder`, this jacobian is computed as

$${}^0\mathbf{J}_{m,rot,alt}^{FFRFreduced} = \left[\mathbf{0},\ {}^{0r}\mathbf{A}\,{}^r\mathbf{G}_{local}\,,\ {}^{0r}\mathbf{A}\,\mathbf{W}^{-1}\sum_i w_i\,{}^r\tilde{\mathbf{p}}_{ref}^{(i)}\,{}^r\mathbf{J}_{pos,f}^{(i)}\right] \tag{8.436}$$

see also the descriptions given after Eq. (8.429) in the 'standard' approach.

EXAMPLE for marker on body 4, mesh nodes 10,11,12,13:

```
MarkerSuperElementRigid(bodyNumber = 4, meshNodeNumber = [10, 11, 12, 13], weightingFactors
= [0.25, 0.25, 0.25, 0.25], referencePosition=[0,0,0])
```

For detailed examples, see `TestModels`.

For examples on MarkerSuperElementRigid see Relevant Examples and TestModels with weblink:

- `CMSexampleCourse.py` (Examples/)
- `netgenSTLtest.py` (Examples/)
- `NGsolveCMStutorial.py` (Examples/)
- `NGsolveCraigBampton.py` (Examples/)
- `NGsolveFFRF.py` (Examples/)
- `NGsolveLinearFEM.py` (Examples/)
- `NGsolveModalAnalysis.py` (Examples/)
- `ObjectFFRFconvergenceTestBeam.py` (Examples/)
- `ObjectFFRFconvergenceTestHinge.py` (Examples/)
- `pendulumVerify.py` (Examples/)
- `serialRobotFlexible.py` (Examples/)
- `abaqusImportTest.py` (TestModels/)
- `ACFtest.py` (TestModels/)
- `linearFEMgenericODE2.py` (TestModels/)
- `linearFEMgenericODE2Test.py` (TestModels/)
- ...

8.9.12 MarkerKinematicTreeRigid

A position and orientation (rigid-body) marker attached to a kinematic tree. The marker is attached to the ObjectKinematicTree object and additionally needs a link number as well as a local position, similar to the SensorKinematicTree. The marker allows to attach loads (LoadForceVector and LoadTorqueVector) at arbitrary links or position. It also allows to attach connectors (e.g., spring dampers or actuators) to the kinematic tree. Finally, joint constraints can be attached, which allows for realization of closed loop structures. NOTE, however, that it is less efficient to attach many markers to a kinematic tree, therefor for forces or joint control use the structures available in kinematic tree whenever possible.

Additional information for MarkerKinematicTreeRigid:

- This `Marker` has/provides the following types = `Object`, `Body`, `Position`, `Orientation`

The item **MarkerKinematicTreeRigid** with type = 'KinematicTreeRigid' has the following parameters:

Name	type	size	default value	description
name	String		"	marker's unique name
objectNumber	ObjectIndex		invalid (-1)	body number to which marker is attached to
linkNumber	UInt		invalid (-1)	number of link in KinematicTree to which marker is attached to
localPosition	Vector3D	3	[0.,0.,0.]	local (link-fixed) position of marker at link n_l, using the link (n_l) coordinate system
visualization	VMarkerKinematicTreeRigid			parameters for visualization of item

The item VMarkerKinematicTreeRigid has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

8.9.12.1 DESCRIPTION of MarkerKinematicTreeRigid:

Information on input parameters:

input parameter	symbol	description see tables above
objectNumber	n_b	
linkNumber	n_l	
localPosition	${}^l\mathbf{b}$	

8.9.12.2 Marker quantities

More information will be added later. The marker computes jacobians according to `Jacobian` in `class Robot`.

For examples on MarkerKinematicTreeRigid see Relevant Examples and TestModels with weblink:

- `humanRobotInteraction.py` (Examples/)
- `openAIgymNLinkAdvanced.py` (Examples/)
- `reinforcementLearningRobot.py` (Examples/)
- `serialRobotKinematicTreeDigging.py` (Examples/)
- `stiffFlyballGovernorKT.py` (Examples/)
- `kinematicTreeConstraintTest.py` (TestModels/)

8.9.13 MarkerObjectODE2Coordinates

A Marker attached to all coordinates of an object (currently only body is possible), e.g. to apply special constraints or loads on all coordinates. The measured coordinates INCLUDE reference + current coordinates.

Additional information for MarkerObjectODE2Coordinates:

- This `Marker` has/provides the following types = `Object`, `Body`, `Coordinate`

The item **MarkerObjectODE2Coordinates** with type = 'ObjectODE2Coordinates' has the following parameters:

Name	type	size	default value	description
name	String		"	marker's unique name
objectNumber	ObjectIndex		invalid (-1)	body number to which marker is attached to
visualization	VMarkerObjectODE2Coordinates			parameters for visualization of item

The item VMarkerObjectODE2Coordinates has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

8.9.13.1 DESCRIPTION of MarkerObjectODE2Coordinates:

For examples on MarkerObjectODE2Coordinates see Relevant Examples and TestModels with weblink:

- `coordinateVectorConstraintGenericODE2.py` (TestModels/)

8.9.14 MarkerBodyCable2DShape

A special Marker attached to a 2D ANCF beam finite element with cubic interpolation and 8 coordinates.

Additional information for MarkerBodyCable2DShape:

- This `Marker` has/provides the following types = `Object`, `Body`, `Coordinate`

The item **MarkerBodyCable2DShape** with type = 'BodyCable2DShape' has the following parameters:

Name	type	size	default value	description
name	String		"	marker's unique name
bodyNumber	ObjectIndex		invalid (-1)	body number to which marker is attached to
numberOfSegments	PInt		3	number of number of segments; each segment is a line and is associated to a data (history) variable; must be same as in according contact element
verticalOffset	Real		0.	vertical offset from beam axis in positive (local) Y-direction; this offset accounts for consistent computation of positions and velocities at the surface of the beam
visualization	VMarkerBodyCable2DShape			parameters for visualization of item

The item VMarkerBodyCable2DShape has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

8.9.14.1 DESCRIPTION of MarkerBodyCable2DShape:

For examples on MarkerBodyCable2DShape see Relevant Examples and TestModels with weblink:

- `ANCFcontactCircle.py` (Examples/)
- `ANCFcontactCircle2.py` (Examples/)

- ANCFmovingRigidbody.py (Examples/)
- ANCFslidingJoint2D.py (Examples/)
- beltDriveALE.py (Examples/)
- beltDriveReevingSystem.py (Examples/)
- beltDrivesComparison.py (Examples/)
- sliderCrank3DwithANCFbeltDrive.py (Examples/)
- sliderCrank3DwithANCFbeltDrive2.py (Examples/)
- ANCFcontactCircleTest.py (TestModels/)
- ANCFcontactFrictionTest.py (TestModels/)
- ANCFmovingRigidBodyTest.py (TestModels/)
- ANCFslidingAndALEjointTest.py (TestModels/)

8.9.15 MarkerBodyCable2DCoordinates

A special Marker attached to the coordinates of a 2D ANCF beam finite element with cubic interpolation.

Additional information for MarkerBodyCable2DCoordinates:

- This `Marker` has/provides the following types = `Object`, `Body`, `Coordinate`

The item **MarkerBodyCable2DCoordinates** with type = 'BodyCable2DCoordinates' has the following parameters:

Name	type	size	default value	description
name	String		''	marker's unique name
bodyNumber	ObjectIndex		invalid (-1)	body number to which marker is attached to
visualization	VMarkerBodyCable2DCoordinates			parameters for visualization of item

The item VMarkerBodyCable2DCoordinates has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

8.9.15.1 DESCRIPTION of MarkerBodyCable2DCoordinates:

For examples on MarkerBodyCable2DCoordinates see Relevant Examples and TestModels with weblink:

- `ANCFmovingRigidbody.py` (Examples/)
- `ANCFslidingJoint2D.py` (Examples/)
- `ANCFslidingJoint2Drigid.py` (Examples/)
- `ANCFswitchingSlidingJoint2D.py` (Examples/)
- `ANCFmovingRigidBodyTest.py` (TestModels/)
- `modelUnitTests.py` (TestModels/)

8.10 Loads

A Load applies a (usually constant) force, torque, mass-proportional or generalized load onto Nodes or Objects via Markers. The requested `Marker` types need to be provided by the used Marker. The marker may provide more types than requested. For non-constant loads, use either a `load...UserFunction` or change the load in every step by means of a `preStepUserFunction` in the `MainSystem` (mbs).

8.10.1 LoadForceVector

Load with (3D) force vector; attached to position-based marker.

Additional information for LoadForceVector:

- Requested `Marker` type = `Position`
- **Short name** for Python = `Force`
- **Short name** for Python visualization object = `VForce`

The item **LoadForceVector** with type = 'ForceVector' has the following parameters:

Name	type	size	default value	description
name	String		''	load's unique name
markerNumber	MarkerIndex		invalid (-1)	marker's number to which load is applied
loadVector	Vector3D		[0.,0.,0.]	vector-valued load [SI:N]; in case of a user function, this vector is ignored
bodyFixed	Bool		False	if bodyFixed is true, the load is defined in body-fixed (local) coordinates, leading to a follower force; if false: global coordinates are used

loadVectorUserFunction	PyFunctionVector3D	mbs	ScalarVector3D 0	A Python function which defines the time-dependent load and replaces loadVector; see description below; NOTE that in static computations, the loadFactor is always 1 for forces computed by user functions (this means for the static computation, that a user function returning [t*5,t*1,0] corresponds to loadVector=[5,1,0] without a user function); NOTE that forces are drawn using the value of loadVector; thus the current values according to the user function are NOT shown in the render window; however, a sensor (SensorLoad) returns the user function force which is applied to the object; to draw forces with current user function values, use a graphicsDataUserFunction of a ground object
visualization	VLoadForceVector			parameters for visualization of item

The item VLoadForceVector has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

8.10.1.1 DESCRIPTION of LoadForceVector:

Information on input parameters:

input parameter	symbol	description see tables above
loadVector	$\mathbf{f}$	
loadVectorUserFunction	$UF \in \mathbb{R}^3$	

8.10.1.2 Details

The load vector acts on a body or node via the local (`bodyFixed = True`) or global coordinates of a body or at a node. The marker transforms the (translational) force via the according jacobian matrix of the object (or node) to object (or node) coordinates.

Userfunction: `loadVectorUserFunction(mbs, t, loadVector)`

A user function, which computes the force vector depending on time and object parameters, which is hereafter applied to object or node.

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs to which load belongs
`t`	Real	current time in mbs
`loadVector`	Vector3D	**f** copied from object; WARNING: this parameter does not work in combination with static computation, as it is changed by the solver over step time
`return value`	Vector3D	computed force vector

User function example:

```
from math import sin, cos, pi
def UFforce(mbs, t, loadVector):
    return [loadVector[0]*sin(t*10*2*pi),0,0]
```

For examples on LoadForceVector see Relevant Examples and TestModels with weblink:

- `interactiveTutorial.py` (Examples/)
- `NGsolveModalAnalysis.py` (Examples/)
- `pendulumVerify.py` (Examples/)
- `ROSMassPoint.py` (Examples/)
- `solutionViewerTest.py` (Examples/)
- `SpringDamperMassUserFunction.py` (Examples/)
- `ballBearningModel.py` (Examples/)
- `cartesianSpringDamper.py` (Examples/)
- `cartesianSpringDamperUserFunction.py` (Examples/)
- `chatGPTupdate.py` (Examples/)
- `chatGPTupdate2.py` (Examples/)
- `rigidBodyTutorial3.py` (Examples/)
- ...
- `perf3DRigidBodies.py` (TestModels/)
- `plotSensorTest.py` (TestModels/)
- `revoluteJointPrismaticJointTest.py` (TestModels/)
- ...

8.10.2 LoadTorqueVector

Load with (3D) torque vector; attached to rigidbody-based marker.

Additional information for LoadTorqueVector:

- Requested `Marker` type = `Orientation`
- **Short name** for Python = `Torque`
- **Short name** for Python visualization object = `VTorque`

The item **LoadTorqueVector** with type = 'TorqueVector' has the following parameters:

Name	type	size	default value	description
name	String		"	load's unique name
markerNumber	MarkerIndex		invalid (-1)	marker's number to which load is applied
loadVector	Vector3D		[0.,0.,0.]	vector-valued load [SI:N]; in case of a user function, this vector is ignored
bodyFixed	Bool		False	if bodyFixed is true, the load is defined in body-fixed (local) coordinates, leading to a follower torque; if false: global coordinates are used
loadVectorUserFunction	PyFunctionVector3DmbsScalarVector3D		0	A Python function which defines the time-dependent load and replaces loadVector; see description below; see also notes on loadFactor and drawing in LoadForceVector! Example for Python function: def f(mbs, t, loadVector): return [loadVector[0]*np.sin(t*10*2*3.1415),0,0]
visualization	VLoadTorqueVector			parameters for visualization of item

The item VLoadTorqueVector has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

8.10.2.1 DESCRIPTION of LoadTorqueVector:

Information on input parameters:

input parameter	symbol	description see tables above
loadVector	τ	
loadVectorUserFunction	UF $\in \mathbb{R}^3$	

8.10.2.2 Details

The torque vector acts on a body or node via the local (`bodyFixed = True`) or global coordinates of a body or at a node. The marker transforms the torque via the according jacobian matrix of the object (or node) to object (or node) coordinates.

Userfunction: `loadVectorUserFunction(mbs, t, loadVector)`

A user function, which computes the torque vector depending on time and object parameters, which is hereafter applied to object or node.

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs to which load belongs
`t`	Real	current time in mbs
`loadVector`	Vector3D	τ copied from object; WARNING: this parameter does not work in combination with static computation, as it is changed by the solver over step time
`return value`	Vector3D	computed torque vector

User function example:

```
from math import sin, cos, pi
def UFforce(mbs, t, loadVector):
    return [loadVector[0]*sin(t*10*2*pi),0,0]
```

For examples on LoadTorqueVector see Relevant Examples and TestModels with weblink:

- leggedRobot.py (Examples/)
- reevingSystem.py (Examples/)
- sliderCrank3DwithANCFbeltDrive2.py (Examples/)
- ballBearningModel.py (Examples/)
- chatGPTupdate.py (Examples/)
- chatGPTupdate2.py (Examples/)

- rigidBodyTutorial3.py (Examples/)
- ANCFcontactCircle.py (Examples/)
- ANCFcontactCircle2.py (Examples/)
- ANCFslidingJoint2D.py (Examples/)
- ANCFtestHalfcircle.py (Examples/)
- ANCFtests2.py (Examples/)
- ...
- ANCFbeltDrive.py (TestModels/)
- ANCFgeneralContactCircle.py (TestModels/)
- createFunctionsTest.py (TestModels/)
- ...

8.10.3 LoadMassProportional

Load attached to MarkerBodyMass marker, applying a 3D vector load (e.g. the vector [0,-g,0] is used to apply gravitational loading of size g in negative y-direction).

Additional information for LoadMassProportional:

- Requested `Marker` type = `Body` + `BodyMass`
- **Short name** for Python = `Gravity`
- **Short name** for Python visualization object = `VGravity`

The item **LoadMassProportional** with type = 'MassProportional' has the following parameters:

Name	type	size	default value	description
name	String		"	load's unique name
markerNumber	MarkerIndex		invalid (-1)	marker's number to which load is applied
loadVector	Vector3D		[0.,0.,0.]	vector-valued load [SI:N/kg = m/s^2]; typically, this will be the gravity vector in global coordinates; in case of a user function, this v is ignored
loadVectorUserFunction	PyFunctionVector3DmbsScalarVector3D 0			A Python function which defines the time-dependent load; see description below; see also notes on loadFactor and drawing in LoadForceVector!
visualization	VLoadMassProportional			parameters for visualization of item

The item VLoadMassProportional has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

8.10.3.1 DESCRIPTION of LoadMassProportional:

Information on input parameters:

input parameter	symbol	description see tables above
loadVector	**b**	

loadVectorUserFunction	UF $\in \mathbb{R}^3$	

8.10.3.2 Details

The load applies a (translational) and distributed load proportional to the distributed body's density. The marker of type `MarkerBodyMass` transforms the loadVector via an according jacobian matrix to object coordinates.

Userfunction: `loadVectorUserFunction(mbs, t, loadVector)`

A user function, which computes the mass proporitional load vector depending on time and object parameters, which is hereafter applied to object or node.

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs to which load belongs
`t`	Real	current time in mbs
`loadVector`	Vector3D	**b** copied from object; WARNING: this parameter does not work in combination with static computation, as it is changed by the solver over step time
`return value`	Vector3D	computed load vector

Example of user function: functionality same as in `LoadForceVector`

8.10.3.3 MINI EXAMPLE for LoadMassProportional

```
node = mbs.AddNode(NodePoint(referenceCoordinates = [1,0,0]))
body = mbs.AddObject(MassPoint(nodeNumber = node, physicsMass=2))
mMass = mbs.AddMarker(MarkerBodyMass(bodyNumber=body))
mbs.AddLoad(LoadMassProportional(markerNumber=mMass, loadVector=[0,0,-9.81]))

#assemble and solve system for default parameters
mbs.Assemble()
mbs.SolveDynamic()

#check result
exudynTestGlobals.testResult = mbs.GetNodeOutput(node, exu.OutputVariableType.Position)
[2]
#final z-coordinate of position shall be -g/2 due to constant acceleration with g=-9.81
#result independent of mass
```

For examples on LoadMassProportional see Relevant Examples and TestModels with weblink:

- CMSexampleCourse.py (Examples/)
- finiteSegmentMethod.py (Examples/)
- NGsolveCMStutorial.py (Examples/)
- NGsolvePostProcessingStresses.py (Examples/)
- ObjectFFRFconvergenceTestBeam.py (Examples/)
- ObjectFFRFconvergenceTestHinge.py (Examples/)
- pendulumGeomExactBeam2D.py (Examples/)
- pendulumVerify.py (Examples/)
- ALEANCFpipe.py (Examples/)
- ANCFmovingRigidbody.py (Examples/)
- ANCFslidingJoint2D.py (Examples/)
- ANCFslidingJoint2Drigid.py (Examples/)
- ...
- fourBarMechanismIftomm.py (TestModels/)
- rigidBody2Dtest.py (TestModels/)
- genericJointUserFunctionTest.py (TestModels/)
- ...

8.10.4 LoadCoordinate

Load with scalar value, which is attached to a coordinate-based marker; the load can be used e.g. to apply a force to a single axis of a body, a nodal coordinate of a finite element or a torque to the rotatory DOF of a rigid body.

Additional information for LoadCoordinate:

- Requested `Marker` type = `Coordinate`

The item **LoadCoordinate** with type = 'Coordinate' has the following parameters:

Name	type	size	default value	description
name	String		"	load's unique name
markerNumber	MarkerIndex		invalid (-1)	marker's number to which load is applied
load	Real		0.	scalar load [SI:N]; in case of a user function, this value is ignored
loadUserFunction	PyFunctionMbsScalar2		0	A Python function which defines the time-dependent load and replaces the load; see description below; see also notes on loadFactor and drawing in LoadForceVector!
visualization	VLoadCoordinate			parameters for visualization of item

The item VLoadCoordinate has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

8.10.4.1 DESCRIPTION of LoadCoordinate:

Information on input parameters:

input parameter	symbol	description see tables above
load	f	
loadUserFunction	$UF \in \mathbb{R}$	

8.10.4.2 Details

The scalar `load` is applied on a coordinate defined by a Marker of type 'Coordinate', e.g., `MarkerNodeCoordinate`. This can be used to create simple 1D problems, or to simply apply a translational force on a Node or even a torque on a rotation coordinate (but take care for its meaning).

Userfunction: `loadUserFunction(mbs, t, load)`

A user function, which computes the scalar load depending on time and the object's `load` parameter.

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs to which load belongs
`t`	Real	current time in mbs
`load`	Real	**b** copied from object; WARNING: this parameter does not work in combination with static computation, as it is changed by the solver over step time
`return value`	Real	computed load

User function example:

```
    from math import sin, cos, pi
    #this example uses the object's stored parameter load to compute a time-dependent
load
    def UFload(mbs, t, load):
        return load*sin(10*(2*pi)*t)

    n0=mbs.AddNode(Point())
    nodeMarker = mbs.AddMarker(MarkerNodeCoordinate(nodeNumber=n0,coordinate=0))
    mbs.AddLoad(LoadCoordinate(markerNumber = markerCoordinate,
                               load = 10,
                               loadUserFunction = UFload))
```

For examples on LoadCoordinate see Relevant Examples and TestModels with weblink:

- `beltDriveALE.py` (Examples/)
- `beltDriveReevingSystem.py` (Examples/)
- `ComputeSensitivitiesExample.py` (Examples/)
- `coordinateSpringDamper.py` (Examples/)
- `geneticOptimizationSliderCrank.py` (Examples/)
- `lavalRotor2Dtest.py` (Examples/)
- `massSpringFrictionInteractive.py` (Examples/)
- `minimizeExample.py` (Examples/)
- `nMassOscillator.py` (Examples/)
- `nMassOscillatorEigenmodes.py` (Examples/)
- `nMassOscillatorInteractive.py` (Examples/)
- `openAIgymInterfaceTest.py` (Examples/)

- ...
- ANCFslidingAndALEjointTest.py (TestModels/)
- complexEigenvaluesTest.py (TestModels/)
- contactCoordinateTest.py (TestModels/)
- ...

8.11 Sensors

A Sensor is used to measure quantities during simulation. Sensors may be attached to Nodes, Objects, Markers or Loads. Sensor values may be directly read via mbs or can be continuously written to files or SensorRecorder during simulation. The exudyn.plot Python utility function PlotSensor(...) can be conveniently used to show Sensor values over time.

8.11.1 SensorNode

A sensor attached to a ODE2 or ODE1 node. The sensor measures OutputVariables and outputs values into a file, showing per line [time, sensorValue[0], sensorValue[1], ...]. Use SensorUserFunction to modify sensor results (e.g., transforming to other coordinates) and writing to file.

The item **SensorNode** with type = 'Node' has the following parameters:

Name	type	size	default value	description
name	String		''	sensor's unique name
nodeNumber	NodeIndex		invalid (-1)	node number to which sensor is attached to
writeToFile	Bool		True	True: write sensor output to file; flag is ignored (interpreted as False), if fileName=''
fileName	String		''	directory and file name for sensor file output; default: empty string generates sensor + sensorNumber + outputVariableType; directory will be created if it does not exist
outputVariableType	OutputVariableType		OutputVariableType::_None	OutputVariableType for sensor
storeInternal	Bool		False	true: store sensor data in memory (faster, but may consume large amounts of memory); false: internal storage not available
visualization	VSensorNode			parameters for visualization of item

The item VSensorNode has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

8.11.1.1 DESCRIPTION of SensorNode:

For examples on SensorNode see Relevant Examples and TestModels with weblink:

- ANCFALEtest.py (Examples/)
- beltDriveALE.py (Examples/)
- beltDriveReevingSystem.py (Examples/)
- beltDrivesComparison.py (Examples/)
- craneReevingSystem.py (Examples/)
- flexiblePendulumANCF.py (Examples/)
- geneticOptimizationSliderCrank.py (Examples/)
- gyroStability.py (Examples/)
- HydraulicActuator2Arms.py (Examples/)
- HydraulicActuatorStaticInitialization.py (Examples/)
- HydraulicsUserFunction.py (Examples/)
- kinematicTreeAndMBS.py (Examples/)
- ...
- ACFtest.py (TestModels/)
- ANCFbeltDrive.py (TestModels/)
- ANCFgeneralContactCircle.py (TestModels/)
- ...

8.11.2 SensorObject

A sensor attached to any object except bodies (connectors, constraint, spring-damper, etc). As a difference to other SensorBody, the connector sensor measures quantities without a local position. The sensor measures OutputVariable and outputs values into a file, showing per line [time, sensorValue[0], sensorValue[1], ...]. Use SensorUserFunction to modify sensor results (e.g., transforming to other coordinates) and writing to file.

The item **SensorObject** with type = 'Object' has the following parameters:

Name	type	size	default value	description
name	String		"	sensor's unique name
objectNumber	ObjectIndex		invalid (-1)	object (e.g. connector) number to which sensor is attached to
writeToFile	Bool		True	True: write sensor output to file; flag is ignored (interpreted as False), if fileName="
fileName	String		"	directory and file name for sensor file output; default: empty string generates sensor + sensorNumber + outputVariableType; directory will be created if it does not exist
outputVariableType	OutputVariableType		OutputVariableType::_None	OutputVariableType for sensor
storeInternal	Bool		False	true: store sensor data in memory (faster, but may consume large amounts of memory); false: internal storage not available
visualization	VSensorObject			parameters for visualization of item

The item VSensorObject has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown; sensors can be shown at the position assiciated with the object - note that in some cases, there might be no such position (e.g. data object)!

8.11.2.1 DESCRIPTION of SensorObject:

For examples on SensorObject see Relevant Examples and TestModels with weblink:

- ballBearningModel.py (Examples/)
- beltDriveALE.py (Examples/)
- beltDriveReevingSystem.py (Examples/)
- bicycleIftommBenchmark.py (Examples/)
- camFollowerExample.py (Examples/)
- ComputeSensitivitiesExample.py (Examples/)
- craneReevingSystem.py (Examples/)
- HydraulicActuator2Arms.py (Examples/)
- HydraulicActuatorStaticInitialization.py (Examples/)
- HydraulicsUserFunction.py (Examples/)
- leggedRobot.py (Examples/)
- lugreFrictionTest.py (Examples/)
- ...
- carRollingDiscTest.py (TestModels/)
- complexEigenvaluesTest.py (TestModels/)
- contactSphereSphereTestEAPM.py (TestModels/)
- ...

8.11.3 SensorBody

A sensor attached to a body-object with local position ${}^{b}\mathbf{b}$. As a difference to SensorObject, the body sensor needs a local position at which the sensor is attached to. The sensor measures OutputVariableBody and outputs values into a file, showing per line [time, sensorValue[0], sensorValue[1], ...]. Use SensorUserFunction to modify sensor results (e.g., transforming to other coordinates) and writing to file.

The item **SensorBody** with type = 'Body' has the following parameters:

Name	type	size	default value	description
name	String		"	sensor's unique name
bodyNumber	ObjectIndex		invalid (-1)	body (=object) number to which sensor is attached to
localPosition	Vector3D	3	[0.,0.,0.]	local (body-fixed) body position of sensor
writeToFile	Bool		True	True: write sensor output to file; flag is ignored (interpreted as False), if fileName="
fileName	String		"	directory and file name for sensor file output; default: empty string generates sensor + sensorNumber + outputVariableType; directory will be created if it does not exist
outputVariableType	OutputVariableType		OutputVariableType::_None	OutputVariableType for sensor
storeInternal	Bool		False	true: store sensor data in memory (faster, but may consume large amounts of memory); false: internal storage not available
visualization	VSensorBody			parameters for visualization of item

The item VSensorBody has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

8.11.3.1 DESCRIPTION of SensorBody:

Information on input parameters:

input parameter	symbol	description see tables above

localPosition	${}^{b}\mathbf{b}$	

For examples on SensorBody see Relevant Examples and TestModels with weblink:

- `ballBearningModel.py` (Examples/)
- `beltDriveALE.py` (Examples/)
- `beltDriveReevingSystem.py` (Examples/)
- `bicycleIftommBenchmark.py` (Examples/)
- `bungeeJump.py` (Examples/)
- `camFollowerExample.py` (Examples/)
- `cartesianSpringDamperUserFunction.py` (Examples/)
- `chainDriveExample.py` (Examples/)
- `contactCurvePolynomial.py` (Examples/)
- `contactCurveWithLongCurve.py` (Examples/)
- `finiteSegmentMethod.py` (Examples/)
- `flexiblePendulumANCF.py` (Examples/)
- ...
- `ANCFoutputTest.py` (TestModels/)
- `carRollingDiscTest.py` (TestModels/)
- `complexEigenvaluesTest.py` (TestModels/)
- ...

8.11.4 SensorSuperElement

A sensor attached to a SuperElement-object with mesh node number. As a difference to other ObjectSensors, the SuperElement sensor has a mesh node number at which the sensor is attached to. The sensor measures OutputVariableSuperElement and outputs values into a file, showing per line [time, sensorValue[0], sensorValue[1], ...]. Use SensorUserFunction to modify sensor results (e.g., transforming to other coordinates) and writing to file.

The item **SensorSuperElement** with type = 'SuperElement' has the following parameters:

Name	type	size	default value	description
name	String		"	sensor's unique name
bodyNumber	ObjectIndex		invalid (-1)	body (=object) number to which sensor is attached to
meshNodeNumber	UInt		invalid (-1)	mesh node number, which is a local node number with in the object (starting with 0); the node number may represent a real Node in mbs, or may be virtual and reconstructed from the object coordinates such as in ObjectFFRFreducedOrder
writeToFile	Bool		True	True: write sensor output to file; flag is ignored (interpreted as False), if fileName="
fileName	String		"	directory and file name for sensor file output; default: empty string generates sensor + sensorNumber + outputVariableType; directory will be created if it does not exist
outputVariableType	OutputVariableType		OutputVariableType::_None	OutputVariableType for sensor, based on the output variables available for the mesh nodes (see special section for super element output variables, e.g, in ObjectFFRFreducedOrder, Section 8.3.4.2)
storeInternal	Bool		False	true: store sensor data in memory (faster, but may consume large amounts of memory); false: internal storage not available
visualization	VSensorSuperElement			parameters for visualization of item

The item VSensorSuperElement has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

8.11.4.1 DESCRIPTION of SensorSuperElement:

For examples on SensorSuperElement see Relevant Examples and TestModels with weblink:

- CMSexampleCourse.py (Examples/)
- NGsolveCraigBampton.py (Examples/)
- NGsolvePostProcessingStresses.py (Examples/)
- ObjectFFRFconvergenceTestBeam.py (Examples/)
- objectFFRFreducedOrderNetgen.py (Examples/)
- pendulumVerify.py (Examples/)
- abaqusImportTest.py (TestModels/)
- NGsolveCMStest.py (TestModels/)
- objectFFRFreducedOrderAccelerations.py (TestModels/)
- objectFFRFreducedOrderStressModesTest.py (TestModels/)
- objectFFRFreducedOrderTest.py (TestModels/)
- objectFFRFTest.py (TestModels/)
- objectFFRFTest2.py (TestModels/)
- objectGenericODE2Test.py (TestModels/)
- perfObjectFFRFreducedOrder.py (TestModels/)
- ...

8.11.5 SensorKinematicTree

A sensor attached to a KinematicTree with local position ${}^{b}\mathbf{b}$ and link number n_l. As a difference to SensorBody, the KinematicTree sensor needs a local position and a link number, which defines the sub-body at which the sensor values are evaluated. The local position is given in sub-body (link) local coordinates. The sensor measures OutputVariableKinematicTree and outputs values into a file, showing per line [time, sensorValue[0], sensorValue[1], ...]. Use SensorUserFunction to modify sensor results (e.g., transforming to other coordinates) and writing to file.

The item **SensorKinematicTree** with type = 'KinematicTree' has the following parameters:

Name	type	size	default value	description
name	String		"	sensor's unique name
objectNumber	ObjectIndex		invalid (-1)	object number of KinematicTree to which sensor is attached to
linkNumber	UInt		invalid (-1)	number of link in KinematicTree to measure quantities
localPosition	Vector3D	3	[0.,0.,0.]	local (link-fixed) position of sensor, defined in link (n_l) coordinate system
writeToFile	Bool		True	True: write sensor output to file; flag is ignored (interpreted as False), if fileName="
fileName	String		"	directory and file name for sensor file output; default: empty string generates sensor + sensorNumber + outputVariableType; directory will be created if it does not exist
outputVariableType	OutputVariableType		OutputVariableType::_None	OutputVariableType for sensor
storeInternal	Bool		False	true: store sensor data in memory (faster, but may consume large amounts of memory); false: internal storage not available
visualization	VSensorKinematicTree			parameters for visualization of item

The item VSensorKinematicTree has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

8.11.5.1 DESCRIPTION of SensorKinematicTree:

Information on input parameters:

input parameter	symbol	description see tables above
linkNumber	n_l	
localPosition	${}^{l}\mathbf{b}$	

For examples on SensorKinematicTree see Relevant Examples and TestModels with weblink:

- openAIgymNLinkAdvanced.py (Examples/)
- openAIgymNLinkContinuous.py (Examples/)
- reinforcementLearningRobot.py (Examples/)
- serialRobotInverseKinematics.py (Examples/)
- serialRobotKinematicTreeDigging.py (Examples/)
- stiffFlyballGovernorKT.py (Examples/)
- kinematicTreeAndMBStest.py (TestModels/)
- kinematicTreeConstraintTest.py (TestModels/)

8.11.6 SensorMarker

A sensor attached to a marker. The sensor measures the selected marker values and outputs values into a file, showing per line [time, sensorValue[0], sensorValue[1], ...]. Depending on markers, it can measure Coordinates (MarkerNodeCoordinate), Position and Velocity (MarkerXXXPosition), Position, Velocity, Rotation and AngularVelocityLocal (MarkerXXXRigid). Note that marker values are only available for the current configuration. Use SensorUserFunction to modify sensor results (e.g., transforming to other coordinates) and writing to file

The item **SensorMarker** with type = 'Marker' has the following parameters:

Name	type	size	default value	description
name	String		"	sensor's unique name
markerNumber	MarkerIndex		invalid (-1)	marker number to which sensor is attached to
writeToFile	Bool		True	True: write sensor output to file; flag is ignored (interpreted as False), if fileName="
fileName	String		"	directory and file name for sensor file output; default: empty string generates sensor + sensorNumber + outputVariableType; directory will be created if it does not exist
outputVariableType	OutputVariableType		OutputVariableType::_None	OutputVariableType for sensor; output variables are only possible according to markertype, see general description of SensorMarker
storeInternal	Bool		False	true: store sensor data in memory (faster, but may consume large amounts of memory); false: internal storage not available
visualization	VSensorMarker			parameters for visualization of item

The item VSensorMarker has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

8.11.6.1 DESCRIPTION of SensorMarker:

For examples on SensorMarker see Relevant Examples and TestModels with weblink:

- bicycleIftommBenchmark.py (Examples/)
- NGsolveCMStutorial.py (Examples/)
- NGsolveModalAnalysis.py (Examples/)
- ObjectFFRFconvergenceTestHinge.py (Examples/)
- pendulumVerify.py (Examples/)
- ROSMobileManipulator.py (Examples/)
- distanceSensor.py (TestModels/)
- pendulumFriction.py (TestModels/)
- plotSensorTest.py (TestModels/)
- reevingSystemSpringsTest.py (TestModels/)

8.11.7 SensorLoad

A sensor attached to a load. The sensor measures the load values and outputs values into a file, showing per line [time, sensorValue[0], sensorValue[1], ...]. Use SensorUserFunction to modify sensor results (e.g., transforming to other coordinates) and writing to file.

The item **SensorLoad** with type = 'Load' has the following parameters:

Name	type	size	default value	description
name	String		"	sensor's unique name
loadNumber	LoadIndex		invalid (-1)	load number to which sensor is attached to
writeToFile	Bool		True	True: write sensor output to file; flag is ignored (interpreted as False), if fileName="
fileName	String		"	directory and file name for sensor file output; default: empty string generates sensor + sensorNumber + outputVariableType; directory will be created if it does not exist
storeInternal	Bool		False	true: store sensor data in memory (faster, but may consume large amounts of memory); false: internal storage not available
visualization	VSensorLoad			parameters for visualization of item

The item VSensorLoad has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown; sensor visualization CURRENTLY NOT IMPLEMENTED

8.11.7.1 DESCRIPTION of SensorLoad:

For examples on SensorLoad see Relevant Examples and TestModels with weblink:

- `leggedRobot.py` (Examples/)
- `nMassOscillator.py` (Examples/)
- `nMassOscillatorInteractive.py` (Examples/)
- `serialRobotInteractiveLimits.py` (Examples/)
- `simulateInteractively.py` (Examples/)

- sliderCrank3DwithANCFbeltDrive2.py (Examples/)
- movingGroundRobotTest.py (TestModels/)
- plotSensorTest.py (TestModels/)
- rightAngleFrame.py (TestModels/)
- serialRobotTest.py (TestModels/)
- springDamperUserFunctionTest.py (TestModels/)

8.11.8 SensorUserFunction

A sensor defined by a user function. The sensor is intended to collect sensor values of a list of given sensors and recombine the output into a new value for output or control purposes. It is also possible to use this sensor without any dependence on other sensors in order to generate output for, e.g., any quantities in mbs or solvers.

The item **SensorUserFunction** with type = 'UserFunction' has the following parameters:

Name	type	size	default value	description
name	String		"	sensor's unique name
sensorNumbers	ArraySensorIndex		[]	optional list of n sensor numbers for use in user function
factors	Vector		[]	optional list of m factors which can be used, e.g., for weighting sensor values
writeToFile	Bool		True	True: write sensor output to file; flag is ignored (interpreted as False), if fileName="
fileName	String		"	directory and file name for sensor file output; default: empty string generates sensor + sensorNumber + outputVariableType; directory will be created if it does not exist
sensorUserFunction	PyFunctionVectorMbsScalarArrayIndexVectorConfiguration		0	A Python function which defines the time-dependent user function, which usually evaluates one or several sensors and computes a new sensor value, see example
storeInternal	Bool		False	true: store sensor data in memory (faster, but may consume large amounts of memory); false: internal storage not available
visualization	VSensorUserFunction			parameters for visualization of item

The item VSensorUserFunction has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown; sensor visualization CURRENTLY NOT IMPLEMENTED

8.11.8.1 DESCRIPTION of SensorUserFunction:

Information on input parameters:

input parameter	symbol	description see tables above
sensorNumbers	$\mathbf{n}_s = [s_0, \ldots, s_n]^T$	
factors	$\mathbf{f}_s = [f_0, \ldots, f_m]^T$	

The sensor collects data via a user function, which completely describes the output itself. Note that the sensorNumbers and factors need to be consistent. The return value of the user function is a list of `float` numbers which cast to a `std::vector` in pybind. This list can have arbitrary dimension, but should be kept constant during simulation.

Userfunction: `sensorUserFunction(mbs, t, sensorNumbers, factors, configuration)`

A user function, which computes a sensor output from other sensor outputs (or from generic time dependent functions). The configuration in general will be the exudyn.ConfigurationType.Current, but others could be used as well except for SensorMarker. The user function arguments are as follows:

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs to which object belongs
`t`	Real	current time in mbs
`sensorNumbers`	Array $\in \mathbb{N}^n$	list of sensor numbers
`factors`	Vector $\in \mathbb{R}^n$	list of factors that can be freely used for the user function
`configuration`	exudyn.ConfigurationType	usually the exudyn.ConfigurationType.Current, but could also be different in user defined functions.
`return value`	Vector $\in \mathbb{R}^{n_r}$	returns list or numpy array of sensor output values; size n_r is implicitly defined by the returned list and may not be changed during simulation.

User function example:

```
import exudyn as exu
from exudyn.itemInterface import *
from math import pi, atan2
SC = exu.SystemContainer()
mbs = SC.AddSystem()
node = mbs.AddNode(NodePoint(referenceCoordinates = [1,1,0],
                             initialCoordinates=[0,0,0],
                             initialVelocities=[0,-1,0]))
mbs.AddObject(MassPoint(nodeNumber = node, physicsMass=1))

sNode = mbs.AddSensor(SensorNode(nodeNumber=node, fileName='solution/sensorTest.txt'
,
                                 outputVariableType=exu.OutputVariableType.Position))
```

```
#user function for sensor, convert position into angle:
def UFsensor(mbs, t, sensorNumbers, factors, configuration):
    val = mbs.GetSensorValues(sensorNumbers[0]) #x,y,z
    phi = atan2(val[1],val[0]) #compute angle from x,y: atan2(y,x)
    return [factors[0]*phi] #return angle in degree

sUser = mbs.AddSensor(SensorUserFunction(sensorNumbers=[sNode], factors=[180/pi],
                                         fileName='solution/sensorTest2.txt',
                                         sensorUserFunction=UFsensor))

#assemble and solve system for default parameters
mbs.Assemble()
mbs.SolveDynamic()

if False:
    from exudyn.plot import PlotSensor
    PlotSensor(mbs, [sNode, sNode, sUser], [0, 1, 0])
```

For examples on SensorUserFunction see Relevant Examples and TestModels with weblink:

- bicycleIftommBenchmark.py (Examples/)
- kinematicTreeAndMBS.py (Examples/)
- lugreFrictionODE1.py (Examples/)
- lugreFrictionTest.py (Examples/)
- pendulumIftommBenchmark.py (Examples/)
- distanceSensor.py (TestModels/)
- fourBarMechanismIftomm.py (TestModels/)
- sensorUserFunctionTest.py (TestModels/)

Chapter 9

Structures and Settings

9.1 Structures for structural elements

This section includes data structures for structural elements, such as beams (and plates in future). These classes are used as interface between Python libraries for structural elements and Exudyn internal classes.

9.1.0.1 PyBeamSection

Data structure for definition of 2D and 3D beam (cross) section mechanical properties. The beam has local coordinates, in which X represents the beam centerline (beam axis) coordinate, being the neutral fiber w.r.t. bending; Y and Z are the local cross section coordinates. Note that most elements do not accept all parameters, which results in an error if those parameters (e.g., stiffness parameters) are non-zero.
PyBeamSection has the following items:

Name	type / function return type	size	default value / function args	description
dampingMatrix	Matrix6D		np.zeros((6,6))	${}^{c}\mathbf{D} \in \mathbb{R}^{6\times6}$ [SI:Nsm^2, Nsm and Ns (mixed)] sectional linear damping matrix related to $\begin{bmatrix} {}^{c}\mathbf{n} \\ {}^{c}\mathbf{m} \end{bmatrix} = {}^{c}\mathbf{D} \begin{bmatrix} {}^{c}\dot{\boldsymbol{\varepsilon}} \\ {}^{c}\dot{\boldsymbol{\kappa}} \end{bmatrix}$; note that this damping models is highly simplified and usually, it cannot be derived from material parameters; however, it can be used to adjust model damping to observed damping behavior. Set with list of lists or numpy array.
inertia	Matrix3D		[[0,0,0], [0,0,0], [0,0,0]]	${}^{c}\mathbf{J} \in \mathbb{R}^{3\times3}$ [SI:kg m^2] sectional inertia for shear-deformable beams. Set with list of lists or numpy array.
massPerLength	UReal		0.	ρA [SI:kg/m] mass per unit length of the beam

stiffnessMatrix	Matrix6D		np.zeros((6,6))	${}^c\mathbf{C} \in \mathbb{R}^{6\times6}$ [SI:Nm^2, Nm and N (mixed)] sectional stiffness matrix related to $\begin{bmatrix} {}^c\mathbf{n} \\ {}^c\mathbf{m} \end{bmatrix} = {}^c\mathbf{C} \begin{bmatrix} {}^c\boldsymbol{\varepsilon} \\ {}^c\boldsymbol{\kappa} \end{bmatrix}$ with sectional normal force ${}^c\mathbf{n}$, torque ${}^c\mathbf{m}$, strain ${}^c\boldsymbol{\varepsilon}$ and curvature ${}^c\boldsymbol{\kappa}$, all quantities expressed in the cross section frame c. Set with list of lists or numpy array.

9.1.0.2 BeamSectionGeometry

Data structure for definition of 2D and 3D beam (cross) section geometrical properties. Used for visualization and contact.
BeamSectionGeometry has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
crossSectionRadiusY	UReal		0.	c_Y [SI:m] Y radius for circular cross section
crossSectionRadiusZ	UReal		0.	c_Z [SI:m] Z radius for circular cross section
crossSectionType	CrossSectionType		CrossSectionType::Polygon	Type of cross section: Polygon, Circular, etc.
polygonalPoints	Vector2DList			$\mathbf{p}\_pg$ [SI: (m,m)] list of polygonal (Y, Z) points in local beam cross section coordinates, defined in positive rotation direction

9.2 Simulation settings

This section includes hierarchical structures for simulation settings, e.g., time integration, static solver, Newton iteration and solution file export.

9.2.0.1 SolutionSettings

General settings for exporting the solution (results) of a simulation.
SolutionSettings has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
appendToFile	bool		False	flag (true/false); if true, solution and solverInformation is appended to existing file (otherwise created); in BINARY mode, files are always replaced and this parameter is ineffective!

binarySolutionFile	bool		False	if true, the solution file is written in binary format for improved speed and smaller file sizes; setting outputPrecision >= 8 uses double (8 bytes), otherwise float (4 bytes) is used; note that appendToFile is ineffective and files are always replaced without asking! If not provided, file ending will read .sol in case of binary files and .txt in case of text files
coordinatesSolutionFileName	FileName		'coordinatesSolution'	filename and (relative) path of solution file (coordinatesSolutionFile) containing all multibody system coordinates versus time; directory will be created if it does not exist; character encoding of string is up to your filesystem, but for compatibility, it is recommended to use letters, numbers and '_' only; filename ending will be added automatically if not provided: .txt in case of text mode and .sol in case of binary solution files (binarySolutionFile=True)
exportAccelerations	bool		True	add ODE2 accelerations to solution file (coordinatesSolutionFile)
exportAlgebraicCoordinates	bool		True	add algebraicCoordinates (=Lagrange multipliers) to solution file (coordinatesSolutionFile)
exportDataCoordinates	bool		True	add DataCoordinates to solution file (coordinatesSolutionFile)
exportODE1Velocities	bool		True	add coordinatesODE1_t to solution file (coordinatesSolutionFile)
exportVelocities	bool		True	add ODE2 velocities to solution file (coordinatesSolutionFile)
flushFilesDOF	PInt		10000	number of DOF, above which solution file (coordinatesSolutionFile) buffers are always flushed, irrespectively of whether flushFilesImmediately is set True or False (see also flushFilesImmediately); for larger files, writing takes so much time that flushing does not add considerable time

flushFilesImmediately	bool		False	flush file buffers after every solution period written (coordinatesSolutionFile and sensor files); if set False, the output is written through a buffer, which is highly efficient, but during simulation, files may be always in an incomplete state; if set True, this may add a large amount of CPU time as the process waits until files are really written to hard disc (especially for simulation of small scale systems, writing 10.000s of time steps; at least 5us per step/file, depending on hardware)
outputPrecision	UInt		10	precision for floating point numbers written to solution and sensor files
recordImagesInterval	Real		-1.	record frames (images) during solving: amount of time to wait until next image (frame) is recorded; set recordImages = -1. if no images shall be recorded; set, e.g., recordImages = 0.01 to record an image every 10 milliseconds (requires that the time steps / load steps are sufficiently small!); for file names, etc., see VisualizationSettings.exportImages
restartFileName	FileName		'restartFile.txt'	filename and (relative) path of text file for storing solution after every restartWritePeriod if writeRestartFile=True; backup file is created with ending .bck, which should be used if restart file is crashed; use Python utility function InitializeFromRestartFile(...) to consistently restart
restartWritePeriod	UReal		0.01	time span (period), determines how often the restart file is updated; this should be often enough to enable restart without too much loss of data; too low values may influence performance
sensorsAppendToFile	bool		False	flag (true/false); if true, sensor output is appended to existing file (otherwise created) or in case of internal storage, it is appended to existing currently stored data; this allows storing sensor values over different simulations

sensorsStoreAndWriteFiles	bool		True	flag (true/false); if false, no sensor files will be created and no sensor data will be stored; this may be advantageous for benchmarking as well as for special solvers which should not overwrite existing results (e.g. ComputeODE2Eigenvalues); settings this value to False may cause problems if sensors are required to perform operations which are needed e.g. in UserSensors as input of loads, etc.
sensorsWriteFileFooter	bool		False	flag (true/false); if true, file footer is written for sensor output (turn off, e.g. for multiple runs of time integration)
sensorsWriteFileHeader	bool		True	flag (true/false); if true, file header is written for sensor output (turn off, e.g. for multiple runs of time integration)
sensorsWritePeriod	UReal		0.01	time span (period), determines how often the sensor output is written to file or internal storage during a simulation
solutionInformation	String		”	special information added to header of solution file (e.g. parameters and settings, modes, ...); character encoding my be UTF-8, restricted to characters in Section 10.5, but for compatibility, it is recommended to use ASCII characters only (95 characters, see wiki)
solutionWritePeriod	UReal		0.01	time span (period), determines how often the solution file (coordinatesSolutionFile) is written during a simulation
solverInformationFileName	FileName		’solverInformation.txt’	filename and (relative) path of text file showing detailed information during solving; detail level according to yourSolver.verboseModeFile; if solutionSettings.appendToFile is true, the information is appended in every solution step; directory will be created if it does not exist; character encoding of string is up to your filesystem, but for compatibility, it is recommended to use letters, numbers and ’_’ only
writeFileFooter	bool		True	flag (true/false); if true, information at end of simulation is written: convergence, total solution time, statistics
writeFileHeader	bool		True	flag (true/false); if true, file header is written (turn off, e.g. for multiple runs of time integration)

writeInitialValues	bool		True	flag (true/false); if true, initial values are exported for the start time; applies to coordinatesSolution and sensor files; this may not be wanted in the append file mode if the initial values are identical to the final values of a previous computation
writeRestartFile	bool		False	flag (true/false), which determines if restart file is written regularly, see restartFileName for details
writeSolutionToFile	bool		True	flag (true/false), which determines if (global) solution vector is written to the solution file (coordinatesSolutionFile); standard quantities that are written are: solution is written as displacements and coordinatesODE1; for additional coordinates in the solution file, see the options below

9.2.0.2 NumericalDifferentiationSettings

Settings for numerical differentiation of a function (needed for computation of numerical jacobian e.g. in implizit integration).
NumericalDifferentiationSettings has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
addReferenceCoordinatesToEpsilon	bool		False	True: for the size estimation of the differentiation parameter, the reference coordinate q^{Ref}_i is added to ODE2 coordinates –> see; False: only the current coordinate is used for size estimation of the differentiation parameter
doSystemWideDifferentiation	bool		False	True: system wide differentiation (e.g. all ODE2 equations w.r.t. all ODE2 coordinates); False: only local (object) differentiation
forAE	bool		False	flag (true/false); false = perform direct computation of jacobian for algebraic equations (AE), true = use numerical differentiation; as there must always exist an analytical implemented jacobian for AE, 'true' should only be used for verification

forODE2	bool		False	flag (true/false); false = perform direct computation (e.g., using autodiff) of jacobian for ODE2 equations, true = use numerical differentiation; numerical differentiation is less efficient and may lead to numerical problems, but may smoothen problems of analytical derivatives; sometimes the analytical derivative may neglect terms
forODE2connectors	bool		False	flag (true/false); false: if also forODE2==false, perform direct computation of jacobian for ODE2 terms for connectors; else: use numerical differentiation; NOTE: THIS FLAG IS FOR DEVELOPMENT AND WILL BE ERASED IN FUTURE
jacobianConnectorDerivative	bool		True	True: for analytic Jacobians of connectors, the Jacobian derivative is computed, causing additional CPU costs and not beeing available for all connectors or markers (thus switching to numerical differentiation); False: Jacobian derivative is neglected in analytic Jacobians (but included in numerical Jacobians), which often has only minor influence on convergence
minimumCoordinateSize	UReal		1e-2	minimum size of coordinates in relative differentiation parameter
relativeEpsilon	UReal		1e-7	relative differentiation parameter epsilon; the numerical differentiation parameter ε follows from the formula (ε = ε_relative $* max(q\_min, \|q\_i + [q^{Ref}\_i]\|)$, with ε_relative=relativeEpsilon, q_min =minimumCoordinateSize, q_i is the current coordinate which is differentiated, and $qRef\_i$ is the reference coordinate of the current coordinate

9.2.0.3 DiscontinuousSettings

Settings for discontinuous iterations, as in contact, friction, plasticity and general switching phenomena.
DiscontinuousSettings has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
ignoreMaxIterations	bool		True	continue solver if maximum number of discontinuous (post Newton) iterations is reached (ignore tolerance)

iterationTolerance	UReal		1	absolute tolerance for discontinuous (post Newton) iterations; the errors represent absolute residuals and can be quite high
maxIterations	UInt		5	maximum number of discontinuous (post Newton) iterations
useRecommendedStepSize	bool		True	some objects (contact-related) provide a recommendedStepSize; if True, this recommendation is used, but may lead to very small step sizes and solver could fail if restrictions are too hard; set to False to ignore this recommendation

9.2.0.4 NewtonSettings

Settings for Newton method used in static or dynamic simulation.
NewtonSettings has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
numericalDifferentiation	NumericalDifferentiationSettings			numerical differentiation parameters for numerical jacobian (e.g. Newton in static solver or implicit time integration)
absoluteTolerance	UReal		1e-10	absolute tolerance of residual for Newton (needed e.g. if residual is fulfilled right at beginning); condition: sqrt(q*q)/numberOfCoordinates <= absoluteTolerance
adaptInitialResidual	bool		True	flag (true/false); false = standard; True: if initialResidual is very small (or zero), it may increase significantely in the first Newton iteration; to achieve relativeTolerance, the initialResidual will by updated by a higher residual within the first Newton iteration
maximumSolutionNorm	UReal		1e38	this is the maximum allowed value for solutionU.L2NormSquared() which is the square of the square norm (i.e., value=u_1^2+u_2^2+...), and solutionV/A...; if the norm of solution vectors is larger, Newton method is stopped; the default value is chosen such that it would still work for single precision numbers (float)
maxIterations	UInt		25	maximum number of iterations (including modified + restart Newton iterations); after that total number of iterations, the static/dynamic solver refines the step size or stops with an error

maxModifiedNewtonIterations	UInt		8	maximum number of iterations for modified Newton (without Jacobian update); after that number of iterations, the modified Newton method gets a jacobian update and is further iterated
maxModifiedNewtonRestartIterations	UInt		7	maximum number of iterations for modified Newton after a Jacobian update; after that number of iterations, the full Newton method is started for this step
modifiedNewtonContractivity	PReal		0.5	maximum contractivity (=reduction of error in every Newton iteration) accepted by modified Newton; if contractivity is greater, a Jacobian update is computed
modifiedNewtonJacUpdatePerStep	bool		False	True: compute Jacobian at every time step (or static step), but not in every Newton iteration (except for bad convergence ==> switch to full Newton)
newtonResidualMode	UInt		0	0 ... use residual for computation of error (standard); 1 ... use ODE2 and ODE1 newton increment for error (set relTol and absTol to same values!) ==> may be advantageous if residual is zero, e.g., in kinematic analysis; TAKE CARE with this flag
relativeTolerance	UReal		1e-8	relative tolerance of residual for Newton (general goal of Newton is to decrease the residual by this factor)
useModifiedNewton	bool		False	True: compute Jacobian only at first call to solver; the Jacobian (and its factorizations) is not computed in each Newton iteration, even not in every (time integration) step; False: Jacobian (and factorization) is computed in every Newton iteration (default, but may be costly)
useNewtonSolver	bool		True	flag (true/false); false = linear computation, true = use Newton solver for nonlinear solution
weightTolerancePerCoordinate	bool		False	flag (true/false); false = compute error as L2-Norm of residual; true = compute error as (L2-Norm of residual) / (sqrt(number of coordinates)), which can help to use common tolerance independent of system size

9.2.0.5 GeneralizedAlphaSettings

Settings for generalized-alpha, implicit trapezoidal or Newmark time integration methods.
GeneralizedAlphaSettings has the following items:

Name	type / function return type	size	default value / function args	description
computeInitialAccelerations	bool		True	True: compute initial accelerations from system EOM in acceleration form; NOTE that initial accelerations that are following from user functions in constraints are not considered for now! False: use zero accelerations
lieGroupAddTangentOperator	bool		True	True: for Lie group nodes, in case that lieGroupSimplifiedKinematicRelations=True, the integrator adds the tangent operator for stiffness and constraint matrices, for improved Newton convergence; not available for sparse matrix mode (EigenSparse)
lieGroupSimplifiedKinematicRelations	bool		False	True: for Lie group nodes, the integrator uses the original kinematic relations of the Bruls and Cardona 2010 paper
newmarkBeta	UReal		0.25	value beta for Newmark method; default value beta = $\frac{1}{4}$ corresponds to (undamped) trapezoidal rule
newmarkGamma	UReal		0.5	value gamma for Newmark method; default value gamma = $\frac{1}{2}$ corresponds to (undamped) trapezoidal rule
resetAccelerations	bool		False	this flag only affects if computeInitialAccelerations=False: if resetAccelerations=True, accelerations are set zero in the solver function InitializeSolverInitialConditions; this may be unwanted in case of repeatedly called SolveSteps() and in cases where solutions shall be prolonged from previous computations
spectralRadius	UReal		0.9	spectral radius for Generalized-alpha solver; set this value to 1 for no damping or to 0 < spectralRadius < 1 for damping of high-frequency dynamics; for position-level constraints (index 3), spectralRadius must be < 1
useIndex2Constraints	bool		False	set useIndex2Constraints = true in order to use index2 (velocity level constraints) formulation
useNewmark	bool		False	if true, use Newmark method with beta and gamma instead of generalized-Alpha

9.2.0.6 ExplicitIntegrationSettings

Settings for explicit solvers, like Explicit Euler, RK44, ODE23, DOPRI5 and others. The settings may significantely influence performance.

ExplicitIntegrationSettings has the following items:

Name	type / function return type	size	default value / function args	description
computeEndOfStepAccelerations	bool		True	accelerations are computed at stages of the explicit integration scheme; if the user needs accelerations at the end of a step, this flag needs to be activated; if True, this causes a second call to the RHS of the equations, which may DOUBLE COMPUTATIONAL COSTS for one-step-methods; if False, the accelerations are re-used from the last stage, being slightly different
computeMassMatrixInversePerBody	bool		False	If true, the solver assumes the bodies to be independent and computes the inverse of the mass matrix for all bodies independently; this may lead to WRONG RESULTS, if bodies share nodes, e.g., two MassPoint objects put on the same node or a beam with a mass point attached at a shared node; however, it may speed up explicit time integration for large systems significantly (multi-threaded)
dynamicSolverType	DynamicSolverType		DynamicSolverType::DOPRI5	selection of explicit solver type (DOPRI5, ExplicitEuler, ExplicitMidpoint, RK44, RK67, VelocityVerlet, ...), for detailed description see DynamicSolverType, Section 6.10.6, but only referring to explicit solvers.
eliminateConstraints	bool		True	True: make explicit solver work for simple CoordinateConstraints, which are eliminated for ground constraints (e.g. fixed nodes in finite element models). False: incompatible constraints are ignored (BE CAREFUL)!
useLieGroupIntegration	bool		True	True: use Lie group integration for rigid body nodes; must be turned on for Lie group nodes (without data coordinates) to work properly; does not work for nodes with data coordinates!

9.2.0.7 TimeIntegrationSettings

General parameters used in time integration; specific parameters are provided in the according solver settings, e.g. for generalizedAlpha.
TimeIntegrationSettings has the following items:

Name	type / function return type	size	default value / function args	description
discontinuous	DiscontinuousSettings			parameters for treatment of discontinuities
explicitIntegration	ExplicitIntegrationSettings			special parameters for explicit time integration
generalizedAlpha	GeneralizedAlphaSettings			parameters for generalized-alpha, implicit trapezoidal rule or Newmark (options only apply for these methods)
newton	NewtonSettings			parameters for Newton method; used for implicit time integration methods only
absoluteTolerance	UReal		1e-8	*a_tol*: if automaticStepSize=True, absolute tolerance for the error control; must fulfill *a_tol* > 0; see Section 11.3
adaptiveStep	bool		True	True: the step size may be reduced if step fails; no automatic stepsize control
adaptiveStepDecrease	UReal		0.5	Multiplicative factor (MUST BE: 0 < factor < 1) for step size to decrese due to discontinuousIteration or Newton errors
adaptiveStepIncrease	UReal		2	Multiplicative factor (MUST BE > 1) for step size to increase after previous step reduction due to discontinuousIteration or Newton errors
adaptiveStepRecoveryIterations	UInt		7	Number of max. (Newton iterations + discontinuous iterations) at which a step increase is considered; in order to immediately increase steps after reduction, chose a high value
adaptiveStepRecoverySteps	UInt		10	Number of steps needed after which steps will be increased after previous step reduction due to discontinuousIteration or Newton errors
automaticStepSize	bool		True	True: for specific integrators with error control (e.g., DOPRI5), compute automatic step size based on error estimation; False: constant step size (step may be reduced if adaptiveStep=True); the maximum stepSize reads $h = h\_max = \frac{t\_end - t\_start}{n\_steps}$
computeLoadsJacobian	UInt		0	0: jacobian of loads not considered (may lead to slow convergence or Newton failure); 1: in case of implicit integrators, compute (numerical) Jacobian of ODE2 and ODE1 coordinates for loads, causing additional computational costs; this is advantageous in cases where loads are related nonlinearly to coordinates; 2: also compute ODE2_t dependencies for jacobian; note that computeLoadsJacobian has no effect in case of doSystemWideDifferentiation, as this anyway includes all load dependencies
endTime	UReal		1	*t_end*: end time of time integration

initialStepSize	UReal		0	*h_init*: if automaticStepSize=True, initial step size; if initialStepSize==0, max. stepSize, which is (endTime-startTime)/numberOfSteps, is used as initial guess; a good choice of initialStepSize may help the solver to start up faster.
minimumStepSize	PReal		1e-8	*h_min*: if automaticStepSize=True or adaptiveStep=True: lower limit of time step size, before integrator stops with adaptiveStep; lower limit of automaticStepSize control (continues but raises warning)
numberOfSteps	PReal		100	*n_steps*: number of steps in time integration; (maximum) stepSize h is computed from $h = \frac{t\_end - t\_start}{n\_steps}$; for automatic stepsize control, this stepSize is the maximum steps size, $h\_max = h$; numberOfSteps can be a float-point type, but must be close to an integer (relative tolerance $100 \cdot \varepsilon$) as it is silently rounded to int
realtimeFactor	PReal		1	if simulateInRealtime=True, this factor is used to make the simulation slower than realtime (factor < 1) or faster than realtime (factor > 1)
realtimeWaitMicroseconds	PInt		1000	if simulateInRealtime=True, a loop runs which waits realtimeWaitMicroseconds until checking again if the realtime is reached; using larger values leads to less CPU usage but less accurate realtime accuracy; smaller values (< 1000) increase CPU usage but improve realtime accuracy
relativeTolerance	UReal		1e-8	*r_tol*: if automaticStepSize=True, relative tolerance for the error control; must fulfill $r\_tol \geq 0$; see Section 11.3
reuseConstantMassMatrix	bool		True	True: does not recompute constant mass matrices (e.g. of some finite elements, mass points, etc.); if False, it always recomputes the mass matrix (e.g. needed, if user changes mass parameters via Python)
simulateInRealtime	bool		False	True: simulate in realtime; the solver waits for computation of the next step until the CPU time reached the simulation time; if the simulation is slower than realtime, it simply continues
startTime	UReal		0	*t_start*: start time of time integration (usually set to zero)

stepInformation	UInt		67	add up the following binary flags: 0 ... show only step time, 1 ... show time to go, 2 ... show newton iterations (Nit) per step or period, 4 ... show Newton jacobians (jac) per step or period, 8 ... show discontinuous iterations (Dit) per step or period, 16 ... show step size (dt), 32 ... show CPU time spent; 64 ... show adaptive step reduction warnings; 128 ... show step increase information; 1024 ... show every time step; time is usually shown in fractions of seconds (s), hours (h), or days
stepSizeMaxIncrease	UReal		2	f_maxInc: if automaticStepSize=True, maximum increase of step size per step, see Section 11.3; make this factor smaller (but > 1) if too many rejected steps
stepSizeSafety	UReal		0.90	r_sfty: if automaticStepSize=True, a safety factor added to estimated optimal step size, in order to prevent from many rejected steps, see Section 11.3. Make this factor smaller if many steps are rejected.
verboseMode	UInt		0	0 ... no output, 1 ... show short step information every 2 seconds (every 30 seconds after 1 hour CPU time), 2 ... show every step information, 3 ... show also solution vector, 4 ... show also mass matrix and jacobian (implicit methods), 5 ... show also Jacobian inverse (implicit methods)
verboseModeFile	UInt		0	same behaviour as verboseMode, but outputs all solver information to file

9.2.0.8 StaticSolverSettings

Settings for static solver linear or nonlinear (Newton).
StaticSolverSettings has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
discontinuous	DiscontinuousSettings			parameters for treatment of discontinuities
newton	NewtonSettings			parameters for Newton method (e.g. in static solver or time integration)
adaptiveStep	bool		True	True: use step reduction if step fails; False: fixed step size
adaptiveStepDecrease	UReal		0.25	Multiplicative factor (MUST BE: 0 < factor < 1) for step size to decrese due to discontinuousIteration or Newton errors

adaptiveStepIncrease	UReal		2	Multiplicative factor (MUST BE > 1) for step size to increase after previous step reduction due to discontinuousIteration or Newton errors
adaptiveStepRecoveryIterations	UInt		7	Number of max. (Newton iterations + discontinuous iterations) at which a step increase is considered; in order to immediately increase steps after reduction, chose a high value
adaptiveStepRecoverySteps	UInt		4	Number of steps needed after which steps will be increased after previous step reduction due to discontinuousIteration or Newton errors
computeLoadsJacobian	bool		True	True: compute (currently numerical) Jacobian for loads, causing additional computational costs; this is advantageous in cases where loads are related nonlinearly to coordinates; False: jacobian of loads not considered (may lead to slow convergence or Newton failure); note that computeLoadsJacobian has no effect in case of doSystemWideDifferentiation, as this anyway includes all load dependencies
constrainODE1coordinates	bool		True	True: ODE1coordinates are constrained to initial values; False: undefined behavior, currently not supported
loadStepDuration	PReal		1	quasi-time for all load steps (added to current time in load steps)
loadStepGeometric	bool		False	if loadStepGeometric=false, the load steps are incremental (arithmetic series, e.g. 0.1,0.2,0.3,...); if true, the load steps are increased in a geometric series, e.g. for $n = 8$ numberOfLoadSteps and $d = 1000$ loadStepGeometricRange, it follows: $1000^{1/8}/1000 = 0.00237$, $1000^{2/8}/1000 = 0.00562$, $1000^{3/8}/1000 = 0.0133$, ..., $1000^{7/8}/1000 = 0.422$, $1000^{8/8}/1000 = 1$
loadStepGeometricRange	PReal		1000	if loadStepGeometric=true, the load steps are increased in a geometric series, see loadStepGeometric
loadStepStart	UReal		0	a quasi time, which can be used for the output (first column) as well as for time-dependent forces; quasi-time is increased in every step i by loadStepDuration/numberOfLoadSteps; loadStepTime = loadStepStart + i*loadStepDuration/numberOfLoadSteps, but loadStepStart untouched ==> increment by user

minimumStepSize	PReal		1e-8	lower limit of step size, before nonlinear solver stops
numberOfLoadSteps	PInt		1	number of load steps; if numberOfLoadSteps=1, no load steps are used and full forces are applied at once
stabilizerODE2term	UReal		0	add mass-proportional stabilizer term in ODE2 part of jacobian for stabilization (scaled), e.g. of badly conditioned problems; the diagnoal terms are scaled with $stabilizer = (1 - loadStepFactor^2)$, and go to zero at the end of all load steps: $loadStepFactor = 1$ -> $stabilizer = 0$
stepInformation	UInt		67	add up the following binary flags: 0 ... show only step time, 1 ... show time to go, 2 ... show newton iterations (Nit) per step or period, 4 ... show Newton jacobians (jac) per step or period, 8 ... show discontinuous iterations (Dit) per step or period, 16 ... show step size (dt), 32 ... show CPU time spent; 64 ... show adaptive step reduction warnings; 128 ... show step increase information; 1024 ... show every time step; time is usually shown in fractions of seconds (s), hours (h), or days
useLoadFactor	bool		True	True: compute a load factor $\in [0,1]$ from static step time; all loads are scaled by the load factor; False: loads are always scaled with 1 – use this option if time dependent loads use a userFunction
verboseMode	UInt		1	0 ... no output, 1 ... show errors and load steps, 2 ... show short Newton step information (error), 3 ... show also solution vector, 4 ... show also jacobian, 5 ... show also Jacobian inverse
verboseModeFile	UInt		0	same behaviour as verboseMode, but outputs all solver information to file

9.2.0.9 LinearSolverSettings

Settings for linear solver, both dense and sparse (Eigen).
LinearSolverSettings has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**

ignoreSingularJacobian	bool		False	[ONLY implemented for dense, Eigen matrix mode] False: standard way, fails if jacobian is singular; True: use Eigen's FullPivLU (thus only works with LinearSolverType.EigenDense) which handles over- and underdetermined systems; can often resolve redundant constraints, but MAY ALSO LEAD TO ERRONEOUS RESULTS!
pivotThreshold	UReal		0	[ONLY available for EXUdense and EigenDense (FullPivot) solver] threshold for dense linear solver, can be used to detect close to singular solutions, setting this to, e.g., 1e-12; solver then reports on equations that are causing close to singularity
reuseAnalyzedPattern	bool		False	[ONLY available for sparse matrices] True: the Eigen SparseLU solver offers the possibility to reuse an analyzed pattern of a previous factorization; this may reduce total factorization time by a factor of 2 or 3, depending on the matrix type; however, if the matrix patterns heavily change between computations, this may even slow down performance; this flag is set for SparseMatrices in InitializeSolverData(...) and should be handled with care!
showCausingItems	bool		True	False: no output, if solver fails; True: if redundant equations appear, they are resolved such that according solution variables are set to zero; in case of redundant constraints, this may help, but it may lead to erroneous behaviour; for static problems, this may suppress static motion or resolve problems in case of instabilities, but should in general be considered with care!

9.2.0.10 Parallel

Settings for linear solver, both dense and sparse (Eigen).
Parallel has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**

multithreadedLLimitJacobians	PInt		20	compute jacobians (ODE2, AE, ...) multi-threaded; this is the limit number of according objects from which on parallelization is used; flag is copied into MainSystem internal flag at InitializeSolverData(...)
multithreadedLLimitLoads	PInt		20	compute loads multi-threaded; this is the limit number of loads from which on parallelization is used; flag is copied into MainSystem internal flag at InitializeSolverData(...)
multithreadedLLimitMassMatrices	PInt		20	compute bodies mass matrices multi-threaded; this is the limit number of bodies from which on parallelization is used; flag is copied into MainSystem internal flag at InitializeSolverData(...)
multithreadedLLimitResiduals	PInt		20	compute RHS vectors, AE, and reaction forces multi-threaded; this is the limit number of objects from which on parallelization is used; flag is copied into MainSystem internal flag at InitializeSolverData(...)
numberOfThreads	PInt		1	number of threads used for parallel computation (1 == scalar processing); do not use more threads than available threads (in most cases it is good to restrict to the number of cores); currently, only one solver can be started with multithreading; if you use several mbs in parallel (co-simulation), you should use serial computing
taskSplitMinItems	PInt		50	number of items from which on the tasks are split into subtasks (which slightly increases threading performance; this may be critical for smaller number of objects, should be roughly between 50 and 5000; flag is copied into MainSystem internal flag at InitializeSolverData(...)
taskSplitTasksPerThread	PInt		16	this is the number of subtasks that every thread receives; minimum is 1, the maximum should not be larger than 100; this factor is 1 as long as the taskSplitMinItems is not reached; flag is copied into MainSystem internal flag at InitializeSolverData(...)
useLoadBalancing	bool		True	if True, parallel computation uses load balancing, which may give better performance in case of non-equilibrated loads; (mobile) Intel CPUs may perform better without load balancing; this flag is coupled to exudyn.special.solver.multiThreadingLoadBalancing (overwritten when solver starts with multithreading)

9.2.0.11 SimulationSettings

General Settings for simulation; according settings for solution and solvers are given in subitems of this structure.
SimulationSettings has the following items:

Name	type/function return type	size	default value / function args	description
linearSolverSettings	LinearSolverSettings			linear solver parameters (used for dense and sparse solvers)
parallel	Parallel			parameters for vectorized and parallelized (multi-threaded) computations
solutionSettings	SolutionSettings			settings for solution files
staticSolver	StaticSolverSettings			static solver parameters
timeIntegration	TimeIntegrationSettings			time integration parameters
cleanUpMemory	bool		False	True: solvers will free memory at exit (recommended for large systems); False: keep allocated memory for repeated computations to increase performance
displayComputationTime	bool		False	display computation time statistics at end of solving
displayGlobalTimers	bool		True	display global timer statistics at end of solving (e.g., for contact, but also for internal timings during development)
displayStatistics	bool		False	display general computation information at end of time step (steps, iterations, function calls, step rejections, ...
linearSolverType	LinearSolverType		LinearSolverType::EXUdense	selection of numerical linear solver: exu.LinearSolverType.EXUdense (dense matrix inverse), exu.LinearSolverType.EigenSparse (sparse matrix LU-factorization), ... (enumeration type)
outputPrecision	UInt		6	precision for floating point numbers written to console; e.g. values written by solver
pauseAfterEachStep	bool		False	pause after every time step or static load step(user press SPACE)

9.3 Visualization settings

This section includes hierarchical structures for visualization settings, e.g., drawing of nodes, bodies, connectors, loads and markers and furthermore openGL, window and save image options. For further information, see Section 2.4.5.

9.3.0.1 VSettingsGeneral

General settings for visualization.
VSettingsGeneral has the following items:

Name	type / function return type	size	default value / function args	description
autoFitScene	bool		True	automatically fit scene within startup after SC.renderer.Start()
axesTiling	PInt		12	global number of segments for drawing cylinders for axes and cones for arrows (reduce this number, e.g. to 4, if many axes are drawn)
backgroundColor	Float4	4	[1.0,1.0,1.0,1.0]	red, green, blue and alpha values for background color of render window (white=[1,1,1,1]; black = [0,0,0,1])
backgroundColorBottom	Float4	4	[0.8,0.8,1.0,1.0]	red, green, blue and alpha values for bottom background color in case that useGradientBackground = True
circleTiling	PInt		16	global number of segments for circles; if smaller than 2, 2 segments are used (flat)
coordinateSystemSize	float		5.	size of coordinate system relative to font size
cylinderTiling	PInt		16	global number of segments for cylinders; if smaller than 2, 2 segments are used (flat)
drawCoordinateSystem	bool		True	false = no coordinate system shown
drawWorldBasis	bool		False	true = draw world basis coordinate system at (0,0,0)
graphicsUpdateInterval	float		0.1	interval of graphics update during simulation in seconds; 0.1 = 10 frames per second; low numbers might slow down computation speed
linuxDisplayScaleFactor	PFloat		1.	Scaling factor for linux, which cannot determined from system by now; adjust this value to scale dialog fonts and renderer fonts
minSceneSize	float		0.1	minimum scene size for initial scene size and for autoFitScene, to avoid division by zero; SET GREATER THAN ZERO
pointSize	float		0.01	global point size (absolute)
rendererPrecision	PInt		4	precision of general floating point numbers shown in render window: total number of digits used (max. 16)
renderWindowString	String		”	string shown in render window (use this, e.g., for debugging, etc.; written below EXUDYN, similar to solutionInformation in SimulationSettings.solutionSettings)

showComputationInfo	bool		True	true = show (hide) all computation information including Exudyn and version
showHelpOnStartup	PInt		5	seconds to show help message on startup (0=deactivate)
showSolutionInformation	bool		True	true = show solution information (from simulationSettings.solution)
showSolverInformation	bool		True	true = solver name and further information shown in render window
showSolverTime	bool		True	true = solver current time shown in render window
sphereTiling	PInt		6	global number of segments for spheres; if smaller than 2, 2 segments are used (flat)
textAlwaysInFront	bool		True	if true, text for item numbers and other item-related text is drawn in front; this may be unwanted in case that you only with to see numbers of objects in front; currently does not work with perspective
textColor	Float4	4	[0.,0.,0.,1.0]	general text color (default); used for system texts in render window
textHasBackground	bool		False	if true, text for item numbers and other item-related text have a background (depending on text color), allowing for better visibility if many numbers are shown; the text itself is black; therefore, dark background colors are ignored and shown as white
textOffsetFactor	UFloat		0.005	This is an additional out of plane offset for item texts (node number, etc.); the factor is relative to the maximum scene size and is only used, if textAlwaysInFront=False; this factor allows to draw text, e.g., in front of nodes
textSize	float		12.	general text size (font size) in pixels if not overwritten; if useWindowsDisplayScaleFactor=True, the the textSize is multplied with the windows display scaling (monitor scaling; content scaling) factor for larger texts on on high resolution displays; for bitmap fonts, the maximum size of any font (standard/large/huge) is limited to 256 (which is not recommended, especially if you do not have a powerful graphics card)
threadSafeGraphicsUpdate	bool		True	true = updating of visualization is thread-safe, but slower for complicated models; deactivate this to speed up computation, but activate for generation of animations; may be improved in future by adding a safe visualizationUpdate state

useBitmapText	bool		True	if true, texts are displayed using predefined bitmaps for the text; may increase the complexity of your scene, e.g., if many (>10000) node numbers shown
useGradientBackground	bool		False	true = use vertical gradient for background;
useMultiThreadedRendering	bool		True	true = rendering is done in separate thread; false = no separate thread, which may be more stable but has lagging interaction for large models (do not interact with models during simulation); set this parameter before call to SC.renderer.Start(); MAC OS: uses always false, because MAC OS does not support multi threaded GLFW
useWindowsDisplayScaleFactor	bool		True	the Windows display scaling (monitor scaling; content scaling) factor is used for increased visibility of texts on high resolution displays; based on GLFW glfwGetWindowContentScale; deactivated on linux compilation as it leads to crashes (adjust textSize manually!)
worldBasisSize	float		1.0	size of world basis coordinate system

9.3.0.2 VSettingsContour

Settings for contour plots; use these options to visualize field data, such as displacements, stresses, strains, etc. for bodies, nodes and finite elements.
VSettingsContour has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
alphaTransparency	float	1	1	default value for contour alpha transparency (RGB color computed from contour value)
automaticRange	bool		True	if true, the contour plot value range is chosen automatically to the maximum range
colorBarPrecision	PInt		4	precision of floating point values shown in color bar; total number of digits used (max. 16)
colorBarTiling	PInt	1	12	number of tiles (segements) shown in the colorbar for the contour plot
maxValue	float	1	1	maximum value for contour plot; set manually, if automaticRange == False
minValue	float	1	0	minimum value for contour plot; set manually, if automaticRange == False

nodesColored	bool		True	if true, the contour color is also applied to nodes (except mesh nodes), otherwise node drawing is not influenced by contour settings
outputVariable	OutputVariableType		OutputVariableType::_None	selected contour plot output variable type; select OutputVariableType._None to deactivate contour plotting.
outputVariableComponent	Int	1	0	select the component of the chosen output variable; e.g., for displacements, 3 components are available: 0 == x, 1 == y, 2 == z component; for stresses, 6 components are available, see OutputVariableType description; to draw the norm of a outputVariable, set component to -1; if a certain component is not available by certain objects or nodes, no value is drawn (using default color)
reduceRange	bool		True	if true, the contour plot value range is also reduced; better for static computation; in dynamic computation set this option to false, it can reduce visualization artifacts; you should also set minVal to max(float) and maxVal to min(float)
rigidBodiesColored	bool		True	if true, the contour color is also applied to triangular faces of rigid bodies and mass points, otherwise the rigid body drawing are not influenced by contour settings; for general rigid bodies (except for ObjectGround), Position, Displacement, DisplacementLocal(=0), Velocity, VelocityLocal, AngularVelocity, and AngularVelocityLocal are available; may slow down visualization!
showColorBar	bool		True	show the colour bar with minimum and maximum values for the contour plot

9.3.0.3 VSettingsNodes

Visualization settings for nodes.
VSettingsNodes has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
basisSize	float		0.2	size of basis for nodes
defaultColor	Float4	4	[0.2,0.2,1.,1.]	default RGBA color for nodes; 4th value is alpha-transparency

defaultSize	float		-1.	global node size; if -1.f, node size is relative to openGL.initialMaxSceneSize
drawNodesAsPoint	bool		True	simplified/faster drawing of nodes; uses general->pointSize as drawing size; if drawNodesAsPoint==True, the basis of the node will be drawn with lines
show	bool		True	flag to decide, whether the nodes are shown
showBasis	bool		False	show basis (three axes) of coordinate system in 3D nodes
showNodalSlopes	UInt		False	draw nodal slope vectors, e.g. in ANCF beam finite elements
showNumbers	bool		False	flag to decide, whether the node number is shown
tiling	PInt		4	tiling for node if drawn as sphere; used to lower the amount of triangles to draw each node; if drawn as circle, this value is multiplied with 4

9.3.0.4 VSettingsBeams

Visualization settings for beam finite elements.
VSettingsBeams has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
axialTiling	PInt		8	number of segments to discretise the beams axis
crossSectionFilled	bool		True	if implemented for element, cross section is drawn as solid (filled) instead of wireframe; NOTE: some quantities may not be interpolated correctly over cross section in visualization
crossSectionTiling	PInt		4	number of quads drawn over height of beam, if drawn as flat objects; leads to higher accuracy of components drawn over beam height or with, but also to larger CPU costs for drawing
drawVertical	bool		False	draw contour plot outputVariables 'vertical' along beam height; contour.outputVariable must be set accordingly
drawVerticalColor	Float4	4	[0.2,0.2,0.2,1.]	color for outputVariable to be drawn along cross section (vertically)
drawVerticalFactor	float		1.	factor for outputVariable to be drawn along cross section (vertically)

drawVerticalLines	bool		True	draw additional vertical lines for better visibility
drawVerticalOffset	float		0.	offset for vertical drawn lines; offset is added before multiplication with drawVerticalFactor
drawVerticalValues	bool		False	show values at vertical lines; note that these numbers are interpolated values and may be different from values evaluated directly at this point!
reducedAxialInterploation	bool		True	if True, the interpolation along the beam axis may be lower than the beam element order; this may be, however, show more consistent values than a full interpolation, e.g. for strains or forces

9.3.0.5 VSettingsKinematicTree

Visualization settings for kinematic trees.
VSettingsKinematicTree has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
frameSize	float		0.2	size of COM and joint frames
showCOMframes	bool		False	if True, a frame is attached to every center of mass
showFramesNumbers	bool		False	if True, numbers are drawn for joint frames (O[i]J[j]) and COM frames (O[i]COM[j]) for object [i] and local joint [j]
showJointFrames	bool		True	if True, a frame is attached to the origin of every joint frame

9.3.0.6 VSettingsBodies

Visualization settings for bodies.
VSettingsBodies has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
beams	VSettingsBeams			visualization settings for beams (e.g. ANCFCable or other beam elements)
kinematicTree	VSettingsKinematicTree			visualization settings for kinematic tree
defaultColor	Float4	4	[0.3,0.3,1.,1.]	default RGBA color for bodies; 4th value is alpha-transparency

defaultSize	Float3	3	[1.,1.,1.]	global body size of xyz-cube
deformationScaleFactor	float		1	global deformation scale factor; also applies to nodes, if drawn; currently only used for scaled drawing of (linear) finite elements in FFRF and FFRFreducedOrder objects
show	bool		True	flag to decide, whether the bodies are shown
showNumbers	bool		False	flag to decide, whether the body(=object) number is shown

9.3.0.7 VSettingsConnectors

Visualization settings for connectors.
VSettingsConnectors has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
contactPointsDefaultSize	float		0.02	DEPRECATED: do not use! global contact points size; if -1.f, connector size is relative to maxSceneSize
defaultColor	Float4	4	[0.2,0.2,1.,1.]	default RGBA color for connectors; 4th value is alpha-transparency
defaultSize	float		0.1	global connector size; if -1.f, connector size is relative to maxSceneSize
jointAxesLength	float		0.2	global joint axes length
jointAxesRadius	float		0.02	global joint axes radius
show	bool		True	flag to decide, whether the connectors are shown
showContact	bool		False	flag to decide, whether contact points, lines, etc. are shown
showJointAxes	bool		False	flag to decide, whether contact joint axes of 3D joints are shown
showNumbers	bool		False	flag to decide, whether the connector(=object) number is shown
springNumberOfWindings	PInt		8	number of windings for springs drawn as helical spring

9.3.0.8 VSettingsMarkers

Visualization settings for markers.
VSettingsMarkers has the following items:

Name	type/function return type	size	default value / function args	description
defaultColor	Float4	4	[0.1,0.5,0.1,1.]	default RGBA color for markers; 4th value is alpha-transparency
defaultSize	float		-1.	global marker size; if -1.f, marker size is relative to maxSceneSize
drawSimplified	bool		True	draw markers with simplified symbols
show	bool		True	flag to decide, whether the markers are shown
showNumbers	bool		False	flag to decide, whether the marker numbers are shown

9.3.0.9 VSettingsLoads

Visualization settings for loads.

VSettingsLoads has the following items:

Name	type/function return type	size	default value / function args	description
defaultColor	Float4	4	[0.7,0.1,0.1,1.]	default RGBA color for loads; 4th value is alpha-transparency
defaultRadius	float		0.005	global radius of load axis if drawn in 3D
defaultSize	float		0.2	global load size; if -1.f, load size is relative to maxSceneSize
drawSimplified	bool		True	draw markers with simplified symbols
drawWithUserFunction	bool		True	draw loads like force vectors time dependent; make sure that fixedLoadSize=false, while otherwise only the direction will change; user functions can only be drawn, if they are either symbolic or for Python user functions if useMultiThreadedRendering=False
fixedLoadSize	bool		True	if true, the load is drawn with a fixed vector length in direction of the load vector, independently of the load size
loadSizeFactor	float		0.1	if fixedLoadSize=false, then this scaling factor is used to draw the load vector
show	bool		True	flag to decide, whether the loads are shown
showNumbers	bool		False	flag to decide, whether the load numbers are shown

9.3.0.10 VSettingsTraces

Visualization settings for traces of sensors. Note that a large number of time points (influenced by simulationSettings.solutionSettings.sensorsWritePeriod) may lead to slow graphics.
VSettingsTraces has the following items:

Name	type / function return type	size	default value / function args	description
lineWidth	UFloat		2.	line width for traces
listOfPositionSensors	ArrayIndex	-1	[]	list of position sensors which can be shown as trace inside render window if sensors have storeInternal=True; if this list is empty and showPositionTrace=True, then all available sensors are shown
listOfTriadSensors	ArrayIndex	-1	[]	list of sensors of with OutputVariableType RotationMatrix; this non-empty list needs to coincide in length with the listOfPositionSensors to be shown if showTriads=True; the triad is drawn at the related position
listOfVectorSensors	ArrayIndex	-1	[]	list of sensors with 3D vector quantities; this non-empty list needs to coincide in length with the listOfPositionSensors to be shown if showVectors=True; the vector quantity is drawn relative to the related position
positionsShowEvery	PInt		1	integer value i; out of available sensor data, show every i-th position
sensorsMbsNumber	Index		0	number of main system which is used to for sensor lists; if only 1 mbs is in the SystemContainer, use 0; if there are several mbs, it needs to specify the number
showCurrent	bool		True	show current trace position (and especially vector quantity) related to current visualization state; this only works in solution viewer if sensor values are stored at time grid points of the solution file (up to a precision of 1e-10) and may therefore be temporarily unavailable
showFuture	bool		False	show trace future to current visualization state if already computed (e.g. in SolutionViewer)
showPast	bool		True	show trace previous to current visualization state
showPositionTrace	bool		False	show position trace of all position sensors if listOfPositionSensors=[] or of specified sensors; sensors need to activate storeInternal=True
showTriads	bool		False	if True, show basis vectors from rotation matrices provided by sensors
showVectors	bool		False	if True, show vector quantities according to description in showPositionTrace

timeSpan	UReal		0	maximum trace time span of past or future trace; given in seconds of simulation time; if zero, it is unused
traceColors	ArrayFloat	-1	[0.2,0.2,0.2,1., 0.8,0.2,0.2,1., 0.2,0.8,0.2,1., 0.2,0.2,0.8,1., 0.2,0.8,0.8,1., 0.8,0.2,0.8,1., 0.8,0.4,0.1,1.]	RGBA float values for traces in one array; using 6x4 values gives different colors for 6 traces; in case of triads, the 0/1/2-axes are drawn in red, green, and blue
triadSize	float		0.1	length of triad axes if shown
triadsShowEvery	PInt		1	integer value i; out of available sensor data, show every i-th triad
vectorScaling	float		0.01	scaling of vector quantities; if, e.g., loads, this factor has to be adjusted significantly
vectorsShowEvery	PInt		1	integer value i; out of available sensor data, show every i-th vector

9.3.0.11 VSettingsSensors

Visualization settings for sensors.
VSettingsSensors has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
traces	VSettingsTraces			settings for showing (position/triad) sensor traces and vector plots in the render window
defaultColor	Float4	4	[0.6,0.6,0.1,1.]	default RGBA color for sensors; 4th value is alpha-transparency
defaultSize	float		-1.	global sensor size; if -1.f, sensor size is relative to maxSceneSize
drawSimplified	bool		True	draw sensors with simplified symbols
show	bool		True	flag to decide, whether the sensors are shown
showNumbers	bool		False	flag to decide, whether the sensor numbers are shown

9.3.0.12 VSettingsContact

Global visualization settings for GeneralContact. This allows to easily switch on/off during visualization.
VSettingsContact has the following items:

Name	type / function return type	size	default value / function args	description
colorBoundingBoxes	Float4	4	[0.9,0.1,0.1,1.]	RGBA color for boudnding boxes, see showBoundingBoxes
colorSearchTree	Float4	4	[0.1,0.1,0.9,1.]	RGBA color for search tree, see showSearchTree
colorSpheres	Float4	4	[0.8,0.8,0.2,1.]	RGBA color for contact spheres, see showSpheres
colorTriangles	Float4	4	[0.5,0.5,0.5,1.]	RGBA color for contact triangles, see showTriangles
contactForcesFactor	float		0.001	factor used for scaling of contact forces is showContactForces=True
contactPointsDefaultSize	float		0.001	global contact points size; if -1.f, connector size is relative to maxSceneSize; used for some contacts, e.g., in ContactFrictionCircle
showBoundingBoxes	bool		False	show computed bounding boxes of all GeneralContacts; Warning: avoid for large number of contact objects!
showContactForces	bool		False	if True, contact forces are drawn for certain contact models
showContactForcesValues	bool		False	if True and showContactForces=True, numerical values for contact forces are shown at certain points
showSearchTree	bool		False	show outer box of search tree for all GeneralContacts
showSearchTreeCells	bool		False	show all cells of search tree; empty cells have colorSearchTree, cells with contact objects have higher red value; Warning: avoid for large number of search tree cells!
showSpheres	bool		False	show contact spheres (SpheresWithMarker, ...)
showTriangles	bool		False	show contact triangles (TrianglesRigidBodyBased, ...)
tilingCurves	PInt		8	tiling for nonlinear/polynomial curves; higher values give smoother curves
tilingSpheres	PInt		4	tiling for spheres; higher values give smoother spheres, but may lead to lower frame rates

9.3.0.13 VSettingsWindow

OpenGL Window and interaction settings for visualization; handle changes with care, as they might lead to unexpected results or crashes.
VSettingsWindow has the following items:

Name	type / function return type	size	default value / function args	description
alwaysOnTop	bool		False	True: OpenGL render window will be always on top of all other windows
ignoreKeys	bool		False	True: ignore keyboard input except escape and 'F2' keys; used for interactive mode, e.g., to perform kinematic analysis; This flag can be switched with key 'F2'

keyPressUserFunction	KeyPressUserFunction		0	add a Python function f(key, action, mods) here, which is called every time a key is pressed; function shall return true, if key has been processed; Example:
				def f(key, action, mods):
				print('key=',key);
				use chr(key) to convert key codes [32 ...96] to ascii; special key codes (>256) are provided in the exudyn.KeyCode enumeration type; key action needs to be checked (0=released, 1=pressed, 2=repeated); mods provide information (binary) for SHIFT (1), CTRL (2), ALT (4), Super keys (8), CAPSLOCK (16)

limitWindowToScreenSize	bool		True	True: render window size is limited to screen size; False: larger window sizes (e.g. for rendering) allowed according to renderWindowSize
maximize	bool		False	True: OpenGL render window will be maximized at startup
reallyQuitTimeLimit	UReal		900	number of seconds after which user is asked a security question before stopping simulation and closing renderer; set to 0 in order to always get asked; set to 1e10 to (nearly) never get asked
renderWindowSize	Index2	2	[1024,768]	initial size of OpenGL render window in pixel
ResetKeyPressUserFunction()	void			set keyPressUserFunction to zero (no function); because this cannot be assign to the variable itself
showMouseCoordinates	bool		False	True: show OpenGL coordinates and distance to last left mouse button pressed position; switched on/off with key 'F3'
showWindow	bool		True	True: OpenGL render window is shown on startup; False: window will be iconified at startup (e.g. if you are starting multiple computations automatically)
startupTimeout	PInt		2500	OpenGL render window startup timeout in ms (change might be necessary if CPU is very slow)

9.3.0.14 VSettingsDialogs

Settings related to dialogs (e.g., visualization settings dialog).
VSettingsDialogs has the following items:

Name	type / function return type	size	default value / function args	description
alphaTransparency	UFloat		0.94	alpha-transparency of dialogs; recommended range 0.7 (very transparent) - 1 (not transparent at all)
alwaysTopmost	bool		True	True: dialogs are always topmost (otherwise, they are sometimes hidden)
fontScalingMacOS	UFloat		1.35	font scaling value for MacOS systems (on Windows, system display scaling is used)
multiThreadedDialogs	bool		True	True: During dialogs, the OpenGL render window will still get updates of changes in dialogs, etc., which may cause problems on some platforms or for some (complicated) models; False: changes of dialogs will take effect when dialogs are closed
openTreeView	bool		False	True: all sub-trees of the visusalization dialog are opened when opening the dialog; False: only some sub-trees are opened

9.3.0.15 VSettingsMaterial

Settings for rendering materials, in particular for the Raytracer (may be available also in the OpenGL renderer in the future). This material (widely follows Phong model) can be either accessed via SC.renderer.materials or directly in visualizationSettings.raytracer.material0, material1, etc.; note that the default values shown in the documentation only reflect material0 but not all 10 default materials.
VSettingsMaterial has the following items:

Name	type / function return type	size	default value / function args	description
alpha	UFloat		1.	alpha-transparency, same as in alpha channel in RGBA colors; 1=opaque, 0=fully transparent; leads to extra rendering costs per transparent pixel
baseColor	Float3	3	[0.5,0.5,0.5]	RGB default material color if face color has R-color channel -1
emission	Float3	3	[0.,0.,0.]	RGB emissive material color (enlightened material)
ior	UFloat		1.	index of refraction for transparent materials (1=no refraction), >1 represents refraction

name	String		'undefined'	material name for easier handling
reflectivity	UFloat		0.	controls reflectivity of material; 0=no reflections (rough, e.g. rubber), 1=fully reflective (mirror); this leads to large extra rendering costs per visible reflective pixel
shininess	UFloat		32.	controls shininess of specular component of lights; values < 5 is not very shiny, while > 50 is very shiny
specular	Float3	3	[0.5,0.5,0.5]	RGB specular material color

9.3.0.16 VSettingsRaytracer

Settings for raytracer (software renderer) which can be used as alternative to classic OpenGL rendering; this option may be erased in future in favor of a modern GPU rendering. To activate the raytracer, simply switch the enable flag to True. The raytracer uses CPU-based rendering and is therefore comparably slow (may take seconds to render one frame). Thus, take care with the window dimension (start with small window size like 400 x 300) and use openGL.multiSampling=1. Note that many parameters are used from openGL settings, like backgroundColor, lineWidth, multiSampling, shadow (only on/off), and lights. See the options to improve appearance and performance.
VSettingsRaytracer has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
material0	VSettingsMaterial			settings for material0
material1	VSettingsMaterial			settings for material1
material2	VSettingsMaterial			settings for material2
material3	VSettingsMaterial			settings for material3
material4	VSettingsMaterial			settings for material4
material5	VSettingsMaterial			settings for material5
material6	VSettingsMaterial			settings for material6
material7	VSettingsMaterial			settings for material7
material8	VSettingsMaterial			settings for material8
material9	VSettingsMaterial			settings for material9
ambientLightColor	Float4	4	[0.6,0.6,0.6,1.]	scene RGBA color for ambient light effect (min. light for regions in shadow); same as openGL.materialAmbientAndDiffuse in OpenGL renderer

backgroundColorReflections	Float4	4	[0.4,0.4,0.4,1.]	scene RGBA color for background that is hit by reflection; while openGL.backgroundColor is used for rays that do not hit an object, this background may - if black or white - not be a suitable color for computing reflections; important, if scene is not inside a room.
enable	bool		False	True: use (software) raytracer; False: use standard OpenGL renderer
globalFogColor	Float4	4	[0.5,0.5,0.5,1.]	scene RGBA fog color
globalFogDensity	Real		0.	global fog density; fog is deactivated if fogDensity=0, otherwise it is a density relative to scene max size; as it is relative, the factor has to be relatively high to be visible (usually >1)
imageSizeFactor	PInt		1	Special size factor (1-16) to allow drawing with smaller resolution (faster); use this for long rendering times for adjustments, etc.
keepWindowActive	bool		False	Special flag, handle with care; True: sends some glfw functions to keep window reactive for long render times (>2 seconds); otherwise, the rendering may not finish due to timeout
lightRadius	float	1	0.1	if lightRadiusVariations>1, it uses the given radius for all lights, to convert point lights into distributed lights (slower)
lightRadiusVariations	PInt	1	1	if lightRadiusVariations>1, this defines the number of positions that are used to compute the effect of distributed lights (larger is slower but better quality); range=1..64
maxReflectionDepth	UInt		2	Maximum number of reflections computed for one ray (note that for each transparent face passed, the reflection depth is reduced by 1); maximum is 32 (but should not be more than 2-4 usually!)
maxTransparencyDepth	UInt		2	Maximum number of transparent faces that can be passed (note that for each reflection, the transparency depth is reduced by 1); maximum is 32 (but should not be more than 2-4 usually!)
numberOfThreads	PInt		8	Number of CPU-threads (max: 256) used for software rendering (should be approx. the number of available threads)
searchTreeFactor	PInt		1	This factor can be used to increase the number of search tree bins, which can improve performance in case of inequilibrated scense; range=1..128

tilesPerThread	PInt		12	Total number of sub-tiles per thread, used to evenly distribute rendering load to threads
verbose	bool		False	True: print out some debug information on rendering, in particular rendering timings and counter
zBiasLines	Real		1e-3	offset for lines to draw in front of faces; relative to scene radius
zOffsetCamera	Real		-0.01	offset for for camera towards the scene; use positive factor put camera inside, e.g. of brick (like a room) or sphere; use (slightly) negative value to make whole scene visible

9.3.0.17 VSettingsOpenGL

OpenGL settings for 2D and 2D rendering. For further details, see the OpenGL functionality.
VSettingsOpenGL has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
clippingPlaneColor	Float4	4	[0.7,0.5,0.5,0.]	RGBA color for clipping plane; if alpha-channel is 0, the cutting plane is not drawn; if alpha-channel is 1, the clippingPlaneColor is used; if alpha-channel is 2, the color of the object interior is used as clipping plane color (which may look strange in case of object-in-object)
clippingPlaneDistance	float		0.	distance of clipping plane on normal vector; see also clippingPlaneNormal
clippingPlaneNormal	Float3	3	[0.,0.,0.]	normal vector of clipping plane, e.g. [0,0,1] to set a xy-clipping plane; the clipped half-space is in direction of the normal; use [0,0,0] to deactivate clipping plane; Note that clipping is mainly made for triangles in order to visualize hidden objects and currently it only fully clips triangles, but does not exactly cut them; see also clippingPlaneDistance
depthSorting	bool	1	False	True (slower): sort triangles by Z-depth to remove transparency artifacts: only works if triangles do not intersect or come close (you may like to refine triangle meshes); False: no depth-sort (faster)
drawFaceNormals	bool	1	False	draws triangle normals, e.g. at center of triangles; used for debugging of faces
drawNormalsLength	PFloat	1	0.1	length of normals; used for debugging

drawVertexNormals	bool	1	False	draws vertex normals; used for debugging
enableLight0	bool	1	True	turn on/off light0
enableLight1	bool	1	True	turn on/off light1
enableLighting	bool	1	True	generally enable lighting (otherwise, colors of objects are used); OpenGL: glEnable(GL_LIGHTING)
faceEdgesColor	Float4	4	[0.2,0.2,0.2,1.]	global RGBA color for face edges
facesTransparent	bool	1	False	True: show faces transparent independent of transparency (A)-value in color of objects; allow to show otherwise hidden node/marker/object numbers
initialCenterPoint	Float3	3	[0.,0.,0.]	centerpoint of scene (3D) at renderer startup; overwritten if autoFitScene = True
initialMaxSceneSize	PFloat		1.	initial maximum scene size (auto: diagonal of cube with maximum scene coordinates); used for 'zoom all' functionality and for visibility of objects; overwritten if autoFitScene = True
initialModelRotation	StdArray33F	3x3	[Matrix3DF[3,3,1.,0.,0., 0.,1.,0., 0.,0.,1.]]	initial model rotation matrix for OpenGl; in python use e.g.: initialModelRotation=[[1,0,0],[0,1,0],[0,0,1]]
initialZoom	UFloat		1.	initial zoom of scene; overwritten/ignored if autoFitScene = True
light0ambient	float	1	0.3	ambient value of GL_LIGHT0
light0constantAttenuation	float	1	1.0	constant attenuation coefficient of GL_LIGHT0, this is a constant factor that attenuates the light source; attenuation factor = 1/(kx +kl*d + kq*d*d); (kc,kl,kq)=(1,0,0) means no attenuation; only used for lights, where last component of light position is 1
light0diffuse	float	1	0.6	diffuse value of GL_LIGHT0
light0linearAttenuation	float	1	0.0	linear attenuation coefficient of GL_LIGHT0, this is a linear factor for attenuation of the light source with distance
light0position	Float4	4	[0.2,0.2,10.,0.]	4D position vector of GL_LIGHT0; 4th value should be 0 for lights like sun, but 1 for directional lights (and for attenuation factor being calculated); light0 is also used for shadows, so you need to adjust this position; see opengl manuals
light0quadraticAttenuation	float	1	0.0	quadratic attenuation coefficient of GL_LIGHT0, this is a quadratic factor for attenuation of the light source with distance

light0specular	float	1	0.5	specular value of GL_LIGHT0
light1ambient	float	1	0.0	ambient value of GL_LIGHT1
light1constantAttenuation	float	1	1.0	constant attenuation coefficient of GL_LIGHT1, this is a constant factor that attenuates the light source; attenuation factor = 1/(kx +kl*d + kq*d*d); only used for lights, where last component of light position is 1
light1diffuse	float	1	0.5	diffuse value of GL_LIGHT1
light1linearAttenuation	float	1	0.0	linear attenuation coefficient of GL_LIGHT1, this is a linear factor for attenuation of the light source with distance
light1position	Float4	4	[1.,1.,-10.,0.]	4D position vector of GL_LIGHT0; 4th value should be 0 for lights like sun, but 1 for directional lights (and for attenuation factor being calculated); see opengl manuals
light1quadraticAttenuation	float	1	0.0	quadratic attenuation coefficient of GL_LIGHT1, this is a quadratic factor for attenuation of the light source with distance
light1specular	float	1	0.6	specular value of GL_LIGHT1
lightModelAmbient	Float4	4	[0.,0.,0.,1.]	global ambient light; maps to OpenGL glLightModeli(GL_LIGHT_MODEL_AMBIENT,[r,g,b,a])
lightModelLocalViewer	bool	1	False	select local viewer for light; maps to OpenGL glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER,...)
lightModelTwoSide	bool	1	False	enlighten also backside of object; may cause problems on some graphics cards and lead to slower performance; maps to OpenGL glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,...)
lightPositionsInCameraFrame	bool	1	False	set False to set light positions and directions relative to model frame; True: lights are in global (camera) frame; this is always False for raytracer; this was True up to Exudyn 1.9.174
lineSmooth	bool	1	True	draw lines smooth
lineWidth	UFloat	1	1.	width of lines used for representation of lines, circles, points, etc.
materialAmbientAndDiffuse	Float4	4	[0.6,0.6,0.6,1.]	RGBA ambient color of material
materialShininess	float	1	32.	shininess of material
materialSpecular	Float4	4	[0.6,0.6,0.6,1.]	RGBA specular color of material

multiSampling	PInt	1	1	NOTE: this parameter must be set before starting renderer; later changes are not affecting visualization; multi sampling turned off (<=1) or turned on to given values (2, 4, 8 or 16); increases the graphics buffers and might crash due to graphics card memory limitations; only works if supported by hardware; if it does not work, try to change 3D graphics hardware settings!
perspective	UFloat		0.	parameter prescribes amount of perspective (0=no perspective=orthographic projection; positive values increase perspective; feasible values are 0.001 (little perspective) ... 0.5 (large amount of perspective); mouse coordinates will not work with perspective
polygonOffset	float		0.01	general polygon offset for polygons, except for shadows; use this parameter to draw polygons behind lines to reduce artifacts for very large or small models
shadeModelSmooth	bool	1	True	True: turn on smoothing for shaders, which uses vertex normals to smooth surfaces
shadow	UFloat		0.	parameter $\in$ [0...1] prescribes amount of shadow for light0 (using light0position, etc.) in OpenGL renderer; if this parameter is different from 1, rendering of triangles becomes approx. 5 times more expensive, so take care in case of complex scenes; for complex object, such as spheres with fine resolution or for particle systems, the present approach has limitations and leads to artifacts and unrealistic shadows; for raytracer, shadow is included by a physics-based model for all lights if shadow>0
shadowPolygonOffset	PFloat		0.1	some special drawing parameter for shadows which should be handled with care; defines some offset needed by openGL to avoid aritfacts for shadows and depends on maxSceneSize; this value may need to be reduced for larger models in order to achieve more accurate shadows, it may be needed to be increased for thin bodies
showFaceEdges	bool	1	False	show edges of faces; using the options showFaces=false and showFaceEdges=true gives are wire frame representation
showFaces	bool	1	True	show faces of triangles, etc.; using the options showFaces=false and showFaceEdges=true gives are wire frame representation

showLines	bool	1	True	show lines (different from edges of faces)
showMeshEdges	bool	1	True	show edges of finite elements; independent of showFaceEdges
showMeshFaces	bool	1	True	show faces of finite elements; independent of showFaces
textLineSmooth	bool	1	False	draw lines for representation of text smooth
textLineWidth	UFloat	1	1.	width of lines used for representation of text

9.3.0.18 VSettingsExportImages

Functionality to export images to files (PNG or TGA format) which can be used to create animations; to activate image recording during the solution process, set SolutionSettings.recordImagesInterval accordingly.
VSettingsExportImages has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
heightAlignment	PInt		2	alignment of exported image height; using a value of 2 helps to reduce problems with video conversion (additional horizontal lines are lost)
saveImageAsTextCircles	bool		True	export circles in save image (only in TXT format)
saveImageAsTextLines	bool		True	export lines in save image (only in TXT format)
saveImageAsTextTexts	bool		False	export text in save image (only in TXT format)
saveImageAsTextTriangles	bool		False	export triangles in save image (only in TXT format)
saveImageFileCounter	UInt		0	current value of the counter which is used to consecutively save frames (images) with consecutive numbers
saveImageFileName	FileName		'images/frame'	filename (without extension!) and (relative) path for image file(s) with consecutive numbering (e.g., frame0000.png, frame0001.png,...); ; directory will be created if it does not exist
saveImageFormat	String		'PNG'	format for exporting figures: currently only PNG, TGA and TXT available; while PNG and TGA represent the according image file formats, the TXT format results in a text file containing the 3D graphics data information as lists of lines, triangles, etc; PNG is not available for Ubuntu18.04 (check use TGA has highest compatibility with all platforms

saveImageSingleFile	bool		False	True: only save single files with given filename, not adding numbering; False: add numbering to files, see saveImageFileName
saveImageTimeOut	PInt		5000	timeout in milliseconds for saving a frame as image to disk; this is the amount of time waited for redrawing; increase for very complex scenes
widthAlignment	PInt		4	alignment of exported image width; using a value of 4 helps to reduce problems with video conversion (additional vertical lines are lost)

9.3.0.19 VSettingsOpenVR

Functionality to interact openVR; requires special hardware or software emulator, see steam / openVR descriptions.
VSettingsOpenVR has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
actionManifestFileName	FileName		'C:/openVRactionsManifest.json'	This string must contain a string representing a valid absolute path to a vr_actions.json manifest, which describes all HMD, tracker, etc. devices as given by openVR
enable	bool		False	True: openVR enabled (if compiled with according flag and installed openVR)
logLevel	Int		1	integer value setting log level of openVR: -1 (no output), 0 (error), 1 (warning), 2 (info), 3 (debug); increase log level to get more output
showCompanionWindow	bool		True	True: openVR will show companion window containing left and right eye view

9.3.0.20 VSettingsInteractive

Functionality to interact with render window; will include left and right mouse press actions and others in future.
VSettingsInteractive has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
openVR	VSettingsOpenVR			openVR visualization settings

highlightColor	Float4	4	[0.8,0.05,0.05,0.75]	RGBA color for highlighted item; 4th value is alpha-transparency
highlightItemIndex	Int		-1	index of item that shall be highlighted (e.g., need to find item due to errors); if set -1, no item is highlighted
highlightItemType	ItemType		ItemType::_None	item type (Node, Object, ...) that shall be highlighted (e.g., need to find item due to errors)
highlightMbsNumber	UInt		0	index of main system (mbs) for which the item shall be highlighted; number is related to the ID in SystemContainer (first mbs = 0, second = 1, ...)
highlightOtherColor	Float4	4	[0.5,0.5,0.5,0.4]	RGBA color for other items (which are not highlighted); 4th value is alpha-transparency
joystickScaleRotation	float		200.	rotation scaling factor for joystick input
joystickScaleTranslation	float		6.	translation scaling factor for joystick input
keypressRotationStep	float		5.	rotation increment per keypress in degree (full rotation = 360 degree)
keypressTranslationStep	float		0.1	translation increment per keypress relative to window size
lockModelView	bool		False	True: all movements (with mouse/keys), rotations, zoom are disabled; initial values are considered ==> initial zoom, rotation and center point need to be adjusted, approx. 0.4*maxSceneSize is a good value
mouseMoveRotationFactor	float		1.	rotation increment per 1 pixel mouse movement in degree
pauseWithSpacebar	bool		True	True: during simulation, space bar can be pressed to pause simulation
selectionHighlights	bool		True	True: mouse click highlights item (default: red)
selectionLeftMouse	bool		True	True: left mouse click on items and show basic information
selectionLeftMouseItemTypes	Index		31	binary flags (1,2,4,8,16) for (Node,Object,Marker,Load,Sensor) that are identified with left mouse click selection
selectionRightMouse	bool		True	True: right mouse click on items and show dictionary (read only!)
selectionRightMouseGraphicsData bool			False	True: right mouse click on items also shows GraphicsData information for inspectation (may sometimes be very large and may not fit into dialog for large graphics objects!)

trackMarker	Int		-1	if valid marker index is provided and marker provides position (and orientation), the centerpoint of the scene follows the marker (and orientation); depends on trackMarkerPosition and trackMarkerOrientation; by default, only position is tracked
trackMarkerMbsNumber	Index		0	number of main system which is used to track marker; if only 1 mbs is in the SystemContainer, use 0; if there are several mbs, it needs to specify the number
trackMarkerOrientation	Float3	3	[0.,0.,0.]	choose which orientation axes (x,y,z) are tracked; currently can only be all zero or all one
trackMarkerPosition	Float3	3	[1.,1.,1.]	choose which coordinates or marker are tracked (x,y,z)
useJoystickInput	bool		True	True: read joystick input (use 6-axis joystick with lowest ID found when starting renderer window) and interpret as (x,y,z) position and (rotx, roty, rotz) rotation: as available from 3Dconnexion space mouse and maybe others as well; set to False, if external joystick makes problems ...
zoomStepFactor	float		1.15	change of zoom per keypress (keypad +/-) or mouse wheel increment

9.3.0.21 VisualizationSettings

Settings for visualization.
VisualizationSettings has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
bodies	VSettingsBodies			body visualization settings
connectors	VSettingsConnectors			connector visualization settings
contact	VSettingsContact			contact visualization settings
contour	VSettingsContour			contour plot visualization settings
dialogs	VSettingsDialogs			dialogs settings
exportImages	VSettingsExportImages			settings for exporting (saving) images to files in order to create animations
general	VSettingsGeneral			general visualization settings
interactive	VSettingsInteractive			Settings for interaction with renderer
loads	VSettingsLoads			load visualization settings
markers	VSettingsMarkers			marker visualization settings
nodes	VSettingsNodes			node visualization settings

openGL	VSettingsOpenGL			OpenGL rendering settings
raytracer	VSettingsRaytracer			Raytracer settings (builds on OpenGL rendering settings)
sensors	VSettingsSensors			sensor visualization settings
window	VSettingsWindow			visualization window and interaction settings

9.4 Solver substructures

This section includes structures contained in the solver, which can be accessed via the Python interface during solution or for building a customized solver in Python. There is plenty of possibilities to interact with the solvers, being it the extraction of data at the end (such as .it or .conv), computation of mass matrix or system matrices, and finally the modification of solver structures (which may have effect or not). In any case, there is no full description for all these methods and the user must always consider the according C++ function to verify the desired behavior.

9.4.0.1 CSolverTimer

Structure for timing in solver. Each Real variable is used to measure the CPU time which certain parts of the solver need. This structure is only active if the code is not compiled with the __FAST_EXUDYN_LINALG option and if displayComputationTime is set True. Timings will only be filled, if useTimer is True.
CSolverTimer has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
AERHS	Real		0.	time for residual evaluation of algebraic equations right-hand-side
errorEstimator	Real		0.	for explicit solvers, additional evaluation
factorization	Real		0.	solve or inverse
integrationFormula	Real		0.	time spent for evaluation of integration formulas
jacobianAE	Real		0.	jacobian of algebraic equations (not counted in sum)
jacobianODE1	Real		0.	jacobian w.r.t. coordinates of ODE1 equations (not counted in sum)
jacobianODE2	Real		0.	jacobian w.r.t. coordinates of ODE2 equations (not counted in sum)
jacobianODE2_t	Real		0.	jacobian w.r.t. coordinates_t of ODE2 equations (not counted in sum)
massMatrix	Real		0.	mass matrix computation
newtonIncrement	Real		0.	Jac^{-1} * RHS; backsubstitution
ODE1RHS	Real		0.	time for residual evaluation of ODE1 right-hand-side

ODE2RHS	Real		0.	time for residual evaluation of ODE2 right-hand-side
overhead	Real		0.	overhead, such as initialization, copying and some matrix-vector multiplication
postNewton	Real		0.	discontinuous iteration / PostNewtonStep
python	Real		0.	time spent for Python functions
reactionForces	Real		0.	CqT * lambda
realtimeIdleCPU	Real		0.	time waited for next frame to compute and draw if simulateInRealtime is True
Reset(...)	void		useSolverTimer	reset solver timings to initial state by assigning default values; useSolverTimer sets the useTimer flag
StartTimer(...)	void		value	start timer function for a given variable; subtracts current CPU time from value
StopTimer(...)	void		value	stop timer function for a given variable; adds current CPU time to value
Sum()	Real			compute sum of all timers (except for those counted multiple, e.g., jacobians
ToString()	String			converts the current timings to a string
total	Real		0.	total time measured between start and end of computation (static/dynamics)
totalJacobian	Real		0.	time for all jacobian computations
useTimer	bool		True	flag to decide, whether the timer is used (true) or not
visualization	Real		0.	time spent for visualization in computation thread
writeSolution	Real		0.	time for writing solution

9.4.0.2 SolverIterationData

Solver internal structure for counters, steps, step size, time, etc.; solution vectors, residuals, etc. are SolverLocalData. The given default values are overwritten by the simulationSettings when initializing the solver.
SolverIterationData has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
adaptiveStep	bool		True	True: the step size may be reduced if step fails; no automatic stepsize control
automaticStepSize	bool		True	True: if timeIntegration.automaticStepSize == True AND chosen integrators supports automatic step size control (e.g., DOPRI5); False: constant step size used (step may be reduced if adaptiveStep=True)

automaticStepSizeError	Real		0	estimated error (relative to atol + rtol*solution) of last step; must be ≤ 1 for a step to be accepted
currentStepIndex	Index		0	current step index; *i*
currentStepSize	Real		0.	stepSize of current step
currentTime	Real		0.	holds the current simulation time, copy of state.current.time; interval is [startTime,tEnd]; in static solver, duration is loadStepDuration
discontinuousIteration	Index		0	number of current discontinuous iteration
discontinuousIterationsCount	Index		0	count total number of discontinuous iterations (min. 1 per step)
endTime	Real		0.	end time of static/dynamic solver
initialStepSize	Real		1e-6	initial stepSize for dynamic solver; only used, if automaticStepSize is activated
lastStepSize	Real		0.	stepSize suggested from last step or by initial step size; only used, if automaticStepSize is activated
maxStepSize	Real		0.	constant or maximum stepSize
minStepSize	Real		0.	minimum stepSize for static/dynamic solver; only used, if automaticStepSize is activated
newtonJacobiCount	Index		0	count total Newton jacobian computations
newtonSteps	Index		0	number of current newton steps
newtonStepsCount	Index		0	count total Newton steps
numberOfSteps	Index		0	number of time steps (if fixed size); *n*
recommendedStepSize	Real		-1.	recommended step size *h_recom* after PostNewton(...): *h_recom* < 0: no recommendation, *h_recom* == 0: use minimum step size, *h_recom* > 0: use specific step size, if no smaller size requested by other reason
rejectedAutomaticStepSizeSteps	Index		0	count the number of rejected steps in case of automatic step size control (rejected steps are repeated with smaller step size)
rejectedModifiedNewtonSteps	Index		0	count the number of rejected modified Newton steps (switch to full Newton)
startTime	Real		0.	time at beginning of time integration
ToString()	String			convert iteration statistics to string; used for displayStatistics option

9.4.0.3 SolverConvergenceData

Solver internal structure for convergence information: residua, iteration loop errors and error flags.
For detailed behavior of these flags, visit the source code!.
SolverConvergenceData has the following items:

Name	type / function return type	size	default value / function args	description
contractivity	Real		0.	Newton contractivity = geometric decay of error in every step
discontinuousIterationError	Real		0.	error of discontinuous iterations (contact, friction, ...) outside of Newton iteration
discontinuousIterationSuccessful	bool		True	true, if last discontinuous iteration had success (failure may be recovered by adaptive step)
errorCoordinateFactor	Real		1.	factor may include the number of system coordinates to reduce the residual
InitializeData()	void			initialize SolverConvergenceData by assigning default values
jacobianUpdateRequested	bool		True	true, if a jacobian update is requested in modified Newton (determined in previous step)
lastResidual	Real		0.	last Newton residual to determine contractivity
linearSolverCausingRow	Index		-1	-1 if successful, 0 ... n-1, the system equation (=coordinate) index which may have caused the problem, at which the linear solver failed
linearSolverFailed	bool		False	true, if linear solver failed to factorize
massMatrixNotInvertible	bool		False	true, if mass matrix is not invertable during initialization or solution (explicit solver)
newtonConverged	bool		False	true, if Newton has (finally) converged
newtonSolutionDiverged	bool		False	true, if Newton diverged (may be recovered)
residual	Real		0.	current Newton residual
stepReductionFailed	bool		False	true, if iterations over time/static steps failed (finally, cannot be recovered)
stopNewton	bool		False	set true by Newton, if Newton was stopped, e.g., because of exceeding iterations or linear solver failed

9.4.0.4 SolverOutputData

Solver internal structure for output modes, output timers and counters.
SolverOutputData has the following items:

Name	type / function return type	size	default value / function args	description
cpuLastTimePrinted	Real		0.	CPU time when output has been printed last time

cpuStartTime	Real		0.	CPU start time of computation (starts counting at computation of initial conditions)
finishedSuccessfully	bool		False	flag is false until solver functions SolveSteps)...) or SolveSystem(...) finished successfully (can be used as external trigger)
initializationSuccessful	bool		False	flag is set during call to InitializeSolver(...); reasons for failure are multiple, either inconsistent solver settings are used, files cannot be written (file locked), or initial conditions could not be computed
InitializeData()	void			initialize SolverOutputData by assigning default values
lastDiscontinuousIterationsCount	Index		0	discontinuous iterations count when written to console (or file) last time
lastImageRecorded	Real		0.	simulation time when last image has been recorded
lastNewtonJacobiCount	Index		0	jacobian update count when written to console (or file) last time
lastNewtonStepsCount	Index		0	newton steps count when written to console (or file) last time
lastSensorsWritten	Real		0.	simulation time when last sensors have been written
lastSolutionWritten	Real		0.	simulation time when last solution has been written
lastVerboseStepIndex	Index		0	step index when last time written to console (or file)
multiThreadingMode	Index		0	multithreading mode that has been used: 0=None (serial), 1=multithreading, 2=multithreading with load balancing; (modes new since 2025-06, V1.9.198)
numberOfThreadsUsed	Index		1	number of threads that have been used in simulation
stepInformation	Index		0	this is a copy of the solvers stepInformation used for console output
verboseMode	Index		0	this is a copy of the solvers verboseMode used for console output
verboseModeFile	Index		0	this is a copy of the solvers verboseModeFile used for file
writeToSolutionFile	bool		False	if false, no solution file is generated and no file is written
writeToSolverFile	bool		False	if false, no solver output file is generated and no file is written

9.4.0.5 MainSolverStatic

PyBind interface (trampoline) class for static solver. With this interface, the static solver and its substructures can be accessed via Python. NOTE that except from SolveSystem(...), these functions are only intended for experienced users and they need to be handled with care, as unexpected crashes may happen if used inappropriate. Furthermore, the functions have a lot of overhead (performance much lower than internal solver) due to Python interfaces, and should thus be used for small systems.
To access the solver in Python, write:
`solver = MainSolverStatic()`
and hereafter you can access all data and functions via 'solver'.
MainSolverStatic has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
conv	SolverConvergenceData			all information about tolerances, errors and residua
it	SolverIterationData			all information about iterations (steps, discontinuous iteration, newton,...)
newton	NewtonSettings			copy of newton settings from timeint or staticSolver
output	SolverOutputData			output modes and timers for exporting solver information and solution
timer	CSolverTimer			timer which measures the CPU time of solver sub functions
CheckInitialized(...)	bool		mainSystem	check if MainSolver and MainSystem are correctly initialized ==> otherwise raise SysError
ComputeAlgebraicEquations(...)	void		mainSystem, velocityLevel=False	compute the algebraic equations in systemResidual in range(nODE2+nODE1, nODE2+nODE1+nAE)
ComputeJacobianAE(...)	void		mainSystem, scalarFactor_ODE2=1., scalarFactor_ODE2_t=0., scalarFactor_ODE1=1., velocityLevel=False	add jacobian of algebraic equations (multiplied with factor) to systemJacobian in cSolver; the scalarFactors are scaling the derivatives w.r.t. ODE2 coordinates, ODE2_t (velocity) coordinates and ODE1 coordinates; if velocityLevel == true, the constraints are evaluated at velocity level; the scalar factors scalarFactor_ODE2=0 and scalarFactor_ODE2 are used for the same ODE2 block in the jacobian

ComputeJacobianODE1RHS(...)	void		mainSystem, scalarFactor_ODE2=1., scalarFactor_ODE2_t=0., scalarFactor_ODE1=1.	ADD jacobian of ODE1RHS (multiplied with factors for ODE2 and ODE1 coordinates) to the according rows (nODE2:nODE2+nODE1) of the exising systemJacobian in cSolver; it requires a prior call to ComputeJacobianODE2RHS(...); the scalar factors scalarFactor_ODE2=0 and scalarFactor_ODE2 are used for the same ODE2 block in the jacobian
ComputeJacobianODE2RHS(...)	void		mainSystem, scalarFactor_ODE2=1., scalarFactor_ODE2_t=0., scalarFactor_ODE1=1., computeLoadsJacobian=0	set systemJacobian to zero, size = (nODE2+nODE1+nAE) x (nODE2+nODE1+nAE), and add jacobian (multiplied with factors for ODE2 and ODE1 coordinates) of ODE2RHS to systemJacobian in cSolver; using (scalarFactor_ODE2=-1,scalarFactor_ODE2=0) gives the stiffness matrix (=derivatives of ODE2 coords) in the nODE2 x nODE2 part, while using (scalarFactor_ODE2=0,scalarFactor_ODE2=-1) gives the damping matrix (= derivatives of ODE2 velocity coordinates) in the same part; a superposition of these two parts makes sense for implicit solvers; if , Index computeLoadsJacobian=0, loads are not considered in the Jacobian computation; for , Index computeLoadsJacobian=1 the ODE2 and ODE1 derivatives of loads are included and for , Index computeLoadsJacobian=2, also the ODE2_t dependencies are added
ComputeLoadFactor(...)	Real		simulationSettings	for static solver, this is a factor in interval [0,1]; MUST be overwritten
ComputeMassMatrix(...)	void		mainSystem, scalarFactor=1.	compute systemMassMatrix (multiplied with factor) in cSolver and return mass nODE2 x nODE2 matrix
ComputeNewtonJacobian(...)	void		mainSystem, simulationSettings	compute jacobian for newton method of given solver method; store result in systemJacobian
ComputeNewtonResidual(...)	Real		mainSystem, simulationSettings	compute residual for Newton method (e.g. static or time step); store residual vector in systemResidual and return scalar residual (specific computation may depend on solver types)
ComputeNewtonUpdate(...)	void		mainSystem, simulationSettings, initial=True	compute update for currentState from newtonSolution (decrement from residual and jacobian); if initial, this is for the initial update with newtonSolution=0

ComputeODE2RHS(...)	void		mainSystem	compute the RHS of ODE2 equations in systemResidual in range(0,nODE2)
DiscontinuousIteration(...)	bool		mainSystem, simulationSettings	perform discontinuousIteration for static step / time step; CALLS ComputeNewtonResidual
FinalizeSolver(...)	void		mainSystem, simulationSettings	write concluding information (timer statistics, messages) and close files
FinishStep(...)	void		mainSystem, simulationSettings	finish static step / time step; write output of results to file
GetAEsize()	Index			number of algebraic equations in solver
GetDataSize()	Index			number of data (history) variables in solver
GetErrorString()	std::string			return error string if solver has not been successful
GetNewtonSolution()	NumpyVector			get locally stored / last computed solution (=increment) of Newton
GetODE1size()	Index			number of ODE1 equations in solver (not yet implemented)
GetODE2size()	Index			number of ODE2 equations in solver
GetSimulationEndTime(...)	Real		simulationSettings	compute simulation end time (depends on static or time integration solver)
GetSolverName()	std::string			get solver name - needed for output file header and visualization window
GetSystemJacobian()	NumpyMatrix			get locally stored / last computed system jacobian of solver
GetSystemMassMatrix()	NumpyMatrix			get locally stored / last computed mass matrix of solver
GetSystemResidual()	NumpyVector			get locally stored / last computed system residual
HasAutomaticStepSizeControl(...)	bool		mainSystem, simulationSettings	return true, if solver supports automatic stepsize control, otherwise false
IncreaseStepSize(...)	void		mainSystem, simulationSettings	increase step size if convergence is good
InitializeSolver(...)	bool		mainSystem, simulationSettings	initialize solverSpecific,data,it,conv; set/compute initial conditions (solver-specific!); initialize output files
InitializeSolverData(...)	void		mainSystem, simulationSettings	initialize all data,it,conv; called from InitializeSolver()
InitializeSolverInitialConditions(...)	void		mainSystem, simulationSettings	set/compute initial conditions (solver-specific!); called from InitializeSolver()
InitializeSolverOutput(...)	void		mainSystem, simulationSettings	initialize output files; called from InitializeSolver()
InitializeSolverPreChecks(...)	bool		mainSystem, simulationSettings	check if system is solvable; initialize dense/sparse computation modes
InitializeStep(...)	void		mainSystem, simulationSettings	initialize static step / time step; Python-functions; do some outputs, checks, etc.
IsStaticSolver()	bool			return true, if static solver; needs to be overwritten in derived class
IsVerboseCheck(...)	bool		level	return true, if file or console output is at or above the given level

loadStepGeometricFactor	Real			multiplicative load step factor; this factor is computed from loadStepGeometric parameters in SolveSystem(...)
Newton(...)	bool		mainSystem, simulationSettings	perform Newton method for given solver method
PostInitializeSolverSpecific(...)	void		mainSystem, simulationSettings	post-initialize for solver specific tasks; called at the end of InitializeSolver
PreInitializeSolverSpecific(...)	void		mainSystem, simulationSettings	pre-initialize for solver specific tasks; called at beginning of InitializeSolver, right after Solver data reset
ReduceStepSize(...)	bool		mainSystem, simulationSettings, severity	reduce step size (1..normal, 2..severe problems); return true, if reduction was successful
SetSystemJacobian(...)	void		systemJacobian	set locally stored system jacobian of solver; must have size nODE2+nODE1+nAE
SetSystemMassMatrix(...)	void		systemMassMatrix	set locally stored mass matrix of solver; must have size nODE2+nODE1+nAE
SetSystemResidual(...)	void		systemResidual	set locally stored system residual; must have size nODE2+nODE1+nAE
SolveSteps(...)	bool		mainSystem, simulationSettings	main solver part: calls multiple InitializeStep(...)/ DiscontinuousIteration(...)/ FinishStep(...); do step reduction if necessary; return true if success, false else
SolveSystem(...)	bool		mainSystem, simulationSettings	solve System: InitializeSolver, SolveSteps, FinalizeSolver
UpdateCurrentTime(...)	void		mainSystem, simulationSettings	update currentTime (and load factor); MUST be overwritten in special solver class
VerboseWrite(...)	void		level, str	write to console and/or file in case of level
WriteCoordinatesToFile(...)	void		mainSystem, simulationSettings	write unique coordinates solution file
WriteSolutionFileHeader(...)	void		mainSystem, simulationSettings	write unique file header, depending on static/ dynamic simulation

9.4.0.6 MainSolverImplicitSecondOrder

PyBind interface (trampoline) class for dynamic implicit solver. Note that this solver includes the classical Newmark method (set useNewmark True; with option of index 2 reduction) as well as the generalized-alpha method. With the interface, the dynamic implicit solver and its substructures can be accessed via Python. NOTE that except from SolveSystem(...), these functions are only intended for experienced users and they need to be handled with care, as unexpected crashes may happen if used inappropriate. Furthermore, the functions have a lot of overhead (still fast, but performance much lower than internal solver) due to Python interfaces, and should thus be used for small systems. To access the solver in Python, write:

```
solver = MainSolverImplicitSecondOrder()
```

and hereafter you can access all data and functions via 'solver'. In this solver, user functions are possible to extend the solver at certain parts, while keeping the overal C++ performance. User func-

tions, which are added with SetUserFunction...(...), have the arguments (MainSolver, MainSystem, simulationSettings), except for ComputeNewtonUpdate which adds the initial flag as an additional argument and ComputeNewtonResidual, which returns the scalar residual.
MainSolverImplicitSecondOrder has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
conv	SolverConvergenceData			all information about tolerances, errors and residua
it	SolverIterationData			all information about iterations (steps, discontinuous iteration, newton,...)
newton	NewtonSettings			copy of newton settings from timeint or staticSolver
output	SolverOutputData			output modes and timers for exporting solver information and solution
timer	CSolverTimer			timer which measures the CPU time of solver sub functions; note that solver structures can only be written indirectly, e.g., timer=dynamicSolver.timer; timer.useTimer = False; dynamicSolver.timer=timer; however, dynamicSolver.timer.useTimer cannot be written.
alphaF	Real			copy of parameter in timeIntegration.generalizedAlpha
alphaM	Real			copy of parameter in timeIntegration.generalizedAlpha
CheckInitialized(...)	bool		mainSystem	check if MainSolver and MainSystem are correctly initialized ==> otherwise raise SysError
ComputeAlgebraicEquations(...)	void		mainSystem, velocityLevel=False	compute the algebraic equations in systemResidual in range(nODE2+nODE1, nODE2+nODE1+nAE)
ComputeJacobianAE(...)	void		mainSystem, scalarFactor_ODE2=1., scalarFactor_ODE2_t=0., scalarFactor_ODE1=1., velocityLevel=False	add jacobian of algebraic equations (multiplied with factor) to systemJacobian in cSolver; the scalarFactors are scaling the derivatives w.r.t. ODE2 coordinates, ODE2_t (velocity) coordinates and ODE1 coordinates; if velocityLevel == true, the constraints are evaluated at velocity level; the scalar factors scalarFactor_ODE2=0 and scalarFactor_ODE2 are used for the same ODE2 block in the jacobian

ComputeJacobianODE1RHS(...)	void		mainSystem, scalarFactor_ODE2=1., scalarFactor_ODE2_t=0., scalarFactor_ODE1=1.	ADD jacobian of ODE1RHS (multiplied with factors for ODE2 and ODE1 coordinates) to the according rows (nODE2:nODE2+nODE1) of the exising systemJacobian in cSolver; it requires a prior call to ComputeJacobianODE2RHS(...); the scalar factors scalarFactor_ODE2=0 and scalarFactor_ODE2 are used for the same ODE2 block in the jacobian
ComputeJacobianODE2RHS(...)	void		mainSystem, scalarFactor_ODE2=1., scalarFactor_ODE2_t=0., scalarFactor_ODE1=1., computeLoadsJacobian=0	set systemJacobian to zero, size = (nODE2+nODE1+nAE) x (nODE2+nODE1+nAE), and add jacobian (multiplied with factors for ODE2 and ODE1 coordinates) of ODE2RHS to systemJacobian in cSolver; using (scalarFactor_ODE2=-1,scalarFactor_ODE2=0) gives the stiffness matrix (=derivatives of ODE2 coords) in the nODE2 x nODE2 part, while using (scalarFactor_ODE2=0,scalarFactor_ODE2=-1) gives the damping matrix (= derivatives of ODE2 velocity coordinates) in the same part; a superposition of these two parts makes sense for implicit solvers; if , Index computeLoadsJacobian=0, loads are not considered in the Jacobian computation; for , Index computeLoadsJacobian=1 the ODE2 and ODE1 derivatives of loads are included and for , Index computeLoadsJacobian=2, also the ODE2_t dependencies are added
ComputeLoadFactor(...)	Real		simulationSettings	for static solver, this is a factor in interval [0,1]; MUST be overwritten
ComputeMassMatrix(...)	void		mainSystem, scalarFactor=1.	compute systemMassMatrix (multiplied with factor) in cSolver and return mass nODE2 x nODE2 matrix
ComputeNewtonJacobian(...)	void		mainSystem, simulationSettings	compute jacobian for newton method of given solver method; store result in systemJacobian
ComputeNewtonResidual(...)	Real		mainSystem, simulationSettings	compute residual for Newton method (e.g. static or time step); store residual vector in systemResidual and return scalar residual (specific computation may depend on solver types)
ComputeNewtonUpdate(...)	void		mainSystem, simulationSettings, initial=True	compute update for currentState from newtonSolution (decrement from residual and jacobian); if initial, this is for the initial update with newtonSolution=0

ComputeODE1RHS(...)	void		mainSystem	compute the RHS of ODE1 equations in systemResidual in range(0,nODE1)
ComputeODE2RHS(...)	void		mainSystem	compute the RHS of ODE2 equations in systemResidual in range(0,nODE2)
DiscontinuousIteration(...)	bool		mainSystem, simulationSettings	perform discontinuousIteration for static step / time step; CALLS ComputeNewtonResidual
factJacAlgorithmic	Real			locally computed parameter from generalizedAlpha parameters
FinalizeSolver(...)	void		mainSystem, simulationSettings	write concluding information (timer statistics, messages) and close files
FinishStep(...)	void		mainSystem, simulationSettings	finish static step / time step; write output of results to file
GetAAlgorithmic()	NumpyVector			get locally stored / last computed algorithmic accelerations
GetAEsize()	Index			number of algebraic equations in solver
GetDataSize()	Index			number of data (history) variables in solver
GetErrorString()	std::string			return error string if solver has not been successful
GetNewtonSolution()	NumpyVector			get locally stored / last computed solution (=increment) of Newton
GetODE1size()	Index			number of ODE1 equations in solver (not yet implemented)
GetODE2size()	Index			number of ODE2 equations in solver
GetSimulationEndTime(...)	Real		simulationSettings	compute simulation end time (depends on static or time integration solver)
GetSolverName()	std::string			get solver name - needed for output file header and visualization window
GetStartOfStepStateAAlgorithmic()	NumpyVector			get locally stored / last computed algorithmic accelerations at start of step
GetSystemJacobian()	NumpyMatrix			get locally stored / last computed system jacobian of solver
GetSystemMassMatrix()	NumpyMatrix			get locally stored / last computed mass matrix of solver
GetSystemResidual()	NumpyVector			get locally stored / last computed system residual
HasAutomaticStepSizeControl(...)	bool		mainSystem, simulationSettings	return true, if solver supports automatic stepsize control, otherwise false
IncreaseStepSize(...)	void		mainSystem, simulationSettings	increase step size if convergence is good
InitializeSolver(...)	bool		mainSystem, simulationSettings	initialize solverSpecific,data,it,conv; set/compute initial conditions (solver-specific!); initialize output files
InitializeSolverData(...)	void		mainSystem, simulationSettings	initialize all data,it,conv; called from InitializeSolver()
InitializeSolverInitialConditions(...)	void		mainSystem, simulationSettings	set/compute initial conditions (solver-specific!); called from InitializeSolver()
InitializeSolverOutput(...)	void		mainSystem, simulationSettings	initialize output files; called from InitializeSolver()

InitializeSolverPreChecks(...)	bool		mainSystem, simulationSettings	check if system is solvable; initialize dense/sparse computation modes
InitializeStep(...)	void		mainSystem, simulationSettings	initialize static step / time step; Python-functions; do some outputs, checks, etc.
IsStaticSolver()	bool			return true, if static solver; needs to be overwritten in derived class
IsVerboseCheck(...)	bool		level	return true, if file or console output is at or above the given level
newmarkBeta	Real			copy of parameter in timeIntegration.generalizedAlpha
newmarkGamma	Real			copy of parameter in timeIntegration.generalizedAlpha
Newton(...)	bool		mainSystem, simulationSettings	perform Newton method for given solver method
PostInitializeSolverSpecific(...)	void		mainSystem, simulationSettings	post-initialize for solver specific tasks; called at the end of InitializeSolver
PostNewton(...)	Real		mainSystem, simulationSettings	call PostNewton for all relevant objects (contact, friction, ... iterations); returns error for discontinuous iteration
PreInitializeSolverSpecific(...)	void		mainSystem, simulationSettings	pre-initialize for solver specific tasks; called at beginning of InitializeSolver, right after Solver data reset
ReduceStepSize(...)	bool		mainSystem, simulationSettings, severity	reduce step size (1..normal, 2..severe problems); return true, if reduction was successful
SetSystemJacobian(...)	void		systemJacobian	set locally stored system jacobian of solver; must have size nODE2+nODE1+nAE
SetSystemMassMatrix(...)	void		systemMassMatrix	set locally stored mass matrix of solver; must have size nODE2+nODE1+nAE
SetSystemResidual(...)	void		systemResidual	set locally stored system residual; must have size nODE2+nODE1+nAE
SetUserFunctionComputeNewtonJacobian(...)	void		mainSystem, userFunction	set user function
SetUserFunctionComputeNewtonResidual(...)	void		mainSystem, userFunction	set user function
SetUserFunctionComputeNewtonUpdate(...)	void		mainSystem, userFunction	set user function
SetUserFunctionDiscontinuousIteration(...)	void		mainSystem, userFunction	set user function
SetUserFunctionFinishStep(...)	void		mainSystem, userFunction	set user function
SetUserFunctionInitializeStep(...)	void		mainSystem, userFunction	set user function
SetUserFunctionNewton(...)	void		mainSystem, userFunction	set user function
SetUserFunctionPostNewton(...)	void		mainSystem, userFunction	set user function
SetUserFunctionUpdateCurrentTime(...)	void		mainSystem, userFunction	set user function

SolveSteps(...)	bool		mainSystem, simulationSettings	main solver part: calls multiple InitializeStep(...)/ DiscontinuousIteration(...)/ FinishStep(...); do step reduction if necessary; return true if success, false else
SolveSystem(...)	bool		mainSystem, simulationSettings	solve System: InitializeSolver, SolveSteps, FinalizeSolver
spectralRadius	Real			copy of parameter in timeIntegration.generalizedAlpha
UpdateCurrentTime(...)	void		mainSystem, simulationSettings	update currentTime (and load factor); MUST be overwritten in special solver class
VerboseWrite(...)	void		level, str	write to console and/or file in case of level
WriteCoordinatesToFile(...)	void		mainSystem, simulationSettings	write unique coordinates solution file
WriteSolutionFileHeader(...)	void		mainSystem, simulationSettings	write unique file header, depending on static/ dynamic simulation

9.4.0.7 MainSolverExplicit

PyBind interface (trampoline) class for dynamic explicit solver. Note that this solver includes the 1st order explicit Euler scheme and the 4th order Runge-Kutta scheme with 5th order error estimation (DOPRI5). With the interface, the solver and its substructures can be accessed via Python. NOTE that except from SolveSystem(...), these functions are only intended for experienced users and they need to be handled with care, as unexpected crashes may happen if used inappropriate. Furthermore, the functions have a lot of overhead (still fast, but performance much lower than internal solver) due to Python interfaces, and should thus be used for small systems. To access the solver in Python, write
`solver = MainSolverExplicit()`
and hereafter you can access all data and functions via 'solver'. In this solver, no user functions are possible, but you can use SolverImplicitSecondOrder instead (turning off Newton gives explicit scheme ...).
MainSolverExplicit has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
conv	SolverConvergenceData			all information about tolerances, errors and residua
it	SolverIterationData			all information about iterations (steps, discontinuous iteration, newton,...)
output	SolverOutputData			output modes and timers for exporting solver information and solution
timer	CSolverTimer			timer which measures the CPU time of solver sub functions
ComputeLoadFactor(...)	Real		simulationSettings	for static solver, this is a factor in interval [0,1]; MUST be overwritten

ComputeMassMatrix(...)	void		mainSystem, scalarFactor=1.	compute systemMassMatrix (multiplied with factor) in cSolver and return mass matrix
ComputeNewtonJacobian(...)	void		mainSystem, simulationSettings	compute jacobian for newton method of given solver method; store result in systemJacobian
ComputeNewtonResidual(...)	Real		mainSystem, simulationSettings	compute residual for Newton method (e.g. static or time step); store residual vector in systemResidual and return scalar residual (specific computation may depend on solver types)
ComputeNewtonUpdate(...)	void		mainSystem, simulationSettings, initial=True	compute update for currentState from newtonSolution (decrement from residual and jacobian); if initial, this is for the initial update with newtonSolution=0
ComputeODE1RHS(...)	void		mainSystem	compute the RHS of ODE1 equations in systemResidual in range(0,nODE1)
ComputeODE2RHS(...)	void		mainSystem	compute the RHS of ODE2 equations in systemResidual in range(0,nODE2)
DiscontinuousIteration(...)	bool		mainSystem, simulationSettings	perform discontinuousIteration for static step / time step; CALLS ComputeNewtonResidual
FinalizeSolver(...)	void		mainSystem, simulationSettings	write concluding information (timer statistics, messages) and close files
FinishStep(...)	void		mainSystem, simulationSettings	finish static step / time step; write output of results to file
GetAEsize()	Index			number of algebraic equations in solver
GetDataSize()	Index			number of data (history) variables in solver
GetErrorString()	std::string			return error string if solver has not been successful
GetMethodOrder()	Index			return order of method (higher value in methods with automatic step size, e.g., DOPRI5=5)
GetNumberOfStages()	Index			return number of stages in current method
GetODE1size()	Index			number of ODE1 equations in solver (not yet implemented)
GetODE2size()	Index			number of ODE2 equations in solver
GetSimulationEndTime(...)	Real		simulationSettings	compute simulation end time (depends on static or time integration solver)
GetSolverName()	std::string			get solver name - needed for output file header and visualization window
GetSystemMassMatrix()	NumpyMatrix			get locally stored / last computed mass matrix of solver
GetSystemResidual()	NumpyVector			get locally stored / last computed system residual
HasAutomaticStepSizeControl(...)	bool		mainSystem, simulationSettings	return true, if solver supports automatic stepsize control, otherwise false
IncreaseStepSize(...)	void		mainSystem, simulationSettings	increase step size if convergence is good

InitializeSolver(...)	bool		mainSystem, simulationSettings	initialize solverSpecific,data,it,conv; set/compute initial conditions (solver-specific!); initialize output files
InitializeSolverData(...)	void		mainSystem, simulationSettings	initialize all data,it,conv; called from InitializeSolver()
InitializeSolverInitialConditions(...)	void		mainSystem, simulationSettings	set/compute initial conditions (solver-specific!); called from InitializeSolver()
InitializeSolverOutput(...)	void		mainSystem, simulationSettings	initialize output files; called from InitializeSolver()
InitializeSolverPreChecks(...)	bool		mainSystem, simulationSettings	check if system is solvable; initialize dense/sparse computation modes
InitializeStep(...)	void		mainSystem, simulationSettings	initialize static step / time step; Python-functions; do some outputs, checks, etc.
IsStaticSolver()	bool			return true, if static solver; needs to be overwritten in derived class
IsVerboseCheck(...)	bool		level	return true, if file or console output is at or above the given level
Newton(...)	bool		mainSystem, simulationSettings	perform Newton method for given solver method
PostInitializeSolverSpecific(...)	void		mainSystem, simulationSettings	post-initialize for solver specific tasks; called at the end of InitializeSolver
PreInitializeSolverSpecific(...)	void		mainSystem, simulationSettings	pre-initialize for solver specific tasks; called at beginning of InitializeSolver, right after Solver data reset
ReduceStepSize(...)	bool		mainSystem, simulationSettings, severity	reduce step size (1..normal, 2..severe problems); return true, if reduction was successful
SetSystemMassMatrix(...)	void		systemMassMatrix	set locally stored mass matrix of solver; must have size nODE2+nODE1+nAE
SetSystemResidual(...)	void		systemResidual	set locally stored system residual; must have size nODE2+nODE1+nAE
SolveSteps(...)	bool		mainSystem, simulationSettings	main solver part: calls multiple InitializeStep(...)/ DiscontinuousIteration(...)/ FinishStep(...); do step reduction if necessary; return true if success, false else
SolveSystem(...)	bool		mainSystem, simulationSettings	solve System: InitializeSolver, SolveSteps, FinalizeSolver
UpdateCurrentTime(...)	void		mainSystem, simulationSettings	update currentTime (and load factor); MUST be overwritten in special solver class
VerboseWrite(...)	void		level, str	write to console and/or file in case of level
WriteCoordinatesToFile(...)	void		mainSystem, simulationSettings	write unique coordinates solution file
WriteSolutionFileHeader(...)	void		mainSystem, simulationSettings	write unique file header, depending on static/ dynamic simulation

This section includes the reference manual for structures (such as for solvers, helper structures, etc.) and settings which are available in the python interface, e.g., simulation settings, visualization settings. The data is auto-generated from the according interfaces in order to keep fully up-to-date

with changes.

Chapter 10

Graphics and visualization

The 3D OpenGL graphics renderer window is kept simple, but useful to see the animated results of the multibody system. The graphics output is restricted to a 3D window (renderwindow) into which the renderer draws the visualization state of the `MainSystem mbs`. Note that visualization parameters can be widely changed (more than 200 parameters ...), see Section 2.4.5.

10.1 Mouse input

The following table includes the mouse functions:

Button	action	**remarks**
left mouse button	move model	keep left mouse button pressed to move the model in the current x/y plane
left mouse button	select item	mouse click on any node, object, etc. to see its basic information in status line; selection is deactivated if mouse coordinates are shown (see button F3)
right mouse button	rotate model	keep right mouse button pressed to rotate model around current current X_1/X_2 axes
right mouse button	show item dictionary	(short) press and release on item
mouse wheel	zoom	use mouse wheel to zoom (on touch screens 'pinch-to-zoom' might work as well)

Current mouse coordinates can be obtained via `SystemContainer.renderer.GetMouseCoordinates()`.

10.1.1 6D mouse

Graphics engines, especially in CAD and finite elements allow input of special 3D or 6D mouse devices. There is a basic interface for so-called 3D mouse / 6D mouse or space mouse, allowing to map the 6D joystick to translation and rotation, see `visualizationSettings.interactive.useJoystickInput` and similar options. The interface only works, if the device maps 6 coordinates to the joystick input of GLFW (tested with 3DCONNEXION mouse).

10.2 Keyboard input

The following table includes the keyboard shortcuts available in the window.

Key(s)	action	**remarks**
1,2,3,4 or 5	visualization update speed	the entered digit controls the visualization update, ranging within 0.02, 0.1 (default), 0.5, 2, and 100 seconds
CTRL+1 or SHIFT+CTRL+1	change view	set view in 1/2-plane (+SHIFT: view from opposite side)
CTRL+2 or SHIFT+CTRL+2	change view	set view in 1/3-plane (+SHIFT: view from opposite side)
CTRL+3 or SHIFT+CTRL+3	change view	set view in 2/3-plane (+SHIFT: view from opposite side)
CTRL+4 or SHIFT+CTRL+4	change view	set view in 2/1-plane (+SHIFT: view from opposite side)
CTRL+5 or SHIFT+CTRL+5	change view	set view in 3/1-plane (+SHIFT: view from opposite side)
CTRL+6 or SHIFT+CTRL+6	change view	set view in 3/2-plane (+SHIFT: view from opposite side)
A	zoom all	set zoom such that the whole scene is visible
CURSOR UP, DOWN, ...	move scene	use coursor keys to move the scene up, down, left, and right (use CTRL for small movements)
N	show/hide nodes	pressing this key switches the visibility of nodes
CTRL N	show/hide nodes numbers	pressing this key switches the visibility of nodes numbers
C	show/hide connectors	pressing this key switches the visibility of connectors
CTRL C	show/hide connector numbers	pressing this key switches the visibility of connector numbers
B	show/hide bodies	pressing this key switches the visibility of bodies
CTRL B	show/hide bodies numbers	pressing this key switches the visibility of bodies numbers
M	show/hide markers	pressing this key switches the visibility of markers
CTRL M	show/hide markers numbers	pressing this key switches the visibility of markers numbers
L	show/hide loads	pressing this key switches the visibility of loads
CTRL L	show/hide loads numbers	pressing this key switches the visibility of loads numbers
S	show/hide sensors	pressing this key switches the visibility of sensors
CTRL S	show/hide sensors numbers	pressing this key switches the visibility of sensors numbers
T	faces / edges mode	switch between faces transparent/ faces transparent + edges / only face edges / full faces with edges / only faces visible
O	change center	change center of rotation to current center of the window (affects only current plane coordinates; rotate model to ajust other coordinates)
Q	stop solver	current solver is stopped (proceeds to next simulation or end of file); after `visualizationSettings.window.reallyQuitTimeLimit` seconds a dialog opens for safety

SPACE	pause/continue simulation	pause simulation, e.g. for model inspection; if simulation is paused, it can be continued by pressing space; use SHIFT+SPACE to continuously activate 'continue simulation'
ESCAPE	close renderer	stops the simulation (and further simulations) and closes the render window (same as close window); after `visualizationSettings.window.reallyQuitTimeLimit` seconds a dialog opens for safety
X	execute command	open dialog to enter a python command (in global python scope), see Section 2.4.6; dialog may appear behind the visualization window!
V	visualization settings	open dialog to modify visualization settings, see Section 2.4.5; dialog may appear behind the visualization window!
F2	ignore keys	switch key input on / off; can be used in combination with keyPressUserFunction to make simulators
F3	show mouse coordinates	shown in status line
'.' or KEYPAD +	zoom in	zoom one step into scene (additionally press CTRL to perform small zoom step)
',' or KEYPAD -	zoom out	zoom one step out of scene (additionally press CTRL to perform small zoom step)
KEYPAD 2/8,4/6,1/9	rotate scene	about 1, 2 and 3-axis (use CTRL for small rotations)

10.3 Render state

The system container function `SC.renderer.GetState()` returns a dictionary with current information on the renderer. This information is updated whenever the renderer performs redrawing or when according changes in the renderer are performed.

When starting with an empty `mbs` and calling `SC.renderer.Start()`, the `SC.renderer.GetState()` will return a dictionary similar to:

```
{'centerPoint': [0.0, 0.0, 0.0],
'maxSceneSize': 1.0,
'zoom': 0.4000000059604645,
'currentWindowSize': [1024, 768],
'displayScaling': 1.0,
'modelRotation': [[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]],
'mouseCoordinates': [0.0, 0.0],
'openGLcoordinates': [0.0, 0.0],
'joystickPosition': [0.0, 0.0, 0.0],
'joystickRotation': [0.0, 0.0, 0.0],
'joystickAvailable': -1}
```

Note that in case that you compiled with OpenVR, there will be a separate key `openVRstate`, containing details on OpenVR, e.g., HMD pose, eye projections and controller poses. Most entries in `renderState` are having single precision due to compatibility with values entered in OpenGL. The most typical scenario for using `SC.renderer.SetState(...)` is to restore a previous view or to start

a simulation with a specific view, projection or similar. Furthermore, mouse and joystick values can be used for interactive models.

There is a set of variables, which can be actively changed by calling `SC.renderer.SetState(renderState)` with `renderState` containing a modified dictionary:

- `centerPoint`: this is a 3D vector (list/numpy-array) containing the center point for the current view; modifying this vector allows to track objects in you simulation, **however**, it is highly recommended to use `trackMarker` in `visualizationSettings.interactive` for tracking of objects!
- `rotationCenterPoint`: the centerpoint for rotation with mouse (pressing right button)
- `maxSceneSize`: this value is used in the 3D view, clipping objects nearer or farer than this size; also used for perspective view; computed automatically based on the model
- `zoom`: this factor changes the zoom for the renderer, in fact for the size of the view; this leads to smaller objects with larger zoom values
- `modelRotation`: this is the 3×3 rotation matrix used for model rotation; changing this matrix allows to rotate the model in the view; overwriting modelRotation, centerPoint and zoom with stored values allows to reset to a certain (default) view
- `projectionMatrix`: the 4×4 matrix for camera projection (as a homogeneous transformation, according to classical OpenGL standard)

Note that other items in renderState are ignored when calling `SC.renderer.SetState(renderState)`. The read only variables in `SC.renderer.GetState()` are:

- `currentWindowSize`: contains current window size, which is different from default values in visualizationSettings, if window is scaled by user
- `displayScaling`: contains display scaling (monitor scaling; content scaling) as returned by GLFW and Windows (always 1 on Linux); used internally in renderer to scale fonts
- `mouseCoordinates`: returns 2D vector of current mouse coordinates on screen
- `openGLcoordinates`: returns 3D vector of current mouse coordinates
- `joystickAvailable`: set True, if a special 6D mouse is available (only works for special hardware, e.g., 3Dconnexion space mouse)
- `joystickPosition`: contains current joystick position vector information
- `joystickRotation`: contains current joystick rotation vector information (linearized rotation angles)

10.4 GraphicsData

All graphics objects are defined by a `GraphicsData` structure. Note that currently the visualization is based on a very simple and ancient OpenGL implementation, as there is currently no simple platform independent alternative. However, most of the heavy load triangle-based operations are implemented in C++ and are realized by very efficient OpenGL commands. However, note that the number of triangles to represent the object should be kept in a feasible range (< 1000000) in order to obtain a fast response of the renderer.

Many objects include a `GraphicsData` dictionary structure for definition of attached visualization of the object. Note that objects expect a list of `GraphicsData`, which can be produced with `exudyn.graphics. ...` functions (until `Exudyn` 1.8.33 with `GraphicsData...(...)`, which are now deprecated). Note that if reading out the `GraphicsData` from the object again, it usually has a different structure sorted by types of `GraphicsData`. Typically, you can use primitives (cube, sphere, ...) or STL data to define the objects appearance. `GraphicsData` dictionaries can be created with functions provided in the utility module `exudyn.graphics`, see Section 7.8.

`GraphicsData` can be transformed into points and triangles (mesh) and can be used for contact computation, as well. **NOTE** that for correct rendering and correct contact computations, all triangle nodes must follow a strict local order and triangle normals – if defined – must point outwards, see Fig. 10.1.

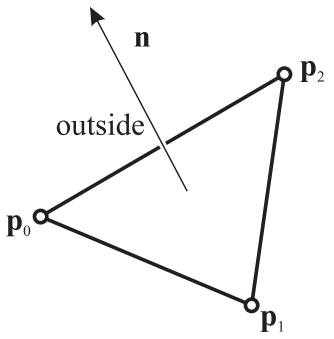


Figure 10.1: Definition of triangle normals and outside/inside regions in Exudyn: the normal to a triangle with vertex positions $\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{p}_2$ is computed from cross product as $\mathbf{n} = \frac{(\mathbf{p}_1-\mathbf{p}_0)\times(\mathbf{p}_2-\mathbf{p}_0)}{|(\mathbf{p}_1-\mathbf{p}_0)\times(\mathbf{p}_2-\mathbf{p}_0)|}$; the normal $\mathbf{n}$ then points to the outside region of the mesh or body; the direction of $\mathbf{n}$ just depends on the ordering of the vertex points (interchange of two points changes the normal direction); correct normals are needed for contact computations as well as for correct shading effects in visualization.

10.4.1 BodyGraphicsData

`BodyGraphicsData` contains a list of `GraphicsData` items, i.e. `bodyGraphicsData = [graphicsItem1, graphicsItem2, ...]`. Every single `graphicsItem` may be defined as one of the following structures using a specific 'type'. The following sections show the different possible types of `GraphicsData`.

10.4.2 GraphicsData: Line

GraphicsData `'type' = 'Line'` draws a polygonal line between all specified points:

Name	type	default value	description
color	list	[0,0,0,1]	list of 4 floats to define RGB-color and transparency
data	list	mandatory	list of float triples of x,y,z coordinates of the line floats to define RGB-color and transparency

Example:

```
#rectangle with side length 1:
graphicsData = {'type':'Line',
                'color': [1,0,0,1], #red
                'data': [0,0,0,
                         1,0,0,
                         1,1,0,
                         0,1,0,
                         0,0,0]}

vGround=VObjectGround(graphicsData=[graphicsData])
oGround=mbs.AddObject(ObjectGround(referencePosition= [0,0,0],
                                   visualization=vGround))
```

Certainly this can be done **much more elegant and shorter with** `graphics.Lines`:

```
import exudyn.graphics as graphics
graphicsData = graphics.Lines([[0,0,0],[1,0,0],[1,1,0],[0,1,0],[0,0,0]],
                              color=graphics.color.red)
```

10.4.3 GraphicsData: Lines

GraphicsData `'type': 'Lines'` draws a list of n lines defined by 2 points each:

Name	type	default value	description
colors	list	mandatory	list [R0,G0,B0,A0, R1,G2,B1,A1, ...] of $2 \times n$ x 4 floats to define RGB-color and transparency of line points
points	list	mandatory	list of $2 \times n$ float triples of x,y,z coordinates of the line points; Example for two lines: data=[0,0,0, 1,0,0, 1,0,0, 1,1,0] ... draws a L-shape with side length 1

10.4.4 GraphicsData: Circle

GraphicsData `'type' = 'Circle'` draws a polygonal line between all specified points:

Name	type	default value	description
color	list	[0,0,0,1]	list of 4 floats to define RGB-color and transparency
radius	float	mandatory	radius of circle

position	list	mandatory	list of float triples of x,y,z coordinates of center point of the circle

Example:

```
graphicsData = {'type':'Circle',
                'color': [0,0,1,1],  #blue
                'radius': 0.5,
                'position':[2,3,0]}
```

10.4.5 GraphicsData: Text

GraphicsData `'type' = 'Text'` places the given text at position:

Name	type	default value	description
color	list	[0,0,0,1]	list of 4 floats to define RGB-color and transparency
text	string	mandatory	text to be displayed, using UTF-8 encoding (see Section 10.5)
position	list	mandatory	list of float triples of [x,y,z] coordinates of the left upper position of the text; e.g. position=[20,10,0]

10.4.6 GraphicsData: TriangleList

GraphicsData `'type' = 'TriangleList'` draws a mesh with flat triangles for given points and connectivity; triangles may look smoothened by using appropriate normals; edges may be added optionally:

Name	type	default value	description
points	list	mandatory	list [x0,y0,z0, x1,y1,z1, ...] containing $n \times 3$ floats (grouped x0,y0,z0, x1,y1,z1, ...) to define x,y,z coordinates of points, n being the number of points (=vertices)
colors	list	[]	list [R0,G0,B0,A0, R1,G2,B1,A1, ...] containing $n \times 4$ floats to define RGB-color and transparency A of triangle vertices (points), where n must be according to number of points; if field 'colors' does not exist, default colors will be used
normals	list	[]	list [n0x,n0y,n0z, ...] containing $n \times 3$ floats to define normal direction of triangles per point, where n must be according to number of points; if field 'normals' does not exist, default normals [0,0,0] will be used
triangles	list	mandatory	list [T0point0, T0point1, T0point2, ...] containing $n_{trig} \times 3$ integers to define point indices of each vertex of the triangles (=connectivity); point indices start with index 0; the maximum index must be ≤ points.size()
edges	list	[]	list [L0point0, L0point1, L1point0, L1point1, ...] containing $n_{lines} \times 2$ integers to define point indices of edges drawn on triangle mesh
edgeColor	list	[0,0,0,1]	list of 4 floats to define RGB-color and transparency of edges

Examples of `GraphicsData` can be found in the Python examples and in the file `graphics.py`, see Section 7.8.

10.5 Character encoding: UTF-8

Character encoding is a major issue in computer systems, as different languages need a huge amount of different characters, see the amusing blog of Joel Spolsky:
The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode ...
More about encoding can be found in Wikipedia:UTF-8. UTF-8 encoding tables can be found within the wikipedia article and a comparison with the first 256 characters of unicode is provided at UTF-8 char table.

For short, `Exudyn` uses UTF-8 character encoding in texts / strings drawn in OpenGL renderer window. However, the set of available UTF-8 characters in `Exudyn` is restricted to a very small set of characters (as compared to available characters in UTF-8).

The following table shows of available characters (using hex codes, e.g. `0x20` = `32`):

unicode (hex code)	UTF-8 (hex code)	character
20	20	' '
21	21	'!'
...	...	...
30	30	'0'
...	...	...
39	39	'9'
40	40	'@'
41	41	'A'
...	...	...
5A	5A	'Z'
...	...	...
7E	7E	'~'
7F	7F	control, not shown
...	...	...
A0	C2 A0	no-break space
A1	C2 A1	inverted exclamation mark: '¡'
...	...	...
BF	C2 BF	inverted '¿'
C0	C3 80	A with grave
...	...	...
FF	C3 BF	y with diaeresis: 'ÿ'
special characters (selected):		
	E2 89 88	≈
	E2 88 82	∂
	E2 88 AB	$\int$
	E2 88 9A	$\surd$
	CE B1	α
	CE B2	β
	...	(complete list of greek letters see below)
	F0 9F 99 82	smiley
	F0 9F 98 92	frowney
	E2 88 9e	infinity: ∞

Note, that unicode character codes are shown only for completeness, but they **cannot be encoded in general by Exudyn**! Greek characters include: $\alpha, \beta, \gamma, \delta, \varepsilon, \zeta, \eta, \theta, \kappa, \lambda, \nu, \xi, \pi, \rho, \sigma, \varphi, \Delta, \Pi, \Sigma, \Omega$.

Chapter 11

Solvers

This section provides and overview on most important solvers. Explicit and implicit dynamic solvers, as well as optimizers are described in more detail.

11.1 Solvers in Exudyn

The user has a couple of basic solvers available in `Exudyn` , see Fig. 11.1:

- `exudyn.SolveStatic(...)`: compute static solution for given problem (may also be used to compute kinematic behavior by prescribing joint motion)
- `exudyn.SolveDynamic(...)`: time integration of equations of motion
- `exudyn.ComputeLinearizedSystem(...)`: computes the linearized system of equations and returns mass, stiffness, damping matrices
- `exudyn.ComputeODE2Eigenvalues(...)`: computes the eigenvalues of the linearized system of equations; only possible if no algebraic constraints in system; uses scipy to compute eigenvalues

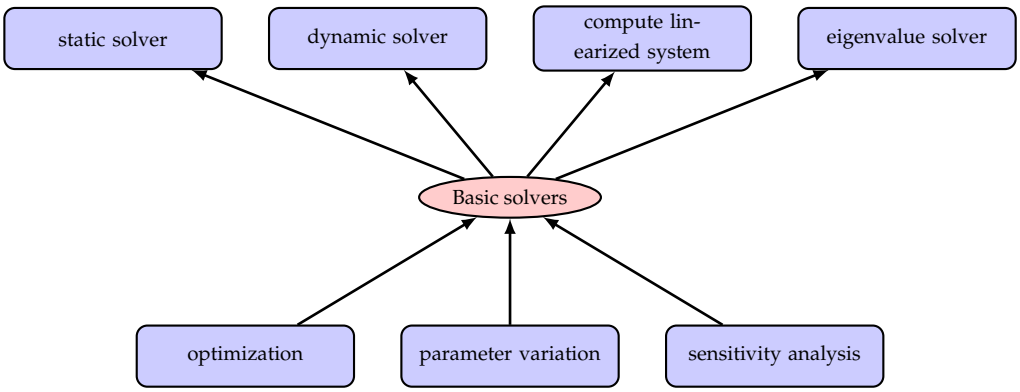


Figure 11.1: Basic and advanced solvers in `Exudyn` ; advanced solvers build upon any basic solver to perform more sophisticated operations

There are advanced solvers, like in `exudyn.processing`:

- **Optimization**:

  - `GeneticOptimization(...)`: find optimum for given set of parameter ranges using genetic optimization; works in parallel
  - `Minimize(...)`: find optimum with `scipy.optimize.minimize(...)`

- `ParameterVariation(...)`: compute a series of simulations for given set(s) of parameters; works in parallel
- `ComputeSensitivities(...)`: compute sensitivities for certain parameters; works in parallel

The advanced methods are build upon the basic solvers and essentially run single simulations in the background, see the according examples.

The basic solvers need a `MainSystem`, usually denoted as `mbs`, to be solved. Furthermore, a couple of options are usually to be given, which are explained shortly:

- `simulationSettings`: This is a big structure, containing all solver options; note that only the according options for `staticSolver` or `timeIntegration` are used. Look at the detailed description of these options in Section 9. These settings influence the output rate and output quantity of the solution, solver reporting, accuracy, solver type, etc. Specifically, the `verboseMode` may be increased (2-4) to see the behavior of the solver and intermediate quantities.
- `solverType`: Only for `exudyn.SolveDynamic(...)`: This is a simpler access to the solverType given in the internal structure of

  `timeIntegration.generalizedAlpha` and
  `simulationSettings.timeIntegration.explicitIntegration.dynamicSolverType`.

The function `exudyn.SolveDynamic(...)` sets the according variables internally. For available solver types, see the description of `exudyn.DynamicSolverType` in Section 6.10.6.

- `storeSolver`: if `True`, the solver is stored in `mbs.sys['staticSolver']` or `mbs.sys['dynamicSolver']` and also solver settings are stored in `mbs.sys['simulationSettings']`. After the solver has finished, `mbs.sys['staticSolver']` can be used to retrieve additional information on convergence, system matrices, etc. (see the solver structure).
- `showHints`: This shows a lot of possible solutions in case of no convergence
- `showCausingItems`: This shows a potential causing item if the linear solver failed; the item number is computed from the coordinate number that caused problems (e.g., a row that became zero during factorization); note that this item may not be the real cause in your problem

11.1.1 System equations of motion

The system equations of motion in `Exudyn` follow the notations of Section 4.3 and are represented as

$$\mathbf{M}\ddot{\mathbf{q}} + \frac{\partial \mathbf{g}}{\partial \mathbf{q}^T}\boldsymbol{\lambda}_q + \frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}^T}\boldsymbol{\lambda}_{\dot{q}} = \mathbf{f}_q(\mathbf{q}, \dot{\mathbf{q}}, t) \tag{11.1}$$

$$\dot{\mathbf{y}} + \frac{\partial \mathbf{g}}{\partial \mathbf{y}^T}\boldsymbol{\lambda} = \mathbf{f}_y(\mathbf{y}, t) \tag{11.2}$$

$$\mathbf{g}(\mathbf{q}, \dot{\mathbf{q}}, \mathbf{y}, \boldsymbol{\lambda}, t) = 0\,. \tag{11.3}$$

Here, we introduce different Lagrange multipliers $\boldsymbol{\lambda}_q$ and $\boldsymbol{\lambda}_{\dot{q}}$ which have equal sizes as $\boldsymbol{\lambda}$, while those λ_i which belong to holonomic constraints, are included in $\boldsymbol{\lambda}_q$ and $\boldsymbol{\lambda}_i$ belonging to non-holonomic constraints, are included in $\boldsymbol{\lambda}_{\dot{q}}$, whereas other components in $\boldsymbol{\lambda}_q$ or $\boldsymbol{\lambda}_{\dot{q}}$ are zero.

It may help to know that for linear mechanical the term $\mathbf{f}_q$ becomes

$$\mathbf{f}_q^{lin} = \mathbf{f}_a - \mathbf{K}\mathbf{q} - \mathbf{D}\dot{\mathbf{q}} \tag{11.4}$$

in which $\mathbf{f}^a$ represents applied forces and stiffness matrix $\mathbf{K}$ and damping matrix $\mathbf{D}$ become part of the system Jacobian for time integration.

11.2 General solver structure

The description of solvers in this section follows the nomenclature given in Chapter 4. Both in the static as well as in the dynamic case, the solvers run in a loop to solve a nonlinear system of (differential and/or algebraic) equations over a given time or load interval. Explicit solvers only perform a factorization of the mass matrix, but the `Newton` loop, see Fig. 11.6, is replaced by an explicit computation of the time step according to a given Runge-Kutta tableau.

In case of an implicit time integration, Fig. 11.2 shows the basic loops for the solution process. The inner loops are shown in Fig. 11.4 andFig. 11.5. The static solver behaves very similar, while no velocities or accelerations need to be solved and time is replaced by load steps.

Settings for the solver substructures, like timer, output, iterations, etc. are described in Sections 9.4.0.1 – 9.4.0.4. The description of interfaces for solvers starts in Section 9.4.0.5.

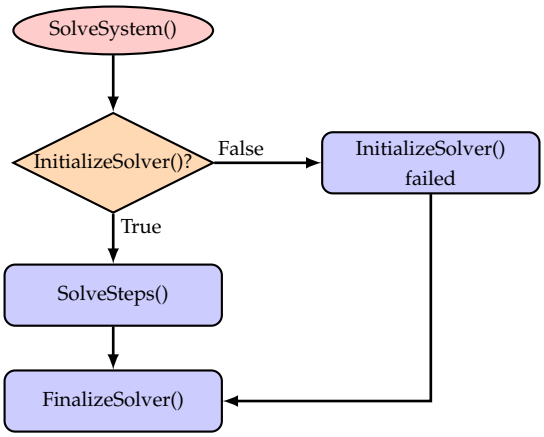


Figure 11.2: Basic solver flow chart for SolveSystem(). This flow chart is the same for static solver and for time integration.

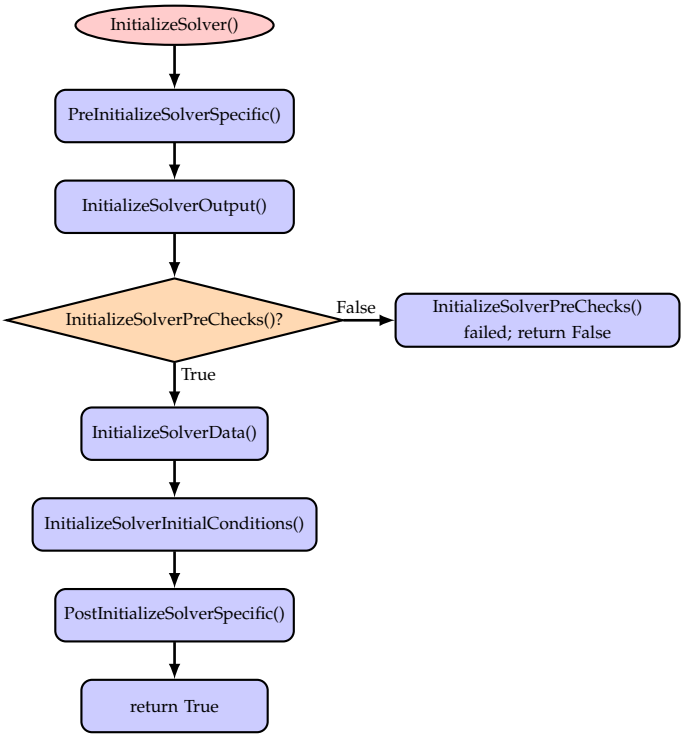


Figure 11.3: Basic solver flow chart for function InitializeSolver().

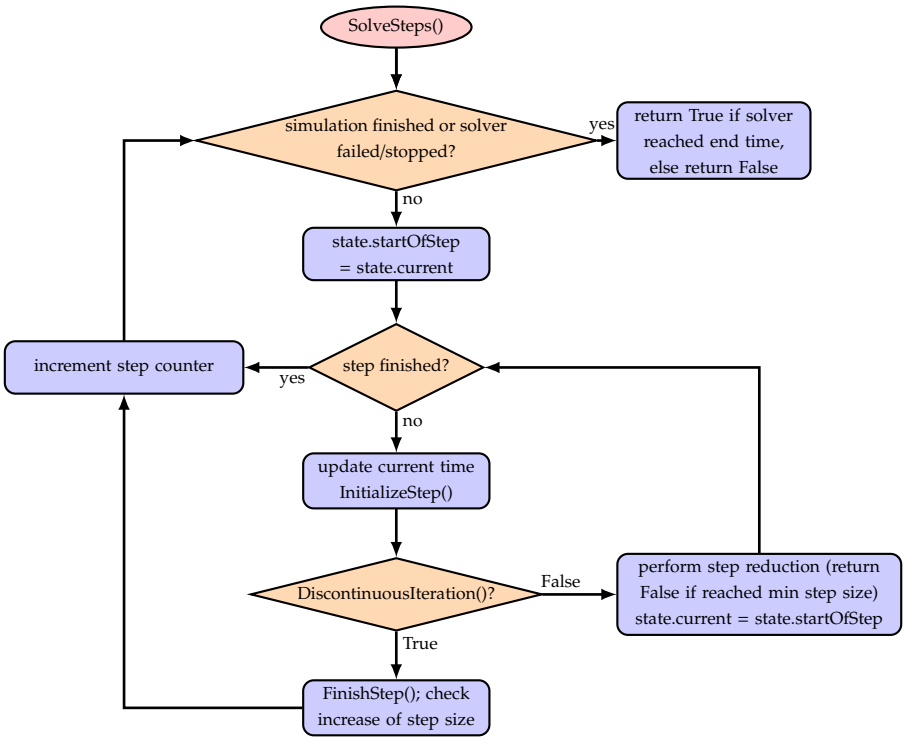


Figure 11.4: Flow chart for SolveSteps(), which is the inner loop of the solver.

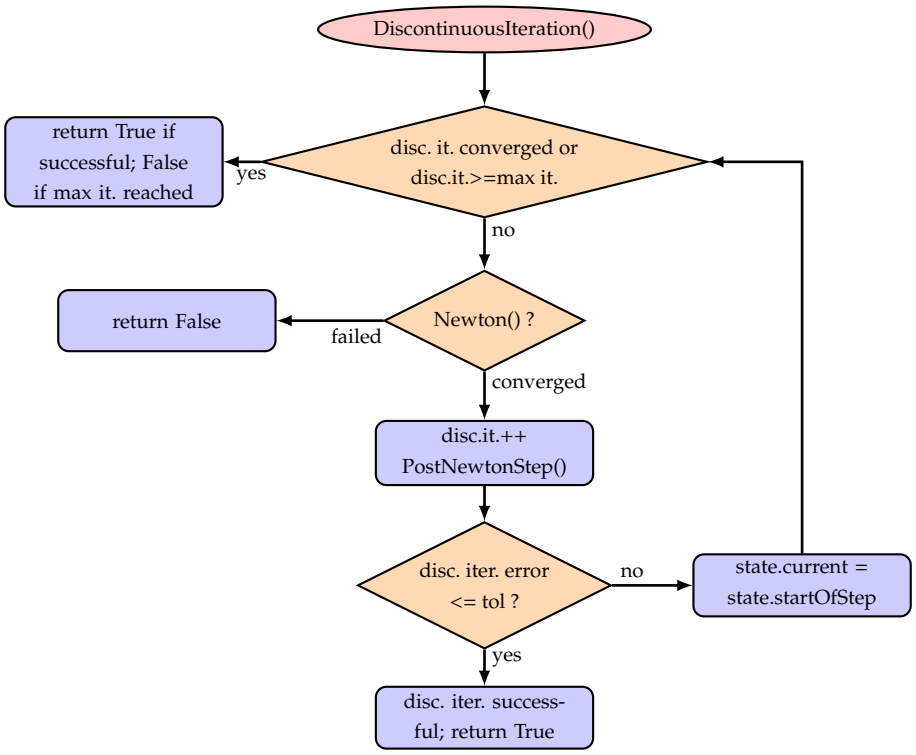


Figure 11.5: Solver flow chart for DiscontinuousIteration(), which is run for every solved step inside the static/dynamic solvers. If the DiscontinuousIteration() returns False, SolveSteps() will try to reduce the step size.

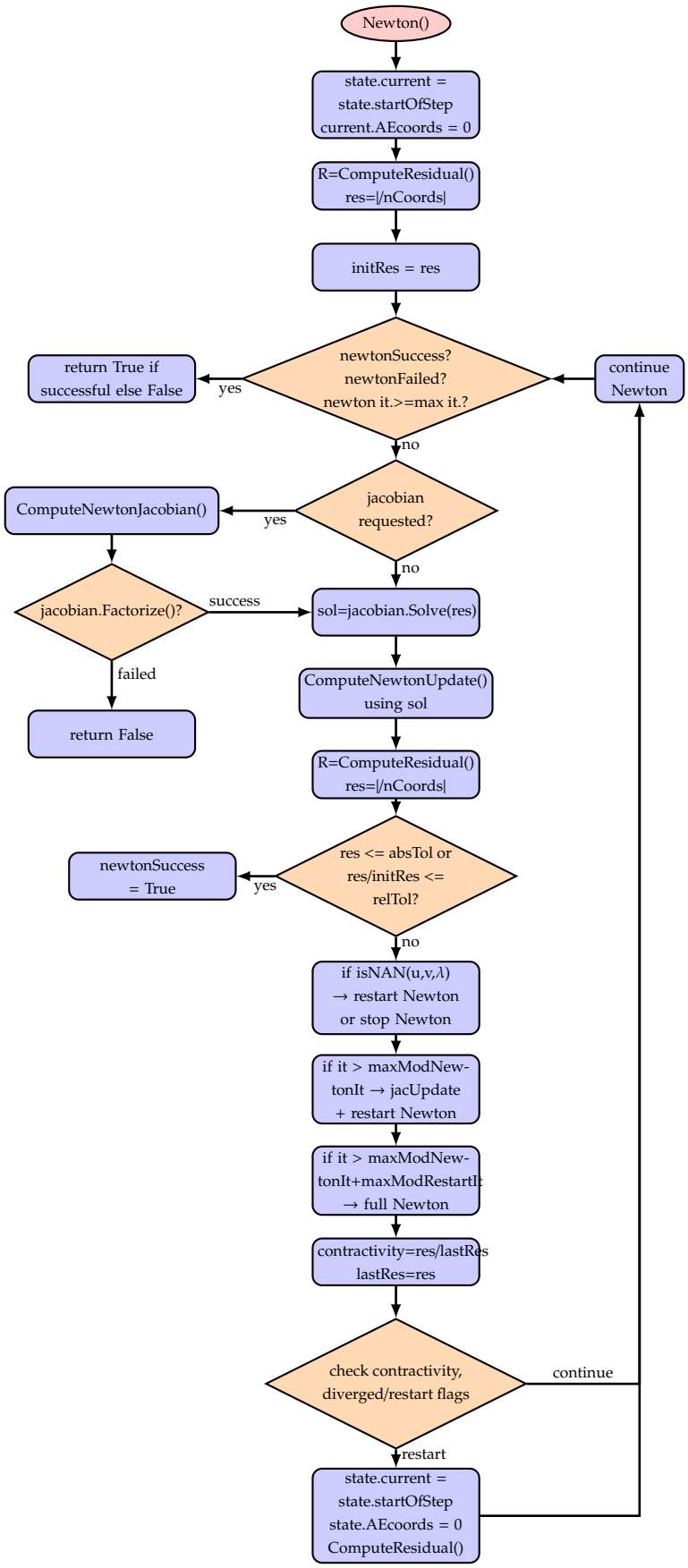


Figure 11.6: Solver flow chart for Newton(), which is run inside the DiscontinuousIteration(). The shown case is valid for newtonResidualMode = 0.

11.3 Explicit solvers

Explicit solvers are in general only applicable for systems without constraints (i.e., no joints!). However, some solvers accept simple `CoordinateConstraint`, e.g., fixing coordinates to the ground. Nevertheless, for constraint-free systems, e.g., with penalty constraints, can be solved for very high order and with great efficiency. A list of explicit solvers is available, see Section 6.10.6, for an overview of all implicit and explicit solvers.

The solution vector ξ (denoted as y in the literature [28]), which is defined as

$$\xi = [\mathbf{q}^T \quad \dot{\mathbf{q}}^T \quad \mathbf{y}^T]^T \tag{11.5}$$

and which includes ODE2 coordinates and velocities and ODE1 coordinates. All coordinates are computed without reference values.

The ODE1 and ODE2 equations of Eq. (11.6), with $\boldsymbol{\lambda} = 0$, are written in explicit form and converted to first order equations,

$$\begin{aligned} \dot{\mathbf{q}} &= \mathbf{v} \\ \dot{\mathbf{v}} &= \mathbf{M}^{-1}\mathbf{f}_q(\mathbf{q}, \mathbf{v}, t) \\ \dot{\mathbf{y}} &= \mathbf{f}_y(\mathbf{y}, t) \end{aligned} \tag{11.6}$$

$$\tag{11.7}$$

The system first order differential equations for explicit solvers thus read

$$\dot{\xi} = \mathbf{f}_e(\xi, t) \tag{11.8}$$

11.3.1 Explicit Runge-Kutta method

Explicit time integration methods seek the solution ξ_{t+h} at time $t + h$ for given initial value ξ_t (at the beginning of one step t or at the beginning of the simulation, $t = 0$),

$$\xi_{t+h} = \xi_t + \Delta\xi\,. \tag{11.9}$$

For any given Runge-Kutta method, the integration of one step with step size h is performed by an approximation

$$\Delta\xi = \int_t^{t+h} \mathbf{f}_e(\tau, \xi(\tau))d\tau \approx h\left[b_1\mathbf{f}_e(t, \xi(t)) + b_2\mathbf{f}_e(t + c_2h, \xi(t + c_2h)) + \ldots + b_s\mathbf{f}_e(t + \xi_s h, u(t + \xi_s h))\right] \tag{11.10}$$

in which $t + c_ih$ is the time for stage i and b_i the according weight given in the integration formula. Stages are within one step (therefor called one-step-methods), where $c_i = 0$ represents the beginning of the step and $c_i = 1$ the end. Note that $c_1 = 0$ for explicit integration formulas.

The unknown solution vectors ξ at the stages are abbreviated by

$$\mathbf{g}_i \approx \xi(t + c_ih) \tag{11.11}$$

and computed by explicit integration (quadrature) formulas of lower order (g_i not to be mixed up with algebraic equations!),

$$\begin{aligned}
\mathbf{g}_1 &= \xi_t \\
\mathbf{g}_2 &= \xi_t + h a_{21} \mathbf{f}_e(t, \mathbf{g}_1) \\
\mathbf{g}_3 &= \xi_t + h \left[a_{31} \mathbf{f}_e(t, \mathbf{g}_1) + a_{32} \mathbf{f}_e(t + c_2 h, \mathbf{g}_2) \right] \\
&\vdots \\
\mathbf{g}_s &= \xi_t + h \left[a_{s1} \mathbf{f}_e(t, \mathbf{g}_1) + a_{s2} \mathbf{f}_e(t + c_2 h, \mathbf{g}_2) + \ldots + a_{s,s-1} \mathbf{f}_e(t + c_{s-1} h, \mathbf{g}_{s-1}) \right]
\end{aligned} \tag{11.12}$$

After all vectors $\mathbf{g}_i$ have been consecutively evaluated, the step is updated by Eq. (11.10).

Conventional explicit Runge-Kutta solvers, such as `ExplicitMidPoint`, `RK44` or `RK67` are based on fixed step size and users must control the error by choosing an appropriate global step size. The tableaus for some lower order methods are given in Table 11.1, using the structure

$$\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^{\mathrm{T}} \end{array}$$

with

$$\begin{array}{c|ccc} 0 & 0 & & \\ c_2 & a_{21} & \ddots & \\ \vdots & \vdots & \ddots & \ddots \\ c_s & a_{s1} & \cdots & a_{s,s-1} 0 \\ \hline & b_1 & \cdots & b_{s-1} \; b_s \end{array}$$

with $\mathbf{c} = [c_0 = 0,\ c_1,\ \ldots,\ c_s]$, $\mathbf{b} = [b_0,\ b_1,\ \ldots,\ b_s]$, and $\mathbf{A}$ having only entries in the lower left triangle. For number of stages $s > 4$, the maximum order of explicit methods is lower than the number of stages, such as for `RK67`, which as order 6 but 7 stages.

11.3.2 Automatic step size control

Advanced solvers, such as `ODE23` and `DOPRI5`, include automatic step size control[1].

We estimate the error of a time step with current step size h by using an embedded Runge-Kutta formula, which includes two approximations (11.10) of order p and $\hat{p} = p - 1$, which is obtained by using two different integration formulas with common coefficients c_i, but two sets of weights b_i and $\hat{b}_i$, leading to two approximations ξ and $\hat{\xi}$. These so-called embedded Runge-Kutta formulas are widely used, for details see Hairer et al. [28].

The according apporximations ξ and $\hat{\xi}$ are used to estimate an error

$$e_j = |\xi_j - \hat{\xi}_j| \tag{11.13}$$

for every component j of the solution vector ξ. A scaling is used for every component of the solution vector, evaluating at the beginning (0) and end (1) of the time step:

$$s_j = a_{tol} + r_{tol} \cdot \max(|\xi_{0j}|, |\xi_{1j}|) \tag{11.14}$$

[1] activated with `timeIntegration.automaticStepSize = True` in simulationSettings

Explicit Euler method, number of stages $s = 1$, order $p = 1$:

0	
	1

Explicit midpoint rule, number of stages $s = 2$, order $p = 2$:

0		
1/2	1/2	
	0	1

Classical explicit Runge-Kutta method (RK44) , number of stages $s = 4$, order $p = 4$:

0				
1/2	1/2			
1/2	0	1/2		
1	0	0	1	
	1/6	1/3	1/3	1/6

Table 11.1: Several examples of tableaus for the implemented Runge-Kutta methods.

Then the relative, scaled, scalar error for the step, which needs to fulfill $err \leq 1$, is computed as

$$err = \sqrt{\frac{1}{n}\sum_{j=1}^{n}\left(\frac{\xi_{1j} - \hat{\xi}_{1j}}{s_j}\right)^2} \tag{11.15}$$

The optimal step size then reads

$$h_{opt} = h \cdot \left(\frac{1}{err}\right)^{(1/(q+1))} \tag{11.16}$$

Currently we use the suggested step size as

$$h_{new} = \min\left(h_{max}, \min\left(h \cdot f_{maxInc}, \max(h_{min}, f_{sfty} \cdot h_{opt})\right)\right) \tag{11.17}$$

With the maximum step size $h_{max} = \frac{t_{start} - t_{end}}{n_{steps}}$ and the minimum step size h_{min}, given in the `timeIntegration` `simulationSettings`. The factor f_{maxInc} limits the increase of the current step size h, the factor f_{sfty} is a safety factor for limiting the chosen step size relative to the optimal one in order to avoid frequent step rejections. If $h_{new} \leq h$, the current step is accepted, otherwise the step is recomputed with h_{new}. For more details, see Hairer et al. [28].

11.3.3 Stability limit

Note that there are hard limitations for every explicit integration method regarding the step size. Especially for stiff systems (basically with high stiffness parameters and small masses, but also with

restrictions to damping), the **step size h has an upper limit**: $h < h_{lim}$. Above that limit the method is inherently unstable, which needs to be considered both for constant and automatic step size selection.

11.3.4 Explicit Lie group integrators

All explicit solvers including the automatic step size solvers (DOPRI5, ODE23) have been equiped with Lie group integration functionality, see Holzinger et al. [31].

Basically, the integration formulas, see Section 11.3.1 are extended for special rotation parameters. Lie group integration is currently only available for `NodeRigidBodyRotVecLG` used in `ObjectRigidBody` (3D rigid body). `FFRFreducedOrder` will be extended to such nodes in the near future. To get Lie group integrators running with rigid body models, all 3D node types need to be set to `NodeRigidBodyRotVecLG` and set `explicitIntegration.useLieGroupIntegration = True`.

11.3.5 Constraints with explicit solvers

Explicit solvers generally do not solve for algebraic constraints, except for very simple `CoordinateConstraint`. All connectors having the additional `type=Constraint`, see the according object in Section 8.6.1ff., are in general not solvable by explicit solvers. Currently, only `CoordinateConstraint` with one coordinate fixed to ground can be accounted for, if `explicitIntegration.eliminateConstraints == True`. However, this offers the great flexibility to compute finite elements (imported meshes or ANCF beams) to be (partially) fixed to ground. A `CoordinateConstraint` that fixes a coordinate with index j to ground leads to the simple algebraic ODE2 equation

$$g_j(\mathbf{q}) = 0 \quad \Leftrightarrow \quad q_j = 0 \tag{11.18}$$

which can be solved by the implemented explicit solvers by just setting $q_j = 0$ previously to every computation and $\dot{q}_j = 0$ after every RHS evaluation.

NOTE that, if `explicitIntegration.eliminateConstraints == False`, constraints are ignored by the explicit solver (and all algebraic variables are set to zero). This may be wanted (e.g. to investigate the free motion of bodies), but in general leads to wrong and meaningless solution.

11.4 Implicit trapezoidal rule-based, Newmark and Generalized-alpha solver

This solver represents a class of solvers, which are – in the undamped case – based on the implicit trapezoidal rule (in the view of Runge-Kutta methods). The interpolation of the quantities for one step includes the start and the end value of the time step, thus being called trapezoidal integration rule. In some special cases in Newmark's method [45], the interpolation might only depend on the start value or the end value.

For now, all implemented solvers can be viewed as a generalization of Newmark's method, but there are called differently in the solver interfaces

- **Implicit trapezoidal rule** (Newmark with $\beta = \frac{1}{4}$ and $\gamma = \frac{1}{2}$)

- **Newmark's method** [45]
- **Generalized-α method** (= generalized Newmark method with additional parameters), see Chung and Hulbert [8] for the original method and Arnold and Brüls [1] for the application to multibody system dynamics.

11.4.1 Newmark and Generalized-alpha method

Newmark's method has two parameters β and γ. The main ideas are given in the following. First, displacements and velocities are linearly interpolated using the accelerations $\ddot{\mathbf{q}}$ of the beginning of the time step (subindex '0') and the end of the time step (subindex 'T'). The 2nd order differential equations displacements and velocities and for 1st order differential equations coordinates are given by (definition of $\mathbf{a}$ will become clear later):

$$\begin{aligned} \mathbf{q}_T &= \mathbf{q}_0 + h\dot{\mathbf{q}}_0 + h^2(\frac{1}{2} - \beta)\mathbf{a}_0 + h^2\beta\mathbf{a}_T \\ \dot{\mathbf{q}}_T &= \dot{\mathbf{q}}_0 + h(1-\gamma)\mathbf{a}_0 + h\gamma\mathbf{a}_T \\ \mathbf{y}_T &= \mathbf{y}_0 + h(1-\gamma_y)\mathbf{v}_y^0 + h\gamma_y\mathbf{v}_y^T \end{aligned} \tag{11.19}$$

Hereafter, the system equations are solved at the end of the time step (T) for the unknown accelerations as well as for 1st order differential equations and algebraic equations coordinates.
Remarks:

- The system of equations may be solved for accelerations $\ddot{\mathbf{q}}$, but also for displacements $\mathbf{q}$ or even velocities as unknowns while the remaining quantities are reconstructed from Eq. (11.19). In case of displacements as unknowns, a scaling of the Jacobian is necessary, see later.
- For consistency reasons, one may set $\gamma_y = \gamma$, but **currently we use** $\gamma_T = \frac{1}{2}$, leading to no numerical damping for ODE1 variables $\mathbf{y}$.
- In the extension to the so-called generalized-α method [8], algorithmic accelerations $\mathbf{a}$ are used in Eq. (11.19).
- Algorithmic accelerations are no longer equivalent to the time derivatives of displacements, $\mathbf{a} \neq \ddot{\mathbf{q}}$; thus, both sets of variables are used independently. In case of Newmark or the implicit trapezoidal rule just use $\mathbf{a} = \ddot{\mathbf{q}}$.
- Implicit solvers are also available with Lie groups, if according rigid body nodes (`NodeRigidBodyRotVecLG`) are used, for theory see Holzinger et al. [31].

For generalized-α, the algorithmic accelerations $\mathbf{a}$ are computed from the recurrence relation

$$(1-\alpha_m)\mathbf{a}_T + \alpha_m\mathbf{a}_0 = (1-\alpha_f)\ddot{\mathbf{u}}_T + \alpha_f\ddot{\mathbf{u}}_0 \tag{11.20}$$

which can be resolved for the unknown $\mathbf{a}_T$,

$$\mathbf{a}_T = \frac{(1-\alpha_f)\ddot{\mathbf{u}}_T + \alpha_f\ddot{\mathbf{u}}_0 - \alpha_m\mathbf{a}_0}{(1-\alpha_m)} \tag{11.21}$$

For the first step, one can simply use $\mathbf{a}_0 = \ddot{\mathbf{q}}_0$.

11.4.2 Parameter selection for Generalized-alpha

Compared to alternative implicit integration methods (including the Newmark method), the generalized-α integrator's parameters break down to one single parameter ρ_∞, which allows to chose numerical damping in a practical way.

Based on a simple single DOF mass-spring-damper model [3], having the eigen frequency $\omega = 2\pi f$ with frequency f and period $T = 1/f$, the spectral radius ρ for the integrator defines the amount of damping for a given step size h related to T, thus using the dimensionless step size $\bar{h} = h/T$.

In Fig. 11.7 the spectral radius is shown versus $\bar{h}$ for various spectral radii at infinity ρ_∞. Here, ρ_∞ specifies the numerical damping of very time step for large step sizes (or very high frequencies). An amount of $\rho_\infty = 0.9$ means that high frequency parts of the system ($\bar{h} \gg 1$); high compared to the step rate) are damped to 90% in every step, reducing an initial value 1 to $2.66e-5$ after 100 steps, which is already much larger than usual physical damping in many cases.

Furthermore, low frequency parts of the system ($\bar{h} \ll 1$) receive almost no numerical damping, see again Fig. 11.7. Exemplarily, consider ρ a low frequency situation with different ρ_∞:

- $\rho(\bar{h} = 0.01, \rho_\infty = 0.9) = 1 - 1.13 \cdot 10^{-9}$
- $\rho(\bar{h} = 0.01, \rho_\infty = 0.6) = 1 - 1.22 \cdot 10^{-7}$

which shows that numerical damping is very low for moderately small step sizes (100 steps for one oscillation).

Obviously, ρ_∞ does not have a large influence for very high or low frequencies in the system as long as it is $\neq 1$ and we could even use $\rho_\infty = 0$. Regarding differential algebraic equations (DAEs), $\rho_\infty < 1$ allows to integrate index 3 DAEs. Typically a value of $\rho_\infty = 0.7$ leads to a stable integration, but values depend on the structure of the multibody system.

Once having chosen ρ_∞, all other parameters follow automatically [8], regarding the αs

$$\alpha_m = \frac{2\rho_\infty - 1}{\rho_\infty + 1}, \quad \alpha_f = \frac{\rho_\infty}{\rho_\infty + 1} \tag{11.22}$$

and Newmarks's parameters,

$$\gamma = \frac{1}{2} - \alpha_m + \alpha_f, \quad \beta = \frac{1}{4}(1 - \alpha_m + \alpha_f)^2 \tag{11.23}$$

11.4.3 Newton iteration

Thus, the residuals at the end of the time step (T) read (put all terms to LHS):

$$\mathbf{r}_q^{G\alpha} = \mathbf{M}\ddot{\mathbf{q}}_T + \frac{\partial \mathbf{g}}{\partial \mathbf{q}^T}\boldsymbol{\lambda}_T - \mathbf{f}_q(\mathbf{q}_T, \dot{\mathbf{q}}_T, t) = 0 \tag{11.24}$$

$$\mathbf{r}_y^{G\alpha} = \dot{\mathbf{y}}_T + \frac{\partial \mathbf{g}}{\partial \mathbf{y}^T}\boldsymbol{\lambda}_T - \mathbf{f}_y(\mathbf{y}_T, t) = 0 \tag{11.25}$$

$$\mathbf{r}_\lambda^{G\alpha} = \mathbf{g}(\mathbf{q}_T, \dot{\mathbf{q}}_T, \mathbf{y}_T, \boldsymbol{\lambda}_T, t) = 0 \tag{11.26}$$

We consider two options for 2$^{\text{nd}}$ order differential equations: (A) solve for unknown accelerations $\ddot{\mathbf{q}}_T$, or (B) for unknown displacements $\mathbf{q}_T$.

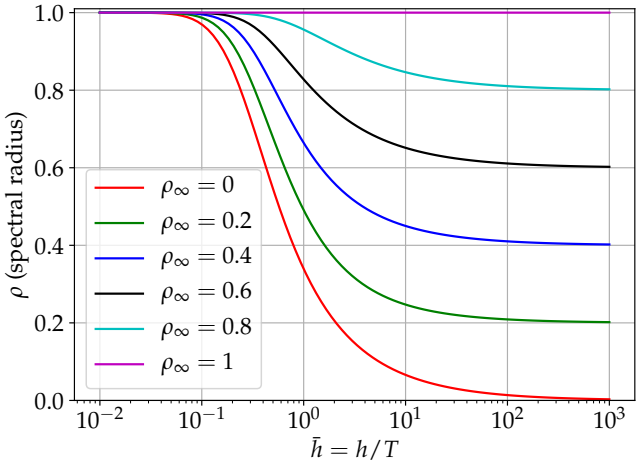


Figure 11.7: Spectral radius for generalized-α method depending on dimensionless step size $\bar{h} = h/T$, in which T is the period of an equivalent single DOF mass-spring-damper system.

11.4.3.1 (A) Solve for unknown accelerations

The unknowns for the Newton method then are

$$\boldsymbol{\xi}_{k+1}^{G\alpha} = \begin{bmatrix} \ddot{\mathbf{q}}_T \\ \mathbf{y}_T \\ \boldsymbol{\lambda}_T \end{bmatrix} \tag{11.27}$$

and at the beginning of the step, we have

$$\boldsymbol{\xi}_{k}^{G\alpha} = \begin{bmatrix} \ddot{\mathbf{q}}_0 \\ \mathbf{y}_0 \\ \boldsymbol{\lambda}_0 \end{bmatrix} \tag{11.28}$$

For the Newton method, we need to compute an update for the unknowns of Eq. (11.27), using the previous residual $\mathbf{r}_k$ and the inverse of the Jacobian $\mathbf{J}_k$ of Newton iteration k,

$$\boldsymbol{\xi}_{k+1}^{G\alpha} = \boldsymbol{\xi}_{k}^{G\alpha} - \mathbf{J}^{-1}\left(\boldsymbol{\xi}_{k}^{G\alpha}\right) \cdot \mathbf{r}^{G\alpha}\left(\boldsymbol{\xi}_{k}^{G\alpha}\right) \tag{11.29}$$

The Jacobian has the following 3×3 structure,

$$\mathbf{J} = \begin{bmatrix} \mathbf{J}_{qq} & \mathbf{J}_{qy} & \mathbf{J}_{q\lambda} \\ \mathbf{J}_{yq} & \mathbf{J}_{yy} & \mathbf{J}_{y\lambda} \\ \mathbf{J}_{\lambda q} & \mathbf{J}_{\lambda y} & \mathbf{J}_{\lambda\lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{J}_{qq} & \mathbf{0} & \mathbf{J}_{q\lambda} \\ \mathbf{0} & \mathbf{J}_{yy} & \mathbf{J}_{y\lambda} \\ \mathbf{J}_{\lambda q} & \mathbf{J}_{\lambda y} & \mathbf{J}_{\lambda\lambda} \end{bmatrix} \tag{11.30}$$

in which we consider $\mathbf{J}_{yq}$ and $\mathbf{J}_{qy}$ to vanish in the current implementations, which means that coupling of ODE1 and ODE2 coordinates is only possible due to algebraic equations.

The remaining terms in the Jacobian are currently (or by default settings) evaluated as:

$$
\begin{aligned}
\mathbf{J}_{qq} &= \frac{\partial \mathbf{r}_q^{G\alpha}}{\partial \ddot{\mathbf{q}}} = \frac{\partial \mathbf{r}_q^{G\alpha}}{\partial \mathbf{q}}\frac{\partial \mathbf{q}}{\partial \ddot{\mathbf{q}}} + \frac{\partial \mathbf{r}_q^{G\alpha}}{\partial \dot{\mathbf{q}}}\frac{\partial \dot{\mathbf{q}}}{\partial \ddot{\mathbf{q}}} = h^2\beta\mathbf{K} + h\gamma\mathbf{D} \\
\mathbf{J}_{q\lambda} &= \frac{\partial \mathbf{r}_q^{G\alpha}}{\partial \boldsymbol{\lambda}} = \frac{\partial \mathbf{g}}{\partial \mathbf{q}} \quad \left(\text{or } \frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}} \text{ for constraints at velocity level}\right) \\
\mathbf{J}_{yy} &= \frac{\partial \mathbf{r}_y^{G\alpha}}{\partial \mathbf{y}} \\
\mathbf{J}_{\lambda q} &= \frac{\partial \mathbf{r}_\lambda^{G\alpha}}{\partial \ddot{\mathbf{q}}} = \frac{\partial \mathbf{g}}{\partial \ddot{\mathbf{q}}} = \frac{\partial \mathbf{g}}{\partial \mathbf{q}}\frac{\partial \mathbf{q}}{\partial \ddot{\mathbf{q}}} + \frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}}\frac{\partial \dot{\mathbf{q}}}{\partial \ddot{\mathbf{q}}} = h^2\beta\frac{\partial \mathbf{g}}{\partial \mathbf{q}} + h\gamma\frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}} \\
\mathbf{J}_{\lambda y} &= \frac{\partial \mathbf{r}_\lambda^{G\alpha}}{\partial \mathbf{y}} \\
\mathbf{J}_{\lambda\lambda} &= \frac{\partial \mathbf{r}_\lambda^{G\alpha}}{\partial \boldsymbol{\lambda}} = \frac{\partial \mathbf{g}}{\partial \boldsymbol{\lambda}}
\end{aligned}
\tag{11.31}
$$

Note that some parts of the Jacobian are **neglected**, such as mass matrix and constraint Jacobian terms in $\mathbf{J}_{qq}$, which are usually of minor influence. Furthermore, Jacobians for state-dependent loads are neglected except for system-wide numerical Jacobians or if `computeLoadsJacobian` in static or time integration solvers is set True.

Once an update $\mathbf{q}_{k+1}^{\text{Newton}}$ has been computed, the interpolation formulas (11.19) need to be evaluated before the next residual and Jacobian can be computed.

11.4.3.2 (B) Solve for unknown displacements

This approach is similar to the previous approach and follows exactly the algorithm given by Arnold and Brüls [1], however, extended for ODE1 variables, which are integrated by the (undamped) trapezoidal rule. Documentation will be added lateron.

11.4.4 Initial accelerations

For the solvers based on the implicit trapezoidal rule, initial accelerations are necessary in order to significantly increase the accuracy of the first time step. For this reason, the constraints $\mathbf{g}(\mathbf{q}_0, \dot{\mathbf{q}}_0, \mathbf{y}_0, \boldsymbol{\lambda}_0, t) = 0$ in Eq. (11.1) are differentiated w.r.t. time,

$$
\dot{\mathbf{g}}(\mathbf{q}_0, \dot{\mathbf{q}}_0, \mathbf{y}_0, \boldsymbol{\lambda}_0, t) = \frac{\partial \mathbf{g}}{\partial \mathbf{q}}\dot{\mathbf{q}}_0 + \frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}}\ddot{\mathbf{q}}_0 + \frac{\partial \mathbf{g}}{\partial \mathbf{y}}\dot{\mathbf{y}}_0 + \frac{\partial \mathbf{g}}{\partial \boldsymbol{\lambda}}\dot{\boldsymbol{\lambda}} + \frac{\partial \mathbf{g}}{\partial t} = 0\,. \tag{11.32}
$$

Currently, we assume $\frac{\partial \mathbf{g}}{\partial \boldsymbol{\lambda}} = 0$ for all further derivations on initial accelerations. For velocity level constraints, Eq. (11.32) is used to extract initial accelerations $\ddot{\mathbf{q}}_0$,

$$
\frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}}\ddot{\mathbf{q}}_0 = -\frac{\partial \mathbf{g}}{\partial \mathbf{q}}\dot{\mathbf{q}}_0 - \frac{\partial \mathbf{g}}{\partial \mathbf{y}}\dot{\mathbf{y}}_0 - \frac{\partial \mathbf{g}}{\partial t}\,. \tag{11.33}
$$

Finally, the equations for the computation of the initial accelerations read for velocity level constraints, note that $\mathbf{y}_{init}$ are the nodal initial values for $\mathbf{y}$,

$$\begin{bmatrix} \mathbf{M} & \mathbf{0} & \frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}^T} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \ddot{\mathbf{q}}_0 \\ \mathbf{y}_0 \\ \lambda_0 \end{bmatrix} = \begin{bmatrix} \mathbf{f}_q(\mathbf{q}_T, \dot{\mathbf{q}}_T, t) \\ \mathbf{y}_{init} \\ -\frac{\partial \mathbf{g}}{\partial \mathbf{q}} \dot{\mathbf{q}}_0 - \frac{\partial \mathbf{g}}{\partial \mathbf{y}} \dot{\mathbf{y}}_0 - \frac{\partial \mathbf{g}}{\partial t} \end{bmatrix}, \tag{11.34}$$

The term $\frac{\partial \mathbf{g}}{\partial t}$ can only occur in case of user functions and therefore currently not implemented, and the ODE1 term $\frac{\partial \mathbf{g}}{\partial \mathbf{y}} = 0$ is not used yet in constraints.

For position level constraints, we assume $\frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}} = 0$ and $\frac{\partial \mathbf{g}}{\partial \mathbf{y}} = 0$ in Eq. (11.32) and perform a second derivation w.r.t. time,

$$\ddot{\mathbf{g}}(\mathbf{q}_0, \dot{\mathbf{q}}_0, \mathbf{y}_0, \lambda_0, t) = \frac{\partial^2 \mathbf{g}}{\partial \mathbf{q}^2} \dot{\mathbf{q}}_0^2 + 2 \frac{\partial^2 \mathbf{g}}{\partial \mathbf{q} \partial t} \dot{\mathbf{q}}_0 + \frac{\partial \mathbf{g}}{\partial \mathbf{q}} \ddot{\mathbf{q}}_0 + \frac{\partial^2 \mathbf{g}}{\partial t^2} = 0 \,. \tag{11.35}$$

For position level constraints, Eq. (11.35) is used to extract initial accelerations $\ddot{\mathbf{q}}_0$,

$$\frac{\partial \mathbf{g}}{\partial \mathbf{q}} \ddot{\mathbf{q}}_0 = -2 \frac{\partial^2 \mathbf{g}}{\partial \mathbf{q} \partial t} \dot{\mathbf{q}}_0 - \frac{\partial^2 \mathbf{g}}{\partial \mathbf{q}^2} \dot{\mathbf{q}}_0^2 - \frac{\partial^2 \mathbf{g}}{\partial t^2} \,. \tag{11.36}$$

Finally, the equations for the computation of the initial accelerations for position level constraints read

$$\begin{bmatrix} \mathbf{M} & \mathbf{0} & \frac{\partial \mathbf{g}}{\partial \mathbf{q}^T} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \frac{\partial \mathbf{g}}{\partial \mathbf{q}} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \ddot{\mathbf{q}}_0 \\ \mathbf{y}_0 \\ \lambda_0 \end{bmatrix} = \begin{bmatrix} \mathbf{f}_q(\mathbf{q}_T, \dot{\mathbf{q}}_T, t) \\ \mathbf{y}_{init} \\ -2 \frac{\partial^2 \mathbf{g}}{\partial \mathbf{q} \partial t} \dot{\mathbf{q}}_0 - \frac{\partial^2 \mathbf{g}}{\partial \mathbf{q}^2} \dot{\mathbf{q}}_0^2 - \frac{\partial^2 \mathbf{g}}{\partial t^2} \end{bmatrix}, \tag{11.37}$$

The linear system of equations, either Eq. (11.34) or Eq. (11.37), is solved prior to an implicit time integration if

```
simulationSettings.timeIntegration.generalizedAlpha.computeInitialAccelerations = True,
```

which is the default value.

11.5 Optimization and parameter variation

The real benefit of powerful multi-body simulation emerges only if combined with modern but also simple analysis and evaluation methods. Therefore, `Exudyn` has been integrated into the Python language, which offers a virtually unlimited number of methods of post-processing, evaluation and optimization. In this section, two methods that are directly integrated into `Exudyn` are revisited.

11.5.1 Parameter Variation

Parameter variation is one of the simplest tools to evaluate the dependency of the solution of a problem on certain parameters. This usually requires the computation of an objective (goal, result) value for a single computation (e.g, some error norm, maximum vibration amplitude, maximum stress, maximum deflection, etc.) for every computation. Furthermore, it needs to be run for a set of parameters, e.g., using a `for` loop. While this could be done manually in `Exudyn`, it is recommended to use built-in functions, which simplify evaluation and postprocessing and directly enable parallelization. The according function `ParameterVariation(...)`, see Section 7.18, performs a set of multi-dimensional parameter variations using a dictionary that describes the variation of parameters. See also `parameterVariationExample.py` in the `Examples` folder for a simple example showing a 2D parameter variation. The function `ParameterVariation(...)` requires the `multiprocessing` Python module which enables simple multi-threaded parallelism and has been tested for up to 80 cores on the LEO4 supercomputer at the University of Innsbruck, achieving a speedup of 50 as compared to a serial computation.

11.5.2 Genetic Optimization

In engineering, we often need to find a set of unknown, independent parameters $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{x}$ being denoted as design variables and $\mathbb{R}^n$ as design space. Sometimes, the design space is further subjected to constraints $\mathbf{g}(\mathbf{x}) = 0$ as well as inequalities $\mathbf{h}(x) \leq 0$, which are not considered here. For simple solutions for constrained optimization problems using penalty methods, see the introductory literature [38].

Optimization problems are written in general in the form

$$\min_{\mathbf{x}} f(\mathbf{x}), \quad \mathbf{x} \in \mathbb{R}^n \,, \tag{11.38}$$

where $f(\mathbf{x})$ denotes the *objective function* (=*fitness function*). If we would like to maximize a function $\bar{f}(\mathbf{x})$, simply set $f(\mathbf{x}) = -\bar{f}(\mathbf{x})$.

In engineering, the optimization problem could seek model parameters, e.g., the geometric dimensions and inertia parameters of a slider crank mechanism, in order to achieve smallest possible forces at the supports. Another example is the identification of unknown physical parameters, such as stiffness, damping of friction. This can be achieved by comparing measurement and simulation data (e.g., accelerations measured at relevant parts of a machine). Lets assume that $\epsilon(t)$ is an error computed in every time step of a computation, then we can set the objective (=fitness) function, e.g.,

as

$$f(x) = \frac{1}{T}\sqrt{\int_{t=0}^{T} \epsilon(t)^2 dt} \tag{11.39}$$

as the integral over the error ϵ between measurement and simulation data. In general, a parameter variation would be sufficient to compute sufficient computations for all combinations within the design space, however, a 3D design space with 100 variations into every direction (e.g., varying the unknown damping coefficient between 1 and 100, etc.) would already require 1000.000 computations, which in an ideal case of 1 second/computation leads to almost 2 weeks of computation time.

As an alternative stochastic methods can be use to compute only the objective function for a smaller set of randomly generated design variables, which usually show regions with better parameters (lower f) in scatter plots.

Genetic algorithms[26, 62] can significantly reduce the necessary amount of objective function evaluations in order to perform the optimization. Genetic identification algorithms have been already successfully applied to multibody system dynamics[11].

The general structure of a (canonical) genetic algorithm is depicted in Fig. 11.8. For details, see the

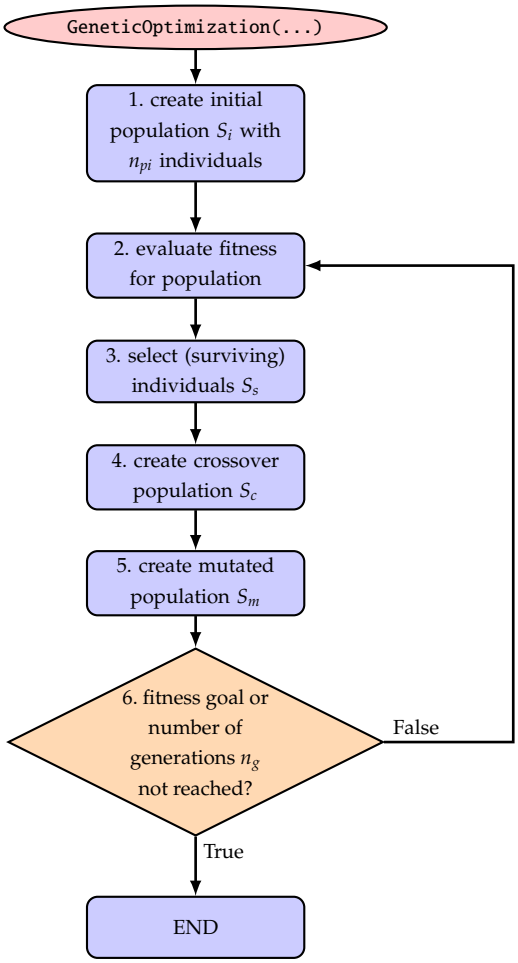


Figure 11.8: Basic solver flow chart genetic algorithm / optimization.

cited literature. Here, we focus on the implementation of the function `GeneticOptimization(...)`,

see Section 7.18. The initial population (step 1) is created with `initialPopulationSize` individuals with uniformly distributed random design variables $[\mathbf{x}_0, \ldots, \mathbf{x}_{n_{pi}-1}]$ [2] in the search space, which is given in the dictionary `parameters`. Herafter (steps 2-6), we iteratively process a population for a certain `numberOfGenerations` generations.

In step 3, the surviving individuals S_s with best fitness (smallest value from evaluation of `objectiveFunction`) are selected and considered further in the optimization. If the `distanceFactor` is used, the surviving individuals must be located within a certain distance (measured relative to the range of the search space) to all other surviving individuals. This option guarantees the search within several local minima, while a conventional search often converges to one single minimum. Crossover (step 4) is performed using a crossover of all available parameters of two randomly selected parents when generating children from the surviving individuals. The crossover of genes is performed only for a part of the new population, defined by `crossoverAmount`.

Finally, in step 5, we apply mutation to all genes, which extends the search to the surrounding design space of the individuals created by crossover. The mutation could be performed by means of certain distribution functions in order to focus on the currently best search regions. However, in the current implementation of `GeneticOptimization(...)` we simply use a uniform random variable to distribute the genes over a certain percentage of the design space, which is reduced in every generation defined by the `rangeReductionFactor` r_r. This allows us to restrict further search to a smaller subregion of the design space and in general allows a reduction of search space by means of $r_r^{n_g}$. In the ideal case, using sufficiently large population sizes and being lucky with the found random values, a range reduction factor $r_r = 0.7$ reduces the search space by a factor of 100 after every 13 generations, allowing to obtain 4 digits of accuracy for design variables after 26 generations for suitable optimization problems.

It should be noted that still this optimization method is based on random values and thus may fail occasionally for any problem case. In order to get reproducible results, set `randomizerInitialization` to any integer value (simply: 0) in order to get identical results for repeated runs. Setting the latter variable guarantees that the Python (numpy) randomizer creates the same series of random values for initial population, mutation, etc.

[2] $\mathbf{x}_i = [x_{i0}, x_{i1}, \ldots]$ being a set of genes, with single genes x_{i0}, x_{i1}, ...

Chapter 12

Issues and bugs

This section contains resolved issues per release and known bugs. Use this information to understand changes compared to previous versions. The author field is omitted if it was Johannes Gerstmayr (JG). The extension `.dev1` is not added in the issues list (e.g., 1.2.2.dev1==1.2.2), as it only marks versions that will not be available in pypi with standard pip install, but only with the `--pre` option or by specifying the exact version name, see versions on https://pypi.org/project/exudyn/. BUG numbers refer to the according issue numbers.
General information on current version:

- Exudyn version = 1.9.201.dev1,
- last change = 2025-06-18,
- Number of issues = 2104,
- Number of resolved issues = 1877 (201 in current version),

12.1 Resolved issues and resolved bugs

The following list contains the issues which have been **RESOLVED** in the according version:

Version 1.9.201: resolved Issue 2103: **Item selection** (extension)
- description: write currently selected item (+type, etc.) into renderer.state
- date resolved: **2025-06-18 17:00**, date raised: 2025-06-18

Version 1.9.200: resolved Issue 2101: **parallel** (extension)
- description: add simulationSettings.parallel.useLoadBalancing to switch two multithreading modes
- date resolved: **2025-06-18 12:00**, date raised: 2025-06-18

Version 1.9.199: resolved Issue 1695: **taskmanager** (change)
- description: extend microthreading for taskmanager-based load management; remove taskmanager from repo and create pure BSD license
- **notes: in C++ ExuThreading being now the only mode; optionally use exu.special.solver.multiThreadingLoadBalancing to switch load balancing on/off**
- date resolved: **2025-06-18 10:00**, date raised: 2023-11-19

Version 1.9.198: resolved Issue 2100: **Command window** (fix)
- description: does not show edit dialog
- date resolved: **2025-06-16 14:52**, date raised: 2025-06-16

Version 1.9.197: resolved Issue 2099: **contour** (extension)
- description: add alpha channel visualizationSettings option for contour colors
- date resolved: **2025-06-16 11:28**, date raised: 2025-06-16

Version 1.9.196: resolved Issue 2098: **graphics.CheckerBoard** (extension)

- description: add arg materialIndex for graphics material of both colors
- date resolved: **2025-06-16 11:03**, date raised: 2025-06-16

Version 1.9.195: resolved Issue 2096: **ContactSphereTriangle** (testing)

- description: add test model sphereTriangleTest.py
- date resolved: **2025-06-15 19:56**, date raised: 2025-06-15

Version 1.9.194: resolved Issue 2095: **CreateKinematicTree** (testing)

- description: add test model
- **notes: add test model createKinematicTreeTest.py**
- date resolved: **2025-06-15 19:55**, date raised: 2025-06-15

Version 1.9.193: resolved Issue 1960: **ObjectContactSphereTriangle** (extension)

- description: add contact similar to GeneralContact and to ObjectContactSphereSphere, but being able to be computed implicitly; add option to exclude certain edges to be able to also correctly handle meshes
- date resolved: **2025-06-15 11:50**, date raised: 2025-02-24

Version 1.9.192: **resolved BUG 2094**: **ContactSphereSphere**

- description: ContactSphereSphere and ContactSphereTorus set frictionRegularizedRegion wrong in case of computeFromData
- **notes: now leads to improved convergence**
- date resolved: **2025-06-15 09:37**, date raised: 2025-06-15

Version 1.9.191: resolved Issue 2092: **FEMinterface** (extension)

- description: ComputePostProcessingModesNGsolve: extend for multi-material coefficient function, using materials dict;
- date resolved: **2025-06-14 10:40**, date raised: 2025-06-13

Version 1.9.190: resolved Issue 2090: **FEMinterface** (extension)

- description: extend ImportMeshFromNGsolve to include several materials given as dict of materials
- date resolved: **2025-06-14 10:31**, date raised: 2025-06-13

Version 1.9.189: resolved Issue 2089: **FEMinterface** (extension)

- description: add function to compute nodeSets from NGsolve boundary names: CreateNGsolveBoundaryNodeSets
- date resolved: **2025-06-14 10:31**, date raised: 2025-06-13

Version 1.9.188: resolved Issue 2088: **FEMinterface** (extension)

- description: add internal function to get node numbers of NGsolve boundary condition: GetNodesOfNGsolveBoundary
- date resolved: **2025-06-14 10:31**, date raised: 2025-06-13

Version 1.9.187: resolved Issue 2093: **FEMinterface** (extension)

- description: adapt class KirchhoffMaterial for multi-domain materials
- date resolved: **2025-06-14 10:30**, date raised: 2025-06-14

Version 1.9.186: resolved Issue 2091: **FEMinterface** (change)

- description: remove deprecated arg computeEigenmodes from ImportMeshFromNGsolve
- date resolved: **2025-06-13 22:16**, date raised: 2025-06-13

Version 1.9.185: resolved Issue 2087: **GetOtherMarker** (change)

- description: return MarkerBodyRigid as body anyway must provide position and orientation
- date resolved: **2025-06-13 22:13**, date raised: 2025-06-13

Version 1.9.184: resolved Issue 2067: **CreateKinematicTree** (extension)

- description: add Create function for KinematicTree, using list of TreeLink; add special class TreeLink to itemInterface
- date resolved: **2025-06-10 08:52**, date raised: 2025-05-29

Version 1.9.183: resolved Issue 2083: **SolidOfRevolution** (extension)

- description: add check to graphics.SolidOfRevolution that order of list is correct (leads to correct inside-outside relations)
- date resolved: **2025-06-10 01:02**, date raised: 2025-06-09

Version 1.9.182: resolved Issue 2086: **RigidBodyInertia** (extension)

- description: Add GetGraphics for combined inertia and graphics generation
- date resolved: **2025-06-09 12:31**, date raised: 2025-06-09

Version 1.9.181: resolved Issue 2056: **graphics** (fix)

- description: Renderer raises warning that some objects contain inconsistencies between computed triangle normals and vertex normals
- **notes: mainly due to SolidOfRevolution**
- date resolved: **2025-06-09 01:14**, date raised: 2025-05-28

Version 1.9.180: resolved Issue 2085: **SolidOfRevolution** (change)

- description: switch triangle order in SolidOfRevolution to have consistent normals and triangles
- date resolved: **2025-06-09 01:06**, date raised: 2025-06-09

Version 1.9.179: resolved Issue 2084: **drawFaceNormals** (change)

- description: openGL.drawFaceNormals shall draw the computed normal from the triangle points, thus showing the correct orientation of triangles
- date resolved: **2025-06-09 00:48**, date raised: 2025-06-09

Version 1.9.178: resolved Issue 2082: **shadow** (fix)

- description: add lightPositionsInCameraFrame flag to shadow computation to consistently draw shadow and lights in openGL
- date resolved: **2025-06-08 18:11**, date raised: 2025-06-08

Version 1.9.177: resolved Issue 2080: **HDF5** (testing)

- description: Add test with all data types
- **notes: added example testHDF5loadSave.py**
- date resolved: **2025-06-08 18:11**, date raised: 2025-06-08

Version 1.9.176: **resolved BUG 2079**: **LoadDictFromHDF5**

- description: does not correctly convert np.array inside list
- date resolved: **2025-06-08 18:11**, date raised: 2025-06-07

Version 1.9.175: resolved Issue 2081: **GL_LIGHT** (extension)

- description: add option to switch between local and global lights, to be compatible with Raytracer
- **notes: added option openGL.lightPositionsInCameraFrame to switch behavior; in raytracer, this setting is always True**
- date resolved: **2025-06-08 15:44**, date raised: 2025-06-08

Version 1.9.174: resolved Issue 2078: **netgen** (extension)

- description: add option to convert netgen / ngsolve mesh into points, triangles and normals including smooth geometries
- date resolved: **2025-06-07 19:12**, date raised: 2025-06-07

Version 1.9.173: resolved Issue 2077: **Raytracer** (extension)

- description: light radius receives circular variation normal to ray, giving an effient and good effect of spherical lights; use lightRadius and lightRadiusVariations
- date resolved: **2025-06-05 21:03**, date raised: 2025-06-05

Version 1.9.172: resolved Issue 2076: **Raytracer** (extension)

- description: add option for spherical lights with radius and random sampling
- **notes: added lightRadius**
- date resolved: **2025-06-04 17:52**, date raised: 2025-06-04

Version 1.9.171: resolved Issue 2059: **Raytracer** (docu)

- description: add short docu part
- **notes: example available in Examples/newtonsCradle.py**
- date resolved: **2025-06-04 10:58**, date raised: 2025-05-28

Version 1.9.170: resolved Issue 2074: **invert triangles** (extension)

- description: Add function to invert triangles and normals of graphicsData, e.g., for inverted sphere or brick
- **notes: added graphics.InvertTriangles and ConsistentTriangleList**
- date resolved: **2025-06-04 01:37**, date raised: 2025-06-02

Version 1.9.169: resolved Issue 2061: **Raytracer** (extension)

- description: add materials interface via SystemContainer: renderer.SetMaterial(index, dict), dict=GetMaterial(index)
- **notes: can do read [] access, Set(), New(), etc.**
- date resolved: **2025-06-03 16:06**, date raised: 2025-05-28

Version 1.9.168: resolved Issue 2063: **Raytracer** (extension)

- description: make first 10 materials in renderer accessible via visualization systems dialog
- date resolved: **2025-06-03 16:05**, date raised: 2025-05-28

Version 1.9.167: resolved Issue 2060: **Raytracer** (extension)

- description: adjust minZ and line offset to scene dimension
- date resolved: **2025-06-02 17:45**, date raised: 2025-05-28

Version 1.9.166: resolved Issue 2058: **Raytracer** (extension)

- description: add separate visualization options; keep lights
- date resolved: **2025-06-02 17:45**, date raised: 2025-05-28

Version 1.9.165: resolved Issue 2070: **Renderer timeout** (check)

- description: check whether calling glfwWaitEventsTimeout, glfwPollEvents or glfwSwapBuffers fixes problems with redraw timeouts with Raytracing
- **notes: can be done with option raytracer.keepWindowActive**
- date resolved: **2025-06-02 17:44**, date raised: 2025-05-31

Version 1.9.164: resolved Issue 2069: **Raytracer** (extension)

- description: add global fog and material fog
- **notes: only global fog added**

- date resolved: **2025-06-02 17:44**, date raised: 2025-05-29

Version 1.9.163: resolved Issue 2068: **Raytracer** (extension)

- description: activate cutting plane for raytracer; include in function IntersectRayWithTriangles, which shall first check for cutting plane and only takes then objects behind cutting plane (if hit); color is then taken from triangle behind cutting plane-without reflections and shadows
- date resolved: **2025-06-02 17:44**, date raised: 2025-05-29

Version 1.9.162: resolved Issue 2073: **font bitmaps** (extension)

- description: Add smoothing at pixel level for base text fonts (at grey scale) to obtain smoother fonts
- date resolved: **2025-06-01 22:53**, date raised: 2025-06-01

Version 1.9.161: resolved Issue 2072: **renderer** (change)

- description: exu.StartRenderer() and exu.StopRenderer() are changed into SC.renderer.Start() and SC.renderer.Stop(), having a SystemContainer SC
- **notes: NOTE that this affects your existing models a lot!**
- date resolved: **2025-06-01 19:13**, date raised: 2025-06-01

Version 1.9.160: resolved Issue 2071: **Raytracer** (extension)

- description: draw system message texts as overlay of render image
- date resolved: **2025-06-01 18:00**, date raised: 2025-06-01

Version 1.9.159: resolved Issue 2036: **renderer** (change)

- description: put renderer-related functions into SystemContainer; preserve compatibility; consider substructure renderer in SC to collect rendering-related functionality
- date resolved: **2025-06-01 02:36**, date raised: 2025-05-21

Version 1.9.158: resolved Issue 2037: **renderer** (change)

- description: homogenize WaitForUserToContinue, DoRendererIdleTasks, DoIdleOperations and WaitForRenderEngineStopFlag as they are widely identical; homogenize functionality; note that DoRendererIdleTasks() with default args has to be replaced by renderer.DoIdleTasks(0)
- **notes: merged into SC.renderer.DoIdleTasks(); NOTE that this change affects your existing models a lot!**
- date resolved: **2025-06-01 02:35**, date raised: 2025-05-21

Version 1.9.157: resolved Issue 2062: **renderer** (extension)

- description: add renderer substructure to SystemContainer (with backlink to MainSystemContainer)
- **notes: all examples, test models, teaching models and exudyn submodules adapted; old functionality preserved so far with warnings**
- date resolved: **2025-06-01 02:34**, date raised: 2025-05-28 (resolved by: CHANGE)

Version 1.9.156: resolved Issue 2045: **visualization** (extension)

- description: use templated function to have only one single function for triangle drawing
- date resolved: **2025-05-30 01:01**, date raised: 2025-05-22

Version 1.9.155: resolved Issue 2066: **graphics** (extension)

- description: add option for special shape of graphics.Brick with rounded edges, offering a transition to an ellipsoid
- date resolved: **2025-05-30 00:59**, date raised: 2025-05-29

Version 1.9.154: resolved Issue 2049: **SetRenderState** (fix)

- description: seems to ignore autoFitScene = False and therefore does not reload zoom factor
- **notes: SetRenderState is extended by a flag which by default waits until the renderer is fully started or redrawn; this can impede performance, which is why this should be set to False in such cases**
- date resolved: **2025-05-29 16:22**, date raised: 2025-05-26

Version 1.9.153: resolved Issue 2065: **SimulationSettings** (change)

- description: timeIntegration.numberOfSteps: change type to Real in order to accept steps as Python float; add warning in solver if numberOfSteps deviates significantely from integer number
- date resolved: **2025-05-29 15:33**, date raised: 2025-05-29

Version 1.9.152: resolved Issue 2064: **exudyn.config** (change)

- description: move further options to config: SetWriteToConsole, flush always, etc.
- date resolved: **2025-05-29 14:47**, date raised: 2025-05-29

Version 1.9.151: resolved Issue 2057: **PrintDelayed** (fix)

- description: printing from visualiuation thread: buffer is never emptied; use idle tasks to do so
- date resolved: **2025-05-28 01:12**, date raised: 2025-05-28

Version 1.9.150: resolved Issue 2055: **exudyn.Print** (change)

- description: replace print commands in exudyn modules by exudyn.Print(...) in order to have common handling of file writing, etc.
- date resolved: **2025-05-28 00:08**, date raised: 2025-05-28

Version 1.9.149: **resolved BUG 2054**: **GeneticOptimization**

- description: computationIndex and parameterFunctionData do not work except for first generation
- date resolved: **2025-05-27 23:39**, date raised: 2025-05-27

Version 1.9.148: resolved Issue 2052: **exudyn.Print** (extension)

- description: function shall accept kwargs end, flush and sep in order to be more compatible with original Python print
- date resolved: **2025-05-27 22:39**, date raised: 2025-05-27

Version 1.9.147: resolved Issue 2053: **FEM** (fix)

- description: ComputePostProcessingModes raises error for larger problems due to print command
- date resolved: **2025-05-27 22:16**, date raised: 2025-05-27

Version 1.9.146: resolved Issue 2051: **Raytracer** (fix)

- description: resolve parallelization issue by removing global hit count
- date resolved: **2025-05-26 18:58**, date raised: 2025-05-26

Version 1.9.145: resolved Issue 2048: **Raytracer** (extension)

- description: add Raytracer as option; map options from openGL to Raytracer (lights, material, multiSampling, shadow, perspective, etc.)
- date resolved: **2025-05-26 10:51**, date raised: 2025-05-26

Version 1.9.144: resolved Issue 2047: **graphics.Quad** (extension)

- description: add normals; also affects graphics.CheckerBoard; normals are not needed, but may be modified in transformations
- date resolved: **2025-05-25 15:40**, date raised: 2025-05-25

Version 1.9.143: resolved Issue 2046: **Raytracer** (extension)

- description: add CPU-based software renderer / raytracer als option to render images for animations
- **notes: note that this is experimental; image resolution shall be small (start with 400x300) as long render times may lead to problems**
- date resolved: **2025-05-25 12:32**, date raised: 2025-05-25

Version 1.9.142: resolved Issue 2044: **visualization** (extension)

- description: add option to sort transparent triangles to improve quality of transparent objects; use simple depth sort for triangle midpoints
- **notes: added option openGL.depthSorting which sorts triangles by their depth for improved transparency view (but requires triangles to be small enough)**
- date resolved: **2025-05-22 16:35**, date raised: 2025-05-22

Version 1.9.141: resolved Issue 2043: **pyi / stub files** (fix)

- description: revise section for exudyn stubs in order to enable completion of exudyn.config and other exudyn functions
- **notes: seems that removing erroneous info like "special." was sufficient to work**
- date resolved: **2025-05-22 11:00**, date raised: 2025-05-22

Version 1.9.140: resolved Issue 2033: **InertiaSphere** (extension)

- description: use density and radius to initialize as alternative option
- date resolved: **2025-05-22 10:32**, date raised: 2025-05-20

Version 1.9.139: resolved Issue 2042: **exudyn** (change)

- description: move exudyn.SetLinalgOutputFormatPython(), exudyn.SetOutputPrecision(), exudyn.SetPrintDelayMilliSeconds(), exudyn.SuppressWarnings(), exudyn.InfoStat(), exudyn.GetVersionString() to according variables and functions in exudyn.config, see current docu
- date resolved: **2025-05-22 09:30**, date raised: 2025-05-22

Version 1.9.138: resolved Issue 2041: **exudyn.Solve** (change)

- description: remove exudyn.SolveStatic exudyn.SolveDynamic and exudyn.ComputeODE2Eigenvalues from exudyn, being only available in MainSystem as mbs.SolveDynamic, etc.
- date resolved: **2025-05-22 08:35**, date raised: 2025-05-22

Version 1.9.137: resolved Issue 2040: **exudyn.Demo1()** (change)

- description: and exudyn.Demo2() only available as exudyn.demos.Demo1() or .Demo2()
- date resolved: **2025-05-22 07:56**, date raised: 2025-05-22

Version 1.9.136: resolved Issue 2039: **exudyn.Go()** (change)

- description: remove function Go() which is not intended to be used
- date resolved: **2025-05-22 07:44**, date raised: 2025-05-22

Version 1.9.135: resolved Issue 2038: **exudyn.SetOutputPrecision** (change)

- description: moved to exudyn.config.outputPrecision
- date resolved: **2025-05-22 07:21**, date raised: 2025-05-22

Version 1.9.134: resolved Issue 1630: **exudyn module** (extension)

- description: consider settings instead of putting all variables globally into module
- date resolved: **2025-05-22 07:20**, date raised: 2023-06-23

Version 1.9.133: resolved Issue 2035: **exudyn.config** (change)

- description: collect specific settings into config structure, see also issue 1630
- date resolved: **2025-05-22 07:19**, date raised: 2025-05-21

Version 1.9.132: resolved Issue 2034: **Sensors.traces** (extension)

- description: add trace time span for past and future traces, to limit length of traces
- date resolved: **2025-05-20 08:19**, date raised: 2025-05-20

Version 1.9.131: resolved Issue 2032: **sensor traces** (fix)

- description: wrong description using visualizationSettings.sensorTraces in docu
- date resolved: **2025-05-20 08:19**, date raised: 2025-05-20

Version 1.9.130: resolved Issue 2028: **OutputVariables** (extension)

- description: add outputvariables to ContactSphereTorus
- date resolved: **2025-05-19 20:18**, date raised: 2025-05-18

Version 1.9.129: resolved Issue 2024: **ObjectContactFrictionCircleCable2DOld** (change)

- description: remove outdated object
- date resolved: **2025-05-18 23:35**, date raised: 2025-05-17

Version 1.9.128: resolved Issue 2030: **SolidOfRevolution** (extension)

- description: add smoothingAngle for contour below which smoothing is applied
- date resolved: **2025-05-18 15:35**, date raised: 2025-05-18

Version 1.9.127: resolved Issue 2029: **graphics** (extension)

- description: add start and end angle for minor radius of torus
- date resolved: **2025-05-18 11:24**, date raised: 2025-05-18

Version 1.9.126: resolved Issue 2022: **ContactTorusSphere** (extension)

- description: add basic contact element; typically used for ball bearings; similar to ContactSphereSphere
- date resolved: **2025-05-18 00:03**, date raised: 2025-05-16

Version 1.9.125: resolved Issue 2027: **CreateRigidBody** (change)

- description: if graphicsDataList is provided, node shall still be shown, but drawsize gets 0; this allows to easily show the node basis
- **notes: also done for CreateMassPoint**
- date resolved: **2025-05-17 23:30**, date raised: 2025-05-17

Version 1.9.124: resolved Issue 2026: **graphics Tube** (extension)

- description: add function graphics.Tube which generates graphicsData for a tube along a line of points with axis vectors
- date resolved: **2025-05-17 21:48**, date raised: 2025-05-17

Version 1.9.123: resolved Issue 2025: **Torus** (extension)

- description: add graphics function for torus
- **notes: using graphics.Tube function**
- date resolved: **2025-05-17 21:48**, date raised: 2025-05-17

Version 1.9.122: resolved Issue 1922: **ContactSphereSphere** (testint)

- description: add test model with various contact models
- date resolved: **2025-05-17 00:15**, date raised: 2024-11-02

Version 1.9.121: resolved Issue 2023: **graphics.Cylinder** (extension)

- description: add option to draw hollow cylinder
- date resolved: **2025-05-17 00:05**, date raised: 2025-05-17

Version 1.9.120: resolved Issue 2021: **ContactSphereSphere** (extension)

- description: add option to include hollow sphere - sphere contact
- date resolved: **2025-05-16 18:10**, date raised: 2025-05-16

Version 1.9.119: resolved Issue 2020: **version files** (change)

- description: remove different version files and use only a single file for .tex and .cpp
- date resolved: **2025-05-15 21:25**, date raised: 2025-05-15

Version 1.9.118: resolved Issue 2019: **Numpy2.0** (fix)

- description: Since Numpy2.0 __cpu_features__ are not available any more, therefore we need to find another way to automatically detect AVX2 features
- date resolved: **2025-05-15 20:20**, date raised: 2025-05-15

Version 1.9.117: resolved Issue 2012: **CObjectContactCurveCircles** (fix)

- description: complete CheckPreAssembleConsistency
- date resolved: **2025-05-15 08:50**, date raised: 2025-05-11

Version 1.9.116: resolved Issue 2014: **ContactCurveCircles** (extension)

- description: check damping mechanism and check negative contact forces

- date resolved: **2025-05-13 15:12**, date raised: 2025-05-11

Version 1.9.115: resolved Issue 1926: **ContactCurveCircles** (testing)
- description: add simple test model
- date resolved: **2025-05-11 22:53**, date raised: 2024-11-04

Version 1.9.114: resolved Issue 2013: **ContactCurveCircles** (extension)
- description: add visualization for contact circles
- date resolved: **2025-05-11 22:07**, date raised: 2025-05-11

Version 1.9.113: resolved Issue 2015: **ContactSphereSphere** (fix)
- description: radius unused in visualization
- date resolved: **2025-05-11 21:44**, date raised: 2025-05-11

Version 1.9.112: resolved Issue 2010: **ContactCurveCircles** (extension)
- description: change dynamic arrays to local temp arrays in CObject, similar to FFRF object
- date resolved: **2025-05-11 15:36**, date raised: 2025-05-11

Version 1.9.111: resolved Issue 2009: **ContactCurveCircles** (extension)
- description: draw active contact segments
- date resolved: **2025-05-11 12:02**, date raised: 2025-05-11

Version 1.9.110: resolved Issue 2008: **ContactCurveCircles** (extension)
- description: draw contact segments
- date resolved: **2025-05-11 12:02**, date raised: 2025-05-11

Version 1.9.109: **resolved BUG 2011**: **ContactCurveCircles**
- description: force computation not correct regarding simultaneous contact with several segments
- date resolved: **2025-05-11 12:01**, date raised: 2025-05-11

Version 1.9.108: resolved Issue 1925: **ContactCurveCircles** (example)
- description: add example of chain drive
- date resolved: **2025-05-11 12:01**, date raised: 2024-11-04

Version 1.9.107: resolved Issue 1952: **CreateTorsionalSpringDamper** (testing)
- description: add test example
- **notes: included in createFunctionsTest**
- date resolved: **2025-05-10 23:23**, date raised: 2025-02-05

Version 1.9.106: resolved Issue 2005: **realtime** (extension)
- description: measure realtime reserve in special timers
- date resolved: **2025-05-10 23:18**, date raised: 2025-05-10

Version 1.9.105: resolved Issue 2001: **Create functions** (extension)
- description: add automatism to create geometry from inertia objects by default (cylinder, sphere, brick, ...); using default colors
- date resolved: **2025-05-10 23:07**, date raised: 2025-05-08

Version 1.9.104: resolved Issue 2006: **RigidBodyInertia** (extension)
- description: add self.data dictionary which stores data of special inertia classes, such as radius of cylinder, etc.; this allows to obtain geometry information
- date resolved: **2025-05-10 20:39**, date raised: 2025-05-10

Version 1.9.103: resolved Issue 1971: **Create functions** (extension)
- description: add general test model
- date resolved: **2025-05-10 19:56**, date raised: 2025-03-05

Version 1.9.102: resolved Issue 1987: **CreateCoordinateConstraint** (extension)
- description: add option to constrain coordinate of object via its nodes; add option to constrain single coordinates or even a list of coordinates
- date resolved: **2025-05-10 19:37**, date raised: 2025-04-08

Version 1.9.101: resolved Issue 0657: **Delete item** (extension)
- description: add functionality to delete items, adding specific features to re-index nodes in objects/markers, etc. if a node, object or marker is deleted; add MaxItemNumber() function to systemData which returns unique name for items even after deletion
- **notes: resolved by reordering dependent items; no maxItem used**
- date resolved: **2025-05-10 01:31**, date raised: 2021-05-01

Version 1.9.100: resolved Issue 2004: **GetJointArgs** (extension)
- description: add a utility function, which takes an existing marker and rotationMarker and another body to compute a new marker and rotationMarker for the other body; can be directly used as joint arguments like in revolute, prismatic or generic joints
- date resolved: **2025-05-09 23:59**, date raised: 2025-05-09

Version 1.9.99: resolved Issue 2003: **GetOtherMarker** (extension)

- description: add a utility function, which the computes a new marker from the reference position of an existing marker and another body; for easier setup of markers
- date resolved: **2025-05-09 15:34**, date raised: 2025-05-09

Version 1.9.98: resolved Issue 1999: **delete** (extension)

- description: add option to delete dependent markers when deleting loads
- date resolved: **2025-05-09 08:08**, date raised: 2025-05-08

Version 1.9.97: resolved Issue 1998: **delete** (extension)

- description: add option to delete dependent markers in connectors/joints when deleting objects
- date resolved: **2025-05-09 08:08**, date raised: 2025-05-07

Version 1.9.96: resolved Issue 2000: **delete** (extension)

- description: consistently delete loads and sensors
- date resolved: **2025-05-08 20:57**, date raised: 2025-05-08

Version 1.9.95: resolved Issue 2002: **BasicTraits.h** (change)

- description: remove and add new AdvancedStuff.h and put there sort, atomics operations, iterators, etc. which are not needed everywhere
- date resolved: **2025-05-08 20:39**, date raised: 2025-05-08

Version 1.9.94: resolved Issue 1997: **delete** (extension)

- description: consistently delete nodes and markers in MainSystem
- date resolved: **2025-05-08 13:35**, date raised: 2025-05-06

Version 1.9.93: resolved Issue 1985: **CreateFunctions** (extension)

- description: check pyi files and workflows to enable code completion and navigation in Spyder with Create functions
- **notes: shall be resolved with issue 1996, as it natively derives from C++ class**
- date resolved: **2025-05-06 22:20**, date raised: 2025-04-02

Version 1.9.92: resolved Issue 1986: **ObjectMassPoint** (extension)

- description: add outputvariables rotation matrix and angular velocity (also to NodePoint) in order to allow spherical joint to be attached.
- date resolved: **2025-05-06 22:12**, date raised: 2025-04-08

Version 1.9.91: **resolved BUG 1995**: **Sensor**

- description: case SensorType::KinematicTree missing in GetTypeDependentIndex
- date resolved: **2025-05-06 17:13**, date raised: 2025-05-06

Version 1.9.90: resolved Issue 1994: **pybind11** (fix)

- description: local files using pybind11 2.6, not suitable for Numpy >= 2.0
- date resolved: **2025-05-06 16:44**, date raised: 2025-05-06

Version 1.9.89: resolved Issue 1993: **MSVC** (fix)

- description: compilation/execution of Exudyn in MSVC and python differ considerably
- date resolved: **2025-05-06 16:00**, date raised: 2025-05-06

Version 1.9.88: resolved Issue 1992: **CSystem** (fix)

- description: systemIsConsistent used instead of systemIsInteger in CheckSystemIntegrity
- date resolved: **2025-05-06 12:07**, date raised: 2025-05-06

Version 1.9.87: resolved Issue 1991: **MainSystem** (fix)

- description: Node, Object, Marker, ... functions have wrong internal range check (not including invalid index)
- date resolved: **2025-05-05 22:43**, date raised: 2025-05-05

Version 1.9.86: resolved Issue 1990: **delete object** (extension)

- description: first step to fulfill issue 657; reorder markers and sensors and assign invalid object number if object is used; add assemble check function for invalid indices
- date resolved: **2025-05-05 22:18**, date raised: 2025-05-05

Version 1.9.85: resolved Issue 1984: **FEM** (extension)

- description: Load/Save of FEM and FFRF data: default value NPY switched to NPZ due to Numpy2.x conflicts; for compatibility set mode to NPY
- **notes: all load/save functions should work with npy, npz, pkl and hdf5**
- date resolved: **2025-04-01 19:21**, date raised: 2025-04-01

Version 1.9.84: resolved Issue 1983: **FEM** (change)

- description: adjust Load/Save functions to by-default support hdf5 or pkl as npy does not work with numpy 2.0
- **notes: changed ObjectFFRFreducedOrderInterface and FEM class to by default use numpy NPZ format**
- date resolved: **2025-04-01 16:44**, date raised: 2025-04-01

Version 1.9.83: resolved Issue 1982: **SolutionViewer** (extension)

- description: add option Make mp4 to create videos with python ffmpeg lib

- date resolved: **2025-04-01 08:51**, date raised: 2025-04-01

Version 1.9.82: resolved Issue 1981: **Images2Video** (extension)

- description: add function ConvertImages2Video and dialog InteractiveImages2Video to convert images to videos directly in Python
- date resolved: **2025-04-01 02:20**, date raised: 2025-04-01

Version 1.9.81: resolved Issue 1980: **PlotSensor** (change)

- description: change to matplotlib.get_backend().lower() when comparing with matplotlib agg render mode; this shall avoid warnings in case of plots with agg in background
- date resolved: **2025-03-31 23:03**, date raised: 2025-03-31

Version 1.9.80: resolved Issue 1972: **artificialIntelligence** (fix)

- description: PreInitializeSolver always uses Generalized alpha solver, but should go in line with SetSolver
- **notes: this issue was erratic and is ignored; PreInitializeSolver shall be replaced by derived class calling SetSolver with the desired solverType; added remarks in artificialIntelligence.py**
- date resolved: **2025-03-31 22:42**, date raised: 2025-03-06

Version 1.9.79: resolved Issue 1976: **mbs.Assemble()** (change)

- description: check if assembled in SolveStatic and SolveDynamic and automatically call mbs.Assemble() prior to solver; only raise warning
- date resolved: **2025-03-31 22:15**, date raised: 2025-03-30

Version 1.9.78: resolved Issue 1979: **ComputeSystemDegreeOfFreedom** (change)

- description: change print to exudyn.Print for workflow consistency
- **notes: also fixed other print commands in solver.py**
- date resolved: **2025-03-31 21:52**, date raised: 2025-03-31

Version 1.9.77: resolved Issue 1978: **SetWriteToFile** (extension)

- description: add flushAlways (default:False) option for immediate writing to file
- date resolved: **2025-03-31 21:48**, date raised: 2025-03-31

Version 1.9.76: resolved Issue 1975: **CreateTorsionalSpringDamper** (fix)

- description: does not work for unlimitedRotations=True due to additional comma
- date resolved: **2025-03-12 15:08**, date raised: 2025-03-12

Version 1.9.75: resolved Issue 1974: **CreateSphericalJoint** (change)

- description: Make it work for bodies that do not offer a rotation matrix (mass points)
- date resolved: **2025-03-12 14:35**, date raised: 2025-03-12

Version 1.9.74: resolved Issue 1973: **GeneralContact** (change)

- description: remove frictionVelocityPenalty as it is not used; also remove macro ANCFuseFrictionPenalty
- date resolved: **2025-03-07 17:00**, date raised: 2025-03-07

Version 1.9.73: resolved Issue 1970: **CreateRollingDisc** (extension)

- description: create test model
- date resolved: **2025-03-05 22:36**, date raised: 2025-03-05

Version 1.9.72: resolved Issue 1969: **CreateRollingDisc** (extension)

- description: add create function to MainSystem
- date resolved: **2025-03-05 22:17**, date raised: 2025-03-05

Version 1.9.71: resolved Issue 1968: **GeneralContact** (extension)

- description: add option to only use dynamic search tree with no duplicated search bins
- **notes: in case of no static triangles, static searchtree is only created once and no additional operations are performed**
- date resolved: **2025-03-05 21:11**, date raised: 2025-03-03

Version 1.9.70: resolved Issue 1967: **Create functions** (change)

- description: change return values of all mbs.Create functions with joints (CreateRevoluteJoint, CreateSphericalJoint, CreatePrismaticJoint, CreateGenericJoint, CreateDistanceConstraints, ...) to the object index instead of lists
- date resolved: **2025-03-03 22:04**, date raised: 2025-03-03

Version 1.9.69: resolved Issue 1949: **RollingDiscPenalty** (extension)

- description: add create function to MainSystem
- **notes: already done with issue 1957**
- date resolved: **2025-03-03 20:41**, date raised: 2025-02-04

Version 1.9.68: resolved Issue 1966: **GeneralContact** (extension)

- description: SearchTree: add option to perform more accurate test for triangles: add simple check to see if bin is fully on one side of the triangle
- date resolved: **2025-03-02 18:50**, date raised: 2025-03-02

Version 1.9.67: resolved Issue 1895: **GeneralContact** (extension)

- description: add option to add static objects to GeneralContact and to SearchTree
- date resolved: **2025-03-02 15:42**, date raised: 2024-10-16

Version 1.9.66: resolved Issue 1965: **GeneralContact** (change)

- description: ODE2RHS timer removed and replaced with CSystem Contact:Overall timer (which includes PostNewtonStep)
- date resolved: **2025-03-02 11:46**, date raised: 2025-03-02

Version 1.9.65: resolved Issue 1962: **CuttingPlane** (extension)

- description: add simple option for cutting plane (point, normal vector, flag) to exclude triangles and other objects from graphics
- **notes: added options openGL.clippingPlaneNormal and openGL.clippingPlaneDistance to enable simple clipping**
- date resolved: **2025-02-28 21:30**, date raised: 2025-02-27 (resolved by: EXTENSION)

Version 1.9.64: resolved Issue 1964: **Cable2D** (extension)

- description: add setting useReducedOrderIntegration=2 to interface docu
- date resolved: **2025-02-28 20:57**, date raised: 2025-02-28

Version 1.9.63: resolved Issue 1963: **CreateRollingDiscPenalty** (testing)

- description: create test model
- date resolved: **2025-02-28 00:22**, date raised: 2025-02-27

Version 1.9.62: resolved Issue 1957: **RollingDiscPenalty** (testing)

- description: add test example
- date resolved: **2025-02-27 16:50**, date raised: 2025-02-09

Version 1.9.61: resolved Issue 1951: **RigidBody2D** (testing)

- description: add test for physicsCenterOfMass != 0
- date resolved: **2025-02-05 15:08**, date raised: 2025-02-05

Version 1.9.60: resolved Issue 1955: **FEM** (fix)

- description: WarnNumpy2 contains error, as it tries to compare int with str
- date resolved: **2025-02-05 15:04**, date raised: 2025-02-05

Version 1.9.59: resolved Issue 1950: **RigidBody2D** (extension)

- description: add parameter physicsCenterOfMass, similar to RigidBody
- date resolved: **2025-02-05 14:53**, date raised: 2025-02-05

Version 1.9.58: resolved Issue 1581: **mainSystemExtensions** (extension)

- description: add LinearSpringDamper and TorsionalSpringDamper
- **notes: transferred to issues 1948 and 1953**
- date resolved: **2025-02-05 08:44**, date raised: 2023-05-21

Version 1.9.57: resolved Issue 1948: **CreateTorsionalSpringDamper** (extension)

- description: add create function to MainSystem
- date resolved: **2025-02-04 13:44**, date raised: 2025-02-04

Version 1.9.56: **resolved BUG 1946**: **SphereSphereContact**

- description: error in PostNewtonStep: always uses data variables
- date resolved: **2025-02-03 15:40**, date raised: 2025-02-03

Version 1.9.55: resolved Issue 1945: **lieGroupSimplifiedKinematicRelations** (change)

- description: lieGroupSimplifiedKinematicRelations used to test more accurate Lie group solver; for now, default value set to false in order to have test suite running
- date resolved: **2025-01-28 11:53**, date raised: 2025-01-28

Version 1.9.54: resolved Issue 1944: **solver** (fix)

- description: always writes 'Solver terminated unsuccessfully' independently of success.
- date resolved: **2025-01-05 18:00**, date raised: 2025-01-05

Version 1.9.53: resolved Issue 1479: **RotationVector2RotationMatrix** (fix)

- description: both in Python and C++, fix range to 0..2*pi, as large angles cause low accuracy
- date resolved: **2024-12-01 19:13**, date raised: 2023-03-27

Version 1.9.52: resolved Issue 1651: **Python 3.11** (extension)

- description: added Python 3.11 workflows for Windows, Linux and MacOS builds (note: problems with Rosetta x86 on MacOS)
- **notes: resolved earlier also for 3.12 and 3.13**
- date resolved: **2024-12-01 12:31**, date raised: 2023-07-20

Version 1.9.51: resolved Issue 1483: **PUMA560** (fix)

- description: COM frames of KinematicTree drawn wrong: serialRobotInverseKinematics; check COM setting
- **notes: resolved already with issue 1857**
- date resolved: **2024-12-01 12:29**, date raised: 2023-03-29

Version 1.9.50: **resolved BUG 1943**: **FEMinterface**

- description: CreateNonlinearFEMObjectGenericODE2NGsolve does not work due to change in internal FEM stiffness and mass matrices, stored as scipy sparse matrices now
- date resolved: **2024-11-27 21:24**, date raised: 2024-11-27

Version 1.9.49: resolved Issue 1942: **System equations of motion** (docu)

- description: add missing term partial g/partial qDot in ODE2 part, in particular for non-holonomic constriants; this part was already implemented in a generic way since the very beginning, but missed to find the way into the documentation.
- date resolved: **2024-11-18 19:15**, date raised: 2024-11-18

Version 1.9.48: resolved Issue 1940: **NumPy2** (extension)

- description: Add warning in FEM for some functions that do not work properly with NumPy 2.x
- date resolved: **2024-11-10 19:57**, date raised: 2024-11-10

Version 1.9.47: resolved Issue 1939: **HDF5 load/save** (extension)

- description: extend to None type (but excluding numpy arrays containing None!
- date resolved: **2024-11-10 18:43**, date raised: 2024-11-10

Version 1.9.46: resolved Issue 1935: **GraphicsData functions** (change)

- description: adjust graphics.Sphere, graphics.Brick, etc. to create numpy arrays for improved efficiency
- date resolved: **2024-11-10 17:24**, date raised: 2024-11-10

Version 1.9.45: resolved Issue 1936: **GraphicsData functions** (change)

- description: adjust graphics conversion functions (graphics.Move, etc.) to handle numpy arrays as well
- date resolved: **2024-11-10 16:37**, date raised: 2024-11-10

Version 1.9.44: resolved Issue 1938: **graphics.Sphere** (fix)

- description: only works if addEdges <=1; add condition for number of edges
- date resolved: **2024-11-10 16:09**, date raised: 2024-11-10

Version 1.9.43: resolved Issue 1937: **graphics.color** (fix)

- description: type completion not working
- date resolved: **2024-11-10 15:32**, date raised: 2024-11-10

Version 1.9.42: resolved Issue 1934: **GraphicsData write** (change)

- description: GetObject now returns GraphicsData as numpy arrays to be consistent with issue 1934
- date resolved: **2024-11-10 14:26**, date raised: 2024-11-10

Version 1.9.41: resolved Issue 1933: **GraphicsData read** (change)

- description: allow that all colors, positions, triangles, points, normals, edges, ... are either lists or numpy.arrays (but flatten to 1D)
- date resolved: **2024-11-10 14:26**, date raised: 2024-11-10

Version 1.9.40: resolved Issue 1929: **GraphicsData** (extension)

- description: extend import/export function of GraphicsData for numpy arrays; speeds up load/safe significantly
- date resolved: **2024-11-10 14:26**, date raised: 2024-11-08

Version 1.9.39: resolved Issue 1908: **StaticSolver** (fix)

- description: does not write appropriate error message, but only writes: ValueError: SolveStatic terminated due to errors
- **notes: together with 1931**
- date resolved: **2024-11-09 17:33**, date raised: 2024-10-24

Version 1.9.38: resolved Issue 1930: **Solver** (change)

- description: change message when solver finishes, distinguishing between solver success and failure: 'Solver terminated unsuccessfully' or 'Solver terminated unsuccessfully'
- date resolved: **2024-11-09 17:32**, date raised: 2024-11-09

Version 1.9.37: resolved Issue 1928: **URDF** (chekc)

- description: check import from roboticstoolbox-python and pymeshlab
- date resolved: **2024-11-08 22:31**, date raised: 2024-11-08

Version 1.9.36: resolved Issue 1927: **SaveDictToHDF5** (extension)

- description: extend for saving int32, int64, float32 and float64
- date resolved: **2024-11-08 22:30**, date raised: 2024-11-08

Version 1.9.35: resolved Issue 1923: **ContactCurveCircles** (extension)

- description: add special contact element between curve defined by segments in contact with circles; 2D curves and circle co-move with rigid body marker at which curve is attached; enables Cam-follower mechanism, chain-sprocket contact, etc.
- date resolved: **2024-11-04 23:36**, date raised: 2024-11-04

Version 1.9.34: resolved Issue 1921: **ContactSphereSphere** (extension)

- description: add option to use restitution coefficient
- date resolved: **2024-11-04 23:32**, date raised: 2024-11-02

Version 1.9.33: resolved Issue 1919: **ContactSphereSphere** (extension)

- description: extend for nonlinear contact model
- date resolved: **2024-11-04 23:32**, date raised: 2024-11-02

Version 1.9.32: resolved Issue 1918: **ContactSphereSphere** (extension)

- description: add basic contact object for sphere-sphere contact, with options for linear and nonlinear contact models as well as adhesion
- date resolved: **2024-11-02 11:49**, date raised: 2024-11-02

Version 1.9.31: resolved Issue 1917: **useRecommendedStepSize** (extension)

- description: add option in timeIntegration.discontinuous to turn on/off step size recommendations for contact and other discontinuous phenomena
- date resolved: **2024-11-02 11:46**, date raised: 2024-11-02

Version 1.9.30: resolved Issue 1916: **KinematicTree** (docu)

- description: add clarification for order to application of joint offset and joint transformation (rotation)
- date resolved: **2024-10-30 10:56**, date raised: 2024-10-30

Version 1.9.29: resolved Issue 1915: **RigidBodyInertia** (extension)

- description: add warning in case of unphysical inertia parameters
- date resolved: **2024-10-29 22:26**, date raised: 2024-10-29

Version 1.9.28: resolved Issue 1914: **GeneralContact** (change)

- description: GetSystemODE2RhsContactForces: change argument reference to copy to make it more consistent with other reference access and avoid confusion with reference configuration; default behavior unchanged
- date resolved: **2024-10-27 10:13**, date raised: 2024-10-27

Version 1.9.27: resolved Issue 1913: **MainSystem.systemData** (extension)

- description: add function GetODE2CoordinatesTotal which includes reference values added to coordinates
- date resolved: **2024-10-27 10:00**, date raised: 2024-10-27

Version 1.9.26: resolved Issue 1911: **OutputVariableType** (change)

- description: change order of types, leading to different numerical values behind OutputVariableType enum (should not affect normal codes)
- **notes: affected by issue 1909**
- date resolved: **2024-10-26 19:30**, date raised: 2024-10-26

Version 1.9.25: resolved Issue 1909: **OutputVariableType** (extension)

- description: Add CoordinatesTotal which includes the reference configuration for output variables, used in sensors, or object and node outputs
- date resolved: **2024-10-26 19:26**, date raised: 2024-10-26

Version 1.9.24: resolved Issue 1505: **reference coordinates** (extension)

- description: add option to get total coordinates, being reference + current coordinates; gives 4 new configurations; use harmonized interface functions
- date resolved: **2024-10-26 19:26**, date raised: 2023-04-08

Version 1.9.23: resolved Issue 1897: **particles** (extension)

- description: Add particles module with functionality for creating densly packed particles in a box with proper regular initialization
- date resolved: **2024-10-19 20:57**, date raised: 2024-10-16

Version 1.9.22: resolved Issue 1903: **GeneralContact** (extension)

- description: GetSystemODE2RhsContactForces(...): add additional arg reference in order to allow linking to contact forces and thus allowing faster access (writing to this vector has no effect!)
- date resolved: **2024-10-19 18:59**, date raised: 2024-10-19

Version 1.9.21: resolved Issue 1902: **FEMinterface** (change)

- description: store elements as consistently as np.array instead of list of lists; check that all large arrays are np.arrays; do this also for surface list of lists
- **notes: now all previous list of lists in FEMinterface are defined to be numpy arrays for speed up of load/save; check your interfaces! GetSurfaceTriangles still returns list of lists**
- date resolved: **2024-10-19 18:48**, date raised: 2024-10-19

Version 1.9.20: resolved Issue 1629: **GetSystemState** (extension)

- description: extend behavior for returning a dictionary with all data incl. accelerations and possibly alg. accelerations for generalized-alpha solver; check also option to link coords, as in issue 1504
- **notes: added mbs.systemData.GetSystemStateDict(...) with option to return a reference (link) to system vectors, which do not require copying and allow modifications directly**
- date resolved: **2024-10-19 18:45**, date raised: 2023-06-23

Version 1.9.19: resolved Issue 1504: **Reference/link** (extension)

- description: extend mbs and systemData functions for reference, e.g., GetODE2Coordinates; use different function with "Link" extension, e.g., GetODE2CoordinatesLink

- **notes: added systemData function GetSystemStateDict which allows to obtain writeable references**
- date resolved: **2024-10-19 01:28**, date raised: 2023-04-08

Version 1.9.18: **resolved BUG 1901**: **FEM LoadFromFile**

- description: does not work correctly for forceVersion>0.
- date resolved: **2024-10-18 14:58**, date raised: 2024-10-18

Version 1.9.17: resolved Issue 1900: **MainSystem UserFunctions** (testing)

- description: Add TestModel for all kinds of PreStep, PostStep, reNewtonResidual, etc. functions
- date resolved: **2024-10-17 23:09**, date raised: 2024-10-16

Version 1.9.16: resolved Issue 1899: **SystemJacobianUserFunction** (extension)

- description: add a MainSystem user function with SetSystemJacobianUserFunction(...) which adds terms to the system jacobian; this is valuable as it may modify only few terms and add appropriate (experimental) terms in extension with the PreNewtonResidualUserFunction which otherwise would lead to bad convergence if important couplings are not added; as mentioned in the description, the solver's user functions are more general and should be also considered
- date resolved: **2024-10-17 23:09**, date raised: 2024-10-16

Version 1.9.15: resolved Issue 1882: **preNewtonResidualUserFunction** (extension)

- description: add user function called in every iteration of Newton solver in static or implicit dynamic computations
- **notes: function SetPreNewtonResidualUserFunction added to MainSystem, see description at MainSystem**
- date resolved: **2024-10-16 23:27**, date raised: 2024-10-10

Version 1.9.14: resolved Issue 0838: **GeneralContact jacobian** (extension)

- description: add jacobian and PostNewton for cable-sphere (circle) contact
- **notes: already done earlier for ANCF beam elements**
- date resolved: **2024-10-16 16:52**, date raised: 2021-12-19

Version 1.9.13: **resolved BUG 1893**: **keyPressUserFunction**

- description: not working any more!
- **notes: works, but needs to activate window.ignoreKeys in order to call user function**
- date resolved: **2024-10-15 23:21**, date raised: 2024-10-15

Version 1.9.12: resolved Issue 1717: **Symbolic** (example)

- description: add examples to testsuite; modify existing examples
- **notes: exu.symbolic used in 5 tests and examples**
- date resolved: **2024-10-15 23:08**, date raised: 2023-12-08

Version 1.9.11: resolved Issue 1718: **Mainsystem extensions** (example)

- description: modify some examples for .Create...(...) functions
- **notes: new functions usually only using CreateGenericJoint, CreateRevoluteJoint, etc.; still keeping old workflows with markers for LLM trainings**
- date resolved: **2024-10-15 23:06**, date raised: 2023-12-08

Version 1.9.10: resolved Issue 1721: **Mainsystem extensions** (change)

- description: in extension to issue 1718, change all AddRigidBody(...) functions to CreateRigidBody functionality
- date resolved: **2024-10-15 23:05**, date raised: 2023-12-08

Version 1.9.9: resolved Issue 1885: **pickleCopyMBS** (testing)

- description: add TestModel for copying and pickling (load/save) of MainSystem mbs
- date resolved: **2024-10-13 22:52**, date raised: 2024-10-11

Version 1.9.8: resolved Issue 1891: **load/save mbs** (extension)

- description: improve functionalities to load and save mbs, using GetDictionary / SetDictionary; see example pickleCopyMbs.py using pickle and HDF5 files
- date resolved: **2024-10-13 19:16**, date raised: 2024-10-13

Version 1.9.7: resolved Issue 1890: **LoadSaveHDF5** (extension)

- description: enable option to load/save Python user functions
- date resolved: **2024-10-11 22:49**, date raised: 2024-10-11

Version 1.9.6: resolved Issue 1886: **NGsolve CMS test** (testing)

- description: add test model for whole CMS functionality but loading from stored data; test all 3 new file formats for storing FEM data
- date resolved: **2024-10-11 09:00**, date raised: 2024-10-11

Version 1.9.5: resolved Issue 1887: **HDF5** (extension)

- description: add functions to advancedUtilities to load/save hierarchical dictionary data
- date resolved: **2024-10-11 01:00**, date raised: 2024-10-11

Version 1.9.4: resolved Issue 1884: **FEMinterface** (extension)

- description: extend LoadFromFile and SaveToFile for HDF5 and PKL (pickle) file formats
- date resolved: **2024-10-11 01:00**, date raised: 2024-10-11

Version 1.9.3: resolved Issue 1883: **RigidBodySpringDamper** (change)

- description: C++: change computation of outputvariables such that springForceTorqueUserFunction is not called for computation of displacement, rotation, etc., but only for forces and torques
- date resolved: **2024-10-11 00:03**, date raised: 2024-10-11

Version 1.9.2: resolved Issue 1881: **Loads visualization** (check)

- description: consider an option to show time-dependent loads, in particular in case of symbolic user functions
- **notes: added flag visualizationSettings.loads.drawWithUserFunction and test model loadUserFunctionTest.py**
- date resolved: **2024-10-10 10:48**, date raised: 2024-10-09

Version 1.9.1: resolved Issue 1723: **Symbolic** (extension)

- description: SymbolicRealVector: add EvaluateItem(i) in operators wherever possible to efficiently evaluate single components instead of all vector components
- date resolved: **2024-10-09 22:23**, date raised: 2023-12-09

Version 1.9.0: resolved Issue 1880: **MatrixContainer** (fix)

- description: when initialized with lists of lists, it prints the lists
- date resolved: **2024-10-09 11:06**, date raised: 2024-10-09

Version 1.8.81: resolved Issue 1879: **MatrixContainer** (extension)

- description: add feature to directly initialize with scipy sparse csr matrix; add new function Initialize to initialize MatrixContainer, and AddSparseMatrix for adding sparse matrices with factor
- date resolved: **2024-10-09 09:39**, date raised: 2024-10-09

Version 1.8.80: resolved Issue 1878: **robotics.mobile** (change)

- description: correct camelcase writing of mobileRobot2MBS to MobileRobot2MBS, getWheelVelocities to GetWheelVelocities, and getCartesianVelocities to GetCartesianVelocities; adjust examples
- date resolved: **2024-10-09 08:04**, date raised: 2024-10-09

Version 1.8.79: resolved Issue 1844: **AddRigidBody** (change)

- description: add warning to this and related functions for deprecation
- date resolved: **2024-10-09 07:40**, date raised: 2024-05-28

Version 1.8.78: resolved Issue 1455: **MarkerSuperElementRigidBody** (fix)

- description: fix derivative of exponential map for velocity level
- **notes: already resolved earlier**
- date resolved: **2024-10-09 07:37**, date raised: 2023-03-05

Version 1.8.77: resolved Issue 1341: **MacOS multithreading** (fix)

- description: resolve compilation problems with NGsolve taskmanager on Apple MacOS
- **notes: already resolved earlier**
- date resolved: **2024-10-09 07:36**, date raised: 2022-12-26

Version 1.8.76: resolved Issue 0496: **controller** (extension)

- description: add possibility to differentiate loads w.r.t. sensor?/object/node (use additional sensor numbers which provide dependencies); add optional dependence on sensors; integrateors using ODE1 or discrete implementation
- **notes: already resolved in version 1.6.84; available via systemData.AddODE2LoadDependencies**
- date resolved: **2024-10-09 07:33**, date raised: 2020-12-09

Version 1.8.75: resolved Issue 1877: **Create functions** (change)

- description: CreateDistanceConstraint, CreateSpringDamper and similar create functions have bodyList instead of bodyNumbers, which is used for joints; use bodyNumbers and allow bodyList as deprecated option
- **notes: kept compatibility with existing bodyList args, but will be removed in future versions**
- date resolved: **2024-10-08 23:04**, date raised: 2024-10-08

Version 1.8.74: resolved Issue 1862: **CSR functionality in FEM** (extension)

- description: change internal CSR format to scipy CSR format; add simple check for Scipy to be installed at start of FEM, set scipyInstalled=True, use CheckSciPyInstalled() function for unified errors
- date resolved: **2024-10-08 17:29**, date raised: 2024-10-02

Version 1.8.73: resolved Issue 1870: **FEMinterface** (check)

- description: check all occurances of GetStiffnessMatrix and GetMassMatrix for sparse mode now using scipy sparse matrix
- **notes: Make sure that using FEMinterface fem, fem.GetMassMatrix of fem.GetStiffnessMatrix now returns a SciPy sparse matrix, which can be converted into the previous form by using ScipySparseCSRtoCSR(fem.GetMassMatrix())**
- date resolved: **2024-10-08 17:28**, date raised: 2024-10-07

Version 1.8.72: resolved Issue 1876: **FEMinterface** (change)

- description: SaveToFile: used wrong default fileVersion=13, which should have been fileVersion=1; as we now shift to fileVersion=2, old files may be loaded with forceVersion=1
- date resolved: **2024-10-08 17:26**, date raised: 2024-10-08

Version 1.8.71: resolved Issue 1871: **MatrixContainer** (fix)

- description: fix compilation issues with pybind11 and scipy coo matrix
- date resolved: **2024-10-08 17:26**, date raised: 2024-10-07

Version 1.8.70: resolved Issue 1875: **FEMinterface** (change)

- description: the mass and stiffness matrices in FEMinterface are either None or given in SciPy-sparse csr format; this gives also a a new load/save fileVersion of FEMinterface
- date resolved: **2024-10-08 13:09**, date raised: 2024-10-08

Version 1.8.69: resolved Issue 1874: **FEMinterface** (change)

- description: SaveToFile: add version 2 which also pickles mass and stiffnessMatrix
- date resolved: **2024-10-08 11:11**, date raised: 2024-10-08

Version 1.8.68: resolved Issue 1873: **FEM module** (change)

- description: replace print(...) with exu.Print(...) commands to better control output flow
- date resolved: **2024-10-08 10:33**, date raised: 2024-10-08

Version 1.8.67: resolved Issue 1872: **MatrixContainer** (extension)

- description: SetWithSparseMatrixCSR: mark as deprecated; instead add new function SetWithSparseMatrix with additional arg factor (default=1) to multply matrix values with factor before adding
- date resolved: **2024-10-07 18:34**, date raised: 2024-10-07

Version 1.8.66: resolved Issue 1869: **FEMinterface** (change)

- description: change default values of massMatrix and stiffnessMatrix to None instead of np.zeros((0,0))
- date resolved: **2024-10-07 01:54**, date raised: 2024-10-07

Version 1.8.65: resolved Issue 1865: **MatrixContainer** (extension)

- description: add functionality to add Scipy csr matrix AddSparseMatrix(..., factor=1) with factor
- date resolved: **2024-10-07 01:35**, date raised: 2024-10-02

Version 1.8.64: resolved Issue 0840: **explicit solvers velocity verlet** (extension)

- description: add velocity verlet integration scheme in particular for particle and contact simulation
- **notes: added, but not yet tested for Lie group case!**
- date resolved: **2024-10-07 00:37**, date raised: 2021-12-19

Version 1.8.63: resolved Issue 1868: **SetWithSparseMatrixCSR** (change)

- description: change default behavior to useDenseMatrix=False, using sparse mode by default
- date resolved: **2024-10-06 22:03**, date raised: 2024-10-06

Version 1.8.62: resolved Issue 1867: **graphicsDataUtilities** (docu)

- description: adapt documentation to replace graphicsDataUtilities with graphics
- date resolved: **2024-10-06 17:24**, date raised: 2024-10-06

Version 1.8.61: resolved Issue 1866: **DOCU GraphicsData: Line** (fix)

- description: Mistake in Line example; add hint to use graphics.Lines function
- date resolved: **2024-10-06 17:06**, date raised: 2024-10-06

Version 1.8.60: resolved Issue 1854: **figure** (docu)

- description: replace figure theoryRotationsTaitBryanAngles, which has been accidentially copied
- date resolved: **2024-10-06 16:51**, date raised: 2024-06-05

Version 1.8.59: resolved BUG 1860: **ComputeODE2Eigenvalues**

- description: in case of computeComplexEigenvalues=True, the option convert2Frequencies=True computes wrong frequencies; this combination thus will be deactivated, as it makes little sense anyhow
- date resolved: **2024-09-19 16:15**, date raised: 2024-09-19

Version 1.8.58: resolved Issue 1859: **Frames** (fix)

- description: frame numbers not shown (e.g. for nodes)
- **notes: fixed together with issue 1858**
- date resolved: **2024-08-07 17:26**, date raised: 2024-08-07

Version 1.8.57: resolved Issue 1858: **KinematicTree** (fix)

- description: frame numbers not shown
- **notes: changing default showFramesNumbers=False**
- date resolved: **2024-08-07 17:26**, date raised: 2024-08-07

Version 1.8.56: resolved BUG 1857: **KinematicTree**

- description: COM not correctly drawn (drawn at joint location)
- date resolved: **2024-08-07 17:05**, date raised: 2024-08-07

Version 1.8.55: resolved Issue 1856: **FromSTLfile** (fix)

- description: function does not work for A or p being different from []
- date resolved: **2024-07-03 11:33**, date raised: 2024-07-03

Version 1.8.54: resolved Issue 1849: **tutorial** (docu)

- description: add FFRFreducedOrder with NGsolve tutorial
- date resolved: **2024-06-23 16:59**, date raised: 2024-06-02

Version 1.8.53: resolved Issue 1855: **lie group integrator** (change)

- description: add improved lie group integrator for generalized alpha, not using the simplified update and thus having higher accuracy
- date resolved: **2024-06-15 13:42**, date raised: 2024-06-15

Version 1.8.52: resolved Issue 1852: **tutorials** (docu)

- description: adapt tutorials to new exudyn.graphics structure
- date resolved: **2024-06-04 21:48**, date raised: 2024-06-04

Version 1.8.51: resolved Issue 1853: **exudyn.graphics** (docu)

- description: add documentation in python utilities for these functions
- date resolved: **2024-06-04 21:06**, date raised: 2024-06-04

Version 1.8.50: resolved Issue 1851: **artificialIntelligence** (change)

- description: adapt examples to new stable-baselines version
- **notes: tested with stable-baselines3 V1.7.0 and V2.3.2**
- date resolved: **2024-06-04 17:24**, date raised: 2024-06-04

Version 1.8.49: resolved Issue 1850: **artificialIntelligence** (change)

- description: adapt to stable-baselines3 2.3; add old mode with gym and new mode with gymnasium
- **notes: kept old mode available if gym is installed with stable-baselines without gymnasium**
- date resolved: **2024-06-04 14:59**, date raised: 2024-06-04 (resolved by: P. Manzl)

Version 1.8.48: resolved Issue 0406: **add NGsolve test** (extension)

- description: with FEMinterface, only for Python37 version
- **notes: outdated and removed (see new issue 1849)**
- date resolved: **2024-06-02 13:30**, date raised: 2020-05-22

Version 1.8.47: resolved Issue 0837: **GeneralContact jacobian** (extension)

- description: add jacobian and PostNewton for Triangle-Sphere contact
- date resolved: **2024-06-02 10:58**, date raised: 2021-12-19

Version 1.8.46: resolved Issue 1847: **GeneralContact** (fix)

- description: change several for-break commands to continue to improve contact behavior
- date resolved: **2024-06-01 20:13**, date raised: 2024-06-01

Version 1.8.45: resolved Issue 1475: **tutorial videos** (docu)

- description: add new tutorial, replace old ones
- **notes: already completed on Feb 15 2024**
- date resolved: **2024-05-20 19:13**, date raised: 2023-03-27

Version 1.8.44: resolved Issue 1474: **tutorial videos** (fix)

- description: fix gettings started video
- **notes: already completed on Feb 15 2024**
- date resolved: **2024-05-20 19:13**, date raised: 2023-03-27

Version 1.8.43: resolved Issue 1842: **add FurtherExampels folder for further examples, especially useful for training of LLMs** (extension)

- description: EXTENSION
- date resolved: **2024-05-20 19:08**, date raised: 2024-05-17

Version 1.8.42: resolved Issue 1843: **simulationSettings** (docu)

- description: correct path of simulationSettings.staticSolverSettings to simulationSettings.staticSolver in RTD description
- date resolved: **2024-05-17 18:36**, date raised: 2024-05-17

Version 1.8.41: resolved Issue 1841: **AddODE2LoadDependencies** (fix)

- description: description is wrong
- date resolved: **2024-05-17 18:29**, date raised: 2024-05-17

Version 1.8.40: resolved Issue 1840: **std::isalpha** (fix)

- description: remove std::isalpha and std::isalnum, as it does not compile on certain older compilers
- date resolved: **2024-05-13 17:08**, date raised: 2024-05-13

Version 1.8.39: resolved Issue 1838: **Examples check** (extension)

- description: add test to check whether examples basically run; use timeout and special setting to turn off all key-press waiting; exclude large examples
- date resolved: **2024-05-12 10:19**, date raised: 2024-05-11

Version 1.8.38: **resolved BUG 1839**: **ALEANCFCable2D**

- description: raises error 'ANCFCable2d:ComputeAxialStrain_t not implemented' if ForceLocal is evaluated

- **notes: results have to be verified**
- date resolved: **2024-05-11 19:18**, date raised: 2024-05-11

Version 1.8.37: resolved Issue 1187: **ALEANCFCable2D** (extension)

- description: add missing terms related to damping terms coupled with delta qALE
- **notes: already implemented and checked with paper up to come**
- date resolved: **2024-05-11 19:00**, date raised: 2022-07-06

Version 1.8.36: resolved Issue 1692: **exudyn.graphics** (fix)

- description: change GraphicsData functions in examples to exudyn.graphics
- **notes: CHECK your files, if there are any issues due to this change; in general, the previous functionality should be maintained**
- date resolved: **2024-05-11 01:18**, date raised: 2023-11-19

Version 1.8.35: resolved Issue 1825: **GraphicsDataCube** (change)

- description: rename to GraphicsDataBrick... functions; use assignment to keep previous functionality
- **notes: not changed, but introduced exudyn.graphics submodule for graphics functions. OrthoCube now available as exudyn.graphics.Brick**
- date resolved: **2024-05-11 01:16**, date raised: 2024-04-25

Version 1.8.34: resolved Issue 1691: **exudyn.graphics** (extension)

- description: map graphicsDataUtilities functions to exudyn.graphics for better readability
- **notes: NOTE that previous GraphicsData functions still work; however, it may be necessary to import exudyn.utilities if you imported exudyn.graphicsDataUtilities directly; examples and test models are adjusted to new functions; it is recommended that users switch to new exudyn.graphics functionality!**
- date resolved: **2024-05-10 19:45**, date raised: 2023-11-19

Version 1.8.33: resolved Issue 1837: **graphics** (check)

- description: test graphics submodule for future replacement of graphicsDataUtilities
- date resolved: **2024-05-10 10:09**, date raised: 2024-05-10

Version 1.8.32: resolved Issue 1832: **theory docu** (docu)

- description: add description of computation of stresses for FFRFreducedOrder
- date resolved: **2024-05-09 17:12**, date raised: 2024-05-03

Version 1.8.31: resolved Issue 1836: **ComputeLinearizedSystem** (extension)

- description: output constraint and nullspace matrices for constrained systems
- date resolved: **2024-05-05 16:18**, date raised: 2024-05-05

Version 1.8.30: resolved Issue 1835: **ComputeLinearizedSystem** (extension)

- description: add option to compute linearized system of constrained system
- date resolved: **2024-05-05 16:18**, date raised: 2024-05-05

Version 1.8.29: resolved Issue 1834: **ComputeLinearizedSystem** (change)

- description: remove sparse solver option useSparseSolver, as it is not implemented
- date resolved: **2024-05-05 16:17**, date raised: 2024-05-05

Version 1.8.28: resolved Issue 1833: **ComputeODE2EigenValues** (extension)

- description: extend to complex eigenvalues
- **notes: added TestModel complexEigenvaluesTest.py**
- date resolved: **2024-05-04 18:12**, date raised: 2024-05-03

Version 1.8.27: resolved Issue 1831: **ObjectJointRollingDisc** (change)

- description: change computation of constraints into local joint frame, enabling lateral and forward constraint independently; see old comment on constrainedAxes
- date resolved: **2024-04-29 08:03**, date raised: 2024-04-29

Version 1.8.26: resolved Issue 1830: **AddLidar** (fix)

- description: angles of sensors should be equally arranged between angleStart and angleEnd using numberOfSensors and having a sensor both at start and end angle
- date resolved: **2024-04-27 17:06**, date raised: 2024-04-27

Version 1.8.25: resolved Issue 1829: **AddLidar** (fix)

- description: several arguments are not passed to CreateDistanceSensor; add all arguments to interface, which may cause some change in behavior!
- date resolved: **2024-04-27 16:48**, date raised: 2024-04-27

Version 1.8.24: resolved Issue 1828: **Examples** (example)

- description: add example mobileMecanumWheelRobotWithLidar.py for a mecanum wheeled robot with lidar and mapping
- date resolved: **2024-04-27 11:11**, date raised: 2024-04-27 (resolved by: P. Manzl)

Version 1.8.23: resolved Issue 1827: **AddLidar** (fix)

- description: argument rotation is not used

- date resolved: **2024-04-27 11:08**, date raised: 2024-04-27

Version 1.8.22: resolved Issue 1826: **AddLidar** (change)

- description: angles of sensor directions are not defined and erronously with respect to Y-axis; angleStart and angleEnde shall be measured w.r.t. the X-axis (angle=0) and in positive rotation sense about local Z-axis
- date resolved: **2024-04-27 11:08**, date raised: 2024-04-27

Version 1.8.21: resolved Issue 1824: **ObjectConnectorCoordinateVector** (docu)

- description: fix docu and remove inexisting parameters
- date resolved: **2024-04-21 19:34**, date raised: 2024-04-21

Version 1.8.20: resolved Issue 1820: **item type info** (change)

- description: change py::dict to py:object in functions such as AddMarker according to github issue #64, otherwise giving typing errors in PyLance and similar
- date resolved: **2024-04-21 19:17**, date raised: 2024-04-19

Version 1.8.19: resolved Issue 1823: **stub files** (change)

- description: change type in AddObject, AddMarker, AddSensor, AddLoad, AddNode from pyObject: dict to pyObject: Any in order to resolve typing errors
- date resolved: **2024-04-21 19:16**, date raised: 2024-04-19

Version 1.8.18: resolved Issue 1819: **KinematicTreePendulum.py** (example)

- description: Version 1.7.71 broke example kinematicTreePendulum with transition from SensorObject to SensorBody
- **notes: changed SensorObject(objectNumber=oKT into SensorBody(bodyNumber=oKT**
- date resolved: **2024-04-17 16:13**, date raised: 2024-04-17 (resolved by: P. Manzl)

Version 1.8.17: resolved Issue 1818: **stub files** (fix)

- description: add types for mainsystem extensions (e.g. SolutionViewer)
- **notes: not resolved, because mainSystemExtension functions do not contain argument types**
- date resolved: **2024-04-14 17:54**, date raised: 2024-04-14

Version 1.8.16: resolved Issue 1817: **stub files** (fix)

- description: add types for system structures (mainly solver functions)
- **notes: not resolved, because this information is not available in the structures for solver methods**
- date resolved: **2024-04-14 17:53**, date raised: 2024-04-14

Version 1.8.15: resolved Issue 1816: **stub files** (fix)

- description: fix a couple of wrong return types and similar wrong typings (in particular return types in SetODE2Coordinates, etc.
- date resolved: **2024-04-14 16:15**, date raised: 2024-04-14

Version 1.8.14: resolved Issue 1814: **stub files** (fix)

- description: .pyi files do not contain correct default args for C++ interfaces
- date resolved: **2024-04-14 16:15**, date raised: 2024-04-14

Version 1.8.13: resolved Issue 1815: **stub files** (fix)

- description: .pyi are missing backslash before underscore (for latex) documentation
- date resolved: **2024-04-14 15:15**, date raised: 2024-04-14

Version 1.8.12: resolved Issue 1812: **GetMarkerOutput** (fix)

- description: not working for MarkerKinematicTreeRigid in case of Reference configuration; adjustments done in CObjectKinematicTree::ComputeTreeTransformations to avoid the need for velocities in the retrieval of positions
- date resolved: **2024-04-03 19:38**, date raised: 2024-04-03

Version 1.8.11: resolved Issue 1811: **Sensitivity Analysis for functions with single outputs** (fix)

- issue author: PM
- description: ComputeSensitivities and PlotSensitivityResults does not work for functions with a single output
- date resolved: **2024-03-26 19:24**, date raised: 2024-03-26 (resolved by: P. Manzl)

Version 1.8.10: resolved Issue 1810: **GeneralContact** (change)

- description: unify computation of contact forces and jacobians for sphere-sphere and trig-sphere contact
- date resolved: **2024-03-25 08:21**, date raised: 2024-03-25

Version 1.8.9: resolved Issue 1809: **GeneralContact** (fix)

- description: apply changes in sphere-sphere contact to Jacobian and to docu
- date resolved: **2024-03-17 19:05**, date raised: 2024-03-17

Version 1.8.8: **resolved BUG 1807**: **GeneralContact**

- description: sphere-shpere contact causes spurious internal torque
- **notes: fixed by adding 0.5*penetration in lever arm; adjusted also documentation; gives new test suite results**
- date resolved: **2024-03-17 10:56**, date raised: 2024-03-17

Version 1.8.7: **resolved BUG 1805**: **CreateRigidBody**

- description: raises exception in case that initialRotationMatrix is not None; SOLUTION: replace == None for ALL cases (check other functions) with is None or is not Note!
- **notes: fixed with 1808**
- date resolved: **2024-03-17 10:55**, date raised: 2024-03-16

Version 1.8.6: resolved BUG 1806: **CreateRigidBody**

- description: initialRotationMatrix has no effect at least in explicit integration
- **notes: wrong operator* used for multiplication of reference rotation matrix and initial rotation matrix; FIXED**
- date resolved: **2024-03-17 10:54**, date raised: 2024-03-16

Version 1.8.5: resolved Issue 1808: **advanced utilities** (extension)

- description: added function to check for None and not None: IsNone(x), IsNotNone(x)
- date resolved: **2024-03-17 10:29**, date raised: 2024-03-17

Version 1.8.4: resolved Issue 1804: **GeneralContact** (check)

- description: check triangle-sphere contact: torque for triangle computed with "((-sphereI.radius)*deltaP0).CrossProduct(fVec)" lever arm should be -(trigPP - rigid.position) ?
- **notes: changed term for torque on rigid body according to added documentation in theory section**
- date resolved: **2024-03-17 10:15**, date raised: 2024-03-16

Version 1.8.3: resolved Issue 1802: **RigidBodySpringDamper** (check)

- description: check intrinsic joint formulation
- **notes: no inconsistencies found or detected in examples**
- date resolved: **2024-03-13 13:10**, date raised: 2024-03-07

Version 1.8.2: resolved Issue 1803: **MarkerSuperElementRigid** (extension)

- description: add option for tangent operator in alternativeFormulation
- date resolved: **2024-03-13 13:09**, date raised: 2024-03-13

Version 1.8.1: resolved Issue 0734: **continuous integration** (coding)

- description: test CI capabilities with GitHub and MacOS compilation
- **notes: resolved with issue 1792**
- date resolved: **2024-03-09 16:04**, date raised: 2021-08-12

Version 1.8.0: resolved Issue 1789: **AvailableItems** (extension)

- description: add exudyn.special function to retrieve available items as dictionary with lists
- date resolved: **2024-03-06 09:02**, date raised: 2024-02-21

Version 1.7.123: resolved Issue 1795: **joint constraints** (docu)

- description: theory: add description for formulation of joint constraints
- **notes: added equations to position markers and JointSpherical**
- date resolved: **2024-03-03 22:04**, date raised: 2024-02-25

Version 1.7.122: resolved Issue 1800: **GeneralContact** (extension)

- description: add option for GetActiveContacts to return number of contacts per contact type in case that itemIndex=-1
- date resolved: **2024-02-29 15:50**, date raised: 2024-02-29

Version 1.7.121: resolved Issue 1799: **exudyn __init__.py** (change)

- description: remove NoAVX option for linux, as linux does not (yet) have a AVX2 option; crashed on linux arm/aarch architecture
- date resolved: **2024-02-29 14:34**, date raised: 2024-02-29

Version 1.7.120: resolved Issue 1797: **Github actions** (extension)

- description: create single line output for testsuite with specific mode; add test suite for github actions and merge outputs into single file
- **notes: put information into filename of text tilde with output of testsuite**
- date resolved: **2024-02-29 14:32**, date raised: 2024-02-27

Version 1.7.119: resolved Issue 1798: **linux arm** (extension)

- description: add multilinux aarch64 wheels to GH build actions
- date resolved: **2024-02-29 14:31**, date raised: 2024-02-29

Version 1.7.118: resolved Issue 1796: **MacOSX universal2** (extension)

- description: add build option for macos universal files on GH actions to have both arm and x86 on board
- **notes: NOTE that pip 20.3 is required to install these wheels!**
- date resolved: **2024-02-27 15:06**, date raised: 2024-02-27

Version 1.7.117: resolved Issue 1794: **fix curly brackets** (docu)

- description: fix curly brackets in RST files
- date resolved: **2024-02-25 20:43**, date raised: 2024-02-25

Version 1.7.116: resolved Issue 1793: **manylinux2014** (extension)

- description: build highly compatible manylinux2014 and manylinux2_17 wheels with github actions docker, to run on CentOS and Rocky Linux as well as ubuntu
- date resolved: **2024-02-24 23:28**, date raised: 2024-02-24

Version 1.7.115: resolved Issue 1792: **Github actions CI** (extension)

- description: add github actions to create automatically Windows, Ubunut and MacOS wheels
- date resolved: **2024-02-24 23:28**, date raised: 2024-02-24

Version 1.7.114: resolved Issue 1787: **Python 3.12** (extension)

- description: include Python 3.12 wheels into build process
- date resolved: **2024-02-24 23:25**, date raised: 2024-02-21

Version 1.7.113: **resolved BUG 1791**: **Autoregistration items**

- description: node does not initialize CData
- date resolved: **2024-02-24 17:45**, date raised: 2024-02-24

Version 1.7.112: resolved Issue 1788: **items auto-registration** (change)

- description: add a simple way to automatically register items; use C++ map to create item in object-factory
- date resolved: **2024-02-21 19:17**, date raised: 2024-02-21

Version 1.7.111: resolved Issue 1790: **exudyn minimal** (extension)

- description: add flag EXUDYN_MINIMAL_ITEMS to achieve fast compilation for testing
- date resolved: **2024-02-21 18:53**, date raised: 2024-02-21

Version 1.7.110: **resolved BUG 1786**: **ComputeODE2singleLoad**

- description: raises error in static computation "inconsistent jacobian"; workaround settings computeLoadsJacobian=False or using sparse solver
- **notes: exception due to inconsistent computation of mass proportional load jacobian with rigid body**
- date resolved: **2024-02-16 11:42**, date raised: 2024-02-16

Version 1.7.109: resolved Issue 1785: **beam tutorial** (docu)

- description: add beam tutorial example and tutorial in theDoc
- date resolved: **2024-02-13 22:08**, date raised: 2024-02-13

Version 1.7.108: resolved Issue 1784: **theory theDoc** (docu)

- description: add introduction to multibody dynamics, kinematics, dynamics, rotations
- date resolved: **2024-02-13 16:10**, date raised: 2024-02-13

Version 1.7.107: resolved Issue 1783: **RST graphics** (docu)

- description: add graphics for readthedocs representation, in solvers; fix references
- date resolved: **2024-02-13 16:10**, date raised: 2024-02-13

Version 1.7.106: resolved Issue 1782: **GeneralContact** (extension)

- description: add flag computeContactForces to settings, which computes contribution of contact forces to system vector (may slow down computations!); similar to issue 936
- date resolved: **2024-02-12 09:07**, date raised: 2024-02-12

Version 1.7.105: resolved Issue 0936: **GeneralContact** (extension)

- description: add interface function to get contact forces
- date resolved: **2024-02-12 09:07**, date raised: 2022-02-10

Version 1.7.104: resolved Issue 1781: **GeneralContact** (extension)

- description: add function Get/SetTriangleRigidBodyBased, to get data or modify data of current contact triangle
- date resolved: **2024-02-08 20:38**, date raised: 2024-02-08

Version 1.7.103: resolved Issue 1780: **GeneralContact** (extension)

- description: add function SetSphereMarkerBased to set data for spheres during simulation
- date resolved: **2024-02-08 20:38**, date raised: 2024-02-08

Version 1.7.102: resolved Issue 1779: **GeneralContact** (change)

- description: GetMarkerBasedSphere: change to GetSphereMarkerBased; add flag to decide whether to add basic data or not
- date resolved: **2024-02-08 19:08**, date raised: 2024-02-08

Version 1.7.101: resolved Issue 1778: **cRGB settings** (change)

- description: change cRGB consistently to RGBA in visualization settings
- date resolved: **2024-02-08 18:57**, date raised: 2024-02-08

Version 1.7.100: resolved Issue 1775: **BodyGraphicsData** (fix)

- description: access with Get/SetObjectParameter is missing for graphicsData
- date resolved: **2024-02-04 22:01**, date raised: 2024-02-04

Version 1.7.99: resolved Issue 1774: **CreateSymbolicUserFunction** (change)

- description: change order of args userFunctionName, itemIndex, and verbose; add additional itemTypeName

- date resolved: **2024-02-04 00:36**, date raised: 2024-02-04

Version 1.7.98: resolved Issue 1773: **CreateSymbolicUserFunction** (extension)

- description: add option to directly pass itemTypeName instead of itemIndex in order to pre-compute user function
- date resolved: **2024-02-04 00:36**, date raised: 2024-02-04

Version 1.7.97: resolved Issue 1771: **TransferUserFunction2Item** (change)

- description: add functionality to allow direct assignment of symbolic user functions to userFunction parameters in objects, loads, etc.; remove TransferUserFunction2Item as this function is then no longer needed
- date resolved: **2024-02-03 23:55**, date raised: 2024-02-03

Version 1.7.96: resolved Issue 1766: **Python user functions** (extension)

- description: use PythonUserFunctionBase class for all Item user functions; add consistent Get/Set function for item access; should then automatically work with pickle
- date resolved: **2024-02-03 23:54**, date raised: 2024-02-02

Version 1.7.95: resolved Issue 1750: **python user functions** (extension)

- description: add additional py::function to user functions in order to store original python function for pickling
- date resolved: **2024-02-03 23:54**, date raised: 2024-01-29

Version 1.7.94: resolved Issue 1759: **Renderer** (fix)

- description: there is an issue when restarting the renderer, which displays previous (old) data; requires to add some function which erases stored graphics data on call of StartRenderer()
- date resolved: **2024-02-03 23:53**, date raised: 2024-01-31

Version 1.7.93: resolved Issue 1770: **mainsystem extensions** (extensions)

- description: add user function to mbs.Create...() functions
- date resolved: **2024-02-03 22:50**, date raised: 2024-02-03

Version 1.7.92: resolved Issue 1763: **Python user functions** (extension)

- description: add pickle functionality (requires issue 1752)
- date resolved: **2024-02-02 10:36**, date raised: 2024-01-31

Version 1.7.91: resolved Issue 1752: **user functions types** (extension)

- description: create UserFunctionBase class and derived classes, containing std::function, and a py::object with metadata (Python function, Symbolic function, etc.); When settings user functions, they can be either initialized with 0 / Python function or with a UserFunction dict, which contains additional decorators; In particular, user functions will be rebuilt as symbolic; return values of user functions are then dictionaries
- date resolved: **2024-02-02 10:36**, date raised: 2024-01-29

Version 1.7.90: resolved Issue 1765: **MainSystem user functions** (extension)

- description: test special class for MainSystem user functions such as preStepUserFunction; if requeste, convert to Dict, in particular for symbolic or other special user functions
- date resolved: **2024-02-02 10:33**, date raised: 2024-02-02

Version 1.7.89: resolved Issue 1762: **SystemContainer** (extension)

- description: add pickle functionality
- date resolved: **2024-01-31 20:22**, date raised: 2024-01-31

Version 1.7.88: resolved Issue 1628: **pickle MainSystem** (extension)

- description: consider a pickle method for certain objects; add consistent info in description; MainSystem, SimulationSettings, VisualizationSettings
- **notes: consider with care, as not all things are copied (user functions, contact, ...)**
- date resolved: **2024-01-31 20:22**, date raised: 2023-06-23

Version 1.7.87: resolved Issue 1761: **pickle** (extension)

- description: add pickle to settings and structures
- date resolved: **2024-01-31 20:21**, date raised: 2024-01-31

Version 1.7.86: resolved Issue 1760: **pickle** (extension)

- description: add pickle to ItemIndices
- date resolved: **2024-01-31 20:21**, date raised: 2024-01-31

Version 1.7.85: resolved Issue 1757: **DictionariesGetSet** (extension)

- description: add C++ GetDictionary(...) function for read/write of system structures; add Get/SetDictionary to pybind interface
- date resolved: **2024-01-31 17:27**, date raised: 2024-01-30

Version 1.7.84: resolved Issue 1758: **StartRenderer** (fix)

- description: add UpdateGraphicsDataNow() after start of renderer in order to avoid showing stored data for second run or renderer
- date resolved: **2024-01-31 12:52**, date raised: 2024-01-31

Version 1.7.83: resolved Issue 1755: **CSystem in MainSystem** (change)

- description: change CSystem* to CSystem in MainSystem, to make copying easier
- date resolved: **2024-01-30 16:43**, date raised: 2024-01-30

Version 1.7.82: resolved Issue 1756: **SystemContainer** (change)

- description: remove SystemContainer and just keep MainSystemContainer, as it is not needed; simplifies copying
- date resolved: **2024-01-30 16:42**, date raised: 2024-01-30

Version 1.7.81: resolved Issue 1754: **MainSystem** (extension)

- description: change creation of MainSystem; allow construction like exudyn.MainSystem(), add new function Append(MainSystem) to MainSystemContainer; this will allow pickling both of MainSystem and MainSystemContainer
- date resolved: **2024-01-30 14:34**, date raised: 2024-01-30

Version 1.7.80: resolved Issue 1753: **postStepUserFunction** (extension)

- description: add user function to be called at end of time step, just before storing results to file; this allows to override results, etc.
- date resolved: **2024-01-30 08:01**, date raised: 2024-01-30

Version 1.7.79: resolved Issue 1748: **user functions** (fix)

- description: check option to set them to 0; PreStepUserFunction as well as item user functions
- **notes: Note that when reading user functions, mbs.GetObjectParameter(...) also consistently gives now 0 instead of previously None**
- date resolved: **2024-01-29 14:41**, date raised: 2024-01-29

Version 1.7.78: resolved Issue 1749: **User function** (extension)

- description: allow assignment to 0 for MainSystem user functions; SetPreStepUserFunction and SetPostNewtonUserFunction
- date resolved: **2024-01-29 14:34**, date raised: 2024-01-29

Version 1.7.77: resolved Issue 1746: **GenerateStraightBeam** (extension)

- description: build generic function to create beams along straight line; add interface both for ANCF (old GenerateStraightLineANCFCable function) as well as for geometrically exact beam
- date resolved: **2024-01-28 19:46**, date raised: 2024-01-28

Version 1.7.76: resolved Issue 1747: **GeometricallyExactBeam2D** (extension)

- description: finish interface for load mass proportional
- **notes: also done for 3D version**
- date resolved: **2024-01-28 18:17**, date raised: 2024-01-28

Version 1.7.75: resolved Issue 1745: **geometrically exact beam 2D** (change)

- description: adjust implementation of reference strains to allow connection of several beams at one node in case that includeReferenceRotations=0
- date resolved: **2024-01-27 22:16**, date raised: 2024-01-27

Version 1.7.74: resolved Issue 1744: **geometrically exact beam 2D** (docu)

- description: fix inconsistent documentation of reference strains
- date resolved: **2024-01-27 22:16**, date raised: 2024-01-27

Version 1.7.73: resolved Issue 1743: **CreateRevoluteJoint** (fix)

- description: Description for local/global axis is wrong; behavior is switched by useGlobalFrame flag
- date resolved: **2024-01-06 11:20**, date raised: 2024-01-06

Version 1.7.72: resolved Issue 1742: **CreatePrismaticJoint** (fix)

- description: Description for local/global axis is wrong; behavior is switched by useGlobalFrame flag
- date resolved: **2024-01-06 11:20**, date raised: 2024-01-06

Version 1.7.71: **resolved BUG 1741**: **ObjectKinematicTree**

- description: SensorObject does not work with OutputVariableType Coordinates; example does not work;
- **notes: Coordinates, Force, etc. now available with GetObjectOutputBody and SensorBody**
- date resolved: **2024-01-06 11:02**, date raised: 2024-01-06

Version 1.7.70: resolved Issue 1739: **symbolic** (fix)

- description: SetValue should raise exception if called with symbolic expression
- date resolved: **2023-12-19 08:30**, date raised: 2023-12-19

Version 1.7.69: resolved Issue 1737: **symbolic** (extension)

- description: add symbolic.pyi stub file for autocompletion of symbolic features
- date resolved: **2023-12-18 19:48**, date raised: 2023-12-18

Version 1.7.68: resolved Issue 1738: **symbolic** (docu)

- description: fix documentation for operators
- date resolved: **2023-12-18 19:47**, date raised: 2023-12-18

Version 1.7.67: resolved Issue 1727: **SymbolicRealMatrix** (docu)

- description: add documentation and example for symbolic matrix for user functions

- date resolved: **2023-12-18 08:24**, date raised: 2023-12-12

Version 1.7.66: **resolved BUG 1736**: **symbolic**

- description: symbolic user function: crashes when user function object is deleted
- date resolved: **2023-12-15 18:01**, date raised: 2023-12-15

Version 1.7.65: resolved Issue 1735: **symbolic** (fix)

- description: check delete counts and reference counts for +=, etc.
- date resolved: **2023-12-15 18:01**, date raised: 2023-12-15

Version 1.7.64: resolved Issue 1729: **Symbolic** (testing)

- description: add vector/matrix tests in comparison with Python numpy and check delete counts
- date resolved: **2023-12-15 13:52**, date raised: 2023-12-12

Version 1.7.63: resolved Issue 1728: **Symbolic** (testing)

- description: add scalar tests in comparison with Python math and check delete counts
- date resolved: **2023-12-15 13:52**, date raised: 2023-12-12

Version 1.7.62: resolved Issue 1733: **symbolic** (check)

- description: check overloading __len__ operator for vector
- date resolved: **2023-12-15 13:06**, date raised: 2023-12-15

Version 1.7.61: resolved Issue 1734: **symbolic** (fix)

- description: write operator[] for Matrix and Vector fails
- date resolved: **2023-12-15 11:51**, date raised: 2023-12-15

Version 1.7.60: **resolved BUG 1732**: **symbolic**

- description: Vector.SetVector(...), Matrix.SetMatrix(...) not working; fix Pybind interface
- date resolved: **2023-12-15 11:12**, date raised: 2023-12-15

Version 1.7.59: resolved Issue 1680: **chatGPTupdate** (example)

- description: add simple example for load userFunction
- date resolved: **2023-12-14 00:01**, date raised: 2023-10-29

Version 1.7.58: resolved Issue 1693: **VObjectGround** (fix)

- description: remove parameter color, as it is not used (check)
- date resolved: **2023-12-13 23:40**, date raised: 2023-11-19

Version 1.7.57: resolved Issue 1726: **SymbolicRealMatrix** (extension)

- description: add symbolic matrix for user functions
- **notes: note: currently implemented less efficient with memory allocations**
- date resolved: **2023-12-13 14:02**, date raised: 2023-12-12

Version 1.7.56: resolved Issue 1731: **ANCFCable2D** (extension)

- description: add user functions for bending moment and axial force, allowing to implement arbitrary material models
- date resolved: **2023-12-13 13:47**, date raised: 2023-12-13

Version 1.7.55: resolved Issue 1730: **GenerateStraightLineANCFCable** (fix)

- description: raises Warning for default values [0,0,0,0,0,0] in 2D case
- date resolved: **2023-12-13 13:27**, date raised: 2023-12-13

Version 1.7.54: resolved Issue 1725: **Symbolic** (extension)

- description: add ResizableConstMatrix and create symbolic matrix-vector functions
- date resolved: **2023-12-12 14:16**, date raised: 2023-12-10

Version 1.7.53: resolved Issue 1716: **Symbolic** (docu)

- description: add symbolic user function description to documentation; also mention in performance section
- date resolved: **2023-12-10 21:57**, date raised: 2023-12-08

Version 1.7.52: resolved Issue 1724: **C++ ToString** (change)

- description: changing the behavior for standard conversion of double and int values to strings, in particular during errors. Now using the same precision as defined with exudyn.SetOutputPrecision()
- date resolved: **2023-12-09 19:26**, date raised: 2023-12-09

Version 1.7.51: resolved Issue 1715: **Symbolic** (docu)

- description: add symbolic section to documentation
- date resolved: **2023-12-09 16:29**, date raised: 2023-12-08

Version 1.7.50: resolved Issue 1708: **Symbolic** (extension)

- description: make symbolic variable space, available globally in exudyn.symbolic as well as in mbs.symbolic; this allows to store/transfer data into user functions without the need for Python; use integer handles which are returned by creation function: a=NamedReal(value, name); handle=AddVariableReal(a)

- **notes: not yet put into mbs and using no integer handles, but std::unordered_map, which has highly efficient hash table included**
- date resolved: **2023-12-09 16:29**, date raised: 2023-12-03

Version 1.7.49: resolved Issue 1699: **exudyn. ...** (docu)

- description: add undocumented features of exudyn module, such as Demo1(), Demo2(), __version__ or C++
- date resolved: **2023-12-08 18:41**, date raised: 2023-11-22

Version 1.7.48: resolved Issue 1698: **experimental, special** (docu)

- description: add experimental and special features to documentation / pybindings
- date resolved: **2023-12-08 18:41**, date raised: 2023-11-21

Version 1.7.47: resolved Issue 1710: **Symbolic** (extension)

- description: add basic (automatic) differentiation feature for expressions: EvaluateDiff()
- date resolved: **2023-12-08 07:22**, date raised: 2023-12-03

Version 1.7.46: resolved Issue 1700: **ResizableConstSizeVector** (extension)

- description: consider a Vector with fixed size, which can be extended if necessary - e.g. for local variables in sensors or GetOutputVariable; is efficient for small vectors and still works for larger one
- date resolved: **2023-12-07 23:03**, date raised: 2023-11-22

Version 1.7.45: resolved Issue 1714: **Symbolic** (extension)

- description: add vector as symbolic expression, allowing vectors in user functions
- date resolved: **2023-12-07 19:55**, date raised: 2023-12-07

Version 1.7.44: resolved Issue 1713: **user functions** (change)

- description: change StdVector to StdVector3D and StdVector6D in relevant cases in order to achieve light-weight interface for symbolic interfaces
- date resolved: **2023-12-07 19:53**, date raised: 2023-12-07

Version 1.7.43: resolved Issue 1712: **Symbolic** (extension)

- description: add automatic creation of user functions to AutoGenerateObjects
- date resolved: **2023-12-05 00:06**, date raised: 2023-12-03

Version 1.7.42: resolved Issue 1711: **Symbolic** (extension)

- description: put most parts of specific user functioninto base SymbolicFunction; EvaluateUF: use variadic args to generalize
- date resolved: **2023-12-05 00:06**, date raised: 2023-12-03

Version 1.7.41: resolved Issue 1709: **Symbolic** (extension)

- description: add most of Pythons math module functions to symbolic functions list
- date resolved: **2023-12-03 15:01**, date raised: 2023-12-03

Version 1.7.40: resolved Issue 1705: **Symbolic user function** (extension)

- description: allow parallel computation of non-Python user functions
- **notes: this is enabled by adding user function after Assemble, thus objects are not registered to have Python user functions**
- date resolved: **2023-12-03 14:58**, date raised: 2023-11-28

Version 1.7.39: resolved Issue 1706: **RigidBodySpringDamper** (docu)

- description: intrinsicFormulation: add test to test suite and document new functionality
- date resolved: **2023-11-30 23:56**, date raised: 2023-11-30

Version 1.7.38: resolved Issue 1707: **RigidBodySpringDamper** (check)

- description: intrinsicFormulation: check conserving properties of joint forces for two freely rotating bodies: additional torque of forces may be required in implementation
- date resolved: **2023-11-30 23:39**, date raised: 2023-11-30

Version 1.7.37: resolved Issue 1486: **RigidBodySpringDamper** (extension)

- description: extend for Lie group formulation, evaluating connectors at mid-configuration according to Masarati and Morandini
- date resolved: **2023-11-30 20:51**, date raised: 2023-04-01

Version 1.7.36: resolved Issue 1696: **exudyn.symbolic** (extension)

- description: add experimental expression trees for building symbolic expressions to be used for user functions
- **notes: basic symbolic functionality added; tested with SpringDamper user function, leading to speedup of 10 against regular Python function**
- date resolved: **2023-11-28 09:08**, date raised: 2023-11-21 (resolved by: EXTENSION)

Version 1.7.35: resolved Issue 1704: **solver timeout** (extension)

- description: added module-wide flag for timeout: exudyn.special.solver.timeout in order to stop simulations after certain time; use with care
- date resolved: **2023-11-22 09:43**, date raised: 2023-11-22

Version 1.7.34: resolved Issue 1703: **Pybind module** (change)

- description: use suggestions of from search: pybind11, how to split my code into multiple modules/files - stackoverflow; should improve compilation time
- **notes: created 2 new pybind module files; no major compilation speedup visible on laptop**
- date resolved: **2023-11-22 08:24**, date raised: 2023-11-22

Version 1.7.33: resolved Issue 1702: **PyErr_CheckSignals** (check)

- description: check if this method available in pybind helps to allow stopping long-lasting computations in exudyn
- **notes: still does not work in Spyder**
- date resolved: **2023-11-22 02:03**, date raised: 2023-11-22

Version 1.7.32: resolved Issue 1701: **RunCppUnitTests** (change)

- description: move to exudyn.special.RunCppUnitTests
- date resolved: **2023-11-22 00:29**, date raised: 2023-11-22

Version 1.7.31: resolved Issue 1697: **exudyn.experimental** (change)

- description: change exudyn.Experimental() into exudyn.experimental structure for clearer view on it; not intended to be used widely
- **notes: see also issue 1613**
- date resolved: **2023-11-21 21:18**, date raised: 2023-11-21

Version 1.7.30: resolved Issue 1694: **load user functions** (change)

- description: add stl and numpy bindings in order to have Vector3D converted into numpy arrays in loadVectorUserFunction
- date resolved: **2023-11-19 23:05**, date raised: 2023-11-19

Version 1.7.29: resolved Issue 1690: **mainSystemExtensions** (extension)

- description: add CreateGround() with referencePosition, referenceRotationMatrix and visualization
- date resolved: **2023-11-19 23:05**, date raised: 2023-11-19

Version 1.7.28: resolved Issue 1679: **CreateTorque** (extension)

- description: add create function for nodes and bodies (using node=None, body=None default) to be used either for nodes or bodies; automatically adds markers; bodyFixed=False; add option to add userFunction
- date resolved: **2023-11-19 22:22**, date raised: 2023-10-29

Version 1.7.27: resolved Issue 1678: **CreateForce** (extension)

- description: add create function for nodes and bodies (using node=None, body=None default) to be used either for nodes or bodies; automatically adds markers; bodyFixed=False; add option to add userFunction
- date resolved: **2023-11-19 22:22**, date raised: 2023-10-29

Version 1.7.26: resolved Issue 1689: **mainSystemExtensions** (extension)

- description: change bodyOrNodeList into bodyList; allow bodyOrNodeList as alternative arg, but avoid using in examples
- date resolved: **2023-11-19 21:04**, date raised: 2023-11-19

Version 1.7.25: resolved Issue 1688: **mainSystemExtensions** (extension)

- description: allow special case distance=0 in CreateDistanceConstraint; this will then create a SphericalJoint
- date resolved: **2023-11-19 21:04**, date raised: 2023-11-19

Version 1.7.24: resolved Issue 1687: **mainSystemExtensions** (extension)

- description: allow referenceLength=0 in ConnectorSpringDamper
- date resolved: **2023-11-19 21:04**, date raised: 2023-11-19

Version 1.7.23: resolved Issue 1667: **ANCFCable** (extension)

- description: add visulization with cylinders
- date resolved: **2023-11-19 19:39**, date raised: 2023-10-16

Version 1.7.22: resolved Issue 1686: **SpringDamper** (extension)

- description: allow springLength=0, as it does not cause problems in computations
- **notes: added special behavior for L=0, but velocity not equal 0; may cause convergence issues in particular for static problems; added special behavior for L=0, but velocity not equal 0; may cause convergence issues in particular for static problems; also allow referenceLength=0**
- date resolved: **2023-11-19 19:11**, date raised: 2023-11-19

Version 1.7.21: resolved Issue 1685: **ANCF cable with rigid marker** (example)

- description: add example with ANCFCable2D and rigid body marker, prescribing rotation of one end
- **notes: ANCFrotatingCable2D.py**
- date resolved: **2023-11-07 14:04**, date raised: 2023-11-07

Version 1.7.20: resolved Issue 1673: **ReadTheDocs** (fix)

- description: add the required .readthedocs.yaml file and move requirements.txt into docs folder, as it is related to sphinx only
- date resolved: **2023-10-25 09:16**, date raised: 2023-10-25

Version 1.7.19: resolved Issue 1672: **LieGroup explicit integration** (change)

- description: fixed lieGroupDataNodes; lieGroupDataNodes renamed into lieGroupNodes in explict integration
- date resolved: **2023-10-16 10:00**, date raised: 2023-10-16

Version 1.7.18: resolved Issue 1671: **PlotSensor** (change)

- description: use IsListOrArray function to check for non-empty offsets and factors, to comply with numpy arrays
- date resolved: **2023-10-16 08:32**, date raised: 2023-10-16

Version 1.7.17: resolved Issue 1670: **GenerateStraightLineANCFCable** (extension)

- description: add GenerateStraightLineANCFCable for 3D cables and adjust 2D version
- date resolved: **2023-10-16 08:31**, date raised: 2023-10-16

Version 1.7.16: resolved Issue 1669: **NodePoint3DSlope** (fix)

- description: adjust all test models and examples for new NodePointSlope... names
- date resolved: **2023-10-16 08:02**, date raised: 2023-10-16

Version 1.7.15: resolved Issue 1663: **NodePoint3DSlope23** (check)

- description: check d/dt(A) ... computation of time derivative of rotation matrix, also check jacobians and angular velocities; add tests also for Slope12 node
- date resolved: **2023-10-16 07:45**, date raised: 2023-10-15

Version 1.7.14: resolved Issue 1666: **NodePoint3DSlope1** (change)

- description: change to NodePointSlope1
- date resolved: **2023-10-15 22:29**, date raised: 2023-10-15

Version 1.7.13: resolved Issue 1665: **Point3DS23** (change)

- description: remove Point3DS23 as its name is inconsistent with 3D convention and not really needed
- date resolved: **2023-10-15 22:24**, date raised: 2023-10-15

Version 1.7.12: resolved Issue 1664: **NodePoint3DSlope23** (change)

- description: change to NodePointSlope23
- date resolved: **2023-10-15 22:23**, date raised: 2023-10-15

Version 1.7.11: resolved Issue 1075: **ANCFCable** (extension)

- description: add 3D version of cable element, not using BeamSection interface for compatibility with Cable2D
- date resolved: **2023-10-15 14:45**, date raised: 2022-05-06

Version 1.7.10: resolved Issue 1661: **NodePoint3DSlope12** (extension)

- description: add ANCF node with slopes x/y for thin plate element
- date resolved: **2023-10-15 14:21**, date raised: 2023-10-15

Version 1.7.9: resolved Issue 1403: **Lie group nodes** (change)

- description: remove LieGroup node RigidBodyRotVecDataLG as it is not needed any more as it can be substituted with more efficient RigidBodyRotVecLG
- **notes: right now added comments in C++ files, not completely removed**
- date resolved: **2023-10-15 11:57**, date raised: 2023-01-18

Version 1.7.8: resolved Issue 1659: **remove math.h** (change)

- description: change with <cmath> for C++ conformity
- date resolved: **2023-10-12 17:04**, date raised: 2023-10-12

Version 1.7.7: resolved Issue 1658: **switch to MSVC2022** (change)

- description: change main_sln_Template.sln for 2022 update; use cl.exe from MSVC2022 for compilation of wheels; slightly increases performance
- **notes: compilation successful and TestSuite runs through**
- date resolved: **2023-10-12 17:04**, date raised: 2023-10-12

Version 1.7.6: resolved Issue 1660: **plot** (change)

- description: change plt.tight_layout and plt.legend to fig. if possible to avoid warnings
- date resolved: **2023-10-12 13:39**, date raised: 2023-10-12

Version 1.7.5: resolved Issue 1657: **Docu** (fix)

- description: latex errors in robotics mobile and ROS
- date resolved: **2023-10-09 20:27**, date raised: 2023-10-09

Version 1.7.4: resolved Issue 1656: **ROS rosInterface** (extension)

- description: Created robotics.rosInterface and Python models in Examples: ROSMassPoint.py, ROSMobileManipulator.py, ROSTurtle.py with supplementary, see Examples/supplementary: ROSControlMobileManipulation.py, ROSControlTurtleVelocity.py, etc.
- date resolved: **2023-09-15 15:50**, date raised: 2023-09-15 (resolved by: Martin Sereinig)

Version 1.7.3: resolved Issue 1655: **SolveStatic** (extension)

- description: add option for static solver to handle ODE1 quantities; currently, the option is to set ODE1coordinates to initial values during static computation
- date resolved: **2023-09-07 21:13**, date raised: 2023-09-07

Version 1.7.2: resolved Issue 1654: **Python3.6 support** (change)

- description: discontinuing testing and creation of pip installers for Python3.6 in Windows, Linux and MacOS as Python3.6 had end-of-life 2021-12-23; Python 3.7 also had end-of-life recently, so please expect discontinued support soon
- date resolved: **2023-09-03 16:03**, date raised: 2023-09-03

Version 1.7.1: resolved Issue 1652: **rosInterface.py** (fix)

- description: add (missing) file to DOCU
- date resolved: **2023-08-08 17:53**, date raised: 2023-08-08

Version 1.7.0: resolved Issue 1649: **release** (release)

- description: switch to new release 1.7
- date resolved: **2023-07-19 16:07**, date raised: 2023-07-19

Version 1.6.189: resolved Issue 1648: **ReevingSystemSprings** (fix)

- description: adjust test example for treating compression forces
- date resolved: **2023-07-17 12:10**, date raised: 2023-07-17

Version 1.6.188: resolved Issue 1647: **sensorTraces** (extension)

- description: add vectors and triads to position traces; show current vector or triad and add some further options for visualization, see visualizationSettings sensors.traces
- **notes: added triads and vectors, showing traces of motion at sensor points**
- date resolved: **2023-07-14 18:34**, date raised: 2023-07-14

Version 1.6.187: resolved Issue 1640: **visualization** (extension)

- description: show trace of sensor positions (incl. frames) in render window; settings and list of sensors provided in visualizationSettings.sensors.traces with list of sensors, positionTrace, listOfPositionSensors=[] (empty means all position sensors, listOfVectorSensors=[] which can provide according vector quantities for positions; showVectors, vectorScaling=0.001, showPast=True, showFuture=False, showCurrent=True, lineWidth=2
- date resolved: **2023-07-14 11:20**, date raised: 2023-07-11

Version 1.6.186: resolved Issue 1646: **ArrayFloat** (extension)

- description: C++ add type
- date resolved: **2023-07-13 16:27**, date raised: 2023-07-13

Version 1.6.185: resolved Issue 1645: **ReevingSystemSprings** (extension)

- description: add way to remove compression forces in rope
- **notes: added parameter regularizationForce with tanh regularization for avoidance of compressive spring force**
- date resolved: **2023-07-12 16:05**, date raised: 2023-07-12

Version 1.6.184: resolved Issue 1644: **Minimize** (extension)

- description: processing.Minimize: improve printout of current error of objective function=loss; only print every 1 second
- date resolved: **2023-07-12 09:29**, date raised: 2023-07-12

Version 1.6.183: **resolved BUG 1643**: **Minimize**

- description: processing.Minimize function has an internal bug, such that it does not work with initialGuess=[]
- date resolved: **2023-07-12 09:29**, date raised: 2023-07-12

Version 1.6.182: **resolved BUG 1642**: **SolutionViewer**

- description: record image not working with visualizationSettings useMultiThreadedRendering=True
- date resolved: **2023-07-11 17:58**, date raised: 2023-07-11

Version 1.6.181: resolved Issue 1641: **SolutionViewer** (fix)

- description: github issue#51: graphicsDataUserFunction in SolutionViewer not called; add call to graphicsData user functions in redraw image loop
- date resolved: **2023-07-11 17:16**, date raised: 2023-07-11

Version 1.6.180: resolved Issue 1638: **GeneticOptimization** (extension)

- description: add argument parameterFunctionData to GeneticOptimization; same as in ParameterVariation, paramterFunctionData allows to pass additional data to the objective function
- date resolved: **2023-07-09 09:15**, date raised: 2023-07-09

Version 1.6.179: resolved Issue 1637: **add ChatGPT update information** (extension)

- description: create Python model Examples/chatGPTupdate.py which includes information that is used by ChatGPT4 to improve abilities to create simple models fully automatic
- date resolved: **2023-06-30 15:01**, date raised: 2023-06-30

Version 1.6.178: resolved Issue 1636: **CreateMassPoint** (change)

- description: change args referenceCoordinates to referencePosition, initialCoordinates to initialDisplacement, and initialVelocities to initialVelocity to be consistent with CreateRigidBody (but different from MassPoint itself)
- date resolved: **2023-06-30 14:09**, date raised: 2023-06-30

Version 1.6.177: resolved Issue 1635: **SmartRound2String** (extension)

- description: add function to basic utilities to enable simple printing of numbers with few digits, including comma dot and not eliminating small numbers, e.g., 1e-5 stays 1e-5

- date resolved: **2023-06-30 09:22**, date raised: 2023-06-30

Version 1.6.176: resolved Issue 1634: **create directories** (extension)
- description: add automatic creation of directories to FEM SaveToFile, plotting and ParameterVariation
- date resolved: **2023-06-29 10:29**, date raised: 2023-06-29

Version 1.6.175: **resolved BUG 1633**: **GetInterpolatedSignalValue**
- description: timeArray needs to be replaced with timeArrayNew in case of 2D input array
- date resolved: **2023-06-28 21:15**, date raised: 2023-06-28

Version 1.6.174: resolved Issue 1632: **PlotFFT** (fix)
- description: matplotlib >= 1.7 complains about ax.grid(b=...) as parameter b has been replaced by visible
- date resolved: **2023-06-27 14:15**, date raised: 2023-06-27

Version 1.6.173: resolved Issue 1626: **mutable args itemInterface** (fix)
- description: also copy dictionaries, mainly for visualization (flat level, but this should be sufficient)
- date resolved: **2023-06-21 10:40**, date raised: 2023-06-21

Version 1.6.172: resolved Issue 1627: **mutable default args** (change)
- description: complete changes and adaptations for default args in Python functions and item interface; note individual adaptations for lists, vectors, matrices and special lists of lists or matrix containers; for itemInterface, anyway all data is copied into C++; for more information see issues 1536, 1540, 1612, 1624, 1625, 1626
- date resolved: **2023-06-21 10:15**, date raised: 2023-06-21

Version 1.6.171: resolved Issue 1625: **change to default arg None** (change)
- description: change default args for Vector2DList, Vector3DList, Vector6DList, Matrix3DList, to None; ArrayNodeIndex, ArrayMarkerIndex, ArraySensorIndex obtain copy method and are copied now; avoid problem of mutable default args
- date resolved: **2023-06-21 00:26**, date raised: 2023-06-20

Version 1.6.170: resolved Issue 1624: **MatrixContainer** (change)
- description: change default values for matrix container to None; avoid problem of mutable default args
- date resolved: **2023-06-20 23:39**, date raised: 2023-06-20

Version 1.6.169: resolved Issue 1540: **mutable args itemInterface** (check)
- description: copy lists in itemInterface in order to avoid change of default args by user n=NodePoint();n.referenceCoordinates[0]=42;n1=NodePo
- **notes: simple vectors, matrices and lists are copied with np.array(...) while complex matrix and list of array types are now initialized with None**
- date resolved: **2023-06-20 22:13**, date raised: 2023-04-28

Version 1.6.168: resolved Issue 1620: **docu MainSystemExtensions** (docu)
- description: reorder MainSystemExtensions with separate section for Create functions and one section for remaining functions
- date resolved: **2023-06-19 22:14**, date raised: 2023-06-13

Version 1.6.167: **resolved BUG 1622**: **mouse click**
- description: fix crash on linux if left / right mouse click on render window (related to OpenGL select window)
- **notes: occurs on WSL with WSLg; using 'export LIBGL_ALWAYS_SOFTWARE=1' will resolve the problem; put this line into your .bashrc**
- date resolved: **2023-06-19 22:11**, date raised: 2023-06-19

Version 1.6.166: **resolved BUG 1621**: **LinearSolverType**
- description: fix crash on linux in function SetLinearSolverType
- date resolved: **2023-06-19 20:27**, date raised: 2023-06-19

Version 1.6.165: resolved Issue 1623: **MouseSelectOpenGL** (extension)
- description: add optional debbuging output
- date resolved: **2023-06-19 19:56**, date raised: 2023-06-19

Version 1.6.164: resolved Issue 1267: **matrix inverse** (extension)
- description: add pivot threshold to options, may improve redundant constraints problems
- **notes: only available for EigenDense with ignoreSingularJacobian and EXUdense linear solvers**
- date resolved: **2023-06-12 13:24**, date raised: 2022-09-21

Version 1.6.163: resolved Issue 1616: **eigen LU** (check)
- description: check fastest solver for regular and overdetermined systems
- **notes: Eigen::PartialPivLU 2.5 times faster for 65 DOF test in factorization, factor 3 faster for backsubst**
- date resolved: **2023-06-12 13:22**, date raised: 2023-06-11

Version 1.6.162: resolved Issue 1607: **Bricard mechanism** (testing)
- description: add example and test ComputeSystemDegreesOfFreedom
- date resolved: **2023-06-12 11:02**, date raised: 2023-06-09

Version 1.6.161: resolved Issue 1617: **solver error message** (fix)
- description: revise hint for ignoreSingularJacobian

- date resolved: **2023-06-12 01:39**, date raised: 2023-06-11

Version 1.6.160: resolved Issue 1615: **pivotThreshold** (fix)
- description: add already existing parameter [which is currently not used in solver!] to solver interface add FactorizeNew arg
- date resolved: **2023-06-12 01:39**, date raised: 2023-06-11

Version 1.6.159: resolved Issue 1266: **matrix inverse** (extension)
- description: add full pivoting mode for matrix inverse, to resolve redundant constraints; consider Eigen FullPivotLU for dense matrices - see classEigen_1_1FullPivLU.html
- **notes: also added new LinearSolverType.EigenDense which allows to chose FullPivLU by settings ignoreSingularJacobian=True**
- date resolved: **2023-06-12 00:17**, date raised: 2022-09-21

Version 1.6.158: resolved Issue 1619: **LinearSolverType** (change)
- description: switch to bit-wise numbering of solver types, in order to alleviate checks
- date resolved: **2023-06-11 23:52**, date raised: 2023-06-11

Version 1.6.157: resolved Issue 1618: **ignoreRedundantConstraints** (change)
- description: remove option ignoreRedundantConstraints as it cannot be applied with Eigen::FullPivLU; use ignoreSingularJacobian instead
- date resolved: **2023-06-11 23:46**, date raised: 2023-06-11

Version 1.6.156: resolved Issue 1613: **Experimental** (extension)
- description: add experimental class, which can be accessed in Python by exudyn.Experimental(); inside C++, just needs to be imported; allows simple testing without interference with main features
- date resolved: **2023-06-11 19:56**, date raised: 2023-06-11

Version 1.6.155: resolved Issue 1536: **mutable arguments** (fix)
- description: check and fix Python functions with mutable arguments such as [] or , with potential risk of changing internally in function, leading to unexpected behavior in second call
- **notes: checked all default list and dict args**
- date resolved: **2023-06-11 00:24**, date raised: 2023-04-27

Version 1.6.154: resolved Issue 1612: **mutable arguments** (fix)
- description: check and fix problems in beams.py, FEM.py and graphicsDataUtilities.py
- **notes: see also issue 1536**
- date resolved: **2023-06-10 21:34**, date raised: 2023-06-10

Version 1.6.153: resolved Issue 1609: **AnimateModes** (extension)
- description: extend for using a set of system eigenmodes
- date resolved: **2023-06-10 20:25**, date raised: 2023-06-10

Version 1.6.152: resolved Issue 1580: **mainSystemExtensions** (extension)
- description: add RigidBodySpringDamper
- date resolved: **2023-06-10 20:25**, date raised: 2023-05-21

Version 1.6.151: resolved Issue 1606: **ComputeODE2Eigenvalues** (extension)
- description: add eigenvector computation to constrained case
- **notes: needs further testing!**
- date resolved: **2023-06-10 19:11**, date raised: 2023-06-08

Version 1.6.150: resolved Issue 1611: **SolutionViewer** (change)
- description: change internal variables from mbs.variables to mbs.sys
- date resolved: **2023-06-10 17:48**, date raised: 2023-06-10

Version 1.6.149: resolved Issue 1610: **AnimateModes** (change)
- description: change internal variables from mbs.variables to mbs.sys
- date resolved: **2023-06-10 17:47**, date raised: 2023-06-10

Version 1.6.148: **resolved BUG 1608**: **visualization zoom all**
- description: when calling ComputeSystemDegreeOfFreedom, ComputeODE2Eigenvalues and similar functions, and StartRenderer is called right afterwards, zoom all does not work
- **notes: shall be resolved just by calling StartRenderer before first call to any solver functionality**
- date resolved: **2023-06-10 11:44**, date raised: 2023-06-10

Version 1.6.147: resolved Issue 1435: **solver** (extension)
- description: add deriviative of loads to regular jacobian computation with flag (default=False); use numerical diff sim. to JacobianODE2
- **notes: already done earlier in #1546**
- date resolved: **2023-06-08 23:35**, date raised: 2023-02-16

Version 1.6.146: resolved Issue 1597: **Command execute** (fix)
- description: switch to grid method for placing widgets, tkinter does not allow pack and grid in different windows

- date resolved: **2023-06-08 21:56**, date raised: 2023-06-05

Version 1.6.145: resolved BUG 1593: **TemporaryComputationDataArray bug**

- description: ERROR: "TemporaryComputationDataArray::operator[]: index out of range" is raised if single-threaded computation is run after multi-threaded simulation; requires restart of Python instance
- date resolved: **2023-06-08 18:50**, date raised: 2023-06-03

Version 1.6.144: resolved Issue 1599: **ComputeODE2Eigenvalues** (extension)

- description: compute eigenmodes in case of algebraic equations
- date resolved: **2023-06-08 18:46**, date raised: 2023-06-08

Version 1.6.143: resolved Issue 1603: **AddSensor** (extension)

- description: add check if no outputVariable is provided-> immediately raise error
- **notes: was already in system checks, which however were not called, see issue 1604**
- date resolved: **2023-06-08 18:45**, date raised: 2023-06-08

Version 1.6.142: resolved Issue 1598: **eigenvalues constrained system** (example)

- description: add example for eigenvalue computation of constrained system
- **notes: added computeODE2AEeigenvaluesTest.py**
- date resolved: **2023-06-08 18:45**, date raised: 2023-06-08

Version 1.6.141: resolved Issue 1605: **ComputeSystemDegreeOfFreedom** (change)

- description: change output to a dictionary in order to have readable results
- date resolved: **2023-06-08 18:35**, date raised: 2023-06-08

Version 1.6.140: resolved Issue 1604: **Sensors** (fix)

- description: PreAssembleConsistencies not called in Assemble; thus, no checks are performed on sensor inputs
- date resolved: **2023-06-08 18:24**, date raised: 2023-06-08

Version 1.6.139: resolved BUG 1602: **MainSystem CreateRigidBody**

- description: referenceRotationMatrix multiplied in wrong way with initialRotationMatrix
- **notes: also rotation parameters in initialVelocities were wrong for initialRotationMatrix!=np.eye(3); fixed, but more testing needed**
- date resolved: **2023-06-08 17:42**, date raised: 2023-06-08

Version 1.6.138: resolved Issue 1600: **stub files** (extension)

- description: extend .pyi files for system structures functions, e.g., ComputeJacobianODE2RHS
- date resolved: **2023-06-08 17:17**, date raised: 2023-06-08

Version 1.6.137: resolved Issue 1601: **stub files** (change)

- description: merge .pyi files to have classes such as MainSystem only appearing once
- date resolved: **2023-06-08 16:37**, date raised: 2023-06-08

Version 1.6.136: resolved Issue 0746: **ComputeODEEigenvalues** (extension)

- description: add possibility to eliminate coordinate constraints, possibly to use SVD/ null space matrix for projection
- **notes: algebraic constraints now considered automatically; algebraic constraints now considered automatically**
- date resolved: **2023-06-07 23:52**, date raised: 2021-09-03 (resolved by: M. Pieber, JG)

Version 1.6.135: resolved Issue 0743: **ComputeODE2Eigenvalues** (extension)

- description: add vector of constrained coordinates which are eliminated; also add functionality for complex eigenvalues
- **notes: already done earlier**
- date resolved: **2023-06-07 23:09**, date raised: 2021-08-20

Version 1.6.134: resolved Issue 1595: **Command dialog** (extension)

- description: extend to multi-line commands; execute code using CTRL-Return
- **notes: Behavior is now DIFFERENT, as variables are not printed automatically; write e.g. print(mbs) to see mbs representation; see section Execute Command and Help**
- date resolved: **2023-06-04 19:40**, date raised: 2023-06-03

Version 1.6.133: resolved Issue 1596: **linuxDisplayScaleFactor** (extension)

- description: add scaling for fonts on linux, specifically for high resolution screens
- date resolved: **2023-06-04 00:13**, date raised: 2023-06-04

Version 1.6.132: resolved Issue 1588: **ParameterVariation, useMPI** (fix)

- description: only accept useMPI if set True
- date resolved: **2023-06-03 19:26**, date raised: 2023-05-31

Version 1.6.131: resolved Issue 1579: **mainSystemExtensions** (extension)

- description: Add distance constraint and CartesianSpringDamper
- date resolved: **2023-06-03 19:26**, date raised: 2023-05-21

Version 1.6.130: resolved Issue 1594: **window closing, key Q** (change)

- description: slightly adapt behavior; fix some smaller issues with expected window behavior

- date resolved: **2023-06-03 17:01**, date raised: 2023-06-03

Version 1.6.129: resolved Issue 1356: **CHECK** (extension)

- description: Add security question on quit/escape if computation Renderer runs longer than 15 minutes
- **notes: added message in render window to click twice on exit window icon (X) after 15 seconds; key Q and escape get tkinter message box**
- date resolved: **2023-06-03 16:00**, date raised: 2023-01-01

Version 1.6.128: resolved Issue 1592: **SolverBase it.endTime** (fix)

- description: in dynamic solver it.endTime is overwritten with simulationSettings.timeIntegration.endTime; this is against the description and does not allow to change it.endTime in command window; remove overwritting to be consistent with description of Execute Command and Help section in EXUDYN Basics
- date resolved: **2023-06-03 14:25**, date raised: 2023-06-03

Version 1.6.127: resolved Issue 1591: **SphericalJoint** (fix)

- description: causes memory allocation; check LinkedDataVector cast
- date resolved: **2023-06-01 11:12**, date raised: 2023-06-01

Version 1.6.126: resolved Issue 1590: **serialRobotKinematicTree.py** (fix)

- description: fixed static torque compensation for kinematic tree and fixed sensor outputs
- date resolved: **2023-05-31 13:07**, date raised: 2023-05-31

Version 1.6.125: resolved Issue 1589: **ObjectKinematicTree** (docu)

- description: add description of SensorKinematicTree output variables per link
- date resolved: **2023-05-31 12:42**, date raised: 2023-05-31

Version 1.6.124: resolved Issue 1587: **solution file footer** (change)

- description: add comma after cpuTime=...
- date resolved: **2023-05-30 23:52**, date raised: 2023-05-30

Version 1.6.123: resolved Issue 1586: **SetPreStepUserFunction, SetPostNewtonUserFunction** (extension)

- description: add exception handling in set function
- date resolved: **2023-05-26 19:52**, date raised: 2023-05-26

Version 1.6.122: resolved Issue 1585: **class name highlighting RTD** (docu)

- description: fixed exporting class names from pythonUtilities
- date resolved: **2023-05-25 15:19**, date raised: 2023-05-25

Version 1.6.121: resolved Issue 1584: **MiniExamples** (change)

- description: remove import of itemInterface
- date resolved: **2023-05-25 14:32**, date raised: 2023-05-25

Version 1.6.120: **resolved BUG 1583**: **GeneticOptimization**

- description: results are erroneous in case of crossoverProbability > 0
- **notes: fixed writing of output files which had mixed order due to parameter cross-over**
- date resolved: **2023-05-23 18:31**, date raised: 2023-05-23

Version 1.6.119: resolved Issue 1582: **mainSystemExtensions** (fix)

- description: remove import of tkinter and matplotlib to resolve errors when loading exudyn and these libs are not installed
- date resolved: **2023-05-22 10:54**, date raised: 2023-05-22

Version 1.6.118: resolved Issue 1577: **examples** (fix)

- description: test if all Examples are still running
- date resolved: **2023-05-20 22:23**, date raised: 2023-05-20

Version 1.6.117: resolved Issue 1578: **ObjectConnectorCoordinateSpringDamper** (fix)

- description: correct docu on object and user function description (still includes friction)
- date resolved: **2023-05-20 21:13**, date raised: 2023-05-20

Version 1.6.116: resolved Issue 1569: **mainSystemExtensions** (example)

- description: add mini-examples for extensions
- **notes: collected miniexamples in mainSystemExtensionsTests.py;**
- date resolved: **2023-05-20 21:05**, date raised: 2023-05-15

Version 1.6.115: resolved Issue 1571: **mainSystemExtensions** (change)

- description: adapt Examples to Python extensions (SolveDynamic, CreateRigidBody, CreateGenericJoint, ...)
- date resolved: **2023-05-18 23:43**, date raised: 2023-05-15

Version 1.6.114: resolved Issue 1570: **mainSystemExtensions** (change)

- description: adapt TestModels to Python extensions
- date resolved: **2023-05-18 23:43**, date raised: 2023-05-15

Version 1.6.113: **resolved BUG 1576**: **multithreading**

- description: running laserScannerTest.py after a multithreaded contact computation raises the EXCEPTION: TemporaryComputationDataArray::operator[]: index out of range
- date resolved: **2023-05-18 23:13**, date raised: 2023-05-18

Version 1.6.112: resolved Issue 1575: **ConnectorDistance** (change)

- description: changed parameter distance to PReal, not allowing zero distance to be prescribed
- date resolved: **2023-05-18 13:03**, date raised: 2023-05-18

Version 1.6.111: resolved Issue 1573: **mainSystemExtensions** (docu)

- description: Adapt tutorials to new functionality
- date resolved: **2023-05-18 12:19**, date raised: 2023-05-16

Version 1.6.110: resolved Issue 1574: **Python utilities** (fix)

- description: classes not appearing in table of contents on RTD
- date resolved: **2023-05-17 20:21**, date raised: 2023-05-16

Version 1.6.109: resolved Issue 1572: **CreateRigidBody** (extension)

- description: added to MainSystem to enable mbs.CreateRigidBody(...)
- date resolved: **2023-05-16 11:54**, date raised: 2023-05-16

Version 1.6.108: resolved Issue 1568: **mainSystemExtensions** (extension)

- description: create first sample of Python extensions for basic joints
- date resolved: **2023-05-15 18:13**, date raised: 2023-05-15

Version 1.6.107: resolved Issue 1567: **DrawSystemGraph** (change)

- description: in case of showItemNames, no item numbers are shown as they confuse with numbers used in names
- date resolved: **2023-05-15 17:18**, date raised: 2023-05-15

Version 1.6.106: resolved Issue 1566: **AddDistanceSensor(...)** (change)

- description: function RENAMED into CreateDistanceSensor(...) to be consistent with future naming; also renamed DistanceSensorSetupGeometry(...) into CreateDistanceSensorGeometry(...)
- date resolved: **2023-05-15 11:34**, date raised: 2023-05-15

Version 1.6.105: resolved Issue 1563: **MainSystem Python extensions** (extension)

- description: add Python utility functions for mbs, such as PlotSensor, SolveDynamic, ...; use identical interfaces to alleviate creation of .pyi files and documentation; add new flag mbsFunction as hint to put docu to MainSystem and make .pyi extension
- **notes: see Section 6.5.2 for extended functionality**
- date resolved: **2023-05-15 01:17**, date raised: 2023-05-09

Version 1.6.104: resolved Issue 1564: **Type definitions** (docu)

- description: fix header structure in latex and RST for Type Definitions
- date resolved: **2023-05-11 11:30**, date raised: 2023-05-11

Version 1.6.103: resolved Issue 1561: **stub files .pyi** (extension)

- description: add .pyi files to setup_tools, copying them from autogenerate folder; use try catch to avoid problems at other platforms
- date resolved: **2023-05-10 23:30**, date raised: 2023-05-09

Version 1.6.102: resolved Issue 1560: **stub files .pyi** (extension)

- description: automatically create stub file for C++ classes such as MainSystem, SystemContainer, GeneralContact, ... by adding return type information and creating .pyi file; use temporary .pyi files in autogenerate folder
- date resolved: **2023-05-10 23:19**, date raised: 2023-05-09

Version 1.6.101: resolved Issue 1562: **stub files .pyi** (extension)

- description: add .pyi files for enums from autoGeneratePyBindings
- date resolved: **2023-05-10 20:43**, date raised: 2023-05-09

Version 1.6.100: resolved Issue 1559: **stub files .pyi** (extension)

- description: automatically create stub file for settings to alleviate auto-completion; type completion now also works for functions, types and structures: tested in Spyder and Visual Studio Code
- date resolved: **2023-05-10 08:39**, date raised: 2023-05-09

Version 1.6.99: resolved Issue 1557: **stub files** (check)

- description: test creating stub files .pyi which are needed for MainSystem Python extensions
- **notes: tested with Spyder 5.1.5 and Visual Studio Code 1.78**
- date resolved: **2023-05-10 08:39**, date raised: 2023-05-07

Version 1.6.98: resolved Issue 1558: **enum in global scope** (fix)

- description: pybind11 translates enums to global module scope, e.g. exudyn.ItemType.Marker is also available as exudyn.Marker; remove export_values() in pybind interface
- **notes: if you by occasion used e.g. exu.DisplacementLocal instead of exu.OutputVariableType.DisplacementLocal you need to adapt your code!**
- date resolved: **2023-05-09 18:30**, date raised: 2023-05-09

Version 1.6.97: resolved Issue 1556: **IsValidPRealPInt, IsValidURealPInt** (extension)

- description: add functions to advancedUtilities for additional checks in Python functions
- date resolved: **2023-05-07 20:33**, date raised: 2023-05-07

Version 1.6.96: resolved Issue 1555: **color4default** (extension)

- description: add default color to graphicsDataUtilities as a default value for items
- date resolved: **2023-05-07 18:12**, date raised: 2023-05-07

Version 1.6.95: resolved Issue 1554: **NodePoint2D** (change)

- description: draw as sphere by default to improve visibility
- date resolved: **2023-05-07 16:31**, date raised: 2023-05-07

Version 1.6.94: resolved Issue 1539: **item names** (change)

- description: remove stored string and replace by empty string in case of default item name
- **notes: not changed: std::string has practically no effect on memory footprint of items; check memory footprint separately (LTG lists, etc.)!**
- date resolved: **2023-05-05 23:32**, date raised: 2023-04-28

Version 1.6.93: resolved Issue 1553: **AccelerationLocal, AngularAccelerationLocal** (fix)

- description: add missing values to Pybind interface
- date resolved: **2023-05-02 17:41**, date raised: 2023-05-02

Version 1.6.92: resolved Issue 1552: **HydraulicActuator** (extension)

- description: add VelocityLocal as output variable, which provides the time derivative of the distance, being the actuator velocity; updated docu and fixed description for outputvariable Velocity
- date resolved: **2023-05-02 16:51**, date raised: 2023-05-02

Version 1.6.91: resolved Issue 1551: **GeometricallyExactBeam** (change)

- description: evaluate rotation at midspan of beam
- date resolved: **2023-05-02 15:02**, date raised: 2023-05-02

Version 1.6.90: resolved Issue 1502: **ANCFBeam** (testing)

- description: check for very large deformations
- **notes: poor performance for large deformations caused by missing load jacobian**
- date resolved: **2023-05-01 19:26**, date raised: 2023-04-08

Version 1.6.89: resolved Issue 1501: **GeometricallyExactBeam** (testing)

- description: check for very large deformations
- **notes: poor performance for large deformations caused by missing load jacobian**
- date resolved: **2023-05-01 19:26**, date raised: 2023-04-08

Version 1.6.88: resolved Issue 1546: **computeLoadsJacobian** (change)

- description: add flag in timeIntegration and staticSolver settings to turn on/off jacobian of loads; will be by default turned on in static solver, turned off in time integration; adapt your models
- date resolved: **2023-05-01 19:22**, date raised: 2023-05-01

Version 1.6.87: resolved Issue 1545: **ComputeSingleLoads** (extension)

- description: adapt for Jacobian computation of loads
- date resolved: **2023-05-01 19:22**, date raised: 2023-05-01

Version 1.6.86: resolved Issue 1547: **GetMarkerOutput** (fix)

- description: in case of OuputVariableType.Coordinates, add check if Marker is of type Coordinate or Coordinates
- date resolved: **2023-05-01 16:17**, date raised: 2023-05-01

Version 1.6.85: resolved Issue 1544: **systemData.GetNodeLocalToGlobal** (extension)

- description: add systemData access functions for LTG mappings of nodes, useful to check node coordinates; works for ODE2, ODE1, AE and Data coordinates; useful to create load dependencies
- date resolved: **2023-04-29 22:03**, date raised: 2023-04-29

Version 1.6.84: resolved Issue 1543: **systemData** (extension)

- description: add InfoLTG function which outputs LTG lists and load dependencies
- date resolved: **2023-04-29 22:03**, date raised: 2023-04-29

Version 1.6.83: resolved Issue 0483: **URDF file** (extension)

- description: check importing an URDF file for robots, urdf_parser_py
- **notes: not done: should be done instead by Corke robotics-toolbox**
- date resolved: **2023-04-29 20:20**, date raised: 2020-12-04

Version 1.6.82: resolved Issue 1542: **HydraulicsActuator** (change)

- description: changing referenceVolume0 and referenceVolume1 to hoseVolume0 and hoseVolume1 with different meaning according to referenced paper; adjust your models!
- **notes: thanks to Qasim Khadim for provigind the model**

- date resolved: **2023-04-29 01:11**, date raised: 2023-04-29

Version 1.6.81: resolved Issue 1541: **HydraulicsActuator** (extension)

- description: extend model of effective bulk modulus acc. to paper https://doi.org/10.1007/s11044-019-09696-y
- date resolved: **2023-04-29 01:10**, date raised: 2023-04-28

Version 1.6.80: resolved Issue 1538: **taskmanager** (fix)

- description: fix problem with taskmanager shutdown due to issue 1532
- date resolved: **2023-04-27 15:35**, date raised: 2023-04-27

Version 1.6.79: resolved Issue 1537: **output.multiThreadingMode** (fix)

- description: has not been set correctly in solver so far; switch output.numberOfThreadsUsed and output.multiThreadingMode
- date resolved: **2023-04-27 15:35**, date raised: 2023-04-27

Version 1.6.78: resolved Issue 1535: **writeSensors** (extension)

- description: add global flag to deactivate sensor file creating/writing and sensor storing; set flag in solver functions such as ComputeLinearizedSystem, etc. to avoid erasing sensor files or sensor data
- date resolved: **2023-04-27 11:34**, date raised: 2023-04-26

Version 1.6.77: resolved Issue 1533: **FinalizeSolver** (fix)

- description: call FinalizeSolver in ComputeLinearizedSystem, ComputeSystemDegreeOfFreedom, ComputeODE2Eigenvalues for consistency reason; also deactivate file writing and sensor writing and solverInformation writing
- date resolved: **2023-04-26 19:25**, date raised: 2023-04-26

Version 1.6.76: resolved Issue 1532: **CSolverBase.cpp** (fix)

- description: close output and sensor files in destructor of CSolverBase
- date resolved: **2023-04-26 19:24**, date raised: 2023-04-26

Version 1.6.75: resolved Issue 1534: **EXUlie** (change)

- description: improve TExpSE3 and TExpSE3Inv regarding small values according to PhD thesis of Stefan Hante
- **notes: improves numerical behavior and convergence of GeometricallyExactBeam**
- date resolved: **2023-04-26 18:33**, date raised: 2023-04-26

Version 1.6.74: resolved Issue 1531: **OpenVR** (fix)

- description: change order of eye transformation and projection in OpenVRinterface GetCurrentViewProjectionMatrix to be consistent with master thesis
- date resolved: **2023-04-26 16:49**, date raised: 2023-04-26

Version 1.6.73: resolved Issue 1530: **ComputeSystemDegreeOfFreedom** (extension)

- description: add exudyn.solver function to numerically compute DOF of constrained mechanisms
- date resolved: **2023-04-26 11:16**, date raised: 2023-04-26

Version 1.6.72: resolved Issue 1528: **structures and settings** (docu)

- description: function arguments in RST / html are inappropriately noted; fix; replace true/false with True/False
- date resolved: **2023-04-26 09:16**, date raised: 2023-04-26

Version 1.6.71: resolved Issue 1527: **GeneralContact** (extension)

- description: add function UpdateContacts, which computes current bounding boxes and active contacts, to be used in access functions to GeneralContact if isActive=False (otherwise this is anyway done in contact computations of every computation step)
- date resolved: **2023-04-23 20:15**, date raised: 2023-04-23

Version 1.6.70: resolved Issue 0935: **GeneralContact** (extension)

- description: add interface function to get contact pairs
- **notes: function GetActiveContacts added to GeneralContact, which returns all active global contact indices for selected contact type**
- date resolved: **2023-04-23 20:13**, date raised: 2022-02-10

Version 1.6.69: resolved Issue 1516: **solver failed function** (extension)

- description: add function to check if solver failed, using stored solver structure as input; return True/False and string (optionally error code) describing failure
- **notes: added function SolverSuccess() to exudyn.solver; returns success and error string as created by solver internal function GetErrorString(...)**
- date resolved: **2023-04-23 18:43**, date raised: 2023-04-19

Version 1.6.68: resolved Issue 1526: **solver GetErrorString** (extension)

- description: MainSolverStatic, MainSolverExplicit, MainSolverImplicit now get function GetErrorString to obtain error string set if SolveSteps or SolveSystem failed (returned false)
- date resolved: **2023-04-23 11:46**, date raised: 2023-04-23

Version 1.6.67: resolved Issue 1525: **output.finishedSuccessfully** (extension)

- description: flag is now set both in SolveSteps(...) as well in SolveSystem(...) to indicate if solver has been successful or failed; practical flag for lateron determination of solver errors

- date resolved: **2023-04-23 11:46**, date raised: 2023-04-23

Version 1.6.66: resolved Issue 1524: **netgen STL file** (examples)

- description: add example with netgen and STL files with meshing
- date resolved: **2023-04-21 17:51**, date raised: 2023-04-21

Version 1.6.65: resolved Issue 1521: **ObjectFFRFreducedOrderInterface** (extension)

- description: add LoadFromFile/SaveToFile similar to FEMinterface
- **notes: this function should be used to store CMS data if FEM is too large to load/store; CMS data still stores all node positions, triangle list (for visualization) and modeBasis for computation tasks; may be still large e.g. for many nodes and large number of modes**
- date resolved: **2023-04-20 21:06**, date raised: 2023-04-20

Version 1.6.64: resolved Issue 1519: **FEMinterface** (change)

- description: add version to LoadFromFile/SaveToFile function; store file version as first field to load/store in order to be able to load older data as well; use forceVersion=0 to load files in old format
- **notes: warning is printed if old file is loaded**
- date resolved: **2023-04-20 20:59**, date raised: 2023-04-19

Version 1.6.63: resolved Issue 1523: **StrNodeType2NodeType** (extension)

- description: add function to rigidBodyUtilities in order to make str to type conversion
- date resolved: **2023-04-20 18:34**, date raised: 2023-04-20

Version 1.6.62: resolved Issue 1522: **ObjectFFRFreducedOrderInterface** (change)

- description: remove femInterface from internal variables, as it is only used for postProcessingModes; store postProcessingModes instead
- date resolved: **2023-04-20 16:34**, date raised: 2023-04-20

Version 1.6.61: resolved Issue 1520: **ImportFromAbaqusInputFile** (change)

- description: use VolumeToSurfaceElements for creating of surface elements; add option to automatically create surface triangles
- **notes: by default, only surface triangles are created!**
- date resolved: **2023-04-20 15:44**, date raised: 2023-04-20

Version 1.6.60: resolved Issue 1518: **ObjectFFRFreducedOrderInterface** (fix)

- description: roundMassMatrix and roundStiffnessMatrix are not used
- **notes: added as arguments in RoundMatrix**
- date resolved: **2023-04-19 19:08**, date raised: 2023-04-19

Version 1.6.59: resolved Issue 1517: **ImportFromAbaqusInputFile** (extension)

- description: extended for Tet4 and Tet10 as well as C3D20R elements and added function ConvertTetToTrigs(...)
- date resolved: **2023-04-19 18:14**, date raised: 2023-04-19

Version 1.6.58: resolved Issue 1515: **unused header files** (cleanup)

- description: remove unused header files for C, Main and Visu: JointPrismatic.h, JointRevolute.h
- date resolved: **2023-04-16 13:03**, date raised: 2023-04-16

Version 1.6.57: resolved Issue 1269: **LaserSensor** (extension)

- description: add advanced distance sensors replicating laser scanner (with axis, revolution speed and initial direction)
- **notes: added function AddLidar(...) into exudyn.robotics.utilities; see laserScannerTest.py**
- date resolved: **2023-04-15 15:33**, date raised: 2022-09-21

Version 1.6.56: resolved Issue 1270: **DistanceSensor** (extension)

- description: add advanced distance sensor with possibility to use a set of beams and averaging or multiple output
- **notes: added DistanceSensorSetupGeometry in exudyn.utilities to set up contact geometry easily; see laserScannerTest.py**
- date resolved: **2023-04-15 15:32**, date raised: 2022-09-21

Version 1.6.55: **resolved BUG 1514**: **GetInterpolatedSignalValue**

- description: check for simple test, seems not to work
- **notes: changed order of args: dataArrayIndex and timeArrayIndex**
- date resolved: **2023-04-15 14:34**, date raised: 2023-04-14

Version 1.6.54: resolved Issue 1513: **RevoluteJoint, PrismaticJoint** (fix)

- description: both ObjectJointPrismaticX and ObjectJointRevoluteZ have wrong internal typename JointRevolute; change to correct names; when analyzing such objects, they will return wrong typenames
- date resolved: **2023-04-14 14:36**, date raised: 2023-04-14

Version 1.6.53: resolved Issue 1511: **AddDistanceSensor** (extension)

- description: add optional color for laser beam
- date resolved: **2023-04-13 17:45**, date raised: 2023-04-13

Version 1.6.52: **resolved BUG 1510**: **AddDistanceSensor**

- description: checks performed with invalid marker number

- date resolved: **2023-04-13 10:28**, date raised: 2023-04-13

Version 1.6.51: resolved Issue 1141: **object factory** (extension)
- description: consider hash tables for object factory; check current performance?
- **notes: no changes; string comparison has no effect on performance as compared to new and pybind11 overheads for call to AddObject, etc.**
- date resolved: **2023-04-10 21:48**, date raised: 2022-06-12

Version 1.6.50: resolved Issue 1506: **ODE2Size** (fix)
- description: and other functions have additional ConfigurationType: remove
- **notes: kept the argument ConfigurationType; usually, all configuration types have same sizes but the call must be related to a speficic configuration**
- date resolved: **2023-04-10 21:42**, date raised: 2023-04-08

Version 1.6.49: resolved Issue 1509: **SystemData** (fix)
- description: prevent from creation of a pure SystemData in exudyn; check constructor used in SC.AddSystem and SystemData()
- date resolved: **2023-04-10 21:37**, date raised: 2023-04-10

Version 1.6.48: resolved Issue 1508: **MainSystem** (fix)
- description: prevent from creation of a pure MainSystem in exudyn; check constructor used in SC.AddSystem and MainSystem()
- date resolved: **2023-04-10 21:37**, date raised: 2023-04-10

Version 1.6.47: resolved Issue 1507: **SystemContainer** (docu)
- description: add description for SystemContainer itself; sying that it behaves like a variable in Python
- date resolved: **2023-04-10 21:36**, date raised: 2023-04-10

Version 1.6.46: resolved Issue 1503: **item description** (docu)
- description: add general description to RST / sphinx
- date resolved: **2023-04-08 19:04**, date raised: 2023-04-08

Version 1.6.45: resolved Issue 1496: **AddMainObjectPyClass** (check)
- description: C++: for adding objects, nodes, etc. currently py::object and py::dict is copied: check performance increase, if it is passed by reference
- **notes: tests show small performance improvements (<10 percent) for creation of items**
- date resolved: **2023-04-08 17:36**, date raised: 2023-04-07

Version 1.6.44: resolved Issue 1498: **MarkerNodeRotationCoordinate** (check)
- description: check if Orientation type is correct, see docu
- **notes: removed Marker::Orientation from MarkerNodeRotationCoordinate as it is not provided**
- date resolved: **2023-04-08 17:30**, date raised: 2023-04-08

Version 1.6.43: **resolved BUG 1497**: **PyBeamSection**
- description: C++: segmentation fault in linux, caused by conversion from numpy array into std::array; use SetConstMatrixTemplateSafely for writing matrix or list
- date resolved: **2023-04-07 19:05**, date raised: 2023-04-07

Version 1.6.42: **resolved BUG 1495**: **BeamSection**
- description: PyBeamSection does not call parent constructor (BeamSection); leads to seg fault in linux version
- date resolved: **2023-04-07 13:32**, date raised: 2023-04-07

Version 1.6.41: resolved Issue 1272: **GeometricallyExactBeam** (check)
- description: check jacobian computation as numerical differentiation works better; similar to #1100
- **notes: fixed bug of GeometricallyExactBeam having wrong jacobian**
- date resolved: **2023-04-06 18:28**, date raised: 2022-09-24

Version 1.6.40: resolved Issue 1492: **Python utilities** (docu)
- description: fix indentation of examples
- date resolved: **2023-04-06 14:55**, date raised: 2023-04-06

Version 1.6.39: resolved Issue 1491: **ProcessParameterList** (fix)
- description: remove addComputationIndex, as it is unused
- date resolved: **2023-04-06 14:35**, date raised: 2023-04-06

Version 1.6.38: resolved Issue 1156: **renderState** (docu)
- description: add description of render state from C++ into theDoc
- **notes: added section Render State in section Graphics and Visualization (GUI)**
- date resolved: **2023-04-05 14:14**, date raised: 2022-06-21

Version 1.6.37: resolved Issue 1468: **sphinx** (docu)
- description: add issue and bugs section with resolved issues and open issues table; resolved issues with version
- **notes: already done earlier**
- date resolved: **2023-04-05 14:08**, date raised: 2023-03-22

Version 1.6.36: resolved Issue 1490: **ANCFBeam** (change)

- description: remove testBeamRectangularSize
- date resolved: **2023-04-05 13:59**, date raised: 2023-04-05

Version 1.6.35: resolved Issue 1088: **ANCFBeam3D** (extension)

- description: add test model, compare to 2013 paper
- date resolved: **2023-04-04 20:59**, date raised: 2022-05-16

Version 1.6.34: **resolved BUG 1489**: **CObjectANCFBeam**

- description: Computation of elastic forces uses global instead of local twist-and-curvature vector
- date resolved: **2023-04-04 20:04**, date raised: 2023-04-04

Version 1.6.33: resolved Issue 1488: **NodePoint3DSlope23** (extension)

- description: add full rigid body output values to node (e.g., Rotation missing)
- date resolved: **2023-04-04 13:31**, date raised: 2023-04-04

Version 1.6.32: resolved Issue 1487: **Acceleration** (extension)

- description: add GetAcceleration function to all nodes; add acceleration to all ODE2-based node output variables
- date resolved: **2023-04-04 13:27**, date raised: 2023-04-04

Version 1.6.31: resolved Issue 1482: **EnterTaskManager** (change)

- description: removed call to EnterTaskManager in serial mode as it causes large overhead; also removed large array for tracer
- date resolved: **2023-03-28 18:47**, date raised: 2023-03-28

Version 1.6.30: resolved Issue 1481: **InverseKinematicsNumerical** (change)

- description: changed SolveIKine to Solve and changed OutputVariable Rotation to RotationMatrix at tool
- date resolved: **2023-03-28 16:30**, date raised: 2023-03-28

Version 1.6.29: resolved Issue 1480: **LogSE3** (extension)

- description: fixed according to LogSO3 and added efficient version with vectors in C++
- date resolved: **2023-03-28 15:10**, date raised: 2023-03-28

Version 1.6.28: resolved Issue 1478: **LogSO3** (fix)

- description: Add new LogSO3 C++ function, also to be used in LogSE3 which fully works for 0..pi rotation range; improved accuracy for very small rotations as well as rotations close to pi, e.g. 0.99999999*pi, where the standard approach fails
- date resolved: **2023-03-28 15:09**, date raised: 2023-03-27

Version 1.6.27: resolved Issue 1469: **RotationMatrix2Rxyz** (fix)

- description: extend implementation for rot[1]=pi/2
- **notes: resolved both in Python and C++; some simulation results may change, especially in output of Rotations in sensors**
- date resolved: **2023-03-28 14:46**, date raised: 2023-03-22

Version 1.6.26: resolved Issue 1477: **LogSO3** (fix)

- description: Add new LogSO3 Python function, also to be used in LogSE3 which fully works for 0..pi rotation range; improved accuracy for very small rotations as well as rotations close to pi, e.g. 0.99999999*pi, where the standard approach fails
- date resolved: **2023-03-27 20:16**, date raised: 2023-03-27

Version 1.6.25: resolved Issue 1476: **RotationMatrix2...** (fix)

- description: RotationMatrix2EulerParameters, RotationMatrix2RotXYZ, RotationMatrix2RotZYZ not working both with list of lists and np.arrays
- date resolved: **2023-03-27 19:58**, date raised: 2023-03-27

Version 1.6.23: resolved Issue 0312: **Add all types to pybind** (extension)

- description: add remaining types to pybind - for user elements
- **notes: important types already added; further types currently not planned**
- date resolved: **2023-03-27 00:45**, date raised: 2020-01-10

Version 1.6.22: resolved Issue 0848: **add demos to github** (docu)

- description: add demo videos to github or to youtube
- **notes: already resolved earlier: added videos to youtube**
- date resolved: **2023-03-27 00:42**, date raised: 2021-12-26

Version 1.6.21: resolved Issue 0843: **connector jacobian springDamper** (extension)

- description: add analytic jacobian for SpringDamper connector
- **notes: already resolved earlier**
- date resolved: **2023-03-27 00:42**, date raised: 2021-12-23

Version 1.6.20: resolved Issue 0851: **ComputeObjectODE2LHS** (extension)

- description: add computation functions for bodies and connectors; for connectors, markerData is computed automatically
- **notes: already resolved earlier with precomputed lists**
- date resolved: **2023-03-27 00:40**, date raised: 2022-01-08

Version 1.6.19: resolved Issue 1325: **MergeGraphicsDataTriangleList** (fix)

- description: merging of edges erroneous or problems with GraphicsDataCylinder
- date resolved: **2023-03-27 00:21**, date raised: 2022-12-19

Version 1.6.18: resolved Issue 1472: **GraphicsData** (extension)

- description: add addEdges option to all GraphicsData Python functions
- **notes: added to sphere, cylinder, SolidOfRevolution, SolidOfExtrusion**
- date resolved: **2023-03-27 00:10**, date raised: 2023-03-26

Version 1.6.17: resolved Issue 1473: **GraphicsData** (change)

- description: improve edges of cylinder and sphere by adding 6 lines along cylinder, some circles for sphere
- date resolved: **2023-03-27 00:09**, date raised: 2023-03-26

Version 1.6.16: resolved Issue 1459: **mention papers** (docu)

- description: add list of papers where exudyn has been used in theDoc and RTD
- date resolved: **2023-03-26 15:04**, date raised: 2023-03-10

Version 1.6.15: resolved Issue 1471: **issues tracker** (docu)

- description: remove html file from docs, use RSTfiles instead / move to readthedocs
- date resolved: **2023-03-24 20:40**, date raised: 2023-03-24

Version 1.6.14: resolved Issue 1470: **GraphicsDataBasis** (extension)

- description: add orientation and duplicate function with homogeneous transformation (HT) argument
- **notes: function called GraphicsDataFrame for Homogeneous transformation**
- date resolved: **2023-03-24 10:04**, date raised: 2023-03-24

Version 1.6.13: resolved Issue 1467: **InverseKinematicsNumerical** (fix)

- description: fix success
- date resolved: **2023-03-22 12:06**, date raised: 2023-03-22 (resolved by: P. Manzl)

Version 1.6.12: resolved Issue 1466: **modKKDH** (change)

- description: consistently renamed into modDHKK in robotics module
- date resolved: **2023-03-22 11:04**, date raised: 2023-03-22

Version 1.6.11: resolved Issue 1465: **robotics.models** (change)

- description: adjust robot definitions, add dhMode to dictionary; fix LinkList2Robot
- date resolved: **2023-03-22 10:39**, date raised: 2023-03-22

Version 1.6.10: resolved Issue 1464: **artificialIntelligence** (fix)

- description: adapt to stable-baselines3 > 1.5.0; Class OpenAIGymInterfaceEnv does not support stable-baselines3 > 1.5.0 because OpenAIGymInterfaceEnv is not inherited from gym.Env
- date resolved: **2023-03-20 12:54**, date raised: 2023-03-20 (resolved by: P. Manzl)

Version 1.6.9: resolved Issue 1463: **mpi4py** (extension)

- description: add option to use MPI (message passing interface) for ProcessParameterList
- **notes: tests on supercomputer successful**
- date resolved: **2023-03-18 22:59**, date raised: 2023-03-18

Version 1.6.8: resolved Issue 1462: **minimum coordinates** (docu)

- description: change consistently to minimal coordinates
- date resolved: **2023-03-14 17:39**, date raised: 2023-03-14

Version 1.6.7: resolved BUG 1461: **isIntType check ParameterVariation**

- description: In the ParameterVariation (part of the processing module) the integer type check failed when Variables of different types were used, now checking each variable independently
- **notes: added isIntType arrays**
- date resolved: **2023-03-14 12:50**, date raised: 2023-03-14 (resolved by: P. Manzl)

Version 1.6.6: resolved Issue 1457: **Newton** (docu)

- description: fix steps and Newton iterations in description of Newton settings (thanks to Martin Arnold!)
- date resolved: **2023-03-12 23:05**, date raised: 2023-03-09

Version 1.6.5: resolved Issue 1456: **sphinx/github pages** (docu)

- description: add examples and test models for better search results
- date resolved: **2023-03-12 23:05**, date raised: 2023-03-09

Version 1.6.4: resolved Issue 1445: **sphinx/github pages** (docu)

- description: add solver description
- date resolved: **2023-03-12 23:05**, date raised: 2023-02-22

Version 1.6.3: resolved Issue 1443: **sphinx/github pages** (docu)

- description: add theory parts as far as possible
- date resolved: **2023-03-12 23:05**, date raised: 2023-02-22

Version 1.6.2: resolved Issue 1460: **theDoc** (docu)

- description: change structure: theory, notations earlier; remove duplicated MatrixContainer description
- date resolved: **2023-03-12 16:20**, date raised: 2023-03-12

Version 1.6.1: resolved Issue 1458: **artificialIntelligence nan test** (extension)

- description: check for nan values in TestModel evaluation
- **notes: failed steps may lead to nan values when evaluating the TestModel in the artificialIntelligence module. When this is detected the flagNan is set and evaluation is stopped. Check for flagNan in the Evaluation should be implemented.**
- date resolved: **2023-03-10 15:59**, date raised: 2023-03-10 (resolved by: P. Manzl)

Version 1.6.0: resolved Issue 1426: **cleanup howto files** (fix)

- description: check if info is still valid (NGsolve, WSL, anaconda, ...)
- **notes: removed outdated files**
- date resolved: **2023-03-08 16:58**, date raised: 2023-02-11

Version 1.5.118: resolved Issue 1454: **julia** (docu)

- description: add sub section on interoperability with julia in Overview on Exudyn / Advanced topics; add some relevant examples for usage
- date resolved: **2023-03-05 16:40**, date raised: 2023-03-05

Version 1.5.117: resolved Issue 1453: **__NOGLFW option not working** (fix)

- description: exclude according functions in rendererPythonInterface
- **notes: added NOGLFW version to automatic build and testsuite**
- date resolved: **2023-02-28 15:18**, date raised: 2023-02-28

Version 1.5.116: resolved Issue 1452: **Single command** (docu)

- description: add some more detailed description in theDoc after Visualization settings dialog
- date resolved: **2023-02-26 00:40**, date raised: 2023-02-26

Version 1.5.115: resolved Issue 1451: **solver interface** (check)

- description: check modification of solver structures like it to be writable
- **notes: added now write functionality for solvers; use with care and only if you know what you do...**
- date resolved: **2023-02-26 00:12**, date raised: 2023-02-25

Version 1.5.114: resolved Issue 1447: **CSensorMarker** (extension)

- description: add RotationMatrix as additional outputvariable type
- **notes: Coordinates also added as output variable**
- date resolved: **2023-02-24 15:46**, date raised: 2023-02-23

Version 1.5.113: resolved Issue 1446: **CSensorMarker** (fix)

- description: does not check types; Make GetOutputVariableTypes as in objects; orientation only for selected markers; check types in Assemble prechecks
- date resolved: **2023-02-24 15:46**, date raised: 2023-02-23

Version 1.5.112: **resolved BUG 1449**: **DOPRI5**

- description: small bug introduced in previous update: currentStepSize not updated any more
- date resolved: **2023-02-24 15:45**, date raised: 2023-02-24

Version 1.5.111: resolved Issue 1448: **GenericJoint** (extension)

- description: add experimental flag alternativeConstraints to switch to different constraint equations, e.g. for 3 constrained rotations; this resolves unphysical 180 degree flips especially in static cases; flag may be removed in future
- **notes: thanks for P. Manzl for raising this problem**
- date resolved: **2023-02-24 13:52**, date raised: 2023-02-24

Version 1.5.110: resolved Issue 1444: **sphinx/github pages** (docu)

- description: fix equation references and figures
- date resolved: **2023-02-24 13:50**, date raised: 2023-02-22

Version 1.5.109: resolved Issue 1442: **add readthedocs.io** (docu)

- description: link github project to readthedocs.io site
- date resolved: **2023-02-22 23:59**, date raised: 2023-02-22

Version 1.5.108: resolved Issue 1441: **sphinx/github pages** (docu)

- description: add more parts on items (equations)
- date resolved: **2023-02-22 23:57**, date raised: 2023-02-22

Version 1.5.107: resolved Issue 1440: **ParameterVariation** (change)

- description: integer should be kept as integers, if they are provided as a list and if variation is only on integers
- **notes: integers are kept if either tuple(start, end, numberOfValues) with start/end and numberOfValues are all integer or list contains only integers**
- date resolved: **2023-02-22 13:46**, date raised: 2023-02-22 (resolved by: S. Holzinger)

Version 1.5.106: resolved Issue 1439: **numpy dependency** (extension)

- description: add install_requires to setup.py to force numpy to be installed; exudynCPP always requires now numpy
- date resolved: **2023-02-21 18:11**, date raised: 2023-02-21

Version 1.5.105: resolved Issue 1438: **sphinx / github pages** (docu)

- description: add input/output tables for items in RST files
- date resolved: **2023-02-21 10:14**, date raised: 2023-02-19

Version 1.5.104: resolved Issue 1437: **sphinx / github pages** (docu)

- description: add items to RST files/Sphinx to show up on github pages
- date resolved: **2023-02-19 21:39**, date raised: 2023-02-19

Version 1.5.103: resolved Issue 1436: **sphinx / github pages** (docu)

- description: add settings and structures; fix latex parts; fix links
- date resolved: **2023-02-18 00:56**, date raised: 2023-02-18

Version 1.5.102: **resolved BUG 1433**: **Lie group integration**

- description: system with > 1 node raises LinkedDataVectorBases exception
- date resolved: **2023-02-16 16:00**, date raised: 2023-02-16

Version 1.5.101: resolved Issue 1432: **artificialIntelligence module** (fix)

- description: fix np.float64 not beeing detected as a scalar (float) for initializationValues of RL environment and possible waiting without activated renderer
- date resolved: **2023-02-16 15:58**, date raised: 2023-02-16 (resolved by: P. Manzl)

Version 1.5.100: resolved Issue 1431: **make github pages** (extension)

- description: install workflow for github pages to host pages for exudyn
- **notes: due to automatic conversion, there are many small errors (typos) remaining; for safety, check theDoc.pdf in any case**
- date resolved: **2023-02-16 00:01**, date raised: 2023-02-16

Version 1.5.99: resolved Issue 1430: **create rst docu files** (extension)

- description: Create rst (markup language) files for C++ command interface and Python utilities
- date resolved: **2023-02-16 00:00**, date raised: 2023-02-15

Version 1.5.98: resolved Issue 1429: **revise docu creation** (fix)

- description: revise several helper functions for docu creation to homogenize latex and rst output
- date resolved: **2023-02-16 00:00**, date raised: 2023-02-15

Version 1.5.97: resolved Issue 1428: **ParameterVariation** (change)

- description: add conversion of parameters which are numpy.float64 to float which simplifies checking agains type(float); computationIndex becomes int
- date resolved: **2023-02-13 17:18**, date raised: 2023-02-13

Version 1.5.96: resolved Issue 1427: **IsReal(x) and IsInteger(x)** (extension)

- description: add checks which allow to check versus any float / numpy.float resp. int / numpy.int values; added to advancedUtilities
- date resolved: **2023-02-13 17:17**, date raised: 2023-02-13

Version 1.5.95: resolved Issue 1425: **sphinx** (extension)

- description: add github pages at https://jgerstmayr.github.io/EXUDYN created with sphinx
- date resolved: **2023-02-11 16:41**, date raised: 2023-02-11

Version 1.5.94: resolved Issue 1423: **numerical Jacobians** (change)

- description: add variadic template to realize numerical differentiation for objects in a consistent way
- date resolved: **2023-02-08 12:17**, date raised: 2023-02-08

Version 1.5.93: resolved Issue 1404: **Lie group method** (fix)

- description: add consistent numerical derivatives for Lie group nodes, needing the composition rule for incremental changes
- date resolved: **2023-02-08 12:17**, date raised: 2023-01-18

Version 1.5.92: resolved Issue 1422: **numerical Jacobians** (change)

- description: add variadic template to realize numerical differentiation in a consistent way
- date resolved: **2023-02-08 00:16**, date raised: 2023-02-07

Version 1.5.91: resolved Issue 1421: **LIE_GROUP_IMPLICIT_SOLVER** (change)

- description: remove preprocessor flag, as it is always set
- date resolved: **2023-02-07 12:07**, date raised: 2023-02-07

Version 1.5.90: resolved Issue 1420: **setup.py** (extension)

- description: reduce output of all compiler options with –quiet mode
- date resolved: **2023-02-02 18:27**, date raised: 2023-02-02

Version 1.5.89: resolved Issue 1419: **window size** (extension)

- description: window size currently limited to screen size and larger windows not accepted; override size limitations by using glfwSetWindowSize
- **notes: added flag window.limitWindowToScreenSize ; by default this is deactivated; added flag window.limitWindowToScreenSize ; by default this is deactivated; added flag window.limitWindowToScreenSize ; by default this is deactivated; added flag window.limitWindowToScreenSize ; by default this is deactivated; added flag window.limitWindowToScreenSize ; by default this is deactivated**
- date resolved: **2023-02-02 10:31**, date raised: 2023-02-02

Version 1.5.88: resolved Issue 1418: **OpenVR** (change)

- description: adapt projection for OpenVR compatibility; exchange multiplication of pose and eye to fulfill classic OpenGL needs
- date resolved: **2023-02-01 10:30**, date raised: 2023-02-01

Version 1.5.87: resolved Issue 1417: **ConnectorSpringDamperExt** (fix)

- description: remove unintended output during Assemble()
- date resolved: **2023-01-25 20:09**, date raised: 2023-01-25

Version 1.5.86: resolved Issue 1405: **AddDistanceSensor** (extension)

- description: add rotation of Marker to ObjectGround in SensorUserFunction
- **notes: added option to draw displaced laser beam; rotation added, if rigid marker used**
- date resolved: **2023-01-23 10:05**, date raised: 2023-01-19

Version 1.5.85: resolved Issue 1416: **AddDistanceSensor** (fix)

- description: UFsensorDistance misses the rotation part in visualization of laser beam with ground object
- date resolved: **2023-01-23 09:39**, date raised: 2023-01-23

Version 1.5.84: resolved Issue 1413: **CoordinateSpringDamperExt** (change)

- description: change friction parameter dimensions to forces (because there is no normal force) and change parameter names
- date resolved: **2023-01-22 23:43**, date raised: 2023-01-21

Version 1.5.83: resolved Issue 1412: **CoordinateSpringDamperExt** (extension)

- description: finalize implementation for bristle friction model
- date resolved: **2023-01-22 23:43**, date raised: 2023-01-21

Version 1.5.82: resolved Issue 1411: **CoordinateSpringDamperExt** (extension)

- description: finalize implementation for limit stops
- date resolved: **2023-01-22 23:43**, date raised: 2023-01-21

Version 1.5.81: resolved Issue 1410: **examples** (change)

- description: adapt Examples/massSpringFrictionInteractive.py and Examples/lugreFrictionTest.py to new CoordinateSpringDamperExt
- date resolved: **2023-01-21 22:04**, date raised: 2023-01-21

Version 1.5.80: resolved Issue 1136: **ConnectorsExt** (extension)

- description: add extended Ext versions of connectors: CoordinateSpringDamperExt, TorsionalSpringDamperExt, LinearSpringDamperExt, which allow for friction, coordinate limitation (with other SD-values) and possibly future extensions
- date resolved: **2023-01-21 21:29**, date raised: 2022-06-10

Version 1.5.79: resolved Issue 1407: **pause** (extension)

- description: add option for pause by pressing space bar
- date resolved: **2023-01-21 21:28**, date raised: 2023-01-21

Version 1.5.78: resolved Issue 1409: **CoordinateSpringDamper** (change)

- description: remove dryFriction and dryFrictionProportionalZone; this functionality will be made available in CoordinateSpringDamperExt
- **notes: see CoordinateSpringDamper in theDoc.pdf how to convert old models using friction parameters to new ones; note user function interfaces have been changed!**
- date resolved: **2023-01-21 17:53**, date raised: 2023-01-21

Version 1.5.77: resolved Issue 1408: **WaitForUserToContinue** (extension)

- description: add argument printMessage, which can be set false to avoid text output in console; default behavior preserved
- date resolved: **2023-01-21 13:51**, date raised: 2023-01-21

Version 1.5.76: resolved Issue 1406: **git tags** (extension)

- description: add git tags for every new version automatically
- **notes: now versions can be found easier in github and they match the version numbers in pypi with pip installer**
- date resolved: **2023-01-19 18:47**, date raised: 2023-01-19

Version 1.5.75: resolved Issue 1314: **Pybind11** (check)

- description: test compilation with higher version of pybind11 (currently pybind11=2.6.0 in setup.py
- **notes: already done earlier; works well and allows compilation with Python3.11; added some switches in setup.py as not all Pybind11 versions work everywhere**

- date resolved: **2023-01-19 01:04**, date raised: 2022-12-14

Version 1.5.74: resolved Issue 1365: **OpenVR** (example)
- description: add Python example with openVR
- date resolved: **2023-01-19 01:02**, date raised: 2023-01-03

Version 1.5.73: resolved Issue 1326: **OpenVR** (change)
- description: add OpenVR license and mention in getting started
- date resolved: **2023-01-19 01:02**, date raised: 2022-12-20

Version 1.5.72: resolved Issue 1402: **openVR** (extension)
- description: add flag to set compilation with openVR in setup.py; copy .dll in Windows case
- date resolved: **2023-01-18 22:57**, date raised: 2023-01-17

Version 1.5.71: resolved Issue 1364: **OpenVR** (extension)
- description: test simple use case
- **notes: basic functionality added; enable compilation with openVR by setting '-D__EXUDYN_USE_OPENVR' in cpp compiler flags of setup.py**
- date resolved: **2023-01-17 09:32**, date raised: 2023-01-03 (resolved by: EXTENSION)

Version 1.5.70: resolved Issue 1401: **lockModelView** (extension)
- description: add flag which allows to fully lock rotation, zoom, etc.; in this case, only initial values in openGL settings are accepted for setup of view, but mouse or key input is ignored
- date resolved: **2023-01-17 09:20**, date raised: 2023-01-17

Version 1.5.69: resolved Issue 1363: **test OpenVR with basic OpenGL** (check)
- description: check if current openGL is sufficient with textures
- date resolved: **2023-01-16 23:11**, date raised: 2023-01-03

Version 1.5.68: resolved Issue 1400: **renderState** (change)
- description: vectors and matrices are returned in numpy format; this allows simpler computation, but does not anymore allow to treat output as list (+ operator!)
- date resolved: **2023-01-16 21:11**, date raised: 2023-01-16

Version 1.5.67: resolved Issue 1399: **renderState** (change)
- description: initialization now done consistently for SC.AttachToRenderEngine() and exudyn.StartRenderer(); behavior should be as before
- date resolved: **2023-01-16 21:10**, date raised: 2023-01-16

Version 1.5.66: resolved Issue 1398: **renderState** (extension)
- description: add projectionMatrix containing the current projection usd (usually identity matrix)
- date resolved: **2023-01-16 20:09**, date raised: 2023-01-16

Version 1.5.65: resolved Issue 1397: **showFaceEdges, showMeshEdges** (fix)
- description: wrong switching causing to show edges only if also some faces are activated
- date resolved: **2023-01-12 22:32**, date raised: 2023-01-12

Version 1.5.64: resolved Issue 1387: **Get...Output** (extension)
- description: add way to work for Reference configuration
- **notes: nodes, bodies and markers allow reference configuration; sensors also allow it for GetSensorValues**
- date resolved: **2023-01-12 22:14**, date raised: 2023-01-12

Version 1.5.63: resolved Issue 1389: **configuration checks** (extension)
- description: check all systemData C++ interface functions regarding illegal configuration
- date resolved: **2023-01-12 22:03**, date raised: 2023-01-12

Version 1.5.62: resolved Issue 1388: **configuration checks** (extension)
- description: IsConfigurationInitialCurrentReferenceVisualization and IsConfigurationInitialCurrentVisualization shall be extended for StartOfStep; add hint that a calling function may have used an illegal configuration or None
- date resolved: **2023-01-12 22:03**, date raised: 2023-01-12

Version 1.5.61: resolved Issue 1396: **Demo** (extnesion)
- description: add a two demos included into the python module; put into exudyn.demos; add hint for larger examples; Demo1() = without graphics, just creating output file; Demo2() is rigid3Dexample with SolutionViewer
- date resolved: **2023-01-12 18:51**, date raised: 2023-01-12

Version 1.5.60: resolved Issue 1385: **help** (extension)
- description: add exudyn.help() function for short notes
- date resolved: **2023-01-12 18:04**, date raised: 2023-01-12

Version 1.5.59: **resolved BUG 1390**: **ComputeLinearizedSystem**
- description: not working because of wrong interface to ComputeJacobianODE2RHS
- date resolved: **2023-01-12 17:53**, date raised: 2023-01-12

Version 1.5.58: resolved Issue 1394: **eigenvalues test** (testing)

- description: refine test model for ComputeODE2Eigenvalues
- date resolved: **2023-01-12 17:44**, date raised: 2023-01-12

Version 1.5.57: resolved Issue 1393: **ComputeJacobianAE** (change)

- description: change default values to fit to conventional tangential stiffness matrix computation
- date resolved: **2023-01-12 17:00**, date raised: 2023-01-12

Version 1.5.56: resolved Issue 1392: **ComputeJacobianODE1RHS** (change)

- description: change default values to fit to conventional tangential stiffness matrix computation
- date resolved: **2023-01-12 17:00**, date raised: 2023-01-12

Version 1.5.55: resolved Issue 1391: **ComputeJacobianODE2RHS** (change)

- description: change default values to fit to conventional tangential stiffness matrix computation
- date resolved: **2023-01-12 17:00**, date raised: 2023-01-12

Version 1.5.54: **resolved BUG 1386**: **GetMarkerOutput**

- description: crashes if used with any configuration before Assemble(); add check that it may be only called after Assemble()
- **notes: add check and raise error**
- date resolved: **2023-01-12 14:15**, date raised: 2023-01-12

Version 1.5.53: resolved Issue 1384: **visualizationSettings** (extension)

- description: add textOffsetFactor in general options to adjust text offset if not drawn always in front
- **notes: also adjusted some text appearance settings**
- date resolved: **2023-01-11 20:42**, date raised: 2023-01-11

Version 1.5.52: resolved Issue 1383: **visualizationSettings** (docu)

- description: update docu of Visualization settings dialog in introduction
- date resolved: **2023-01-11 17:41**, date raised: 2023-01-11

Version 1.5.51: resolved Issue 1382: **visualizationSettings** (change)

- description: increase initial size of visualizations dialog for larger screens; see Section 2.4.5 how to change to a smaller window size
- date resolved: **2023-01-11 16:57**, date raised: 2023-01-11

Version 1.5.50: resolved Issue 1381: **visualizationSettings** (change)

- description: change order of items to have easier access to each tree node
- **notes: dialogs.openTreeView=True can be used to switch to original behavior**
- date resolved: **2023-01-11 16:34**, date raised: 2023-01-11

Version 1.5.49: resolved Issue 1379: **item text color** (change)

- description: draw colors in item texts (node numbers, etc.) different from item color to improve visibility
- date resolved: **2023-01-11 16:28**, date raised: 2023-01-11

Version 1.5.48: resolved Issue 1308: **OpenGL texts** (extension)

- description: add sub function for drawing text bitmaps; use different order of drawing for transparent and non-transparent triangles to get improved text drawing
- **notes: now having possibility to draw item texts in front or transparent**
- date resolved: **2023-01-11 16:28**, date raised: 2022-12-08

Version 1.5.47: resolved Issue 1380: **loads** (change)

- description: draw load numbers at end of load vectors
- **notes: made Force consistent with Torque and LoadMassProportional**
- date resolved: **2023-01-11 14:43**, date raised: 2023-01-11

Version 1.5.46: resolved Issue 1378: **font drawing** (extension)

- description: add visualizationSettings.general for textDrawInFront and textHasBackGround for having a (currently) white background color
- date resolved: **2023-01-11 09:46**, date raised: 2023-01-11

Version 1.5.45: resolved Issue 1377: **openvr** (change)

- description: not compatible right now with Exudyn under Windows while linux seems to work
- date resolved: **2023-01-09 21:11**, date raised: 2023-01-09

Version 1.5.44: resolved Issue 1370: **advancedUtilities** (extension)

- description: add advanced utilities depnding on numpy and math; functions not suitable for exudyn.basicUtilities or exudyn.utilities
- date resolved: **2023-01-07 01:57**, date raised: 2023-01-06

Version 1.5.43: resolved Issue 1375: **initial accelerations** (extension)

- description: add flag to decide whether initial accelerations are erased at beginning of simulation; the background is that they should not be erased, if they are prolonged for subsequent simulation; check if this changes behavior, as ODE2_tt coordinates are anyway resetted at Assemble()
- date resolved: **2023-01-07 01:56**, date raised: 2023-01-06

Version 1.5.42: resolved Issue 1376: **advancedUtilities** (change)

- description: move functions exudyn.utilities from to exudyn.advancedUtilities; still included in exudyn.utilities
- date resolved: **2023-01-06 22:54**, date raised: 2023-01-06

Version 1.5.41: **resolved BUG 1374**: **InteractiveDialog**

- description: accelerations are not reused from last period; need to copy current accelerations into initial accelerations
- date resolved: **2023-01-06 21:21**, date raised: 2023-01-06

Version 1.5.40: resolved Issue 1373: **GraphicsData** (change)

- description: add option to add edges in GraphicsData...Cube... functions; switch some order of options
- date resolved: **2023-01-06 19:25**, date raised: 2023-01-06

Version 1.5.39: resolved Issue 1372: **utilities** (fix)

- description: ComputeSkewMatrix: remove duplicate (with less functionality) from utilities and put into rigidBodyUtilities; correct import in FEM
- date resolved: **2023-01-06 19:19**, date raised: 2023-01-06

Version 1.5.38: resolved Issue 1371: **exudyn.utilities** (change)

- description: remove functions CheckInputVector and CheckInputIndexArray as they are unused and replaced with improved functions in advancedUtilities
- date resolved: **2023-01-06 17:46**, date raised: 2023-01-06

Version 1.5.37: resolved Issue 1369: **add sub-module machines** (extension)

- description: will include mechanical engineering and machine element relevant topics, such as bearings, gears, mechanisms, etc.
- date resolved: **2023-01-06 11:47**, date raised: 2023-01-06

Version 1.5.36: resolved Issue 1368: **GraphicsDataCylinder** (extension)

- description: new option to only add edges without faces
- date resolved: **2023-01-05 23:03**, date raised: 2023-01-05

Version 1.5.35: resolved Issue 1367: **GraphicsDataSolidExtrusion** (extension)

- description: add edges and normals for smoothening
- date resolved: **2023-01-05 23:03**, date raised: 2023-01-05

Version 1.5.34: resolved Issue 1366: **GraphicsData** (extension)

- description: add functionality for GraphicsDataSolidExtrusion to include circles: CirclePointsAndSegments() and to manipulate point lists defining geometries: SegmentsFromPoints()
- date resolved: **2023-01-05 18:15**, date raised: 2023-01-05

Version 1.5.33: resolved Issue 1362: **add OpenVR interface** (extension)

- description: add interface, but by default not compiled
- date resolved: **2023-01-03 23:00**, date raised: 2023-01-03

Version 1.5.32: resolved Issue 1361: **Python 3.11** (extension)

- description: create first development wheels for Python 3.11; Windows+Linux
- **notes: problems to get conda with Python3.11 running, especially on ubuntu**
- date resolved: **2023-01-03 18:48**, date raised: 2023-01-03

Version 1.5.31: resolved Issue 1360: **Python 3.11** (extension)

- description: adjust setup.py to support Python 3.11; requires Pybind11 2.10
- date resolved: **2023-01-03 14:37**, date raised: 2023-01-03

Version 1.5.30: resolved Issue 1335: **RollingDiscPenalty** (extension)

- description: add test example for ground moving (rotating table
- date resolved: **2023-01-02 19:07**, date raised: 2022-12-25

Version 1.5.29: resolved Issue 1359: **ObjectGround** (extension)

- description: extend for referenceRotation to allow rotation of ground objects (especially for visualization and contact)
- date resolved: **2023-01-02 11:49**, date raised: 2023-01-02

Version 1.5.28: resolved Issue 0839: **multithreaded jacobian** (extension)

- description: add functionality for multithreaded jacobian and mass matrix
- **notes: mass matrix resolved; other is redundant with #1203**
- date resolved: **2023-01-02 01:04**, date raised: 2021-12-19

Version 1.5.27: resolved Issue 0828: **MT integration2** (extension)

- description: add multithreading to mass matrix and jacobian; add multithreaded adding of vector with special templated functions to allow special (templated) solver functions
- **notes: redundant with #839**
- date resolved: **2023-01-02 01:02**, date raised: 2021-12-09

Version 1.5.26: resolved Issue 0791: **test Eigen SimplicialLDLT** (extension)

- description: use systemMatricesArePD to enable LDLT solver
- **notes: tested earlier; does not work in most cases**
- date resolved: **2023-01-02 01:01**, date raised: 2021-11-02

Version 1.5.25: resolved Issue 0437: **solvercontainer+ systemcontainer** (change)

- description: remove SolverContainer and SystemContainer from systemStructuresDefinition and work manually
- **notes: already done much earlier**
- date resolved: **2023-01-02 00:59**, date raised: 2020-07-21

Version 1.5.24: resolved Issue 0315: **Add user object** (extension)

- description: add user object
- **notes: done with GenericODE2 and CoordinateVectorConstraint**
- date resolved: **2023-01-02 00:57**, date raised: 2020-01-10

Version 1.5.23: resolved Issue 0261: **MarkerDataJacobians** (extension)

- description: Add access functions for jacobians and other marker data: SetPositionJacobian, GetPositionJacobian, etc.; add jacobian types = markertypes, which check if wrong jacobian is accessed
- **notes: not suitable anymore**
- date resolved: **2023-01-02 00:56**, date raised: 2019-09-11

Version 1.5.22: resolved Issue 1331: **exudyn cpp** (extension)

- description: add __repr__ and help which writes some information on workflow (github, theDoc, Examples, ...)
- **notes: not possible**
- date resolved: **2023-01-02 00:51**, date raised: 2022-12-21

Version 1.5.21: resolved Issue 1323: **AVX2** (extension)

- description: update documentation for improved functionality with AVX2 code
- **notes: already done when resolving #1330**
- date resolved: **2023-01-02 00:50**, date raised: 2022-12-17

Version 1.5.20: resolved Issue 1355: **visualizationSettings** (check)

- description: check possibility for update loop to continue simulation
- **notes: not feasible now, as it requires to modify time integration loop**
- date resolved: **2023-01-02 00:48**, date raised: 2022-12-31

Version 1.5.19: resolved Issue 1353: **visualizationSettings** (extension)

- description: add KEY Q shortcut to close dialog
- date resolved: **2023-01-02 00:48**, date raised: 2022-12-31

Version 1.5.18: resolved Issue 1358: **DictionariesGetSet** (change)

- description: add new types VectorFloat and MatrixFloat in order to distinguish from double values
- **notes: also fixed matrix conversion in visualizationSettings**
- date resolved: **2023-01-02 00:47**, date raised: 2023-01-02

Version 1.5.17: resolved Issue 1357: **visualizationSettings** (extension)

- description: add single click edit event
- date resolved: **2023-01-01 23:27**, date raised: 2023-01-01

Version 1.5.16: resolved Issue 1354: **visualizationSettings** (extension)

- description: double click on bool variables changes state
- date resolved: **2023-01-01 23:22**, date raised: 2022-12-31

Version 1.5.15: resolved Issue 1352: **tkinter dialogs** (fix)

- description: add option for transparency to dialogs
- **notes: visualizationSettings.dialogs.transparency**
- date resolved: **2022-12-31 11:01**, date raised: 2022-12-31

Version 1.5.14: resolved Issue 1351: **tkinter MacOS** (fix)

- description: adjust font size and row size in ttk right mouse dialog
- **notes: visualizationSettings.dialogs.fontScalingMacOS**
- date resolved: **2022-12-31 11:00**, date raised: 2022-12-31

Version 1.5.13: resolved Issue 1350: **tkinter MacOS** (fix)

- description: adjust font size and row size in ttk visualizationSettings dialog
- **notes: visualizationSettings.dialogs.fontScalingMacOS**
- date resolved: **2022-12-31 11:00**, date raised: 2022-12-31

Version 1.5.12: resolved BUG 0752: **tkinter MacOS**

- description: tkinter fails when loaded inside interactive.py; early call to Tk() inside an example works and seems to help; check options to correctly load tkinter in MacOS (Rosetta 2, on M1)
- **notes: problem fixed with single threaded renderer, doing illegal operations in glfw callbacks**
- date resolved: **2022-12-31 01:01**, date raised: 2021-09-20

Version 1.5.11: resolved Issue 1349: **RequireVersion** (fix)

- description: some bugs including exu.GetVersionString and not raising exception
- date resolved: **2022-12-30 15:03**, date raised: 2022-12-30

Version 1.5.10: resolved Issue 1343: **tkinter in exudyn** (docu)

- description: add comment in FAQ on potential problems with tkinter; add option to let exudyn know if tkinter is already running
- date resolved: **2022-12-28 23:39**, date raised: 2022-12-27

Version 1.5.9: resolved Issue 1347: **visualizationSettings** (extension)

- description: add flag visualizationSettings.dialog.multiThreadedDialogs to turn on/off immediate apply of visualizationSettings changes; extend exudyn.GUI and rendererPythonInterface.h accordingly
- **notes: NOTE that this flag should be turned off in case of crashes during/after dialogs; could make problems on special platforms such as MacOS**
- date resolved: **2022-12-28 23:05**, date raised: 2022-12-28

Version 1.5.8: resolved Issue 1348: **EditDictionaryWithTypeInfo** (change)

- description: change interface to pass directly visualizationSettings, allowing to update data
- date resolved: **2022-12-28 21:50**, date raised: 2022-12-28

Version 1.5.7: resolved Issue 1346: **visualizationSettings** (check)

- description: check if visualiuation settings dialog can be installed such that updating of data immediately affects renderer window
- date resolved: **2022-12-28 16:53**, date raised: 2022-12-28

Version 1.5.6: resolved Issue 1339: **tkinter MacOS** (fix)

- description: add tkinter.Tk() into exudyn.sys["tkinterRoot"] before call to StartRenderer(); check for tkinterRoot in exudyn.sys on startup of interactive dialog; resolves crash on MacOS for SolutionViewer
- date resolved: **2022-12-27 23:50**, date raised: 2022-12-26

Version 1.5.5: resolved Issue 1345: **interface default args** (fix)

- description: change true/false to True/False in theDoc for PYthon command interface
- date resolved: **2022-12-27 23:24**, date raised: 2022-12-27

Version 1.5.4: resolved Issue 1344: **exudyn** (extension)

- description: add function IsRendererRunning(), to avoid Warnings when renderer is restarted
- date resolved: **2022-12-27 23:23**, date raised: 2022-12-27

Version 1.5.3: resolved Issue 1338: **MacOS** (fix)

- description: DoRendererIdleTasks() crashes if no Renderer is active; add check in python call to avoid crash
- **notes: crash resolved on Windows; MacOS test still open**
- date resolved: **2022-12-27 17:36**, date raised: 2022-12-26

Version 1.5.2: resolved Issue 1340: **MacOS multithreading** (change)

- description: activate simplified multithreading by switching to TinyThreading in case of MacOS
- date resolved: **2022-12-27 17:19**, date raised: 2022-12-26

Version 1.5.1: resolved Issue 1342: **MacOS M1** (extension)

- description: add string ARM to Platform string, e.g., used in solution and sensor files
- date resolved: **2022-12-27 17:03**, date raised: 2022-12-27

Version 1.5.0: resolved Issue 1336: **test suite** (fix)

- description: resolve linux problem with RigidBodySpringDamper.py MiniExample
- **notes: problem in testsuite due to AVX differences, causing all non-AVX cases to fail (also linux); fixed with special case in reference solutions**
- date resolved: **2022-12-25 20:10**, date raised: 2022-12-25

Version 1.4.65: resolved Issue 1264: **RollingDiscPenalty** (extension)

- description: correct torque on ground, which currently has no effect, but will be used for moving ground
- **notes: resolved earlier**
- date resolved: **2022-12-25 12:15**, date raised: 2022-09-17

Version 1.4.64: resolved Issue 1298: **ReevingSystemLinear** (testing)

- description: add test model to test suite
- date resolved: **2022-12-25 12:11**, date raised: 2022-12-01

Version 1.4.63: resolved Issue 1333: **gcc linux** (change)

- description: adjust -march compilation options for improved performance on linux; try -march=skylake for AVX2 optimization
- **notes: did not work: either gives compilation errors or segmentation faults**
- date resolved: **2022-12-25 00:45**, date raised: 2022-12-24

Version 1.4.62: resolved Issue 1334: **NOGLFW** (change)

- description: remove SystemContainer constructor warning for AttachToRenderEngine in case of NOGLFW
- **notes: also done for DetachFromRenderEngine**
- date resolved: **2022-12-25 00:06**, date raised: 2022-12-25

Version 1.4.61: resolved Issue 1332: **MacOS** (fix)

- description: compilation does not finish due to -framework Cocoa, etc. errors
- **notes: changed -framework library lists by separating commands into two separate strings; compilation now runs through for Python 3.8-3.10 on MacOS with M1 and 3.7.-3.10 on _x86 emulation**
- date resolved: **2022-12-24 22:20**, date raised: 2022-12-24

Version 1.4.60: resolved Issue 1330: **AVX2** (fix)

- description: fix check for AVX and AVX2 on module import
- date resolved: **2022-12-21 19:33**, date raised: 2022-12-21

Version 1.4.59: resolved Issue 1329: **SolutionViewer** (example)

- description: add example for SolutionViewer with multiple static simulations performed, writing results into single file; see Examples/solutionViewerMultipleSimulations.py
- date resolved: **2022-12-21 18:05**, date raised: 2022-12-21

Version 1.4.58: resolved Issue 1328: **simulationSettings.solutionSettings** (extension)

- description: add option writeInitialValues in order to turn on/off writing of initial values for coordinatesSolution and sensors; by default turned on as was done so far
- date resolved: **2022-12-21 14:07**, date raised: 2022-12-21

Version 1.4.57: resolved Issue 1037: **sensor files** (extension)

- description: add file footer information same as in coordinatesSolutionFile including CPUtimeElapsed with 3 digits
- **notes: turned on by default, but should be switched off if it causes compatibility problems for your postprocessing tool**
- date resolved: **2022-12-21 12:09**, date raised: 2022-04-07

Version 1.4.56: resolved Issue 1327: **sensorsWriteFileHeader** (fix)

- description: not working
- **notes: added option into solver, now turning on/off as desired; added option into solver, now turning on/off as desired; added option into solver, now turning on/off as desired**
- date resolved: **2022-12-21 12:06**, date raised: 2022-12-21

Version 1.4.55: resolved Issue 1317: **DistanceSensor** (example)

- description: Add example with distance sensor
- date resolved: **2022-12-20 13:49**, date raised: 2022-12-16

Version 1.4.54: resolved Issue 1318: **DistanceSensor** (extension)

- description: Add measurement for GeneralContact trigsRigidBody
- date resolved: **2022-12-19 21:49**, date raised: 2022-12-16

Version 1.4.53: resolved Issue 1316: **DistanceSensor** (extension)

- description: add utilities function to create DistanceSensor based on general contact
- **notes: added AddDistanceSensor to exudyn utilities.py**
- date resolved: **2022-12-19 01:09**, date raised: 2022-12-16

Version 1.4.52: resolved Issue 1324: **DistanceSensor** (extension)

- description: utilities function AddDistanceSensor extended for measureVelocity to measure velocity similar as in a laser Doppler vibrometer (LDV)
- date resolved: **2022-12-18 20:59**, date raised: 2022-12-18

Version 1.4.51: resolved Issue 1320: **GeneralContact** (extension)

- description: add binary contact types to pybind interface; add conversion of binary types to type indices
- **notes: only TypeIndex added which is sufficient for DistanceSensor**
- date resolved: **2022-12-17 00:32**, date raised: 2022-12-16

Version 1.4.50: resolved Issue 1319: **DistanceSensor** (extension)

- description: Add radius for option to measure with cylinder with given radius instead of line; useful for particles
- date resolved: **2022-12-17 00:32**, date raised: 2022-12-16

Version 1.4.49: resolved Issue 1321: **DistanceSensor** (extension)

- description: Add option to select which contact types are considered
- date resolved: **2022-12-17 00:31**, date raised: 2022-12-16

Version 1.4.48: resolved Issue 1322: **AVX2 import** (extension)

- description: add checks based on numpy.core._multiarray_umath to find if CPU has AVX2 support; in release, user may directly import noAVX version by setting sys.exudynCPUhasAVX2=False
- date resolved: **2022-12-16 23:42**, date raised: 2022-12-16

Version 1.4.47: resolved BUG 1315: **SetSearchTreeBox**

- description: has no effect, and searchTree is always computed automatically
- **notes: now search tree can be initialized smaller or larger than initial geometry in order to optimize for specific problem**
- date resolved: **2022-12-15 21:20**, date raised: 2022-12-15

Version 1.4.46: resolved Issue 1268: **DistanceSensor** (extension)

- description: add simple distance sensor
- **notes: can be realized with GeneralContact and SensorUserFunction**
- date resolved: **2022-12-15 20:23**, date raised: 2022-09-21

Version 1.4.45: resolved Issue 1271: **GeneralContact** (extension)

- description: Add interface functions for GeneralContact: allowing to measure distance along a line; get markers/spheres in box; get beams in box; etc.
- date resolved: **2022-12-15 20:20**, date raised: 2022-09-21

Version 1.4.44: resolved Issue 1313: **AVX2** (extension)

- description: add additional library without AVX and use try/except to import exudynCPPnoAVX in case that AVX2 is not available
- date resolved: **2022-12-14 20:27**, date raised: 2022-12-14

Version 1.4.43: resolved Issue 1312: **SolveDynamic** (extension)

- description: add flag computeMassMatrixInversePerBody for explicit time integration to compute mass matrix inverse per body; read theDoc and use with care!
- **notes: check your models! ComputeMassMatrix has been adapted for every body!!!**
- date resolved: **2022-12-13 21:00**, date raised: 2022-12-13

Version 1.4.42: resolved Issue 1311: **DynamicSolver** (fix)

- description: include flag timeIntegration.reuseConstantMassMatrix into explicit solver; up to now, this option was always turned on
- date resolved: **2022-12-13 18:51**, date raised: 2022-12-13

Version 1.4.41: resolved Issue 1310: **searchTreeUpdateCounter** (fix)

- description: needs reset to 0 after reaching limit
- date resolved: **2022-12-13 17:58**, date raised: 2022-12-13

Version 1.4.40: resolved Issue 0847: **GeneralContact searchtree** (extension)

- description: add options to recompute searchtree size if particles are moving out of region; add option to flush all dynamical arrays (searchtree, allActiveContacts, etc.) after certain time
- **notes: added option resetSearchTreeInterval into GeneralContact settings**
- date resolved: **2022-12-12 10:40**, date raised: 2021-12-23

Version 1.4.39: resolved Issue 1309: **rigidBodyUtilities** (extension)

- description: AddRigidBody: add default argument for nodeType as RotationEulerParameters; simplifies creation of rigid bodies
- date resolved: **2022-12-08 18:16**, date raised: 2022-12-08

Version 1.4.38: resolved Issue 1302: **GL_POLYGON_OFFSET_FILL** (fix)

- description: correct polygon offset setting for regular faces/lines and add adjustable parameter in visualizationSettings
- date resolved: **2022-12-08 01:50**, date raised: 2022-12-06

Version 1.4.37: resolved Issue 1307: **OpenGL** (change)

- description: change line and polygon drawing in order to avoid line artifacts; newly introduced polygon offset may cause problems: check your visualization, send reports if problems and set polygonOffset=0 in severe cases
- date resolved: **2022-12-08 01:06**, date raised: 2022-12-08

Version 1.4.36: resolved Issue 1304: **ConnectorRollingDiscPenalty** (docu)

- description: extend docu for arbitrary planeNormal and moving ground
- date resolved: **2022-12-07 21:57**, date raised: 2022-12-07

Version 1.4.35: resolved Issue 1306: **ObjectJointRollingDisc** (extension)

- description: add discAxis to object parameters, being able to change from default x-axis
- date resolved: **2022-12-07 20:18**, date raised: 2022-12-07

Version 1.4.34: resolved Issue 1305: **ObjectJointRollingDisc** (change)

- description: change computation of trail, generalized for two bodies moving relative to each other (needs further testing)
- date resolved: **2022-12-07 20:18**, date raised: 2022-12-07

Version 1.4.33: resolved Issue 1303: **ConnectorRollingDiscPenalty** (extension)

- description: extended formulation for arbitrary planeNormal; ground body can now also move in space (testing needed)

- date resolved: **2022-12-07 17:49**, date raised: 2022-12-07

Version 1.4.32: resolved Issue 1301: **graphicsDataUtilities** (fix)
- description: AddEdgesAndSmoothenNormals fixed to process colors
- date resolved: **2022-12-06 12:37**, date raised: 2022-12-06

Version 1.4.31: resolved Issue 1300: **graphicsDataUtilities** (extension)
- description: GraphicsDataFromPointsAndTrigs extended to accept color per point or 4 RGBA values
- date resolved: **2022-12-06 12:37**, date raised: 2022-12-06

Version 1.4.30: **resolved BUG 1299**: **GeneralContact**
- description: searchTreeBox not computed automatically (must be set manually)
- **notes: computation of searchtree performed automatically now if no search tree box is specified; output message written in case of verboseMode=1**
- date resolved: **2022-12-02 12:51**, date raised: 2022-12-02

Version 1.4.29: resolved Issue 1297: **ConnectorSpringDamper** (extension)
- description: return scalar spring-damper force with OutputVariableType ForceLocal
- date resolved: **2022-12-01 16:31**, date raised: 2022-12-01

Version 1.4.28: resolved Issue 1296: **dynamic solver** (check)
- description: add check that useIndex2=True is consistent with useNewmark=True
- date resolved: **2022-11-30 14:12**, date raised: 2022-11-29

Version 1.4.27: resolved Issue 0925: **ReevingSystemSprings** (extension)
- description: create reeving system along points defined by markers, using massless springs and one total length; add rigid body markers for position of sheaves; use coordinate markers for prescribed change of length at end of reeving system
- **notes: see new ObjectConnectorReevingSystemSprings**
- date resolved: **2022-11-19 01:11**, date raised: 2022-02-03

Version 1.4.26: resolved Issue 1295: **BeamGeometricallyExact2D** (extension)
- description: add damping terms for bending, axial and shear deformation
- date resolved: **2022-11-17 15:59**, date raised: 2022-11-17

Version 1.4.25: resolved Issue 1294: **BeamGeometricallyExact2D** (extension)
- description: add reference strain/curvature
- date resolved: **2022-11-17 15:59**, date raised: 2022-11-17

Version 1.4.24: resolved Issue 1293: **MarkerDataStructure** (extension)
- description: extend for more than 2 MarkerData; adjust caller functions
- date resolved: **2022-11-13 16:53**, date raised: 2022-11-13

Version 1.4.23: resolved Issue 1291: **Lie Group integration** (fix)
- description: CompositionRule in LieGroup nodes does not consider reference position; add reference configuration in composition rule (and subtract afterwards)
- date resolved: **2022-11-09 15:00**, date raised: 2022-11-09

Version 1.4.22: **resolved BUG 1289**: **visualizationSettings**
- description: interactive.trackMarker has wrong type leading to crash when closing visualizationSettings
- date resolved: **2022-11-05 14:15**, date raised: 2022-11-05

Version 1.4.21: resolved Issue 1286: **trackMarker** (fix)
- description: selection of objects wrong / check mouse selection procedure
- date resolved: **2022-11-04 20:46**, date raised: 2022-11-02

Version 1.4.20: resolved Issue 1288: **ConnectorRollingDiscPenalty** (extension)
- description: add useLinearProportionalZone which performs better in implicit time integration
- date resolved: **2022-11-04 17:16**, date raised: 2022-11-04

Version 1.4.19: resolved Issue 1287: **ConnectorRollingDiscPenalty** (extension)
- description: add viscousFriction, using separate values for local X/Y coordinates
- date resolved: **2022-11-04 17:16**, date raised: 2022-11-04

Version 1.4.18: resolved Issue 1276: **Renderer: marker tracking** (extension)
- description: add option to track markers in renderer
- **notes: see visualizationSettings.interactive for several trackMarker... options**
- date resolved: **2022-11-02 17:13**, date raised: 2022-10-12

Version 1.4.17: resolved Issue 1279: **release_assert** (change)
- description: remove all asserts (in automatic code generation)
- date resolved: **2022-11-02 17:10**, date raised: 2022-10-13

Version 1.4.16: resolved Issue 1285: **Lobatto, LobattoIntegrate** (fix)

- description: order nomenclature is wrong. Lobatto2 needs to be renamed into Lobatto1 and so on
- date resolved: **2022-11-02 16:43**, date raised: 2022-11-02

Version 1.4.15: resolved Issue 1284: **extend ALEANCF beam** (extension)

- description: add effects due to axial and bending viscous damping in case of movingMassFactor==1
- **notes: new damping terms for axially moving beams implemented according to 2022 Paper of Pieber, Ntarladima, Gerstmayr only for case movingMassFactor=1**
- date resolved: **2022-10-31 15:43**, date raised: 2022-10-25

Version 1.4.14: resolved BUG 1283: **ProcessParameterList**

- description: in case useMultiProcessing=False the output file does not contain correct ranges
- date resolved: **2022-10-20 21:59**, date raised: 2022-10-20

Version 1.4.13: resolved Issue 1282: **PlotSensor** (extension)

- description: added return value including [plt, fig, ax, line] to be used for subsequent operations
- date resolved: **2022-10-19 20:30**, date raised: 2022-10-19

Version 1.4.12: resolved Issue 1281: **results monitor** (extension)

- description: extend results monitor for viewing more detailed results of ParameterVariation with colorVariations option; add function SingleIndex2SubIndices
- date resolved: **2022-10-17 07:30**, date raised: 2022-10-17

Version 1.4.11: resolved BUG 1280: **ParameterVariation**

- description: resultsFile not working (problem with ProcessParameterList
- date resolved: **2022-10-14 08:22**, date raised: 2022-10-14

Version 1.4.10: resolved Issue 1278: **center point** (fix)

- description: add key O to change center of rotation; resolve finally original issue 1155
- date resolved: **2022-10-12 22:37**, date raised: 2022-10-12

Version 1.4.9: resolved Issue 1277: **shadowPolygonOffset** (change)

- description: decrease default value from 10 to 0.1; adjust in your models
- date resolved: **2022-10-12 20:27**, date raised: 2022-10-12

Version 1.4.8: resolved Issue 1275: **GetMarkerOutput** (extension)

- description: add function mbs.GetMarkerOutput() to return position, velocitiy, rotation matrix and angular velocity for markers if available
- date resolved: **2022-10-12 09:18**, date raised: 2022-10-12

Version 1.4.7: resolved BUG 1274: **ALECable2D**

- description: missing term L in preComputedB terms in C++ implementation
- date resolved: **2022-09-25 14:17**, date raised: 2022-09-25

Version 1.4.6: resolved Issue 1242: **Beam3D** (change)

- description: rename ANCFBeam3D to ANCFBeam and GeometricallyExactBeam3D in same way for consistency reasons
- date resolved: **2022-09-24 19:35**, date raised: 2022-08-24

Version 1.4.5: resolved Issue 1265: **RollingDisc** (fix)

- description: correct description of coordinate systems in RollingDisc and RollingDiscPenalty; remove wrong transposed sign
- date resolved: **2022-09-19 17:46**, date raised: 2022-09-19

Version 1.4.4: resolved Issue 1263: **RollingDiscPenalty** (extension)

- description: add arbitrary local wheel axis (discAxis) instead of x-axis only
- date resolved: **2022-09-17 10:42**, date raised: 2022-09-17

Version 1.4.3: resolved Issue 1262: **RigidBodyInertia** (extension)

- description: add += operator
- date resolved: **2022-09-17 08:23**, date raised: 2022-09-17

Version 1.4.2: resolved Issue 1261: **Lie group integration** (extension)

- description: improve implicit Lie group integration, adapt old rotation vector approach, add tangent operator
- date resolved: **2022-09-16 18:14**, date raised: 2022-09-16

Version 1.4.1: resolved BUG 1260: **MacOS**

- description: compilation not working on MacOS with taskmanager adaptions
- date resolved: **2022-09-15 11:14**, date raised: 2022-09-15

Version 1.4.0: resolved Issue 0860: **SparseVectorDomain** (extension)

- description: ?NEEDED (see 0862): create templated SparseVector with IndexValue+domain, having C-array of ResizableArray<IndexValue>, used for multithreaded creation of sparse vectors, associated indices for lateron fill into system vectors
- **notes: done in TemporaryComputationData now**
- date resolved: **2022-09-15 08:50**, date raised: 2022-01-13

Version 1.3.105: resolved Issue 0861: **SparseVectorParallel** (extension)

- description: create SparseVector with: mainSparseVector + ArrayIndex2 with per-thread max index and current index; SparseTriplets exceeding the index go into ResizableArray<SparseVector*> threadSparseVector; Function to finally fill all triplets into main vector
- **notes: done in TemporaryComputationData now**
- date resolved: **2022-09-15 08:49**, date raised: 2022-01-13

Version 1.3.104: resolved Issue 1259: **MarkerSuperElementRigid** (extension)

- description: extend offset for case that it is large
- date resolved: **2022-09-14 15:18**, date raised: 2022-09-14

Version 1.3.103: resolved Issue 1258: **AnimateModes** (extension)

- description: add option to change sign of mode
- date resolved: **2022-09-14 08:51**, date raised: 2022-09-14

Version 1.3.102: **resolved BUG 1257**: **AnimateModes**

- description: button for "Faces only" not working due to new way edges are turned on / off; workaround by pressing T in render window
- **notes: buttons now affect the mesh faces / edges instead of regular edges / faces**
- date resolved: **2022-09-14 08:28**, date raised: 2022-09-13

Version 1.3.101: resolved Issue 1256: **GUI** (extension)

- description: added some variables in exudyn.GUI which may be used to adjust appearance of dialogs, specifically for extreme display scaling
- date resolved: **2022-09-13 16:42**, date raised: 2022-09-13

Version 1.3.100: resolved Issue 1252: **use display scaling in GUI** (extension)

- description: use display scaling in visualizationSettings, etc.
- date resolved: **2022-09-13 16:42**, date raised: 2022-09-13

Version 1.3.99: resolved Issue 1255: **tkinter** (change)

- description: put root calls into try-except clause in order to preserve operation on special systems like MacOS or Linux
- date resolved: **2022-09-13 14:49**, date raised: 2022-09-13

Version 1.3.98: resolved Issue 1254: **SolutionViewer, AnimateModes** (extension)

- description: add font size and title; this allows to scale fonts as monitor scaling is not active here
- date resolved: **2022-09-13 14:13**, date raised: 2022-09-13

Version 1.3.97: resolved Issue 1253: **renderState** (docu)

- description: add description into theDoc, in section 3D graphics visualization
- date resolved: **2022-09-13 11:12**, date raised: 2022-09-13

Version 1.3.96: resolved Issue 1251: **useWindowsDisplayScaleFactor** (change)

- description: changed visualizationsSettings.general option name from useWindowsMonitorScaleFactor to useWindowsDisplayScaleFactor
- date resolved: **2022-09-13 10:33**, date raised: 2022-09-13

Version 1.3.95: resolved Issue 0538: **displayScaling** (extension)

- description: add displayScaling to renderState and use this value for GUI.py in visualizationSettings
- **notes: added displayScaling and automatic updating when Windows display scaling is changed or window is moved to other display**
- date resolved: **2022-09-13 10:19**, date raised: 2021-01-06

Version 1.3.94: resolved Issue 1250: **FEM HurtyCraigBampton** (extension)

- description: ComputeHurtyCraigBamptonModes now includes option to compute RBE3 case; adds optional boundary node weights
- date resolved: **2022-09-07 12:33**, date raised: 2022-09-06

Version 1.3.93: resolved Issue 1249: **FEM interface** (extension)

- description: add function GetNodeWeightsFromSurfaceAreas which computes correct weights for linear finite elements (tested for tetrahedrals); this weighting can now reduce erroneous offset in MarkerSuperElementRigid significantly; also used for RBE3 mode computation
- date resolved: **2022-09-07 12:33**, date raised: 2022-09-06

Version 1.3.92: resolved Issue 1248: **AddObjectFFRFreducedOrderWithUserFunctions** (fix)

- description: wrong description of user functions; add missing itemIndex in description
- date resolved: **2022-09-05 18:39**, date raised: 2022-09-05

Version 1.3.91: resolved Issue 1246: **DrawSystemGraph** (extension)

- description: add option to create multi-line graphs; improving appearance and readability
- date resolved: **2022-09-02 09:05**, date raised: 2022-09-02

Version 1.3.90: resolved Issue 1244: **pre-compiled linux** (extension)

- description: add all Python 3.6 - 3.10 linux 64 bit versions to pypi with pip installer

- date resolved: **2022-09-01 10:02**, date raised: 2022-08-25

Version 1.3.89: resolved Issue 0562: **Gen alpha Lie** (extension)
- description: add Lie groups to new generalized alpha integrator
- date resolved: **2022-08-26 14:09**, date raised: 2021-01-26

Version 1.3.88: resolved Issue 1245: **Rotation output variable** (change)
- description: changed/fixed OutputVariableType.Rotation for NodeRigidBodyRotVecLG to output Tait-Bryan rotations instead of rotation parameters; corrected description for NodeRigidBodyRxyz: returns rotation parameters directly, NOT recomputed from RotationMatrix
- date resolved: **2022-08-26 10:25**, date raised: 2022-08-26

Version 1.3.87: resolved Issue 1243: **Lie group integration** (extension)
- issue author: S. Holzinger
- description: add new functions for implicit Lie group integration (FIRST TESTS)
- date resolved: **2022-08-24 17:13**, date raised: 2022-08-24 (resolved by: S. Holzinger)

Version 1.3.86: resolved BUG 1239: **Timer registration**
- description: self-registration of Timers fails on Linux and may lead to crashes; depends on order of initialization of global variables
- **notes: changed timer registration to suggested way with scalar variables, guaranteeing initialization**
- date resolved: **2022-08-24 15:10**, date raised: 2022-08-24

Version 1.3.85: resolved BUG 1225: **Linux TestSuite**
- description: segmentation fault when running ANCFgeneralContactCircle.py
- date resolved: **2022-08-24 09:13**, date raised: 2022-08-11

Version 1.3.84: resolved Issue 1238: **visualization dialog** (change)
- description: resort options, such that contour options are on top
- date resolved: **2022-08-23 15:19**, date raised: 2022-08-23

Version 1.3.83: resolved Issue 1237: **Beams** (extension)
- description: add option for drawing filled cross-sections (or alternatively wire frames)
- date resolved: **2022-08-23 15:19**, date raised: 2022-08-23

Version 1.3.82: resolved Issue 1236: **GeometricallyExactBeam** (fix)
- description: GeometricallyExactBeam2D and GeometricallyExactBeam3D do not show values in contour plot
- date resolved: **2022-08-23 14:15**, date raised: 2022-08-23

Version 1.3.81: resolved Issue 1235: **GeometricallyExactBeam2D** (fix)
- description: add missing output variables (strain, curvatureLocal, forces,torques)
- date resolved: **2022-08-23 12:02**, date raised: 2022-08-23

Version 1.3.80: resolved BUG 1233: **KinematicTree**
- description: Jacobian in KinematicTree has too many approximations: either missing velocity terms have large influence or double entries for connectors on single KinematicTree; check stiffFlyballGovernor w/o systemWideDifferentiation
- **notes: fixed JacobianODE2 for duplicate global indices, e.g., in case that Connector is attached with two markers to same object (kinematic tree)**
- date resolved: **2022-08-23 11:41**, date raised: 2022-08-22

Version 1.3.79: resolved Issue 1234: **Newton / Jacobian** (extension)
- description: add new Newton / numericalDifferentiation setting jacobianConnectorDerivative for faster Jacobian computations
- date resolved: **2022-08-23 10:15**, date raised: 2022-08-23

Version 1.3.78: resolved Issue 1224: **KinematicTree** (check)
- description: visulization problems with kinematicTreeConstraintTest.py with 10 links and more; may be caused by wrong states in visualization or possible bug in KinematicTreeMarker
- **notes: resolved with issue 1232 by adding additional temporary variables for visualization**
- date resolved: **2022-08-22 23:26**, date raised: 2022-08-07

Version 1.3.77: resolved BUG 1232: **KinematicTree**
- description: visualization of joints uses illegal temporary data; leads to data race and erroneous results
- date resolved: **2022-08-22 23:23**, date raised: 2022-08-22

Version 1.3.76: resolved BUG 1231: **BasicDefinitions**
- description: C++: definition of MAXREAL wrong; affects searchtree and contact
- date resolved: **2022-08-22 21:33**, date raised: 2022-08-22

Version 1.3.75: resolved Issue 1230: **GetInitialVector** (change)
- description: change GetInitialVector into GetInitialCoordinateVector for consistency reasons; samge for GetInitialVector_t, SetInitialVector, SetInitialVector_t
- date resolved: **2022-08-17 17:53**, date raised: 2022-08-17

Version 1.3.74: resolved BUG 1229: **CMarkerBodyCable2DShape**

- description: system error due to incorrect initialization of matrix
- date resolved: **2022-08-15 15:31**, date raised: 2022-08-15

Version 1.3.73: resolved Issue 1228: **add LieGroup node with data coordinates** (extension)

- description: add special Lie group node for implicit integration, containing the start-of-step configuration in data coordinates and additionally use regular ODE2 coordinates for the incremental motion
- date resolved: **2022-08-12 19:35**, date raised: 2022-08-12

Version 1.3.72: resolved Issue 1227: **CNode.cpp** (change)

- description: C++: remove exceptions for illegal index access in GetCurrentCoordinate(...)
- date resolved: **2022-08-12 19:18**, date raised: 2022-08-12

Version 1.3.71: resolved Issue 1226: **initial coordinates** (extension)

- description: C++: nodes get a separate SetInitialCoordinateVector() function; used for special nodes with mixed coordinates
- date resolved: **2022-08-12 19:17**, date raised: 2022-08-12

Version 1.3.70: resolved Issue 1115: **KinematicTree** (extension)

- description: C++ implement efficient T66 transformations
- **notes: stable implementation, with formulas different to Featherstone formulas, but in line with 6D matrix manipulations; further tests and documentation needed**
- date resolved: **2022-08-07 22:55**, date raised: 2022-05-29

Version 1.3.69: **resolved BUG 1223**: **T66MotionInverse**

- description: C++: RigidBodyMath implementation of T66 inverse is wrong, could affect special KinematicTree force reaction
- date resolved: **2022-08-05 21:45**, date raised: 2022-08-05

Version 1.3.68: resolved Issue 1222: **KinematicTree** (change)

- description: remove some temporary variables from interface as new efficient transformations cannot be converted to Python easily
- date resolved: **2022-08-04 16:28**, date raised: 2022-08-04

Version 1.3.67: resolved Issue 1221: **Transformations66List** (change)

- description: C++: rename into Transformation66List
- date resolved: **2022-08-04 16:27**, date raised: 2022-08-04

Version 1.3.66: resolved Issue 1220: **PlotImage** (extension)

- description: add options for orthogonal projection and removing axes and background
- date resolved: **2022-07-26 09:34**, date raised: 2022-07-26

Version 1.3.65: resolved Issue 1205: **LinkedDataVectorParallel** (check)

- description: C++: check if LinkedDataVector can obtain performance mode from ResizableVectorParallel
- date resolved: **2022-07-22 20:45**, date raised: 2022-07-12

Version 1.3.64: resolved Issue 1206: **parallel** (extension)

- description: C++: parallelize important vector-vector and matrix-vector (MultMatrix, MultAdd, ...) operations with optional commands
- **notes: already done in ResizableVectorParallel, but extensions in LinkedDataVector needed**
- date resolved: **2022-07-22 19:49**, date raised: 2022-07-12

Version 1.3.63: resolved Issue 1218: **SolverExplicit** (extension)

- description: parallelize Lie group updates in explicit solver
- date resolved: **2022-07-22 19:26**, date raised: 2022-07-22

Version 1.3.62: resolved Issue 1219: **SolverExplicit** (change)

- description: turn off Lie group integration if no Lie group nodes available
- date resolved: **2022-07-22 18:15**, date raised: 2022-07-22

Version 1.3.61: resolved Issue 1160: **numpy arrays** (change)

- description: change conversion behavior for Vector3D and Matrix3D, automatically transformed into numpy arrays instead of std::vector which gives a list right now
- **notes: for items containing VectorXD or MatrixXD, the parameter returned in GetObject() and similar functions is now giving numpy arrays; for system structures, this is anyway already implemented; specifically, the changed behaviour can be used for referenceCoordinates in user functions; BEHAVIOUR CHANGED: check your models!**
- date resolved: **2022-07-21 19:33**, date raised: 2022-06-26

Version 1.3.60: resolved Issue 1209: **parallel** (extension)

- description: C++: add multithreaded parallelization for PostNewton
- date resolved: **2022-07-21 10:02**, date raised: 2022-07-14

Version 1.3.59: resolved Issue 1035: **TestModels** (change)

- description: change modelUnitTests imports in TestModels such that they also work in case that example is run outside TestModels directory
- **notes: done earlier**

- date resolved: **2022-07-20 14:33**, date raised: 2022-04-07

Version 1.3.58: resolved Issue 1016: **TestSuite** (testing)
- description: add example of reeving system
- **notes: done earlier**
- date resolved: **2022-07-20 14:33**, date raised: 2022-03-28

Version 1.3.57: resolved Issue 1217: **GraphicsData** (extension)
- description: add functions for drawing text, line and circle: GraphicsDataLine, GraphicsDataText, GraphicsDataCircle
- date resolved: **2022-07-20 09:45**, date raised: 2022-07-20

Version 1.3.56: **resolved BUG 1216**: **renderer: Circle**
- description: circles drawn wrongly, not closed for circleTiling <= 6 and producing overly many lines
- date resolved: **2022-07-19 20:11**, date raised: 2022-07-19

Version 1.3.55: resolved Issue 1214: **Export lines** (extension)
- description: add option to export all lines from renderer similar to RenderImage, however, just exporting the raw line information (2 points, RGBA color)
- date resolved: **2022-07-19 18:39**, date raised: 2022-07-19

Version 1.3.54: resolved Issue 1215: **GraphicsData addEdges** (fix)
- description: some examples still in a previous state expecting wrong GraphicsData format
- date resolved: **2022-07-19 12:14**, date raised: 2022-07-19

Version 1.3.53: **resolved BUG 1213**: **mbs.systemData.Info()**
- description: mbs.systemData.Info() gives error for KinematicTree
- date resolved: **2022-07-15 15:34**, date raised: 2022-07-15

Version 1.3.52: resolved Issue 1212: **DynamicSolverType::RK67** (fix)
- description: shows DynamicSolverType::invalid
- date resolved: **2022-07-15 15:00**, date raised: 2022-07-15

Version 1.3.51: resolved Issue 1211: **PlotSensor** (extension)
- description: add listMarkerStyles and listMarkerStylesFilled to exudyn.plot for use in loops
- date resolved: **2022-07-15 09:28**, date raised: 2022-07-15

Version 1.3.50: resolved Issue 1210: **microThread** (check)
- description: check if threading can be optimized by only using sync atomic variables
- **notes: no big improvement found, also using bit-wise thread communication and saving some atomic operations**
- date resolved: **2022-07-14 20:40**, date raised: 2022-07-14

Version 1.3.49: resolved Issue 1208: **timer PostNewton** (extension)
- description: add timer for PostNewtonStep
- date resolved: **2022-07-14 00:15**, date raised: 2022-07-14

Version 1.3.48: resolved Issue 1207: **microThreading** (extension)
- description: add micro threading library which already takes effect for small systems (10-20 3D rigid bodies); currently included with separate compile option in BasicDefinitions.h
- date resolved: **2022-07-13 13:43**, date raised: 2022-07-13

Version 1.3.47: resolved Issue 1199: **adjust testSuite** (testing)
- description: due to change of state vector to ResizableVectorParallel with AVX arithmetic, minor changes in reference solutions happened
- date resolved: **2022-07-12 17:06**, date raised: 2022-07-11

Version 1.3.46: resolved Issue 1193: **SparseSolver analyzePattern** (extension)
- description: add option to reuse result of analyzePattern() for successive computations; especially in time integration, using the number of non-zeros as indicator is something has changed; add some initializationFunctions to reset, especially after non-convergence
- date resolved: **2022-07-12 17:06**, date raised: 2022-07-10

Version 1.3.45: resolved Issue 1204: **parallel / multithreaded** (extension)
- description: C++: add multithreading for AlgebraicEquations
- date resolved: **2022-07-12 15:03**, date raised: 2022-07-12

Version 1.3.44: resolved Issue 1201: **parallel / multithreaded** (extension)
- description: C++: add multithreading for ProjectedReactionForces
- date resolved: **2022-07-12 13:04**, date raised: 2022-07-12

Version 1.3.43: resolved Issue 1200: **CSystem::Jacobians** (change)
- description: C++: removed single flags for jacobians and replaced by JacobianType; check your results (bugs may happen)
- date resolved: **2022-07-12 10:55**, date raised: 2022-07-12

Version 1.3.42: resolved Issue 1198: **parallel MassMatrix** (extension)

- description: use multithreading to compute mass matrix; treat objects with user functions separately
- date resolved: **2022-07-11 23:18**, date raised: 2022-07-11

Version 1.3.41: resolved BUG 1197: **ResizableArray**

- description: C++: copy of illegal parts of memory when enlarging array
- date resolved: **2022-07-11 22:58**, date raised: 2022-07-11

Version 1.3.40: resolved Issue 1195: **remove openmp** (change)

- description: remove openmp compile options from setup.py as it is not used for now
- date resolved: **2022-07-11 19:02**, date raised: 2022-07-11

Version 1.3.39: resolved Issue 1191: **SystemState** (change)

- description: C++: replace Vector state with ResizableVector(Parallel)
- date resolved: **2022-07-10 14:50**, date raised: 2022-07-10

Version 1.3.38: resolved Issue 1186: **ALEANCFCable2D** (extension)

- description: add missing terms related to curvature and strain coupling with delta qALE
- date resolved: **2022-07-09 21:25**, date raised: 2022-07-06

Version 1.3.37: resolved Issue 1190: **ContactFrictionCircleCable2D** (extension)

- description: now adapted to work in general with ALECable2D, however, only for tangential frictionStiffness=0
- date resolved: **2022-07-09 14:33**, date raised: 2022-07-09

Version 1.3.36: resolved BUG 1188: **beam.py**

- description: missing eii structure before Point2DS1
- date resolved: **2022-07-08 16:22**, date raised: 2022-07-08

Version 1.3.35: resolved Issue 1185: **ContactFrictionCircleCable2D** (extension)

- description: add ALE term to marker and contact element
- date resolved: **2022-07-06 15:30**, date raised: 2022-07-06

Version 1.3.34: resolved Issue 1183: **star imports** (change)

- description: remove * imports from all .py modules except utilities.py
- **notes: also fixed some undetected bugs in unused functions in exudyn.* utilities**
- date resolved: **2022-07-06 11:56**, date raised: 2022-07-06

Version 1.3.33: resolved Issue 1184: **exudyn.utilities** (change)

- description: remove import of time and copy; needs to be included separately into models
- **notes: check your models!**
- date resolved: **2022-07-06 11:55**, date raised: 2022-07-06

Version 1.3.32: resolved Issue 1182: **roboticsCore.py** (change)

- description: remove * import of many exudyn packages
- date resolved: **2022-07-06 10:58**, date raised: 2022-07-06

Version 1.3.31: resolved Issue 1181: **setup.py** (change)

- description: due to deprecation warning, use namespace_packages instead of packages in setup(...); remove include_package_data=True
- date resolved: **2022-07-06 09:43**, date raised: 2022-07-06

Version 1.3.30: resolved Issue 0954: **ObjectContactFrictionCircleCable2D** (extension)

- description: Add exception in PreAssembleChecks if marker is not a MarkerBody, as implementation only works for MarkerBody but not for MarkerNode
- **notes: resolved, as now a MarkerNodeRigid may also be used**
- date resolved: **2022-07-05 21:18**, date raised: 2022-02-27

Version 1.3.29: resolved Issue 1180: **AddEdgesAndSmoothenNormals** (extension)

- description: specialized function for STL file enhancement
- date resolved: **2022-07-05 20:51**, date raised: 2022-07-05

Version 1.3.28: resolved Issue 1179: **show lines** (extension)

- description: add separate visualization.openGL flag for showing/hiding lines
- date resolved: **2022-07-05 11:20**, date raised: 2022-07-05

Version 1.3.27: resolved Issue 1178: **STL import/export** (extension)

- description: add option to invert triangles and/or normls on import or export of STL meshes
- date resolved: **2022-07-05 08:38**, date raised: 2022-07-05

Version 1.3.26: resolved Issue 1177: **selection right mouse** (extension)

- description: now also showing GraphicsData optionally with visualizationSettings.interactive.selectionRightMouseGraphicsData
- date resolved: **2022-07-05 08:38**, date raised: 2022-07-05

Version 1.3.25: resolved Issue 1099: **TriangleList** (extension)

- description: extend GraphicsData TriangleList with two optional lists (which may be empty) containing edges (tuples of point numbers) as well as edgeColors; allows to easily add edges to graphics representation
- date resolved: **2022-07-05 00:04**, date raised: 2022-05-22

Version 1.3.24: resolved Issue 1174: **mbs.GetObject** (extension)

- description: add option to receive graphicsData (or not)
- **notes: default behavior kept same, not returning graphicsData in dict**
- date resolved: **2022-07-04 23:59**, date raised: 2022-07-04

Version 1.3.23: resolved Issue 1159: **BodyGraphicsData** (extension)

- description: add method to convert bodyGraphicsData into dictionary; add flag to GetObject to by default not show graphicsData
- date resolved: **2022-07-04 23:59**, date raised: 2022-06-26

Version 1.3.22: resolved Issue 1176: **GraphicsDataCylinder...()** (change)

- description: returns always a GraphicsData dictionary, independently of addEdges is True or False
- date resolved: **2022-07-04 23:33**, date raised: 2022-07-04

Version 1.3.21: resolved Issue 1175: **GraphicsDataOrthoCube...()** (change)

- description: returns always a GraphicsData dictionary, independently of adding edges or not
- date resolved: **2022-07-04 23:33**, date raised: 2022-07-04

Version 1.3.20: resolved Issue 1169: **conversion to STL** (extension)

- description: add function to convert graphicsData triangle meshes into STL; allows import into other tools
- date resolved: **2022-07-04 20:55**, date raised: 2022-07-03

Version 1.3.19: resolved Issue 0821: **include numpy-stl** (extension)

- description: include library to import stl files; add Warning to GraphicsDataFromSTLfileTxt for large file sizes and check if is ascii; in binary case, try switching to numpy-stl; add example to load binary files; add example to convert stl ascii to binary files
- **notes: example for stl import added: Examples/stlFileImport.py**
- date resolved: **2022-07-04 20:55**, date raised: 2021-12-06

Version 1.3.18: resolved Issue 1173: **GetObjectOutputBody** (change)

- description: add default argument localPosition=[0,0,0]
- date resolved: **2022-07-04 11:45**, date raised: 2022-07-04

Version 1.3.17: resolved Issue 1172: **GetObjectOutput** (change)

- description: include configurationType in interface (default: Current); not available in connectors; add objectNumber in C++ interface
- date resolved: **2022-07-04 11:43**, date raised: 2022-07-04

Version 1.3.16: **resolved BUG 1171**: **MarkerKinematicTreeRigid**

- description: incorrect GetPosition, GetVelocity, GetAngularVelocity, ...
- date resolved: **2022-07-04 11:43**, date raised: 2022-07-03

Version 1.3.15: resolved Issue 1161: **KinematicTree** (change)

- description: change GetObjectOutputBody to GetObjectOutput as localPosition does not make sense here
- date resolved: **2022-07-04 11:39**, date raised: 2022-06-27

Version 1.3.14: resolved Issue 1170: **MarkerKinematicTreeRigid** (extension)

- description: add visualization
- date resolved: **2022-07-03 20:23**, date raised: 2022-07-03

Version 1.3.13: resolved Issue 1168: **left mouse** (change)

- description: deactivate mouse select if renderer is showing mouse coordinates (and doing measuring)
- date resolved: **2022-07-02 11:29**, date raised: 2022-07-02

Version 1.3.12: resolved Issue 1166: **InteractiveDialog** (extension)

- description: In all interactive dialogs, especially the SolutionViewer now stopping the render window (with Q or Escape) also stops the interactive dialog; behavior changed with checkRenderEngineStopFlag
- date resolved: **2022-06-29 10:15**, date raised: 2022-06-29

Version 1.3.11: resolved Issue 1165: **class Robot** (change)

- description: drawing of cylinder for first body in robot removed; this part must be drawn manually at base (or not drawn)
- date resolved: **2022-06-29 10:02**, date raised: 2022-06-29

Version 1.3.10: resolved Issue 0720: **AddObjectFFRFreducedOrder** (extension)

- description: add gravity as with user functions version, similar to AddRigidBody
- date resolved: **2022-06-29 09:27**, date raised: 2021-07-12

Version 1.3.9: resolved Issue 1018: **MacOS** (change)

- description: adjust mouse scroll factor in case of Apple/Mac OS compilation (e.g. multiply with 0.05 by default)
- **notes: changed way to compute zoomFactor for larger yOffsets in scroll callback**

- date resolved: **2022-06-28 15:30**, date raised: 2022-03-29

Version 1.3.8: resolved Issue 1164: **OpenGL shadow** (fix)
- description: resolve issues with shadows drawing of many objects; added _WRAP to stencil INCR and DECR operations to resolve overflows
- date resolved: **2022-06-27 19:20**, date raised: 2022-06-27

Version 1.3.7: resolved Issue 1163: **OpenGL normals** (fix)
- description: correct orientation of triangles and normals in GraphicsData functions
- date resolved: **2022-06-27 16:22**, date raised: 2022-06-27

Version 1.3.6: resolved Issue 1162: **OpenGL normals** (change)
- description: correct normals in GraphicsDataSphere, GraphicsDataCylinder and GraphicsDataSolidOfRevolution to point outwards; turn off GL_LIGHT_MODEL_TWO_SIDE by default
- date resolved: **2022-06-27 14:12**, date raised: 2022-06-27

Version 1.3.5: resolved Issue 1158: **RigidBodyInertia** (extension)
- description: add Transformed() function to return rigid body inertia transformed by homogeneous transformation; used in class Robot for certain transformations of link frame
- date resolved: **2022-06-27 01:20**, date raised: 2022-06-26

Version 1.3.4: resolved Issue 1072: **class Robot** (extension)
- description: extend KinematicTree export in order to be able to create robots with localHT!=HT0()
- date resolved: **2022-06-27 00:41**, date raised: 2022-05-06

Version 1.3.3: resolved Issue 1131: **class Robot** (extension)
- description: add list of graphicsDataLists to each body for individual graphics of links allowing also graphics for tools
- date resolved: **2022-06-26 23:46**, date raised: 2022-06-03

Version 1.3.2: resolved Issue 1094: **add links to other github repos** (docu)
- description: link more directly to ngsolve and openAI / stable_baslines3 / etc.; refer to openAI in github links and add ref to theDoc.pdf ; add example/teaser?
- date resolved: **2022-06-24 09:20**, date raised: 2022-05-21

Version 1.3.1: resolved Issue 1157: **C++ user functions** (testing)
- description: check if linking C++ user functions directly boosts performance
- **notes: C++ functions are translated by including pybind11/functional.h; workaround would be two user functions, one without MainSystem and without including functional.h; C++ functions could then be compiled in a very simple manner**
- date resolved: **2022-06-23 16:57**, date raised: 2022-06-23

Version 1.3.0: resolved BUG 0677: **single threaded renderer**
- description: correct crash with visualization dialog (MacOS)
- **notes: not resolved, but it is an issue of missing tkinter capabilities in MacOS**
- date resolved: **2022-06-22 07:59**, date raised: 2021-05-12

Version 1.2.146: resolved Issue 0735: **parallel build** (check)
- description: check parallel build with MSbuild to reduce compilation times
- **notes: already done earlier**
- date resolved: **2022-06-22 07:58**, date raised: 2021-08-12

Version 1.2.145: resolved Issue 0863: **RaspberryPi** (extension)
- description: check compilation on Raspi, make adaptation of ngsolve includes to run
- **notes: already done earlier**
- date resolved: **2022-06-22 07:48**, date raised: 2022-01-14

Version 1.2.144: resolved Issue 0968: **MarkerBodyCable2DShape** (extension)
- description: add offset from beam axis, to compute position, velocity and jacobians at contact surface
- **notes: already done earlier**
- date resolved: **2022-06-22 07:46**, date raised: 2022-03-03

Version 1.2.143: resolved Issue 0972: **ContactFrictionCircleCable2D** (docu)
- description: add / extend description
- date resolved: **2022-06-22 07:44**, date raised: 2022-03-09

Version 1.2.142: resolved Issue 1155: **center point** (extension)
- description: add key "o" option to set center point to current center point of view; this allows to rotate around the current center point of the view
- **notes: currently deactivated due to transformation of coordinates issue**
- date resolved: **2022-06-21 18:16**, date raised: 2022-06-21

Version 1.2.141: resolved Issue 1152: **Renderer** (extension)
- description: add separate function to compute accurate scene size, store as 3D vector and as norm
- date resolved: **2022-06-21 17:56**, date raised: 2022-06-20

Version 1.2.140: resolved Issue 1154: **Zoom all** (change)

- description: change procedures for zoom all and for computation of maximum scene coordinates; may affect appearance of your models; necessary for consistently computing perspective and shadow parameters
- date resolved: **2022-06-21 08:38**, date raised: 2022-06-21

Version 1.2.139: resolved Issue 1153: **noglfw** (check)

- description: check compilation without GLFW
- date resolved: **2022-06-20 10:58**, date raised: 2022-06-20

Version 1.2.138: resolved Issue 1150: **OpenGL** (extension)

- description: add shadows (simple)
- **notes: activate with SC.VisualizationSettings().openGL.shadow, chosing a value between 0. and 1; good results obtained with 0.5**
- date resolved: **2022-06-20 01:49**, date raised: 2022-06-18

Version 1.2.137: resolved Issue 1151: **linux builds** (change)

- description: remove -g flag from linux builds, leading to 2.6MB instead of 38MB binaries; to enable debug information (e.g. to detect origin of some crashes, remove the -g0 flag in setup.py)
- date resolved: **2022-06-19 18:34**, date raised: 2022-06-19

Version 1.2.136: resolved Issue 1149: **OpenGL** (extension)

- description: add perspective
- **notes: added openGL option perspective; EXPERIMENTAL!**
- date resolved: **2022-06-19 01:59**, date raised: 2022-06-18

Version 1.2.135: resolved Issue 1148: **jacobian ODE1 ODE2** (fix)

- description: fix jacobian computations for mixed ODE1-ODE2 components; check with HydraulicActuatorSimple (lower number of jacobians and iterations)
- date resolved: **2022-06-18 23:38**, date raised: 2022-06-18

Version 1.2.134: resolved Issue 1147: **ODE1Coordinates_t** (extension)

- description: add missing function SetODE1Coordinates_t and GetODE1Coordinates_t into Python interface
- date resolved: **2022-06-17 01:14**, date raised: 2022-06-17

Version 1.2.133: resolved Issue 1096: **hydraulics** (example)

- description: add hydraulic actuator with new element HydraulicActuatorSimple
- date resolved: **2022-06-17 00:06**, date raised: 2022-05-22

Version 1.2.132: resolved Issue 1146: **solver jacobians** (extension)

- description: extended jacobians for ODE1 and ODE2 residuals for ODE1-ODE2 coupling in ComputeJacobianODE2RHS, ComputeJacobianODE1RHS, and ComputeJacobianAE; changed default values as well; check your code if you used these functions in Python user functions
- date resolved: **2022-06-16 19:35**, date raised: 2022-06-16

Version 1.2.131: resolved Issue 1145: **static solver** (extension)

- description: add check if system contains ODE1 variables
- date resolved: **2022-06-16 18:52**, date raised: 2022-06-16

Version 1.2.130: resolved Issue 1144: **PlotSensor** (extension)

- description: added option to allow components=[plot.componentNorm] for displaying norm of sensors
- date resolved: **2022-06-16 17:32**, date raised: 2022-06-16

Version 1.2.129: resolved Issue 1143: **numba jit** (example)

- description: add example for numba jit speedup of Python user functions
- **notes: added Example springDamperUserFunctionNumbaJIT.py showing speedup of 4 for simple Python function; note that mbs functions cannot be processed by numba**
- date resolved: **2022-06-16 17:01**, date raised: 2022-06-16

Version 1.2.128: resolved Issue 1106: **HydraulicsActuator** (extension)

- description: add HydraulicsActuator as object, containing pressure equations
- **notes: added new double acting HydraulicActuatorSimple with internal pressure equations and possibility to modify valves by user functions**
- date resolved: **2022-06-16 12:00**, date raised: 2022-05-23

Version 1.2.127: resolved Issue 1138: **artificialIntelligence** (extension)

- description: OpenAIGymInterfaceEnv obtained additional member variable randomInitializationValue which may be adapted; in future, this may be done in a separate function
- date resolved: **2022-06-10 12:26**, date raised: 2022-06-10

Version 1.2.126: resolved Issue 1137: **exudynFast** (change)

- description: for linux builds, do not activate exudynFast, only for noglfw
- date resolved: **2022-06-10 12:16**, date raised: 2022-06-10

Version 1.2.125: resolved BUG 1135: **KinematicTree**

- description: linux version of KinematicTree gives significantly (6e-7) different results, check initialization
- **notes: resulted due to high sensitivity to disturbances (1e-15), especially in acceleration sensors, and ONLY for mbs results!**
- date resolved: **2022-06-09 20:32**, date raised: 2022-06-09

Version 1.2.124: resolved Issue 1134: **KinematicTree** (extension)

- description: add Jacobian computation according to class Robot in order to realize AccessFunction needed from MarkerKinematicTreeRigid; uses special access function because the SuperElement interface does not provide link number
- date resolved: **2022-06-08 18:30**, date raised: 2022-06-06

Version 1.2.123: resolved Issue 1071: **KinematicTree** (extension)

- description: add special MarkerKinematicTreeRigidBody with link number and local position
- date resolved: **2022-06-08 18:30**, date raised: 2022-05-05

Version 1.2.122: resolved Issue 1130: **class Robot** (change)

- description: tool only working for serial robots; in case of tree structure, tool should not be used as it is attached only to last link
- **notes: added warning to class Robot.AddLink(...) which raises Warning if tool is defined and tree structure is generated**
- date resolved: **2022-06-06 00:32**, date raised: 2022-06-03

Version 1.2.121: resolved Issue 1132: **class Robot** (extension)

- description: extend Jacobian, LinkHT and JointHT functions for tree structure; check StaticTorques
- **notes: added comparison for class Robot functions in kinematicTreeAndMBStest.py**
- date resolved: **2022-06-06 00:31**, date raised: 2022-06-03

Version 1.2.120: resolved Issue 1133: **SolutionViewer** (extension)

- description: bind additional Button "Q" in interactive dialogs for quit (in addition to Escape)
- date resolved: **2022-06-05 18:25**, date raised: 2022-06-05

Version 1.2.119: resolved Issue 1114: **class Robot** (testing)

- description: create new example(s) comparing CreateKinematicTree and CreateRedundantCoordinateMBS with control and prismatic joints
- date resolved: **2022-06-05 14:33**, date raised: 2022-05-29

Version 1.2.118: resolved BUG 1127: **class Robot**

- description: CreateKinematicTree not working for tree structure
- date resolved: **2022-06-03 00:28**, date raised: 2022-06-03

Version 1.2.117: resolved BUG 1129: **class Robot**

- description: GetParentIndex(...), HasParent(...) not working for tree structure
- date resolved: **2022-06-03 00:19**, date raised: 2022-06-03

Version 1.2.116: resolved BUG 1128: **class Robot**

- description: CreateRedundantCoordinateMBS not working for tree structure
- date resolved: **2022-06-03 00:19**, date raised: 2022-06-03

Version 1.2.115: resolved BUG 1126: **KinematicTree**

- description: wrong formula in computation of acceleration
- date resolved: **2022-06-02 22:41**, date raised: 2022-06-02

Version 1.2.114: resolved Issue 1123: **class Robot** (extension)

- description: extend CreateRedundantCoordinateMBS for jointSpringDamperUserFunctionList with prismatic joints
- date resolved: **2022-06-01 23:44**, date raised: 2022-06-01

Version 1.2.113: resolved Issue 1125: **class Robot** (extension)

- description: extend CreateRedundantCoordinateMBS for control of PrsimaticJoints with new LinearSpringDamper
- date resolved: **2022-06-01 23:43**, date raised: 2022-06-01

Version 1.2.112: resolved Issue 1124: **LinearSpringDamper** (extension)

- description: add LinearSpringDamper which is the corresponding spring damper/actuator for prismatic joints (like TorsionalSpringDamper for revolute joints), being aligned with a rigid marker, having no limits compared to regular (distance based) SpringDamper
- date resolved: **2022-06-01 23:43**, date raised: 2022-06-01

Version 1.2.111: resolved Issue 0503: **serialrobot** (extension)

- description: adapt serial robot for prismatic joints
- **notes: functionality with Prismatic joints now included into class Robot, converting to mbs with CreateRedundantCoordinateMBS or CreateKinematicTree, see issue**
- date resolved: **2022-06-01 20:38**, date raised: 2020-12-16

Version 1.2.110: resolved Issue 1121: **AnimateSolution** (change)

- description: mark as deprecated, use SolutionViewer instead
- date resolved: **2022-06-01 20:25**, date raised: 2022-05-31

Version 1.2.109: resolved BUG 1122: **GraphicsDataBasis**

- description: kwargs argument radius not working
- date resolved: **2022-05-31 15:32**, date raised: 2022-05-31

Version 1.2.108: resolved Issue 1108: **KinematicTree** (check)

- description: check OutputVariable functions and compare with redundant-MBS based bodies
- date resolved: **2022-05-30 21:06**, date raised: 2022-05-27

Version 1.2.107: resolved Issue 1095: **hydraulics** (example)

- description: add hydraulic actuator with user function
- **notes: example HydraulicsUserFunction.py added already 5 days earlier**
- date resolved: **2022-05-30 20:06**, date raised: 2022-05-22

Version 1.2.106: resolved Issue 1109: **class Robot** (extension)

- description: CreateRedundantCoordinateMBS: jointLoadUserFunctionList and createJointTorqueLoads marked as deprecated; use NEW jointSpringDamperUserFunctionList which allows to directly actuate at revolute/prismatic joints
- date resolved: **2022-05-30 20:05**, date raised: 2022-05-29

Version 1.2.105: resolved BUG 1120: **KinematicTree**

- description: composite inertia contains wrong index in calculation
- date resolved: **2022-05-30 20:04**, date raised: 2022-05-30

Version 1.2.104: resolved Issue 1113: **KinematicTree** (testing)

- description: test for prismatic joint
- date resolved: **2022-05-30 20:04**, date raised: 2022-05-29

Version 1.2.103: resolved BUG 1119: **KinematicTree**

- description: wrong signs (missing inverse) of prismatic joints in C++ and Python implementation of KinematicTree
- date resolved: **2022-05-30 17:55**, date raised: 2022-05-30

Version 1.2.102: resolved BUG 1118: **ObjectKinematicTree**

- description: conversion to dict in mbs.GetObject(...) does not work for Matrix3D
- date resolved: **2022-05-30 16:02**, date raised: 2022-05-30

Version 1.2.101: resolved Issue 1117: **Python3.8** (change)

- description: the exudyn version for Python3.8 now includes range checks (and is slower than before); for fast version of exudyn without range checks, check issues 1116 and section 'Performance and ways to speed up computations' in theDoc
- date resolved: **2022-05-30 00:34**, date raised: 2022-05-30

Version 1.2.100: resolved Issue 1116: **exudynFast** (extension)

- description: add separate track for fast exudyn versions; the compiler flag _FAST_EXUDYN_LINALG, previously only activated in Python 3.8, is now used for Python 3.7 and Python 3.8 versions ONLY if according import flags are set by doing the following 3 steps: import sys; sys.exudynFast=True; import exudyn
- date resolved: **2022-05-30 00:34**, date raised: 2022-05-29

Version 1.2.99: resolved Issue 1112: **class Robot** (change)

- description: CreateKinematicTree: jointForceVector, jointPositionOffsetVector, jointVelocityOffsetVector, jointPControlVector, jointDControlVector removed and replaced by PDcontrol structure in RobotLink; forceUserFunction kept as is; return value slightly changed
- date resolved: **2022-05-29 19:10**, date raised: 2022-05-29

Version 1.2.98: resolved Issue 1111: **class Robot** (extension)

- description: RobotLink adds feature to define PDcontrol, used in robots for joint control of link
- date resolved: **2022-05-29 19:10**, date raised: 2022-05-29

Version 1.2.97: resolved Issue 1110: **class Robot** (extension)

- description: CreateRedundantCoordinateMBS: returns additional list springDamperList, which contains more efficient spring dampers for joint control
- date resolved: **2022-05-29 17:05**, date raised: 2022-05-29

Version 1.2.96: resolved Issue 1070: **KinematicTree** (extension)

- description: add SensorSuperElement (BUT call it SensorKinematicTree!) functionality for Position, Velocity, Acceleration, RotationMatrix, AngularVelocity, ...
- **notes: created new SensorKinematicTree; tests not done yet**
- date resolved: **2022-05-27 23:43**, date raised: 2022-05-05

Version 1.2.95: resolved Issue 1107: **Sensors** (change)

- description: C++: move sensor-specific consistency tests from CSystem to CheckPreAssembleConsistency
- date resolved: **2022-05-27 23:41**, date raised: 2022-05-26

Version 1.2.94: resolved Issue 1103: **Object** (extension)

- description: C++: enable objects to be mixed of ODE2, ODE1 and AE variables (most general case); add jacobian_ODE1ODE2 flags and functions in case of ODE1

- **notes: NumericalJacobianODE1RHS modified accordingly**
- date resolved: **2022-05-23 23:06**, date raised: 2022-05-23

Version 1.2.93: resolved Issue 1101: **HydraulicActuator** (example)

- description: add HydraulicActuator with user function example
- **notes: Duplicate of issue 1102**
- date resolved: **2022-05-23 22:10**, date raised: 2022-05-23

Version 1.2.92: resolved Issue 1105: **NodeGenericAE** (extension)

- description: add node with algebraic variables
- date resolved: **2022-05-23 22:02**, date raised: 2022-05-23

Version 1.2.91: resolved Issue 0850: **GetNodeODE1Index, GetNodeAEIndex** (extension)

- description: add missing access functions
- date resolved: **2022-05-23 21:48**, date raised: 2022-01-08

Version 1.2.90: resolved Issue 1102: **HydraulicActuator** (example)

- description: add HydraulicActuator with user function example
- **notes: Examples/HydraulicsUserFunction.py**
- date resolved: **2022-05-23 19:53**, date raised: 2022-05-23

Version 1.2.89: resolved Issue 1098: **GraphicsDataCylinder** (extension)

- description: add option addEdges which in addition returns GraphicsData for edges; returns then list of dictionaries
- date resolved: **2022-05-22 20:54**, date raised: 2022-05-22

Version 1.2.88: resolved Issue 1097: **GraphicsDataOrthoCubePoint** (extension)

- description: add option addEdges which in addition returns GraphicsData for edges; returns then list of dictionaries
- date resolved: **2022-05-22 20:54**, date raised: 2022-05-22

Version 1.2.87: **resolved BUG 1093**: **ANCFBeam3D**

- description: giving wrong results because of error in template<class TMatrix> ConstSizeMatrixBase& operator+= (const TMatrix& matrix); leading to twice the mass matrix
- date resolved: **2022-05-20 16:07**, date raised: 2022-05-20

Version 1.2.86: resolved Issue 1092: **openAI gym** (extension)

- description: add example for interface with openAI gym using cart-pole model
- **notes: see Examples/testGymCartpole.py**
- date resolved: **2022-05-18 09:47**, date raised: 2022-05-18

Version 1.2.85: **resolved BUG 1091**: **ComputeODE2Eigenvalues**

- description: eigenvectors sorting is not according to eigenvalues sorting
- date resolved: **2022-05-17 10:00**, date raised: 2022-05-17

Version 1.2.84: resolved Issue 1090: **ComputeODE2Eigenvalues** (extension)

- description: solver.ComputeODE2Eigenvalues gets additional flag constrainedCoordinates to specify list of constrained coordinates in system for eigenvalue computation
- date resolved: **2022-05-16 23:01**, date raised: 2022-05-16

Version 1.2.83: **resolved BUG 1089**: **ComputeODE2Eigenvalues**

- description: setInitialValues has no effect
- **notes: flag erased**
- date resolved: **2022-05-16 22:03**, date raised: 2022-05-16

Version 1.2.82: **resolved BUG 1086**: **ParameterVariation**

- description: numberOfThreads cannot be passed to parameter variation as it is duplicated by **args
- **notes: numberOfThreads is now a named argument**
- date resolved: **2022-05-16 16:34**, date raised: 2022-05-16

Version 1.2.81: resolved Issue 1079: **Beam drawing** (extension)

- description: add generic drawing function for beams, especially for contour plot
- date resolved: **2022-05-15 23:24**, date raised: 2022-05-09

Version 1.2.80: resolved Issue 1076: **ANCFBeam3D** (extension)

- description: add shear deformable 3D ANCF beam according to Nachbagauer Gerstmayr Pechstein using structural mechanics formulation
- date resolved: **2022-05-15 23:24**, date raised: 2022-05-06

Version 1.2.79: resolved Issue 1081: **linux testSuite** (check)

- description: check failed tests under linux; see tests for 1.2.75 linux
- **notes: resolved issues related to initial accelerations; still deviation in test generalContactFrictionTests.py**
- date resolved: **2022-05-11 18:54**, date raised: 2022-05-11

Version 1.2.78: **resolved BUG 1084**: **node frame drawing**

- description: node frames shows letter N unintended
- date resolved: **2022-05-11 18:23**, date raised: 2022-05-11

Version 1.2.77: resolved BUG 1083: **initial accelerations**

- description: not correctly handled in linux build (and probably also on MacOS
- date resolved: **2022-05-11 16:18**, date raised: 2022-05-11

Version 1.2.76: resolved Issue 1082: **testsuite linux** (extension)

- description: extend test suite to run always for linux and add file ending for linux and MacOS
- date resolved: **2022-05-11 12:51**, date raised: 2022-05-11

Version 1.2.75: resolved BUG 1080: **gcc compile**

- description: errors when compiling PyVectorLists and PyMatrixLists on gcc
- date resolved: **2022-05-11 09:27**, date raised: 2022-05-11

Version 1.2.74: resolved Issue 0471: **geometrically exact beam3D** (extension)

- description: add to CPP
- **notes: added first version based on conventional parameterization of nodes, but using Lie groups for beam section forces in ode2LHS**
- date resolved: **2022-05-09 21:24**, date raised: 2020-11-21

Version 1.2.73: resolved Issue 1077: **TExpSO3(Omega)** (change)

- description: reduce number of terms in termExpanded; only up to x**4 needed for 16 digits
- date resolved: **2022-05-09 21:22**, date raised: 2022-05-07

Version 1.2.72: resolved Issue 1074: **BeamSection** (extension)

- description: add new structure (cpp+Python) for beam cross section as well as BeamSectionGeometry for geometrical representation
- date resolved: **2022-05-09 21:22**, date raised: 2022-05-06

Version 1.2.71: resolved Issue 1073: **structures description** (extension)

- description: add description for simulationSettings, visualizationSettings, solver structures, etc. in Python bindings
- date resolved: **2022-05-06 13:54**, date raised: 2022-05-06

Version 1.2.70: resolved Issue 1069: **KinematicTree** (extension)

- description: add interface to class Robot, to create either redundant or minimal coordinates kinematic tree
- **notes: however, is not capable of creating robots with localHT!=HT0()**
- date resolved: **2022-05-06 00:07**, date raised: 2022-05-05

Version 1.2.69: resolved Issue 1064: **KinematicTree** (docu)

- description: add basic description (equations)
- date resolved: **2022-05-05 22:14**, date raised: 2022-05-05

Version 1.2.68: resolved Issue 1067: **KinematicTree** (extension)

- description: add standalone example
- **notes: see TestModels/kinematicTreeTest.py**
- date resolved: **2022-05-05 20:00**, date raised: 2022-05-05

Version 1.2.67: resolved Issue 1065: **KinematicTree** (docu)

- description: add user function description
- date resolved: **2022-05-05 20:00**, date raised: 2022-05-05

Version 1.2.66: resolved Issue 1068: **KinematicTree** (extension)

- description: add test
- **notes: added test and MiniExample**
- date resolved: **2022-05-05 19:59**, date raised: 2022-05-05

Version 1.2.65: resolved BUG 1063: **KinematicTree**

- description: cpp version gives wrong results as compared to MBS
- **notes: corrected inertia - must be w.r.t. COM**
- date resolved: **2022-05-05 16:14**, date raised: 2022-05-05

Version 1.2.64: resolved Issue 1066: **BodyGraphicsDataList** (extension)

- description: new interface used in Kinematic tree which holds a list of GraphicsData lists
- date resolved: **2022-05-05 11:32**, date raised: 2022-05-05

Version 1.2.63: resolved Issue 0655: **add KinematicTree** (extension)

- description: ObjectKinematicTree (minimal coordinates)
- **notes: this is a first version, but tests and validation are necessary!**
- date resolved: **2022-05-05 10:27**, date raised: 2021-05-01

Version 1.2.62: resolved Issue 0913: **KinematicTree** (extension)

- description: add efficient C++ functionality for operating on 6D vectors and matrices for kinematics/dynamics

- date resolved: **2022-05-05 10:23**, date raised: 2022-02-02

Version 1.2.61: resolved Issue 1062: **RigidBodyMath.h** (change)
- description: remove duplicate from src/Utilities
- date resolved: **2022-05-03 11:33**, date raised: 2022-05-03

Version 1.2.60: resolved Issue 1047: **Matrix3DList** (extension)
- description: add new structure Matrix3DList; used to create list of 3D matrices and transfer into MainSystem mbs, e.g. in KinematicTree
- date resolved: **2022-05-01 00:47**, date raised: 2022-04-25

Version 1.2.59: resolved Issue 1038: **simulateInRealtime** (extension)
- description: put waitMicroSeconds into settings
- date resolved: **2022-04-30 20:41**, date raised: 2022-04-07

Version 1.2.58: resolved Issue 1049: **MacOS** (check)
- description: check if flag -mmacosx-version-min=11.0 is needed in setup.py and if this causes problems on older pre-AppleM1 which needs wheel version 10.9
- **notes: x86 compilation requires 11.0, otherwise it fails; need to find another way for compilation**
- date resolved: **2022-04-29 09:59**, date raised: 2022-04-25

Version 1.2.57: resolved Issue 1060: **buildDate.tex** (change)
- description: remove by default change of buildDate.tex in setup.py
- date resolved: **2022-04-29 08:20**, date raised: 2022-04-28

Version 1.2.56: resolved BUG 1059: **MacOS compile**
- description: sse2neon.h not found
- **notes: exclude parallel threads from MacOS version until resolving compilation problem**
- date resolved: **2022-04-28 11:58**, date raised: 2022-04-28

Version 1.2.55: resolved Issue 0905: **Add description of Pybind11 interaction** (docu)
- description: add some description of C++-Python interaction and entry points into C++
- date resolved: **2022-04-27 20:37**, date raised: 2022-02-01

Version 1.2.54: resolved Issue 1015: **TestSuite** (docu)
- description: add notes in Exudyn on testsuite (add remark: in many cases for tracking of changes, not for validation)
- date resolved: **2022-04-27 20:21**, date raised: 2022-03-28

Version 1.2.53: resolved Issue 1058: **ClearWorkspace** (change)
- description: also close open matplotlib figures before they are lost
- date resolved: **2022-04-27 19:47**, date raised: 2022-04-27

Version 1.2.52: resolved BUG 1052: **ClearWorkspace**
- description: has no effect on global variables
- **notes: NOTE: matplotlib looses figures which cannot be closed with closeAll in PlotSensor**
- date resolved: **2022-04-27 19:03**, date raised: 2022-04-25

Version 1.2.51: resolved BUG 1053: **coordinatesSolution**
- description: coordinatesSolutionFile is corrupted for ACF test in text mode
- **notes: not repeatable; may have been caused by iPython crash**
- date resolved: **2022-04-27 10:29**, date raised: 2022-04-26

Version 1.2.50: resolved Issue 1051: **CreateNonlinearFEMObjectGenericODE2NGsolve** (testing)
- description: test FEM.CreateNonlinearFEMObjectGenericODE2NGsolve with simple example
- **notes: included in ACFtest.py**
- date resolved: **2022-04-27 10:23**, date raised: 2022-04-25

Version 1.2.49: resolved Issue 1057: **solutionInformation** (extension)
- description: replace new lines in solutionInformation when writing into coordinatesSolution file in order to preserve readable file structure
- date resolved: **2022-04-27 10:22**, date raised: 2022-04-27

Version 1.2.48: resolved Issue 1056: **MarkerSuperElementRigid** (extension)
- description: add check for node sizes used in marker
- date resolved: **2022-04-26 22:29**, date raised: 2022-04-26

Version 1.2.47: resolved BUG 1055: **NodeGenericODE2 error**
- description: NodeGenericODE2 raises system error CNodeODE2::GetVelocity: call illegal during solver initialize
- **notes: added GetVelocity and GetAcceleration to NodeGenericODE2, with risk that these functions are used unintended**
- date resolved: **2022-04-26 22:29**, date raised: 2022-04-26

Version 1.2.46: resolved Issue 1054: **NodeIndex, ...** (extension)
- description: add possibility to allow simple arithmetic operations +,-,*,-() for NodeIndex, MarkerIndex, etc.

- date resolved: **2022-04-26 22:29**, date raised: 2022-04-26

Version 1.2.45: resolved BUG 1050: **CreateLinearFEMObjectGenericODE2**

- description: FEM.CreateLinearFEMObjectGenericODE2 not working
- date resolved: **2022-04-25 17:34**, date raised: 2022-04-25

Version 1.2.44: resolved Issue 1046: **Vector3DList** (extension)

- description: add new structure Vector3DList; used to create list of 3D vectors and transfer into MainSystem mbs, e.g. in KinematicTree
- date resolved: **2022-04-25 08:40**, date raised: 2022-04-25

Version 1.2.43: resolved Issue 1045: **dispy cluster** (extension)

- description: fixed some bugs in ProcessParameterList (http_server) ; added some checks (if dispy is installed); cluster is used now iff clusterHostNames != [] and useMultiProcessing==True
- date resolved: **2022-04-23 17:09**, date raised: 2022-04-23

Version 1.2.42: resolved BUG 1044: **GetNodesOnLine**

- description: FEM.GetNodesOnLine raises exception
- date resolved: **2022-04-22 18:28**, date raised: 2022-04-22

Version 1.2.41: resolved BUG 1043: **serialRobotTest.py**

- description: class robotics.Robot computes wrong static torque compensation in function StaticTorques(HT)
- **notes: due to #0744 this bug has been magically resolved!**
- date resolved: **2022-04-21 15:15**, date raised: 2022-04-21

Version 1.2.40: resolved Issue 1042: **multithreaded compilation** (extension)

- description: enable setup.py with multithreaded compilation, using Monkey patch for compiler with multithreading library; enable parallel build with: python setup.py install –parallel
- date resolved: **2022-04-21 02:00**, date raised: 2022-04-21

Version 1.2.39: resolved Issue 0744: **CreateRedundantCoordinateMBS** (change)

- description: interchange joint markers, affecting the sign of the measured joint rotation angle, being now consistent with minimal coordinate formulations / kinematicTree; also interchange jointTorque0List/1List to be consistent with marker list
- **notes: adapted SerialRobotTestDH2.py and SerialRobotTSD.py in Example folders for corrected signs**
- date resolved: **2022-04-20 00:37**, date raised: 2021-09-01

Version 1.2.38: resolved Issue 1041: **Pluecker transforms** (change)

- description: correct Pluecker transform T66 functions, such as RotationTranslation2T66, etc. and introduce inverse functions for usage in Featherstone algorithm
- **notes: note that rotations in RotationX2T66, RotationY2T66, RotationZ2T66 are transposed/CHANGED as compared to previous version; T66toRotationTranslation is corrected/CHANGED and now is consistent with the backtransformation; InverseT66toRotationTranslation does the inverse operation (but also for rotation); RotationTranslation2T66 corrected/CHANGED; HT2T66 is corrected/CHANGED with inverse version HT2T66Inverse; rotations in Exudyn now consistent between T66, HT and other rotation matrices**
- date resolved: **2022-04-19 16:16**, date raised: 2022-04-19

Version 1.2.37: resolved Issue 1021: **twitter** (extension)

- description: add twitter account for exudyn: https://twitter.com/RExudyn
- **notes: Follow me!**
- date resolved: **2022-04-12 16:28**, date raised: 2022-03-31

Version 1.2.36: resolved Issue 1040: **contour bodies** (extension)

- description: show contour colors for GraphicsData added to bodies
- **notes: option turned on by default, but may slow down visualization for larger models; turn off in case of problems...**
- date resolved: **2022-04-10 17:21**, date raised: 2022-04-10

Version 1.2.35: resolved Issue 1039: **mesh edges** (extension)

- description: show mesh edges independently of visualization mode (edges on/off) - allowing to show mesh edges but not other graphics edges
- date resolved: **2022-04-10 11:42**, date raised: 2022-04-10

Version 1.2.34: resolved Issue 1036: **solution file** (change)

- description: fixed inconsistent information in line 3 of coordinatesSolutionFile: ODE2 coordinates
- date resolved: **2022-04-07 20:30**, date raised: 2022-04-07

Version 1.2.33: resolved Issue 1034: **solution and sensor files** (extension)

- description: add Python version to e.g. Exudyn version = 1.x.y Python3.9 / Python3.6(32bits) in coordinatesSolutionFile and sensor output files to identify exactly the versions and platform used for computation; check also Parameter and optimization output files for updated information
- date resolved: **2022-04-07 12:02**, date raised: 2022-04-07

Version 1.2.32: resolved Issue 1033: **InteractiveDialog** (extenison)

- description: also add tkInter DoubleVar for sliders to make bi-directional interaction simpler (but not needed)

- date resolved: **2022-04-04 21:26**, date raised: 2022-04-04

Version 1.2.31: resolved Issue 1031: **InteractiveDialog** (extension)
- description: extended with option addLabelStringVariables to be able to modify strings in text labels
- date resolved: **2022-04-04 17:20**, date raised: 2022-04-04

Version 1.2.30: resolved Issue 1030: **AVX2** (change)
- description: change both Python 3.6 32bit/46bit versions to compilation without AVX
- date resolved: **2022-04-04 15:39**, date raised: 2022-04-04

Version 1.2.29: resolved Issue 1024: **PERFORM_UNIT_TESTS** (change)
- description: set PERFORM_UNIT_TESTS for P3.7 instead of P3.6
- date resolved: **2022-04-04 15:38**, date raised: 2022-03-31

Version 1.2.28: **resolved BUG 1029**: **ContactFrictionCircleCable2D**
- description: computes wrong segment length internally, leading to wrong sticking position
- date resolved: **2022-04-04 12:14**, date raised: 2022-04-04

Version 1.2.27: **resolved BUG 1028**: **cnt in Render window**
- description: debug information cnt=.. shown in Render window
- date resolved: **2022-04-04 11:50**, date raised: 2022-04-04

Version 1.2.26: resolved Issue 1027: **RotationMatrix2EulerParameters** (change)
- description: RotationMatrix2EulerParameters computes Euler parameters with large deviations from unit norm in case of inaccurate rotation matrices; this may lead to failure of CheckPreAssembleConsistencies; resolved by adding normalization before returning Euler parameters
- date resolved: **2022-04-03 00:24**, date raised: 2022-04-03

Version 1.2.25: resolved Issue 1025: **build venv** (extension)
- description: switch to building windows purely on virtual conda environments, allowing to build all windows and linux versions in parallel
- date resolved: **2022-04-02 00:28**, date raised: 2022-04-02

Version 1.2.24: resolved Issue 1022: **virtual environments** (change)
- description: switch to virtual environments in anaconda for compilation on different platforms
- date resolved: **2022-04-02 00:28**, date raised: 2022-03-31

Version 1.2.23: resolved Issue 1023: **remove /Zi in MSVC** (change)
- description: remove compilation flag /Zi in setup.py which prevents from parallel runs of MSVC cl.exe
- date resolved: **2022-03-31 11:28**, date raised: 2022-03-31

Version 1.2.22: resolved Issue 1020: **robotics links** (docu)
- description: links to github are not resolved correctly for robotics module
- date resolved: **2022-03-30 10:47**, date raised: 2022-03-30

Version 1.2.21: resolved Issue 1019: **SpaceMouse** (extension)
- description: add functionality for 3D mouse / spacemouse by reading joystick inputs and interpret as 3D position and rotation data; add visualization flag interactive.useJoystickInput (default=True); deactivate this flag if your external device makes problems
- date resolved: **2022-03-29 14:37**, date raised: 2022-03-29

Version 1.2.20: resolved Issue 1017: **ContactFrictionCircleCable2D** (testing)
- description: check and test computation of sticking position segment length: undeformed versus deformed
- **notes: switched to reference length in computation of relative sticking position as given in theDoc; leads to improved results**
- date resolved: **2022-03-28 20:33**, date raised: 2022-03-28

Version 1.2.19: resolved Issue 1014: **SaveImage** (change)
- description: make window height divisible by 2 (skip one line if necessary)
- **notes: added alignment option for width and height; default options work well for ffmpeg conversion**
- date resolved: **2022-03-28 14:25**, date raised: 2022-03-28

Version 1.2.18: resolved Issue 1013: **TGA output** (change)
- description: switch from TGA to .png output using stb_image_write.h in GLFW
- **notes: added mode exportImages.saveImageFormat to chose between PNG and TGA - PNG with much smaller files!**
- date resolved: **2022-03-28 14:24**, date raised: 2022-03-28

Version 1.2.17: resolved Issue 1012: **GLFW** (change)
- description: switch to GLFW3.3.6 includes in order to be in line with AppleM1 version
- date resolved: **2022-03-28 12:03**, date raised: 2022-03-28

Version 1.2.16: resolved Issue 1011: **Apple M1** (change)
- description: use universal glfw libs for both Apple x86 and Apple arm M1

- date resolved: **2022-03-28 12:03**, date raised: 2022-03-28

Version 1.2.15: resolved Issue 1010: **autodiff** (change)
- description: extract autodiff as separate module
- **notes: now using AutomaticDifferentiation.h in Utilities**
- date resolved: **2022-03-28 11:56**, date raised: 2022-03-28

Version 1.2.14: **resolved BUG 1009**: **solutionViewer**
- description: does not load automatically due to change of coordinatesSolution filename ending
- **notes: changed loading of default file in SolutionViewer**
- date resolved: **2022-03-28 09:11**, date raised: 2022-03-28

Version 1.2.13: **resolved BUG 1007**: **sensor double values**
- description: sensor outputs two times for a single time step
- **notes: error occured due to automaticStepSize activated and call to ReduceStepSize even in case adaptiveStep=0; automaticStepSize now deactivated for solvers without step size control**
- date resolved: **2022-03-27 19:31**, date raised: 2022-03-24

Version 1.2.12: resolved Issue 1006: **ContactFrictionCircleCable2D** (change)
- description: LHS computation: exclude undefined state from sticking position computation
- date resolved: **2022-03-22 12:43**, date raised: 2022-03-22

Version 1.2.11: resolved Issue 1005: **PostNewton timer** (change)
- description: remove timer from timer structures and activate as special timer
- date resolved: **2022-03-22 12:30**, date raised: 2022-03-22

Version 1.2.10: resolved Issue 1004: **ContactFrictionCircleCable2D** (extension)
- description: add option usePointWiseNormals flag as an additional option to control the way forces are applied to cable
- date resolved: **2022-03-21 10:12**, date raised: 2022-03-21

Version 1.2.9: resolved Issue 1003: **setup.py** (extension)
- description: include manifest.in and readme.md in main
- date resolved: **2022-03-20 11:47**, date raised: 2022-03-20

Version 1.2.8: resolved Issue 1002: **pypi problems** (change)
- description: previous version marked beta
- date resolved: **2022-03-19 10:54**, date raised: 2022-03-19

Version 1.2.7: resolved Issue 0986: **ANCFCable2D** (docu)
- description: extend/finalize description - specifically for integration points and OutputVariable functions
- date resolved: **2022-03-18 23:09**, date raised: 2022-03-15

Version 1.2.6: resolved Issue 1001: **pypi** (change)
- description: finalized conversion of markup file for description at pypi.org: use only links to github, as pypi does not recognize .rst file in github format
- date resolved: **2022-03-18 20:54**, date raised: 2022-03-18

Version 1.2.5: resolved Issue 1000: **adjust version in docu** (docu)
- description: recompile
- date resolved: **2022-03-18 19:07**, date raised: 2022-03-18

Version 1.2.4: resolved Issue 0999: **pypi** (extension)
- description: improved versioning
- date resolved: **2022-03-18 18:28**, date raised: 2022-03-18

Version 1.2.3: resolved Issue 0998: **pypi** (extension)
- description: add description and tags
- date resolved: **2022-03-18 18:09**, date raised: 2022-03-18

Version 1.2.2: resolved Issue 0997: **pre-release version** (extension)
- description: add ".dev" tag in version and wheel name for pre-releases, allowing to distinguish on pypi for different versions
- date resolved: **2022-03-18 17:02**, date raised: 2022-03-18

Version 1.2.1: resolved Issue 0996: **pre-release** (extension)
- description: allow pre-releases to be uploaded on pypi, fetched with "pip install exudyn –pre"
- date resolved: **2022-03-18 15:52**, date raised: 2022-03-18

Version 1.2.0: resolved Issue 0995: **add to pypi index** (extension)
- description: add exudyn to pypi index; allows to use "pip install exudyn"
- date resolved: **2022-03-18 10:44**, date raised: 2022-03-18

Version 1.1.177: **resolved BUG 0994**: **ObjectContactFrictionCircleCable2D**
- description: forces on circle added up twice, because weighting factor not considered

- **notes: tested force on circle with static computation in belt drive**
- date resolved: **2022-03-18 08:30**, date raised: 2022-03-18

Version 1.1.176: resolved Issue 0880: **User function for ConnectorCoordinateVector** (extension)

- description: add user function for constraint and for jacobian; test with double pendulum made of masses
- date resolved: **2022-03-17 18:14**, date raised: 2022-01-24

Version 1.1.175: **resolved BUG 0993**: **ObjectConnectorDistance**

- description: activeConnector=False produces wrong jacobian
- date resolved: **2022-03-17 17:52**, date raised: 2022-03-17

Version 1.1.174: resolved Issue 0992: **PlotSensor** (extension)

- description: add PlotSensorDefaults function which allows to set default values for all subsequent PlotSensor calls
- **notes: use e.g. PlotSensorDefaults().fontSize=16 to change fontSize for all subsequent calls**
- date resolved: **2022-03-17 17:28**, date raised: 2022-03-17

Version 1.1.173: resolved Issue 0991: **velocityOffset** (extension)

- description: add velocity offset for SpringDamper and TorsionalSpringDamper, allowing simple controllers using offset and velocityOffset in preStepUser functions
- date resolved: **2022-03-16 23:51**, date raised: 2022-03-16

Version 1.1.172: resolved Issue 0989: **convergence problems** (docu)

- description: add specific section in theDoc - Exudyn Basics related to ways for resolving convergence problems
- date resolved: **2022-03-16 18:59**, date raised: 2022-03-16

Version 1.1.171: resolved Issue 0938: **ANCFCable2D** (extension)

- description: add drawing function for forces in normal direction with factor
- date resolved: **2022-03-15 20:33**, date raised: 2022-02-10

Version 1.1.170: resolved Issue 0933: **RigidBody** (extension)

- description: add accelerationLocal and angularAccelerationLocal output
- **notes: added to ObjectRigidBody and ObjectRigidBody2D**
- date resolved: **2022-03-15 20:14**, date raised: 2022-02-07

Version 1.1.169: resolved Issue 0967: **ANCFCable2D** (extension)

- description: add OutputVariables RotationMatrix, Rotation, AngularVelocity(Local) and AngularAcceleration
- **notes: Acceleration, AngularVelocity and AngularAcceleration currently only implemented for ANCFCable2D but not for ALEANCFCable2D**
- date resolved: **2022-03-15 20:01**, date raised: 2022-03-03

Version 1.1.168: resolved Issue 0966: **ANCFCable2D** (extension)

- description: add missing terms for OutputVariables and AccessFunctions to allow constraints that are at local position y!=0
- date resolved: **2022-03-15 19:54**, date raised: 2022-03-03

Version 1.1.167: resolved Issue 0983: **plot** (extension)

- description: add function to create plot-ready data from mbs.GetObjectOutputBody for a list of consecutive beams, e.g., axial force, displacement or curvature along axial reference coordinate
- **notes: added function DataArrayFromSensorList which allows to create data from a list of sensors**
- date resolved: **2022-03-14 20:41**, date raised: 2022-03-11

Version 1.1.166: resolved Issue 0982: **PlotSensor** (extentsion)

- description: add option to plot 2D arrays
- **notes: numpy arrays are used instead of sensorNumbers; these arrays must have the same format as data stored in sensor files; this data format does not create any labels**
- date resolved: **2022-03-14 16:24**, date raised: 2022-03-11

Version 1.1.165: resolved Issue 0985: **startOfStepState** (extension)

- description: initialize startOfStepState together with currentState in InitializeSolverInitialConditions(...) in order to be valid when sensors are written in initialization
- date resolved: **2022-03-14 13:07**, date raised: 2022-03-14

Version 1.1.164: resolved Issue 0981: **ObjectContactFrictionCircleCable2D** (extension)

- description: add OutputVariable functions for Coordinates (gap, slip), Coordinates_t (gap_t, slip_t), and contact and friction forces per segment (ForceLocal)
- date resolved: **2022-03-14 12:58**, date raised: 2022-03-11

Version 1.1.163: resolved Issue 0980: **renderer precision** (extension)

- description: add options for general precision in renderer: general.rendererPrecision as well as precision for colorbars in contour: contour.colorBarPrecision
- date resolved: **2022-03-11 14:05**, date raised: 2022-03-11

Version 1.1.162: resolved Issue 0979: **VisualizationSettings** (extension)

- description: VisualizationSettings.showContactForcesValues is added to show numerical values for contact forces

- date resolved: **2022-03-11 10:19**, date raised: 2022-03-11

Version 1.1.161: resolved Issue 0978: **ObjectANCFCable2D** (extension)
- description: add improved axial strain computation for reduced order integration
- **notes: works for axial strain and axial force if reducedAxialInterploation=True**
- date resolved: **2022-03-10 20:36**, date raised: 2022-03-10

Version 1.1.160: resolved Issue 0977: **ObjectANCFCable2D** (extension)
- description: add new mode useReducedOrderIntegration=2 with good performance/accuracy and exceptional representation of axial strains
- date resolved: **2022-03-10 20:08**, date raised: 2022-03-10

Version 1.1.159: resolved Issue 0976: **ContactCircleCable2D** (extension)
- description: add visualization flag showContactCircle; uses circleTiling*4 for tiling (from VisualizationSettings.general)!
- date resolved: **2022-03-10 14:15**, date raised: 2022-03-10

Version 1.1.158: resolved Issue 0975: **VisualizationSettings** (extension)
- description: added showContactForces and contactForcesFactor in contact; this flag is currently only available in ContactCircleCable2D
- date resolved: **2022-03-10 13:52**, date raised: 2022-03-10

Version 1.1.157: resolved Issue 0974: **VisualizationSettings** (change)
- description: moved contactPointsDefaultSize from connectors to contact; connectors.contactPointsDefaultSize is inactive from now!
- date resolved: **2022-03-10 13:50**, date raised: 2022-03-10

Version 1.1.156: resolved Issue 0953: **ContactFrictionCircleCable2D** (extension)
- description: add improved (static) friction model
- **notes: also added improved PostNewton switching strategies and changed initial values for NodeGenericData**
- date resolved: **2022-03-09 21:42**, date raised: 2022-02-25

Version 1.1.155: resolved Issue 0932: **ANCFCable2D** (extension)
- description: add velocityLocal + accelerationLocal output in axial/normal direction
- **notes: only velocityLocal added**
- date resolved: **2022-03-06 18:24**, date raised: 2022-02-07

Version 1.1.154: resolved Issue 0931: **ANCFCable2D** (extension)
- description: add acceleration as output variable
- **notes: only added for ANCF, but not for ALEANCF due to coupling terms with vALE**
- date resolved: **2022-03-06 18:24**, date raised: 2022-02-07

Version 1.1.153: resolved Issue 0970: **PlotSensor** (extension)
- description: PlotSensor allows to set fileCommentChar and fileDelimiterChar
- date resolved: **2022-03-04 13:27**, date raised: 2022-03-04

Version 1.1.152: resolved Issue 0969: **exudyn.plot** (extension)
- description: added method to convert output files from other codes; in particular plot.FileStripSpaces(...) can be used to strip leading / trailing spaces and remove double spaces
- date resolved: **2022-03-04 13:27**, date raised: 2022-03-04

Version 1.1.151: resolved Issue 0965: **ANCFCable2D** (extension)
- description: add option for strainIsRelativeToReference, which if set to 1. accounts for the reference geometry as the stress-free configuration; also works for ALE Cable2D
- date resolved: **2022-03-03 08:13**, date raised: 2022-03-03

Version 1.1.150: resolved Issue 0964: **CreateReevingCurve** (change)
- description: corrected sign of returned curvatures, now can be directly used for beam elements
- date resolved: **2022-03-02 17:20**, date raised: 2022-03-02

Version 1.1.149: **resolved BUG 0963**: **CreateReevingCurve**
- description: removeFirstLine=True not working: wrong case i==0 needs to be changed to i==1
- date resolved: **2022-03-02 15:25**, date raised: 2022-03-02

Version 1.1.148: resolved Issue 0962: **CreateReevingCurve** (change)
- description: adjust number of nodes in case of closed Curve (numberOfANCFnodes=20 gives 20 elements in this case)
- date resolved: **2022-03-02 15:25**, date raised: 2022-03-02

Version 1.1.147: resolved Issue 0961: **stepInformation** (change)
- description: changed flags in timeIntegration and staticSolver stepInformation: value of 1024 now causes output at every step; all values > 16 have been divided by two; 255=detailed overall output, 2047=detailed output every step; see SimulationSettings in theDoc
- date resolved: **2022-03-02 10:33**, date raised: 2022-03-02

Version 1.1.146: resolved Issue 0959: **coordinatesSolution** (change)

- description: change ending of coordinates solution files from .txt to .sol in case of binary files
- date resolved: **2022-03-02 10:14**, date raised: 2022-03-02

Version 1.1.145: resolved Issue 0960: **ObjectContactFrictionCircleCable2D** (change)

- description: move to ObjectContactFrictionCircleCable2DOld and improve new version
- date resolved: **2022-03-02 09:50**, date raised: 2022-03-02

Version 1.1.144: resolved Issue 0958: **LoadForceVector** (docu)

- description: add note to description in forces that user function values are available in sensors, but they are not updated during drawing
- date resolved: **2022-03-01 19:40**, date raised: 2022-02-28

Version 1.1.143: resolved Issue 0952: **rolling joints** (extension)

- description: ConnectorRollingDiscPenalty and JointRollingDisc now add a OutputVariable RotationMatrix, containing the J1 to global transformation
- date resolved: **2022-03-01 19:35**, date raised: 2022-02-25

Version 1.1.142: resolved Issue 0951: **Friction** (test)

- description: test LuGre friction model as ODE1 model versus position/history based model
- **notes: added lugreFrictionODE1.py as a demo showing the LuGre model based on a ODE1 user function**
- date resolved: **2022-03-01 19:34**, date raised: 2022-02-24

Version 1.1.141: resolved Issue 0956: **ProfileLinearAccelerationsList** (extension)

- description: add linear acceleration profile to robotics.motion, currently only allowing to create profile directly from accelerations
- date resolved: **2022-02-28 19:05**, date raised: 2022-02-28

Version 1.1.140: resolved Issue 0955: **robotics.motion** (change)

- description: change PTPprofile to BasicProfile
- date resolved: **2022-02-28 09:59**, date raised: 2022-02-28

Version 1.1.139: **resolved BUG 0950**: **JointRollingDisc**

- description: OutputVariable VelocityLocal is returned in global coordinates; description in theDoc is wrong; will be changed to outputVariable Velocity; local velocity will represent the slippage in special local joint J1 coordinates; see theDoc for updated functionality and outputs
- date resolved: **2022-02-25 14:32**, date raised: 2022-02-22

Version 1.1.138: **resolved BUG 0949**: **ConnectorRollingDiscPenalty**

- description: OutputVariable VelocityLocal local is returned in global coordinates; description in theDoc is wrong; will be changed to outputVariable Velocity; see theDoc for updated functionality and outputs
- date resolved: **2022-02-25 14:32**, date raised: 2022-02-22

Version 1.1.137: resolved Issue 0946: **sensors storeInternal** (extension)

- description: adapt most TestModels to store sensordata internally
- date resolved: **2022-02-21 00:43**, date raised: 2022-02-20

Version 1.1.136: resolved Issue 0947: **TestModels** (change)

- description: change most test models to use Sensors storeInternal mode; this avoids creating many files during TestSuite runs
- date resolved: **2022-02-20 20:40**, date raised: 2022-02-20

Version 1.1.135: resolved Issue 0921: **PlotSensor** (extension)

- description: add option to add subplots
- **notes: allows to create subplots, adjusting also the plot size using sizeInches; for examples see Examples/plotSensorExamples.py**
- date resolved: **2022-02-19 22:45**, date raised: 2022-02-02

Version 1.1.134: resolved Issue 0922: **PlotSensor** (extension)

- description: add option to add linewidth, markersize, markerStyles=["o",...], markerSizes=[], lineStyles=["-",...], colors=[..], markerDensity=...; lineWidths=[] allowing to plot a reduced number of markers on top of a line
- **notes: added many options for line and marker styles, check: colors, lineStyles, lineWidths, markerStyles, markerSizes, markerDensity**
- date resolved: **2022-02-19 22:41**, date raised: 2022-02-02

Version 1.1.133: resolved Issue 0920: **PlotSensor** (extension)

- description: add x-range and y-range for zoom
- **notes: added rangeX and rangeY options to specify range**
- date resolved: **2022-02-19 16:59**, date raised: 2022-02-02

Version 1.1.132: resolved Issue 0945: **PlotSensor** (extension)

- description: extend option for offsets, allowing sensor data to use as offset (e.g., loaded from file or from internal sensor data)
- **notes: see plotSensorExamples for some particular usage**
- date resolved: **2022-02-19 16:45**, date raised: 2022-02-19

Version 1.1.131: resolved Issue 0944: **PlotSensor** (extension)

- description: add argument labels, which can be string (for one sensor) or list of strings (according to number of sensors resp. components) representing the labels used in legend; if not provided, automatically generated legend is used
- date resolved: **2022-02-19 15:47**, date raised: 2022-02-19

Version 1.1.130: resolved Issue 0918: **Sensors store values internally** (extension)

- description: add option to store sensor values internally; using ResizableMatrix internally; rows added and matrix is automatically resized
- **notes: storeInternal boost the speed of writing sensor values, however, most of time is spent on computing sensor values, which typically about 2 seconds for 1e6 time steps per sensor; file write adds usually 3 seconds extra on that bill**
- date resolved: **2022-02-19 00:13**, date raised: 2022-02-02

Version 1.1.129: resolved Issue 0943: **mbs.GetSensorStoredData()** (extension)

- description: add MainSystem functionality to retrieve internally stored data in sensor; used, e.g., for PlotSensor to plot data without storing in files
- date resolved: **2022-02-19 00:10**, date raised: 2022-02-18

Version 1.1.128: resolved BUG 0942: **sensorsAppendToFile**

- description: appendToFile is used instead of sensorsAppendToFile for switching between append and replace operations for files
- date resolved: **2022-02-18 20:54**, date raised: 2022-02-18

Version 1.1.127: resolved Issue 0901: **SimulationSettings** (extension)

- description: improve type completion by adding py::init<...> functions for all subclasses
- **notes: not solvable in this simple way as type completion is not improved when adding this information**
- date resolved: **2022-02-18 20:30**, date raised: 2022-01-31

Version 1.1.126: resolved Issue 0941: **MotionInterpolator** (extension)

- description: mark as deprecated; instead, created motion submodule in robotics with class Trajectory; uses classes ProfileConstantAcceleration and ProfilePTP to construct piecewise profiles for trajectory; precomputes acceleration profiles in first step and thus is several factors faster in Python implementation; Trajectory converts to dict and can be printed in order to obtain key values of computed trajectories
- date resolved: **2022-02-18 16:31**, date raised: 2022-02-15

Version 1.1.125: resolved Issue 0940: **MotionInterpolator** (extension)

- description: add option to add trajectories defined by duration as well as by maxVelocity and maxAcceleration using synchronous PTP trajectory generation
- date resolved: **2022-02-15 12:38**, date raised: 2022-02-15

Version 1.1.124: resolved BUG 0939: **NodeRigidBody2D**

- description: does not correctly measure rotations (returns x-coordinate instead of angle)
- date resolved: **2022-02-15 09:58**, date raised: 2022-02-15

Version 1.1.123: resolved Issue 0745: **SensitivityAnalysis()** (extension)

- description: add functionality to processing, evaluating the sensitivities of certain sensor values w.r.t. parameters; same interface as ParameterVariation
- **notes: see processing.ComputeSensitivities(...)**
- date resolved: **2022-02-14 09:50**, date raised: 2021-09-03 (resolved by: P. Manzl)

Version 1.1.122: resolved BUG 0934: **exudyn.signal**

- description: conflicts with Python 3.8.8 and Python 3.9.7 (and possibly other) with internal Python signal package; change exudyn.signal to exudyn.signalProcessing
- date resolved: **2022-02-10 09:23**, date raised: 2022-02-10

Version 1.1.121: resolved Issue 0870: **binary output** (extension)

- description: add option to create binary solution files
- **notes: notes: use outputPrecision to switch between float and double - see there; speeds up file writing considerably and reduces file sizes; Integrated into LoadSolutionFile**
- date resolved: **2022-02-09 08:45**, date raised: 2022-01-18

Version 1.1.120: resolved Issue 0930: **LoadSolutionFile** (extension)

- description: extend function to check if binary file; if yes, switches to binary mode
- date resolved: **2022-02-09 08:44**, date raised: 2022-02-07

Version 1.1.119: resolved Issue 0929: **coordinatesSolution, sensors** (change)

- description: add version to coordinatesSolutionFile and sensor output files; should not affect current parsing of output files
- date resolved: **2022-02-07 00:02**, date raised: 2022-02-07

Version 1.1.118: resolved Issue 0878: **sort settings options** (docu)

- description: sort settings representation in Python by sorting dictionaries prior to writing inteface files; keep current sorting (grouping) in latex documentation
- **notes: type completion may change sorting afterwards**
- date resolved: **2022-02-06 21:57**, date raised: 2022-01-24

Version 1.1.117: resolved Issue 0928: **CPP UNIT TESTS** (testing)

- description: fail because of change of ConstSizeVector to use move assignment and move constructor
- **notes: adapted test to avoid move assignment**
- date resolved: **2022-02-06 00:04**, date raised: 2022-02-06

Version 1.1.116: resolved Issue 0919: **Minimize** (extension)

- description: add optimization with same interface as GenticOptimization but based on scipy.minimize
- **notes: added Minimize to processing; usage is nearly same as GeneticOptimization(...); example under Examples/minimize-Example.py**
- date resolved: **2022-02-04 15:45**, date raised: 2022-02-02 (resolved by: S. Holzinger)

Version 1.1.115: **resolved BUG 0924**: **GeneralContact**

- description: WARNING message raised in ANCF contact
- date resolved: **2022-02-03 12:01**, date raised: 2022-02-03

Version 1.1.114: resolved Issue 0923: **CreateReevingCurve** (extension)

- description: add function CreateReevingCurve(...) in exudyn.beams to create reeving system along circles, allows to create curve and nodes for ANCFCable2D elements created with PointsAndSlopes2ANCFCable2D(...); see Examples/reevingSystem.py
- date resolved: **2022-02-03 12:00**, date raised: 2022-02-02

Version 1.1.113: resolved Issue 0803: **PostNewton** (check)

- description: Check discontinuous iterations in combination with adaptive step (immediately reduces step size even if ignoreMaxSteps=True)
- **notes: did not further show up; possibly due to non-convergence**
- date resolved: **2022-02-02 08:20**, date raised: 2021-11-25

Version 1.1.112: resolved Issue 0906: **PlotSensor** (extension)

- description: add option componentsX to add x-components for figures, e.g., to plot y over x position, instead over time
- date resolved: **2022-02-01 18:34**, date raised: 2022-02-01

Version 1.1.111: resolved Issue 0864: **automatic example referencing** (fix)

- description: fix searching for examples, e.g., NodePoint as NodePoint(in order no to find NodePoint2D examples
- date resolved: **2022-01-31 23:01**, date raised: 2022-01-14

Version 1.1.110: **resolved BUG 0903**: **GeneralContact**

- description: autocomputed searchTree gives very large values and visualization not working
- **notes: ANCFCable2D bounding box computation had 1 wrong else case**
- date resolved: **2022-01-31 21:34**, date raised: 2022-01-31

Version 1.1.109: resolved Issue 0899: **GenerateCircularArcANCFCable2D** (extension)

- description: add function to create beams along circular arc
- date resolved: **2022-01-31 14:20**, date raised: 2022-01-30

Version 1.1.108: **resolved BUG 0902**: **GenerateStraightLineANCFCable2D**

- description: cableNodePositionList does not returns 3D vectors for nodes except first node
- date resolved: **2022-01-31 14:19**, date raised: 2022-01-31

Version 1.1.107: resolved Issue 0898: **GenerateStraightLineANCFCable2D** (extension)

- description: add option to use existing nodes in generation of beams
- date resolved: **2022-01-31 10:11**, date raised: 2022-01-30

Version 1.1.106: resolved Issue 0897: **add Python utility beams** (extension)

- description: add exudyn.beams utility module, containing helper functions for creating beams, etc.; move existing functions to this module
- date resolved: **2022-01-31 10:11**, date raised: 2022-01-30

Version 1.1.105: **resolved BUG 0900**: **GenerateStraightLineANCFCable2D**

- description: arguments vALE and ConstrainAleCoordinate are not implemented and need to be removed
- date resolved: **2022-01-30 23:32**, date raised: 2022-01-30

Version 1.1.104: resolved Issue 0896: **theDoc objects** (docu)

- description: sort objects into bodies, basic connectors, constraints and joints
- date resolved: **2022-01-30 23:12**, date raised: 2022-01-30

Version 1.1.103: resolved Issue 0895: **ConnectorGravity** (extension)

- description: add connector representing gravitational forces between heavy masses (planet, satellite, etc.)
- date resolved: **2022-01-30 19:05**, date raised: 2022-01-30

Version 1.1.102: resolved Issue 0894: **colors** (extension)

- description: added several colors in graphicsDataUtilities, added color4black, extended color4list to 16 colors
- date resolved: **2022-01-30 18:10**, date raised: 2022-01-30

Version 1.1.101: resolved Issue 0893: **GeneralContact** (change)

- description: changed flag introSpheresContact to sphereSphereContact; added flag for special mode to recycle last Post Newton step friction force
- date resolved: **2022-01-28 18:55**, date raised: 2022-01-28

Version 1.1.100: resolved Issue 0891: **adapt to Python3.9** (extension)

- description: make wheels and installers for Python3.9, running on Anaconda3-2021-11 Windows-x86_64
- date resolved: **2022-01-25 23:31**, date raised: 2022-01-25

Version 1.1.99: resolved Issue 0889: **PlotSensor** (testing)

- description: add test for PlotSensor, returning 1 if all plotting runs without crashing
- date resolved: **2022-01-25 19:03**, date raised: 2022-01-25

Version 1.1.98: resolved Issue 0890: **PlotSensor** (extension)

- description: add possibility to use filenames instead of sensor numbers, automatically loading these files
- date resolved: **2022-01-25 18:00**, date raised: 2022-01-25

Version 1.1.97: **resolved BUG 0887**: **iPython output stops**

- description: after a solver error, output of iPython stops
- **notes: changed to correct catching/throwing of Python and C++ exception types; outpur continues after solver error**
- date resolved: **2022-01-25 14:25**, date raised: 2022-01-25

Version 1.1.96: resolved Issue 0885: **GetInterpolatedSignalValue** (extension)

- description: add check in case that time values are distributed non-uniform; add tolerance as option
- date resolved: **2022-01-25 12:10**, date raised: 2022-01-25

Version 1.1.95: resolved Issue 0884: **PlotSensor** (extension)

- description: add optional title to plot
- date resolved: **2022-01-25 11:52**, date raised: 2022-01-25

Version 1.1.94: resolved Issue 0883: **PlotSensor** (extension)

- description: if sensorNumbers is scalar, components is a list, sensorNumbers is automatically adjusted; accepts now e.g. PlotSensor(mbs, 0, components=[0,1,2])
- date resolved: **2022-01-25 11:36**, date raised: 2022-01-25

Version 1.1.93: resolved Issue 0881: **PlotSensor** (extension)

- description: add option to apply factors and offsets to plotted signals; add option to add labels which appears at legend
- date resolved: **2022-01-25 11:36**, date raised: 2022-01-25

Version 1.1.92: resolved Issue 0882: **PlotSensor** (change)

- description: add option to use X, Y and Z components instead of 0, 1, 2 for Position, Displacement etc.; this option is enabled by default and changes the appearance -> set False, to preserve the old mode; component now also shown if only one curve plotted
- date resolved: **2022-01-25 11:19**, date raised: 2022-01-25

Version 1.1.91: resolved Issue 0879: **LoadSolutionFile** (extension)

- description: changed safeMode to loading sinle lines, which saves memory enormously, and added new options for loading huge files
- date resolved: **2022-01-24 14:53**, date raised: 2022-01-24

Version 1.1.90: **resolved BUG 0877**: **AddSensorRecorder**

- description: fails for scalar Sensors OutputVariableTypes (e.g. when measuring Rotation of TorsionalSpringDamper)
- **notes: added special scalar case**
- date resolved: **2022-01-21 09:17**, date raised: 2022-01-21

Version 1.1.89: resolved Issue 0876: **flush files** (extension)

- description: add option to flush solution and sensor files immediately after writing, simplifying the readout process; add option for large scale simulations, which are always flushed - helping for continuation of computations on supercomputers
- date resolved: **2022-01-20 14:01**, date raised: 2022-01-20

Version 1.1.88: resolved Issue 0875: **PlotSensor** (extension)

- description: fixed fontSize option and add options for minor/major ticks and SAVE figure to PlotSensor(...) using fileName=...
- date resolved: **2022-01-19 11:08**, date raised: 2022-01-19

Version 1.1.87: resolved Issue 0800: **GeneralContact ANCFCable** (extension)

- description: add ANCFCable2D to GeneralContact, enabling contact with planar spheres (cylinders)
- date resolved: **2022-01-18 18:54**, date raised: 2021-11-19

Version 1.1.86: resolved Issue 0866: **numberOfThreads** (extension)

- description: move simulationSettings.numberOfThreads into new section parallel in simulationSettings; remove comment [not implemented]
- date resolved: **2022-01-18 17:43**, date raised: 2022-01-15

Version 1.1.85: resolved Issue 0872: **preStepPyExecute** (change)

- description: remove preStepPyExecute from docu, time integration / static solver interface and from CSolverBase
- date resolved: **2022-01-18 16:57**, date raised: 2022-01-18

Version 1.1.84: resolved Issue 0874: **improved Newton restart** (change)

- description: added an additional Newton iteration after restarting modified Newton or when switching to full Newton; this reduces effects in generalized alpha and may improve behaviour with severe nonlinearities
- date resolved: **2022-01-18 13:44**, date raised: 2022-01-18

Version 1.1.83: resolved Issue 0869: **adaptiveStepRecoveryIterations** (change)

- description: add option to static and dynamic solvers to adjust max. Newton+disc. iterations prior to increase of step size; changed (previous internal) default value from 5 to 7
- date resolved: **2022-01-17 19:54**, date raised: 2022-01-17

Version 1.1.82: resolved Issue 0868: **solverSettings.stepInformation** (extension)

- description: change modes to add up binary flags; ADD several new options to show Newton iterations, jacobians, discontinuous iterations, per step or period; also add option to show output at every step
- date resolved: **2022-01-17 12:11**, date raised: 2022-01-17

Version 1.1.81: resolved Issue 0857: **GetInterpolatedSignalValue** (extension)

- description: new function to interpolate a numeric signal with time/data vectors at a certain time point
- date resolved: **2022-01-11 15:41**, date raised: 2022-01-11

Version 1.1.80: resolved Issue 0856: **IndexFromValue** (extension)

- description: function got faster mode in case of constant sampling rate
- date resolved: **2022-01-11 14:39**, date raised: 2022-01-11

Version 1.1.79: resolved Issue 0854: **Add LTG description** (docu)

- description: add section on local-to-global mapping of coordinates Section 2.3
- date resolved: **2022-01-08 12:09**, date raised: 2022-01-08

Version 1.1.78: resolved Issue 0849: **publications directory** (docu)

- description: create separate directory Examples/publications/ for publication data, Python files of numerical examples, etc.
- date resolved: **2022-01-06 16:32**, date raised: 2022-01-06

Version 1.1.77: resolved Issue 0846: **analytic jacobians** (extension)

- description: add analytic jacobians general functionality, realized for CartesianSpringDamper and CoordinateSpringDamper; deactivate with newton.numericalDifferentiation.forODE2connectors = False
- date resolved: **2021-12-23 11:07**, date raised: 2021-12-23

Version 1.1.76: resolved Issue 0770: **Marker jacobian derivative2** (extension)

- description: add jacobian derivative to most important connector markers
- **notes: implemented for markers except MarkerSuperElement; only MarkerPosition and MarkerRigidBody affected mostly; first tests show that jacobianDerivative agrees with numerical differentiation, BUT even increases iteration numbers**
- date resolved: **2021-12-22 21:21**, date raised: 2021-09-28

Version 1.1.75: resolved Issue 0842: **implement AccessFunctionType::JacobianTtimesVector_q** (extension)

- description: implement function needed for MarkerBody for objects ANCFCable, ObjectFFRF and ObjectFFRFreducedOrder; currently raising exception if used in this setup!
- date resolved: **2021-12-22 21:18**, date raised: 2021-12-22

Version 1.1.74: resolved Issue 0831: **Explicit solvers** (change)

- description: remove second ComputeODE2Acceleration() call and copy solutionODE2_tt from beginning of time step rk.stageDerivODE2_t[0], same for ODE1 variables; add flag but change that by default as it speeds up 2x; could also use information from previous step ...?
- date resolved: **2021-12-20 13:09**, date raised: 2021-12-15

Version 1.1.73: resolved Issue 0832: **explicit integrator** (extension)

- description: add flag timeintegration.explicitIntegration.computeEndOfStepAccelerations to compute end-of-step accelerations; this computation doubles the effort of explicit one-step-methods, particularly relevant in particles or contact simulations
- date resolved: **2021-12-17 12:38**, date raised: 2021-12-17

Version 1.1.72: resolved Issue 0450: **MT integration** (extension)

- description: fully integrate multithreading into system.cpp, vector.cpp and dense solver
- **notes: integrated into system, missing mass matrix and jacobian**
- date resolved: **2021-12-09 18:17**, date raised: 2020-09-16

Version 1.1.71: resolved Issue 0799: **GeneralContact description** (docu)

- description: add general section in theDoc for description of GeneralContact
- date resolved: **2021-12-09 12:36**, date raised: 2021-11-19

Version 1.1.70: resolved Issue 0793: **performance section** (docu)

- description: add performance and speedup section to theDoc, explaining most useful settings like modifiedNewton, EigenSparse, writeToFile, step size, numberOfThreads, constant mass matrix, etc.

- date resolved: **2021-12-09 12:36**, date raised: 2021-11-02

Version 1.1.69: resolved Issue 0823: **add trig-sphere friction tests** (extension)

- description: add simple test cases to check friction implementation
- date resolved: **2021-12-09 08:28**, date raised: 2021-12-06

Version 1.1.68: resolved Issue 0822: **shrink mesh** (extension)

- description: add method to shrink meshes using max distance to surface with normals; used for contact trig-sphere implementation
- **notes: currently slows down for larger meshes due to elimination of duplicate points!**
- date resolved: **2021-12-09 08:28**, date raised: 2021-12-06

Version 1.1.67: **resolved BUG 0827**: **GraphicsData cube**

- description: cubes has wrong numbering of nodes
- **notes: FIXED numbering in order to allow for correct normals needed in contact computation**
- date resolved: **2021-12-07 17:14**, date raised: 2021-12-07

Version 1.1.66: resolved Issue 0826: **GraphicsData normals** (extension)

- description: add normals to some GraphicsData cube objects
- date resolved: **2021-12-07 16:27**, date raised: 2021-12-07

Version 1.1.65: resolved Issue 0825: **GraphicsData add defaults** (extension)

- description: add some default values, especially to (center)point of GraphicsDataSphere, GraphicsDataCylinder, GraphicsDataOrthoCube, etc. in order to reduce interface sizes for objects added in the centerpoint [0,0,0]; thus some defaults were also necessary for radius or sizes!
- date resolved: **2021-12-07 14:41**, date raised: 2021-12-07

Version 1.1.64: resolved Issue 0824: **PlotSensor** (change)

- description: added serval NEW options, including: newFigure, colorCodeOffset, figureName and closeAll; NOTE that now PlotSensor by default opens a new figure!
- date resolved: **2021-12-07 12:11**, date raised: 2021-12-07

Version 1.1.63: **resolved BUG 0820**: **RigidBody visualization**

- description: normals in rigid body visualization transformed wrongly
- **notes: before, rotating bodies may have shown shading, now resolved**
- date resolved: **2021-12-05 17:32**, date raised: 2021-12-05

Version 1.1.62: resolved Issue 0816: **GraphicsData convert** (extension)

- description: add function GraphicsData2TrigsAndPoints(...) to convert graphicsData into triangles and points
- date resolved: **2021-12-05 15:43**, date raised: 2021-12-02

Version 1.1.61: resolved Issue 0814: **robotics.future, robotics.utilities, robotics.mobile** (extension)

- description: add special submodules for robotics functions; future contains currently developed submodules that will be available in future at a different location in robotics
- date resolved: **2021-12-05 15:43**, date raised: 2021-12-02

Version 1.1.60: resolved Issue 0819: **ObjectRigidBody** (change)

- description: correct HasConstantMassMatrix for Lie group nodes and COM=0
- date resolved: **2021-12-05 11:19**, date raised: 2021-12-05

Version 1.1.59: resolved Issue 0804: **GeneralContact TriangleMesh** (extension)

- description: Add rigid-body-marker based triangle mesh to GeneralContact
- date resolved: **2021-12-04 10:08**, date raised: 2021-11-25

Version 1.1.58: resolved Issue 0818: **MergeGraphicsDataTriangleList** (extension)

- description: now works if either both lists contain normals or both do not
- date resolved: **2021-12-03 20:15**, date raised: 2021-12-03

Version 1.1.57: resolved Issue 0817: **NodePointGround** (extension)

- description: add Node::Orientation to NodeType, such that it can also be used as a rigidBody node
- date resolved: **2021-12-03 14:34**, date raised: 2021-12-03

Version 1.1.56: resolved Issue 0815: **tCPU showing wrong time** (change)

- description: minor bug; time shown is since starting of iPython
- date resolved: **2021-12-02 17:48**, date raised: 2021-12-02

Version 1.1.55: resolved Issue 0813: **exudyn.robotics.special** (change)

- description: extend and move roboticsSpecial to robotics.special; add special robotics functionality likde manipulability
- date resolved: **2021-12-02 11:03**, date raised: 2021-12-02 (resolved by: M. Sereinig)

Version 1.1.54: resolved Issue 0618: **robotics submodule** (extension)

- description: Create robotics.special and robotics.mecanum or robotics.ros submodules with subdirectories
- date resolved: **2021-12-02 11:02**, date raised: 2021-03-23

Version 1.1.53: resolved Issue 0812: **Suppress warnings** (extension)

- description: add flag to globally suppress warnings
- **notes: use exudyn.SuppressWarnings(True) to turn off warnings**
- date resolved: **2021-12-01 08:50**, date raised: 2021-12-01

Version 1.1.52: resolved Issue 0811: **add default constructors** (change)

- description: add rule of five default constructors to SlimVector, SlimArray, ConstSizeVector and ConstSizeMatrix; remove mutable from data
- date resolved: **2021-11-28 21:56**, date raised: 2021-11-28

Version 1.1.51: resolved Issue 0810: **GeneralContact visualization** (extension)

- description: add visualization for searchtree (box) and bounding boxes
- **notes: use SC.visualizationSettings.contact to adjust the various options to visualize the contact search tree**
- date resolved: **2021-11-26 23:20**, date raised: 2021-11-26

Version 1.1.50: resolved Issue 0809: **ParameterVariation** (extension)

- description: Added a additional argument parameterFunctionData= to function ParameterVariation(). The argument parameterFunctionData can be used to make global data available inside the parameterFunction.
- date resolved: **2021-11-26 12:57**, date raised: 2021-11-26 (resolved by: S. Holzinger)

Version 1.1.49: resolved Issue 0808: **Performance test for GeneralContact** (testint)

- description: added test with sphere contact
- date resolved: **2021-11-26 10:49**, date raised: 2021-11-26

Version 1.1.48: resolved Issue 0802: **GeneralContact** (testing)

- description: add TestModel for Sphere-Sphere GeneralContact
- date resolved: **2021-11-25 22:58**, date raised: 2021-11-25

Version 1.1.47: resolved Issue 0807: **GeneralContact** (change)

- description: added new functions to initialize searchTree, searchTreeBox and frictionPairings which were previously in FinalizeContact(...)
- date resolved: **2021-11-25 22:16**, date raised: 2021-11-25

Version 1.1.46: resolved Issue 0806: **removed FinalizeContact** (change)

- description: removed this function, which is now automatically called in mbs.Assemble()
- date resolved: **2021-11-25 22:15**, date raised: 2021-11-25

Version 1.1.45: resolved Issue 0805: **AssembleSystemInitialize** (extension)

- description: add additional function inside mbs.Assemble() to initialize GeneralContact
- date resolved: **2021-11-25 22:14**, date raised: 2021-11-25

Version 1.1.44: resolved Issue 0779: **sensor recorder** (extension)

- description: add utilities function for recording signals internally in mbs; avoids writing to sensor files, which helps reducing overhead in ParameterVariation and GeneticOptimization
- **notes: available in Python utilities function AddSensorRecorder(...)**
- date resolved: **2021-11-25 11:10**, date raised: 2021-10-17

Version 1.1.43: resolved Issue 0801: **Box3D** (change)

- description: check Ubuntu20 warnings; replace Vector3D pmin with Real pmin[3] to avoid warnings and gain speedup
- date resolved: **2021-11-25 11:08**, date raised: 2021-11-23

Version 1.1.42: resolved Issue 0798: **Implicit GeneralContact** (extension)

- description: add ODE2RHS jacobian and PostNewton to GeneralContact
- date resolved: **2021-11-19 16:07**, date raised: 2021-11-19

Version 1.1.41: resolved Issue 0787: **GeneralContact** (extension)

- description: add contact object, directly in mbs, which allows different types of contact with efficient computation and search trees; start with simple spherical contact
- date resolved: **2021-11-19 16:06**, date raised: 2021-11-01

Version 1.1.40: resolved BUG 0784: **verboseMode**

- description: output of every step in verboseMode=1 after long time or if there are some very long lasting steps
- **notes: not fully clarified, but modified time when data is output; may occur in case of very long running iPython?**
- date resolved: **2021-11-14 23:23**, date raised: 2021-10-31

Version 1.1.39: resolved BUG 0790: **multithreading fails for user functions**

- description: build separate lists for objects and loads with user functions, excluded in MT evaluation
- date resolved: **2021-11-14 23:21**, date raised: 2021-11-02

Version 1.1.38: resolved BUG 0794: **error when switching from n to 1 threads**

- description: RuntimeError: TemporaryComputationDataArray::operator[]: index out of range caused when switching from numberOfThreads>1 to 1 thread

- date resolved: **2021-11-13 23:59**, date raised: 2021-11-04

Version 1.1.37: resolved Issue 0788: **multithreaded ODE2RHS and compute loads** (extension)

- description: use multithreaded computation for ODE2 RHS and for loads computation; use simulationSettings.numberOfThreads > 1 for multithreaded computation
- date resolved: **2021-11-13 23:59**, date raised: 2021-11-01

Version 1.1.36: resolved Issue 0796: **GL list GeneralContact** (extension)

- description: speed up visualization with GL list for spheres in GeneralContact
- date resolved: **2021-11-13 23:03**, date raised: 2021-11-10

Version 1.1.35: resolved Issue 0797: **itemInterface** (extension)

- description: add representation for item interface classes, using __repr__() = dict(self)
- date resolved: **2021-11-10 08:40**, date raised: 2021-11-10

Version 1.1.34: resolved Issue 0789: **constant mass matrix** (extension)

- description: do not recompute mass matrix if it is totally constant in implicit solver; add flag to solver options to force recompute
- date resolved: **2021-11-08 10:30**, date raised: 2021-11-02

Version 1.1.33: resolved Issue 0795: **add TCP/IP interface** (extension)

- description: add CreateTCPIPconnection and other functions to utilities for interconnection with other programs via TCP/IP
- date resolved: **2021-11-08 10:29**, date raised: 2021-11-08

Version 1.1.32: resolved Issue 0786: **TemporaryComputationDataArray** (extension)

- description: add array of TemporaryCompData for multithreaded computation
- date resolved: **2021-11-01 23:01**, date raised: 2021-11-01

Version 1.1.31: resolved Issue 0785: **ComputeSystemODE1RHS** (extension)

- description: add list of loads with ODE1 relevancy, otherwise all loads are computed even if there are no ODE1 coordinates
- **notes: added simple flag to avoid computation if no ODE1 coordinates available**
- date resolved: **2021-11-01 21:28**, date raised: 2021-11-01

Version 1.1.30: resolved Issue 0781: **add n-mass-oscillator** (example)

- description: add interactive example based on simulateInteractively for n-mass-oscillator with step and frequency excitation
- date resolved: **2021-10-28 16:44**, date raised: 2021-10-25

Version 1.1.29: resolved Issue 0747: **ComputeLinearizedSystem** (extension)

- description: add exudyn.ComputeLinearizedSystem similar to what is done in ComputeODEEigenvalues, returning M, K, D, ...
- date resolved: **2021-10-27 18:40**, date raised: 2021-09-03

Version 1.1.28: resolved Issue 0780: **enable close window button** (extension)

- description: close window button is now enabled, which stops current and following simulations until render window is restarted or SetRenderEngineStopFlag(False)
- **notes: if a simulation with renderer is quit (also ESCAPE button), then a further call to solver will be ignored until the simulation is reset, or SetRenderEngineStopFlag(False)**
- date resolved: **2021-10-21 09:31**, date raised: 2021-10-21

Version 1.1.27: resolved Issue 0778: **GenericODE2 FEM** (extension)

- description: add FEMinterface function for creation of ObjectGenericODE2 with linear FEM model and nonlinear FEM model (using NGsolve)
- date resolved: **2021-10-16 23:23**, date raised: 2021-10-16

Version 1.1.26: **resolved BUG 0776**: **MatrixContainer::SetWithSparseMatrixCSR**

- description: does not set number of columns and rows due to error in MatrixContainer::SetAllZero()
- date resolved: **2021-10-09 16:53**, date raised: 2021-10-08

Version 1.1.25: resolved Issue 0767: **sparse ObjectJacobianODE2** (extension)

- description: add dense and sparse interface to ObjectJacobianODE2
- date resolved: **2021-10-09 16:53**, date raised: 2021-09-27

Version 1.1.24: **resolved BUG 0775**: **ObjectGenericODE2, ObjectFFRFreducedOrder**

- description: visualization fails if outputVariable = None
- date resolved: **2021-10-08 17:26**, date raised: 2021-10-08

Version 1.1.23: resolved Issue 0773: **ImportMeshFromNGsolve** (extension)

- description: added option meshOrder which allows to use second order elements with meshOrder=2, leading to much higher accuracy of displacements and stresses
- date resolved: **2021-10-01 14:04**, date raised: 2021-10-01

Version 1.1.22: resolved Issue 0774: **ComputePostProcessingModesNGsolve** (extension)

- description: added improved functionality for ComputePostProcessingModes using NGsolve, speeding up computations by factor of 10

- date resolved: **2021-10-01 14:03**, date raised: 2021-10-01

Version 1.1.21: resolved Issue 0772: **compute HCB modes with NGsolve** (extension)
- description: add much faster computation function ComputeHurtyCraigBamptonModesNGsolve for computation of eigenmodes
- date resolved: **2021-10-01 14:03**, date raised: 2021-10-01

Version 1.1.20: resolved Issue 0771: **GeneticOptimization** (extension)
- description: add normal distribution and distanceFactorGenerations
- date resolved: **2021-09-29 17:38**, date raised: 2021-09-29

Version 1.1.19: resolved Issue 0769: **Marker jacobian derivative** (extension)
- description: add jacobian derivative to markers to allow analytical differentiation of connectors
- date resolved: **2021-09-28 18:57**, date raised: 2021-09-28

Version 1.1.18: resolved Issue 0768: **Newton.useNumericalDifferentiation** (change)
- description: change to Newton.numericalDifferentiation.forAE and Newton.numericalDifferentiation.forODE2; previous Newton.useNumericalDifferentiation only affected AE (algebraic equations) and should be changed now to Newton.numericalDifferentiation.forAE; default is False
- date resolved: **2021-09-27 15:11**, date raised: 2021-09-27

Version 1.1.17: resolved Issue 0612: **sparse object matrices** (extension)
- description: add sparse matrix computation mode for ComputeMassMatrix and for ObjectJacobianODE2; consider Lie algebra derivatives
- **notes: sparse ObjectJacobianODE2 computation not yet implemented and moved to issue767**
- date resolved: **2021-09-27 14:35**, date raised: 2021-03-21

Version 1.1.16: resolved Issue 0766: **ObjectGenericODE2** (change)
- description: change mass matrix, stiffnessmatrix, etc. types to PyMatrixContainer in order to accept dense and sparse matrices
- date resolved: **2021-09-27 14:32**, date raised: 2021-09-26

Version 1.1.15: resolved Issue 0765: **describe Python types** (docu)
- description: describe Python types such as NumpyMatrix or PyMatrixContainer in intro to objects, nodes, ...
- date resolved: **2021-09-27 14:32**, date raised: 2021-09-26

Version 1.1.14: resolved Issue 0764: **PyMatrixContainer** (extension)
- description: extend PyMatrixContainer to accept numpy.array or list of lists as input
- date resolved: **2021-09-26 23:35**, date raised: 2021-09-26

Version 1.1.13: resolved Issue 0763: **ComputeMassMatrix** (extension)
- description: add sparse mode with MatrixContainer
- date resolved: **2021-09-26 18:01**, date raised: 2021-09-26

Version 1.1.12: resolved Issue 0762: **MatrixBase** (performance)
- description: remove virtual from begin/end operators
- date resolved: **2021-09-26 17:36**, date raised: 2021-09-26

Version 1.1.11: **resolved BUG 0750**: **ANCF/ALE contour plot**
- description: contour plot not showing displacements or forces
- **notes: bug due to issue 760, which has been resolved now!**
- date resolved: **2021-09-24 08:48**, date raised: 2021-09-03

Version 1.1.10: resolved Issue 0761: **LinkedDataVectorBase** (extension)
- description: allowing SetNumberOfItems to make LinkedDataVectors smaller after linking
- date resolved: **2021-09-24 08:47**, date raised: 2021-09-24

Version 1.1.9: **resolved BUG 0760**: **contour plot**
- description: nodes in contour plot leading to exudyn crash due to ConstSizeVector decoupled from Vector
- date resolved: **2021-09-24 08:47**, date raised: 2021-09-24

Version 1.1.8: resolved Issue 0758: **FEM CMSObjectComputeNorm** (extension)
- description: add function into FEM to compute maximum stress / strain / etc for CMSObject (ObjectFFRFreducedOrder), using only the objectNumber as an input; options are outputVariableType=StressLocal, norm="" (Mises, L2norm, none), nodeNumbers=[] ... providing optional list of nodes to restrict the computation
- date resolved: **2021-09-23 19:25**, date raised: 2021-09-22

Version 1.1.7: resolved Issue 0757: **FEM GetNodePositionsMean** (extension)
- description: add function into FEMinterface to compute mean (average) position based on nodeNumbers (as list)
- date resolved: **2021-09-23 17:38**, date raised: 2021-09-22

Version 1.1.6: **resolved BUG 0759**: **contour plot**
- description: equivalent stress showing negative color bar values in contour plot
- **notes: contour plot with norm (component -1) showing only positive min and max values when tested**

- date resolved: **2021-09-23 17:19**, date raised: 2021-09-23

Version 1.1.5: resolved Issue 0756: **FEM MisesStress** (extension)

- description: add function that computes Mises stress from 6 stress components as obtained in stress sensor
- **notes: put into exudyn.physics module**
- date resolved: **2021-09-23 16:02**, date raised: 2021-09-22

Version 1.1.4: resolved Issue 0755: **GetKinematicTree66 in robotics** (extension)

- description: add function to export KinematicTree66 from Robotic class
- date resolved: **2021-09-22 18:06**, date raised: 2021-09-22

Version 1.1.3: resolved Issue 0754: **performance tests** (test)

- description: add automated performance tests for solver speed to determine significant drop of performance
- date resolved: **2021-09-22 10:10**, date raised: 2021-09-22

Version 1.1.2: resolved Issue 0620: **TorsionalSpringDamper** (extension)

- description: add torsional spring damper similar to SpringDamper, fixed on a single local axis of marker0, allowing to realize controllers and torques
- date resolved: **2021-09-16 10:45**, date raised: 2021-04-06

Version 1.1.1: resolved Issue 0679: **Renderer tkinter** (extension)

- description: add flag to disable calls to tkinter from Renderer, which is not possible if tkinter is already used for interactive dialogs. This allows to open visualizationSettings in AnimateModes and SolutionViewer
- date resolved: **2021-09-14 10:21**, date raised: 2021-05-14

Version 1.1.0: resolved BUG 0751: **SetMarkerParameter**

- description: causes internal error, because of wrong index check; workaround: use int(..) to cast marker index
- date resolved: **2021-09-10 16:22**, date raised: 2021-09-10

Version 1.0.295: resolved BUG 0749: **ObjectALEANCFCable2D**

- description: precomputed mass terms not computed accordingly; leads to crash if not static solution computed in advance
- date resolved: **2021-09-03 15:12**, date raised: 2021-09-03

Version 1.0.294: resolved Issue 0748: **Extend solver description** (docu)

- description: add some description for SolveStatic / SolveDynamic in solver chapter
- date resolved: **2021-09-03 13:54**, date raised: 2021-09-03

Version 1.0.293: resolved Issue 0504: **serialrobot** (extension)

- description: build completely from homogenouos transformations, COMs, inertia tensors, masses, axes, axesTypes
- **notes: done earlier**
- date resolved: **2021-08-22 11:12**, date raised: 2020-12-16

Version 1.0.292: resolved Issue 0540: **github** (extension)

- description: update README.rst file and make it similar to other packages (e.g. pydy)
- **notes: done earlier**
- date resolved: **2021-08-22 11:11**, date raised: 2021-01-09

Version 1.0.291: resolved Issue 0729: **README.rst** (docu)

- description: add .rst readme file containing gettingStarted, introduction, tutorial and other information
- date resolved: **2021-08-22 10:58**, date raised: 2021-08-05

Version 1.0.290: resolved Issue 0740: **robotics** (extension)

- description: extend Robot class for Modified DH parameters and general transformations; add transformations before and after joint axis
- date resolved: **2021-08-19 00:22**, date raised: 2021-08-18

Version 1.0.289: resolved BUG 0739: **robotics**

- description: CreateRedundantCoordinateMBS draws wrong axes
- date resolved: **2021-08-19 00:22**, date raised: 2021-08-18

Version 1.0.288: resolved BUG 0742: **robotics**

- description: Robot.JointHT computes LinkHT
- date resolved: **2021-08-18 23:31**, date raised: 2021-08-18

Version 1.0.287: resolved Issue 0741: **robotics** (change)

- description: add base and tool class to robotics to have more flexibility for future developments; replace toolHT to tool.HT, baseHT to base.HT; add tool.visualization and base.visualization
- date resolved: **2021-08-18 15:07**, date raised: 2021-08-18

Version 1.0.286: resolved Issue 0733: **ContactCoordinate** (extension)

- description: add recommendedStepSize to ContactCoordinate and find optimal solution with data variable from StartOfStep configuration; check if step size is permanently reduced with recommendedStepSize; check a way of an overall recommendedStepSize (with filter) or allow a single event not to change global step size

- date resolved: **2021-08-13 13:23**, date raised: 2021-08-12

Version 1.0.285: resolved Issue 0313: **Add user node ODE2** (extension)
- description: add user node with getposition, rotation, access functions
- **notes: not needed: GenericNodes can be used for that**
- date resolved: **2021-08-10 12:57**, date raised: 2020-01-10

Version 1.0.284: resolved Issue 0730: **RigidBody tutorial** (docu)
- description: add tutorial for rigid body with AddRigidBody(...), AddRevoluteJoint(...) functionalities
- date resolved: **2021-08-06 20:05**, date raised: 2021-08-05

Version 1.0.283: resolved Issue 0732: **DrawSystemGraph** (extension)
- description: improve visualization and return graph and other information
- date resolved: **2021-08-06 17:58**, date raised: 2021-08-06

Version 1.0.282: **resolved BUG 0731**: **AddRevoluteJoint**
- description: AddRevoluteJoint shows error in axis definition
- date resolved: **2021-08-05 13:40**, date raised: 2021-08-05

Version 1.0.281: resolved Issue 0704: **Optimization2** (optimize)
- description: add direct function to NodeRigidBody to retrieve essential data for rigid body EOM and MarkerRigidBody; add flag, if rotation matrix and other quantities needed
- **notes: still no optimization for Lie group nodes, which however have simpler matrices**
- date resolved: **2021-07-31 22:33**, date raised: 2021-07-04

Version 1.0.280: resolved Issue 0514: **general wheel** (extension)
- description: add general wheel model (with general rotation body); for mecanum wheel rolls
- **notes: implemented ObjectContactConvexRoll for general usage in Mecanum wheels and other applications**
- date resolved: **2021-07-31 21:20**, date raised: 2020-12-19 (resolved by: P. Manzl)

Version 1.0.279: resolved Issue 0727: **NodeRigidBody2D** (docu)
- description: Outputvariable Rotation gives 3D vector, but wrong description in DOCU
- **notes: additionally: rotation is now directly copied from rotation coordinate and is not recomputed from Tait-Bryan angles of rotation matrix**
- date resolved: **2021-07-31 21:18**, date raised: 2021-07-14

Version 1.0.278: resolved Issue 0726: **description of nodes** (docu)
- description: finish detailed description of 3D nodes and add rotation parameter description for Tait-Bryan in theory part
- date resolved: **2021-07-13 21:17**, date raised: 2021-07-13

Version 1.0.277: resolved Issue 0725: **unify description** (docu)
- description: unify notation for special vectors, e.g., reference point or local position; add unified abbreviations for ODE2, etc.
- date resolved: **2021-07-13 21:17**, date raised: 2021-07-13

Version 1.0.276: **resolved BUG 0724**: **Linux version**
- description: exudyn fails after import exudyn on Ubuntu18.04 and 20.04, showing error with RenderStateMachine selectionString
- **notes: version 276 tested with Ubuntu20.04, working again**
- date resolved: **2021-07-12 22:47**, date raised: 2021-07-12

Version 1.0.275: resolved Issue 0723: **SolutionViewer** (change)
- description: remove SolutionViewer from exudyn init file, as it causes problems if no tkinter or matplotlib installed
- **notes: use exudyn.interactive.SolutionViewer(...) instead**
- date resolved: **2021-07-12 20:23**, date raised: 2021-07-12

Version 1.0.274: **resolved BUG 0722**: **glfwGetWindowContentScale**
- description: function causes immediate crash on linux (UBUNTU) when importing exudyn
- **notes: added flag for linux compilation, excluding font scaling**
- date resolved: **2021-07-12 19:31**, date raised: 2021-07-12

Version 1.0.273: resolved Issue 0719: **Pybind11 2.6** (change)
- description: switch to Pybind11 2.6 in included C++ files
- date resolved: **2021-07-12 16:57**, date raised: 2021-07-12

Version 1.0.272: resolved Issue 0718: **Python3.8 FASTLINALG** (change)
- description: using now __FAST_EXUDYN_LINALG option, which excludes all range checks and other checks in arrays, matrices, etc.; leads usually to 30percent higher performance
- date resolved: **2021-07-12 16:57**, date raised: 2021-07-12

Version 1.0.271: **resolved BUG 0721**: **FEM.GetNodesOnLine(..)**
- description: fails because self. missing in call to GetNodesOnCylinder
- date resolved: **2021-07-12 16:35**, date raised: 2021-07-12

Version 1.0.270: resolved Issue 0717: **SC.StaticSolve, SC.TimeIntegrationSolve** (change)

- description: remove these deprecated functions from interface
- date resolved: **2021-07-12 15:33**, date raised: 2021-07-11

Version 1.0.269: resolved Issue 0669: **remove old solvers** (change)

- description: remove old static and dynamic solvers as they are not any more up to date with graphics interface
- date resolved: **2021-07-12 15:32**, date raised: 2021-05-10

Version 1.0.268: resolved Issue 0716: **SystemIsConsistent** (change)

- description: add checks for functions that may not be called if not SystemIsConsistent
- date resolved: **2021-07-11 17:30**, date raised: 2021-07-11

Version 1.0.267: resolved BUG 0714: **mbs.GetSensorValues**

- description: raises error for ObjectFFRFreducedOrder: ERROR: LinkedDataVectorBase(const VectorBase<T>&, Index), startPosition < 0
- **notes: caused when called before Assemble(); checks added in future**
- date resolved: **2021-07-11 17:08**, date raised: 2021-07-10

Version 1.0.266: resolved BUG 0715: **ObjectFFRFreducedOrder**

- description: OutputVariable Displacement includes localPosition, but should not
- **notes: GetMeshNodeLocalPosition included reference position twice**
- date resolved: **2021-07-11 16:33**, date raised: 2021-07-10

Version 1.0.265: resolved Issue 0706: **ConstSizeVector** (optimize)

- description: decouple ConstSizeVector and ConstSizeMatrix from Vector / Matrix and avoid virtual calls, erase all rule of 5 member functions, optimize algebra
- **notes: improved speed up to factor 2 for some items!**
- date resolved: **2021-07-11 15:06**, date raised: 2021-07-06

Version 1.0.264: resolved BUG 0713: **ObjectFFRFreducedOrder**

- description: GetOutputVariableSuperElement does not agree with types described in theDoc; object does not provide Displacement or Position, sensors return wrong values
- **notes: corrected C++ implementation and theDoc.pdf for OutputVariableTypesSuperElement**
- date resolved: **2021-07-09 20:53**, date raised: 2021-07-09

Version 1.0.263: resolved Issue 0712: **serialRobot** (change)

- description: improve speed of serial robot by transferring controllers from load userfunctions to mbs.SetPreStepUserFunction
- date resolved: **2021-07-09 13:28**, date raised: 2021-07-09

Version 1.0.262: resolved Issue 0711: **generator files** (change)

- description: changed backslash to slash in generator files such that they can also be executed on Linux and MacOS
- date resolved: **2021-07-09 12:18**, date raised: 2021-07-09

Version 1.0.261: resolved Issue 0699: **CMarkerBodyRigid::ComputeMarkerData** (optimize)

- description: implement optimized version for Rigid node and ObjectRigidBody and avoid repeated computation of rotation matrix, etc.
- date resolved: **2021-07-08 00:46**, date raised: 2021-07-01

Version 1.0.260: resolved BUG 0709: **Linux/MacOS compile error**

- description: compiler error caused by EXU::Square
- date resolved: **2021-07-07 19:11**, date raised: 2021-07-07

Version 1.0.259: resolved Issue 0708: **preprocessor flags** (change)

- description: move EXUDYN_RELEASE to preprocessor flags in setup.py
- date resolved: **2021-07-07 08:53**, date raised: 2021-07-07

Version 1.0.258: resolved Issue 0705: **Optimization3** (optimize)

- description: optimize ObjectRigidBody EOM, take Glocal columns instead numberOfRotationCoordinates, move rot_t into loop, etc.
- date resolved: **2021-07-06 23:04**, date raised: 2021-07-04

Version 1.0.257: resolved Issue 0703: **ComputeOrthonormalBasis** (change)

- description: changed rigidBodyUtilities function, which returns a list of basis vectors, into ComputeOrthonormalBasisVectors, while ComputeOrthonormalBasis now returns a rotation matrix
- date resolved: **2021-07-02 08:49**, date raised: 2021-07-02

Version 1.0.256: resolved Issue 0702: **AddPrismaticJoint** (extension)

- description: add convenient utility function to add prismatic joint based on 2 bodies, point and axis, doing all necessary work in background
- date resolved: **2021-07-02 08:49**, date raised: 2021-07-02

Version 1.0.255: resolved Issue 0701: **AddRevoluteJoint** (extension)

- description: add convenient utility function to add revolute joint based on 2 bodies, point and axis, doing all necessary work in background
- date resolved: **2021-07-02 08:49**, date raised: 2021-07-02

Version 1.0.254: resolved Issue 0700: **add links for utility functions** (docu)

- description: ADDED LINKS to Examples/ and TestModels/ example files at end of each python utility function and class, see Section 7
- date resolved: **2021-07-01 21:46**, date raised: 2021-07-01

Version 1.0.253: **resolved BUG 0697**: **GenericJoint**

- description: index2 equations not properly implemented for prismatic joints
- **notes: added second term for index2 case if joint position not constrained; TrapezoidalIndex2 solver now works if translational joint axes not constrained**
- date resolved: **2021-07-01 15:50**, date raised: 2021-07-01

Version 1.0.252: resolved Issue 0696: **add PrismaticJoint** (extension)

- description: add 3D prismatic joint with rotationMarker0/1 to adjust local coordinate systems and joint local x axis as the free axis of the joint
- date resolved: **2021-07-01 13:43**, date raised: 2021-07-01

Version 1.0.251: resolved Issue 0369: **add RevoluteJoint** (extension)

- description: add 3D revolute joint with rotationMarker0/1 to adjust joint coordinates and joint local z axis as rotation axis
- date resolved: **2021-07-01 13:43**, date raised: 2020-04-10

Version 1.0.250: resolved Issue 0695: **Solution functions** (change)

- description: remove exu and SC arguments from exudyn.utilities functions SetSolutionState(...), AnimateSolution(...); remove functoin SetVisualizationState(...): use SetSolutionState instead!
- date resolved: **2021-06-29 17:47**, date raised: 2021-06-29

Version 1.0.249: resolved Issue 0694: **SolutionViewer** (extension)

- description: add interactive dialog to view solution based on coordinateSolution.txt
- date resolved: **2021-06-29 17:31**, date raised: 2021-06-29

Version 1.0.248: resolved Issue 0693: **RigidBody user function** (extension)

- description: add test model for GenericODE2 user function based rigid body with Euler parameter and constraint
- date resolved: **2021-06-28 19:34**, date raised: 2021-06-28

Version 1.0.247: resolved Issue 0564: **NodeRigidBodyEP** (change)

- description: transfer EP constraint from object to node, for future application to 3D beams
- date resolved: **2021-06-28 19:34**, date raised: 2021-01-28

Version 1.0.246: resolved Issue 0413: **ConnectorCoordinateVectorUF** (extension)

- description: implement coordinate vector constraint user function; can be used as generic joint
- date resolved: **2021-06-28 16:19**, date raised: 2020-05-25

Version 1.0.245: resolved Issue 0691: **extend ConnectorCoordinateVector** (extension)

- description: extend ConnectorCoordinateVector for quadratic terms to be used as Euler Parameters constraint
- date resolved: **2021-06-27 23:29**, date raised: 2021-06-27

Version 1.0.244: resolved Issue 0690: **add MarkerNodeCoordinates** (extension)

- description: used for CoordinateVector constraint
- date resolved: **2021-06-27 23:29**, date raised: 2021-06-27

Version 1.0.243: resolved Issue 0689: **add itemIndex to user functions** (change)

- description: CHANGE OF userFunctions interface with additional itemIndex for ConnectorSpringDamper, ConnectorCartesianSpringDamper, ConnectorRigidBodySpringDamper, ConnectorCoordinate, ConnectorCoordinateVector, ConnectorJointGeneric; see theDoc for changes in the interface of these user functions and adapt your models!
- **notes: WARNING: Interface of user functions for ConnectorSpringDamper, ConnectorCartesianSpringDamper, ConnectorRigidBodySpringDamper, ConnectorCoordinate, ConnectorCoordinateVector, ConnectorJointGeneric changed!!!**
- date resolved: **2021-06-27 20:45**, date raised: 2021-06-27

Version 1.0.242: resolved Issue 0688: **ObjectGenericODE2** (change)

- description: extend ObjectGenericODE2 and ObjectGenericODE1 user functions for the item index to have access to nodes and other informaiton: WARNING: you need to adapt your existing user functions!
- **notes: WARNING: Interface of user functions for ObjectGenericODE2, ObjectFFRF... changed!!!**
- date resolved: **2021-06-27 20:44**, date raised: 2021-06-24

Version 1.0.241: resolved Issue 0687: **ObjectRigidBody** (extension)

- description: add output variable VelocityLocal to 2D and 3D rigid body objects
- date resolved: **2021-06-22 16:55**, date raised: 2021-06-22

Version 1.0.240: resolved Issue 0686: **eigenvalue solver** (extension)

- description: use ngsolve solver for eigenvalue computation speedup

- date resolved: **2021-06-14 15:21**, date raised: 2021-06-14

Version 1.0.239: resolved BUG 0684: **openGL.multisampling**
- description: Mac OS multisampling option in visualizationSettings crashes; ==> do not change this option under Mac OS
- **notes: excluded multisampling option for MacOS compilation**
- date resolved: **2021-05-30 12:02**, date raised: 2021-05-25

Version 1.0.238: resolved BUG 0681: **ObjectALEANCFCable2D**
- description: ObjectALEANCFCable2D position jacobian does not provide axially moving part for ObjectContactFrictionCircleCable2D, NEED TO BE ADDED
- **notes: had been already included in MarkerBodyCable2Dshape and works for roll contact**
- date resolved: **2021-05-30 12:01**, date raised: 2021-05-17

Version 1.0.237: resolved Issue 0685: **NodeRigidBody2D** (change)
- description: add same drawing as 3D nodes (with reference frame)
- date resolved: **2021-05-30 11:37**, date raised: 2021-05-30

Version 1.0.236: resolved Issue 0683: **SlidingJointRigid** (extension)
- description: Add functionality for rigid sliding joint or add a flag for sliding joint to do both options
- date resolved: **2021-05-20 09:30**, date raised: 2021-05-19

Version 1.0.235: resolved Issue 0682: **copy paste code from theDoc.pdf** (extension)
- description: changed ’ characters to enable copy/paste of code with quotes
- date resolved: **2021-05-19 08:54**, date raised: 2021-05-19

Version 1.0.234: resolved BUG 0680: **SetSystemState**
- description: mbs.systemData.SetSystemState does not set data coordinates
- date resolved: **2021-05-15 11:07**, date raised: 2021-05-15

Version 1.0.233: resolved Issue 0676: **single threaded renderer** (change)
- description: improve exu.DoRendererIdleTasks() and use it in all python function - AnimateModes, Interactive, etc.
- **notes: can now be also called in multithreaded renderer**
- date resolved: **2021-05-14 21:42**, date raised: 2021-05-12

Version 1.0.232: resolved BUG 0678: **mouseInteractiveExample**
- description: not running any more, check new Renderer functions
- date resolved: **2021-05-14 21:41**, date raised: 2021-05-14

Version 1.0.231: resolved Issue 0630: **HurtyCraigBampton** (extension)
- description: extend computation to work with 0 eigenmodes
- date resolved: **2021-05-12 23:51**, date raised: 2021-04-23

Version 1.0.230: resolved Issue 0642: **ComputeHurtyCraigBamptonModes** (extension)
- description: add possibility to add position only interfaces
- **notes: abandoned, because makes no sense with RBE2 modes**
- date resolved: **2021-05-12 23:35**, date raised: 2021-04-30

Version 1.0.229: resolved BUG 0674: **OutputVariable.StressLocal**
- description: norm of OutputVariable stresses does not work
- date resolved: **2021-05-12 23:21**, date raised: 2021-05-12

Version 1.0.228: resolved BUG 0456: **ObjectFFRF bug with GenericJoint**
- description: raises error: CSolverBase::SolveSteps CObjectSuperElement:GetAccessFunctionSuperElement: AngularVelocity_qt not implemented; cannot compute jacobian for orientation
- date resolved: **2021-05-12 22:27**, date raised: 2020-10-13

Version 1.0.226: resolved Issue 0100: **UPDATE Lest tests** (new feature)
- description: update lest tests (select C++ vs. python tests)
- date resolved: **2021-05-12 22:21**, date raised: 2019-04-01

Version 1.0.225: resolved Issue 0372: **add manual solver example** (extension)
- description: add manual for solver and example; also add new prestep user function
- **notes: resolved earlier**
- date resolved: **2021-05-12 22:20**, date raised: 2020-04-10

Version 1.0.224: resolved Issue 0384: **Solver interface** (extension)
- description: change solver interface such that it stores MainSystem/mbs for user functions; MainSolverXYZ and CSolverXYZ take MainSystem as argument in constructor
- date resolved: **2021-05-12 22:18**, date raised: 2020-05-06

Version 1.0.223: resolved Issue 0675: **PostProcessingModes** (extension)
- description: compute PostProcessingModes with multiprocessing
- date resolved: **2021-05-12 22:10**, date raised: 2021-05-12

Version 1.0.222: resolved Issue 0672: **right-mouse-dialog** (extension)

- description: add item indices to right mouse dialog
- date resolved: **2021-05-12 22:05**, date raised: 2021-05-11

Version 1.0.221: resolved Issue 0673: **FEM.PostProcessingModes** (extension)

- description: compute PostProcessingModes in FEMinterface with multiprocessing option
- date resolved: **2021-05-12 15:37**, date raised: 2021-05-12

Version 1.0.220: resolved Issue 0430: **stress modes FEMinterface** (extension)

- description: add into FEMinterface and allow storing that data
- **notes: resolved already earlier, see issue 623**
- date resolved: **2021-05-12 15:36**, date raised: 2020-07-01

Version 1.0.219: **resolved BUG 0631**: **ObjectFFRFreducedOrder**

- description: freefree eigenmodes and Hurty-Craig-Bampton modes do not converge to same results
- **notes: convergence for eigenmodes and HCB modes given, but differences due to HCB boundary sets, inconsistent initial conditions for MarkerSuperElementRigid; pure beam bending converges well**
- date resolved: **2021-05-12 14:05**, date raised: 2021-04-23

Version 1.0.218: **resolved BUG 0671**: **mbs.Reset() and SC.Reset()**

- description: reset MainSystem mbs and SystemContainer SC hangs; current SC is erronously stolen from renderer when another SC is deleted
- date resolved: **2021-05-11 10:55**, date raised: 2021-05-11

Version 1.0.217: resolved Issue 0670: **MacOS graphics support** (extension)

- description: add compatibility to MacOS in single-threaded graphics mode (tested with OS X 10.7)
- date resolved: **2021-05-10 22:34**, date raised: 2021-05-10

Version 1.0.216: resolved Issue 0647: **Single Thread Renderer** (extension)

- description: Implement single thread renderer version for MAC OS compatibility test
- date resolved: **2021-05-10 22:32**, date raised: 2021-04-30

Version 1.0.215: resolved Issue 0634: **Set visualization state** (extension)

- description: add thread-safe variant for updating the visualization state
- date resolved: **2021-05-10 18:32**, date raised: 2021-04-25

Version 1.0.214: resolved Issue 0668: **multiple mbs and SystemContainer support** (extension)

- description: adapt renderer and MainSystemContainer to work with multiple MainSystems (mbs) and SC instances at same time; add SC.AttachToRenderEnginer, SC.DetachFromRenderEngine
- date resolved: **2021-05-10 18:20**, date raised: 2021-05-10

Version 1.0.213: **resolved BUG 0665**: **StartRenderer()**

- description: without being in a render loop (e.g., SC.WaitForRenderEngineStopFlag()), the pure StartRenderer() crashes upon left mouse click
- date resolved: **2021-05-10 14:51**, date raised: 2021-05-05

Version 1.0.212: resolved Issue 0664: **right mouse** (change)

- description: add function to retrieve py::dict from items safely in python thread into temporary storage; check also other python calls to operate fully in main thread
- date resolved: **2021-05-10 14:51**, date raised: 2021-05-05

Version 1.0.211: **resolved BUG 0662**: **Render window**

- description: during open tkinter dialogs, the render window responds on keyboard or mouse input, which calls again python functions that hang up the system
- date resolved: **2021-05-10 14:51**, date raised: 2021-05-04

Version 1.0.210: resolved Issue 0633: **WaitAndLockSemaphoreIgnore** (check)

- description: check which atomic_flags are needed in C++ to make code threadsafe
- date resolved: **2021-05-10 14:51**, date raised: 2021-04-25

Version 1.0.209: resolved Issue 0667: **tkinter dialogs focus and on top** (extension)

- description: when opening tkinter dialogs - visualizationSettings, edit dialogs, help, ... - they immediately get focus and are on top
- date resolved: **2021-05-10 14:49**, date raised: 2021-05-10

Version 1.0.208: resolved Issue 0666: **make renderer Python and thread safe** (change)

- description: add strict separation between renderer (thread) and Python (thread); add rendererPythonInterface between both threads; left and right mouse clicks now safe to press; render window does not accept any input as long as tkinter window is open, but does not produce crashes any more
- date resolved: **2021-05-10 14:49**, date raised: 2021-05-10

Version 1.0.207: resolved Issue 0663: **help button** (docu)

- description: show "press h for help" as startup message for 10 seconds and sync help message with theDoc.pdf

- date resolved: **2021-05-05 10:02**, date raised: 2021-05-05

Version 1.0.206: resolved Issue 0653: **add right mouse edit dialog** (extension)

- description: open Edit dialog for item on right-mouse-press
- date resolved: **2021-05-04 20:58**, date raised: 2021-05-01

Version 1.0.205: resolved Issue 0652: **identify itemID under mouse coursor** (extension)

- description: identify object/node/... under mouse using unique color for itemID (left mouse button press)
- date resolved: **2021-05-04 12:49**, date raised: 2021-05-01

Version 1.0.204: resolved Issue 0637: **Python3.8 windows wheels** (extension)

- description: create Python3.8 windows wheels automatically
- date resolved: **2021-05-03 18:51**, date raised: 2021-04-26

Version 1.0.203: resolved Issue 0661: **add C++ unit tests** (extension)

- description: add C++ unit tests to Python3.6 64bits version and to testSuite. Changed initialization of all vector types to avoid errors of Vector(5), now allowing only Vector(4.) in constructors
- date resolved: **2021-05-03 18:50**, date raised: 2021-05-03

Version 1.0.202: resolved Issue 0660: **initializerList** (check)

- description: check if Vector is used with initializer list with one item - Vector(10), converting to std::vector or Vector(10)
- date resolved: **2021-05-03 18:50**, date raised: 2021-05-03

Version 1.0.201: resolved Issue 0659: **Troubleshooting** (docu)

- description: Add Trouble shooting section, treating common Python and solver errors to theDoc.pdf
- date resolved: **2021-05-03 10:18**, date raised: 2021-05-03

Version 1.0.200: resolved Issue 0650: **Highlight item#** (extension)

- description: Highlight item# for object/node/etc.; add to visualizationSettings (itemType, item#, colorHighlightItem, colorOtherItems), draw all other items in gray
- date resolved: **2021-05-03 01:18**, date raised: 2021-05-01

Version 1.0.199: resolved Issue 0649: **add ItemType** (extension)

- description: Add enum ItemType: Node, Object, ...
- date resolved: **2021-05-03 01:18**, date raised: 2021-05-01

Version 1.0.198: resolved Issue 0651: **add itemID to graphics objects** (extension)

- description: Add itemID (nodes, objects, markers, loads, sensors, in that order) to graphics objects (for right-mouse-press)
- date resolved: **2021-05-02 21:26**, date raised: 2021-05-01

Version 1.0.197: resolved Issue 0658: **add VisualizationSettings() interactive** (change)

- description: move visualizationSettings window functions keypressRotationStep, mouseMoveRotationFactor, keypressTranslationStep, zoomStepFactor to new substructure "interactive"
- **notes: adapt your models if you used these options!**
- date resolved: **2021-05-01 23:36**, date raised: 2021-05-01

Version 1.0.196: resolved Issue 0654: **coordinates sizes** (extension)

- description: add function ODE2Size(...), ODE1Size(...), SystemSize(...) to mbs.systemData to retrieve number of ODE2,ODE1,AE and Data coordinates for certain configurationType; only works after mbs.Assemble()
- date resolved: **2021-05-01 23:23**, date raised: 2021-05-01

Version 1.0.195: resolved Issue 0656: **mbs.systemData** (change)

- description: removed GetCurrentTime() and SetVisualizationTime(...) which have been marked as deprecated already
- **notes: Use GetTime(...) and SetTime(...) in mbs.systemData instead**
- date resolved: **2021-05-01 23:08**, date raised: 2021-05-01

Version 1.0.194: resolved Issue 0644: **solver messages** (extension)

- description: add solver message if not converged with helpful hints (especially if invert fails or newton fails)
- date resolved: **2021-05-01 02:31**, date raised: 2021-04-30

Version 1.0.193: resolved Issue 0349: **add causing row** (extension)

- description: output causing row/column (=coordinate) which leads to singular matrix; do this for Matrix.Invert as well as for SparseLU .info code; matrix class creates string with error message!
- date resolved: **2021-05-01 02:30**, date raised: 2020-03-02

Version 1.0.192: resolved Issue 0646: **jacobian singular** (extension)

- description: resolve singularities in general jacobian: resolves coordinates which are still free for static problems, but is marked as unsafe
- date resolved: **2021-05-01 02:29**, date raised: 2021-04-30

Version 1.0.191: resolved Issue 0645: **redundant constraints** (extension)

- description: resolve redundant constraints: add flag linearSolverSettings.ignoreSingularJacobian in SC.SimulationSettings() to ignore singular constraint jacobians

- date resolved: **2021-05-01 02:29**, date raised: 2021-04-30

Version 1.0.190: resolved Issue 0333: **node numbers with type** (extension)
- description: extend node/object/... numbers as python class with type information to check if item numbers are mixed illegally
- **notes: done already earlier, but still marked as unresolved**
- date resolved: **2021-04-30 17:04**, date raised: 2020-02-06

Version 1.0.189: resolved Issue 0641: **ObjectContactFrictionCircleCable2D** (docu)
- description: add description and figure for theory and computation
- date resolved: **2021-04-29 08:40**, date raised: 2021-04-29

Version 1.0.188: resolved BUG 0423: **fix MarkerSuperElementRigidBody**
- description: fix velocity level for MarkerSuperElementRigidBody (check constraint equations)
- **notes: fixed several errors and test examples work now on velocity level, but further checks are necessary**
- date resolved: **2021-04-27 18:24**, date raised: 2020-06-09

Version 1.0.187: resolved Issue 0635: **AnimateModes** (check)
- description: check, why animate modes has threading-conflicts; use std::cout to find issues
- **notes: resolved threading conflicts, but visualization state set inbetween graphics update, which needs to resolve #634**
- date resolved: **2021-04-27 18:20**, date raised: 2021-04-25

Version 1.0.186: resolved Issue 0640: **MarkerSuperElementRigid** (extension)
- description: remove referencePosition and add offset instead (to correct errors of midpoint due to small mesh-unsymmetries)
- **notes: CHANGED interface: MarkerSuperElementRigid does not have a referencePosition anymore, but adds a parameter offset**
- date resolved: **2021-04-27 18:18**, date raised: 2021-04-26

Version 1.0.185: resolved BUG 0639: **ObjectFFRFreducedOrder**
- description: incorrect AccessFunction AccessFunctionType::AngularVelocity_qt, missing correct reference point = midpoint for computation of rotation
- date resolved: **2021-04-26 21:47**, date raised: 2021-04-26

Version 1.0.184: resolved Issue 0638: **MarkerSuperElementRigid** (extension)
- description: use consistent reference point = midpoint for computation of rotation and use exponential Map for rotation matrix
- date resolved: **2021-04-26 21:47**, date raised: 2021-04-26

Version 1.0.183: resolved Issue 0636: **Python3.8** (extension)
- description: add python 3.8 compilation tests; resolve issues with __index__ method needed fore NodeIndex, MarkerIndex, etc.
- date resolved: **2021-04-26 16:25**, date raised: 2021-04-26

Version 1.0.182: resolved BUG 0629: **mesh visualization**
- description: visualization artifacts in larger FE meshes due to multithreading
- **notes: added flag threadSafeGraphicsUpdate to avoid thread conflicts between graphics and computation, which is by default set True and MAY SLOW DOWN your computation speed if True**
- date resolved: **2021-04-25 22:28**, date raised: 2021-04-22

Version 1.0.181: resolved Issue 0632: **CMS theory** (docu)
- description: add theory section for Hurty-Craig-Bampton modes and eigenmode computation
- date resolved: **2021-04-23 19:03**, date raised: 2021-04-23

Version 1.0.180: resolved BUG 0628: **FEMinterface.GetNodesOnCylinder**
- description: returns erroneous indices
- **notes: corrected indexing and add warnings for illegal node types**
- date resolved: **2021-04-22 22:59**, date raised: 2021-04-22

Version 1.0.179: resolved Issue 0616: **Craig-Bampton** (extension)
- description: add static modes (Hurty-Craig-Bampton) to computation of modes in CMSinterface
- **notes: implemented in FEMinterface.ComputeHurtyCraigBamptonModes(...)**
- date resolved: **2021-04-21 11:09**, date raised: 2021-03-21

Version 1.0.178: resolved Issue 0624: **norm in contour plots** (extension)
- description: show norm (of vectors or stresses) in contour plot, using special outputVariable component=-1
- date resolved: **2021-04-09 18:18**, date raised: 2021-04-09

Version 1.0.177: resolved Issue 0623: **postProcessingModes** (extension)
- description: add function to compute stress or strain modes for postprocessing, working for linear tetraherons (Tet4); see Examples/NGsolvePostProcessingStresses.py
- date resolved: **2021-04-09 15:46**, date raised: 2021-04-09

Version 1.0.176: resolved Issue 0424: **show modes** (extension)
- description: add feature to visualize eigenmodes, e.g. using ObjectFFRFreducedOrder and set one initialCoordinate nonzero

- date resolved: **2021-04-07 13:48**, date raised: 2020-06-12

Version 1.0.175: resolved Issue 0619: **Eigenmode visualizer** (extension)
- description: visualize eigenmodes with interactive tools with new function AnimateModes(...) to show eigenmodes of system or ObjectFFRFreducedOrder (see Section 7.11)
- date resolved: **2021-04-07 13:47**, date raised: 2021-03-30

Version 1.0.174: resolved Issue 0622: **InteractiveDialog** (extension)
- description: improved functionality of InteractiveDialog in interactive.py, specially for animating modes
- date resolved: **2021-04-07 12:20**, date raised: 2021-04-07

Version 1.0.173: **resolved BUG 0621**: **mbs.GetNodeODE2Index**
- description: Fails for NodeRigidBodyEP, because mix of AE and ODE2 variables
- **notes: added correct type check in MainSystem::PyGetNodeODE2Index**
- date resolved: **2021-04-07 09:12**, date raised: 2021-04-07

Version 1.0.172: resolved Issue 0403: **CMS C++** (extension)
- description: add ObjectFFRFreducedOrder (CMS) equations in C++ and clean up code
- date resolved: **2021-03-30 16:57**, date raised: 2020-05-21

Version 1.0.171: resolved Issue 0470: **geometrically exact beam2D** (extension)
- description: add to CPP
- date resolved: **2021-03-25 18:07**, date raised: 2020-11-21

Version 1.0.170: resolved Issue 0563: **ODE1Coordinate** (extension)
- description: add MarkerODE1Coordinate and extend LoadCoordinate for ODE1
- date resolved: **2021-03-25 07:43**, date raised: 2021-01-27

Version 1.0.169: resolved Issue 0615: **MarkerNodeCoordinate** (extension)
- description: add check in CSystem for valid coordinate numbers in MarkerNodeCoordinate
- date resolved: **2021-03-22 14:18**, date raised: 2021-03-21

Version 1.0.168: resolved Issue 0611: **adaptiveStep** (extension)
- description: add adaptiveStepIncrease, Decrease and RecoverySteps options to control behavior in case of discontinuous problems
- date resolved: **2021-03-21 00:21**, date raised: 2021-03-21

Version 1.0.167: resolved Issue 0603: **loadFactor** (change)
- description: exclude load factor for loads with user functions in static computations
- date resolved: **2021-03-20 23:24**, date raised: 2021-03-18

Version 1.0.166: resolved Issue 0610: **startOfStep** (extension)
- description: add access function for nodal coordinates at startOfStep configuration, used in mbs.GetNodeOutput(configuration = exu.ConfigurationType.startOfStep)
- date resolved: **2021-03-20 23:23**, date raised: 2021-03-20

Version 1.0.165: resolved Issue 0609: **SolveDynamic, SolveStatic** (change)
- description: store dynamicSolver and staticSolver in mbs.sys dictionary immediately after creation, which allows to use these structures in user functions during static or dynamic solution
- date resolved: **2021-03-20 23:23**, date raised: 2021-03-20

Version 1.0.164: resolved Issue 0607: **test recommendedStepSize** (test)
- description: test recommendedStepSize and PostNewtonUserFunction with simple elastic contact example
- date resolved: **2021-03-20 23:23**, date raised: 2021-03-20

Version 1.0.163: resolved Issue 0605: **UIndex, UReal** (extension)
- description: change all relevant unsigned quantities to UIndex and UReal, as well as Vectors of UReal and Arrays of UIndex
- date resolved: **2021-03-20 23:23**, date raised: 2021-03-19

Version 1.0.162: resolved Issue 0604: **UIndex check** (extension)
- description: add automatic check in item interface to check for correctness of UIndex and UReal quantities
- date resolved: **2021-03-20 23:23**, date raised: 2021-03-19

Version 1.0.161: resolved Issue 0337: **local quantities beam** (change)
- description: change beam output of Force, Torque, Curvature, Stress and Strain to ForceLocal, TorqueLocal, CurvatureLocal, StressLocal, and StrainLocal
- **notes: WARNING: you need to adapt force, torque and stress, strain and curvature output variables accordingly as they may have changed specifically for ANCF beams; adapt all your model files regarding Force, Torque, etc.**
- date resolved: **2021-03-20 23:22**, date raised: 2020-02-18

Version 1.0.160: resolved Issue 0139: **Index** (change)
- description: change Index to (signed) int and use UIndex in python interface for unsigned parameters

- **notes: ATTENTION: this change affects many routines. All TestSuite examples passed the change but there may still be open problems due to this major change.**
- date resolved: **2021-03-20 23:21**, date raised: 2019-05-20

Version 1.0.159: resolved Issue 0606: **resolve errors 32bit testsuite** (test)

- description: add extra tolerances for 32bit
- date resolved: **2021-03-20 23:19**, date raised: 2021-03-20

Version 1.0.158: resolved BUG 0575: **new genAlpha solver**

- description: new generalized alpha/implicit trapezoidal solver does not call solver user functions; Solution: derive CSolverImplicitSecondOrderTimeIntNew from CSolverBase and add user functions on top
- **notes: solved by removing old solver structure; new solver fully supports user functions now**
- date resolved: **2021-03-18 21:34**, date raised: 2021-02-08

Version 1.0.157: resolved Issue 0602: **PostNewton step size recommendation** (extension)

- description: add step recommendation as outcome of PostNewton function to improve contact and friction accuracy
- date resolved: **2021-03-18 21:33**, date raised: 2021-03-18

Version 1.0.156: resolved Issue 0601: **mbs.postNewtonUserFunction** (extension)

- description: add function PostNewton(...) to be called after step update (Newton or explicit step)
- date resolved: **2021-03-18 21:33**, date raised: 2021-03-18

Version 1.0.155: resolved Issue 0600: **ImplicitSecondOrderSolver** (cleanup)

- description: remove old solver
- date resolved: **2021-03-18 21:33**, date raised: 2021-03-18

Version 1.0.154: resolved Issue 0598: **rigidBodyUtilities** (extension)

- description: add G matrices for Rxyz (Tait-Bryan angles) and also time derivatives of G
- date resolved: **2021-03-18 17:05**, date raised: 2021-03-18

Version 1.0.153: resolved Issue 0594: **CMS rotations** (extension)

- description: test and extend CMS / ObjectFFRFreducedOrder object for other rotation parameterizations (Tait-Bryan and rotation vector/Lie group) such that they work with explicit codes
- date resolved: **2021-03-18 17:04**, date raised: 2021-02-24

Version 1.0.152: resolved Issue 0597: **ObjectRigidBody** (description)

- description: fix description of equations of motion (missing m) and add steps in derivation
- date resolved: **2021-03-18 08:22**, date raised: 2021-03-18

Version 1.0.151: resolved Issue 0283: **cylinder with hole** (new feature)

- description: add TriangleList for cylinder with hole
- **notes: not implemented, because it can be easily created with GraphicsDataSolidOfRevolution**
- date resolved: **2021-03-16 16:59**, date raised: 2019-12-07

Version 1.0.150: resolved Issue 0396: **description** (description)

- description: add latex description for ObjectFFRF
- date resolved: **2021-03-16 16:57**, date raised: 2020-05-16

Version 1.0.149: resolved Issue 0394: **description** (description)

- description: add latex description for ObjectSuperElement
- date resolved: **2021-03-16 16:57**, date raised: 2020-05-16

Version 1.0.148: resolved Issue 0595: **ObjectFFRFreducedOrder** (extension)

- description: add general nodeType to AddObjectFFRFreducedOrderWithUserFunctions
- date resolved: **2021-03-16 16:56**, date raised: 2021-03-14

Version 1.0.147: resolved Issue 0461: **ObjectRigidBody** (check)

- description: check discription of output variables
- date resolved: **2021-03-16 16:55**, date raised: 2020-11-12

Version 1.0.146: resolved Issue 0596: **GetRigidBodyNode** (extension)

- description: add function GetRigidBodyNode into rigidBodyUtilities, which returns a node item for an according node type, e.g., Euler parameters or rotation vector
- date resolved: **2021-03-14 14:43**, date raised: 2021-03-14

Version 1.0.145: resolved Issue 0583: **FFRF docu** (docu)

- description: add documentation for FFRF and FFRFreducedOrder (CMS) to documentation
- date resolved: **2021-03-01 12:15**, date raised: 2021-02-16

Version 1.0.144: resolved Issue 0455: **FEM help** (docu)

- description: add comments to ObjectFFRF (Tisserand frame!) and reduced that there are convenient helper functions in FEM, etc. for creating objects
- date resolved: **2021-02-24 21:21**, date raised: 2020-10-13

Version 1.0.143: resolved Issue 0593: **Add MacOS support** (change)

- description: make minor adjustments for MacOS to run setup.py
- date resolved: **2021-02-22 13:10**, date raised: 2021-02-22

Version 1.0.142: **resolved BUG 0592**: **StartRenderer**

- description: flag verbose=True not working
- date resolved: **2021-02-22 10:08**, date raised: 2021-02-22

Version 1.0.141: resolved Issue 0590: **SensorMarker visualization** (extension)

- description: add visualization for SensorMarker according to marker position
- date resolved: **2021-02-19 10:21**, date raised: 2021-02-19

Version 1.0.140: resolved Issue 0400: **add SensorMarker** (extension)

- description: add sensor for markers, restricting to position/velocity and rotation/angular velocity
- date resolved: **2021-02-18 19:15**, date raised: 2020-05-20

Version 1.0.139: resolved Issue 0585: **UserSensor** (extension)

- description: add user sensor, which enables the user to add any kind of sensor, specifically to combine several sensor outputs into one single sensor
- date resolved: **2021-02-18 19:14**, date raised: 2021-02-17

Version 1.0.138: resolved Issue 0589: **solver updateInitialValues** (change)

- description: update also initial coordinates in order to avoid jumps in accelerations when continuing simulation
- date resolved: **2021-02-18 18:15**, date raised: 2021-02-18

Version 1.0.137: resolved Issue 0588: **Solver file header** (change)

- description: move writing of solution file and sensor files headers from InitializeSolverPreChecks(...) to InitializeSolverInitialConditions(...) to avoid sensor evaluation for initial configuration
- date resolved: **2021-02-18 16:23**, date raised: 2021-02-18

Version 1.0.136: resolved Issue 0587: **ALEANCFCable2D** (fix)

- description: change mass proportional load to include force in direction of ALE coordinate
- date resolved: **2021-02-17 18:19**, date raised: 2021-02-17

Version 1.0.135: resolved Issue 0586: **GeneticOptimization** (fix)

- description: add special warnings and adaptations in order to catch cases where elitistRatio*populationSize < 1 and if distanceFactor >= 1
- date resolved: **2021-02-17 18:17**, date raised: 2021-02-17

Version 1.0.134: resolved Issue 0584: **parameter variation** (extension)

- description: add possibility to prescribe set of parameters using list, e.g., 'mass':[1,2,4,8] instead of of tuple which describes the range: 'mass':(0,6,4)
- date resolved: **2021-02-16 09:43**, date raised: 2021-02-16

Version 1.0.133: resolved Issue 0582: **optimization** (docu)

- description: add parameter variation and genetic optiization to documentation of solvers
- date resolved: **2021-02-16 08:21**, date raised: 2021-02-16

Version 1.0.132: **resolved BUG 0581**: **SetODE2Coordinates_tt**

- description: SetODE2Coordinates_tt writes to velocities instead of accelerations
- date resolved: **2021-02-14 11:12**, date raised: 2021-02-14

Version 1.0.131: resolved Issue 0580: **ParameterVariation** (extension)

- description: processing.ParameterVariation(...): write to resultsFile to show progress in resultsMonitor.py
- date resolved: **2021-02-11 17:49**, date raised: 2021-02-11

Version 1.0.130: resolved Issue 0576: **add ClearWorkspace** (extension)

- description: add ClearWorkspace() to basicUtilities which allows simple and save cleanup of globals() in python environment; recommended to be called at beginning of complex models
- date resolved: **2021-02-10 12:35**, date raised: 2021-02-08

Version 1.0.129: resolved Issue 0569: **contour text** (fix)

- description: add space to computation info text before contour plot text
- date resolved: **2021-02-10 12:35**, date raised: 2021-02-05

Version 1.0.128: resolved Issue 0568: **Renderer axes** (fix)

- description: use X(0), Y(1) and Z(2) for axes description to be compliant with Python indexing starting with 0 as well as contour components
- date resolved: **2021-02-10 12:35**, date raised: 2021-02-05

Version 1.0.127: resolved Issue 0579: **ClearWorkspace** (extension)

- description: add ClearWorkspacefunction to exudyn.basicUtilities, which allows to reset global variables in ipython; see example in function description

- date resolved: **2021-02-10 12:13**, date raised: 2021-02-10

Version 1.0.126: resolved Issue 0578: **SmoothStep** (extension)
- description: add SmoothStep function to exudyn.utilities, which produces a smooth step function using cosine
- date resolved: **2021-02-10 12:13**, date raised: 2021-02-10

Version 1.0.125: resolved Issue 0572: **void** (fix)
- description: redundant with issue 568
- date resolved: **2021-02-10 12:05**, date raised: 2021-02-05

Version 1.0.124: resolved Issue 0571: **void** (fix)
- description: redundant with issue 568
- date resolved: **2021-02-10 12:05**, date raised: 2021-02-05

Version 1.0.123: resolved Issue 0570: **void** (fix)
- description: redundant with issue 568
- date resolved: **2021-02-10 12:05**, date raised: 2021-02-05

Version 1.0.122: **resolved BUG 0577**: **PostNewtonStep**
- description: perform PostNewtonStep and PostDiscontinuousIterationStep only for active objects
- date resolved: **2021-02-09 14:00**, date raised: 2021-02-09

Version 1.0.121: resolved Issue 0355: **generalized alpha** (extension)
- description: implement version of Brüls and Arnold for generalized alpha solver
- **notes: WARNING: switched to new solver based on displacement increments (Arnold/Bruls,2007), which leads to DIFFERENT (but improved) RESULTS than previous dynamic implicit integrator; new implicit solver now works with ODE1 variables**
- date resolved: **2021-02-08 01:56**, date raised: 2020-03-08

Version 1.0.120: resolved Issue 0573: **merge solver documentation** (docu)
- description: merge docu on solver in EXUDYN overview and in solver section
- date resolved: **2021-02-07 17:33**, date raised: 2021-02-07

Version 1.0.119: resolved Issue 0567: **solvers description** (docu)
- description: extend description for equations of motion, explicit and implicit solvers
- date resolved: **2021-02-04 01:14**, date raised: 2021-02-03

Version 1.0.118: resolved Issue 0560: **Impl integrator ODE1** (extension)
- description: add ODE1 coordinates to implicit integrator
- date resolved: **2021-02-02 15:16**, date raised: 2021-01-26

Version 1.0.117: resolved Issue 0508: **implicit Lie group integrator** (extension)
- description: implement implicit index2/index3 Lie group integrator as python function
- **notes: cancelled, because will be directly done in C++**
- date resolved: **2021-02-02 15:15**, date raised: 2020-12-17

Version 1.0.116: resolved Issue 0566: **memory alloc cnt** (check)
- description: add control to check whether large amount of memory allocations happen during time integration+test suite
- date resolved: **2021-02-01 01:38**, date raised: 2021-01-29

Version 1.0.115: resolved Issue 0541: **objectODE1/2, constraint lists** (extension)
- description: add lists of ODE1 and ODE2 objects, constraints, etc. in cSystemData in order to speed up processing
- date resolved: **2021-02-01 01:38**, date raised: 2021-01-13

Version 1.0.114: resolved Issue 0557: **RK with constraints** (extension)
- description: add CoordinateConstraints to explict Runge-Kutta solvers
- **notes: only ground constraints included for now**
- date resolved: **2021-01-27 17:38**, date raised: 2021-01-26

Version 1.0.113: resolved Issue 0558: **Lie group tests** (test)
- description: add Lie group integrator simple tests
- date resolved: **2021-01-27 12:00**, date raised: 2021-01-26

Version 1.0.112: resolved Issue 0550: **GraphicsDataArrow** (extension)
- description: add arrow to graphicsDataUtilities
- **notes: also added GraphicsDataBasis(...) for drawing 3 orthogonal basis vectors, GraphicsDataCheckerBoard(...) for simple drawing of checker board background and MergeGraphicsDataTriangleList(...) for merging graphicsData triangle lists**
- date resolved: **2021-01-27 00:10**, date raised: 2021-01-17

Version 1.0.111: resolved Issue 0495: **add ODE1 coordinates** (extension)
- description: extend system (Jacobian, etc.) for ODE1 coordinates
- date resolved: **2021-01-26 13:21**, date raised: 2020-12-09

Version 1.0.110: resolved Issue 0556: **explicit RK tests** (test)

- description: add tests for explicit Runge Kutta integrators to TestModels
- date resolved: **2021-01-26 13:17**, date raised: 2021-01-25

Version 1.0.109: resolved Issue 0555: **explicit Lie group integrator** (extension)

- description: add existing Lie group integrator in C++
- date resolved: **2021-01-26 13:17**, date raised: 2021-01-25

Version 1.0.108: resolved Issue 0554: **explicit integrator** (extension)

- description: add explicit integrator with automatic step size control (DOPRI5, ODE23); checkout Section 11.3 for description of explicit solvers and Section 6.10.6 for available solver types
- date resolved: **2021-01-25 00:54**, date raised: 2021-01-24

Version 1.0.107: resolved Issue 0513: **add RK4 integrator** (extension)

- description: put existing python RK4 integrator into CPP
- date resolved: **2021-01-25 00:54**, date raised: 2020-12-19

Version 1.0.106: resolved Issue 0533: **ObjectGenericODE1** (extension)

- description: add object ObjectGenericODE1 for generic first order ODEs
- date resolved: **2021-01-21 17:27**, date raised: 2021-01-04

Version 1.0.105: resolved Issue 0553: **create physics submodule** (extension)

- description: create exudyn.physics and add friction functions
- date resolved: **2021-01-20 10:25**, date raised: 2021-01-20

Version 1.0.104: **resolved BUG 0552**: **DrawSystemGraph**

- description: does not work with RigidBodySpringDamper due to invalid GenericNodeData number
- date resolved: **2021-01-19 14:43**, date raised: 2021-01-19

Version 1.0.103: resolved Issue 0551: **InteractiveDialog** (extension)

- description: add interactive tkinter dialog and new submodule exudyn.interactive to interact with models
- date resolved: **2021-01-19 00:26**, date raised: 2021-01-19

Version 1.0.102: resolved Issue 0549: **show solver name and time** (extension)

- description: add options to show/hide solver name and current time in render window
- date resolved: **2021-01-17 17:42**, date raised: 2021-01-17

Version 1.0.101: **resolved BUG 0545**: **mbs.WaitForUserToContinue()**

- description: call to WaitForUserToContinue() does not always wait for keypress. Check StartRender() function and flag settings
- date resolved: **2021-01-17 16:55**, date raised: 2021-01-15

Version 1.0.100: **resolved BUG 0548**: **SolveDynamic/SolveStatic**

- description: option updateInitialValues not working
- **notes: corrected SetSystemState call**
- date resolved: **2021-01-17 00:08**, date raised: 2021-01-17

Version 1.0.99: resolved Issue 0542: **GeneticOptimization** (extension)

- description: store values continuously to file, add automatic loader and animate optimized values
- **notes: added resultsMonitor.py to exudyn module**
- date resolved: **2021-01-15 15:18**, date raised: 2021-01-13

Version 1.0.98: resolved Issue 0547: **realtimeSimulation** (extension)

- description: add factor for timeIntegration.simulateInRealtime
- date resolved: **2021-01-15 15:16**, date raised: 2021-01-15

Version 1.0.97: resolved Issue 0546: **add __version__ version to module** (extension)

- description: enable exudyn.__version__ as commonly used in other modules
- date resolved: **2021-01-15 15:05**, date raised: 2021-01-15

Version 1.0.96: resolved Issue 0544: **geneticOptimization** (extension)

- description: add optional argument resultsFile to specify a file for output of results data
- date resolved: **2021-01-14 23:17**, date raised: 2021-01-14

Version 1.0.95: resolved Issue 0543: **add results monitor** (extension)

- description: add resultsMonitor.py to be called from command line for doing continuous visualization of sensors and geneticOptimization output
- date resolved: **2021-01-14 23:08**, date raised: 2021-01-14

Version 1.0.94: resolved Issue 0532: **NodeGenericODE1** (extension)

- description: add node NodeGenericODE1 for arbitrary number of ODE1 coordinates
- date resolved: **2021-01-13 20:12**, date raised: 2021-01-04

Version 1.0.93: resolved Issue 0531: **solidExtrusion** (extension)

- description: add graphicsData for solid extrusion (prismatic) body; based on 2D point and segment list for flat boundaries

- date resolved: **2021-01-10 20:43**, date raised: 2021-01-04

Version 1.0.92: resolved Issue 0539: **RigidBodySpringDamper** (extension)
- description: add postNewtonStepUserFunction and dataCoordinates
- date resolved: **2021-01-08 14:34**, date raised: 2021-01-08

Version 1.0.91: **resolved BUG 0537**: **Render window**
- description: double calling of Render(...) function could happen from RunLoop/Render and glfwSetWindowRefreshCallback (set in InitCreateWindow(...)); check if semaphore would remove visualization problems
- **notes: added semaphore but FEM visualization anomalies are still there**
- date resolved: **2021-01-07 11:23**, date raised: 2021-01-06

Version 1.0.90: resolved Issue 0385: **add solver eigenvalues example** (extension)
- description: add Examples/solverFunctionsTestEigenvalues to test suite
- **notes: added ComputeODE2EigenvaluesTest.py using new functionality exudyn.solver.ComputeODE2Eigenvalues(...)**
- date resolved: **2021-01-07 11:08**, date raised: 2020-05-06

Version 1.0.89: resolved Issue 0494: **add all tests** (extension)
- description: add all TestModel/*.py to testsuite and also examples before making changes to solver
- date resolved: **2021-01-07 11:04**, date raised: 2020-12-09

Version 1.0.88: resolved Issue 0515: **user function connector** (extension)
- description: add forceUserFunction for ObjectConnectorRigidBodySpringDamper to enable User connector
- date resolved: **2021-01-07 11:03**, date raised: 2020-12-19

Version 1.0.87: resolved Issue 0506: **utilities docu** (extension)
- description: complete documentation for all exudyn python utilities and add unique headers for documentation
- date resolved: **2021-01-06 22:57**, date raised: 2020-12-16

Version 1.0.86: resolved Issue 0530: **solidOfRevolution** (extension)
- description: add graphicsData for solid of revoluation
- date resolved: **2021-01-06 00:31**, date raised: 2021-01-04

Version 1.0.85: resolved Issue 0536: **GraphicsDataPlane** (extension)
- description: add graphicsData for simple rectangular plane with option for checkerboard pattern
- date resolved: **2021-01-05 22:57**, date raised: 2021-01-05

Version 1.0.84: resolved Issue 0535: **alternating color for cylinder** (extension)
- description: add alternatingColor argument in GraphicsDataCylinder for visualization of rotation of cylindric bodies
- date resolved: **2021-01-05 21:46**, date raised: 2021-01-05

Version 1.0.83: resolved Issue 0529: **add MainSystem to userFunctions** (change)
- description: add MainSystem "mbs" to all user functions as first argument (WARNING: this changes ALL user functions!!!
- date resolved: **2021-01-05 14:31**, date raised: 2021-01-04

Version 1.0.82: resolved Issue 0447: **test examples** (check)
- description: test all examples with new index types
- date resolved: **2021-01-05 14:31**, date raised: 2020-09-09

Version 1.0.81: **resolved BUG 0534**: **PlotSensor**
- description: PlotSensor crashes for Load sensors because no outputVariableType exists
- **notes: added special treatment for load sensors**
- date resolved: **2021-01-04 20:11**, date raised: 2021-01-04

Version 1.0.80: resolved Issue 0527: **faces transparent** (extension)
- description: add general transparency flag for faces in visualizationSettings.openGL, switchable with button "T"; allows to make node/marker/object numbers visible
- date resolved: **2021-01-03 21:53**, date raised: 2021-01-03

Version 1.0.79: resolved Issue 0509: **ComputeODE2Eigenvalues** (test)
- description: add example in TestModels
- date resolved: **2021-01-03 10:44**, date raised: 2020-12-18

Version 1.0.78: resolved Issue 0528: **textured fonts** (extension)
- description: use TEXTURED based bitmap fonts based stored in glLists, allowing better interpolation, scalability (currently up to font size 64 without quality drop) and much higher performance
- date resolved: **2021-01-03 10:29**, date raised: 2021-01-03

Version 1.0.77: **resolved BUG 0526**: **solver.ComputeODE2Eigenvalues**
- description: dense mode returned unsorted eigenvalues==>add sorting
- date resolved: **2021-01-03 10:21**, date raised: 2021-01-03

Version 1.0.76: resolved Issue 0524: **interpret UTF8** (change)

- description: add conversion from UTF8 to unicode to interpret most central European characters + some important characters correctly (see Section 10.4)
- date resolved: **2021-01-02 20:13**, date raised: 2020-12-29

Version 1.0.75: resolved Issue 0525: **opengl write UTF8** (extension)

- description: use UTF8 encoding in opengl text output
- date resolved: **2020-12-29 21:00**, date raised: 2020-12-29

Version 1.0.74: resolved Issue 0523: **show version** (extension)

- description: show current version info in openGL window; can be switched off with showComputationInfo=False
- date resolved: **2020-12-27 01:33**, date raised: 2020-12-27

Version 1.0.73: resolved Issue 0522: **openGl issues** (fix)

- description: fix positioning problems of coordinate system and contour colorbar
- **notes: now using pixel coordinates for info texts and different font sizes**
- date resolved: **2020-12-27 01:31**, date raised: 2020-12-27

Version 1.0.72: resolved Issue 0521: **useWindowsDisplayScaleFactor** (extension)

- description: add new option useWindowsDisplayScaleFactor in visualizationSettings.general to include display scaling factor for font sizes
- date resolved: **2020-12-27 01:25**, date raised: 2020-12-27

Version 1.0.71: resolved Issue 0520: **useBitmapText** (extension)

- description: add new option useBitmapText in visualizationSettings.general to activate bitmap fonts (deprecated; now using textured fonts)
- date resolved: **2020-12-27 01:25**, date raised: 2020-12-27

Version 1.0.70: resolved Issue 0516: **add bitmap font** (extension)

- description: add font using OpenGL bitmaps to improve visibility of texts (deprecated, now using textured fonts)
- date resolved: **2020-12-27 01:23**, date raised: 2020-12-21

Version 1.0.69: resolved Issue 0519: **correct coordinateSystemSize** (change)

- description: set visualizationSettings.general.coordinateSystemSize relative to fontSize which scales better with larger screens
- date resolved: **2020-12-24 01:25**, date raised: 2020-12-24

Version 1.0.68: resolved Issue 0518: **windows display scaling** (extension)

- description: include windows display (screen) scaling into drawing of texts to increase visibility on high dpi screens
- **notes: added flag in visualizationSettings: general.useWindowsDisplayScaleFactor**
- date resolved: **2020-12-24 00:22**, date raised: 2020-12-24

Version 1.0.67: resolved Issue 0511: **GeneticOptimization** (test)

- description: add example in TestModels
- date resolved: **2020-12-19 23:31**, date raised: 2020-12-19

Version 1.0.66: resolved Issue 0510: **ParameterVariation** (test)

- description: add example in TestModels
- date resolved: **2020-12-19 23:31**, date raised: 2020-12-19

Version 1.0.65: resolved Issue 0502: **rigidbodyinertia** (docu)

- description: add description for rigidBodyUtilities class RigidBodyInertia
- date resolved: **2020-12-19 23:28**, date raised: 2020-12-14

Version 1.0.64: **resolved BUG 0512**: **testsuite**

- description: EXUDYN build date referred shows wrong path
- **notes: refer now to installed module**
- date resolved: **2020-12-19 00:44**, date raised: 2020-12-19

Version 1.0.63: resolved Issue 0507: **changes** (extension)

- description: incorporate resolved issues and bugs into theDoc.pdf
- date resolved: **2020-12-17**, date raised: 2020-12-16

Version 1.0.62: resolved Issue 0505: **rigidBodyUtilities** (extension)

- description: add description for RigidBodyInertia class
- date resolved: **2020-12-17**, date raised: 2020-12-16

Version 1.0.61: resolved Issue 0501: **geneticOptimization add crossover** (extension)

- description: added crossover and improved parameters for GeneticOptimization
- date resolved: **2020-12-14**, date raised: 2020-12-14

Version 1.0.60: resolved Issue 0497: **genetic algorithm** (check)

- description: check if stochsearch or genetic algorithm has simpler interface
- date resolved: **2020-12-14**, date raised: 2020-12-10

Version 1.0.59: resolved Issue 0500: **FilterSignal** (extension)

- description: put in signal module, make it working for 1D signals as well
- date resolved: **2020-12-11**, date raised: 2020-12-10

Version 1.0.58: resolved Issue 0492: **FEMinterface GetNodesInOrthoCube** (extension)

- description: add function which returns all nodes lying in cube aligned with global coordinate system, using [pMin, pMax], with tolerance
- date resolved: **2020-12-11**, date raised: 2020-12-08

Version 1.0.57: resolved Issue 0491: **FEMinterface GetNodesOnCylinder** (extension)

- description: add function which returns all nodes lying on specific cylinder, with tolerance
- date resolved: **2020-12-11**, date raised: 2020-12-08

Version 1.0.56: **resolved BUG 0499**: **key V gives error**

- description: keypress V for visualizationSettings dialog gives error
- date resolved: **2020-12-10**, date raised: 2020-12-10

Version 1.0.55: **resolved BUG 0498**: **SensorObject position**

- description: wrong position shown in sensor
- date resolved: **2020-12-10**, date raised: 2020-12-10

Version 1.0.54: resolved Issue 0484: **test DEAP** (test)

- description: test genetic optimization with DEAP
- **notes: too many parameters and too involved to simply include**
- date resolved: **2020-12-10**, date raised: 2020-12-04

Version 1.0.53: **resolved BUG 0493**: **CheckForValidFunction**

- description: modify / add this check to setParameters; additional if for setting this to 0
- date resolved: **2020-12-09**, date raised: 2020-12-09

Version 1.0.52: **resolved BUG 0490**: **keypress crash**

- description: find out causes for crash in keyPress user function; find way to deactivate the user function (set it to 0)
- date resolved: **2020-12-09**, date raised: 2020-12-07

Version 1.0.51: resolved Issue 0389: **MainSystem includes** (cleanup)

- description: put SystemIntegrity item checks into separate file, to reduce includig MainSystem into every .cpp item file
- date resolved: **2020-12-09**, date raised: 2020-05-13

Version 1.0.50: resolved Issue 0357: **solver flag prolong solution** (extension)

- description: add flag for solvers that current state at end of computation is set as initial state for next solving
- **notes: added into new python interface of solver**
- date resolved: **2020-12-09**, date raised: 2020-03-11

Version 1.0.49: resolved Issue 0489: **add gradient background** (extension)

- description: add according visualization.general option
- date resolved: **2020-12-06**, date raised: 2020-12-06

Version 1.0.48: **resolved BUG 0488**: **problem with coordinate sys**

- description: fix problems with drawing of coordinate system: text moves strangely and axes dissappear after rotation
- date resolved: **2020-12-06**, date raised: 2020-12-06

Version 1.0.47: resolved Issue 0487: **draw world basis** (extension)

- description: add option to draw coordinate system at origin (world basis)
- date resolved: **2020-12-06**, date raised: 2020-12-06

Version 1.0.46: resolved Issue 0486: **realtime** (extension)

- description: add flag to time integration to simulate in realtime
- date resolved: **2020-12-05**, date raised: 2020-12-05

Version 1.0.45: resolved Issue 0485: **mouse coordinates** (extension)

- description: store mouse coordinates in renderState
- date resolved: **2020-12-05**, date raised: 2020-12-05

Version 1.0.44: resolved Issue 0467: **mouse coordinates** (extension)

- description: show mouse coordinates in render window (without transformation)
- date resolved: **2020-12-05**, date raised: 2020-11-19

Version 1.0.43: resolved Issue 0325: **key callback** (extension)

- description: add key callback function into graphics module to enable interactive settings, etc.; transfer latin letters, SHIFT, CTRL, ALT, 0-9,A-Z,.,SPACE as ASCII code
- date resolved: **2020-12-05**, date raised: 2020-01-26

Version 1.0.42: resolved Issue 0460: **test accelerations** (test)

- description: test GetODE2Coordinates_tt, nodal accelerations and rigidbody2D/3D accelerations

- date resolved: **2020-12-04**, date raised: 2020-11-12

Version 1.0.41: resolved Issue 0482: **store model view** (extension)

- description: store renderState in exudyn.sys dictionary after exu.StopRenderer() for subsequent simulations
- date resolved: **2020-12-03**, date raised: 2020-12-03

Version 1.0.40: resolved Issue 0478: **link examples** (docu)

- description: automatically add links to examples in thedoc
- date resolved: **2020-12-03**, date raised: 2020-12-02

Version 1.0.39: resolved Issue 0477: **links in theDoc** (extension)

- description: add links between user functions, add labels to item sections
- date resolved: **2020-12-03**, date raised: 2020-12-02

Version 1.0.38: resolved Issue 0463: **accelerations** (extension)

- description: add accelerations Outputvariable to Super elements
- date resolved: **2020-12-03**, date raised: 2020-11-18

Version 1.0.37: resolved Issue 0481: **eigenvalue solver** (extension)

- description: add eigenvalue computation interface for mbs in python
- date resolved: **2020-12-02**, date raised: 2020-12-02

Version 1.0.36: resolved Issue 0480: **python solver** (extension)

- description: add solver interfaces in python for MainSolverStatic and MainSolverImplicitSecondOrder, helping to retrieve solver data and to make solvers accessible for users
- date resolved: **2020-12-02**, date raised: 2020-12-02

Version 1.0.35: resolved Issue 0479: **solver return** (extension)

- description: add return value to solvers and copy solver structures to mbs.sys variables after finishing
- **notes: added python interfaces and kept old cpp solvers**
- date resolved: **2020-12-02**, date raised: 2020-12-02

Version 1.0.34: resolved Issue 0469: **userfunctions** (extension)

- description: put user function generation in objectdefinition, with seperate U userfunction flag - this will automatically document the user function parameters (AND return values); this improves documentation and adds a unique interface in C++ using exception handling as well as GIL handling
- date resolved: **2020-12-02**, date raised: 2020-11-20

Version 1.0.33: resolved Issue 0458: **graphicsDataUserFunction** (docu)

- description: add example to docu in ObjectGround and GenericODE2 and add more accurate docu to ALL python user functions
- date resolved: **2020-12-02**, date raised: 2020-11-10

Version 1.0.32: resolved Issue 0428: **queue user functions** (extension)

- description: implement drawing user functions as global function similar to PyProcessQueue, in order to avoid messing up the CSystem and visualization modules
- date resolved: **2020-12-02**, date raised: 2020-06-26

Version 1.0.31: resolved Issue 0476: **add RequireVersion** (extension)

- description: functionality to allow to add a simple check to see if the installed version meets the requirements
- date resolved: **2020-11-30**, date raised: 2020-11-30

Version 1.0.30: resolved Issue 0475: **rolling disc ext** (extension)

- description: add force on ground and moving ground for ObjectJointRollingDisc
- **notes: needs to be tested further**
- date resolved: **2020-11-29**, date raised: 2020-11-26

Version 1.0.29: resolved Issue 0474: **auto compilation** (check)

- description: check automatic compilation; check version in wheels; check linux wheels
- **notes: linux wheels can not be built with admin rights**
- date resolved: **2020-11-29**, date raised: 2020-11-25

Version 1.0.28: resolved Issue 0473: **no glfw option** (extension)

- description: add simple option in setup.py to deactivate glfw both in setup.py as well as in C++ part
- date resolved: **2020-11-29**, date raised: 2020-11-25

Version 1.0.27: resolved Issue 0457: **GetVersionString** (extension)

- description: put into docu with pybindings
- date resolved: **2020-11-29**, date raised: 2020-11-07

Version 1.0.26: resolved Issue 0472: **examples in utilities** (extension)

- description: activate lstlisting for examples
- date resolved: **2020-11-25**, date raised: 2020-11-25

Version 1.0.25: resolved Issue 0468: **test WSL2** (test)

- description: test compilation on WSL2 - Windows subsystem for Linux
- **notes: WSL2 now used to automatically create linux wheels**
- date resolved: **2020-11-21**, date raised: 2020-11-19

Version 1.0.24: resolved BUG 0465: **SC.GetSystem(..)**

- description: raises RuntimeError: should return reference instead of copy
- date resolved: **2020-11-21**, date raised: 2020-11-18

Version 1.0.23: resolved Issue 0446: **NodeIndex in arrays** (check)

- description: use additional functionality to enable index type checks also in arrays, e.g., ArrayIndex of node numbers
- **notes: not needed for now**
- date resolved: **2020-11-21**, date raised: 2020-09-09

Version 1.0.22: resolved Issue 0383: **pybind11 submodule** (extension)

- description: used for advanced functions, not necessarily included in exudyn or make other module
- **notes: not needed for now**
- date resolved: **2020-11-21**, date raised: 2020-05-06

Version 1.0.21: resolved Issue 0191: **Newton lambda** (check)

- description: Check whether Newton can be implemented as lambda-function
- **notes: not needed for now**
- date resolved: **2020-11-21**, date raised: 2019-06-17

Version 1.0.20: resolved Issue 0466: **main/bin** (change)

- description: remove main/bin from github and from Tools folder
- date resolved: **2020-11-19**, date raised: 2020-11-19

Version 1.0.19: resolved Issue 0464: **processing module** (extension)

- description: create processing module for parameter variation and optimization using multiprocessing library
- date resolved: **2020-11-18**, date raised: 2020-11-18

Version 1.0.18: resolved Issue 0462: **AVX Celeron problems** (docu)

- description: add info to documentation - FAQ AND common problems and installation instructions that CPUs without AVX support only work with 32bit version
- date resolved: **2020-11-18**, date raised: 2020-11-16

Version 1.0.17: resolved Issue 0459: **lie group utilities** (extension)

- description: add documented lie group utilities to exudyn (python) lib
- date resolved: **2020-11-11**, date raised: 2020-11-11 (resolved by: S. Holzinger)

Version 1.0.16: resolved Issue 0454: **add item graph** (extension)

- description: add graph containing nodes, objects, etc.
- date resolved: **2020-10-08**, date raised: 2020-10-08

Version 1.0.15: resolved Issue 0453: **systemdata.NumberOfSensors** (extension)

- description: add access function for systemdata.NumberOfSensors()
- date resolved: **2020-10-08**, date raised: 2020-10-08

Version 1.0.14: resolved Issue 0449: **MT ngsolve** (extension)

- description: add NGsolve multithreading library (task manager)
- **notes: first tests made**
- date resolved: **2020-09-16**, date raised: 2020-09-15

Version 1.0.13: resolved Issue 0330: **correct ODE2RHS** (change)

- description: correct ODE2RHS to ODE2Terms in objects because it is left-hand-side
- **notes: changed object computation function from RHS to LHS, as it always computed the LHS (the system.cpp function ComputeODE2RHS then puts it to RHS)**
- date resolved: **2020-09-10**, date raised: 2020-02-03

Version 1.0.12: resolved Issue 0435: **check runtimeError** (check)

- description: check if exception runtimeerror works for all catch cases (test in windows?)
- date resolved: **2020-09-09**, date raised: 2020-07-21

Version 1.0.11: resolved Issue 0445: **remove GetItemByName()** (change)

- description: remove GetNodeByName, GetObjectByName, etc. from C++ interface; already disabled in python interface before
- date resolved: **2020-09-08**, date raised: 2020-09-08

Version 1.0.10: resolved Issue 0288: **Item::CallFunction** (change)

- description: Disable Item::CallFunction functionality from EXUDYN; either outputvariables can be used, or some functions are automatically created including the documentation
- **notes: already removed from python interface earlier**
- date resolved: **2020-09-08**, date raised: 2019-12-10

Version 1.0.9: resolved Issue 0443: **SensorObject** (warning)

- description: add error message, if sensorobject is used for a body (and check if SensorBody excepts object other than body
- **notes: added test for SensorObject if attached to body**
- date resolved: **2020-09-04**, date raised: 2020-09-03

Version 1.0.8: resolved BUG 0442: **difference MSC and setuptools**

- description: compilation with MSC and setuptools gives different results
- **notes: problem with VS2019 compilation of Eigen; resolved by removing VS2019 installation**
- date resolved: **2020-08-25**, date raised: 2020-08-24

Version 1.0.7: resolved Issue 0431: **auto create dirs** (extension)

- description: automatically create dictionaries if they do not exist
- date resolved: **2020-08-25**, date raised: 2020-07-01

Version 1.0.6: resolved Issue 0439: **setuptools** (extension)

- description: use setuptools for installation
- date resolved: **2020-08-17**, date raised: 2020-08-13

Version 1.0.5: resolved Issue 0381: **test pybind11_2020** (test)

- description: downloaded in Download folder
- date resolved: **2020-08-17**, date raised: 2020-05-06

Version 1.0.4: resolved Issue 0378: **setup tools** (extension)

- description: use setup tools to install EXUDYN on local user accounts; use installed python version to decide which version to install
- date resolved: **2020-08-17**, date raised: 2020-05-04

Version 1.0.3: resolved Issue 0441: **remove WorkingRelease path** (change)

- description: do not include WorkingRelease to sys.path any more, but require installation of modules
- date resolved: **2020-08-14**, date raised: 2020-08-14

Version 1.0.2: resolved Issue 0440: **exudyn package** (extension)

- description: make a package with sub .py files in exudyn package - requires renaming of C++ module
- date resolved: **2020-08-14**, date raised: 2020-08-13

Version 1.0.1: resolved Issue 0438: **UBUNTU** (extension)

- description: adapt setup.py and implementation for gcc and UBUNTU
- date resolved: **2020-08-13**, date raised: 2020-08-13

Version 1.0.0: resolved BUG 0434: **CheckSystemIntegrity**

- description: gives wrong node, marker, etc. numbers for some checks
- date resolved: **2020-07-20**, date raised: 2020-07-20

12.2 Known open bugs

- open **BUG 1889**: **symbolic**
  - description: GetLoad and similar functions do not work with symbolic user functions and raise TypeError: Object of type 'exudyn.exudynCPP.symbolic.UserFunction' is not an instance of 'function'; see also issue with mbs.GetDictionary()
  - date raised: 2024-10-11
- open **BUG 1772**: **item.GetDictionary**
  - description: item.GetDictionary not working for new user function interface with symbolic user function
  - date raised: 2024-02-04
- open **BUG 1639**: **SolveDynamic FFRF**
  - description: repeated call to mbs.SolveDynamic gives divergence; attributed to FFRFreducedOrder model; workaround uses repeated build of model before calling solver again; may be related to FFRF or MarkerSuperElement-internal variables
  - date raised: 2023-07-10
- open **BUG 1085**: **GeneralContact**

  - description: generalContactFrictionTests.py gives considerably different results after t=0.05 seconds between Windows and linux compiled version; may be caused by some initialization problems (bugs...); needs further tests
  - date raised: 2022-05-11

- open **BUG 1048**: **sse2neon.h**
  - description: on Apple, sse2neon.h is missing (include from github) and compilation fails; check if this only happens on M1 and change include modes of sse2neon.h; add this file to python setup.py for other cases
  - date raised: 2022-04-25

- open **BUG 0830**: **PostNewton**
  - description: PostNewton missing in explicit solvers; add warning or add after single steps (but exclude in contact computation!)
  - date raised: 2021-12-15

- open **BUG 0738**: **ObjectContactCoordinate**
  - description: modified Newton does not work, no Jacobian update computed when switching
  - date raised: 2021-08-13

- open **BUG 0448**: **ObjectGenericODE2 bug**
  - description: ObjectGenericODE2 crashes without message when initialized with invalid node numbers
  - date raised: 2020-09-09

References

[1] M. Arnold and O. Brüls. Convergence of the generalized-α scheme for constrained mechanical systems. *Multibody System Dynamics*, 85:187–202, 2007.

[2] M. C. C. Bampton and R. R. Craig Jr. Coupling of substructures for dynamic analyses. *American Institute of Aeronautics and Astronautics Journal*, 6(7):1313–1319, 1968.

[3] O. A. Bauchau. *Flexible Multibody Dynamics*. Springer New York, Philadelphia, 2011.

[4] M. Berzeri and A. Shabana. Development of simple models for the elastic forces in the absolute nodal co-ordinate formulation. *Journal of Sound and Vibration*, 235(4):539–565, 2000.

[5] O. Brüls, M. Arnold, and A. Cardona. Two Lie group formulations for dynamic multibody systems with large rotations. In *Proceedings of IDETC/MSNDC 2011, ASME 2011 International Design Engineering Technical Conferences*, Washington, USA, 2011.

[6] A. S. Carvalho and J. M. Martins. Exact restitution and generalizations for the Hunt Crossley contact model. *Mechanism and Machine Theory*, 139:174–194, 2019.

[7] P. Chiacchio and M. Concilio. The dynamic manipulability ellipsoid for redundant manipulators. *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No.98CH36146)*, 1:95–100 vol.1, 1998.

[8] J. Chung and G. M. Hulbert. A Time Integration Algorithm for Structural Dynamics With Improved Numerical Dissipation: The Generalized-α Method. *Journal of Applied Mechanics*, 60(2):371, 1993.

[9] P. Corke. *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*. Springer Publishing Company, Incorporated, 1st edition, 2013.

[10] C. C. de Wit, H. Olsson, K. Aström, and P. Lischinsky. Dynamic friction models and control design. In *1993 American Control Conference*, pages 1920–1926, 1993.

[11] R. Eder and J. Gerstmayr. Special genetic identification algorithm with smoothing in the frequency domain. *Advances in Engineering Software*, 70:113–122, 2014.

[12] R. Featherstone, editor. *Rigid Body Dynamics Algorithms*. Springer, 2008.

[13] P. Flores, J. Ambrósio, J. Pimenta Claro, and H. Lankarani. *Kinematics and Dynamics of Multibody Systems with Imperfect Joints*. Springer Berlin, 2008.

[14] P. Gangl, K. Sturm, M. Neunteufel, and J. Schöberl. Fully and semi-automated shape differentiation in ngsolve, 2020.

[15] C. Gaz, M. Cognetti, A. Oliva, P. Robuffo Giordano, and A. De Luca. Dynamic identification of the franka emika panda robot with retrieval of feasible parameters using penalty-based optimization. *IEEE Robotics and Automation Letters*, 4(4):4147–4154, 2019.

[16] M. Geradin and A. Cardona. *Flexible Multibody Dynamics*. John Wiley & Sons, 2001.

[17] J. Gerstmayr. HOTINT – A C++ Environment for the simulation of multibody dynamics systems and finite elements. In K. Arczewski, J. Fraczek, and M. Wojtyra, editors, *Proceedings of the Multibody Dynamics 2009 Eccomas Thematic Conference*, 2009.

[18] J. Gerstmayr. Exudyn github repository. https://github.com/jgerstmayr/EXUDYN (accessed on March 8, 2023), 2023.

[19] J. Gerstmayr. Exudyn – a C++-based Python package for flexible multibody systems. *Multibody System Dynamics*, 60:533–561, 2024.

[20] J. Gerstmayr, A. Dorninger, R. Eder, P. Gruber, D. Reischl, M. Saxinger, M. Schörgenhumer, A. Humer, K. Nachbagauer, A. Pechstein, and Y. Vetyukov. HOTINT: A Script Language Based Framework for the Simulation of Multibody Dynamics Systems. In *9th International Conference on Multibody Systems, Nonlinear Dynamics, and Control, International Design Engineering Technical Conferences and Computers and Information in Engineering Conference (ASME IDETC 2013)*, 2013.

[21] J. Gerstmayr and S. Holzinger. Explicit time integration of multibody systems modelled with three rotation parameters. In *International Conference on Multibody Systems, Nonlinear Dynamics, and Control, International Design Engineering Technical Conferences and Computers and Information in Engineering Conference (ASME IDETC 2020)*, 2020.

[22] J. Gerstmayr and H. Irschik. On the correct representation of bending and axial deformation in the absolute nodal coordinate formulation with an elastic line approach. *Journal of Sound and Vibration*, 318(3):461–487, 2008.

[23] J. Gerstmayr, P. Manzl, and M. Pieber. Multibody models generated from natural language. preprint, Research Square, https://www.researchsquare.com/article/rs-3552291/v1, 2023.

[24] J. Gerstmayr and M. Stangl. High-Order Implicit Runge-Kutta Methods for Discontinuous Multibody Systems. In D. A. Indeitsev, editor, *Proceedings of the {XXXII} Summer School on Advanced Problems in Mechanics ({APM} 2004)*, 2004.

[25] A. Gfrerrer. Geometry and kinematics of the mecanum wheel. *Computer Aided Geometric Design*, 25(9):784–791, 2008.

[26] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA., 1989.

[27] Y. Gonthier, J. McPhee, C. Lange, and J.-C. Piedbœuf. A Regularized Contact Model with Asymmetric Damping and Dwell Time Dependent Friction. *Multibody System Dynamics*, 11(3):209–233, 2004.

[28] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving ordinary differential equations I, nonstiff problems*. Springer Berlin Heidelberg, 1987.

[29] G. H. Heirman and W. Desmet. Interface reduction of flexible bodies for efficient modeling of body flexibility in multibody dynamics. *Multibody system dynamics*, 24:219–234, 2010.

[30] D. Henderson. Euler angles, quaternions, and transformation matrices for space shuttle analysis. Technical report, NASA, 1977.

[31] S. Holzinger, M. Arnold, and J. Gerstmayr. Evaluation and implementation of Lie group integration methods for rigid multibody systems. preprint, Research Square, https://www.researchsquare.com/article/rs-2715112/v1, 2023.

[32] S. Holzinger and J. Gerstmayr. Time integration of rigid bodies modelled with three rotation parameters. *Multibody System Dynamics*, 53(4):345–378, 2021.

[33] S. Holzinger, M. Schieferle, C. Gutmann, M. Hofer, and J. Gerstmayr. Modeling and Parameter Identification for a Flexible Rotor With Impacts. *Journal of Computational and Nonlinear Dynamics*, 17(5), 2022. 051008.

[34] S. Holzinger, J. Schöberl, and J. Gerstmayr. The equations of motion for a rigid body using non-redundant unified local velocity coordinates. *Multibody System Dynamics*, 48(3):283–309, 2020.

[35] K. H. Hunt and F. R. E. Crossley. Coefficient of Restitution Interpreted as Damping in Vibroimpact. *Journal of Applied Mechanics*, 42(2):440–445, 06 1975.

[36] W. C. Hurty. Dynamic analysis of structural systems using component modes. *American Institute of Aeronautics and Astronautics Journal*, 4(3):678–685, 1965.

[37] W. Jakob, J. Rhinelander, and D. Moldovan. pybind11 – Seamless operability between C++11 and Python, 2016. https://github.com/pybind/pybind11.

[38] J. Kiusalaas. *Numerical methods in engineering with Python 3*. Cambridge University Press, 2013.

[39] U. Lugrís, J. Escalona, D. Dopico, and J. Cuadrado. Efficient and accurate simulation of the rope-sheave interaction in weight-lifting machines. *Proceedings of the Institution of Mechanical Engineers, Part K: Journal of Multi-body Dynamics*, 225(4):331–343, 2011.

[40] P. Manzl and J. Gerstmayr. An improved dynamic model of the mecanum wheel for multibody simulations. In *ASME 2021 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. American Society of Mechanical Engineers Digital Collection, 2021.

[41] P. Manzl, O. Rogov, J. Gerstmayr, A. Mikkola, and G. Orzechowski. Reliability evaluation of reinforcement learning methods for mechanical systems with increasing complexity. preprint, Research Square, https://www.researchsquare.com/article/rs-3066420/v1, 2023.

[42] J. Morrissey, S. Thakur, and J. Ooi. EDEM Contact Model: Adhesive Elasto-Plastic Model. 2014.

[43] A. Müller. Coordinate Mappings for Rigid Body Motions. *Journal of Computational and Nonlinear Dynamics*, 12(2):10, 2017.

[44] R. Neurauter and J. Gerstmayr. A novel motion-reconstruction method for inertial sensors with constraints. *Multibody System Dynamics*, 57:181–209, 2023.

[45] N. M. Newmark. A Method of Computation for Structural Dynamics. *ASCE Journal of the Engineering Mechanics Division*, 85(3):67–94, 1959.

[46] P. E. Nikravesh. *Computer-Aided Analysis of Mechanical Systems*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1988.

[47] A. Pechstein and J. Gerstmayr. A Lagrange-Eulerian formulation of an axially moving beam based on the absolute nodal coordinate formulation. *Multibody System Dynamics*, 30(3):343–358, 2013.

[48] M. Pieber, K. Ntarladima, and J. Gerstmayr. A Hybrid Arbitrary Lagrangian Eulerian Formulation for the Investigation of the Stability of Pipes Conveying Fluid and Axially Moving Beams. *Journal of Computational and Nonlinear Dynamics*, 17(5):051006 (13 pages), 2022.

[49] Z. Qian, D. Zhang, and C. Jin. A regularized approach for frictional impact dynamics of flexible multi-link manipulator arms considering the dynamic stiffening effect. *Multibody System Dynamics*, 43:229–255, 2018.

[50] J. Rahikainen, F. González, M. Á. Naya, J. Sopanen, and A. Mikkola. On the cosimulation of multibody systems and hydraulic dynamics. *Multibody System Dynamics*, 50, 10 2020.

[51] J. Schöberl. NETGEN An advancing front 2D/3D-mesh generator based on abstract rules. *Computing and Visualization in Science*, 1:41–52, 1997.

[52] J. Schöberl. C++11 Implementation of Finite Elements in NGSolve, ASC Report 30/2014. https://www.asc.tuwien.ac.at/~schoeberl/wiki/publications/ngs-cpp11.pdf (accessed on August 16, 2022), 2014.

[53] R. Schwertassek and O. Wallrapp. *Dynamik flexibler Mehrkörpersysteme*. Grundlagen und Fortschritte der Ingenieurwissenschaften. Vieweg+Teubner Verlag, 1999.

[54] M. Sereinig, P. Manzl, and J. Gerstmayr. Task dependent comfort zone, a base placement strategy for autonomous mobile manipulators using manipulability measures. *Robotics and Autonomous Systems, submitted*, 2023.

[55] A. A. Shabana. Definition of the slopes and the finite element absolute nodal coordinate formulation. *Multibody system dynamics*, 1(3):339–348, 1997.

[56] A. A. Shabana. *Dynamics of Multibody Systems*. Cambridge University Press, Cambridge, 4th edition, 2013.

[57] B. Siciliano and O. Khatib, editors. *Springer Handbook of Robotics*. Springer, 2016.

[58] J. C. Simo and L. Vu-Quoc. On the dynamics in space of rods undergoing large motions - A geometrically exact approach. *Computer Methods in Applied Mechanics and Engineering*, 66(2):125–161, 1988.

[59] V. Sonneville, O. Brüls, and O. A. Bauchau. Interpolation schemes for geometrically exact beams: A motion approach. *International Journal for Numerical Methods in Engineering*, 112:1129–1153, 2017.

[60] V. Sonneville, A. Cardona, and O. Brüls. Geometrically exact beam finite element formulated on the special Euclidean group SE(3). *Computer Methods in Applied Mechanics and Engineering*, 268:451–474, 2014.

[61] Z. Terze, A. Müller, and D. Zlatar. Singularity-free time integration of rotational quaternions using non-redundant ordinary differential equations. *Multibody System Dynamics*, 38(3):201–225, 2016.

[62] D. Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4:65–85, 1994.

[63] C. Woernle. *Mehrkörpersysteme: Eine Einführung in die Kinematik und Dynamik von Systemen starrer Körper*. Lecture Notes in Applied and Computational Mechanics, Springer Berlin Heidelberg, 2016.

[64] C. Woernle. *Multibody Systems – An Introduction to the Kinematics and Dynamics of Systems of Rigid Bodies*. Springer, 2024.

[65] T. Yoshikawa. Manipulability of robotic mechanisms. *The International Journal of Robotics Research*, 4(2):3–9, 1985.

[66] A. Zwölfer and J. Gerstmayr. A concise nodal-based derivation of the floating frame of reference formulation for displacement-based solid finite elements. *Multibody System Dynamics*, 49(3):291–313, 2020.

[67] A. Zwölfer and J. Gerstmayr. The nodal-based floating frame of reference formulation with modal reduction. *Acta Mechanica*, 232(3):835–851, 2021.

Chapter 13

License

```
EXUDYN General License (Version 1.0)
====================================

Copyright (c) 2018-2025 Johannes Gerstmayr, Institute of Mechatronics, University of Innsbruck,
    Austria.

Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are
met:

- Redistributions of source code must retain the above copyright
  notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright
  notice, this list of conditions and the following disclaimer listed
  in this license in the documentation and/or other materials
  provided with the distribution.

- Neither the name of the copyright holders nor the names of its
  contributors may be used to endorse or promote products derived from
  this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


EXUDYN is making use of the following third party codes and libraries, which are mentioned in the
    following incl. the licenses:

HotInt General License (Version 1.0)
====================================

Copyright (c) 1997 - 2018 Johannes Gerstmayr, Linz Center of Mechatronics GmbH,
Austrian Center of Competence in Mechatronics GmbH, Institute of Technical Mechanics
at the Johannes Kepler Universitaet Linz, Austria. All rights reserved.

Copyright (c) 2018 Institute of Mechatronics, University of Innsbruck, Austria.

Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are
met:

- Redistributions of source code must retain the above copyright
  notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright
  notice, this list of conditions and the following disclaimer listed
```

in this license in the documentation and/or other materials
provided with the distribution.

- Neither the name of the copyright holders nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.

The copyright holders provide no reassurances that the source code
provided does not infringe any patent, copyright, or any other
intellectual property rights of third parties. The copyright holders
disclaim any liability to any recipient for claims brought against
recipient by any third party for infringement of that parties
intellectual property rights.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

=======================================
NGsolve / NETGEN:
Exudyn has lots of interfaces to NGsolve and Netgen, but does not
include NGsolve or Netgen source code since version 1.9.198.
However some implementations closely follow the NGsolve project,
see the respective comments in the code.

=======================================
GLFW 3.3 - www.glfw.org

Copyright (c) 2002-2006 Marcus Geelnard
Copyright (c) 2006-2016 Camilla Löwy <elmindreda@glfw.org>

This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
claim that you wrote the original software. If you use this software
in a product, an acknowledgment in the product documentation would
be appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not
be misrepresented as being the original software.

3. This notice may not be removed or altered from any source
distribution.

=======================================
PYBIND11
Copyright (c) 2016 Wenzel Jakob <wenzel.jakob@epfl.ch>, All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this
list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
this list of conditions and the following disclaimer in the documentation
and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its contributors
may be used to endorse or promote products derived from this software
without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

You are under no obligation whatsoever to provide any bug fixes, patches, or
upgrades to the features, functionality or performance of the source code
("Enhancements") to anyone; however, if you choose to make your Enhancements
available either publicly, or directly to the author of this software, without
imposing a separate written license agreement for such Enhancements, then you
hereby grant the following license: a non-exclusive, royalty-free perpetual
license to install, use, modify, prepare derivative works, incorporate into
other computer software, distribute, and sublicense such enhancements or
derivative works thereof, in binary and source code form.

OpenVR: Exudyn includes a link to OpenVR, using basic interfacing libraries from Valve:

Copyright (c) 2015, Valve Corporation
All rights reserved.

Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this
list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
this list of conditions and the following disclaimer in the documentation and/or
other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its contributors
may be used to endorse or promote products derived from this software without
specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

LEST - Copyright 2013-2018 by Martin Moene
Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization
obtaining a copy of the software and accompanying documentation covered by
this license (the "Software") to use, reproduce, display, distribute,
execute, and transmit the Software, and to prepare derivative works of the
Software, and to permit third-parties to whom the Software is furnished to
do so, all subject to the following:

The copyright notices in the Software and this entire statement, including
the above license grant, this restriction and the following disclaimer,
must be included in all copies of the Software, in whole or in part, and
all derivative works of the Software, unless such copies or derivative
works are solely in the form of machine-executable object code generated by
a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER

DEALINGS IN THE SOFTWARE.

==========
Eigen3 is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.
For more information go to http://eigen.tuxfamily.org/.
Eigen3 is used under the Mozilla Public License Version 2.0 (MPL2) license:

Mozilla Public License Version 2.0
==========

1. Definitions

1.1. "Contributor"
    means each individual or legal entity that creates, contributes to
    the creation of, or owns Covered Software.

1.2. "Contributor Version"
    means the combination of the Contributions of others (if any) used
    by a Contributor and that particular Contributor's Contribution.

1.3. "Contribution"
    means Covered Software of a particular Contributor.

1.4. "Covered Software"
    means Source Code Form to which the initial Contributor has attached
    the notice in Exhibit A, the Executable Form of such Source Code
    Form, and Modifications of such Source Code Form, in each case
    including portions thereof.

1.5. "Incompatible With Secondary Licenses"
    means

    (a) that the initial Contributor has attached the notice described
        in Exhibit B to the Covered Software; or

    (b) that the Covered Software was made available under the terms of
        version 1.1 or earlier of the License, but not also under the
        terms of a Secondary License.

1.6. "Executable Form"
    means any form of the work other than Source Code Form.

1.7. "Larger Work"
    means a work that combines Covered Software with other material, in
    a separate file or files, that is not Covered Software.

1.8. "License"
    means this document.

1.9. "Licensable"
    means having the right to grant, to the maximum extent possible,
    whether at the time of the initial grant or subsequently, any and
    all of the rights conveyed by this License.

1.10. "Modifications"
    means any of the following:

    (a) any file in Source Code Form that results from an addition to,
        deletion from, or modification of the contents of Covered
        Software; or

    (b) any new file in Source Code Form that contains any Covered
        Software.

1.11. "Patent Claims" of a Contributor
    means any patent claim(s), including without limitation, method,
    process, and apparatus claims, in any patent Licensable by such
    Contributor that would be infringed, but for the grant of the
    License, by the making, using, selling, offering for sale, having
    made, import, or transfer of either its Contributions or its
    Contributor Version.

1.12. "Secondary License"
    means either the GNU General Public License, Version 2.0, the GNU
    Lesser General Public License, Version 2.1, the GNU Affero General
    Public License, Version 3.0, or any later versions of those
    licenses.

1.13. "Source Code Form"
    means the form of the work preferred for making modifications.

1.14. "You" (or "Your")
    means an individual or a legal entity exercising rights under this
    License. For legal entities, "You" includes any entity that
    controls, is controlled by, or is under common control with You. For
    purposes of this definition, "control" means (a) the power, direct
    or indirect, to cause the direction or management of such entity,
    whether by contract or otherwise, or (b) ownership of more than
    fifty percent (50%) of the outstanding shares or beneficial
    ownership of such entity.

2. License Grants and Conditions

2.1. Grants

Each Contributor hereby grants You a world-wide, royalty-free,
non-exclusive license:

(a) under intellectual property rights (other than patent or trademark)
    Licensable by such Contributor to use, reproduce, make available,
    modify, display, perform, distribute, and otherwise exploit its
    Contributions, either on an unmodified basis, with Modifications, or
    as part of a Larger Work; and

(b) under Patent Claims of such Contributor to make, use, sell, offer
    for sale, have made, import, and otherwise transfer either its
    Contributions or its Contributor Version.

2.2. Effective Date

The licenses granted in Section 2.1 with respect to any Contribution
become effective for each Contribution on the date the Contributor first
distributes such Contribution.

2.3. Limitations on Grant Scope

The licenses granted in this Section 2 are the only rights granted under
this License. No additional rights or licenses will be implied from the
distribution or licensing of Covered Software under this License.
Notwithstanding Section 2.1(b) above, no patent license is granted by a
Contributor:

(a) for any code that a Contributor has removed from Covered Software;
    or

(b) for infringements caused by: (i) Your and any other third party's
    modifications of Covered Software, or (ii) the combination of its
    Contributions with other software (except as part of its Contributor
    Version); or

(c) under Patent Claims infringed by Covered Software in the absence of
    its Contributions.

This License does not grant any rights in the trademarks, service marks,
or logos of any Contributor (except as may be necessary to comply with
the notice requirements in Section 3.4).

2.4. Subsequent Licenses

No Contributor makes additional grants as a result of Your choice to
distribute the Covered Software under a subsequent version of this
License (see Section 10.2) or under the terms of a Secondary License (if
permitted under the terms of Section 3.3).

2.5. Representation

Each Contributor represents that the Contributor believes its
Contributions are its original creation(s) or it has sufficient rights
to grant the rights to its Contributions conveyed by this License.

2.6. Fair Use

This License is not intended to limit any rights You have under
applicable copyright doctrines of fair use, fair dealing, or other
equivalents.

2.7. Conditions

Sections 3.1, 3.2, 3.3, and 3.4 are conditions of the licenses granted in Section 2.1.

3. Responsibilities

3.1. Distribution of Source Form

All distribution of Covered Software in Source Code Form, including any Modifications that You create or to which You contribute, must be under the terms of this License. You must inform recipients that the Source Code Form of the Covered Software is governed by the terms of this License, and how they can obtain a copy of this License. You may not attempt to alter or restrict the recipients' rights in the Source Code Form.

3.2. Distribution of Executable Form

If You distribute Covered Software in Executable Form then:

(a) such Covered Software must also be made available in Source Code Form, as described in Section 3.1, and You must inform recipients of the Executable Form how they can obtain a copy of such Source Code Form by reasonable means in a timely manner, at a charge no more than the cost of distribution to the recipient; and

(b) You may distribute such Executable Form under the terms of this License, or sublicense it under different terms, provided that the license for the Executable Form does not attempt to limit or alter the recipients' rights in the Source Code Form under this License.

3.3. Distribution of a Larger Work

You may create and distribute a Larger Work under terms of Your choice, provided that You also comply with the requirements of this License for the Covered Software. If the Larger Work is a combination of Covered Software with a work governed by one or more Secondary Licenses, and the Covered Software is not Incompatible With Secondary Licenses, this License permits You to additionally distribute such Covered Software under the terms of such Secondary License(s), so that the recipient of the Larger Work may, at their option, further distribute the Covered Software under the terms of either this License or such Secondary License(s).

3.4. Notices

You may not remove or alter the substance of any license notices (including copyright notices, patent notices, disclaimers of warranty, or limitations of liability) contained within the Source Code Form of the Covered Software, except that You may alter any license notices to the extent required to remedy known factual inaccuracies.

3.5. Application of Additional Terms

You may choose to offer, and to charge a fee for, warranty, support, indemnity or liability obligations to one or more recipients of Covered Software. However, You may do so only on Your own behalf, and not on behalf of any Contributor. You must make it absolutely clear that any such warranty, support, indemnity, or liability obligation is offered by You alone, and You hereby agree to indemnify every Contributor for any liability incurred by such Contributor as a result of warranty, support, indemnity or liability terms You offer. You may include additional disclaimers of warranty and limitations of liability specific to any jurisdiction.

4. Inability to Comply Due to Statute or Regulation

If it is impossible for You to comply with any of the terms of this License with respect to some or all of the Covered Software due to statute, judicial order, or regulation then You must: (a) comply with the terms of this License to the maximum extent possible; and (b) describe the limitations and the code they affect. Such description must be placed in a text file included with all distributions of the Covered Software under this License. Except to the extent prohibited by statute

or regulation, such description must be sufficiently detailed for a recipient of ordinary skill to be able to understand it.

5. Termination

5.1. The rights granted under this License will terminate automatically if You fail to comply with any of its terms. However, if You become compliant, then the rights granted under this License from a particular Contributor are reinstated (a) provisionally, unless and until such Contributor explicitly and finally terminates Your grants, and (b) on an ongoing basis, if such Contributor fails to notify You of the non-compliance by some reasonable means prior to 60 days after You have come back into compliance. Moreover, Your grants from a particular Contributor are reinstated on an ongoing basis if such Contributor notifies You of the non-compliance by some reasonable means, this is the first time You have received notice of non-compliance with this License from such Contributor, and You become compliant prior to 30 days after Your receipt of the notice.

5.2. If You initiate litigation against any entity by asserting a patent infringement claim (excluding declaratory judgment actions, counter-claims, and cross-claims) alleging that a Contributor Version directly or indirectly infringes any patent, then the rights granted to You by any and all Contributors for the Covered Software under Section 2.1 of this License shall terminate.

5.3. In the event of termination under Sections 5.1 or 5.2 above, all end user license agreements (excluding distributors and resellers) which have been validly granted by You or Your distributors under this License prior to termination shall survive termination.

```
************************************************************************
*                                                                      *
*  6. Disclaimer of Warranty                                           *
*  -------------------------                                           *
*                                                                      *
*  Covered Software is provided under this License on an "as is"       *
*  basis, without warranty of any kind, either expressed, implied, or  *
*  statutory, including, without limitation, warranties that the       *
*  Covered Software is free of defects, merchantable, fit for a        *
*  particular purpose or non-infringing. The entire risk as to the     *
*  quality and performance of the Covered Software is with You.        *
*  Should any Covered Software prove defective in any respect, You     *
*  (not any Contributor) assume the cost of any necessary servicing,   *
*  repair, or correction. This disclaimer of warranty constitutes an   *
*  essential part of this License. No use of any Covered Software is   *
*  authorized under this License except under this disclaimer.         *
*                                                                      *
************************************************************************

************************************************************************
*                                                                      *
*  7. Limitation of Liability                                          *
*  --------------------------                                          *
*                                                                      *
*  Under no circumstances and under no legal theory, whether tort      *
*  (including negligence), contract, or otherwise, shall any           *
*  Contributor, or anyone who distributes Covered Software as          *
*  permitted above, be liable to You for any direct, indirect,         *
*  special, incidental, or consequential damages of any character      *
*  including, without limitation, damages for lost profits, loss of    *
*  goodwill, work stoppage, computer failure or malfunction, or any    *
*  and all other commercial damages or losses, even if such party      *
*  shall have been informed of the possibility of such damages. This   *
*  limitation of liability shall not apply to liability for death or   *
*  personal injury resulting from such party's negligence to the       *
*  extent applicable law prohibits such limitation. Some               *
*  jurisdictions do not allow the exclusion or limitation of           *
*  incidental or consequential damages, so this exclusion and          *
*  limitation may not apply to You.                                    *
*                                                                      *
************************************************************************
```

8. Litigation

Any litigation relating to this License may be brought only in the

courts of a jurisdiction where the defendant maintains its principal
place of business and such litigation shall be governed by laws of that
jurisdiction, without reference to its conflict-of-law provisions.
Nothing in this Section shall prevent a party's ability to bring
cross-claims or counter-claims.

9. Miscellaneous

This License represents the complete agreement concerning the subject
matter hereof. If any provision of this License is held to be
unenforceable, such provision shall be reformed only to the extent
necessary to make it enforceable. Any law or regulation which provides
that the language of a contract shall be construed against the drafter
shall not be used to construe this License against a Contributor.

10. Versions of the License

10.1. New Versions

Mozilla Foundation is the license steward. Except as provided in Section
10.3, no one other than the license steward has the right to modify or
publish new versions of this License. Each version will be given a
distinguishing version number.

10.2. Effect of New Versions

You may distribute the Covered Software under the terms of the version
of the License under which You originally received the Covered Software,
or under the terms of any subsequent version published by the license
steward.

10.3. Modified Versions

If you create software not governed by this License, and you want to
create a new license for such software, you may create and use a
modified version of this License if you rename the license and remove
any references to the name of the license steward (except to note that
such modified license differs from this License).

10.4. Distributing Source Code Form that is Incompatible With Secondary
Licenses

If You choose to distribute Source Code Form that is Incompatible With
Secondary Licenses under the terms of this version of the License, the
notice described in Exhibit B of this License must be attached.

Exhibit A - Source Code Form License Notice

  This Source Code Form is subject to the terms of the Mozilla Public
  License, v. 2.0. If a copy of the MPL was not distributed with this
  file, You can obtain one at http://mozilla.org/MPL/2.0/.

If it is not possible or desirable to put the notice in a particular
file, then You may include the notice in a location (such as a LICENSE
file in a relevant directory) where a recipient would be likely to look
for such a notice.

You may add additional accurate notices of copyright ownership.

Exhibit B - "Incompatible With Secondary Licenses" Notice

  This Source Code Form is "Incompatible With Secondary Licenses", as
  defined by the Mozilla Public License, v. 2.0.
