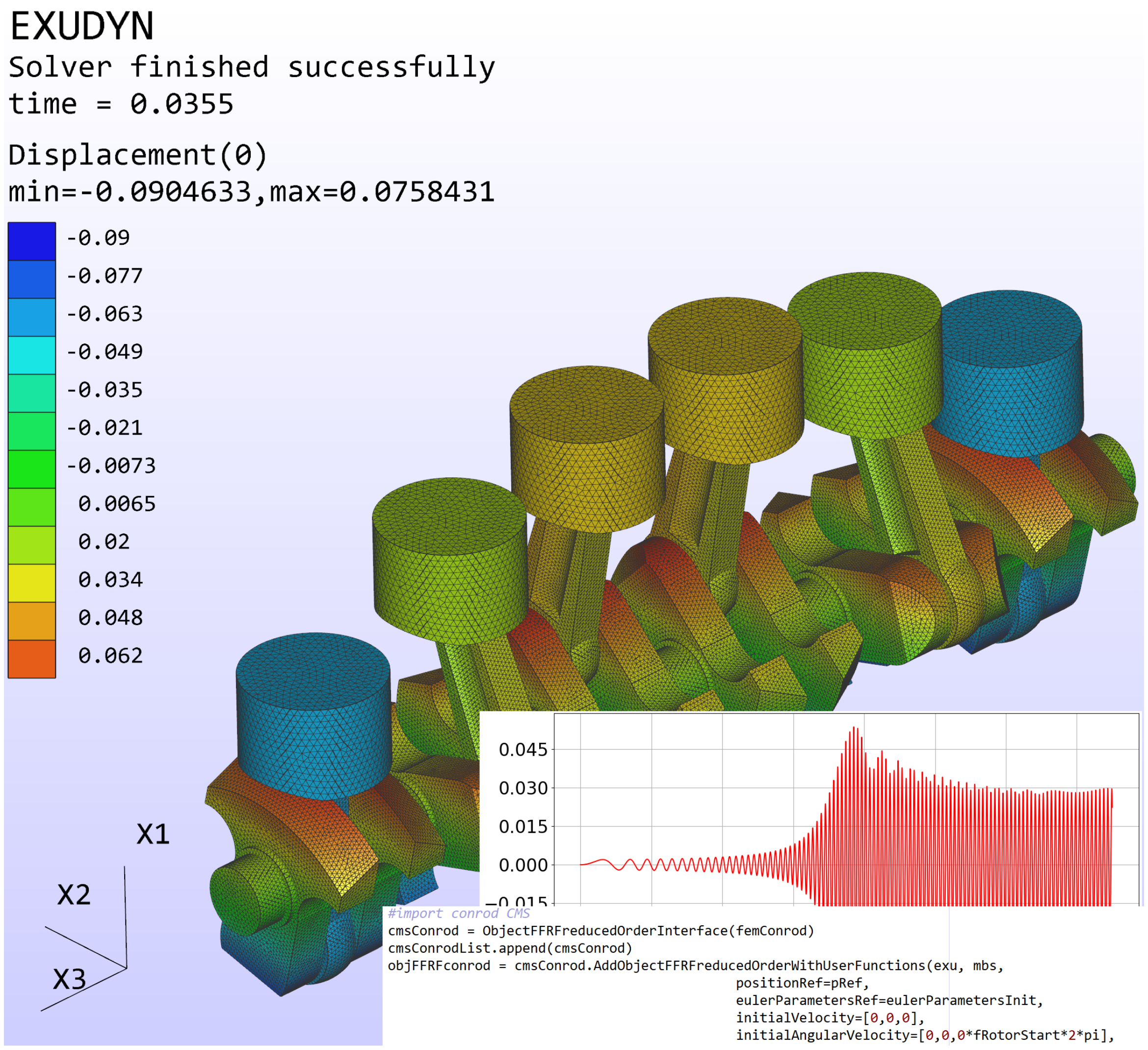
Flexible Multibody Dynamics Systems with Python and C++

EXUDYN

USER DOCUMENTATION



```
#import conrod CMS
cmsConrod = ObjectFFRFreducedOrderInterface(femConrod)
cmsConrodList.append(cmsConrod)
objFFRFconrod = cmsConrod.AddObjectFFRFreducedOrderWithUserFunctions(exu, mbs,
                                        positionRef=pRef,
                                        eulerParametersRef=eulerParametersInit,
                                        initialVelocity=[0,0,0],
                                        initialAngularVelocity=[0,0,0*fRotorStart*2*pi],
```

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

EXUDYN version = 1.0.130
CHECK section Section 2.5 and Section 11 for changes from previous versions!!!

University of Innsbruck, Department of Mechatronics, February 10, 2021,

Johannes Gerstmayr

Table of Contents

Chapter 1

Getting Started

The documentation for EXUDYN is split into this introductory section, including a quick start up, code structure and important hints, as well as a couple of sections containing references to the available Python interfaces to interact with EXUDYN and finally some information on theory (e.g., 'Solver').

EXUDYN is hosted on GitHub [8]:

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

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 Spyder
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. The open source code HOTINT reached limits of further (efficient) development and it seemed impossible to continue from this code as it is outdated regarding programming techniques and the numerical formulation.

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.

1.1.2 Developers of EXUDYN and thanks

EXUDYN is currently (2-2021) 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. 15 years of development led to a lot of lessions learned and after 20 years, a code must be re-designed.

Some specific earlier code parts 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`**[17], originally written by Jakob Wenzel, see `https://github.com/pybind/pybind11`. Without **`Pybind11`** we couldn't have made this project.

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 the examples:

- Stefan Holzinger, Michael Pieber, Joachim Schöberl, Manuel Schieferle, Martin Knapp, Lukas March, Dominik Sponring, David Wibmer, Andreas Zwölfer

– thanks a lot! –

1.2 Installation instructions

1.2.1 How to install EXUDYN?

In order to run EXUDYN, you need an appropriate Python installation. We currently (2021-02) recommend to use

- **Anaconda, 64bit, Python 3.7.7**[1]
- In case that you have an older CPU, which does not support AVX2, use: Anaconda, 32bit, Python 3.6.5)[2]
- Alternative option (Spyder runs more stable ...): Anaconda, 64bit, Python 3.6.5)[3]
- **Spyder 4.1.3** with Python 3.7.7, 64bit, which is included in the Anaconda installation[4]

If you plan to extend the C++ code, we recommend to use VS2017[5] to compile your code, which offers Python 3.7 compatibility. Once again, remember that Python versions and the version of the EXUDYN module must be identical (e.g., Python 3.6 32 bit **both** in the EXUDYN module and in Spyder).

1.2.2 Install with Windows MSI installer

The simplest way on Windows 10 (and maybe also Windows 7), which works well **if you installed only one python version** and if you installed Anaconda with the option **'Register Anaconda as my default Python 3.x'** or similar, then you can use the provided `.msi` installers in the `main/dist` directory:

- For the 64bits python 3.6 version, double click on (version may differ):
  `exudyn-1.0.8.win-amd64-py3.6.msi`
- Follow the instructions of the installer
- If python / Anaconda is not found by the installer, provide the 'python directory' as the installation directory of Anaconda3, which usually is installed in:
  `C:\ProgramData\Anaconda3`

1.2.3 Install from Wheel (UBUNTU and Windows)

The **standard way to install** the python package EXUDYN is to use the so-called 'wheels' (file ending `.whl`) provided at the directory wheels in the EXUDYN repository.

[1]Anaconda3 64bit with Python3.7.7 can be downloaded via the repository archive `https://repo.anaconda.com/archive/` choosing `Anaconda3-2020.02-Windows-x86_64.exe`

[2]Anaconda 32bit with Python3.6 can be downloaded via the repository archive `https://repo.anaconda.com/archive/` choosing `Anaconda3-5.2.0-Windows-x86.exe`.

[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]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.

[5]previously, VS2019 was recommended: However, VS2019 has problems with the library 'Eigen' and therefore leads to erroneous results with the sparse solver. VS2017 can also be configured with Python 3.7 now.

For UBUNTU18.04 (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 UBUNTU20.04 (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`[6], 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
...
```

- → type `exit()` to close python

Go to the folder `Exudyn_git/main` (where `setup.py` lies) and choose the wheel in subdirectory `main/dist` according to your system (windows/UBUNTU), python version (3.6 or 3.7) and 32 or 64 bits.

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 Work without installation and editing `sys.path`

The **uncommon and old way** (→ not recommended for EXUDYN versions ≥ 1.0.0) is to use Python's sys module to link to your `exudyn` (previously `WorkingRelease`) directory, for example:

```python
import sys
sys.path.append('C:/DATA/cpp/EXUDYN_git/bin/EXUDYN32bitsPython36')
```

The folder `EXUDYN32bitsPython36` needs to be adapted to the location of the according EXUDYN package.

[6]python3 under UBUNTU 18.04

1.2.5 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:

- 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)
  - go to your Scripts folder inside the Anaconda folder (e.g. `C:\ProgramData\Anaconda\Scripts`)
  - run 'activate.bat'
- go to 'main' of your cloned github folder of exudyn
- run: `python setup.py install`
- 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 (requires installation of Microsoft Visual Studio; recommended: VS2017):

```
python setup.py bdist_wheel
```

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

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 UBUNTU18.04, otherwise use `sudo apt install python3`)[7]:
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
```

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

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

```
sudo apt-get install freeglut3 freeglut3-dev
sudo apt-get install mesa-common-dev
sudo apt-get install libglfw3 libglfw3-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):

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

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 (UBUNTU18.04) and with same python version (e.g., 3.6):

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

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 EXUDYNworks well on UBUNTU18.04 with the original `Python 3.6.9` and `gcc-7.5.0` version as well as with UBUNTU20.04 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 using the C++ code (advanced)?

EXUDYN is still under intensive development of core modules. There are several ways to 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
  - 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.: not fully supported yet; however, all external libraries are Linux-compatible and thus should run with minimum adaptation efforts.

1.3 Further notes

1.3.1 Goals of EXUDYN

After the first development phase (2019-2020), it shall

- be a small multibody library, which can be easily linked to other projects,
- allow to efficiently simulate small scale systems (compute 100000s time steps per second for systems with $n_{DOF} < 10$),
- allow to efficiently simulate medium scaled systems for problems with $n_{DOF} < 1\,000\,000$,
- safe and widely accessible module for Python,
- allow to add user defined objects in C++,
- allow to add user defined solvers in Python.

Future goals are:

- extend tests,
- add more multi-threaded parallel computing techniques (first trials implemented, improvements planned: Q3 2021),
- add vectorization,
- add specific and advanced connectors/constraints (3D revolute joint and prismatic joint instead of generic joint, extended wheels, contact, control connector)
- more interfaces for robotics,
- add 3D beams,
- extend floating frame of reference formulation with modal reduction

For specific open issues, see `trackerlog.html`.

Listing 1.1: My first example

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

1.3.2 Run a simple example in Spyder

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. For the following example, either

- open Spyder and copy the example provided in Listing 1.1 into a new file, or
- open `myFirstExample.py` from your `EXUDYN32bitsPython36`[8] directory

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/bin/EXUDYN32bitsPython36/myFirstExample.py',
  wdir='C:/DATA/cpp/EXUDYN_git/main/bin/EXUDYN32bitsPython36')
++++++++++++++++++++++++++++++
EXUDYN V1.0.1 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 generalized alpha solver solution file
  #simulation started=2019-11-14,20:35:12
```

[8]or any other directory according to your python version

```
#columns contain: time, ODE2 displacements, ODE2 velocities, ODE2 accelerations, AE
    coordinates, ODE2 velocities
#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
#
0,0,0,0,0,0.0001,0
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=2019-11-14,20:35:12
#Solver Info: errorOccurred=0,converged=1,solutionDiverged=0,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 solution of coordinates, their derivatives and Lagrange multipliers on system level. 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.

1.4 Trouble shooting and FAQ

Known issues:

1. Sometimes the `exudyn` module cannot be loaded into Python. There are several reasons and workarounds:

   - You mixed up 32 and 64 bits (see below) version
   - 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
   - `ModuleNotFoundError: No module named 'exudynCPP'`:
     A known reason is that your CPU[9] does not support AVX2, while exudyn is compiled with the AVX2 option;

[9]modern Intel Core-i3, Core-i5 and Core-i7 processors as well as AMD processors, espcially Zen and Zen-2 architectures should have no problems with AVX; however, low-cost Celeron and Pentium processors **will not support AVX**, 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

→ **workaround** to solve the AVX problem: use the Python 3.6 32bits version, which is compiled without AVX2; you can also compile for your specific Python version without AVX if you adjust the `setup.py` file in the `main` folder.
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.

2. When importing EXUDYN in python (windows) I get the error (or similar):

```
Traceback (most recent call last):
  File "C:\DATA\cpp\EXUDYN_git\main\pythonDev\pytest.py", line 18, in <module>
    import exudyn as exu
  ImportError: DLL load failed: %1 is no valid Win32 application.
```

→ probably this is a 32/64bit problem. Your Python installation and EXUDYN need to be **BOTH** either 64bit OR 32bit (Check in your python help; exudyn in WorkingRelease64 is the 64bit version, in WorkingRelease it is the 32bit version) and the Python installation and EXUDYN need to have **BOTH** the same version and 1st subversion number (e.g., 3.6.5 should be compatible with 3.6.2).

3. I do not understand the python errors – how can I find the reason of the error or crash?

   → 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`

4. 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

List of Frequently asked questions:

1. Where do I find the '.exe' file?

   → EXUDYN is only available via the python interface as exudyn.pyd library, which is located in folder: `main/bin/WorkingRelease`. This means that you need to run python (best: Spyder) and import the EXUDYN module.

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

   → probably you used wrong item indices, see beginning of Section 4.

- E.g., an object number `oNum = mbs.AddObject(...)` is used at a place where a `NodeIndex` is expected: `mbs.AddObject(MassPoint(nodeNumber=oNum, ...))`
- Usually, this is an ERROR in your code, it does not make sense to mix up these indices!
- In the exceptional case, that you want to convert numbers, see beginning of Section 4.

3. Why does type auto completion does not work for mbs (Main system)?

   → UPDATE 2020-06-01: with Spyder 4, using Python 3.7, type auto completion works much better, but may find too many completions.

   → most python environments (e.g., with Spyder 3) only have information up to the first sub-structure, e.g., `SC=exu.SystemContainer()` provides full access to SC in the type completion, but `mbs=SC.AddSystem()` is at the second sub-structure of the module and is not accessible. WORKAROUND: type `mbs=MainSystem()` **before** the `mbs=SC.AddSystem()` command and the interpreter will know what type mbs is. This also works for settings, e.g., simulation settings 'Newton'.

4. 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.

5. 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.

6. I get an error in `SC.TimeIntegrationSolve(mbs, 'GeneralizedAlpha', simulationSettings)` 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>
   SC.TimeIntegrationSolve(mbs, 'GeneralizedAlpha', simulationSettings)
   SystemError:  <built-in method TimeIntegrationSolve of PyCapsule object at 0x0CC63590> returned
   a result with an error set
   ```

   → The prechecks, 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 catched by EXUDYN. However, you can just check all your user functions, if they will run without EXUDYN. E.g., a load user function UFload(mbs, t, load), which tries to access load[4] 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.
- → 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?*

7. 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 `exu.StartRenderer()`

Chapter 2

Overview on EXUDYN

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 10.

2.1.1 Overview of modules

Currently, the module structure is simple:

- Python parts:
  - `itemInterface`: contains the interface, which transfers python classes (e.g., of a NodePoint) to dictionaries that can be understood by the C++ module
  - `exudynUtilities`: constains helper classes in Python, which allows simpler working with EXUDYN
- C++ parts, see Figs. 2.1 and 2.2:
  - `exudyn`[1]: on this level, there are just very few functions: SystemContainer(), StartRenderer(), StopRenderer()
  - `SystemContainer`: contains the systems (most important), solvers (static, dynamics, ...), visualization settings

[1]For versions $<$ 1.0.0: there is a second module, called exudynFast, which deactivates all range-, index- or memory allocation checks at the gain of higher speed (probably 30 percent in regular cases and up to 100 percent in the 64 bit version). This module is included by `import exudynFast as exu` and can be used same as exudyn. To check the version, just type exu.__doc__ and you will see a note on 'exudynFast' in the exudynFast module.

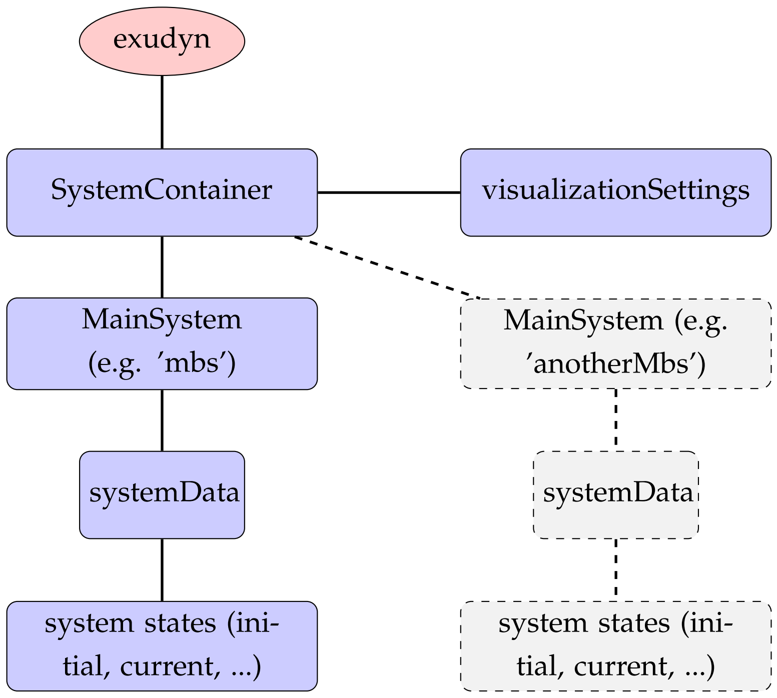

exudyn
SystemContainer
visualizationSettings
MainSystem (e.g. 'mbs')
MainSystem (e.g. 'anotherMbs')
systemData
systemData
system states (initial, current, ...)
system states (initial, current, ...)

Figure 2.1: Overview of exudyn module.

- `mbs`: system 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 6;
- `mbs.systemData`: contains the initial, current, visualization, ... states of the system and holds the items, see Fig. 2.2

2.1.2 Conventions: items, indices, coordinates

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

$$\text{item} \in \{\text{node}, \text{object}, \text{marker}, \text{load}\} \tag{2.1}$$

Indices: arrays and vector starting with 0:

As known from Python, all **indices** of arrays, vectors, etc. 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:

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.

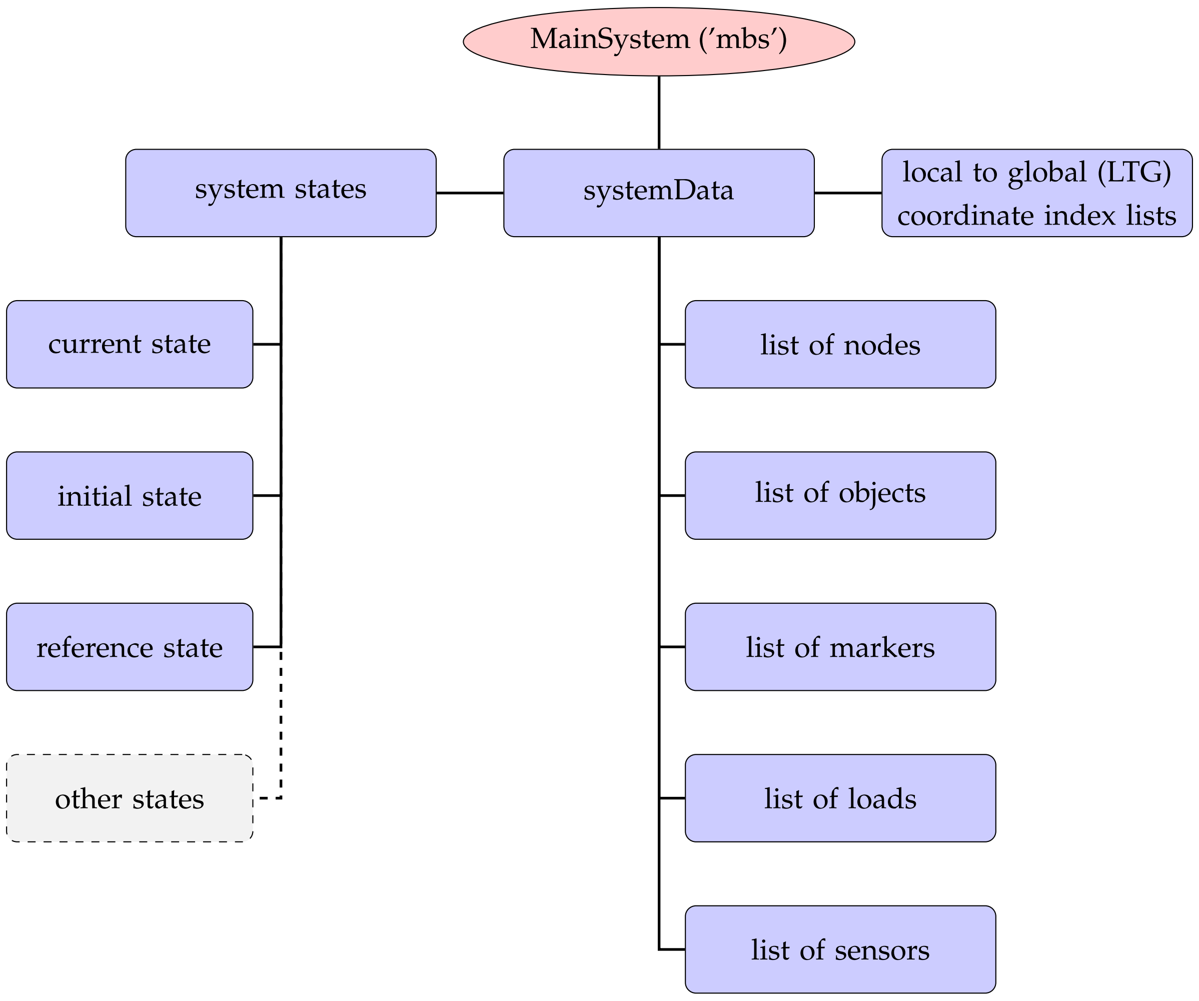


Figure 2.2: Overview of systemData, which connects items and states. Note that access to items is provided via functions in `system`.

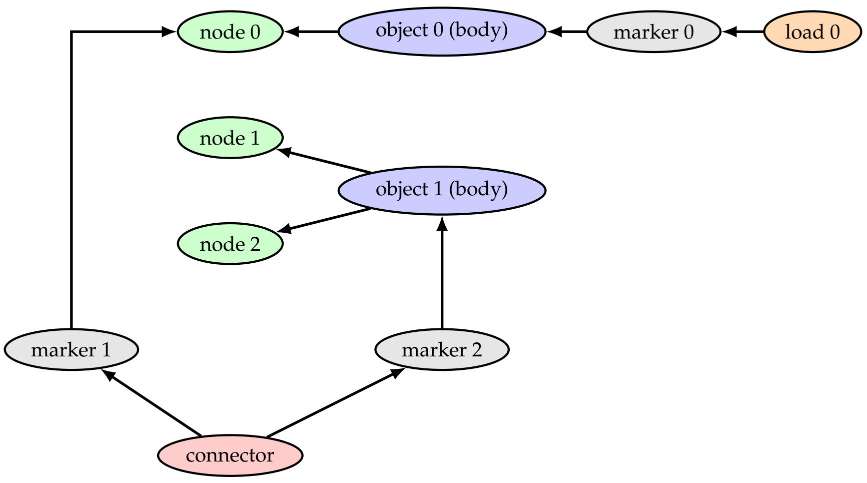


Figure 2.3: Typical interaction of items in a multibody system. Note that both, bodies and connectors/constraints are (computational) objects. The arrows indicate, that, e.g., object 1 has node 1 and node 2 (indices) and that marker 0 is attached to object 0, while load 0 uses marker 0 to apply the load. Sensors could additionally be attached to certain items.

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)
- 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. 2.3

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).

2.2.2 Objects

Objects are 'computational objects' and they provide equations to your system. Objects additionally often provide derivatives and have measurable quantities (e.g. displacement) and they provide access, which can be used to apply, e.g., forces.

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`.

2.3 Exudyn Basics

This section will show:

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

2.3.1 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` lives in C++ and will 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 `SC.TimeIntegrationSolve(...)`) 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.

2.3.2 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 Sections 7.1.1 – 7.1.9.

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.,

```
      #10 seconds of simulation time:
simulationSettings.timeIntegration.endTime = 10
      #1000 steps for time integration:
simulationSettings.timeIntegration.numberOfSteps = 1000
      #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
```

2.3.3 Visualization settings

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 Sections 7.2.1 – 7.2.13.

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.GetRenderState()
#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
```

2.3.3.1 Storing the model view

There is a simple way to store the current view (zoom, centerpoint, orientation, etc.) by using `SC.GetRenderState()` and `SC.SetRenderState()`. A simple way is to reload the stored render state (model view) after simulating your model once at the end of the simulation[2]:

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

#store model view for next run:
StopRenderer() #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.GetRenderState()
Out[1]:
{'centerPoint': [1.0, 0.0, 0.0],
 'maxSceneSize': 2.0,
 'zoom': 1.0,
 'currentWindowSize': [1024, 768],
```

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

```
 'modelRotation': [[ 0.34202015,  0.        ,  0.9396926 ],
                  [-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.SetRenderState` in your Python code to have a fixed model view in every simulation (`SC.SetRenderState` AFTER `exu.StartRenderer()`):

```
SC.visualizationSettings.general.autoFitScene = False #prevent from autozoom
exu.StartRenderer()
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.SetRenderState(renderState)
#.... further code for simulation here
```

2.3.4 Generating output and results

The solvers provide a number of options in `solutionSettings` to generate a solution file. As a default, exporting solution 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
```

2.3.5 Graphics pipeline

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

- compute new step
- 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 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)
  $\rightarrow$ 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

2.3.6 Graphics user Python functions

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 STL files (`GraphicsDataFromSTLfileTxt`).
- Some objects, e.g., `ObjectGenericODE2` or `ObjectRigidBody`, provide customized a function `graphicsDataUserFunction`. This user function just returns a list of GraphicsData, see Section 8.3. 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 graphicsUserPythonFunctions 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.

2.3.7 Color and RGBA

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

- red (R) channel
- green (G) channel

- blue (B) channel
- alpha (A) value, representing 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 `exudynGraphicsDataUtilities.py`, e.g., `color4red` or `color4steelblue` as well a list of 10 colors `color4list`, which is convenient to be used in a loop creating objects.

2.3.8 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. 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.tga, frame0001.tga,...'. Note that '.tga' files 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. 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.tga, frame00001.tga, ...  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.
        tga -c:v libx264 -vf "fps=25,format=yuv420p" animation.mp4
)
```

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

```
REM 2019-12-23, Johannes Gerstmayr
REM helper file for EXUDYN
REM delete all .tga images of current directory

del *.tga
```

2.4 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.

2.4.1 Focus of the C++ code

Four principles:

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

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 1:1 image of the system and the processes running in it, which can be controlled with the extensive possibilities of Python.

2.4.2 C++ Code structure

The functionality of the code is based on systems (MainSystem/CSystem) 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)
- node: used to hold coordinates (unknowns)
- (computational) object: leads to equations, using nodes
- marker: defines a consistent interface to objects (bodies) and nodes; write access ('AccessFunction') – provides jacobian and read access ('OutputVariable')
- load: acts on an object or node via a marker
- 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)

2.4.3 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 (indices 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

2.4.4 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

2.4.5 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., $f = M * q_t t + K * q$, GBar, ...
- use UpperCamelCase for functions, classes, structs, ...
- Special cases for CamelCase: write 'ODEsystem', BUT: 'ODE1Equations'
- '[...]Init' ... in arguments, for initialization of variables; e.g. 'valueInit' for initialization of member variable 'value'
- use American English troughout: Visualization, etc.
- for (abbreviations) in captial letters, e.g. ODE, use a lower case letter afterwards:
- do not use consecutive capitalized words, e.g. DO NOT WRITE 'ODEAE'
- for functions use `ODEComputeCoords()`, for variables avoid 'ODE' at beginning: use nODE or write odeCoords
- 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()`, `SetRotationalPara`
  `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

2.4.6 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 (5-line member functions).

Exceptions to the NO-ABBREVIATIONS-RULE:

- ODE ... ordinary differential equations;
- ODE2 ... marks parts related to second order differential equations (SOS2, EvalF2 in HOTINT)
- ODE1 ... marks parts related to first order differential equations (ES, EvalF in HOTINT)
- AE ... algebraic equations (IS, EvalG in HOTINT); write 'AEcoordinates' for 'algebraicEquationsCoordinates'
- 'C[...]' ... Computational, e.g. for ComputationalNode ==> use 'CNode'
- min, max ... minimum and maximum

- 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'

2.5 Changes

The following list covers changes in the python interface and functionality:

- **Version 1.0.59→ Version 1.0.61**
  - **Major changes**: Changed interface of `processing.GeneticOptimization()`;
- **Version 1.0.56→ Version 1.0.57**
  - **Major changes**: added new submodule `exudyn.signal`;
  - **Major changes**: `FilterSignal` renamed into `FilterSensorOutput` and moved from `exudyn.utilities` to `exudyn.signal`
- **Version 1.0.49→ Version 1.0.51**
  - user functions can now be set to zero, e.g.:
    ```
    mbs.AddObject(CoordinateSpringDamper(markerNumbers=[m0,m1],
        stiffness=k, damping=0.01*k,
        springForceUserFunction=0)
    ```
  - Exception: for `visualizationSettings.window.keyPressUserFunction` use `SC.visualizationSettings.window.ResetKeyPressUserFunction()` to set it to zero
- **Version 1.0.42→ Version 1.0.49**
  - **Major changes**: OpenGL default settings for lights and material have been changed; slightly different appearance might result from that! per default, light1 is active as well.
  - **Major changes**: Added key callback functionality `keyPressUserFunction` to `VisualizationSettings.window`
  - added mouse coordinates (OpenGL and screen pixels) to renderState and added a `GetCurrentMouseCoordinates()` function to SystemContainer
  - added mouse coordinates to renderer window (press 'F3')
  - identified problems with OpenGL lights; light1 is now working
  - solved problems with coordinateSystem
  - added a world basis (coordinate system) at origin of model (0,0,0); see `VisualizationSettings.general.drawWorldBasis`
  - added flag `simulateInRealtime` to `simulationSettings.timeIntegration`
- **Version 1.0.38→ Version 1.0.42**
  - completed `Acceleration` output variables for all relevant objects and nodes;
  - added FilterSignal(...) (Section 5.14) functionality to utilities, which enables numerical differentiation and filtering, using savgol filter, applied to all output values of data loaded from sensors.

  - for example, see `TestModels/objectFFRFreducedOrderAccelerations.py`.

- **Version 1.0.37→ Version 1.0.38**
  - `SC.StopRenderer()` now stores the last renderState in `exu.sys['renderState']`, which can be used in subsequent simulations to always have the same view and zoom; for an example, see Section 2.3.3.1.

- **Version 1.0.32→ Version 1.0.37**
  - added Python interfaces (exudyn/solver.py) for static/dynamic solvers and eigensolvers: exu.SolveStatic(mbs) and exu.SolveDynamic(mbs) → recommended to be used in future
  - TimeIntegrationSolve and StaticSolve → deprecated; get additional return value for success

- **Version 1.0.17 → Version 1.0.18**
  - **removed** AVX compilation flag for Python 3.6 32bits version due to incompatibility on older Celeron processors
  - added acceleration sensor functionality to most objects and nodes
  - added Lie group utilities (see `exudyn.lieGroup`)
  - added processing utilities for parameter variation and optimization (see `exudyn.processing`)

- **Version 1.0.12 → Version 1.0.13**

  corrected LHS (left-hand-side) and RHS (right-hand-side) terminology (issue: 330), see Section 9.1:
  - objects, connectors, etc., use LHS conventions: all terms (mass, stiffness, elastic forces, damping) are computed at LHS of equation
  - forces are written at the RHS
  - system quantities are always written on RHS: $m\ddot{q} = f_{RHS}$

- **Version 1.0.8 → Version 1.0.9**

  change from Index in mbs.AddNode(...), mbs.AddObject, ... to special 'item indices' (issue: 333):
  - before: `mbs.AddNode(...)` → Index; **now**: `mbs.AddNode(...)` → NodeIndex
  - before: `mbs.AddObject(...)` → Index; **now**: `mbs.AddObject(...)` → ObjectIndex
  - before: `mbs.AddMarker(...)` → Index; **now**: `mbs.AddMarker(...)` → MarkerIndex
  - before: `mbs.AddLoad(...)` → Index; **now**: `mbs.AddLoad(...)` → LoadIndex
  - before: `mbs.AddSensor(...)` → Index; **now**: `mbs.AddSensor(...)` → SensorIndex
  - Functions previously requiring an itemNumber have been changed to the according itemIndex, e.g., `mbs.SetNodeParameter(nodeNumber=...,...)` now requires a `nodeNumber` of type `NodeIndex` in order to avoid mistakes due to wrong types of indices.
  - for further details and specific usage, see beginning of Section 4!

  finally removed functions mbs.CallNodeFunction(...) and mbs.CallObjectFunction(...) (issue: 288)

  removed functions mbs.GetNodeByName(...), GetObjectByName(...), etc. (issue: 445)

- **Version 1.0.6 → Version 1.0.7**

  autocreate directories (issue: 431):
  - directories (folders) will be created for given paths
  - this applies, e.g., to sensor's `fileName` or simulation settings `coordinatesSolutionFileName`
  - previously, a non-existing directory led to an exception

- **Version 0.1.368 → Version 1.0.0**

Major changes in the python interface, as the utilities moved into the exudyn package:

  - `from itemInterface import *` → `from exudyn.itemInterface import *`
  - `from exudynUtilities import *` → `from exudyn.utilities import *`
  - `from exudynBasicUtilities import *` → `from exudyn.basicUtilities import *`
  - `from exudynFEM import *` → `from exudyn.FEM import *`
  - `from exudynGraphicsDataUtilities import *` → `from exudyn.graphicsDataUtilities import *`
  - `from exudynGUI import *` → `from exudyn.GUI import *`
  - `from exudynLieGroupIntegration import *` → `from exudyn.lieGroupIntegration import *`
  - `from exudynRigidBodyUtilities import *` → `from exudyn.rigidBodyUtilities import *`
  - `from exudynRobotics import *` → `from exudyn.robotics import *`

- **Version 0.1.360 → Version 0.1.361**

  Changes in the python interface:

  - `simulationSettings.timeIntegration.preStepPyExecute` and `simulationSettings.staticSolver.preStepPyExecute` are deprecated, DON'T USE any more
  - Use `mbs.SetPreStepUserFunction(...)` instead!

- **Version 0.1.352 → Version 0.1.353**

  Changes in the renderer screen:

  - Keys '0' and 'KEYPAD 0' → not available any more (set default rotation x/y)
  - Use keys CTRL+'1', SHIFT+CTRL+'1', CTRL+'2', ... → keys for new standard views!

- **Version 0.1.288 → Version 0.1.289**

  Changes in the python interface **(ESSENTIAL!)**:

  - Added time 't' as additional first argument in user functions: `ObjectCoordinateSpringDamper`, `ObjectConnectorCoordinateSpringDamper`, `ObjectConnectorCartesianSpringDamper`

- **Version 0.1.287 → Version 0.1.288**

  Changes in the python interface **(ESSENTIAL!)**:

  - changed the name of **initialDisplacements** to initialCoordinates in all Nodes for consistency reasons with rotation parameters!

- **Version 0.1.282 → Version 0.1.284**

  Changes in the python interface:

  - all bodyFixed parameters in MarkerRigidBody, which were inactive so far, have been eliminated

- **Version 0.1.260 → Version 0.1.263**

  Changes in the python interface:

  - `mbs.systemData.GetCurrentTime()` → `mbs.systemData.GetTime()`
  - `mbs.systemData.GetVisualizationTime()` → `mbs.systemData.GetTime(configurationType=exu.Configurat`

- **Version 0.1.244 → Version 0.1.245**

  Changes in the implementation / solver (LEADS TO DIFFERENT RESULTS):

  - **Solvers updated**: static solver and time integration have been updated; old solvers are still available with the 'OldSolver' extension

  Changes in the python interface (new functions / interface to call the old solvers):

  - `SC.SolveStaticOldSolver(...)`
  - `SC.TimeIntegrationSolve(mbs, 'GeneralizedAlphaOldSolver', simulationSettings)`

- **Version 0.1.243 → Version 0.1.244**

  Changes in the python interface:

  - `simulationSettings.staticSolver.pauseAfterEachStep`
    → `simulationSettings.pauseAfterEachStep (merged with timeIntegration.pauseAfterEachStep)`

- **Version 0.1.238 → Version 0.1.240**

  Changes in the implementation / solver (LEADS TO DIFFERENT RESULTS):

  - **generalizedAlpha**: corrected initialization of algorithmic acceleration for discontinuous iteration
  - **time integration**: corrected time *t* for evaluation of RHS from beginning to end of time step (improves accuracy for time-dependent loads significantly)

  Changes in the python interface:

  - `simulationSettings.timeIntegration.pauseAfterEachStep`
    → `simulationSettings.pauseAfterEachStep`
  - `ADDED: simulationSettings.timeIntegration.verboseModeFile`
  - `ADDED: simulationSettings.staticSolver.verboseModeFile`

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 2D rigid-body model (*coming soon*)
- Links to examples section

The python source code of this section can be found in the file:

`main/pythonDev/Examples/springDamperTutorial.py`

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

`main/pythonDev/Examples`

`main/pythonDev/TestModels`

This 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.itemInterface import *
import numpy as np #for postprocessing
```

Next, we need a SystemContainer, which contains all computable systems and add a new system. 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. This version is in line with the issue tracker and marks the number of open/closed issues added to EXUDYN:

```
print('EXUDYN version='+exu.__version__)
```

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 – 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))
```

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

[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.

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 loads can be applied to the first coordinate of node `n1` via marker with number `nodeMarker`, which is in fact of type `MarkerIndex`.

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

```
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))
```

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

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

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)
mbs.Assemble()
```

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:

```
steps = 1000  #number of steps to show solution
tEnd = 1     #end time of simulation
```

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 = steps
simulationSettings.timeIntegration.endTime = tEnd
```

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):

```
exu.StartRenderer()              #start graphics visualization
#mbs.WaitForUserToContinue()    #wait for pressing SPACE bar to continue
```

As the simulation is still very fast, we will not see the motion of our node. Using e.g. `steps=10000000` in the lines above allows you online visualize the resulting oscillations.

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

```
SC.TimeIntegrationSolve(mbs, 'GeneralizedAlpha', simulationSettings)
```

After simulation, our renderer needs to be stopped (otherwise it would stay in background and prohibit further simulations). Sometimes you would like to wait until closing the render window, using `WaitForRenderEngineStopFlag()`:

```
#SC.WaitForRenderEngineStopFlag()#wait for pressing 'Q' to quit
exu.StopRenderer()               #safely close rendering window!
```

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

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. For convenient reading the file containing commented lines, 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')
```

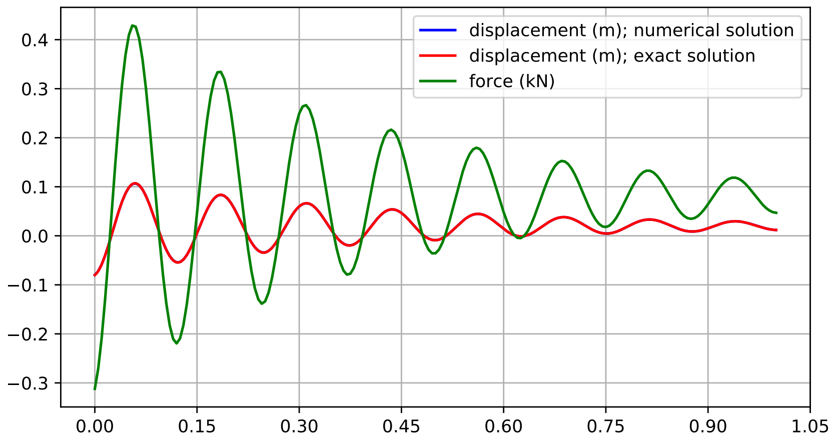
The sensor result can be loaded in the same way. The sensor output format contains time in the first column and sensor values in the remaining columns. The number of columns depends on the sensor and the output quantity (scalar, vector, ...):

```
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 like this:



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

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

Further examples can be found in your copy of exudyn:

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

Chapter 4

Python-C++ command interface

This section lists the basic interface functions which can be used to set up a EXUDYN model in Python.

To import the module, just include the EXUDYN module in Python (for compatibility with examples and other users, we recommend to use the 'exu' abbreviation throughout)[1]:

```
import exudyn as exu
```

In addition, you may work with a convenient interface for your items, therefore also always include the line

```
from exudyn.itemInterface import *
```

Everything you work with is provided by the class `SystemContainer`, 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 usually makes no sense.

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 as exu
from exudyn.itemInterface import *
SC = exu.SystemContainer()
exu.InfoStat() #prints some statistics if available
exu.Go() #creates a systemcontainer and main system
nInvalid = exu.InvalidIndex() #the invalid index, depends on architecture and version
```

[1]note that there is a second module, called exudynFast, which does deactivates all range-, index- or memory allocation checks at the gain of higher speed (probably 30 percent in regular cases). To check which version you have, just type exu.__doc__ and you will see a note on 'exudynFast' in the exudynFast module.

Understanding the usage of functions for python object 'SystemContainer' provided by EXUDYN, the following examples might help:

```
import exudyn as exu
from exudyn.itemInterface import *
SC = exu.SystemContainer()
mbs = SC.AddSystem()
nSys = SC.NumberOfSystems()
print(nSys)
SC.Reset()
```

If you run a parameter variation (check `Examples/parameterVariationExample.py`), you may delete the created `MainSystem` `mbs` and the `SystemContainer` `SC` before creating new instances in order to avoid memory growth.

Many functions will work with node numbers ('`NodeIndex`'), object numbers ('`ObjectIndex`'), marker numbers ('`MarkerIndex`') and others. These numbers are special 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)`
- You can convert any item index, e.g., NodeIndex `ni` into an integer number using `int(ni)` of `no.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 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

4.1 EXUDYN

These are the access functions to the EXUDYN module.

function/structure name	**description**
GetVersionString()	Get EXUDYN module version as string
RequireVersion(requiredVersionString)	Checks if the installed version is according to the required version. Major, micro and minor version must agree the required level. Example: RequireVersion("1.0.31")
StartRenderer(verbose = false)	Start OpenGL rendering engine (in separate thread); use verbose=True to output information during OpenGL window creation
StopRenderer()	Stop OpenGL rendering engine

SolveStatic(mbs, simulationSettings = exudyn.SimulationSettings(), updateInitialValues = False, storeSolver = True)	Static solver function, mapped from module `solver`; for details on the python interface see Section 5.15; for background on solvers, see Section 10
SolveDynamic(mbs, simulationSettings = exudyn.SimulationSettings(), solverType = exudyn.DynamicSolverType.GeneralizedAlpha, updateInitialValues = False, storeSolver = True)	Dynamic solver function, mapped from module `solver`; for details on the python interface see Section 5.15; for background on solvers, see Section 10
ComputeODE2Eigenvalues(mbs, simulationSettings = exudyn.SimulationSettings(), useSparseSolver = False, numberOfEigenvalues = -1, setInitialValues = True, convert2Frequencies = False)	Simple interface to scipy eigenvalue solver for eigenvalue analysis of the second order differential equations part in mbs, mapped from module `solver`; for details on the python interface see Section 5.15
SetOutputPrecision(numberOfDigits)	Set the precision (integer) for floating point numbers written to console (reset when simulation is started!)
SetLinalgOutputFormatPython(flagPythonFormat)	true: use python format for output of vectors and matrices; false: use matlab format
SetWriteToConsole(flag)	set flag to write (true) or not write to console; default = true

SetWriteToFile(filename, flagWriteToFile = true, flagAppend = 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
	EXAMPLE:
	`exu.SetWriteToConsole(False) #no output to console`
	`exu.SetWriteToFile(filename='testOutput.log', flagWriteToFile=True, flagAppend=False)`
	`exu.Print('print this to file')`
	`exu.SetWriteToFile(", False) #terminate writing to file which closes the file`

SetPrintDelayMilliSeconds(delayMilliSeconds)	add some delay (in milliSeconds) to printing to console, in order to let Spyder process the output; default = 0
Print()	this allows printing via exudyn with similar syntax as in python print(args) except for keyword arguments: print('test=',42); allows to redirect all output to file given by SetWriteToFile(...); does not output in case that SetWriteToConsole is set to false
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
Go()	Creates a SystemContainer SC and a main system mbs

InvalidIndex()	This function provides the invalid index, which depends on the kind of 32-bit, 64-bit signed or unsigned integer; e.g. node index or item index in list; in future, the invalid index may be changed to -1, therefore you should use this variable
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
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 exu.StopRenderer() and can be used for subsequent simulations

4.2 SystemContainer

The SystemContainer is the top level of structures in EXUDYN. The container holds all systems, solvers and all other data structures for computation. Currently, only one container shall be used. In future, multiple containers might be usable at the same time.
Example:

```
import exudyn as exu
SC = exu.SystemContainer()
mbs = SC.AddSystem()
```

function/structure name	**description**
AddSystem()	add a new computational system

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

TimeIntegrationSolve(mainSystem, solverName, simulationSettings)	DEPRECATED, use exu.SolveDynamic(...) instead, see efSectionsec:solver:SolveDynamic! Call time integration solver for given system with solverName ('RungeKutta1'...explicit solver, 'GeneralizedAlpha'...implicit solver); use simulationSettings to individually configure the solver
	EXAMPLE:
	`simSettings = exu.SimulationSettings()`
	`simSettings.timeIntegration.numberOfSteps = 1000`
	`simSettings.timeIntegration.endTime = 2`
	`simSettings.timeIntegration.verboseMode = 1`
	`SC.TimeIntegrationSolve(mbs, 'GeneralizedAlpha', simSettings)`

StaticSolve(mainSystem, simulationSettings)	DEPRECATED, use exu.SolveStatic(...) instead efSection-sec:solver:SolveStatic! Call solver to compute a static solution of the system, considering acceleration and velocity coordinates to be zero (initial velocities may be considered by certain objects)
	EXAMPLE:
	`simSettings = exu.SimulationSettings()`
	`simSettings.staticSolver.newton.relativeTolerance = 1e-6`
	`SC.StaticSolve(mbs, simSettings)`

GetRenderState()	Get dictionary with current render state (openGL zoom, modelview, etc.)
	EXAMPLE:
	`SC = exu.SystemContainer()`
	`renderState = SC.GetRenderState()`
	`print(renderState['zoom'])`

SetRenderState(renderState)	Set current render state (openGL zoom, modelview, etc.) with given dictionary; usually, this dictionary has been obtained with GetRenderState
	EXAMPLE:
	`SC = exu.SystemContainer()`
	`SC.SetRenderState(renderState)`

WaitForRenderEngineStopFlag()	Wait for user to stop render engine (Press 'Q' or Escape-key)
RenderEngineZoomAll()	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)
GetCurrentMouseCoordinates(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 GetCurrent-MouseCoordinates(useOpenGLcoordinates=True) to obtain OpenGLcoordinates of projected plane
Reset()	delete all systems and reset SystemContainer (including graphics)
NumberOfSystems()	obtain number of systems available in system container
GetSystem(systemNumber)	obtain systems with index from system container

4.3 MainSystem

This is the structure which defines a (multibody) system. In C++, there is a MainSystem (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'. It can be created, visualized and computed. Use the following functions for system manipulation.

Usage:

```
import exudyn as exu
SC = exu.SystemContainer()
mbs = SC.AddSystem()
```

function/structure name	**description**
Assemble()	assemble items (nodes, bodies, markers, loads, ...); Calls CheckSystemIntegrity(...), AssembleCoordinates(), AssembleLTGLists(), and AssembleInitializeSystemCoordinates()
AssembleCoordinates()	assemble coordinates: assign computational coordinates to nodes and constraints (algebraic variables)
AssembleLTGLists()	build local-to-global (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
Reset()	reset all lists of items (nodes, bodies, markers, loads, ...) and temporary vectors; deallocate memory
WaitForUserToContinue()	interrupt further computation until user input –> 'pause' function
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()	set the current stop simulation flag; set to false, in order to continue a previously user-interrupted simulation

SetPreStepUserFunction()	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
	EXAMPLE:
	`def PreStepUserFunction(mbs, t):`
	`    print(mbs.systemData.NumberOfNodes())`
	`    if(t>1):`
	`        return False`
	`    return True`
	`mbs.SetPreStepUserFunction(PreStepUserFunction)`

__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,...)

4.3.1 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.

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)`

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 = ConfigurationType.Current)	get the ouput of the node specified with the OutputVariableType; default configuration = 'current'; output may be scalar or array (e.g. displacement vector) EXAMPLE: `mbs.GetNodeOutput(nodeNumber=0, variableType='exu.OutputVariable.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)`
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
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

4.3.2 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.

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)`

GetObjectNumber(objectName)	get object's number by name (string) EXAMPLE: `n = mbs.GetObjectNumber('heavy object')`
GetObject(objectNumber)	get object's dictionary by object number (type ObjectIndex) EXAMPLE: `objectDict = mbs.GetObject(0)`
ModifyObject(objectNumber, objectDict)	modify object's dictionary by object number (type ObjectIndex) 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)	get object's current output variable from object number (type ObjectIndex) and OutputVariableType; can only be computed for exu.ConfigurationType.Current configuration!

GetObjectOutputBody(objectNumber, variableType, localPosition, configuration = ConfigurationType.Current)	get body's output variable from object number (type ObjectIndex) and OutputVariableType **EXAMPLE**: `u = mbs.GetObjectOutputBody(objectNumber = 1, variableType = exu.OutputVariableType.Position, localPosition=[1,0,0], configuration = exu.ConfigurationType.Initial)`
GetObjectOutputSuperElement(objectNumber, variableType, meshNodeNumber, configuration = 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
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

4.3.3 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).

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)`

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

4.3.4 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.

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)`

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

4.3.5 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.

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)`

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 = ConfigurationType.Current)	get sensors's values for configuration; can be a scalar or vector-valued return value!
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

4.4 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.
Usage:

```
#obtain current ODE2 system vector (e.g.  after static simulation finished):
u = mbs.systemData.GetODE2Coordinates()
#set initial ODE2 vector for next simulation:
mbs.systemData.SetODE2Coordinates(coordinates=u,configurationType=exu.ConfigurationType.Initial)
```

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())`
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)`
GetCurrentTime()	DEPRICATED; get current (simulation) time; time is updated in time integration solvers and in static solver; use this function e.g. during simulation to define time-dependent loads EXAMPLE: `mbs.systemData.GetCurrentTime()`
SetVisualizationTime()	DEPRICATED; set time for render window (visualization) EXAMPLE: `mbs.systemData.SetVisualizationTime(1.3)`
Info()	print detailed system information for every item; for short information use print(mbs) EXAMPLE: `mbs.systemData.Info()`

4.4.1 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**
GetODE2Coordinates(configuration = exu.ConfigurationType.Current)	get ODE2 system coordinates (displacements) for given configuration (default: exu.Configuration.Current) EXAMPLE: `uCurrent = mbs.systemData.GetODE2Coordinates()`
SetODE2Coordinates(coordinates, configuration = exu.ConfigurationType.Current)	set ODE2 system coordinates (displacements) 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)	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)	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)	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 (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)	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)	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)`

4.4.2 SystemData: Get object local-to-global (LTG) coordinate mappings

This section provides access functions the LTG-lists for every object (body, constraint, ...) in the system.

function/structure name	**description**
GetObjectLTGODE2(objectNumber)	get 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 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 local-to-global coordinate mapping (list of global coordinate indices) for algebraic equations (AE) coordinates; only available after Assemble() **EXAMPLE**: `ltgObject4 = mbs.systemData.GetObjectLTGODE2(4)`
GetObjectLTGData(objectNumber)	get local-to-global coordinate mapping (list of global coordinate indices) for data coordinates; only available after Assemble() **EXAMPLE**: `ltgObject4 = mbs.systemData.GetObjectLTGData(4)`

4.5 Type definitions

This section defines a couple of structures, 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.

Conversion to integer is possible:

```
x = int(exu.OutputVariableType.Displacement)
```

and also conversion from integer:

```
varType = exu.OutputVariableType(8)
```

4.5.1 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.

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
Velocity	measure (translational) velocity of node or object
Acceleration	measure (translational) acceleration of node or object
RotationMatrix	measure rotation matrix of rigid body node or object
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
Rotation	measure, e.g., scalar rotation of 2D body, Euler angles of a 3D object or rotation within a joint
Coordinates	measure the coordinates of a node or object; coordinates usually just contain displacements, but not the position values
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 force, e.g., in joint or beam (resultant force)
Torque	measure torque, e.g., in joint or beam (resultant couple/-moment)
Strain	measure strain, e.g., axial strain in beam
Stress	measure stress, e.g., axial stress in beam
Curvature	measure curvature; may be scalar or vectorial: twist and curvature
DisplacementLocal	measure local displacement, e.g. in local joint coordinates
VelocityLocal	measure local (translational) velocity, , e.g. in local joint coordinates
ForceLocal	measure local force, e.g., in joint or beam (resultant force)
TorqueLocal	measure local torque, e.g., in joint or beam (resultant couple/moment)
ConstraintEquation	evaluates constraint equation (=current deviation or drift of constraint equation)
EndOfEnumList	this marks the end of the list, usually not important to the user

4.5.2 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

4.5.3 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; allows to set variables also for Newmark and trapezoidal implicit index 2 solvers
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"; (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

4.5.4 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

4.5.5 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 systems; specifically, memory allocation is performed during a factorization process

Chapter 5

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 usually 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.

5.1 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
```

5.2 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
- **notes**: It is recommended to call ClearWorkspace() at the very beginning of your models
- **example**:

```
#do this at the very beginning!
from exudyn.utilities import ClearWorkspace
ClearWorkspace()       #clear old SC and mbs variables
#now import modules
import exudyn as exu
from exudyn.itemInterface import *
SC = exu.SystemContainer()
mbs = SC.AddSystem()
```

---

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

---

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

---

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

---

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

---

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 = s*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], ...]

---

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

---

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 containing the skew-symmetric matrix computed from v: $\begin{bmatrix} 0 & -v[2] & v[1] \\ v[2] & 0 & -v[0] \\ -v[1] & v[0] & 0 \end{bmatrix}$

---

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

5.3 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.

Author: Johannes Gerstmayr, Stefan Holzinger

Date: 2020-03-10 (created)

Notes: CSR matrices contain 3 numbers per row: [row, column, value]

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**: convert internal compressed CSR to scipy.sparse csr matrix

---

def **ScipySparseCSRtoCSR** (*scipyCSR*)

- **function description**: convert scipy.sparse csr matrix to internal compressed CSR

---

def **ResortIndicesOfCSRmatrix** (*mXXYYZZ*, *numberOfRows*)

- **function description**:

  resort indices of given CSR matrix in XXXYYYZZZ format to XYZXYZXYZ format; numberOfRows must be equal to columns

  needed for import from NGsolve

---

def **ConvertHexToTrigs** (*nodeNumbers*)

- **function description**:

  convert list of Hex8/C3D8 element with 8 nodes in nodeNumbers into triangle-List also works for Hex20 elements, but does only take the corner nodes!

---

def **ConvertDenseToCompressedRowMatrix** (*denseMatrix*)

- **function description**: convert numpy.array dense matrix to (internal) compressed row format

---

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.

5.3.1 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
- **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)

5.3.2 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, 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

  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 **AddObjectFFRFreducedOrderWithUserFunctions** (*self*, *exu*, *mbs*, *positionRef*= [0,0,0], *eulerParametersRef*= [1,0,0,0], *initialVelocity*= [0,0,0], *initialAngularVelocity*= [0,0,0], *gravity*= [0,0,0], *UFforce*= 0, *UFmassMatrix*= 0, *massProportionalDamping*= 0, *stiffnessProportionalDamping*= 0, *color*= [0.1,0.9,0.1,1.])

- **classFunction**: add according nodes, objects and constraints for ObjectFFRFreducedOrder object to MainSystem mbs
- **input**:

  exu: the exudyn module

  mbs: a MainSystem object

  positionRef: reference position of created ObjectFFRFreducedOrder (set in rigid body node underlying to ObjectFFRFreducedOrder)

  eulerParametersRef: reference euler parameters 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)

  initialAngularVelocity: initial angular velocity 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: provide a user function, which computes the quadratic velocity vector and applied forces; usually this function reads like:

  ```
  def UFforceFFRFreducedOrder(mbs, t, qReduced, qReduced_t):
      return cms.UFforceFFRFreducedOrder(exu, mbs, t, qReduced, qReduced_t)
  ```

  UFmassMatrix: provide a user function, which computes the quadratic velocity vector and applied forces; usually this function reads like:

  ```
  def UFmassFFRFreducedOrder(mbs, t, qReduced, qReduced_t):
      return cms.UFmassFFRFreducedOrder(exu, mbs, t, qReduced, qReduced_t)
  ```

  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

  color: provided as list of 4 RGBA values

---

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!

5.3.3 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, 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()`

---

def **SaveToFile** (*self*, *fileName*)

- **classFunction**: save all data (nodes, elements, ...) to a data filename; this function is much faster than the text-based import functions
- **input**: use filename without ending ==> ".npy" will be added

---

def **LoadFromFile** (*self*, *fileName*)

- **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**: use filename without ending ==> ".npy" will be added

---

def **ImportFromAbaqusInputFile** (*self*, *fileName*, *typeName*= 'Part', *name*= 'Part-1', *verbose*= 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
  only works for Hex8, Hex20, Tet4 and Tet10 (C3D4, C3D8, C3D10, C3D20) elements
  return node numbers as numpy array

---

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 **ImportMeshFromNGsolve** (*self*, *mesh*, *density*, *youngsModulus*, *poissonsRatio*, *verbose*= False)

- **classFunction**: import mesh from NETGEN/NGsolve and setup mechanical problem
- **input**:

  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

  verbose: set True to print out some status information
- **notes**:

  The interface to NETGEN/NGsolve has been created together with Joachim Schöberl, main developer of NETGEN/NGsolve; 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 **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

---

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

  if not found, it returns an empty list

---

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 **GetSurfaceTriangles** (*self*)

– **classFunction**: return surface trigs as node number list (for drawing in EXUDYN)

---

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 one type ('Hex8') of elements

---

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 **ComputeEigenmodes** (*self*, *nModes*, *excludeRigidBodyModes*= 0, *useSparseSolver*= True)

- **classFunction**:

  compute nModes smallest eigenvalues and eigenmodes from mass and stiffnessMatrix

  store mode vector in modeBasis, but exclude a number of 'excludeRigidBodyModes' rigid body modes from modeBasis

  if excludeRigidBodyModes > 0, then the computed modes is nModes + excludeRigidBodyModes, from which excludeRigidBodyModes smallest eigenvalues are excluded

---

def **GetEigenFrequenciesHz** (*self*)

- **classFunction**: return list of eigenvalues in Hz of previously computed eigenmodes

---

def **ComputeCampbellDiagram** (*self*, *terminalFrequency*, *nEigenfrequencies*= 10, *frequencySteps*= 25, *rotationAxis*= 2, *plotDiagram*= False, *verbose*= 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

- **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)

5.4 Module: graphicsDataUtilities

Utility functions for visualization, which provides functions for basic shapes like cube, cylinder, sphere, solid of revolution. Functions generate dictionaries which contain line, text or triangle primitives for drawing in Exudyn using OpenGL.

Author: Johannes Gerstmayr

Date: 2020-07-26 (created)

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, color4lightred, color4lightgreen, color4steelblue, color4grey, color4darkgrey, color4lightgrey, color4white

Additionally, a list of 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 **MergeGraphicsDataTriangleList** (*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

---

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

---

def **GraphicsDataOrthoCubeLines** (*xMin*, *yMin*, *zMin*, *xMax*, *yMax*, *zMax*, *color*= [0.,0.,0.,1.])

- **function description**: generate graphics data for orthogonal cube 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

---

def **GraphicsDataOrthoCube** (*xMin*, *yMin*, *zMin*, *xMax*, *yMax*, *zMax*, *color*= [0.,0.,0.,1.])

- **function description**: generate graphics data for orthogonal 3D cube with min and max dimensions
- **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

---

def **GraphicsDataOrthoCubePoint** (*centerPoint*, *size*, *color*= [0.,0.,0.,1.])

- **function description**: generate graphics data forfor orthogonal 3D cube with center point and size
- **input**: center point and size of cube (as 3D list or np.array); color provided as list of 4 RGBA values
- **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects

---

def **GraphicsDataCube** (*pList*, *color*= [0.,0.,0.,1.], *faces*= [1,1,1,1,1,1])

- **function description**: generate graphics data for general cube 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
- **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects

---

def **GraphicsDataSphere** (*point*, *radius*, *color*= [0.,0.,0.,1.], *nTiles*= 8)

- **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
- **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects

---

def **GraphicsDataCylinder** (*pAxis*, *vAxis*, *radius*, *color*= [0.,0.,0.,1.], *nTiles*= 16, *angleRange*= [0,2*np.pi], *lastFace*= True, *cutPlain*= 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

  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 ...

  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

---

def **GraphicsDataRigidLink** (*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

---

def **GraphicsDataFromSTLfileTxt** (*fileName*, *color*= [0.,0.,0.,1.], *verbose*= False)

- **function description**: generate graphics data from STL file (text format!) and use color for visualization
- **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
- **output**: interchanged 2nd and 3rd component of list

---

def **GraphicsDataSolidOfRevolution** (*pAxis*, *vAxis*, *contour*, *color*= [0.,0.,0.,1.], *nTiles*= 16, *smoothContour*= False, ****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
  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 = GraphicsDataSolidOfRevolution(pAxis=[0,0.5,0], vAxis=[1,0,0],
                                     contour=contour, color=color4red,
                                     alternatingColor=color4grey)
#draw torus:
contour=[]
r = 0.2 #small radius of torus
R = 0.5 #big radius 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 = GraphicsDataSolidOfRevolution(pAxis=[0,0.5,0], vAxis=[1,0,0],
                                     contour=contour, color=color4red,
                                     nTiles = 32*2, smoothContour=True)
```

---

def **GraphicsDataArrow** (*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

---

def **GraphicsDataBasis** (*origin*= [0,0,0], *length*= 1, *colors*= [color4red, color4green, color4blue], *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)
  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

---

def **GraphicsDataQuad** (*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 = GraphicsDataQuad([[-8, 0, -8],[ 8, 0, -8,],[ 8, 0, 8],[-8, 0, 8]],
                          color4darkgrey, nTiles=8,
                          alternatingColor=color4lightgrey)
oGround=mbs.AddObject(ObjectGround(referencePosition=[0,0,0],
                      visualization=VObjectGround(graphicsData=[plane])))
```

---

def **GraphicsDataCheckerBoard** (*point*= [0,0,0], *normal*= [0,0,1], *size*= 1, *color*= color4lightgrey, *alternatingColor*= color4lightgrey2, *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

- **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects

- **example**:

```
plane = GraphicsDataQuad([[-8, 0, -8],[ 8, 0, -8,],[ 8, 0, 8],[-8, 0, 8]],
                         color4darkgrey, nTiles=8,
                         alternatingColor=color4lightgrey)
oGround=mbs.AddObject(ObjectGround(referencePosition=[0,0,0],
                      visualization=VObjectGround(graphicsData=[plane])))
```

---

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)
- **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)
 print(tri.simplices)
```

---

def **GraphicsDataSolidExtrusion** (*vertices, segments, height, rot*= np.diag([1,1,1]), *pOff*= [0,0,0], *color*= [0,0,0,1])

- **function description**:
  create graphicsData for solid extrusion based on 2D points and segments;
  additional transformations are possible to translate and rotate
- **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

– **output**: graphicsData dictionary, to be used in visualization of EXUDYN objects

5.5 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 **EditDictionaryWithTypeInfo** (*dictionaryData*, *exu*= None, *dictionaryName*= 'edit')

– **function description**: edit dictionaryData and return modified (new) dictionary

– **input**:

dictionaryData: dictionary obtained from SC.visualizationSettings.GetDictionaryWithTypeInfo()

exu: exudyn module

dictionaryName: name displayed in dialog

– **output**: returns modified dictionary, which can be used, e.g., for SC.visualizationSettings.SetDictionary(...)

5.6 Module: interactive

Utilities for interactive simulation and results monitoring

Author: Johannes Gerstmayr

Date: 2021-01-17 (created)

5.6.1 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, 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, 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
       #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
       'sizeInches':(12,12)}        #specific x and y size of figure in inches (
           using 100 dpi)
```

def **__init__** (*self*, *mbs*, *simulationSettings*, *simulationFunction*, *dialogItems*, *plots*= [], *period*= 0.04, *realtimeFactor*= 1, *userStartSimulation*= False, *title*= '', *showTime*= False, *fontSize*= 12)

- **classFunction**: initialize an InteractiveDialog
- **input**:
  - *mbs*: a multibody system to be simulated
  - *simulationSettings*: exu.SimulationSettings() according to user settings
  - *simulationFunction*: a function which is called before a simulation for the short period is started (e.g, assign special values, etc.)
  - *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
  - *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
  - *userStartFunction*: 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
- **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

---

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

5.7 Module: lieGroupBasics

Lie group methods and formulas for Lie group integration.

Author: Stefan Holzinger

Date: 2020-09-11

References:

[Bruels2011] Bruels, O., Cardona, A.: Two Lie Group Formulations for Dynamic Multibody Systems With Large Rotations, Proceedings of the ASME 2011 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference IDETC/CIE 2011, August 28-31, 2011, Washington, USA

[Sonneville2014] Sonneville, V., Cardona A., Bruels, O.: Geometrically exact beam finite element formulated on the special Euclidean group SE(3)

[Sonneville2017] Sonneville, Bruels, O., Bauchau, O.A.: Interpolation schemes for geometrically exact beams: A motion approach. International Journal for Numerical Methods in Engineering 112, 1129-1153 (2017)

[Terze2016] Terze, Z., Mueller, A., Zlatar, D.: Singularity-free time integration of rotational quaternions using non-redundant ordinary differential equations. Multibody System Dynamics 38(3), 201-225 (2016)

[Henderson1977] Henderson, D.: Euler angles, quaternions, and transformation matrices for space shuttle analysis. Tech. rep., NASA (1977)

[Holzinger2020] Holzinger, S., Gerstmayr, J.: Explicit time integration of multibody systems modelled with three rotation parameters. In: Proceedings of the ASME 2020 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference IDETC/CIE2020, August 17-19. Virtual, Online (2020)

[Holzinger2021] Holzinger, S., Gerstmayr, J.: Time integration of rigid bodies modelled with three rotation parameters, Multibody System Dynamics 2021

[Simo1988] Simo, J.C., Vu-Quoc, L.: On the dynamics in space of rods undergoing large motions-A geometrically exact approach. Computer methods in applied mechanics and engineering 66, 125-161 (1988)

[Mueller2017] Mueller, A.: Coordinate Mappings for Rigid Body Motions. Journal of Computational and Nonlinear Dynamics 12(2), 10 (2017)

def **Sinc** (*x*)

- **function description**: compute the cardinal sine function in radians
- **input**: scalar float or int value
- **output**: float value in radians

---

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

---

def **R3xSO3Matrix2RotationMatrix** (*G*)

- **function description**: computes 3x3 rotation matrix from 7x7 R3xSO(3) matrix, see [3]
- **input**: G: 7x7 matrix as np.array
- **output**: 3x3 rotation matrix as np.array

---

def **R3xSO3Matrix2Translation** (*G*)

- **function description**: computes translation part of R3xSO(3) matrix, see [3]
- **input**: G: 7x7 matrix as np.array
- **output**: 3D vector as np.array containg translational part of R3xSO(3)

---

def **R3xSO3Matrix** (*x*, *R*)

- **function description**: builds 7x7 matrix as element of the Lie group R3xSO(3), see [3]
- **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

---

def **ExpSO3** (*Omega*)

- **function description**: compute the matrix exponential map on the Lie group SO(3), see [18]
- **input**: 3D rotation vector as np.array
- **output**: 3x3 matrix as np.array

---

def **ExpS3** (*Omega*)

- **function description**: compute the quaternion exponential map on the Lie group S(3), see [25, 18]
- **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

---

def **LogSO3** (*R*)

- **function description**: compute the matrix logarithmic map on the Lie group SO(3), see [24, 23]
- **input**: 3x3 rotation matrix as np.array
- **output**: 3x3 skew symmetric matrix as np.array

---

def **TExpSO3** (*Omega*)

- **function description**: compute the tangent operator corresponding to ExpSO3, see [3]
- **input**: 3D rotation vector as np.array
- **output**: 3x3 matrix as np.array

---

def **TExpSO3Inv** (*Omega*)

- **function description**:

  compute the inverse of the tangent operator TExpSO3, see [24]

  this function was improved, see coordinateMaps.pdf by Stefan Holzinger
- **input**: 3D rotation vector as np.array
- **output**: 3x3 matrix as np.array

---

def **ExpSE3** (*x*)

- **function description**: compute the matrix exponential map on the Lie group SE(3), see [3]
- **input**: 6D incremental motion vector as np.array
- **output**: 4x4 homogeneous transformation matrix as np.array

---

def **LogSE3** (*H*)

- **function description**: compute the matrix logarithm on the Lie group SE(3), see [24]
- **input**: 4x4 homogeneous transformation matrix as np.array
- **output**: 4x4 skew symmetric matrix as np.array

---

def **TExpSE3** (*x*)

- **function description**: compute the tangent operator corresponding to ExpSE3, see [3]
- **input**: 6D incremental motion vector as np.array
- **output**: 6x6 matrix as np.array

---

def **TExpSE3Inv** (*x*)

- **function description**: compute the inverse of tangent operator TExpSE3, see [24]
- **input**: 6D incremental motion vector as np.array
- **output**: 6x6 matrix as np.array

---

def **ExpR3xSO3** (*x*)

- **function description**: compute the matrix exponential map on the Lie group R3xSO(3), see [3]
- **input**: 6D incremental motion vector as np.array
- **output**: 7x7 matrix as np.array

---

def **TExpR3xSO3** (*x*)

- **function description**: compute the tangent operator corresponding to ExpR3xSO3, see [3]
- **input**: 6D incremental motion vector as np.array
- **output**: 6x6 matrix as np.array

---

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

---

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

---

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

---

def **CompositionRuleDirectProductR3AndR3RotVec** (*q0*, *incrementalMotionVector*)

- **function description**:

  compute composition operation for pairs in the group obtained from the direct product of R3 and R3, see [11]

  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

---

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

---

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

---

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

---

def **CompositionRuleForEulerParameters** (*q*, *p*)

- **function description**:
  compute composition operation for Euler parameters (unit quaternions)
  this composition operation is quaternion multiplication, see [25]
- **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

---

def **CompositionRuleForRotationVectors** (*v0*, *Omega*)

- **function description**: compute composition operation for rotation vectors v0 and Omega, see [16]
- **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

---

def **CompositionRuleRotXYZAnglesRotationVector** (*alpha0*, *Omega*)

- **function description**: compute composition operation for RotXYZ angles, see [16]
- **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

5.8 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,

$$f(v) = \begin{cases} (\mu_d + \mu_{s_{off}})v, & \text{if} \quad |v| <= v_{reg} \\ \mathrm{Sign}(v)\left(\mu_d + \mu_{s_{off}} \mathrm{e}^{-(|v|-v_{reg})/v_{exp}} + \mu_v(|v| - v_{reg})\right), & \text{else} \end{cases}$$

- **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)

---

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: Fores et al. [7], Qian et al. [21]

5.9 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 utilities.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'], 'variableType',

---

def **PlotSensor** (*mbs*, *sensorNumbers*, *components*= 0, ***kwargs*)

- **function description**: helper for matplotlib in order to easily visualize sensor output

- **input**:

  mbs: must be a valid MainSystem (mbs)

  sensorNumbers: consists of one or a list of sensor numbers (type SensorIndex) as returned by the mbs function AddSensor(...)

  components: consists of one or a list of components according to the component of the sensor to be plotted;

  **kwargs*: additional options, e.g.:

  xLabel -> string for text at x-axis (otherwise time is used)

  yLabel -> string for text at y-axis (otherwise outputvalues are used)

  fontSize -> default = 16, which is a little bit larger than default (12)

- **output**: plots the sensor data

- **example**:

```
s0=mbs.AddSensor(SensorNode(nodeNumber=0))
  s1=mbs.AddSensor(SensorNode(nodeNumber=1))
  Plot(mbs, s0, 0)
  Plot(mbs, sensorNumbers=[s0,s1], components=[0,2], xlabel='time in seconds')
```

---

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.signal

  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

5.10 Module: processing

The processing module supports multiple execution of EXUDYN models. It includes parameter variation and (genetic) optimization functionality.

Author: Johannes Gerstmayr

Date: 2020-11-17

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 **ProcessParameterList** (*parameterFunction*, *parameterList*, *addComputationIndex*, *useMultiProcessing*, ***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, e.g., ['mass': 10, 'mass':20, ...]

  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.!

  numberOfThreads: default: same as number of cpus (threads); used for multiprocessing lib;
- **output**: returns values containing the results according to parameterList
- **notes**: options are passed from Parametervariation

---

def **ParameterVariation** (*parameterFunction*, *parameters*, *useLogSpace*= False, *debugMode*= False, *addComputationIndex*= False, *useMultiProcessing*= False, *showProgress*= True, ***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 triple of (begin, end and steps) same as in np.linspace(...), e.g. 'mass':(10,50,10) for a mass varied from 10 to 50, using 10 steps *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)

numberOfThreads: default: same as number of cpus (threads); used for multiprocessing lib;

- **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**:

```
ParameterVariation(parameters={'mass':(1,10,10), 'stiffness':(1000,10000,10)},
    parameterFunction=Test, useMultiProcessing=True)
```

---

def **GeneticOptimization** (*objectiveFunction*, *parameters*, *populationSize*= 100, *numberOfGenerations*= 10, *elitistRatio*= 0.1, *crossoverProbability*= 0.25, *crossoverAmount*= 0.5, *rangeReductionFactor*= 0.7, *distanceFactor*= 0.1, *debugMode*= False, *addComputationIndex*= False, *useMultiProcessing*= False, *showProgress*= True, ***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

  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

  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 *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

numberOfChildren: (DEPRECATED, UNUSED) number childrens of surviving population

survivingIndividuals: (DEPRECATED) number of surviving individuals after children are born

- **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)})
```

---

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

5.11 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 **ComputeOrthonormalBasis** (*vector0*)

- **function description**: compute orthogonal basis vectors (normal1, normal2) for given vector0 (non-unique solution!); if vector0 == [0,0,0], then any normal basis is returned

---

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)

---

def **Skew** (*vector*)

- **function description**: compute skew symmetric 3x3-matrix from 3x1- or 1x3-vector

---

def **Skew2Vec** (*skew*)

- **function description**: convert skew symmetric matrix m to vector

---

def **ComputeSkewMatrix** (*v*)

- **function description**: compute (3 x 3*n) skew matrix from (3*n) vector

---

def **EulerParameters2G** (*eulerParameters*)

- **function description**: convert Euler parameters (ep) to G-matrix (=$\partial\boldsymbol{\omega}/\partial\mathbf{p}_t$)

- **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

---

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

---

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

---

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 $\boldsymbol{\omega} = \mathbf{G}\mathbf{p}_t$ ==> $\mathbf{G}^T\boldsymbol{\omega} = \mathbf{G}^T \cdot \mathbf{G} \cdot \mathbf{p}_t$ ==> $\mathbf{G}^T\mathbf{G} = 4(\mathbf{I}_{4x4} - \mathbf{p}\cdot\mathbf{p}^T)\mathbf{p}_t = 4(\mathbf{I}_{4x4})\mathbf{p}_t$
- **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 [22]
- **input**: 3D rotation vector as list or np.array
- **output**: 3x3 rotation matrix as np.array

---

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

---

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

---

def **RotXYZ2RotationMatrix** (*rot*)

- **function description**: compute rotation matrix from consecutive xyz rotations (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

---

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

---

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 [15]
- **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 **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

---

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

---

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

---

def **HomogeneousTransformation** (*A*, *r*)

- **function description**: compute homogeneous transformation matrix from rotation matrix A and translation vector r

---

def **HTtranslate** (*r*)

- **function description**: homogeneous transformation for translation with vector r

---

def **HT0** ()

- **function description**: identity homogeneous transformation:

---

def **HTrotateX** (*angle*)

- **function description**: homogeneous transformation for rotation around axis X (first axis)

---

def **HTrotateY** (*angle*)

- **function description**: homogeneous transformation for rotation around axis X (first axis)

---

def **HTrotateZ** (*angle*)

- **function description**: homogeneous transformation for rotation around axis X (first axis)

---

def **HT2translation** (*T*)

- **function description**: return translation part of homogeneous transformation

---

def **HT2rotationMatrix** (*T*)

- **function description**: return rotation matrix of homogeneous transformation

---

def **InverseHT** (*T*)

- **function description**: return inverse homogeneous transformation such that inv(T)*T = np.eye(4)

---

def **AddRigidBody** (*mainSys*, *inertia*, *nodeType*, *position*= [0,0,0], *velocity*= [0,0,0], *rotationMatrix*= [], *rotationParameters*= [], *angularVelocity*= [0,0,0], *gravity*= [0,0,0], *graphicsDataList*= [])

- **function description**: 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**:

  position: reference position as list or numpy array with 3 components

  velocity: initial translational velocity as list or numpy array with 3 components

  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

  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]

5.11.1 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**: It is recommended to start with com=[0,0,0] and then to use Translated(...) and Rotated(...) to get transformed inertia parameters.
- **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))

- classFunction: initialize RigidBodyInertia with scalar mass, 3x3 inertiaTensor and center of mass com

---

def **__add__** (*self*, *otherBodyInertia*)

- classFunction:
  add (+) operator allows adding another inertia information with SAME local coordinate system
  only inertias with same center of rotation can be added!
- example:
```
J = InertiaSphere(2,0.1) + InertiaRodX(1,2)
```

---

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 **GetInertia6D** (*self*)

- classFunction: get vector with 6 inertia components (Jxx, Jyy, Jzz, Jyz, Jxz, Jxy) as needed in ObjectRigidBody

5.11.2 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])
```

5.11.3 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]

5.11.4 CLASS InertiaMassPoint(RigidBodyInertia) (in module rigidBodyUtilities)

class description: create RigidBodyInertia with moment of inertia and mass of mass point with 'mass'; inertia w.r.t. center of mass, com=[0,0,0]

5.11.5 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]

5.11.6 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

5.11.7 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]

5.12 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

Author: Johannes Gerstmayr

Date: 2020-04-14

Example: see ComputeJointHT for the definition of the 'robot' dictionary.

def **DH2HT** (*DHparameters*)

- **function description**: compute homogeneous transformation HT from standard DH-parameters

---

def **ComputeJointHT** (*robot, configuration*)

- **function description**: compute list of homogeneous transformations HT from base to every joint for given configuration
- **example**:

```
link0={'stdDH':[0,0,0,np.pi/2],
         'mass':20,  #not needed!
         'inertia':np.diag([1e-8,0.35,1e-8]), #w.r.t. COM!
         'COM':[0,0,0]}
link1={'stdDH':[0,0,0.4318,0],
         'mass':17.4,
         'inertia':np.diag([0.13,0.524,0.539]), #w.r.t. COM!
         'COM':[-0.3638, 0.006, 0.2275]}
robot={'links':[link0, link1],
         'jointType':[1,1], #1=revolute, 0=prismatic
         'base':{'HT':HT0()},
         'tool':{'HT':HTtranslate([0,0,0.1])},
         'gravity':[0,0,9.81],
         'referenceConfiguration':[0]*2 #reference configuration for bodies; at
            which the robot is built
         }
HTlist = ComputeJointHT(robot, [np.pi/8]*2)
```

---

def **ComputeCOMHT** (*robot, HT*)

- **function description**: compute list of homogeneous transformations HT from base to every COM using HT list from ComputeJointHT

---

def **ComputeStaticTorques** (*robot, HT*)

- **function description**: compute list joint torques for serial robot under gravity (gravity and mass as given in robot)

---

def **Jacobian** (*robot*, *HT*, *toolPosition*= [], *mode*= 'all')

- **function description**: compute jacobian for translation and rotation at toolPosition using joint HT

---

def **SerialRobot2MBS** (*mbs*, *robot*, *jointLoadUserFunctionList*, *baseMarker*, **args*, ***kwargs*)

- **function description**:

  add items to existing mbs from the robot structure, a baseMarker (can be ground object or body)

  and the user function list for the joints; there are options that can be passed as args / kwargs, which can contains options as described below. For details, see the python file and `serialRobotTest.py` in TestModels
- **input**:

  mbs: the multibody system, which will be extended

  robot: the robot model as dictionary, described in function ComputeJointHT

  jointLoadUserFunctionList: a list of user functions for actuation of joints according to a LoadTorqueVector userFunction, see serialRobotTest.py as an example; can be empty list

  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

  rotationMarkerBase: used in Generic joint between first joint and base; note, that for moving base, the static compensation does not work (base rotation must be updated)

  showCOM: a list of 3 floats [a,b,c], indicates to show center of mass as rectangular block with size [a,b,c]

  bodyAlpha: a float value in range [0..1], adds transparency to links if value < 1

  toolGraphicsSize: list of 3 floats [sx,sy,sz], giving the size of the tool for graphics representation; set sx=0 to disable tool drawing or do not provide this optional variable

  drawLinkSize: draw parameters for links as list of 3 floats [r,w,0], r=radius of joint, w=radius of link, set r=0 to disable link drawing

  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
- **output**: the function returns a dictionary containing information on nodes, bodies, joints, markers, torques, for every joint

5.13 Module: roboticsSpecial

Library for additional support functions for robotics; The library is built on standard Denavit-Hartenberg Parameters and Homogeneous Transformations (HT) to describe transformations and coordinate systems

Author: Martin Sereinig

Date: 2020-12-08

def **VelocityManipulability** (*robot*, *HT*, *mode*)

- **function description**: compute velocity manipulability measure for given pose (homogenious transformation)
- **input**:

  robot: robot structure

  HT: actual pose as hoogenious transformaton matrix

  mode: rotational or translational part of the movement
- **output**: velocity manipulability measure as scalar value, defined as $\sqrt{(det(JJ^T))}$
- **notes**: compute velocity dependent manipulability definded by Yoshikawa, see [27]

---

def **ForceManipulability** (*robot*, *HT*, *mode*)

- **function description**: compute force manipulability measure for given pose (homogenious transformation)
- **input**:

  robot: robot structure

  HT: actual pose as hoogenious transformaton matrix

  mode: rotational or translational part of the movement
- **output**: force manipulability measure as scalar value, defined as $\sqrt{((det(JJ^T))^{-1})}$
- **notes**: compute force dependent manipulability definded by Yoshikawa, see [27]

---

def **StiffnessManipulability** (*robot*, *JointStiffness*, *HT*, *mode*)

- **function description**: compute cartesian stiffness measure for given pose (homogenious transformation)
- **input**:

  robot: robot structure

  JointStiffness: joint stiffness matrix

  HT: actual pose as hoogenious transformaton matrix

  mode: rotational or translational part of the movement
- **output**:

  stiffness manipulability measure as scalar value, defined as minimum Eigenvalaue of the Cartesian stiffness matrix

  Cartesian stiffness matrix
- **notes**:

– **status**: this function is **currently under development** and under testing!

---

def **JointJacobian** (*robot*, *HT*)

– **function description**: compute joint jacobian for each frame for given pose (homogenious transformation)

– **input**:

  robot: robot structure

  HT: actual pose as hoogenious 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 [26]

– **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 [6]

---

def **MassMatrix** (*robot*, *HT*, *jointJacobian*)

– **function description**: compute mass matrix from jointJacobian

– **input**:

  robot: robot structure

  HT: actual pose as hoogenious transformaton matrix

  jointJacobian: provide list of jacobians as provided by function JointJacobian(...)

– **output**: MM: Mass matrix

– **notes**:

  Mass Matrix calculation calculated in joint coordinates regarding (std) DH parameter:

  ** Dynamic equations in minimal coordinates as described in Mehrkoerpersysteme by Woernle, [26], p206, eq6.90.

  ** Caclulations in link coordinate system at the end of each link

---

def **DynamicManipulability** (*robot*, *HT*, *MassMatrix*, *Tmax*, *mode*)

– **function description**: compute dynamic manipulability measure for given pose (homogenious transformation)

– **input**:

  robot: robot structure

HT: actual pose as hoogenious transformaton matrix

Tmax: maximum joint torques

mode: rotational or translational part of the movement

MassMatrix: Mass (inertia) Maxtrix provided by the function MassMatrix

- **output**:

  dynamic manipulability measure as scalar value, defined as minimum Eigenvalaue of the dynamic manipulability matrix N

  dynamic manipulability matrix

- **notes**: acceleration dependent manipulability definded by Chiacchio, see [4], 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!

5.14 Module: signal

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

---

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

---

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

5.15 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 exu.SolveStatic(...)

def **SolveStatic** (*mbs*, *simulationSettings*= exudyn.SimulationSettings(), *updateInitialValues*= False, *storeSolver*= True)

- **function description**: solves the static mbs problem using simulationSettings; check theDoc.pdf for MainSolverStatic for further details of the static solver
- **input**:

  mbs: the MainSystem containing the assembled system; note that mbs may be changed upon several runs of this function

  simulationSettings: specific simulation settings used for computation of jacobian (e.g., sparse mode in static solver enables sparse computation)

  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']
- **output**: returns True, if successful, False if fails; if storeSolver = True, mbs.sys contains staticSolver, which allows to investigate solver problems (check theDoc.pdf section Section 7.3 and the items described in Section 7.3.6)
- **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()
 sims = exu.SimulationSettings()
 sims.timeIntegration.endTime = 10
 success = exu.SolveStatic(mbs, sims, storeSolver = True)
 print("success =", success)
 print("iterations = ", mbs.sys['staticSolver'].it)
 print("pos=", mbs.GetObjectOutputBody(body,localPosition=[0,0,0],
       variableType=exu.OutputVariableType.Position))
```

---

def **SolveDynamic** (*mbs*, *simulationSettings*= exudyn.SimulationSettings(), *solverType*= exudyn.DynamicSolverType.GeneralizedAlpha, *updateInitialValues*= False, *storeSolver*= 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
- **input**:

  mbs: the MainSystem containing the assembled system; note that mbs may be changed upon several runs of this function *simulationSettings*: specific simulation settings

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']

experimentalNewSolver: this allows to use the new solver; flag only used during development - will be removed in future!

- **output**: returns True, if successful, False if fails; if storeSolver = True, mbs.sys contains staticSolver, which allows to investigate solver problems (check theDoc.pdf section Section 7.3 and the items described in Section 7.3.6)
- **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()
sims = exu.SimulationSettings()
sims.timeIntegration.endTime = 10
success = exu.SolveDynamic(mbs, sims, storeSolver = True)
print("success =", success)
print("iterations = ", mbs.sys['dynamicSolver'].it)
print("pos=", mbs.GetObjectOutputBody(body,localPosition=[0,0,0],
      variableType=exu.OutputVariableType.Position))
```

---

def **ComputeODE2Eigenvalues** (*mbs*, *simulationSettings*= exudyn.SimulationSettings(), *useSparseSolver*= False, *numberOfEigenvalues*= 0, *setInitialValues*= True, *convert2Frequencies*= False)

- **function description**: compute eigenvalues for unconstrained ODE2 part of mbs, not considering the effects of algebraic constraints; 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 4.4.1).
- **input**:

mbs: the MainSystem containing the assembled system *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! 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 problems!)

convert2Frequencies: if True, the eigen values are converted into frequencies (Hz) and the output is [eigenFrequencies, eigenVectors]

– **output**: [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

– **example**:

```
#take any example from the Examples or TestModels folder, e.g., '
   cartesianSpringDamper.py' and run it
#then execute the following commands in the console (or add it to the file):
[values, vectors] = exu.ComputeODE2Eigenvalues(mbs)
print("eigenvalues=", values)
#==>values contains the eigenvalues of the ODE2 part of the system in the
   current configuration
```

---

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.InfoStat() from previous step, numberOfEvaluations is a counter which is proportional to number of RHS evaluations in method

5.16 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 **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

---

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!

---

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)

---

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)

---

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

---

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)

---

def **IndexFromValue** (*data*, *value*, *tolerance*= 1e-7)

– **function description**: get index from value in given data vector (numpy array); usually used to get specific index of time vector

– **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)

– **output**: index

---

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 **ComputeSkewMatrix** (*v*)

- **function description**: compute (3 x 3*n) skew matrix from (3*n) vector; used for ObjectFFRF and CMS implementation
- **input**: a vector v in np.array format, containing 3*n components
- **output**: (3 x 3*n) skew matrix in np.array format

---

def **CheckInputVector** (*vector*, *length*= -1)

- **function description**: check if input is list or array with according length; if length==-1, the length is not checked; raises Exception if the check fails
- **input**:

  vector: a vector in np.array or list format

  length: desired length of vector; if length=-1, it is ignored
- **output**: None

---

def **CheckInputIndexArray** (*indexArray*, *length*= -1)

- **function description**: check if input is list or array with according length and positive indices; if length==-1, the length is not checked; raises Exception if the check fails
- **input**:

  indexArray: a vector in np.array or list format

  length: desired length of vector; if length=-1, it is ignored
- **output**: None

---

def **LoadSolutionFile** (*fileName*)

- **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
- **output**: dictionary with 'data': the matrix of stored solution vectors, 'columnsExported': a list with binary values showing the exported columns [nODE2, nVel2, nAcc2, nODE1, nVel1, nAlgebraic, nData],'nColumns': the number of data columns and 'nRows': the number of data rows

---

def **SetSolutionState** (*exu*, *mbs*, *solution*, *row*, *configuration*)

- **function description**: load selected row of solution dictionary (previously loaded with LoadSolutionFile) into specific state

---

def **SetVisualizationState** (*exu*, *mbs*, *solution*, *row*)

- **function description**: load selected row of solution dictionary into visualization state and redraw
- **input**:
  exu: the exudyn library
  mbs: the system, where the state is applied to
  solution: solution dictionary previously loaded with LoadSolutionFile
  row: the according row of the solution file which is visualized
- **output**: renders the scene in mbs and changes the visualization state in mbs

---

def **AnimateSolution** (*exu*, *SC*, *mbs*, *solution*, *rowIncrement*= 1, *timeout*= 0.04, *createImages*= False, *runLoop*= False)

- **function description**: consecutively load the rows of a solution file and visualize the result
- **input**:
  exu: the exudyn library
  SC: the system container, where the mbs lives in
  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

---

def **DrawSystemGraph** (*mbs*, *showLoads*= True, *showSensors*= True, *useItemNames*= False, *useItemTypes*= False)

– **function description**: helper function which draws system graph of a MainSystem (mbs); several options let adjust the appearance of the graph

– **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 (ObjectMassPoint, ...) are shown instead of basic object types (Node, Load, ...)

– **output**: nothing

---

def **GenerateStraightLineANCFCable2D** (*mbs*, *positionOfNode0*, *positionOfNode1*, *numberOfElements*, *cableTemplate*, *massProportionalLoad*= [0,0,0], *fixedConstraintsNode0*= [0,0,0,0], *fixedConstraintsNode1*= [0,0,0,0], *vALE*= 0, *ConstrainAleCoordinate*= True)

– **function description**: generate cable elements along straight line with certain discretization

– **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) *fixedConstraintsNode1*: a list of 4 binary values, indicating the coordinate contraints on the last node (x,y-position and x,y-slope)

vALE: used for ObjectALEANCFCable2D objects

- **output**: returns a list [cableNodeList, cableObjectList, loadList, cableNodePositionList, cableCoordinateConstraintList]
- **example**:

```
see Examples/ANCF_cantilever_test.py
```

---

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.

---

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.

Chapter 6

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**).

6.1 Notation

6.1.1 Common types in item descriptions

There are certain types, which are heavily used in the description of items:

- `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
- `Index` ... an integer number ('`int`' in Python)
- `NodeIndex, MarkerIndex, ...` ... a special number object to represent integer indices of nodes, markers, ...
- `String` ... a string
- `ArrayIndex` ... a list of integer numbers (either list or in some cases `numpy` arrays may be allowed)
- `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 8.3
- `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
- `NumpyMatrix` ... a 2D `numpy` array (matrix) with real numbers (size given by according object)

6.1.2 States and coordinate attributes

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

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

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 differential equations coordinates
- ODE1 … first order differential equations coordinates; CURRENTLY NOT AVAILABLE
- 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 '$\tilde{\ }$' symbol,

$$\mathbf{a} \times \mathbf{b} = \tilde{\mathbf{a}}\, \mathbf{b} = -\tilde{\mathbf{b}}\, \mathbf{a} \tag{6.1}$$

6.1.3 Symbols and abbreviations in equations

The following table contains the common notation:

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

python name: OutputVariable	**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	$\boldsymbol{\theta} = [\theta_0, \ldots, \theta_n]^\mathrm{T}$	vector of **rotation parameters** (e.g., Euler parameters, Tait Bryan angles, ...) with n coordinates θ_i in any configuration
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): ${}^0\mathbf{x} = {}^{0b}\mathbf{A}\ {}^b\mathbf{x}$
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

python name: input to nodes, markers, etc.	**symbol**	**description**
referenceCoordinates	$\mathbf{c}_\mathrm{ref} = [c_0, \ldots, c_n]_\mathrm{ref}^\mathrm{T} = [c_{\mathrm{Ref},0}, \ldots, c_{\mathrm{Ref},n}]_\mathrm{ref}^\mathrm{T}$	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)
localPosition	${}^b\mathbf{p} = [{}^bp_0, {}^bp_1, {}^bp_2]^\mathrm{T}$	local (body-fixed) position vector with 3 position coordinates p_i in any configuration; used for local position of markers, sensors, etc.

6.1.4 Reference and current coordinates

An important fact on the coordinates is upon the splitting of quantities (e.g. position, rotation parameters, etc.) into reference and current (initial/visualization/...) coordinates. For the current position of a point node we have, e.g.,

$$\mathbf{p}_\mathrm{cur} = \mathbf{p}_\mathrm{ref} + \mathbf{u}_\mathrm{cur} \tag{6.2}$$

The same holds, e.g., for rotation parameters,

$$\boldsymbol{\theta}_{\mathrm{cur}} = \boldsymbol{\theta}_{\mathrm{ref}} + \boldsymbol{\psi}_{\mathrm{cur}} \tag{6.3}$$

6.1.5 Coordinate Systems

The left indices provide information about the coordinate system, e.g.,

$$^{0}\mathbf{u} \tag{6.4}$$

is the displacement vector in the global (inertial) coordinate systme 0, while

$$^{m1}\mathbf{u} \tag{6.5}$$

represents the displacement vector in marker 1 ($m1$) coordinates. Typical coordinate systems:

- $^{0}\mathbf{u}$ ... global 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 $m1$

To transform the local coordinates $^{m0}\mathbf{u}$ of marker 0 into global coordinates $^{0}\mathbf{x}$, we use

$$^{0}\mathbf{u} = {}^{0,m0}\mathbf{A}\; {}^{m0}\mathbf{u} \tag{6.6}$$

in which $^{0,m0}\mathbf{A}$ is the transformation matrix of (the body or node of) the underlying marker 0.

6.2 Nodes

6.2.1 NodePoint

A 3D point node for point masses or solid finite elements which has 3 displacement degrees of freedom for second order differential equations (ODE2).

Additional information for NodePoint:

- The Node has 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

---

6.2.1.1 DESCRIPTION of NodePoint:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
referenceCoordinates	$\mathbf{q}_{\mathrm{ref}} = [q_0,\, q_1,\, q_2]_{\mathrm{ref}}^{\mathrm{T}} = \mathbf{p}_{\mathrm{ref}} = [r_0,\, r_1,\, r_2]^{\mathrm{T}}$	
initialCoordinates	$\mathbf{q}_{\mathrm{ini}} = [q_0,\, q_1,\, q_2]_{\mathrm{ini}}^{\mathrm{T}} = \mathbf{u}_{\mathrm{ini}} = [u_0,\, u_1,\, u_2]_{\mathrm{ini}}^{\mathrm{T}}$	
initialVelocities	$\dot{\mathbf{q}}_{\mathrm{ini}} = \mathbf{v}_{\mathrm{ini}} = [\dot{q}_0,\, \dot{q}_1,\, \dot{q}_2]_{\mathrm{ini}}^{\mathrm{T}}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
Position	$\mathbf{p}_{\mathrm{config}} = [p_0,\, p_1,\, p_2]^T_{\mathrm{config}} = \mathbf{u}_{\mathrm{config}} + \mathbf{p}_{\mathrm{ref}}$	global 3D position vector of node; $\mathbf{u}_{\mathrm{ref}} = 0$
Displacement	$\mathbf{u}_{\mathrm{config}} = [q_0,\, q_1,\, q_2]^T_{\mathrm{config}}$	global 3D displacement vector of node
Velocity	$\mathbf{v}_{\mathrm{config}} = [\dot q_0,\, \dot q_1,\, \dot q_2]^T_{\mathrm{config}}$	global 3D velocity vector of node
Acceleration	$\mathbf{a}_{\mathrm{config}} = \ddot{\mathbf{q}}_{\mathrm{config}} = [\ddot q_0,\, \ddot q_1,\, \ddot q_2]^T_{\mathrm{config}}$	global 3D acceleration vector of node
Coordinates	$\mathbf{c}_{\mathrm{config}} = \mathbf{u}_{\mathrm{config}} = [q_0,\, q_1,\, q_2]^T_{\mathrm{config}}$	coordinate vector of node
Coordinates_t	$\dot{\mathbf{c}}_{\mathrm{config}} = \mathbf{v}_{\mathrm{config}} = [\dot q_0,\, \dot q_1,\, \dot q_2]^T_{\mathrm{config}}$	velocity coordinates vector of node
Coordinates_tt	$\ddot{\mathbf{c}}_{\mathrm{config}} = \mathbf{a}_{\mathrm{config}} = [\ddot q_0,\, \ddot q_1,\, \ddot q_2]^T_{\mathrm{config}}$	acceleration coordinates vector of node

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 6.3.1

For further examples on NodePoint see Examples:

- `interactiveTutorial.py`
- `Spring_with_constraints.py`
- `ALEANCF_pipe.py`
- `ANCF_cantilever_test_dyn.py`
- `ANCF_contact_circle.py`
- `ANCF_contact_circle2.py`
- `ANCF_moving_rigidbody.py`
- `ANCF_slidingJoint2D.py`
- `ANCF_switchingSlidingJoint2D.py`
- `ANCF_tests2.py`
- ...

For further examples on NodePoint see TestModels:

- `objectGenericODE2Test.py`
- `ANCFcontactCircleTest.py`
- `ANCFcontactFrictionTest.py`
- `computeODE2EigenvaluesTest.py`
- `explicitLieGroupIntegratorPythonTest.py`
- `explicitLieGroupIntegratorTest.py`
- `explicitLieGroupMBSTest.py`
- `fourBarMechanismTest.py`
- `genericODE2test.py`
- `geneticOptimizationTest.py`
- ...

6.2.2 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:

- The Node has 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

---

6.2.2.1 DESCRIPTION of NodePoint2D:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
referenceCoordinates	$\mathbf{q}_{\mathrm{ref}} = [q_0,\, q_1]_{\mathrm{ref}}^{\mathrm{T}} = \mathbf{p}_{\mathrm{ref}} = [r_0,\, r_1]^{\mathrm{T}}$	
initialCoordinates	$\mathbf{q}_{\mathrm{ini}} = [q_0,\, q_1]_{\mathrm{ini}}^{\mathrm{T}} = [u_0,\, u_1]_{\mathrm{ini}}^{\mathrm{T}}$	
initialVelocities	$\dot{\mathbf{q}}_{\mathrm{ini}} = \mathbf{v}_{\mathrm{ini}} = [\dot q_0,\, \dot q_1]_{\mathrm{ini}}^{\mathrm{T}}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
Position	$\mathbf{p}_{\mathrm{config}} = [p_0,\, p_1,\, 0]_{\mathrm{config}}^{\mathrm{T}} = \mathbf{u}_{\mathrm{config}} + \mathbf{p}_{\mathrm{ref}}$	global 3D position vector of node; $\mathbf{u}_{\mathrm{ref}} = 0$

Displacement	$\mathbf{u}_{\mathrm{config}} = [q_0,\, q_1,\, 0]^T_{\mathrm{config}}$	global 3D displacement vector of node
Velocity	$\mathbf{v}_{\mathrm{config}} = [\dot q_0,\, \dot q_1,\, 0]^T_{\mathrm{config}}$	global 3D velocity vector of node
Acceleration	$\mathbf{a}_{\mathrm{config}} = [\ddot q_0,\, \ddot q_1,\, 0]^T_{\mathrm{config}}$	global 3D acceleration vector of node
Coordinates	$\mathbf{c}_{\mathrm{config}} = [q_0,\, q_1]^T_{\mathrm{config}}$	coordinate vector of node
Coordinates_t	$\dot{\mathbf{c}}_{\mathrm{config}} = [\dot q_0,\, \dot q_1]^T_{\mathrm{config}}$	velocity coordinates vector of node
Coordinates_tt	$\ddot{\mathbf{c}}_{\mathrm{config}} = \mathbf{a}_{\mathrm{config}} = [\ddot q_0,\, \ddot q_1]^T_{\mathrm{config}}$	acceleration coordinates vector of node

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 6.3.2

For further examples on NodePoint2D see Examples:

- `myFirstExample.py`
- `pendulum2Dconstraint.py`
- `sliderCrank3DwithANCFbeltDrive2.py`
- `SpringDamperMassUserFunction.py`
- `ALEANCF_pipe.py`
- `ANCF_cantilever_test_dyn.py`
- `ANCF_contact_circle.py`
- `ANCF_contact_circle2.py`
- `ANCF_moving_rigidbody.py`
- `ANCF_slidingJoint2D.py`
- ...

For further examples on NodePoint2D see TestModels:

- `sparseMatrixSpringDamperTest.py`
- `ANCFcontactCircleTest.py`
- `ANCFcontactFrictionTest.py`
- `computeODE2EigenvaluesTest.py`
- `manualExplicitIntegrator.py`
- `modelUnitTests.py`
- `sliderCrankFloatingTest.py`

6.2.3 NodeRigidBodyEP

A 3D rigid body node based on Euler parameters for rigid bodies or beams; the node has 3 displacement coordinates (displacements of center of mass - COM: ux,uy,uz) and four rotation coordinates (Euler parameters = quaternions).

Additional information for NodeRigidBodyEP:

- The Node has 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
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

---

6.2.3.1 DESCRIPTION of NodeRigidBodyEP:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
referenceCoordinates	$\mathbf{q}_{\mathrm{ref}} = [q_0,\, q_1,\, q_2,\, \psi_0,\, \psi_1,\, \psi_2,\, \psi_3]^{\mathrm{T}}_{\mathrm{ref}} = [\mathbf{p}^{\mathrm{T}}_{\mathrm{ref}},\, \boldsymbol{\psi}^{\mathrm{T}}_{\mathrm{ref}}]^{\mathrm{T}}$	

initialCoordinates	$\mathbf{q}_{\mathrm{ini}} = [q_0,\, q_1,\, q_2,\, \psi_0,\, \psi_1,\, \psi_2,\, \psi_3]_{\mathrm{ini}}^{\mathrm{T}} = [\mathbf{u}_{\mathrm{ini}}^{\mathrm{T}},\, \boldsymbol{\psi}_{\mathrm{ini}}^{\mathrm{T}}]^{\mathrm{T}}$	
initialVelocities	$\dot{\mathbf{q}}_{\mathrm{ini}} = [\dot q_0,\, \dot q_1,\, \dot q_2,\, \dot\psi_0,\, \dot\psi_1,\, \dot\psi_2,\, \dot\psi_3]_{\mathrm{ini}}^{\mathrm{T}} = [\dot{\mathbf{u}}_{\mathrm{ini}}^{\mathrm{T}},\, \dot{\boldsymbol{\psi}}_{\mathrm{ini}}^{\mathrm{T}}]^{\mathrm{T}}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
Position	${}^{0}\mathbf{p}_{\mathrm{config}} = {}^{0}[p_0,\, p_1,\, p_2]_{\mathrm{config}}^{\mathrm{T}} = {}^{0}\mathbf{u}_{\mathrm{config}} + {}^{0}\mathbf{p}_{\mathrm{ref}}$	global 3D position vector of node; $\mathbf{u}_{\mathrm{ref}} = 0$
Displacement	${}^{0}\mathbf{u}_{\mathrm{config}} = [q_0,\, q_1,\, q_2]_{\mathrm{config}}^{\mathrm{T}}$	global 3D displacement vector of node
Velocity	${}^{0}\mathbf{v}_{\mathrm{config}} = [\dot q_0,\, \dot q_1,\, \dot q_2]_{\mathrm{config}}^{\mathrm{T}}$	global 3D velocity vector of node
Acceleration	${}^{0}\mathbf{a}_{\mathrm{config}} = [\ddot q_0,\, \ddot q_1,\, \ddot q_2]_{\mathrm{config}}^{\mathrm{T}}$	global 3D acceleration vector of node
Coordinates	$\mathbf{c}_{\mathrm{config}} = [q_0,\, q_1,\, q_2,\, \psi_0,\, \psi_1,\, \psi_2,\, \psi_3]_{\mathrm{config}}^{\mathrm{T}}$	coordinate vector of node, having 3 displacement coordinates and 4 Euler parameters
Coordinates_t	$\dot{\mathbf{c}}_{\mathrm{config}} = [\dot q_0,\, \dot q_1,\, \dot q_2,\, \dot\psi_0,\, \dot\psi_1,\, \dot\psi_2,\, \dot\psi_3]_{\mathrm{config}}^{\mathrm{T}}$	velocity coordinates vector of node
Coordinates_tt	$\ddot{\mathbf{c}}_{\mathrm{config}} = [\ddot q_0,\, \ddot q_1,\, \ddot q_2,\, \ddot\psi_0,\, \ddot\psi_1,\, \ddot\psi_2,\, \ddot\psi_3]_{\mathrm{config}}^{\mathrm{T}}$	acceleration coordinates vector of node
RotationMatrix	$[A_{00},\, A_{01},\, A_{02},\, A_{10},\, \ldots,\, A_{21},\, A_{22}]_{\mathrm{config}}^{\mathrm{T}}$	vector with 9 components of the rotation matrix ${}^{0b}\mathbf{A}_{\mathrm{config}}$ in row-major format, in any configuration; the rotation matrix transforms local (b) to global (0) coordinates
Rotation	$[\varphi_0,\, \varphi_1,\, \varphi_2]_{\mathrm{config}}^{\mathrm{T}}$	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
AngularVelocity	${}^{0}\boldsymbol{\omega}_{\mathrm{config}} = {}^{0}[\omega_0,\, \omega_1,\, \omega_2]_{\mathrm{config}}^{\mathrm{T}}$	global 3D angular velocity vector of node
AngularVelocityLocal	${}^{b}\boldsymbol{\omega}_{\mathrm{config}} = {}^{b}[\omega_0,\, \omega_1,\, \omega_2]_{\mathrm{config}}^{\mathrm{T}}$	local (body-fixed) 3D angular velocity vector of node
AngularAcceleration	${}^{0}\boldsymbol{\alpha}_{\mathrm{config}} = {}^{0}[\alpha_0,\, \alpha_1,\, \alpha_2]_{\mathrm{config}}^{\mathrm{T}}$	global 3D angular acceleration vector of node

Detailed information: All coordinates $\mathbf{c}_{\mathrm{config}}$ lead to second order differential equations, but there is one additional constraint equation for the quaternions. The additional constraint equation, which needs to be provided by the object, reads

$$1 - \sum_{i=0}^{3} \theta_i^2 = 0. \tag{6.7}$$

The rotation matrix ${}^{0b}\mathbf{A}_{\mathrm{config}}$ transforms local (body-fixed) 3D positions ${}^{b}\mathbf{p} = {}^{b}[p_0,\, p_1,\, p_2]^{\mathrm{T}}$ to global 3D positions,

$$ {}^{0}\mathbf{p}_{\mathrm{config}} = {}^{0b}\mathbf{A}_{\mathrm{config}}\, {}^{b}\mathbf{p} \tag{6.8}$$

Note that the Euler parameters $\boldsymbol{\theta}_{\mathrm{cur}}$ are computed as sum of current coordinates plus reference coordinates,

$$\boldsymbol{\theta}_{\mathrm{cur}} = \boldsymbol{\psi}_{\mathrm{cur}} + \boldsymbol{\psi}_{\mathrm{ref}}. \tag{6.9}$$

The rotation matrix is defined as function of the rotation parameters $\boldsymbol{\theta} = [\theta_0,\, \theta_1,\, \theta_2,\, \theta_3]^\mathrm{T}$

$$
{}^{0b}\mathbf{A} = \left[\begin{array}{ccc} -2\theta_3^2 - 2\theta_2^2 + 1 & -2\theta_3\theta_0 + 2\theta_2\theta_1 & 2*\theta_3\theta_1 + 2*\theta_2\theta_0 \\ 2\theta_3\theta_0 + 2\theta_2\theta_1 & -2\theta_3^2 - 2\theta_1^2 + 1 & 2\theta_3\theta_2 - 2\theta_1\theta_0 \\ -2\theta_2\theta_0 + 2\theta_3\theta_1 & 2\theta_3\theta_2 + 2\theta_1\theta_0 & -2\theta_2^2 - 2\theta_1^2 + 1 \end{array}\right] \qquad (6.10)
$$

The derivatives of the angular velocity vectors w.r.t. the rotation velocity coordinates $\dot{\boldsymbol{\theta}} = [\dot\theta_0,\, \dot\theta_1,\, \dot\theta_2,\, \dot\theta_3]^\mathrm{T}$ lead to the **G** matrices, as used in the equations of motion for rigid bodies,

$$
{}^{0}\boldsymbol{\omega} = {}^{0}\mathbf{G}\,\dot{\boldsymbol{\theta}}, \qquad (6.11)
$$

$$
{}^{b}\boldsymbol{\omega} = {}^{b}\mathbf{G}\,\dot{\boldsymbol{\theta}}. \qquad (6.12)
$$

For creating a `NodeRigidBodyEP`, there is a `rigidBodyUtilities` function `AddRigidBody`, see Section 5.11, which simplifies the setup of a rigid body significantely!

For further examples on NodeRigidBodyEP see Examples:

- `rigid3Dexample.py`
- `rigidBodyIMUtest.py`
- `rigidRotor3DbasicBehaviour.py`
- `rigidRotor3DFWBW.py`
- `rigidRotor3Dnutation.py`
- `rigidRotor3Drunup.py`
- `mouseInteractionExample.py`
- `rigidBodyTutorial.py`
- `sliderCrank3DwithANCFbeltDrive2.py`
- `stiffFlyballGovernor2.py`
- ...

For further examples on NodeRigidBodyEP see TestModels:

- `explicitLieGroupIntegratorPythonTest.py`
- `explicitLieGroupIntegratorTest.py`
- `explicitLieGroupMBSTest.py`
- `genericJointUserFunctionTest.py`
- `heavyTop.py`
- `objectFFRFTest.py`
- `sphericalJointTest.py`
- `carRollingDiscTest.py`
- `driveTrainTest.py`
- `mecanumWheelRollingDiscTest.py`
- ...

6.2.4 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:

- The Node has 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

---

6.2.4.1 DESCRIPTION of NodeRigidBodyRxyz:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
referenceCoordinates	$\mathbf{q}_{\mathrm{ref}} = [q_0, q_1, q_2, \psi_0, \psi_1, \psi_2]_{\mathrm{ref}}^{\mathrm{T}} = [\mathbf{p}_{\mathrm{ref}}^{\mathrm{T}}, \boldsymbol{\psi}_{\mathrm{ref}}^{\mathrm{T}}]^{\mathrm{T}}$	

initialCoordinates	$\mathbf{q}_{\mathrm{ini}} = [q_0,\, q_1,\, q_2,\, \psi_0,\, \psi_1,\, \psi_2]^{\mathrm{T}}_{\mathrm{ini}} = [\mathbf{u}^{\mathrm{T}}_{\mathrm{ini}},\, \boldsymbol{\psi}^{\mathrm{T}}_{\mathrm{ini}}]^{\mathrm{T}}$	
initialVelocities	$\dot{\mathbf{q}}_{\mathrm{ini}} = [\dot q_0,\, \dot q_1,\, \dot q_2,\, \dot\psi_0,\, \dot\psi_1,\, \dot\psi_2]^{\mathrm{T}}_{\mathrm{ini}} = [\dot{\mathbf{u}}^{\mathrm{T}}_{\mathrm{ini}},\, \dot{\boldsymbol{\psi}}^{\mathrm{T}}_{\mathrm{ini}}]^{\mathrm{T}}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
Position	${}^0\mathbf{p}_{\mathrm{config}} = {}^0[p_0,\, p_1,\, p_2]^{\mathrm{T}}_{\mathrm{config}} = {}^0\mathbf{u}_{\mathrm{config}} + {}^0\mathbf{p}_{\mathrm{ref}}$	global 3D position vector of node; $\mathbf{u}_{\mathrm{ref}} = 0$
Displacement	${}^0\mathbf{u}_{\mathrm{config}} = [q_0,\, q_1,\, q_2]^{\mathrm{T}}_{\mathrm{config}}$	global 3D displacement vector of node
Velocity	${}^0\mathbf{v}_{\mathrm{config}} = [\dot q_0,\, \dot q_1,\, \dot q_2]^{\mathrm{T}}_{\mathrm{config}}$	global 3D velocity vector of node
Acceleration	${}^0\mathbf{a}_{\mathrm{config}} = [\ddot q_0,\, \ddot q_1,\, \ddot q_2]^{\mathrm{T}}_{\mathrm{config}}$	global 3D acceleration vector of node
Coordinates	$\mathbf{c}_{\mathrm{config}} = [q_0,\, q_1,\, q_2,\, \psi_0,\, \psi_1,\, \psi_2]^{\mathrm{T}}_{\mathrm{config}}$	coordinate vector of node, having 3 displacement coordinates and 3 Euler angles
Coordinates_t	$\dot{\mathbf{c}}_{\mathrm{config}} = [\dot q_0,\, \dot q_1,\, \dot q_2,\, \dot\psi_0,\, \dot\psi_1,\, \dot\psi_2]^{\mathrm{T}}_{\mathrm{config}}$	velocity coordinates vector of node
Coordinates_tt	$\ddot{\mathbf{c}}_{\mathrm{config}} = [\ddot q_0,\, \ddot q_1,\, \ddot q_2,\, \ddot\psi_0,\, \ddot\psi_1,\, \ddot\psi_2]^{\mathrm{T}}_{\mathrm{config}}$	acceleration coordinates vector of node
RotationMatrix	$[A_{00},\, A_{01},\, A_{02},\, A_{10},\, \ldots,\, A_{21},\, A_{22}]^{\mathrm{T}}_{\mathrm{config}}$	vector with 9 components of the rotation matrix ${}^{0b}\mathbf{A}_{\mathrm{config}}$ in row-major format, in any configuration; the rotation matrix transforms local (b) to global (0) coordinates
Rotation	$[\varphi_0,\, \varphi_1,\, \varphi_2]^{\mathrm{T}}_{\mathrm{config}} = [\psi_0,\, \psi_1,\, \psi_2]^{\mathrm{T}}_{\mathrm{ref}} + [\psi_0,\, \psi_1,\, \psi_2]^{\mathrm{T}}_{\mathrm{config}}$	vector with 3 components of the Euler / Tait-Bryan 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
AngularVelocity	${}^0\boldsymbol{\omega}_{\mathrm{config}} = {}^0[\omega_0,\, \omega_1,\, \omega_2]^{\mathrm{T}}_{\mathrm{config}}$	global 3D angular velocity vector of node
AngularVelocityLocal	${}^b\boldsymbol{\omega}_{\mathrm{config}} = {}^b[\omega_0,\, \omega_1,\, \omega_2]^{\mathrm{T}}_{\mathrm{config}}$	local (body-fixed) 3D angular velocity vector of node
AngularAcceleration	${}^0\boldsymbol{\alpha}_{\mathrm{config}} = {}^0[\alpha_0,\, \alpha_1,\, \alpha_2]^{\mathrm{T}}_{\mathrm{config}}$	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 $\mathbf{c}_{\mathrm{config}}$ lead to second order differential equations. The rotation matrix ${}^{0b}\mathbf{A}_{\mathrm{config}}$ transforms local (body-fixed) 3D positions ${}^b\mathbf{p} = {}^b[p_0,\, p_1,\, p_2]^{\mathrm{T}}$ to global 3D positions,

$$ {}^0\mathbf{p}_{\mathrm{config}} = {}^{0b}\mathbf{A}_{\mathrm{config}}\, {}^b\mathbf{p} \tag{6.13}$$

Note that the Euler angles $\boldsymbol{\theta}_{\mathrm{cur}}$ are computed as sum of current coordinates plus reference coordinates,

$$\boldsymbol{\theta}_{\mathrm{cur}} = \boldsymbol{\psi}_{\mathrm{cur}} + \boldsymbol{\psi}_{\mathrm{ref}}. \tag{6.14}$$

The rotation matrix is defined as function of the rotation parameters $\boldsymbol{\theta} = [\theta_0,\, \theta_1,\, \theta_2]^{\mathrm{T}}$

$$ {}^{0b}\mathbf{A} = \mathbf{A}_0(\theta_0)\mathbf{A}_1(\theta_1)\mathbf{A}_2(\theta_2) \tag{6.15}$$

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,

$$ {}^0\boldsymbol{\omega} = {}^0\mathbf{G}\,\dot{\boldsymbol{\theta}}, \tag{6.16}$$

$$ {}^b\boldsymbol{\omega} = {}^b\mathbf{G}\,\dot{\boldsymbol{\theta}}. \tag{6.17}$$

For creating a `NodeRigidBodyRxyz`, there is a `rigidBodyUtilities` function `AddRigidBody`, see Section 5.11, which simplifies the setup of a rigid body significantely!

For further examples on NodeRigidBodyRxyz see Examples:

- `performanceMultiThreadingNG.py`

For further examples on NodeRigidBodyRxyz see TestModels:

- `explicitLieGroupIntegratorPythonTest.py`
- `explicitLieGroupIntegratorTest.py`
- `explicitLieGroupMBSTest.py`
- `heavyTop.py`
- `connectorRigidBodySpringDamperTest.py`

6.2.5 NodeRigidBodyRotVecLG

A 3D rigid body node based on rotation vector and Lie group methods for rigid bodies or beams; the node has 3 displacement coordinates and three rotation coordinates.

Additional information for NodeRigidBodyRotVecLG:

- The Node has the following types = `Position`, `Orientation`, `RigidBody`, `RotationRotationVector`, `RotationLieGroup`
- **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

---

6.2.5.1 DESCRIPTION of NodeRigidBodyRotVecLG:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
referenceCoordinates	$\mathbf{q}_{\mathrm{ref}} = [q_0, q_1, q_2, \nu_0, \nu_1, \nu_2]_{\mathrm{ref}}^T = [\mathbf{p}_{\mathrm{ref}}^T, \boldsymbol{\nu}_{\mathrm{ref}}^T]^T$	
initialCoordinates	$\mathbf{q}_{\mathrm{ini}} = [q_0, q_1, q_2, \nu_0, \nu_1, \nu_2]_{\mathrm{ini}}^T = [\mathbf{u}_{\mathrm{ini}}^T, \boldsymbol{\nu}_{\mathrm{ini}}^T]^T$	

initialVelocities	$\dot{\mathbf{q}}_{\rm ini} = [\dot{q}_0,\, \dot{q}_1,\, \dot{q}_2,\, \dot{\nu}_0,\, \dot{\nu}_1,\, \dot{\nu}_2]_{\rm ini}^{\rm T} = [\dot{\mathbf{u}}_{\rm ini}^{\rm T},\, \dot{\boldsymbol{\nu}}_{\rm ini}^{\rm T}]^{\rm T}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
Position	${^0}\mathbf{p}_{\rm config} = {^0}[p_0,\, p_1,\, p_2]^{\rm T}_{\rm config} = {^0}\mathbf{u}_{\rm config} + {^0}\mathbf{p}_{\rm ref}$	global 3D position vector of node; $\mathbf{u}_{\rm ref} = 0$
Displacement	${^0}\mathbf{u}_{\rm config} = [q_0,\, q_1,\, q_2]^{\rm T}_{\rm config}$	global 3D displacement vector of node
Velocity	${^0}\mathbf{v}_{\rm config} = [\dot{q}_0,\, \dot{q}_1,\, \dot{q}_2]^{\rm T}_{\rm config}$	global 3D velocity vector of node
Coordinates	$\mathbf{c}_{\rm config} = [q_0,\, q_1,\, q_2,\, \nu_0,\, \nu_1,\, \nu_2]^{\rm T}_{\rm config}$	coordinate vector of node, having 3 displacement coordinates and 3 Euler angles
Coordinates_t	$\dot{\mathbf{c}}_{\rm config} = [\dot{q}_0,\, \dot{q}_1,\, \dot{q}_2,\, \dot{\nu}_0,\, \dot{\nu}_1,\, \dot{\nu}_2]^{\rm T}_{\rm config}$	velocity coordinates vector of node
RotationMatrix	$[A_{00},\, A_{01},\, A_{02},\, A_{10},\, \ldots,\, A_{21},\, A_{22}]^{\rm T}_{\rm config}$	vector with 9 components of the rotation matrix ${^{0b}}\mathbf{A}_{\rm config}$ in row-major format, in any configuration; the rotation matrix transforms local (b) to global (0) coordinates
Rotation	$[\varphi_0,\, \varphi_1,\, \varphi_2]^{\rm T}_{\rm config}$	vector with 3 components of the Euler / Tait-Bryan angles in xyz-sequence (${^{0b}}\mathbf{A}_{\rm config} =: \mathbf{A}_0(\varphi_0) \cdot \mathbf{A}_1(\varphi_1) \cdot \mathbf{A}_2(\varphi_2)$), recomputed from rotation matrix
AngularVelocity	${^0}\boldsymbol{\omega}_{\rm config} = {^0}[\omega_0,\, \omega_1,\, \omega_2]^{\rm T}_{\rm config}$	global 3D angular velocity vector of node
AngularVelocityLocal	${^b}\boldsymbol{\omega}_{\rm config} = {^b}[\omega_0,\, \omega_1,\, \omega_2]^{\rm T}_{\rm config}$	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 [11].

The node has 3 displacement coordinates $[q_0,\, q_1,\, q_2]^{\rm T}$ and three rotation coordinates, which is the rotation vector

$$\boldsymbol{\nu} = \varphi \mathbf{n} = \boldsymbol{\nu}_{\rm config} + \boldsymbol{\nu}_{\rm ref}, \tag{6.18}$$

with the rotation angle φ and the rotation axis $\mathbf{n}$. All coordinates $\mathbf{c}_{\rm config}$ 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 rotation matrix ${^{0b}}\mathbf{A}_{\rm config}$ transforms local (body-fixed) 3D positions ${^b}\mathbf{p} = {^b}[p_0,\, p_1,\, p_2]^{\rm T}$ to global 3D positions,

$${^0}\mathbf{p}_{\rm config} = {^{0b}}\mathbf{A}_{\rm config}\, {^b}\mathbf{p} \tag{6.19}$$

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.

For creating a `NodeRigidBodyRotVecLG`, there is a `rigidBodyUtilities` function `AddRigidBody`, see Section 5.11, which simplifies the setup of a rigid body significantely!

For further examples on NodeRigidBodyRotVecLG see TestModels:

- `explicitLieGroupIntegratorPythonTest.py`
- `explicitLieGroupIntegratorTest.py`
- `explicitLieGroupMBSTest.py`

6.2.6 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:

- The Node has 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

---

6.2.6.1 DESCRIPTION of NodeRigidBody2D:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
referenceCoordinates	$\mathbf{q}_{\mathrm{ref}} = [q_0,\, q_1,\, \psi_0]^{\mathrm{T}}_{\mathrm{ref}}$	
initialCoordinates	$\mathbf{q}_{\mathrm{ini}} = [q_0,\, q_1,\, \psi_0]^{\mathrm{T}}_{\mathrm{ini}}$	
initialVelocities	$\dot{\mathbf{q}}_{\mathrm{ini}} = [\dot q_0,\, \dot q_1,\, \dot\psi_0]^{\mathrm{T}}_{\mathrm{ini}} = [v_0,\, v_1,\, \omega_2]^{\mathrm{T}}_{\mathrm{ini}}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
Position	${}^{0}\mathbf{p}_{\rm config} = {}^{0}[p_0,\, p_1,\, 0]^{\rm T}_{\rm config} = {}^{0}\mathbf{u}_{\rm config} + {}^{0}\mathbf{p}_{\rm ref}$	global 3D position vector of node; $\mathbf{u}_{\rm ref} = 0$
Displacement	${}^{0}\mathbf{u}_{\rm config} = [q_0,\, q_1,\, 0]^{\rm T}_{\rm config}$	global 3D displacement vector of node
Velocity	${}^{0}\mathbf{v}_{\rm config} = [\dot q_0,\, \dot q_1,\, 0]^{\rm T}_{\rm config}$	global 3D velocity vector of node
Acceleration	${}^{0}\mathbf{a}_{\rm config} = [\ddot q_0,\, \ddot q_1,\, 0]^{\rm T}_{\rm config}$	global 3D acceleration vector of node
AngularVelocity	${}^{0}\boldsymbol{\omega}_{\rm config} = {}^{0}[0,\, 0,\, \dot\psi_0]^{\rm T}_{\rm config}$	global 3D angular velocity vector of node
Coordinates	$\mathbf{c}_{\rm config} = [q_0,\, q_1,\, \psi_0]^{\rm T}_{\rm config}$	coordinate vector of node, having 2 displacement coordinates and 1 angle
Coordinates_t	$\dot{\mathbf{c}}_{\rm config} = [\dot q_0,\, \dot q_1,\, \dot\psi_0]^{\rm T}_{\rm config}$	velocity coordinates vector of node
Coordinates_tt	$\ddot{\mathbf{c}}_{\rm config} = [\ddot q_0,\, \ddot q_1,\, \ddot\psi_0]^{\rm T}_{\rm config}$	acceleration coordinates vector of node
RotationMatrix	$[A_{00},\, A_{01},\, A_{02},\, A_{10},\, \ldots,\, A_{21},\, A_{22}]^{\rm T}_{\rm config}$	vector with 9 components of the rotation matrix ${}^{0b}\mathbf{A}_{\rm config}$ in row-major format, in any configuration; the rotation matrix transforms local (b) to global (0) coordinates
Rotation	$[\theta_0]^{\rm T}_{\rm config} = [\psi_0]^{\rm T}_{\rm ref} + [\psi_0]^{\rm T}_{\rm config}$	vector with 1 angle around out of plane axis
AngularVelocityLocal	${}^{b}\boldsymbol{\omega}_{\rm config} = {}^{b}[0,\, 0,\, \dot\psi_0]^{\rm T}_{\rm config}$	local (body-fixed) 3D angular velocity vector of node
AngularAcceleration	${}^{0}\boldsymbol{\alpha}_{\rm config} = {}^{0}[0,\, 0,\, \ddot\psi_0]^{\rm T}_{\rm config}$	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). Using the rotation parameter $\theta_{0\rm config} = \psi_{0ref} + \psi_{0\rm config}$, the rotation matrix is defined as

$$ {}^{0b}\mathbf{A}_{\rm config} = \left[\begin{array}{ccc} \cos(\theta_0) & -\sin(\theta_0) & 0 \\ \sin(\theta_0) & \cos(\theta_0) & 0 \\ 0 & 0 & 1 \end{array}\right]_{\rm config} \qquad (6.20)$$

Example for NodeRigidBody2D: see ObjectRigidBody2D

For further examples on NodeRigidBody2D see Examples:

- `sliderCrank3DwithANCFbeltDrive2.py`
- `ANCF_moving_rigidbody.py`
- `ANCF_slidingJoint2D.py`
- `ANCF_switchingSlidingJoint2D.py`
- `geneticOptimizationSliderCrank.py`
- `lavalRotor2Dtest.py`
- `rigid_pendulum.py`
- `SliderCrank.py`
- `sliderCrank3DwithANCFbeltDrive.py`
- `slidercrankWithMassSpring.py`
- ...

For further examples on NodeRigidBody2D see TestModels:

- `scissorPrismaticRevolute2D.py`

- ACNFslidingAndALEjointTest.py
- ANCFcontactFrictionTest.py
- ANCFmovingRigidBodyTest.py
- compareFullModifiedNewton.py
- modelUnitTests.py
- PARTS_ATEs_moving.py
- pendulumFriction.py
- sliderCrankFloatingTest.py

6.2.7 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:

- The Node has 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

6.2.7.1 DESCRIPTION of Node1D:

Information on input parameters:

input parameter	symbol	description see tables above
referenceCoordinates	$[q_0]^T_{\mathrm{ref}}$	
initialCoordinates	$[q_0]^T_{\mathrm{ini}}$	
initialVelocities	$[\dot{q}_0]^T_{\mathrm{ini}}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Coordinates	$\mathbf{q}_{\mathrm{config}} = [q_0]^T_{\mathrm{config}}$	ODE2 coordinate of node (in vector form)
Coordinates_t	$\dot{\mathbf{q}}_{\mathrm{config}} = [\dot{q}_0]^T_{\mathrm{config}}$	ODE2 velocity coordinate of node (in vector form)

Coordinates_tt	$\ddot{\mathbf{q}}_{\mathrm{config}} = [\ddot{q}_0]^T_{\mathrm{config}}$	ODE2 acceleration coordinate of node (in vector form)

Detailed information: The current position/rotation coordinate of the 1D node is computed from

$$p_0 = q_{0_{\mathrm{ref}}} + q_{0_{\mathrm{cur}}} \tag{6.21}$$

The coordinate leads to one second order differential equation. The graphical representation and the (internal) position of the node is

$$p_{\mathrm{config}} = \begin{bmatrix} p_{0_{\mathrm{config}}} \\ 0 \\ 0 \end{bmatrix} \tag{6.22}$$

The (internal) velocity vector is $[p_{0_{\mathrm{config}}}, 0, 0]^T$.

For further examples on Node1D see Examples:

- `multiprocessingTest.py`
- `sliderCrankCMSacme.py`

For further examples on Node1D see TestModels:

- `driveTrainTest.py`

6.2.8 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:

- The Node has 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

---

6.2.8.1 DESCRIPTION of NodePoint2DSlope1:

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
Position		global 3D position vector of node (=displacement+reference position)
Displacement		global 3D displacement vector of node

Velocity		global 3D velocity vector 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 further examples on NodePoint2DSlope1 see Examples:

- `sliderCrank3DwithANCFbeltDrive2.py`
- `ALEANCF_pipe.py`
- `ANCF_cantilever_test_dyn.py`
- `ANCF_contact_circle.py`
- `ANCF_contact_circle2.py`
- `ANCF_moving_rigidbody.py`
- `ANCF_slidingJoint2D.py`
- `ANCF_switchingSlidingJoint2D.py`
- `ANCF_tests2.py`
- `ANCF_test_halfcircle.py`
- ...

For further examples on NodePoint2DSlope1 see TestModels:

- `ANCFcontactCircleTest.py`
- `ANCFcontactFrictionTest.py`
- `computeODE2EigenvaluesTest.py`
- `manualExplicitIntegrator.py`
- `modelUnitTests.py`

6.2.9 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:

- The Node has 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	Index		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

---

6.2.9.1 DESCRIPTION of NodeGenericODE2:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
referenceCoordinates	$\mathbf{q}_{\mathrm{ref}} = [q_0, \ldots, q_{nc}]_{\mathrm{ref}}^{\mathrm{T}}$	
initialCoordinates	$\mathbf{q}_{\mathrm{ini}} = [q_0, \ldots, q_{nc}]_{\mathrm{ini}}^{\mathrm{T}}$	
initialCoordinates_t	$\dot{\mathbf{q}}_{\mathrm{ini}} = [\dot q_0, \ldots, \dot q_{n_c}]_{\mathrm{ini}}^{\mathrm{T}}$	
numberOfODE2Coordinates	n_c	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
Coordinates	$\mathbf{q}_{\mathrm{config}} = [q_0, \ldots, q_{nc}]_{\mathrm{config}}^{\mathrm{T}}$	coordinates vector of node
Coordinates_t	$\dot{\mathbf{q}}_{\mathrm{config}} = [\dot q_0, \ldots, \dot q_{nc}]_{\mathrm{config}}^{\mathrm{T}}$	velocity coordinates vector of node
Coordinates_tt	$\ddot{\mathbf{q}}_{\mathrm{config}} = [\ddot q_0, \ldots, \ddot q_{nc}]_{\mathrm{config}}^{\mathrm{T}}$	acceleration coordinates vector of node

For further examples on NodeGenericODE2 see Examples:

- `ALEANCF_pipe.py`
- `ANCF_moving_rigidbody.py`
- `simulateInteractively.py`

For further examples on NodeGenericODE2 see TestModels:

- `ACNFslidingAndALEjointTest.py`
- `ANCFmovingRigidBodyTest.py`
- `solverExplicitODE1ODE2test.py`

6.2.10 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	Index		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

---

6.2.10.1 DESCRIPTION of NodeGenericODE1:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
referenceCoordinates	$\mathbf{y}_{\mathrm{ref}} = [y_0, \ldots, y_{nc}]_{\mathrm{ref}}^{\mathrm{T}}$	
initialCoordinates	$\mathbf{y}_{\mathrm{ini}} = [y_0, \ldots, y_{nc}]_{\mathrm{ini}}^{\mathrm{T}}$	
numberOfODE1Coordinates	n_c	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
Coordinates	$\mathbf{y}_{\mathrm{config}} = [y_0, \ldots, y_{nc}]_{\mathrm{config}}^{\mathrm{T}}$	ODE1 coordinates vector of node
Coordinates_t	$\dot{\mathbf{y}}_{\mathrm{config}} = [\dot{y}_0, \ldots, \dot{y}_{nc}]_{\mathrm{config}}^{\mathrm{T}}$	ODE1 velocity coordinates vector of node

---

For further examples on NodeGenericODE1 see TestModels:

- `solverExplicitODE1ODE2test.py`

6.2.11 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:

- The Node has 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	Index		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

---

6.2.11.1 DESCRIPTION of NodeGenericData:

Information on input parameters:

input parameter	symbol	description see tables above
initialCoordinates	$\mathbf{x}_{\mathrm{ini}} = [x_0, \ldots, x_{n_c}]_{\mathrm{ini}}^{\mathrm{T}}$	
numberOfDataCoordinates	n_c	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Coordinates	$\mathbf{x}_{\mathrm{config}} = [x_0, \ldots, x_{nc}]_{\mathrm{config}}^{\mathrm{T}}$	data coordinates (history variables) vector of node

---

For further examples on NodeGenericData see Examples:

- `ANCF_contact_circle.py`
- `ANCF_contact_circle2.py`
- `ANCF_moving_rigidbody.py`
- `ANCF_slidingJoint2D.py`
- `ANCF_switchingSlidingJoint2D.py`
- `sliderCrank3DwithANCFbeltDrive.py`

- sliderCrank3DwithANCFbeltDrive2.py

For further examples on NodeGenericData see TestModels:

- ACNFslidingAndALEjointTest.py
- ANCFcontactCircleTest.py
- ANCFcontactFrictionTest.py
- ANCFmovingRigidBodyTest.py
- carRollingDiscTest.py
- mecanumWheelRollingDiscTest.py
- modelUnitTests.py
- rollingCoinPenaltyTest.py

6.2.12 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.

Additional information for NodePointGround:

- The Node has the following types = `Ground`, `Position2D`, `Position`, `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

---

6.2.12.1 DESCRIPTION of NodePointGround:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
referenceCoordinates	$\mathbf{q}_{\mathrm{ref}} = [q_0,\, q_1,\, q_2]_{\mathrm{ref}}^{\mathrm{T}} = \mathbf{p}_{\mathrm{ref}} = [r_0,\, r_1,\, r_2]^{\mathrm{T}}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
Position	$\mathbf{p}_{\mathrm{config}} = [p_0,\, p_1,\, p_2]_{\mathrm{config}}^{\mathrm{T}} = \mathbf{p}_{\mathrm{ref}}$	global 3D position vector of node (=reference position)
Displacement	$\mathbf{u}_{\mathrm{config}} = [0,\, 0,\, 0]_{\mathrm{config}}^{\mathrm{T}}$	zero 3D vector
Velocity	$\mathbf{v}_{\mathrm{config}} = [0,\, 0,\, 0]_{\mathrm{config}}^{\mathrm{T}}$	zero 3D vector

Coordinates	$\mathbf{c}_{\mathrm{config}} = []$	vector of length zero
Coordinates_t	$\dot{\mathbf{c}}_{\mathrm{config}} = []$	vector of length zero

---

For further examples on NodePointGround see Examples:

- `ALEANCF_pipe.py`
- `ANCF_cantilever_test.py`
- `ANCF_cantilever_test_dyn.py`
- `ANCF_contact_circle.py`
- `ANCF_contact_circle2.py`
- `ANCF_moving_rigidbody.py`
- `ANCF_slidingJoint2D.py`
- `ANCF_switchingSlidingJoint2D.py`
- `ANCF_tests2.py`
- `ANCF_test_halfcircle.py`
- ...

For further examples on NodePointGround see TestModels:

- `ACNFslidingAndALEjointTest.py`
- `ANCFcontactCircleTest.py`
- `ANCFcontactFrictionTest.py`
- `ANCFmovingRigidBodyTest.py`
- `computeODE2EigenvaluesTest.py`
- `connectorRigidBodySpringDamperTest.py`
- `geneticOptimizationTest.py`
- `manualExplicitIntegrator.py`
- `modelUnitTests.py`
- `objectFFRFTest.py`
- ...

6.3 Objects

6.3.1 ObjectMassPoint

A 3D mass point which is attached to a position-based node, usually NodePoint.

Additional information for ObjectMassPoint:

- The Object has 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		MAXINT	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

---

6.3.1.1 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	${}^0\mathbf{p}_{\mathrm{config}}({}^b\mathbf{p}) = {}^0\mathbf{u}_{\mathrm{config}} + {}^0\mathbf{p}_{\mathrm{ref}} + {}^{0b}\mathbf{A}\,{}^b\mathbf{p}$	global position vector of translated local position; local (body) coordinate system = global coordinate system
Displacement	${}^0\mathbf{u}_{\mathrm{config}} = [q_0,\; q_1,\; q_2]^{\mathrm{T}}_{\mathrm{config}}$	global displacement vector of mass point

Velocity	${}^{0}\mathbf{v}_{\,\mathrm{config}} = {}^{0}\dot{\mathbf{u}}_{\,\mathrm{config}} = [\dot{q}_0,\ \dot{q}_1,\ \dot{q}_2]^{\mathrm{T}}_{\mathrm{config}}$	global velocity vector of mass point
Acceleration	${}^{0}\mathbf{a}_{\,\mathrm{config}} = {}^{0}\ddot{\mathbf{u}}_{\,\mathrm{config}} = [\ddot{q}_0,\ \ddot{q}_1,\ \ddot{q}_2]^{\mathrm{T}}_{\mathrm{config}}$	global acceleration vector of mass point

6.3.1.2 Definition of quantities

intermediate variables	**symbol**	**description**
node position	${}^{0}\mathbf{p}_{\,\mathrm{config}} = {}^{0}\mathbf{p}\,(n_0)_{\mathrm{config}}$	position of mass point which is provided by node n_0 in any configuration
node displacement	${}^{0}\mathbf{u}_{\,\mathrm{config}} = [q_0,\ q_1,\ q_2]^{\mathrm{T}}_{\mathrm{config}} = {}^{0}\mathbf{u}\,(n_0)_{\mathrm{config}}$	displacement of mass point which is provided by node n_0 in any configuration
node velocity	${}^{0}\mathbf{v}_{\,\mathrm{config}} = [\dot{q}_0,\ \dot{q}_1,\ \dot{q}_2]^{\mathrm{T}}_{\mathrm{config}} = {}^{0}\mathbf{v}\,(n_0)_{\mathrm{config}}$	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	${}^{0}\mathbf{f} = [f_0,\ f_1,\ f_2]^{\mathrm{T}}$	residual of all forces on mass point
applied forces	${}^{0}\mathbf{f}_{\,a} = [f_0,\ f_1,\ f_2]^{\mathrm{T}}$	applied forces (loads, connectors, joint reaction forces, ...)

6.3.1.3 Equations of motion

$$\begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{bmatrix} \begin{bmatrix} \ddot{q}_0 \\ \ddot{q}_1 \\ \ddot{q}_2 \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ f_2 \end{bmatrix}. \qquad (6.23)$$

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 $\mathbf{p}_{\mathrm{config}}$. The **position jacobian**

$$\mathbf{J}_{pos} = \partial \mathbf{p}_{\mathrm{cur}} / \partial \mathbf{c}_{\mathrm{cur}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad (6.24)$$

transforms the action of global applied forces ${}^{0}\mathbf{f}_{\,a}$ of position-based markers on the coordinates $\mathbf{c}$

$$\mathbf{Q} = \mathbf{J}_{pos}\, {}^{0}\mathbf{f}_{\,a}. \qquad (6.25)$$

6.3.1.4 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()
SC.TimeIntegrationSolve(mbs, 'GeneralizedAlpha', exu.SimulationSettings())

#check result
exudynTestGlobals.testResult = mbs.GetNodeOutput(node, exu.OutputVariableType.
    Position)[0]
#final x-coordinate of position shall be 2
```

For further examples on ObjectMassPoint see Examples:

- `interactiveTutorial.py`
- `ANCF_slidingJoint2D.py`
- `cartesianSpringDamper.py`
- `coordinateSpringDamper.py`
- `geneticOptimizationExample.py`
- `geneticOptimizationSliderCrank.py`
- `massSpringFrictionInteractive.py`
- `parameterVariationExample.py`
- `pendulum2Dconstraint.py`
- `simulateInteractively.py`
- ...

For further examples on ObjectMassPoint see TestModels:

- `fourBarMechanismTest.py`
- `genericODE2test.py`
- `geneticOptimizationTest.py`
- `modelUnitTests.py`
- `sliderCrankFloatingTest.py`
- `sparseMatrixSpringDamperTest.py`
- `springDamperUserFunctionTest.py`

6.3.2 ObjectMassPoint2D

A 2D mass point which is attached to a position-based 2D node.

Additional information for ObjectMassPoint2D:

- The Object has 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		MAXINT	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

---

6.3.2.1 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	${}^0\mathbf{p}_{\mathrm{config}}({}^b\mathbf{p}) = {}^0\mathbf{u}_{\mathrm{config}} + {}^0\mathbf{p}_{\mathrm{ref}} + {}^{0b}\mathbf{A}\,{}^b\mathbf{p}$	global position vector of translated local position; local (body) coordinate system = global coordinate system
Displacement	${}^0\mathbf{u}_{\mathrm{config}} = [q_0,\ q_1,\ 0]^{\mathrm{T}}_{\mathrm{config}}$	global displacement vector of mass point
Velocity	${}^0\mathbf{v}_{\mathrm{config}} = {}^0\dot{\mathbf{u}}_{\mathrm{config}} = [\dot q_0,\ \dot q_1,\ 0]^{\mathrm{T}}_{\mathrm{config}}$	global velocity vector of mass point
Acceleration	${}^0\mathbf{a}_{\mathrm{config}} = {}^0\ddot{\mathbf{u}}_{\mathrm{config}} = [\ddot q_0,\ \ddot q_1,\ 0]^{\mathrm{T}}_{\mathrm{config}}$	global acceleration vector of mass point

6.3.2.2 Definition of quantities

intermediate variables	symbol	description
node position	${}^{0}\mathbf{p}_{\mathrm{config}} = {}^{0}\mathbf{p}(n_0)_{\mathrm{config}}$	position of mass point which is provided by node n_0 in any configuration
node displacement	${}^{0}\mathbf{u}_{\mathrm{config}} = [q_0,\ q_1,\ 0]^{\mathrm{T}}_{\mathrm{config}} = {}^{0}\mathbf{u}(n_0)_{\mathrm{config}}$	displacement of mass point which is provided by node n_0 in any configuration
node velocity	${}^{0}\mathbf{v}_{\mathrm{config}} = [\dot q_0,\ \dot q_1,\ 0]^{\mathrm{T}}_{\mathrm{config}} = {}^{0}\mathbf{v}(n_0)_{\mathrm{config}}$	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	${}^{0}\mathbf{f} = [f_0,\ f_1]^{\mathrm{T}}$	residual of all forces on mass point
applied forces	${}^{0}\mathbf{f}_a = [f_0,\ f_1,\ f_2]^{\mathrm{T}}$	applied forces (loads, connectors, joint reaction forces, ...)

6.3.2.3 Equations of motion

$$\begin{bmatrix} m & 0 \\ 0 & m \end{bmatrix} \begin{bmatrix} \ddot q_0 \\ \ddot q_1 \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \end{bmatrix}. \tag{6.26}$$

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 $\mathbf{p}_{\mathrm{config}}$. The **position jacobian**

$$\mathbf{J}_{pos} = \partial \mathbf{p}_{\mathrm{cur}} / \partial \mathbf{c}_{\mathrm{cur}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{6.27}$$

transforms the action of global applied forces ${}^{0}\mathbf{f}_a$ of position-based markers on the coordinates $\mathbf{c}$

$$\mathbf{Q} = \mathbf{J}_{pos}\, {}^{0}\mathbf{f}_a. \tag{6.28}$$

6.3.2.4 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()
SC.TimeIntegrationSolve(mbs, 'GeneralizedAlpha', exu.SimulationSettings())

#check result
exudynTestGlobals.testResult = mbs.GetNodeOutput(node, exu.OutputVariableType.
    Position)[0]
#final x-coordinate of position shall be 2
```

For further examples on ObjectMassPoint2D see Examples:

- `myFirstExample.py`
- `ANCF_slidingJoint2D.py`
- `geneticOptimizationSliderCrank.py`
- `pendulum2Dconstraint.py`
- `SliderCrank.py`
- `sliderCrank3DwithANCFbeltDrive2.py`
- `slidercrankWithMassSpring.py`
- `SpringDamperMassUserFunction.py`
- `switchingConstraintsPendulum.py`

For further examples on ObjectMassPoint2D see TestModels:

- `modelUnitTests.py`
- `sliderCrankFloatingTest.py`
- `sparseMatrixSpringDamperTest.py`

6.3.3 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:

- The Object has 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		MAXINT	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

---

6.3.3.1 DESCRIPTION of ObjectMass1D:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
physicsMass	m	
nodeNumber	$n0$	
referencePosition	${}^{0}\mathbf{p}_0$	
referenceRotation	${}^{0b}\mathbf{A}_0 \in \mathbb{R}^{3\times 3}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
Position	${}^{0}\mathbf{p}_{\mathrm{config}}$	global position vector; for interpretation see intermediate variables
Displacement	${}^{0}\mathbf{u}_{\mathrm{config}}$	global displacement vector; for interpretation see intermediate variables
Velocity	${}^{0}\mathbf{v}_{\mathrm{config}}$	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 angles in xyz-sequence (R=Rx*Ry*Rz), recomputed from rotation matrix ${}^{0b}\mathbf{A}$
AngularVelocity	${}^{0}\boldsymbol{\omega}_{\mathrm{config}}$	angular velocity of body
AngularVelocityLocal	${}^{b}\boldsymbol{\omega}_{\mathrm{config}}$	local (body-fixed) 3D velocity vector of node

6.3.3.2 Definition of quantities

intermediate variables	**symbol**	**description**
position coordinate	$p_{0\mathrm{config}} = c_{0\mathrm{config}} + c_{0\mathrm{ref}}$	position coordinate of node (nodal coordinate c_0) in any configuration
displacement coordinate	$u_{0\mathrm{config}} = c_{0\mathrm{config}}$	displacement coordinate of mass node in any configuration
velocity coordinate	$u_{0\mathrm{config}}$	velocity coordinate of mass node in any configuration
Position	${}^{0}\mathbf{p}_{\mathrm{config}} = {}^{0}\mathbf{p}_0 + {}^{0b}\mathbf{A}_0 \; {}^{b}\begin{bmatrix} p_0 \\ 0 \\ 0 \end{bmatrix}_{\mathrm{config}}$	(translational) position of mass object in any configuration
Displacement	${}^{0}\mathbf{u}_{\mathrm{config}} = {}^{0b}\mathbf{A}_0 \; {}^{b}\begin{bmatrix} q_0 \\ 0 \\ 0 \end{bmatrix}_{\mathrm{config}}$	(translational) displacement of mass object in any configuration
Velocity	${}^{0}\mathbf{v}_{\mathrm{config}} = {}^{0b}\mathbf{A}_0 \; {}^{b}\begin{bmatrix} \dot{q}_0 \\ 0 \\ 0 \end{bmatrix}_{\mathrm{config}}$	(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.

6.3.3.3 Equations of motion

$$m \cdot \ddot{q}_0 = f. \qquad (6.29)$$

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

$$f = {}^{b}[1,\,0,\,0]\;{}^{b0}\mathbf{A}_0\;{}^{0}\mathbf{f}_a \qquad (6.30)$$

Thus, the **position jacobian** reads

$$\mathbf{J}_{pos} = \partial \mathbf{p}_{\mathrm{cur}} / \partial {q_0}_{\mathrm{cur}} = {}^{b}[1, 0, 0]\ {}^{b0}\mathbf{A}_0 \qquad (6.31)$$

6.3.3.4 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()
SC.TimeIntegrationSolve(mbs, 'GeneralizedAlpha', exu.SimulationSettings())

#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 further examples on ObjectMass1D see Examples:

- multiprocessingTest.py
- sliderCrankCMSacme.py

For further examples on ObjectMass1D see TestModels:

- driveTrainTest.py

6.3.4 ObjectRotationalMass1D

A 1D rotational inertia (mass) which is attached to Node1D.

Additional information for ObjectRotationalMass1D:

- The Object has 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		MAXINT	node number (type NodeIndex) of Node1D, providing rotation coordinate $\psi_0 = c_0$
referencePosition	Vector3D	3	[0.,0.,0.]	a constant reference position, 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

---

6.3.4.1 DESCRIPTION of ObjectRotationalMass1D:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
physicsInertia	J	
nodeNumber	$n0$	
referencePosition	${}^0\mathbf{p}_0$	
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	${}^{0}\mathbf{p}_{\mathrm{config}}$	global position vector; for interpretation see intermediate variables
Displacement	${}^{0}\mathbf{u}_{\mathrm{config}}$	global displacement vector; for interpretation see intermediate variables
Velocity	${}^{0}\mathbf{v}_{\mathrm{config}}$	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 angles in xyz-sequence (R=Rx*Ry*Rz), recomputed from rotation matrix ${}^{0b}\mathbf{A}$
AngularVelocity	${}^{0}\boldsymbol{\omega}_{\mathrm{config}}$	angular velocity of body
AngularVelocityLocal	${}^{b}\boldsymbol{\omega}_{\mathrm{config}}$	local (body-fixed) 3D velocity vector of node

6.3.4.2 Definition of quantities

intermediate variables	**symbol**	**description**
position coordinate	$\theta_{0\mathrm{config}} = c_{0\mathrm{config}} + c_{0\mathrm{ref}}$	total rotation coordinate of node (e.g., Node1D) in any configuration (nodal coordinate c_0)
displacement coordinate	$\psi_{0\mathrm{config}} = c_{0\mathrm{config}}$	change of rotation coordinate of mass node (e.g., Node1D) in any configuration (nodal coordinate c_0)
velocity coordinate	$\dot\psi_{0\mathrm{config}}$	rotation velocity coordinate of mass node (e.g., Node1D) in any configuration
Position	${}^{0}\mathbf{p}_{\mathrm{config}} = {}^{0}\mathbf{p}_0$	constant (translational) position of mass object in any configuration
Displacement	${}^{0}\mathbf{u}_{\mathrm{config}} = [0,0,0]^{\mathrm{T}}$	(translational) displacement of mass object in any configuration
Velocity	${}^{0}\mathbf{v}_{\mathrm{config}} = [0,0,0]^{\mathrm{T}}$	(translational) velocity of mass object in any configuration
AngularVelocity	${}^{0}\boldsymbol{\omega}_{\mathrm{config}} = {}^{0i}\mathbf{A}_0 \; {}^{i}\begin{bmatrix} 0 \\ 0 \\ \dot\psi_0 \end{bmatrix}^{\mathrm{T}}$	
AngularVelocityLocal	${}^{b}\boldsymbol{\omega}_{\mathrm{config}} = {}^{i}\begin{bmatrix} 0 \\ 0 \\ \dot\psi_0 \end{bmatrix}^{\mathrm{T}}$	
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	${}^{0}\mathbf{f}_a = [f_0,\; f_1,\; f_2]^{\mathrm{T}}$	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, forces will have no effect and torques will only have effect in 'direction' of the coordinate.

6.3.4.3 Equations of motion

$$J \cdot \ddot{\psi}_0 = \tau. \tag{6.32}$$

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

$$\tau = {}^b[0, 0, 1] \; {}^{b0}\mathbf{A}_0 \; {}^0\boldsymbol{\tau}_a \tag{6.33}$$

Thus, the **rotation jacobian** reads

$$\mathbf{J}_{rot} = \partial \boldsymbol{\omega}_{\mathrm{cur}} / \partial \dot{q}_{0,cur} = {}^b[0, 0, 1] \; {}^{b0}\mathbf{A}_0 \tag{6.34}$$

6.3.4.4 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()
SC.TimeIntegrationSolve(mbs, 'GeneralizedAlpha', exu.SimulationSettings())

#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 further examples on ObjectRotationalMass1D see TestModels:

- `driveTrainTest.py`

6.3.5 ObjectRigidBody

A 3D rigid body which is attached to a 3D rigid body node. Equations of motion with the displacements $[u_x \;\; u_y \;\; u_z]^T$ of the center of mass and the rotation parameters (Euler parameters) $\mathbf{q}$, the mass m, inertia $\mathbf{J} = [J_{xx}, J_{xy}, J_{xz}; J_{yx}, J_{yy}, J_{yz}; J_{zx}, J_{zy}, J_{zz}]$ and the residual of all forces and moments $[R_x \;\; R_y \;\; R_z \;\; R_{q0} \;\; R_{q1} \;\; R_{q2} \;\; R_{q3}]^T$ are given as ...; REMARK: Use the class RigidBodyInertia and AddRigidBody(...) of exudynRigidBodyUtilities.py to handle inertia, COM and mass.

Additional information for ObjectRigidBody:

- The Object has 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]: $[J_{xx}, J_{yy}, J_{zz}, J_{yz}, J_{xz}, J_{xy}]$ of rigid body w.r.t. to the reference point of the body, NOT w.r.t. to center of mass; use the class RigidBodyInertia and AddRigidBody(...) of exudynRigidBodyUtilities.py to handle inertia, COM and mass
physicsCenterOfMass	Vector3D	3	[0.,0.,0.]	local position of center of mass (COM); 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		MAXINT	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 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

6.3.5.1 DESCRIPTION of ObjectRigidBody:

Information on input parameters:

input parameter	symbol	description see tables above
physicsMass	m	
physicsInertia	J	
physicsCenterOfMass	${}^{b}\mathbf{p}_{COM}$	
nodeNumber	$n0$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	symbol	description
Position	${}^{0}\mathbf{p}_{\mathrm{config}}({}^{b}\mathbf{p}) = {}^{0}\mathbf{u}_{\mathrm{config}} + {}^{0}\mathbf{p}_{\mathrm{ref}} + {}^{0b}\mathbf{A}\,{}^{b}\mathbf{p}$	global position vector of body-fixed point given by local position vector
Displacement	${}^{0}\mathbf{u}_{\mathrm{config}} + {}^{0b}\mathbf{A}\,{}^{b}\mathbf{p}$	global displacement vector of body-fixed point given by local position vector
Velocity	${}^{0}\mathbf{v}_{\mathrm{config}}({}^{b}\mathbf{p}) = {}^{0}\dot{\mathbf{u}}_{\mathrm{config}} + {}^{0b}\mathbf{A}({}^{b}\boldsymbol{\omega} \times {}^{b}\mathbf{p}_{\mathrm{config}})$	global velocity vector of body-fixed point given by local position vector
RotationMatrix		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	${}^{0}\boldsymbol{\omega}_{\mathrm{config}}$	angular velocity of body
AngularVelocityLocal	${}^{b}\boldsymbol{\omega}_{\mathrm{config}}$	local (body-fixed) 3D velocity vector of node
Acceleration	${}^{0}\mathbf{a}_{\mathrm{config}}({}^{b}\mathbf{p}) = {}^{0}\ddot{\mathbf{u}} + {}^{0}\boldsymbol{\alpha} \times ({}^{0b}\mathbf{A}\,{}^{b}\mathbf{p}) + {}^{0}\boldsymbol{\omega} \times ({}^{0}\boldsymbol{\omega} \times ({}^{0b}\mathbf{A}\,{}^{b}\mathbf{p}))$	global acceleration vector of body-fixed point given by local position vector
AngularAcceleration	${}^{0}\boldsymbol{\alpha}_{\mathrm{config}}$	angular acceleration vector of body

6.3.5.2 Definition of quantities

Detailed equations on rigid body coming soon → check C++ code for now!

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 8.3. Use `graphicsDataUtilities` functions, see Section 5.4, 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 6.3.13.

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 8.3

For creating a `ObjectRigidBody`, there is a `rigidBodyUtilities` function `AddRigidBody`, see Section 5.11, which simplifies the setup of a rigid body significantely!

For further examples on ObjectRigidBody see Examples:

- `rigid3Dexample.py`
- `rigidBodyIMUtest.py`
- `mouseInteractionExample.py`
- `rigidBodyTutorial.py`
- `sliderCrank3DwithANCFbeltDrive2.py`
- `stiffFlyballGovernor2.py`
- `ANCF_moving_rigidbody.py`
- `ANCF_slidingJoint2D.py`
- `ANCF_switchingSlidingJoint2D.py`
- `flexibleRotor3Dtest.py`
- ...

For further examples on ObjectRigidBody see TestModels:

- `explicitLieGroupIntegratorPythonTest.py`
- `explicitLieGroupIntegratorTest.py`
- `explicitLieGroupMBSTest.py`
- `genericJointUserFunctionTest.py`
- `heavyTop.py`
- `objectFFRFTest.py`
- `sphericalJointTest.py`
- `carRollingDiscTest.py`
- `driveTrainTest.py`
- `mecanumWheelRollingDiscTest.py`
- ...

6.3.6 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:

- The Object has 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. center of mass
nodeNumber	NodeIndex		MAXINT	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

---

6.3.6.1 DESCRIPTION of ObjectRigidBody2D:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
physicsMass	m	
physicsInertia	J	
nodeNumber	n_0	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
Position	${}^{0}\mathbf{p}_{\mathrm{config}}({}^{b}\mathbf{p}) = {}^{0}\mathbf{u}_{\mathrm{config}} + {}^{0}\mathbf{p}_{\mathrm{ref}} + {}^{0b}\mathbf{A}\,{}^{b}\mathbf{p}$	global position vector of body-fixed point given by local position vector
Displacement	${}^{0}\mathbf{u}_{\mathrm{config}} + {}^{0b}\mathbf{A}\,{}^{b}\mathbf{p}$	global displacement vector of body-fixed point given by local position vector
Velocity	${}^{0}\mathbf{v}_{\mathrm{config}}({}^{b}\mathbf{p}) = {}^{0}\dot{\mathbf{u}}_{\mathrm{config}} + {}^{0b}\mathbf{A}({}^{b}\boldsymbol{\omega} \times {}^{b}\mathbf{p}_{\mathrm{config}})$	global velocity vector of body-fixed point given by local position vector
Rotation	$\theta_{0\mathrm{config}}$	scalar rotation angle of body
AngularVelocity	${}^{0}\boldsymbol{\omega}_{\mathrm{config}}$	angular velocity vector of body
RotationMatrix	$\mathrm{vec}({}^{0b}\mathbf{A}) = [A_{00}, A_{01}, A_{02}, A_{10}, \ldots, A_{21}, A_{22}]^{\mathrm{T}}_{\mathrm{config}}$	rotation matrix in vector form (stored in row-major order)
Acceleration	${}^{0}\mathbf{a}_{\mathrm{config}}({}^{b}\mathbf{p}) = {}^{0}\ddot{\mathbf{u}} + {}^{0}\boldsymbol{\alpha} \times ({}^{0b}\mathbf{A}\,{}^{b}\mathbf{p}) + {}^{0}\boldsymbol{\omega} \times ({}^{0}\boldsymbol{\omega} \times ({}^{0b}\mathbf{A}\,{}^{b}\mathbf{p}))$	global acceleration vector of body-fixed point given by local position vector
AngularAcceleration	${}^{0}\boldsymbol{\alpha}_{\mathrm{config}}$	angular acceleration vector of body

6.3.6.2 Definition of quantities

intermediate variables	**symbol**	**description**
COM position	${}^{0}\mathbf{p}_{\mathrm{config}} = {}^{0}\mathbf{p}(n_0)_{\mathrm{config}}$	position of center of mass (COM) which is provided by node n_0 in any configuration
COM displacement	${}^{0}\mathbf{u}_{\mathrm{config}} = [q_0,\ q_1,\ 0]^{\mathrm{T}}_{\mathrm{config}} = {}^{0}\mathbf{u}(n_0)_{\mathrm{config}}$	displacement of center of mass which is provided by node n_0 in any configuration
COM velocity	${}^{0}\mathbf{v}_{\mathrm{config}} = [\dot{q}_0,\ \dot{q}_1,\ 0]^{\mathrm{T}}_{\mathrm{config}} = {}^{0}\mathbf{v}(n_0)_{\mathrm{config}}$	velocity of center of mass which is provided by node n_0 in any configuration
body rotation	${}^{0}\theta_{0\mathrm{config}} = \theta_0(n_0)_{\mathrm{config}} = \psi_0(n_0)_{\mathrm{ref}} + \psi_0(n_0)_{\mathrm{config}}$	rotation of body as provided by node n_0 in any configuration
body rotation matrix	${}^{0b}\mathbf{A}_{\mathrm{config}} = {}^{0b}\mathbf{A}(n_0)_{\mathrm{config}}$	rotation matrix which transforms local to global coordinates as given by node
local position	${}^{b}\mathbf{p} = [{}^{b}p_0,\ {}^{b}p_1,\ 0]^{\mathrm{T}}$	local position as used by markers or sensors
body angular velocity	${}^{0}\boldsymbol{\omega}_{\mathrm{config}} = {}^{0}[\omega_0(n_0),\ 0,\ 0]^{\mathrm{T}}_{\mathrm{config}}$	rotation of body as provided by node n_0 in any configuration
(generalized) coordinates	$\mathbf{c}_{\mathrm{config}} = [q_0, q_1,\ \psi_0]^{\mathrm{T}}$	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, ...)

6.3.6.3 Equations of motion

$$\begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & J \end{bmatrix} \begin{bmatrix} \ddot{q}_0 \\ \ddot{q}_1 \\ \ddot{\psi}_0 \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ \tau_2 \end{bmatrix} = \mathbf{f}. \tag{6.35}$$

For example, a LoadCoordinate on coordinate 2 of the node would add a torque τ_2 on the RHS.

Position-based markers can measure position $\mathbf{p}_{\mathrm{config}}$ depending on the local position ${}^b\mathbf{p}$. The **position jacobian** depends on the local position ${}^b\mathbf{p}$ and is defined as,

$$\mathbf{J}_{pos} = \partial \mathbf{p}_{\mathrm{cur}} / \partial \mathbf{c}_{\mathrm{cur}} = \begin{bmatrix} 1 & 0 & -\sin(\theta)\,{}^b p_0 - \cos(\theta)\,{}^b p_1 \\ 0 & 1 & \cos(\theta)\,{}^b p_0 - \sin(\theta)\,{}^b p_1 \\ 0 & 0 & 0 \end{bmatrix} \quad (6.36)$$

which transforms the action of global forces ${}^0\mathbf{f}$ of position-based markers on the coordinates $\mathbf{c}$,

$$\mathbf{Q} = \mathbf{J}_{pos}\,{}^0\mathbf{f}_a \quad (6.37)$$

The **rotation jacobian**

$$\mathbf{J}_{rot} = \partial \mathbf{p}_{\mathrm{cur}} / \partial \mathbf{c}_{\mathrm{cur}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (6.38)$$

transforms the action of global torques ${}^0\boldsymbol{\tau}$ of orientation-based markers on the coordinates $\mathbf{c}$,

$$\mathbf{Q} = \mathbf{J}_{rot}\,{}^0\boldsymbol{\tau}_a \quad (6.39)$$

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 8.3. Use `graphicsDataUtilities` functions, see Section 5.4, 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 6.3.13.

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 8.3

6.3.6.4 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()
SC.TimeIntegrationSolve(mbs, 'GeneralizedAlpha', exu.SimulationSettings())
```

```
#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 further examples on ObjectRigidBody2D see Examples:

- sliderCrank3DwithANCFbeltDrive2.py
- ANCF_moving_rigidbody.py
- ANCF_slidingJoint2D.py
- ANCF_switchingSlidingJoint2D.py
- geneticOptimizationSliderCrank.py
- lavalRotor2Dtest.py
- rigid_pendulum.py
- SliderCrank.py
- sliderCrank3DwithANCFbeltDrive.py
- slidercrankWithMassSpring.py
- ...

For further examples on ObjectRigidBody2D see TestModels:

- ACNFslidingAndALEjointTest.py
- ANCFcontactFrictionTest.py
- ANCFmovingRigidBodyTest.py
- compareFullModifiedNewton.py
- modelUnitTests.py
- PARTS_ATEs_moving.py
- pendulumFriction.py
- scissorPrismaticRevolute2D.py
- sliderCrankFloatingTest.py

6.3.7 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:

- The Object has 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	NumpyMatrix		Matrix[]	mass matrix of object in python numpy format
stiffnessMatrix	NumpyMatrix		Matrix[]	stiffness matrix of object in python numpy format
dampingMatrix	NumpyMatrix		Matrix[]	damping matrix of object in python numpy format
forceVector	NumpyVector		[]	generalized force vector added to RHS
forceUserFunction	PyFunctionVectorMbsScalar2Vector		0	A python user function which computes the generalized user force vector for the ODE2 equations; see description below
massMatrixUserFunction	PyFunctionMatrixMbsScalar2Vector		0	A python user function which computes the mass matrix instead of the constant mass matrix; 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

6.3.7.1 DESCRIPTION of ObjectGenericODE2:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
nodeNumbers	$\mathbf{n}_n = [n_0, \ldots, n_n]^T$	
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	$\mathbf{M}_{user} \in \mathbb{R}^{n \times n}$	
tempCoordinates	$\mathbf{c}_{temp} \in \mathbb{R}^{n_f}$	
tempCoordinates_t	$\dot{\mathbf{c}}_{temp} \in \mathbb{R}^{n_f}$	
tempCoordinates_tt	$\ddot{\mathbf{c}}_{temp} \in \mathbb{R}^{n_f}$	

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)

6.3.7.2 Additional output variables for superelement node access

Functions like `GetObjectOutputSuperElement(...)`, see Section 4.3.2, or `SensorSuperElement`, see Section 4.3.5, 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`.

6.3.7.3 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

$$n = \sum_i n_{c_i}, \tag{6.40}$$

which is used throughout the description of this object.

6.3.7.4 Equations of motion

$$\mathbf{M}\ddot{\mathbf{q}} + \mathbf{D}\dot{\mathbf{q}} + \mathbf{K}\mathbf{q} = \mathbf{f} + \mathbf{f}_{user}(mbs, t, \mathbf{q}, \dot{\mathbf{q}}) \tag{6.41}$$

Note that the user function $\mathbf{f}_{user}(mbs, t, \mathbf{q}, \dot{\mathbf{q}})$ may be empty (=0).

In case that a user mass matrix is specified, Eq. (6.41) is replaced with

$$\mathbf{M}_{user}(mbs, t, \mathbf{q}, \dot{\mathbf{q}})\ddot{\mathbf{q}} + \mathbf{D}\dot{\mathbf{q}} + \mathbf{K}\mathbf{q} = \mathbf{f} + \mathbf{f}_{user}(mbs, t, \mathbf{q}, \dot{\mathbf{q}}) \tag{6.42}$$

CoordinateLoads are added for the respective ODE2 coordinate on the RHS of the latter equation.

Userfunction: `forceUserFunction(mbs, t, 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
`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, 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
`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`	NumpyMatrix $\in \mathbb{R}^{n \times n}$	returns mass matrix for object

---

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 8.3. Use `graphicsDataUtilities` functions, see Section 5.4, 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 6.3.13.

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 8.3

User function example:

```
#user function, using variables M, K, ... from mini example, replacing
    ObjectGenericODE2(...)
KD = numpy.diag([200,100])
#nonlinear force example
def UFforce(mbs, t, q, q\_t):
    return np.dot(KD, q_t*q) #add nonlinear function for q_t and q, q_t*q gives
        vector

#non-constant mass matrix:
def UFmass(mbs, t, q, q\_t):
    return return (q[0]+1)*M #uses mass matrix from mini example
```

```
#non-constant mass matrix:
def UFgraphics(mbs, objectNum):
    t = mbs.systemData.GetTime(exu.ConfigurationType.Visualization) #get time if
        needed
    p = mbs.GetObjectOutputSuperElement(objectNumber=objectNum, variableType = exu.
        OutputVariableType.Position,
                                        meshNodeNumber = 0, #get first node's
                                            position
                                        configuration = exu.ConfigurationType.
                                            Visualization)
    graphics1=GraphicsDataSphere(point=p,radius=0.1, color=color4red)
        graphics2 = {'type':'Line', 'data': list(p)+[0,0,0], 'color':color4blue}
    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)
```

6.3.7.5 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()

sims=exu.SimulationSettings()
sims.timeIntegration.generalizedAlpha.spectralRadius=1
SC.TimeIntegrationSolve(mbs, 'GeneralizedAlpha', sims)

#check result at default integration time
exudynTestGlobals.testResult = mbs.GetNodeOutput(nMass1, exu.OutputVariableType.
    Position)[0]
```

For further examples on ObjectGenericODE2 see Examples:

- `simulateInteractively.py`

For further examples on ObjectGenericODE2 see TestModels:

- `genericODE2test.py`
- `objectFFRFTest.py`
- `objectGenericODE2Test.py`
- `solverExplicitODE1ODE2test.py`

6.3.8 ObjectGenericODE1

A system of n second 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:

- The Object has 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	PyFunctionVectorMbsScalarVector		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

---

6.3.8.1 DESCRIPTION of ObjectGenericODE1:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
nodeNumbers	$\mathbf{n}_n = [n_0, \ldots, n_n]^T$	

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	$\mathbf{c}_{temp} \in \mathbb{R}^{n_f}$	
tempCoordinates_t	$\dot{\mathbf{c}}_{temp} \in \mathbb{R}^{n_f}$	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
Coordinates		all ODE1 coordinates
Coordinates_t		all ODE1 velocity coordinates

6.3.8.2 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

$$n = \sum_i n_{c_i}, \tag{6.43}$$

which is used throughout the description of this object.

6.3.8.3 Equations of motion

$$\dot{\mathbf{q}} = \mathbf{f} + \mathbf{f}_{user}(mbs, t, \mathbf{q}) \tag{6.44}$$

Note that the user function $\mathbf{f}_{user}(mbs, t, \mathbf{q})$ may be empty (=0). CoordinateLoads are added for the respective ODE1 coordinate on the RHS of the latter equation.

Userfunction: `rhsUserFunction(mbs, t, 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).

arguments / return	**type or size**	**description**
`mbs`	MainSystem	provides MainSystem mbs to which object belongs
`t`	Real	current time in mbs
`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
def UFrhs(mbs, t, q, q\_t):
    return np.dot(A, q) + np.array([0,2])

nODE1 = mbs.AddNode(NodeGenericODE1(referenceCoordinates=[0,0],
                                    initialCoordinates=[1,0]))

#now add object instead of object in mini-example:
oGenericODE1 = mbs.AddObject(ObjectGenericODE1(nodeNumbers=[nODE1],
                   rhsUserFunction=UFrhs)
```

6.3.8.4 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
exu.SolveDynamic(mbs, solverType=solverType, simulationSettings=sims)

#check result at default integration time
exudynTestGlobals.testResult = mbs.GetNodeOutput(nODE1, exu.OutputVariableType.
    Coordinates)[0]
```

For further examples on ObjectGenericODE1 see TestModels:

- solverExplicitODE1ODE2test.py

6.3.9 ObjectFFRF

This object is used to represent equations modelled by the floating frame of reference formulation (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.

Additional information for ObjectFFRF:

- The Object has 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 $(n_f + 1)$ 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, global force vector added to RHS; the rigid body part $\mathbf{f}_r$ is directly applied to rigid body coordinates while the flexible part $\mathbf{f}_{ff}$ is transformed from global to local coordinates
forceUserFunction	PyFunctionVectorMbsScalar2Vector		0	A python user function which computes the generalized user force vector for the ODE2 equations; The function takes the time, coordinates q (without reference values) and coordinate velocities q_t; see description below
massMatrixUserFunction	PyFunctionMatrixMbsScalar2Vector		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 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	flag used to correctly initialize all FFRF matrices; as soon as this flag is set false, FFRF matrices and terms are recomputed
physicsMass	UReal		0.	total mass [SI:kg] of FFRF object, auto-computed from mass matrix $\mathbf{M}$
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 $\mathbf{M}_{ff}$
physicsCenterOfMass	Vector3D	3	[0.,0.,0.]	local position of center of mass (COM); auto-computed from mass matrix $\mathbf{M}$
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'

6.3.9.1 DESCRIPTION of ObjectFFRF:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
nodeNumbers	$\mathbf{n}_n = [n_0, \ldots, n_{n_f}]^\mathrm{T}$	
massMatrixFF	$\mathbf{M}_{ff} \in \mathbb{R}^{n_{c_f} \times n_{c_f}}$	
stiffnessMatrixFF	$\mathbf{K}_{ff} \in \mathbb{R}^{n_{c_f} \times n_{c_f}}$	
dampingMatrixFF	$\mathbf{D}_{ff} \in \mathbb{R}^{n_{c_f} \times n_{c_f}}$	
forceVector	$\mathbf{f} \in \mathbb{R}^{n_c}$	
forceUserFunction	$\mathbf{f}_{user} \in \mathbb{R}^{n_c}$	
massMatrixUserFunction	$\mathbf{M}_{user} \in \mathbb{R}^{n_c \times n_c}$	
physicsMass	m	
physicsInertia	$J_r \in \mathbb{R}^{3\times 3}$	
physicsCenterOfMass	${}^b\mathbf{p}_{COM}$	
PHItTM	$\Phi_t^\mathrm{T} \in \mathbb{R}^{n_{c_f} \times 3}$	
referencePositions	$\mathbf{x}_f \in \mathbb{R}^{n_f}$	
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	$\tilde{\mathbf{p}}_f \in \mathbb{R}^{n_{c_f} \times 3}$	
tempVelSkew	$\tilde{\dot{\mathbf{c}}}_f \in \mathbb{R}^{n_{c_f} \times 3}$	

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)

6.3.9.2 Additional output variables for superelement node access

Functions like `GetObjectOutputSuperElement(...)`, see Section 4.3.2, or `SensorSuperElement`, see Section 4.3.5, 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.

6.3.9.3 Super element output variables

super element output variables	symbol	description
Position	${}^{0}\mathbf{p}_{\,\mathrm{config}}(n_i) = {}^{0}\mathbf{r}_{\,\mathrm{config}} + {}^{0b}\mathbf{A}_{\,\mathrm{config}}\, {}^{b}\mathbf{p}_{\,\mathrm{config}}(n_i)$	global position of mesh node n_i including rigid body motion and flexible deformation
Displacement	${}^{0}\mathbf{u}_{\,\mathrm{config}}(n_i) = {}^{0}\mathbf{p}_{\,\mathrm{config}}(n_i) - {}^{0}\mathbf{p}_{\,\mathrm{ref}}(n_i)$	global displacement of mesh node n_i including rigid body motion and flexible deformation
Velocity	${}^{0}\mathbf{v}_{\,\mathrm{config}}(n_i) = {}^{0}\dot{\mathbf{r}}_{\,\mathrm{config}} + {}^{0b}\mathbf{A}_{\,\mathrm{config}}({}^{b}\dot{\mathbf{x}}_{\,\mathrm{config}}(n_i) + {}^{b}\boldsymbol{\omega}_{\,\mathrm{config}} \times {}^{b}\mathbf{x}_{\,\mathrm{config}}(n_i))$	global velocity of mesh node n_i including rigid body motion and flexible deformation
Acceleration	${}^{0}\mathbf{a}_{\,\mathrm{config}}(n_i) = {}^{0}\ddot{\mathbf{r}}_{\,\mathrm{config}} + {}^{0b}\mathbf{A}_{\,\mathrm{config}}\, {}^{b}\ddot{\mathbf{x}}_{\,\mathrm{config}}(n_i) + 2\, {}^{0}\boldsymbol{\omega}_{\,\mathrm{config}} \times {}^{0b}\mathbf{A}_{\,\mathrm{config}}\, {}^{b}\dot{\mathbf{x}}_{\,\mathrm{config}}(n_i) + {}^{0}\boldsymbol{\alpha}_{\,\mathrm{config}} \times {}^{0}\mathbf{x}_{\,\mathrm{config}}(n_i)) + {}^{0}\boldsymbol{\omega}_{\,\mathrm{config}} \times {}^{0}\boldsymbol{\omega}_{\,\mathrm{config}} \times {}^{0}\mathbf{x}_{\,\mathrm{config}}(n_i))$	global acceleration of mesh node n_i including rigid body motion and flexible deformation; note that ${}^{0}\mathbf{x}_{\,\mathrm{config}}(n_i) = {}^{0b}\mathbf{A}\, {}^{b}\mathbf{x}_{\,\mathrm{config}}(n_i)$
DisplacementLocal	${}^{b}\mathbf{d}_{\,\mathrm{config}}(n_i) = {}^{b}\mathbf{x}_{\,\mathrm{config}}(n_i) - {}^{b}\mathbf{x}_{\,\mathrm{ref}}(n_i)$	local displacement of mesh node n_i, representing the flexible deformation within the body frame; note that ${}^{0}\mathbf{u}_{\,\mathrm{config}} \neq {}^{0b}\mathbf{A}\, {}^{b}\mathbf{d}_{\,\mathrm{config}}$!
VelocityLocal	${}^{b}\dot{\mathbf{x}}_{\,\mathrm{config}}(n_i)$	local velocity of mesh node n_i, representing the rate of flexible deformation within the body frame

6.3.9.4 Definition of quantities

intermediate variables	symbol	description
object coordinates	$\mathbf{c} = [\mathbf{c}_r,\, \mathbf{q}_f]^{\mathrm{T}}$	object coordinates
rigid body coordinates	$\mathbf{c}_r = [q_0,\, q_1,\, q_2,\, \psi_0,\, \psi_1,\, \psi_2,\, \psi_3]^{\mathrm{T}}$	rigid body coordinates in case of Euler parameters
rotation coordinates	$\boldsymbol{\theta}_{\mathrm{cur}} = [\psi_0,\, \psi_1,\, \psi_2,\, \psi_3]^{\mathrm{T}}_{\mathrm{ref}} + [\psi_0,\, \psi_1,\, \psi_2,\, \psi_3]^{\mathrm{T}}_{\mathrm{cur}}$	rigid body coordinates in case of Euler parameters
flexible coordinates	$\mathbf{q}_f$	flexible, body-fixed coordinates
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
reference frame origin	${}^{0}\mathbf{r}_{\,\mathrm{config}} = [p_0,\, p_1,\, p_2]^{\mathrm{T}}_{\mathrm{config}}$	global position of underlying rigid body node n_0 which defines the reference frame origin
reference frame orientation	${}^{0b}\mathbf{A}_{\,\mathrm{config}}$	transformation matrix for transformation of local (reference frame) to global coordinates, given by underlying rigid body node n_0

6.3.9.5 Equations of motion

Consider an object with $n = 1 + n_f$ nodes, n_f being the number of 'flexible' nodes. It has node numbers $[n_0,\, \ldots,\, n_{n_f}]$ and according numbers of nodal coordinates $[n_{c_0},\, \ldots,\, n_{c_n}]$, where n_0 denotes the rigid body node. This gives n_c total nodal coordinates,

$$n_c = \sum_{i=0}^{n_f} n_{c_i}. \qquad (6.45)$$

whereof the flexible coordinates are

$$n_{c_f} = \sum_{i=1}^{n_f} n_{c_i}. \tag{6.46}$$

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 $n_{c_r} = n_{c_0}$ coordinates (e.g., $n_{c_r} = 6$ coordinates for Euler angles and $n_{c_r} = 7$ coordinates in case of Euler parameters).

Equations of motion, in case that `computeFFRFterms = True`:

$$\left(\mathbf{M}_{user}(mbs, t, \mathbf{c}, \dot{\mathbf{c}}) + \begin{bmatrix} \mathbf{M}_{tt} & \mathbf{M}_{tr} & \mathbf{M}_{tf} \\ & \mathbf{M}_{rr} & \mathbf{M}_{rf} \\ \mathrm{sym.} & & \mathbf{M}_{ff} \end{bmatrix}\right) \ddot{\mathbf{c}} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathbf{D}_{ff} \end{bmatrix} \dot{\mathbf{c}} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathbf{K}_{ff} \end{bmatrix} \mathbf{c} = \mathbf{f}_Q(\mathbf{c}, \dot{\mathbf{c}}) + \begin{bmatrix} \mathbf{f}_r \\ {}^{0b}\mathbf{A}_{bd}^{\mathrm{T}}\, \mathbf{f}_{ff} \end{bmatrix} + \mathbf{f}_{user}(mbs, t, \mathbf{c}, \dot{\mathbf{c}}) \tag{6.47}$$

In case that `computeFFRFterms = False`, the mass terms $\mathbf{M}_{tt} \ldots \mathbf{M}_{ff}$ are zero (not computed) and the quadratic velocity vector $\mathbf{f}_Q = \mathbf{0}$. Note that the user functions $\mathbf{f}_{user}(mbs, t, \mathbf{c}, \dot{\mathbf{c}})$ and $\mathbf{M}_{user}(mbs, t, \mathbf{c}, \dot{\mathbf{c}})$ may be empty (=0). The detailed equations of motion for this element can be found in [28].

CoordinateLoads are integrated for each ODE2 coordinate on the RHS of the latter equation.
If the rigid body node is using Euler parameters $\boldsymbol{\theta} = [\theta_0,\, \theta_1,\, \theta_2,\, \theta_3]^{\mathrm{T}}$, an **additional constraint** (constraint nr. 0) is added automatically for the Euler parameter norm, reading

$$1 - \sum_{i=0}^{3} \theta_i^2 = 0. \tag{6.48}$$

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

$$\Phi_t^{\mathrm{T}} \mathbf{M} \mathbf{c}_f = 0 \tag{6.49}$$

and 3 constraints for the rotation,

$$\tilde{\mathbf{x}}_f^{\mathrm{T}} \mathbf{M} \mathbf{c}_f = 0 \tag{6.50}$$

Userfunction: `forceUserFunction(mbs, t, 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
`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, 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
`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 further examples on ObjectFFRF see Examples:

- `sliderCrankCMSacme.py`

For further examples on ObjectFFRF see TestModels:

- `objectFFRFTest.py`
- `objectFFRFTest2.py`

6.3.10 ObjectFFRFreducedOrder

This object is used to represent modally reduced flexible bodies using the floating frame of reference formulation (FFRF) and the component mode synthesis. It contains a RigidBodyNode (always node 0) and a NodeGenericODE2 representing the modal coordinates.

Additional information for ObjectFFRFreducedOrder:

- The Object has 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	PyFunctionVectorMbsScalar2Vector		0	A python user function which computes the generalized user force vector for the ODE2 equations; see description below
massMatrixUserFunction	PyFunctionMatrixMbsScalar2Vector		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 $[u_{x,n_0},\, u_{y,n_0},\, u_{z,n_0},\, \ldots,\, u_{x,n_n},\, u_{y,n_n},\, u_{z,n_n}]^\mathrm{T}$
outputVariableModeBasis	NumpyMatrix		Matrix[]	mode basis, which transforms reduced coordinates to output variables per mode; 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
physicsMass	UReal		0.	total mass [SI:kg] of FFRF object, auto-computed from mass matrix $\mathbf{M}$
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 $\mathbf{M}_{ff}$
physicsCenterOfMass	Vector3D	3	[0.,0.,0.]	local position of center of mass (COM); auto-computed from mass matrix $\mathbf{M}$
PHItTM	NumpyMatrix		Matrix[]	projector matrix; may be removed in future
tempUserFunctionForce	NumpyVector		[]	temporary vector for UF force
tempRefPosSkew	NumpyMatrix		Matrix[]	matrix with skew symmetric local (deformed) node positions
tempVelSkew	NumpyMatrix		Matrix[]	matrix with skew symmetric local node velocities
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'

6.3.10.1 DESCRIPTION of ObjectFFRFreducedOrder:

Information on input parameters:

input parameter	symbol	description see tables above
nodeNumbers	$\mathbf{n} = [n_0,\, n_1]^\mathrm{T}$	
massMatrixReduced	$\mathbf{M}_{red} \in \mathbb{R}^{n_{c_f} \times n_{c_f}}$	
stiffnessMatrixReduced	$\mathbf{K}_{red} \in \mathbb{R}^{n_{c_f} \times n_{c_f}}$	
dampingMatrixReduced	$\mathbf{D}_{red} \in \mathbb{R}^{n_{c_f} \times n_{c_f}}$	
forceUserFunction	$\mathbf{f}_{user} \in \mathbb{R}^{n_c}$	
massMatrixUserFunction	$\mathbf{M}_{user} \in \mathbb{R}^{n_c \times n_c}$	
modeBasis	$\boldsymbol{\psi} \in \mathbb{R}^{n_{c_f} \times n_m}$	
outputVariableModeBasis	$\boldsymbol{\psi}_{OV} \in \mathbb{R}^{n_n \times (n_m \cdot s_{OV})}$	
referencePositions	${}^b\mathbf{r}_f \in \mathbb{R}^{n_f}$	
physicsMass	m	
physicsInertia	$J_r \in \mathbb{R}^{3 \times 3}$	
physicsCenterOfMass	${}^b\mathbf{p}_{COM}$	
PHItTM	$\Phi_t^\mathrm{T} \in \mathbb{R}^{n_{c_f} \times 3}$	
tempUserFunctionForce	$\mathbf{v}_{temp} \in \mathbb{R}^{n_c}$	
tempRefPosSkew	$\tilde{\mathbf{p}}_f \in \mathbb{R}^{n_{c_f} \times 3}$	
tempVelSkew	$\dot{\tilde{\mathbf{c}}}_f \in \mathbb{R}^{n_{c_f} \times 3}$	

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
Force		generalized forces for all coordinates (residual of all forces except mass*accleration; corresponds to ComputeODE2LHS)
Stress		allows to compute linearized, corotational nodal stresses (in mesh nodes, in body frame) based on modal stress values provided in outputVariableModeBasis; the flag outputVariableTypeModeBasis must be set in this case to exu.Outputvariable.Stress
Strain		allows to compute linearized, corotational nodal strains (in mesh nodes, in body frame) based on modal strain values provided in outputVariableModeBasis; the flag outputVariableTypeModeBasis must be set in this case to exu.Outputvariable.Strain

6.3.10.2 Additional output variables for superelement node access

Functions like `GetObjectOutputSuperElement(...)`, see Section 4.3.2, or `SensorSuperElement`, see Section 4.3.5, 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.

6.3.10.3 Super element output variables

super element output variables	symbol	description
Position	${}^0\mathbf{p}_{\mathrm{config}}(n_i) = {}^0\mathbf{r}_{\mathrm{config}} + {}^{0b}\mathbf{A}_{\mathrm{config}}\,{}^b\mathbf{p}_{\mathrm{config}}(n_i)$	global position of mesh node n_i including rigid body motion and flexible deformation
Displacement	${}^0\mathbf{u}_{\mathrm{config}}(n_i) = {}^0\mathbf{p}_{\mathrm{config}}(n_i) - {}^0\mathbf{p}_{\mathrm{ref}}(n_i)$	global displacement of mesh node n_i including rigid body motion and flexible deformation
Velocity	${}^0\mathbf{v}_{\mathrm{config}}(n_i) = {}^0\dot{\mathbf{r}}_{\mathrm{config}} + {}^{0b}\mathbf{A}_{\mathrm{config}}({}^b\dot{\mathbf{x}}_{\mathrm{config}}(n_i) + {}^b\boldsymbol{\omega}_{\mathrm{config}} \times {}^b\mathbf{x}_{\mathrm{config}}(n_i))$	global velocity of mesh node n_i including rigid body motion and flexible deformation
Acceleration	${}^0\mathbf{a}_{\mathrm{config}}(n_i) = {}^0\ddot{\mathbf{r}}_{\mathrm{config}} + {}^{0b}\mathbf{A}_{\mathrm{config}}\,{}^b\ddot{\mathbf{x}}_{\mathrm{config}}(n_i) + 2\,{}^0\boldsymbol{\omega}_{\mathrm{config}} \times {}^{0b}\mathbf{A}_{\mathrm{config}}\,{}^b\dot{\mathbf{x}}_{\mathrm{config}}(n_i) + {}^0\boldsymbol{\alpha}_{\mathrm{config}} \times {}^0\mathbf{x}_{\mathrm{config}}(n_i)) + {}^0\boldsymbol{\omega}_{\mathrm{config}} \times {}^0\boldsymbol{\omega}_{\mathrm{config}} \times {}^0\mathbf{x}_{\mathrm{config}}(n_i))$	global acceleration of mesh node n_i including rigid body motion and flexible deformation; note that ${}^0\mathbf{x}_{\mathrm{config}}(n_i) = {}^{0b}\mathbf{A}\,{}^b\mathbf{x}_{\mathrm{config}}(n_i)$
DisplacementLocal	${}^b\mathbf{d}_{\mathrm{config}}(n_i) = {}^b\mathbf{x}_{\mathrm{config}}(n_i) - {}^b\mathbf{x}_{\mathrm{ref}}(n_i)$	local displacement of mesh node n_i, representing the flexible deformation within the body frame; note that ${}^0\mathbf{u}_{\mathrm{config}} \neq {}^{0b}\mathbf{A}\,{}^b\mathbf{d}_{\mathrm{config}}$!
VelocityLocal	${}^b\dot{\mathbf{x}}_{\mathrm{config}}(n_i)$	local velocity of mesh node n_i, representing the rate of flexible deformation within the body frame

6.3.10.4 Definition of quantities

The object additionally provides the following output variables for mesh nodes (use `mbs.GetObjectOutputSuperElement(..` or `SensorSuperElement`):

mesh node output variables	symbol	description
Position	${}^0\mathbf{r}_{n_i}$	position of node with mesh node number n_i in global coordinates
Position	${}^0\mathbf{r}_{n_i}$	position of node with mesh node number n_i in global coordinates
DisplacementLocal (mesh node *i*)	${}^b\mathbf{u}_{f,i}$	local nodal mesh displacement in reference (body) frame
VelocityLocal (mesh node *i*)	${}^b\dot{\mathbf{u}}_{f,i}$	local nodal mesh velocity in reference (body) frame
Displacement (mesh node *i*)	${}^0\mathbf{u}_{i,config} = {}^0\mathbf{q}_{t,config} + {}^{0b}\mathbf{A}_{config}\,{}^b\mathbf{p}_{f,i,config} - ({}^0\mathbf{q}_{t,ref} + {}^{0b}\mathbf{A}_{ref}\,{}^b\mathbf{r}_{f,i})$	nodal mesh displacement in global coordinates
Position (mesh node *i*)	${}^0\mathbf{p}_i = {}^0\mathbf{p}_t + {}^{0b}\mathbf{A}\,{}^b\mathbf{p}_{f,i}$	nodal mesh displacement in global coordinates
Velocity (mesh node *i*)	${}^0\dot{\mathbf{u}}_i = {}^0\dot{\mathbf{q}}_t + {}^{0b}\mathbf{A}\,({}^b\dot{\mathbf{u}}_{f,i} + {}^b\tilde{\boldsymbol{\omega}}\,{}^b\mathbf{u}_{f,i})$	nodal mesh velocity in global coordinates
Stress (mesh node *i*)	${}^b\boldsymbol{\sigma}_i = (\boldsymbol{\psi}_{OV}\boldsymbol{\zeta})_{3\cdot i \ldots 3\cdot i+5}$	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 $\boldsymbol{\psi}_{OV}$ is provided and `outputVariableTypeModeBasis==exu.OutputVariableT`

Strain (mesh node i)	${}^b\boldsymbol{\varepsilon}_i = (\boldsymbol{\psi}_{OV}\boldsymbol{\zeta})_{3\cdot i \ldots 3\cdot i+5}$	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 $\boldsymbol{\psi}_{OV}$ is provided and `outputVariableTypeModeBasis==exu.OutputVariableT`

intermediate variables	symbol	description
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
coordinate vector	$\mathbf{q} = [{}^0\mathbf{q}_t, \boldsymbol{\psi}, \boldsymbol{\zeta}]$	vector of object coordinates; $\mathbf{q}_t$ and $\boldsymbol{\psi}$ are the translation and rotation part of displacements of the reference frame, provided by the rigid body node (node number 0)
reference frame origin	${}^0\mathbf{r}_{\mathrm{config}} = {}^0\mathbf{q}_{t,config} + {}^0\mathbf{q}_{t,ref}$	reference frame position in any configuration except reference
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	${}^{0b}\mathbf{A}_{\mathrm{config}}$	transformation matrix for transformation of local (reference frame) to global coordinates, given by underlying rigid body node n_0
vector of modal coordinates	$\boldsymbol{\zeta} = [\zeta_0, \ldots, \zeta_{n_m-1}]^{\mathrm{T}}$	vector of modal coordinates
vector of mesh coordinates	${}^b\mathbf{q}_f = \boldsymbol{\psi}\boldsymbol{\zeta}$	vector of alternating x,y, an z coordinates of local (in body frame) mesh displacements reconstructed from modal coordinates $\boldsymbol{\zeta}$
local mesh displacements	${}^b\mathbf{u}_{f,i} = \begin{bmatrix} {}^b\mathbf{q}_{f,i\cdot 3} \\ {}^b\mathbf{q}_{f,i\cdot 3+1} \\ {}^b\mathbf{q}_{f,i\cdot 3+2} \end{bmatrix}$	nodal mesh displacement in local coordinates (body frame)
local mesh position	${}^b\mathbf{p}_{f,i} = \begin{bmatrix} {}^b\mathbf{q}_{f,i\cdot 3} \\ {}^b\mathbf{q}_{f,i\cdot 3+1} \\ {}^b\mathbf{q}_{f,i\cdot 3+2} \end{bmatrix} + \begin{bmatrix} {}^b\mathbf{r}_{f,i\cdot 3} \\ {}^b\mathbf{r}_{f,i\cdot 3+1} \\ {}^b\mathbf{r}_{f,i\cdot 3+2} \end{bmatrix}$	(deformed) nodal mesh position in local coordinates (body frame)

6.3.10.5 Equations motion

Some definitions:

- body frame (b) = reference frame
- n_n ... number of mesh nodes
- $n_c = 3 \cdot n_n$ ... number of mesh coordinates
- n_{rigid} ... number of rigid body node coordinates: 6 in case of Euler angles and 7 in case of Euler parameters
- $n_{ODE2} = n_c + n_{rigid}$ ... total number of object coordinates
- n_m ... number of modal coordinates; computed from number of columns in modeBasis
- $\boldsymbol{\psi}$ ... mode basis, containing eigenmodes and static modes
- ${}^b\mathbf{r}_f$ ... node reference coordinates for mesh nodes

Equations of motion, in case that `computeFFRFterms = True`:

$$\left(\mathbf{M}_{user}(mbs, t, \mathbf{c}, \dot{\mathbf{c}}) + \begin{bmatrix} \mathbf{M}_{tt} & \mathbf{M}_{tr} & \mathbf{M}_{tf} \\ & \mathbf{M}_{rr} & \mathbf{M}_{rf} \\ \mathrm{sym.} & & \mathbf{M}_{ff} \end{bmatrix} \right) \ddot{\mathbf{c}} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathbf{D}_{ff} \end{bmatrix} \dot{\mathbf{c}} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathbf{K}_{ff} \end{bmatrix} \mathbf{c} = \mathbf{f}_Q(\mathbf{c}, \dot{\mathbf{c}}) + \mathbf{f}_{user}(mbs, t, \mathbf{c}, \dot{\mathbf{c}}) \quad (6.51)$$

$\rightarrow$ will be completed later, see according literature of Zwölfer and Gerstmayr, 2020.

In case that `computeFFRFterms = False`, the mass terms $\mathbf{M}_{tt} \ldots \mathbf{M}_{ff}$ are zero (not computed) and the quadratic velocity vector $\mathbf{f}_Q = \mathbf{0}$. Note that the user functions $\mathbf{f}_{user}(mbs, t, \mathbf{c}, \dot{\mathbf{c}})$ and $\mathbf{M}_{user}(mbs, t, \mathbf{c}, \dot{\mathbf{c}})$ may be empty (=0). The detailed equations of motion for this element can be found in [29].

CoordinateLoads are added for each ODE2 coordinate on the RHS of the latter equation.

Userfunction: `forceUserFunction(mbs, t, 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
`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, 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
`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 further examples on ObjectFFRFreducedOrder see Examples:

- `NGsolvePistonEngine.py`
- `objectFFRFreducedOrderNetgen.py`
- `sliderCrankCMSacme.py`

For further examples on ObjectFFRFreducedOrder see TestModels:

- `NGsolveCrankShaftTest.py`

- objectFFRFreducedOrderAccelerations.py
- objectFFRFreducedOrderStressModesTest.py
- objectFFRFreducedOrderTest.py
- superElementRigidJointTest.py

6.3.11 ObjectANCFCable2D

A 2D cable finite element using 2 nodes of type NodePoint2DSlope1.

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.	reference length L [SI:m] of beam; such that the total volume (e.g. for volume load) gives ρAL
physicsMassPerLength	UReal		0.	mass ρA [SI:kg/m^2] of beam
physicsBendingStiffness	UReal		0.	bending stiffness EI [SI:Nm^2] of beam; the bending moment is $m = EI(\kappa - \kappa_0)$, in which κ is the material measure of curvature
physicsAxialStiffness	UReal		0.	axial stiffness EA [SI:N] 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.	bending damping d_{EI} [SI:Nm^2/s] 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.	axial stiffness d_{EA} [SI:N/s] of beam; the additional virtual work due to damping is $\delta W_{\dot\varepsilon} = \int_0^L \dot\varepsilon \delta\varepsilon dx$
physicsReferenceAxialStrain	UReal		0.	reference axial strain of beam (pre-deformation) ε_0 [SI:1] of beam; without external loading the beam will statically keep the reference axial strain value
physicsReferenceCurvature	UReal		0.	reference curvature of beam (pre-deformation) κ_0 [SI:1/m] of beam; without external loading the beam will statically keep the reference curvature value
nodeNumbers	NodeIndex2		[MAXINT, MAXINT]	two node numbers ANCF cable element
useReducedOrderIntegration	Bool		False	false: use Gauss order 9 integration for virtual work of axial forces, order 5 for virtual work of bending moments; true: use Gauss order 7 integration for virtual work of axial forces, order 3 for virtual work of bending moments
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

---

6.3.11.1 DESCRIPTION of ObjectANCFCable2D:

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
Director1		(axial) slope vector of local axis position
Strain		axial strain (scalar)
Curvature		axial strain (scalar)
Force		(local) section normal force (scalar)
Torque		(local) bending moment (scalar)

A 2D cable finite element using 2 nodes of type NodePoint2DSlope1. The element has 8 coordinates and uses cubic polynomials for position interpolation. The Bernoulli-Euler beam is capable of large deformation as it employs the material measure of curvature for the bending.

For a detailed description of this beam element, see Gerstmayr and Irschik [12].

For further examples on ObjectANCFCable2D see Examples:

- `sliderCrank3DwithANCFbeltDrive2.py`
- `ALEANCF_pipe.py`
- `ANCF_cantilever_test_dyn.py`
- `ANCF_contact_circle.py`
- `ANCF_contact_circle2.py`
- `ANCF_moving_rigidbody.py`
- `ANCF_slidingJoint2D.py`
- `ANCF_switchingSlidingJoint2D.py`
- `ANCF_tests2.py`
- `ANCF_test_halfcircle.py`
- ...

For further examples on ObjectANCFCable2D see TestModels:

- ANCFcontactCircleTest.py
- ANCFcontactFrictionTest.py
- computeODE2EigenvaluesTest.py
- manualExplicitIntegrator.py
- modelUnitTests.py

6.3.12 ObjectALEANCFCable2D

A 2D cable finite element using 2 nodes of type NodePoint2DSlope1 and a axially moving coordinate of type NodeGenericODE2.

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.	reference length L [SI:m] of beam; such that the total volume (e.g. for volume load) gives ρAL
physicsMassPerLength	UReal		0.	mass ρA [SI:kg/m^2] of beam
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.	bending stiffness EI [SI:Nm^2] of beam; the bending moment is $m = EI(\kappa - \kappa_0)$, in which κ is the material measure of curvature
physicsAxialStiffness	UReal		0.	axial stiffness EA [SI:N] 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.	bending damping d_{EI} [SI:Nm^2/s] 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.	axial stiffness d_{EA} [SI:N/s] of beam; the additional virtual work due to damping is $\delta W_{\dot\varepsilon} = \int_0^L \dot\varepsilon \delta\varepsilon dx$
physicsReferenceAxialStrain	UReal		0.	reference axial strain of beam (pre-deformation) ε_0 [SI:1] of beam; without external loading the beam will statically keep the reference axial strain value
physicsReferenceCurvature	UReal		0.	reference curvature of beam (pre-deformation) κ_0 [SI:1/m] 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)
nodeNumbers	NodeIndex3		[MAXINT, MAXINT, MAXINT]	two node numbers ANCF cable element, third node=ALE GenericODE2 node
useReducedOrderIntegration	Bool		False	false: use Gauss order 9 integration for virtual work of axial forces, order 5 for virtual work of bending moments; true: use Gauss order 7 integration for virtual work of axial forces, order 3 for virtual work of bending moments
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

---

6.3.12.1 DESCRIPTION of ObjectALEANCFCable2D:

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
Director1		(axial) slope vector of local axis position
Strain		axial strain (scalar)
Curvature		axial strain (scalar)
Force		(local) section normal force (scalar)
Torque		(local) bending moment (scalar)

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.

A detailed paper on this element is yet under submission, but a similar formulation can be found in [20].

For further examples on ObjectALEANCFCable2D see Examples:

- `ALEANCF_pipe.py`
- `ANCF_moving_rigidbody.py`

6.3.13 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:

- The Object has 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 position for ground object; local position is added on top of reference position for a ground object
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 bodyGraphicsData object, which is a list of graphics data in a dictionary computed by the user function
color	Float4		[-1.,-1.,-1.,-1.]	RGB node color; if R==-1, use default color
graphicsData	BodyGraphicsData			Structure contains data for body visualization; data is defined in special list / dictionary structure

---

6.3.13.1 DESCRIPTION of ObjectGround:

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

output variable	**symbol**	**description**
Position		global position vector of rotated and translated local position
Displacement		global displacement vector of local position
Velocity		global velocity vector of local position
AngularVelocity		angular velocity of body
RotationMatrix		rotation matrix in vector form (stored in row-major order)

6.3.13.2 Equations

ObjectGround has no equations, as it only provides a static object, at which joints and connectors can be attached. The object cannot move and forces or torques do not have an effect.

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 8.3. Use `graphicsDataUtilities` functions, see Section 5.4, 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 8.3

User function example:

```
import exudyn as exu
from math import sin, cos, pi
from exudyn.itemInterface import *
from exudyn.graphicsDataUtilities import *
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=GraphicsDataSphere(point=[sin(t*2*pi), cos(t*2*pi), 0],
                                 radius=0.1, color=color4red, 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
```

```
exu.StartRenderer() #perform zoom all (press 'a' several times) after startup to see
    the sphere
exu.SolveDynamic(mbs, sims)
exu.StopRenderer()
```

For further examples on ObjectGround see Examples:

- ALEANCF_pipe.py
- ANCF_cantilever_test.py
- ANCF_cantilever_test_dyn.py
- ANCF_contact_circle.py
- ANCF_contact_circle2.py
- ANCF_moving_rigidbody.py
- ANCF_slidingJoint2D.py
- ANCF_switchingSlidingJoint2D.py
- ANCF_tests2.py
- ANCF_test_halfcircle.py
- ...

For further examples on ObjectGround see TestModels:

- ACNFslidingAndALEjointTest.py
- ANCFcontactCircleTest.py
- ANCFcontactFrictionTest.py
- ANCFmovingRigidBodyTest.py
- carRollingDiscTest.py
- compareFullModifiedNewton.py
- connectorRigidBodySpringDamperTest.py
- driveTrainTest.py
- explicitLieGroupIntegratorPythonTest.py
- explicitLieGroupIntegratorTest.py
- ...

6.3.14 ObjectConnectorSpringDamper

An simple spring-damper element with additional force; connects to position-based markers.

Additional information for ObjectConnectorSpringDamper:

- The Object has 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		[MAXINT, MAXINT]	list of markers used in connector
referenceLength	UReal		0.	reference length [SI:m] of spring
stiffness	UReal		0.	stiffness [SI:N/m] of spring; acts against (length-initialLength)
damping	UReal		0.	damping [SI:N/(m s)] of damper; acts against d/dt(length)
force	UReal		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
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temorarily) a connector or constraint
springForceUserFunction	PyFunctionMbsScalar6		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

6.3.14.1 DESCRIPTION of ObjectConnectorSpringDamper:

Information on input parameters:

input parameter	symbol	description see tables above
markerNumbers	$[m0, m1]^{\mathrm{T}}$	
referenceLength	L_0	
stiffness	k	
damping	d	
force	f_a	

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		spring-damper force

6.3.14.2 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
Distance	L	$\lvert \Delta^{0}\mathbf{p} \rvert$
Displacement	$\Delta^{0}\mathbf{p}$	${}^{0}\mathbf{p}_{m1} - {}^{0}\mathbf{p}_{m0}$
Velocity	$\Delta^{0}\mathbf{v}$	${}^{0}\mathbf{v}_{m1} - {}^{0}\mathbf{v}_{m0}$
Force	$\mathbf{f}$	see below

6.3.14.3 Connector forces

The unit vector in force direction reads (raises SysError if $L = 0$),

$$\mathbf{v}_f = \frac{1}{L}\Delta^{0}\mathbf{p} \tag{6.52}$$

If `activeConnector = True`, the scalar spring force is computed as

$$f_{SD} = k \cdot (L - L_0) + d \cdot \Delta^{0}\mathbf{v}^{\mathrm{T}}\mathbf{v}_f + f_a \tag{6.53}$$

If the springForceUserFunction UF is defined, **f** instead becomes (t is current time)

$$f_{SD} = \mathrm{UF}(mbs, t, L - L_0, \Delta^{0}\mathbf{v}^{\mathrm{T}}\mathbf{v}_f, k, d, f_a) \tag{6.54}$$

if `activeConnector = False`, f_{SD} is set to zero.: The vector of the spring force applied at both markers finally reads

$$\mathbf{f} = f_{SD}\mathbf{v}_f \tag{6.55}$$

Userfunction: **`springForceUserFunction(mbs, t, 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

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs to which object belongs
`t`	Real	current time in mbs
`deltaL`	Real	$L - L_0$, spring elongation
`deltaL_t`	Real	$\Delta^0\mathbf{v}^\mathrm{T}\mathbf{v}_f$, spring velocity
`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, 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))
```

6.3.14.4 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()
```

```
SC.TimeIntegrationSolve(mbs, 'GeneralizedAlpha', exu.SimulationSettings())

#check result at default integration time
exudynTestGlobals.testResult = mbs.GetNodeOutput(node, exu.OutputVariableType.
    Position)[0]
```

For further examples on ObjectConnectorSpringDamper see Examples:

- SpringDamperMassUserFunction.py
- ANCF_contact_circle.py
- ANCF_contact_circle2.py
- ANCF_slidingJoint2D.py
- Spring_with_constraints.py
- stiffFlyballGovernor2.py

For further examples on ObjectConnectorSpringDamper see TestModels:

- ANCFcontactCircleTest.py
- modelUnitTests.py
- PARTS_ATEs_moving.py
- stiffFlyballGovernor.py

6.3.15 ObjectConnectorCartesianSpringDamper

An 3D spring-damper element acting accordingly in three (global) directions (x,y,z) which connects to position-based markers.

Additional information for ObjectConnectorCartesianSpringDamper:

- The Object has 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		[MAXINT, MAXINT]	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	PyFunctionVector3DmbsScalar5Vector3D		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 (temorarily) 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

---

6.3.15.1 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	$\mathbf{v}_{\mathrm{off}}$	

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

6.3.15.2 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	${}^0\mathbf{v}_{m1}$	

6.3.15.3 Connector forces

Displacement between marker m0 to marker m1 positions,

$$\Delta{}^0\mathbf{p} = {}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0} \tag{6.56}$$

and relative velocity,

$$\Delta{}^0\mathbf{v} = {}^0\mathbf{v}_{m1} - {}^0\mathbf{v}_{m0} \tag{6.57}$$

If `activeConnector = True`, the spring force vector is computed as

$$\mathbf{f}_{SD} = \left(\mathbf{k} \cdot (\Delta{}^0\mathbf{p} - \mathbf{v}_{\mathrm{off}}) + \mathbf{d}\Delta{}^0\mathbf{v}\right) \tag{6.58}$$

If the springForceUserFunction UF is defined, $\mathbf{f}_{SD}$ instead becomes (t is current time)

$$\mathbf{f}_{SD} = \mathrm{UF}(mbs, t, \Delta{}^0\mathbf{p}, \Delta{}^0\mathbf{v}, \mathbf{k}, \mathbf{d}, \mathbf{v}_{\mathrm{off}}) \tag{6.59}$$

if `activeConnector = False`, $\mathbf{f}_{SD}$ is set to zero.:

Userfunction: **`springForceUserFunction(mbs, t, 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

arguments / return	type or size	description
mbs	MainSystem	provides MainSystem mbs in which underlying item is defined
t	Real	current time in mbs
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, 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))
```

6.3.15.4 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()
SC.TimeIntegrationSolve(mbs, 'GeneralizedAlpha', exu.SimulationSettings())
```

```
#check result at default integration time
exudynTestGlobals.testResult = mbs.GetNodeOutput(nMass, exu.OutputVariableType.
    Displacement)[1]
```

For further examples on ObjectConnectorCartesianSpringDamper see Examples:

- mouseInteractionExample.py
- rigid3Dexample.py
- sliderCrank3DwithANCFbeltDrive2.py
- ANCF_contact_circle.py
- ANCF_contact_circle2.py
- ANCF_slidingJoint2D.py
- cartesianSpringDamper.py
- flexibleRotor3Dtest.py
- lavalRotor2Dtest.py
- NGsolvePistonEngine.py
- ...

For further examples on ObjectConnectorCartesianSpringDamper see TestModels:

- scissorPrismaticRevolute2D.py
- sphericalJointTest.py
- ANCFcontactCircleTest.py
- carRollingDiscTest.py
- driveTrainTest.py
- explicitLieGroupMBSTest.py
- genericODE2test.py
- mecanumWheelRollingDiscTest.py
- modelUnitTests.py
- objectFFRFreducedOrderAccelerations.py
- ...

6.3.16 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:

- The Object has 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		[MAXINT, MAXINT]	list of markers used in connector
nodeNumber	NodeIndex		MAXINT	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
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temorarily) a connector or constraint

springForceTorqueUserFunction	PyFunctionVector6Dmbs4Vector3D2Matrix6D2Matrix3DVector6D		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	PyFunctionVectorMbs4VectorVector3D2Matrix6D2Matrix3DVector6D		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

---

6.3.16.1 DESCRIPTION of ObjectConnectorRigidBodySpringDamper:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
nodeNumber	n_d	

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

6.3.16.2 Definition of quantities

input parameter	symbol	description
stiffness	$\mathbf{k} \in \mathbb{R}^{6\times 6}$	stiffness in $J0$ coordinates
damping	$\mathbf{d} \in \mathbb{R}^{6\times 6}$	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	${}^{b}\boldsymbol{\omega}_{m0}$	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	${}^{0}\Delta\mathbf{p}$	${}^{0}\mathbf{p}_{m1} - {}^{0}\mathbf{p}_{m0}$
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	${}^{J0}\Delta\boldsymbol{\omega}$	$\left({}^{0,m0}\mathbf{A}\;{}^{m0,J0}\mathbf{A}\right)^{\mathrm{T}}\left({}^{0,m1}\mathbf{A}\;{}^{m1}\boldsymbol{\omega} - {}^{0,m0}\mathbf{A}\;{}^{m0}\boldsymbol{\omega}\right)$

6.3.16.3 Connector forces

If `activeConnector = True`, the vector spring force is computed as

$$\begin{bmatrix} {}^{J0}\mathbf{f}_{SD} \\ {}^{J0}\mathbf{m}_{SD} \end{bmatrix} = \mathbf{k}\left(\begin{bmatrix} {}^{J0}\Delta\mathbf{p} \\ {}^{J0}\boldsymbol{\theta} \end{bmatrix} - {}^{J0}\mathbf{v}_{\mathrm{off}}\right) + \mathbf{d}\begin{bmatrix} {}^{J0}\Delta\mathbf{v} \\ {}^{J0}\Delta\omega \end{bmatrix} \qquad (6.60)$$

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)

$$\mathbf{f}_{SD} = \mathrm{UF}(mbs, t, {}^{J0}\Delta\mathbf{p}\,, {}^{J0}\boldsymbol{\theta}, {}^{J0}\Delta\mathbf{v}\,, {}^{J0}\Delta\omega\,, stiffness, damping, rotationMarker0, rotationMarker1, offset) \qquad (6.61)$$

Userfunction: **`springForceTorqueUserFunction(mbs, t, 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).

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
`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, dataCoordinates, displacement, rotation, velocity, angularVelocity, stiffness, damping, rotJ0, rotJ1, offset)`**

A user function which computes the error of the PostNewtonStep ε_{PN}, a suggestion for stepsize reduction t_{red} (currently not used by solvers) 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
`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	$\left[\varepsilon_{PN}, \; t_{red}, \; d_0^k, \; d_1^k, \ldots\right]$ where k indicates the current step

---

User function example:

```
#define simple force for spring-damper:
def UFforce(mbs, t, 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))
```

6.3.16.4 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()
exu.SolveDynamic(mbs, exu.SimulationSettings())

#check result at default integration time
exudynTestGlobals.testResult = mbs.GetNodeOutput(nBody, exu.OutputVariableType.
    Displacement)[1]
```

For further examples on ObjectConnectorRigidBodySpringDamper see Examples:

- stiffFlyballGovernor2.py

For further examples on ObjectConnectorRigidBodySpringDamper see TestModels:

- connectorRigidBodySpringDamperTest.py

- stiffFlyballGovernor.py
- superElementRigidJointTest.py

6.3.17 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:

- The Object has 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 = 'ConnectorCoordinateSpringDamper' has the following parameters:

Name	**type**	**size**	**default value**	**description**
name	String		''	connector's unique name
markerNumbers	ArrayMarkerIndex		[MAXINT, MAXINT]	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
dryFriction	Real		0.	dry friction force [SI:N] against relative velocity; assuming a normal force f_N, the friction force can be interpreted as $f_\mu = \mu f_N$
dryFrictionProportionalZone	Real		0.	limit velocity [m/s] up to which the friction is proportional to velocity (for regularization / avoid numerical oscillations)
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temorarily) a connector or constraint
springForceUserFunction	PyFunctionMbsScalar8		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

6.3.17.1 DESCRIPTION of ObjectConnectorCoordinateSpringDamper:

Information on input parameters:

input parameter	symbol	description see tables above
stiffness	k	
damping	d	
offset	l_{off}	
dryFriction	f_μ	
dryFrictionProportionalZone	v_μ	

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

6.3.17.2 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}	

6.3.17.3 Connector forces

Displacement between marker m0 to marker m1 coordinates (does NOT include reference coordinates),

$$\Delta q = q_{m1} - q_{m0} \tag{6.62}$$

and relative velocity,

$$\Delta v = v_{m1} - v_{m0} \tag{6.63}$$

If $f_\mu > 0$, the friction force is computed as

$$f_{\mathrm{friction}} = \begin{cases} \mathrm{Sgn}(\Delta v) \cdot f_\mu & \text{if} \quad |\Delta v| \geq v_\mu \\ \dfrac{\Delta v}{v_\mu} f_\mu & \text{if} \quad |\Delta v| < v_\mu \end{cases} \tag{6.64}$$

If `activeConnector = True`, the scalar spring force vector is computed as

$$f_{SD} = k\left(\Delta q - l_{\mathrm{off}}\right) + d \cdot \Delta v + f_{\mathrm{friction}} \tag{6.65}$$

If the springForceUserFunction UF is defined, $\mathbf{f}_{SD}$ instead becomes (t is current time)

$$f_{SD} = \mathrm{UF}(mbs, t, \Delta q, \Delta v, k, d, l_{\mathrm{off}}, f_\mu, v_\mu) \tag{6.66}$$

if `activeConnector = False`, f_{SD} is set to zero.:

Userfunction: `springForceUserFunction(mbs, t, 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.

arguments / return	type or size	description
`mbs`	MainSystem	provides MainSystem mbs in which underlying item is defined
`t`	Real	current time in mbs
`displacement`	Real	Δq
`velocity`	Real	Δv
`stiffness`	Real	copied from object
`damping`	Real	copied from object
`offset`	Real	copied from object
`dryFriction`	Real	copied from object
`dryFrictionProportionalZone`	Real	copied from object
`return value`	Real	scalar value of computed force

User function example:

```
#see also mini example!
def UFforce(mbs, t, u, v, k, d, offset, dryFriction, dryFrictionProportionalZone):
    return k*(u-offset) + d*v
```

6.3.17.4 MINI EXAMPLE for ObjectConnectorCoordinateSpringDamper

```
def springForce(mbs, t, u, v, k, d, offset, dryFriction, dryFrictionProportionalZone)
    :
    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()
SC.TimeIntegrationSolve(mbs, 'GeneralizedAlpha', exu.SimulationSettings())

#check result at default integration time
exudynTestGlobals.testResult = mbs.GetNodeOutput(nMass, exu.OutputVariableType.
    Displacement)[0]
```

For further examples on ObjectConnectorCoordinateSpringDamper see Examples:

- slidercrankWithMassSpring.py
- ANCF_switchingSlidingJoint2D.py
- coordinateSpringDamper.py
- geneticOptimizationExample.py
- geneticOptimizationSliderCrank.py
- massSpringFrictionInteractive.py
- mouseInteractionExample.py
- parameterVariationExample.py
- simulateInteractively.py
- sliderCrankCMSacme.py
- ...

For further examples on ObjectConnectorCoordinateSpringDamper see TestModels:

- scissorPrismaticRevolute2D.py
- ACNFslidingAndALEjointTest.py
- ANCFcontactFrictionTest.py
- geneticOptimizationTest.py
- modelUnitTests.py
- objectGenericODE2Test.py
- sliderCrankFloatingTest.py
- springDamperUserFunctionTest.py

6.3.18 ObjectConnectorDistance

Connector which enforces constant or prescribed distance between two bodies/nodes.

Additional information for ObjectConnectorDistance:

- The Object has 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		[MAXINT, MAX-INT]	list of markers used in connector
distance	UReal		0.	prescribed distance [SI:m] of the used markers
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temorarily) 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

---

6.3.18.1 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 in global coordinates

6.3.18.2 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

6.3.18.3 Connector forces constraint equations

If `activeConnector = True`, the index 3 algebraic equation reads

$$\left| {}^0\Delta\mathbf{p} \right| - d_0 = 0 \tag{6.67}$$

The index 2 (velocity level) algebraic equation reads

$$\left(\frac{{}^0\Delta\mathbf{p}}{\left| {}^0\Delta\mathbf{p} \right|} \right)^{\mathrm{T}} \Delta\mathbf{v} = 0 \tag{6.68}$$

if `activeConnector = False`, the algebraic equation reads

$$\lambda_0 = 0 \tag{6.69}$$

6.3.18.4 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
SC.TimeIntegrationSolve(mbs, 'GeneralizedAlpha', sims)

#check result at default integration time
exudynTestGlobals.testResult = mbs.GetNodeOutput(nMass, exu.OutputVariableType.
    Position)[0]
```

For further examples on ObjectConnectorDistance see Examples:

- `pendulum2Dconstraint.py`

For further examples on ObjectConnectorDistance see TestModels:

- `fourBarMechanismTest.py`
- `modelUnitTests.py`
- `PARTS_ATEs_moving.py`

6.3.19 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:

- The Object has 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		[MAXINT, MAXINT]	list of markers used in connector
offset	UReal		0.	An offset between the two values
factorValue1	UReal		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	PyFunctionMbsScalar2		0	A python function which defines the time-dependent offset; see description below
offsetUserFunction_t	PyFunctionMbsScalar2		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 (temorarily) 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

6.3.19.1 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	$\text{UF}(mbs, t, l_{\text{off}})$	
offsetUserFunction_t	$\text{UF}_t(mbs, t, l_{\text{off}})$	

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)

6.3.19.2 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	$\Delta v = v_{m1} - v_{m0}$	

6.3.19.3 Connector constraint equations

If `activeConnector = True`, the index 3 algebraic equation reads

$$\mathbf{c}(q_{m0}, q_{m1}) = k_{m1} \cdot q_{m1} - q_{m0} - l_{\text{off}} = 0 \tag{6.70}$$

If the offsetUserFunction UF is defined, **c** instead becomes (t is current time)

$$\mathbf{c}(q_{m0}, q_{m1}) = k_{m1} \cdot q_{m1} - q_{m0} - \text{UF}(mbs, t, l_{\text{off}}) = 0 \tag{6.71}$$

The `activeConnector = True`, index 2 (velocity level) algebraic equation reads

$$\dot{\mathbf{c}}(\dot{q}_{m0}, \dot{q}_{m1}) = k_{m1} \cdot \dot{q}_{m1} - \dot{q}_{m0} - d = 0 \tag{6.72}$$

The factor d in velocity level equations is zero, except if parameters.velocityLevel = True, then $d = l_\mathrm{off}$. 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)

$$\dot{\mathbf{c}}(\dot{q}_{m0}, \dot{q}_{m1}) = k_{m1} \cdot \dot{q}_{m1} - \dot{q}_{m0} - \mathrm{UF}_t(mbs, t, l_\mathrm{off}) = 0 \tag{6.73}$$

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:

$$\mathbf{c}(\lambda_0) = \lambda_0 = 0 \tag{6.74}$$

Userfunction: `offsetUserFunction(mbs, t, 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`.

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
`lOffset`	Real	l_off
`return value`	Real	computed offset for given time

Userfunction: `offsetUserFunction_t(mbs, t, 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.

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
`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, lOffset):
    return 0.5*lOffset*(1-cos(0.5*pi*t))

def UFoffset_t(mbs, t, 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))
```

6.3.19.4 MINI EXAMPLE for ObjectConnectorCoordinate

```
def OffsetUF(mbs, t, 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()
SC.TimeIntegrationSolve(mbs, 'GeneralizedAlpha', exu.SimulationSettings())

#check result at default integration time
exudynTestGlobals.testResult  = mbs.GetNodeOutput(nMass, exu.OutputVariableType.
    Displacement)[0]
```

For further examples on ObjectConnectorCoordinate see Examples:

- sliderCrank3DwithANCFbeltDrive2.py

- ALEANCF_pipe.py
- ANCF_cantilever_test_dyn.py
- ANCF_contact_circle.py
- ANCF_contact_circle2.py
- ANCF_moving_rigidbody.py
- ANCF_slidingJoint2D.py
- ANCF_switchingSlidingJoint2D.py
- ANCF_tests2.py
- ANCF_test_halfcircle.py
- ...

For further examples on ObjectConnectorCoordinate see TestModels:

- ACNFslidingAndALEjointTest.py
- ANCFcontactCircleTest.py
- ANCFcontactFrictionTest.py
- ANCFmovingRigidBodyTest.py
- driveTrainTest.py
- explicitLieGroupIntegratorPythonTest.py
- explicitLieGroupIntegratorTest.py
- explicitLieGroupMBSTest.py
- fourBarMechanismTest.py
- heavyTop.py
- ...

6.3.20 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:

- The Object has 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		[MAXINT, MAXINT]	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
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
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temorarily) 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

6.3.20.1 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	$\mathbf{X}_{m1} \in \mathbb{R}^{n_{ae} \times n_{q_{m1}}}$	
offset	$\mathbf{v}_\mathrm{off} \in \mathbb{R}^{n_{ae}}$	

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

6.3.20.2 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	$\Delta\mathbf{v} = \dot{\mathbf{q}}_{m1} - \dot{\mathbf{q}}_{m0}$	

6.3.20.3 Remarks

The number of algebraic equations depends on the number of rows in $\mathbf{X}_{m0}$, which must be same as the number of rows in $\mathbf{X}_{m1}$. The number of columns in $\mathbf{X}_{m0}$ must agree with the length of the coordinate vector $\mathbf{q}_{m0}$ and the number of columns in $\mathbf{X}_{m1}$ must agree with the length of the coordinate vector $\mathbf{q}_{m1}$. If one marker k is a ground marker (node/object), the length of $\mathbf{q}_{m,k}$ is zero and also the according matrix $\mathbf{X}_{m,k}$ has zero size and will not be considered in the computation of the constraint equations.

If the number of rows of $\mathbf{X}_{m0}$ plus the number of rows of $\mathbf{X}_{m1}$ is larger than the total number of coordinates ($\mathbf{q}_{m0}$ and $\mathbf{q}_{m1}$), the algebraic equations are underdetermined and probably not solvable.

6.3.20.4 Connector constraint equations

If `activeConnector = True`, the index 3 algebraic equations

$$\mathbf{c}(\mathbf{q}_{m0}, \mathbf{q}_{m1}) = \mathbf{X}_{m1} \cdot \mathbf{q}_{m1} - \mathbf{X}_{m0}\mathbf{q}_{m0} - \mathbf{v}_{\mathrm{off}} = 0 \qquad (6.75)$$

If the offsetUserFunction UF is defined, $\mathbf{c}$ instead becomes (t is current time)

$$\mathbf{c}(\mathbf{q}_{m0}, \mathbf{q}_{m1}) = \mathbf{X}_{m1} \cdot \mathbf{q}_{m1} - \mathbf{X}_{m0}\mathbf{q}_{m0} - \mathrm{UF}(mbs, t, \mathbf{v}_{\mathrm{off}}) = 0 \qquad (6.76)$$

The `activeConnector = True`, index 2 (velocity level) algebraic equation reads

$$\dot{\mathbf{c}}(\dot{\mathbf{q}}_{m0}, \dot{\mathbf{q}}_{m1}) = \mathbf{X}_{m1} \cdot \dot{\mathbf{q}}_{m1} - \mathbf{X}_{m0}\dot{\mathbf{q}}_{m0} - \mathbf{d}_{\mathrm{off}} = 0 \qquad (6.77)$$

The vector dv in velocity level equations is zero, except if parameters.velocityLevel = True, then $\mathbf{d} = \mathbf{v}_{\mathrm{off}}$.

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)

$$\dot{\mathbf{c}}(\dot{\mathbf{q}}_{m0}, \dot{\mathbf{q}}_{m1}) = \mathbf{X}_{m1} \cdot \dot{\mathbf{q}}_{m1} - \mathbf{X}_{m0}\dot{\mathbf{q}}_{m0} - \mathrm{UF}_t(mbs, t, \mathbf{v}_{\mathrm{off}}) = 0 \qquad (6.78)$$

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:

$$\mathbf{c}(\boldsymbol{\lambda}) = \boldsymbol{\lambda} = 0 \qquad (6.79)$$

6.3.21 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 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:

- The Object has 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	[MAXINT, MAXINT]	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
nodeNumber	NodeIndex		MAXINT	node number of a NodeGenericData (size=3) for 3 dataCoordinates
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	Real		0.	normal contact stiffness [SI:N/m]
contactDamping	Real		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)
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!

activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temorarily) a connector or constraint
discRadius	Real		0	defines the disc radius
planeNormal	Vector3D		[0,0,1]	normal to the contact / rolling plane; cannot be changed at the moment
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

---

6.3.21.1 DESCRIPTION of ObjectConnectorRollingDiscPenalty:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
markerNumbers	$[m0, m1]^{\mathrm{T}}$	
nodeNumber	n_d	
dryFrictionAngle	α_t	
contactStiffness	k_c	
contactDamping	d_c	
dryFriction	$[\mu_x, \mu_y]^{\mathrm{T}}$	
dryFrictionProportionalZone	v_μ	
rollingFrictionViscous	μ_r	

The following output variables are available as OutputVariableType in sensors, Get...Output() and other functions:

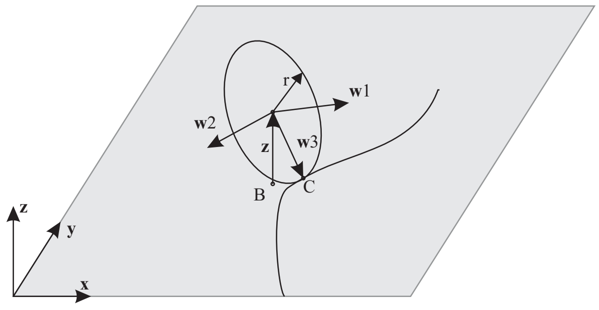
output variable	**symbol**	**description**
Position	${}^{0}\mathbf{p}_{G}$	current global position of contact point between rolling disc and ground
VelocityLocal	${}^{D}\mathbf{v}_{G}$	current velocity of the trail (contact) point in disc coordinates; this is the velocity with which the contact moves over the ground plane
ForceLocal	${}^{J1}\mathbf{f}$	disc-ground force in special marker 1 joint coordinates, f_0 being the lateral force, f_1 being the longitudinal force and f_2 being the normal force

6.3.21.2 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 position; 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	${}^0\mathbf{v}_{PN}$	normalized normal vector to the ground, currently [0,0,1]
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	${}^0\mathbf{w}_1 = {}^{0,m1}\mathbf{A} \cdot [1,0,0]^\mathrm{T}$	normalized disc axis vector, currently $[1,0,0]^\mathrm{T}$ in local coordinates
longitudinal vector	${}^0\mathbf{w}_2$	vector in longitudinal (motion) direction
lateral vector	${}^0\mathbf{w}_l = {}^0\mathbf{v}_{PN} \times {}^0\mathbf{w}_2 = [-\mathbf{w}_{2,y}, \mathbf{w}_{2,x}, 0]$	vector in lateral direction, lies in ground plane
contact point vector	${}^0\mathbf{w}_3$	normalized vector from disc center point in direction of contact point C
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)

6.3.21.3 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,

$$ {}^0\mathbf{x} = {}^0\mathbf{w}_1 \times {}^0\mathbf{v}_{PN} \tag{6.80}$$

we obtain a disc coordinate system, representing the longitudinal direction,

$$ {}^0\mathbf{w}_2 = \frac{1}{|{}^0\mathbf{x}|}\,{}^0\mathbf{x} \tag{6.81}$$

and the vector to the contact point,

$$ {}^0\mathbf{w}_3 = {}^0\mathbf{w}_1 \times {}^0\mathbf{w}_2 \tag{6.82}$$

The contact point can be computed from

$$ {}^0\mathbf{p}_C = {}^0\mathbf{p}_{m1} + r \cdot {}^0\mathbf{w}_3 \tag{6.83}$$

The velocity of the contact point at the disc is computed from,

$$ {}^0\mathbf{v}_C = {}^0\mathbf{v}_{m1} + {}^0\boldsymbol{\omega}_{m1} \times (r \cdot {}^0\mathbf{w}_3) \tag{6.84}$$

The connector forces at the contact point C are computed as follows. The normal contact force reads

$$ f_n = k_c \cdot {}^0\mathbf{p}_{C,z} + d_c \cdot {}^0\mathbf{v}_{C,z} \tag{6.85}$$

The tangential forces are computed from the inplane velocity $\mathbf{v}_t = [{}^0\mathbf{v}_{C,x}, {}^0\mathbf{v}_{C,y}]^\mathrm{T}$

$$ \mathbf{f}_t = \boldsymbol{\mu} \cdot \phi(|\mathbf{v}_t|, v_\mu) \cdot f_n \cdot \mathbf{e}_t \tag{6.86}$$

with the regularization function:

$$ \phi(v, v_\mu) = \begin{cases} (2 - \frac{v}{v_\mu})\frac{v}{v_\mu} & \text{if} \quad v \le v_\mu \\ 1 & \text{if} \quad v > v_\mu \end{cases} \tag{6.87}$$

and the direction of tangential slip

$$ \mathbf{e}_t = \frac{\mathbf{v}_t}{|\mathbf{v}_t|} \tag{6.88}$$

The friction coefficient matrix $\boldsymbol{\mu}$ is computed from

$$ \boldsymbol{\mu} = \begin{bmatrix} \mu_x & 0 \\ 0 & \mu_y \end{bmatrix} \tag{6.89}$$

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

$$ \boldsymbol{\mu} = \begin{bmatrix} \cos(\alpha_t) & \sin(\alpha_t) \\ -\sin(\alpha_t) & \cos(\alpha_t) \end{bmatrix} \begin{bmatrix} \mu_x & 0 \\ 0 & \mu_y \end{bmatrix} \begin{bmatrix} \cos(\alpha_t) & -\sin(\alpha_t) \\ \sin(\alpha_t) & \cos(\alpha_t) \end{bmatrix} \tag{6.90}$$

6.3.21.4 Connector forces

Finally, the connector forces read in joint coordinates

$$ {}^{J1}\mathbf{f} = \begin{bmatrix} f_{t,x} \\ f_{t,y} \\ f_n \end{bmatrix} \tag{6.91}$$

and in global coordinates, they are computed from

$$^{0}\mathbf{f} = f_{t,x}\mathbf{w}_l + f_{t,y}\mathbf{w}_2 + f_n\mathbf{v}_{PN} \qquad (6.92)$$

The moment caused by the contact forces are given as

$$^{0}\mathbf{f} = (r \cdot {}^{0}\mathbf{w}_3) \times {}^{0}\mathbf{f} \qquad (6.93)$$

if `activeConnector = False`,

$$^{J1}\mathbf{f} = \mathbf{0} \qquad (6.94)$$

For further examples on ObjectConnectorRollingDiscPenalty see TestModels:

- `carRollingDiscTest.py`
- `mecanumWheelRollingDiscTest.py`
- `rollingCoinPenaltyTest.py`

6.3.22 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:

- The Object has 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		[MAXINT, MAX-INT]	markers define contact gap
nodeNumber	NodeIndex		MAXINT	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	UReal		0.	offset [SI:m] of contact
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temorarily) 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

6.3.22.1 DESCRIPTION of ObjectContactCoordinate:

For further examples on ObjectContactCoordinate see Examples:

- `ANCF_contact_circle.py`
- `ANCF_contact_circle2.py`

For further examples on ObjectContactCoordinate see TestModels:

- `ANCFcontactCircleTest.py`

6.3.23 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:

- The Object has 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		[MAXINT, MAXINT]	markers define contact gap
nodeNumber	NodeIndex		MAXINT	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	UReal		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 (temorarily) 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
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

---

6.3.23.1 DESCRIPTION of ObjectContactCircleCable2D:

For further examples on ObjectContactCircleCable2D see Examples:

- `ANCF_contact_circle.py`
- `ANCF_contact_circle2.py`
- `ANCF_moving_rigidbody.py`
- `ANCF_slidingJoint2D.py`

For further examples on ObjectContactCircleCable2D see TestModels:

- `ACNFslidingAndALEjointTest.py`
- `ANCFcontactCircleTest.py`
- `ANCFmovingRigidBodyTest.py`

6.3.24 ObjectContactFrictionCircleCable2D

A very specialized penalty-based contact/friction condition between a 2D circle in the local x/y plane (=marker0, a Rigid-Body Marker) 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=1), last friction position]; 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 ObjectContactFrictionCircleCable2D:

- The Object has 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		[MAXINT, MAXINT]	markers define contact gap
nodeNumber	NodeIndex		MAXINT	node number of a NodeGenericData with 3 × 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
frictionVelocityPenalty	UReal		0.	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.	CURRENTLY NOT IMPLEMENTED: 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 μ [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
offset	UReal		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 (temorarily) 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
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

---

6.3.24.1 DESCRIPTION of ObjectContactFrictionCircleCable2D:

For further examples on ObjectContactFrictionCircleCable2D see Examples:

- `sliderCrank3DwithANCFbeltDrive.py`
- `sliderCrank3DwithANCFbeltDrive2.py`

For further examples on ObjectContactFrictionCircleCable2D see TestModels:

- `ACNFslidingAndALEjointTest.py`
- `ANCFcontactFrictionTest.py`
- `ANCFmovingRigidBodyTest.py`

6.3.25 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 can be used to define three rotation axes and/or sliding axes

Additional information for ObjectJointGeneric:

- The Object has 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	[MAXINT, MAXINT]	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 (temorarily) a connector or constraint
offsetUserFunctionParameters	Vector6D		[0.,0.,0.,0.,0.,0.]	vector of 6 parameters for joint's offsetUserFunction
offsetUserFunction	PyFunctionVector6DmbsScalarVector6D		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	PyFunctionVector6DmbsScalarVector6D		0	(NOT IMPLEMENTED YET)time derivative of offsetUserFunction using the same parameters
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

---

6.3.25.1 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	$UF(mbs, t, \mathbf{p}_{par})$	
offsetUserFunction_t	$UF_t(mbs, t, \mathbf{p}_{par})$	

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

6.3.25.2 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	${}^{0,J0}\mathbf{A} = [\mathbf{v}_{x0},\ \mathbf{v}_{y0},\ \mathbf{v}_{z0}]^{\mathrm{T}}$	orientation vectors 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	${}^{0,J1}\mathbf{A} = [\mathbf{v}_{x1},\ \mathbf{v}_{y1},\ \mathbf{v}_{z1}]^{\mathrm{T}}$	orientation vectors 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

6.3.25.3 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),

$$ {}^{0}\mathbf{p}_{m1} - {}^{0}\mathbf{p}_{m0} - UF_{0,1,2}(mbs,t,\mathbf{p}_{par}) = \mathbf{0} \qquad (6.95)$$

and the translational index 2 constraints read

$$ {}^{0}\mathbf{v}_{m1} - {}^{0}\mathbf{v}_{m0} - UF_{t;0,1,2}(mbs,t,\mathbf{p}_{par}) = \mathbf{0} \qquad (6.96)$$

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 ${}^{J0}\Delta\mathbf{p}$

$$\begin{aligned} {}^{J0}\Delta p_k - UF_k(mbs, t, \mathbf{p}_{par}) &= 0 \quad \text{if} \quad j_k = 1 \quad \text{and} & (6.97)\\ \lambda_k &= 0 \quad \text{if} \quad j_k = 0 & (6.98)\\ & & (6.99) \end{aligned}$$

and the translational index 2 constraints read for every component $k \in [0,1,2]$ of the vector ${}^{J0}\Delta\mathbf{v}$

$$\begin{aligned} {}^{J0}\Delta v_k - UF\_t_k(mbs, t, \mathbf{p}_{par}) &= 0 \quad \text{if} \quad j_k = 1 \quad \text{and} & (6.100)\\ \lambda_k &= 0 \quad \text{if} \quad j_k = 0 & (6.101)\\ & & (6.102) \end{aligned}$$

Equations for rotational part (`activeConnector = True`) :

The following equations are exemplarily for certain constrained rotation axes configurations, which shall represent all other possibilities. 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**, $[j_3, \ldots, j_5] = [1,1,1]^\mathrm{T}$, the index 3 constraint equations read

$$\begin{aligned} \mathbf{v}_{z0}^\mathrm{T}\mathbf{v}_{y1} &= 0 & (6.103)\\ \mathbf{v}_{z0}^\mathrm{T}\mathbf{v}_{x1} &= 0 & (6.104)\\ \mathbf{v}_{x0}^\mathrm{T}\mathbf{v}_{y1} &= 0 & (6.105) \end{aligned}$$

If **2 rotation axes are constrained**, e.g., $[j_3, \ldots, j_5] = [0,1,1]^\mathrm{T}$, the index 3 constraint equations read

$$\begin{aligned} \lambda_3 &= 0 & (6.106)\\ \mathbf{v}_{x0}^\mathrm{T}\mathbf{v}_{y1} &= 0 & (6.107)\\ \mathbf{v}_{x0}^\mathrm{T}\mathbf{v}_{z1} &= 0 & (6.108) \end{aligned}$$

If **1 rotation axis is constrained**, e.g., $[j_3, \ldots, j_5] = [1,0,0]^\mathrm{T}$, the index 3 constraint equations read

$$\begin{aligned} \mathbf{v}_{y0}^\mathrm{T}\mathbf{v}_{z1} &= 0 & (6.109)\\ \lambda_4 &= 0 & (6.110)\\ \lambda_5 &= 0 & (6.111) \end{aligned}$$

if `activeConnector = False`,

$$\mathbf{z} = \mathbf{0} \qquad (6.112)$$

Userfunction: `offsetUserFunction(mbs, t, 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`.

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
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, 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.

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
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, offsetUserFunctionParameters):
    return [offsetUserFunctionParameters[0]*(1 - cos(t*10*2*pi)), 0,0,0,0,0]
```

For further examples on ObjectJointGeneric see Examples:

- mouseInteractionExample.py
- rigidBodyTutorial.py
- sliderCrank3DwithANCFbeltDrive.py
- stiffFlyballGovernor2.py

For further examples on ObjectJointGeneric see TestModels:

- driveTrainTest.py
- carRollingDiscTest.py
- genericJointUserFunctionTest.py
- mecanumWheelRollingDiscTest.py
- objectFFRFreducedOrderAccelerations.py

- objectFFRFreducedOrderStressModesTest.py
- objectFFRFreducedOrderTest.py
- objectFFRFTest.py
- rigidBodyCOMtest.py
- sliderCrank3Dbenchmark.py
- ...

6.3.26 ObjectJointSpherical

A spherical joint, which constrains the relative translation between two position based markers.

Additional information for ObjectJointSpherical:

- The Object has 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	[MAXINT, MAX-INT]	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 (temorarily) 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

---

6.3.26.1 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

6.3.26.2 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

6.3.26.3 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

$$ {}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0} = \mathbf{0} \qquad (6.113)$$

and the translational index 2 constraints read

$$ {}^0\mathbf{v}_{m1} - {}^0\mathbf{v}_{m0} = \mathbf{0} \qquad (6.114)$$

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 ${}^0\Delta\mathbf{p}$

$$\begin{aligned} {}^0\Delta p_k &= 0 \quad \text{if} \quad j_k = 1 \quad \text{and} && (6.115)\\ \lambda_k &= 0 \quad \text{if} \quad j_k = 0 && (6.116)\\ & && (6.117)\end{aligned}$$

and the translational index 2 constraints read for every component $k \in [0,1,2]$ of the vector ${}^0\Delta\mathbf{v}$

$$\begin{aligned} {}^0\Delta v_k &= 0 \quad \text{if} \quad j_k = 1 \quad \text{and} && (6.118)\\ \lambda_k &= 0 \quad \text{if} \quad j_k = 0 && (6.119)\\ & && (6.120)\end{aligned}$$

`activeConnector = False`:

$$\mathbf{z} = \mathbf{0} \tag{6.121}$$

For further examples on ObjectJointSpherical see Examples:

- `objectFFRFreducedOrderNetgen.py`
- `sliderCrank3DwithANCFbeltDrive.py`
- `sliderCrankCMSacme.py`

For further examples on ObjectJointSpherical see TestModels:

- `driveTrainTest.py`
- `genericJointUserFunctionTest.py`
- `objectFFRFreducedOrderAccelerations.py`
- `objectFFRFreducedOrderStressModesTest.py`
- `objectFFRFreducedOrderTest.py`
- `objectFFRFTest.py`
- `objectFFRFTest2.py`
- `sphericalJointTest.py`
- `superElementRigidJointTest.py`

6.3.27 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 case of a moving ground body needs to be tested further, check your results!

Additional information for ObjectJointRollingDisc:

- The Object has 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	[MAXINT, MAXINT]	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]	flag, which determines which constraints are active, in which j_0, j_1 represent the tangential motion and j_2 represents the normal (contact) direction
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temorarily) a connector or constraint
discRadius	Real		0	defines the disc radius
planeNormal	Vector3D		[0,0,1]	normal to the contact / rolling plane; cannot be changed at the moment
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

6.3.27.1 DESCRIPTION of ObjectJointRollingDisc:

Information on input parameters:

input parameter	symbol	description see tables above
markerNumbers	$[m0, m1]^\mathrm{T}$	
constrainedAxes	$\mathbf{j} = [j_0, \ldots, j_2]$	

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
VelocityLocal	${}^D\mathbf{v}_G$	current velocity of the trail (contact) point in disc coordinates; this is the velocity with which the contact moves over the ground plane
ForceLocal	${}^D\mathbf{f} = [-\mathbf{z}^T\mathbf{w}_l,\ -\mathbf{z}^T\mathbf{w}_2,\ -\mathbf{z}_z]^\mathrm{T}$	contact forces acting on disc, in special disc coordinates, f_x being the lateral force, f_y being the longitudinal force and f_z being the normal force

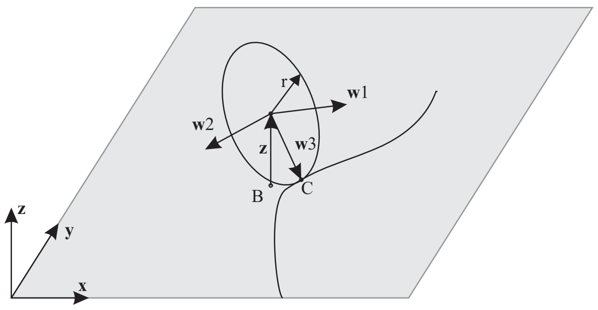
6.3.27.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^0\mathbf{p}_{m0}$	current global position of marker $m0$ body reference position (ground body reference position); 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	${}^0\mathbf{v}_{PN}$	normalized normal vector to the ground plane, currently [0,0,1]
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	${}^0\mathbf{w}_1 = {}^{0,m1}\mathbf{A} \cdot [1,0,0]^\mathrm{T}$	normalized disc axis vector, currently $[1,0,0]^\mathrm{T}$ in local coordinates
longitudinal vector	${}^0\mathbf{w}_2$	vector in longitudinal (motion) direction
lateral vector	${}^0\mathbf{w}_l = {}^0\mathbf{v}_{PN} \times {}^0\mathbf{w}_2 = [-\mathbf{w}_{2,y}, \mathbf{w}_{2,x}, 0]$	vector in lateral direction, lies in ground plane
contact point vector	${}^0\mathbf{w}_3$	normalized vector from disc center point in direction of contact point C
algebraic variables	$\mathbf{z} = [\lambda_0,\ \lambda_1,\ \lambda_2]^\mathrm{T}$	vector of algebraic variables (Lagrange multipliers) according to the algebraic equations

6.3.27.3 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,

$$ {}^0\mathbf{x} = {}^0\mathbf{w}_1 \times {}^0\mathbf{v}_{PN} \tag{6.122}$$

we obtain a disc coordinate system, representing the longitudinal direction,

$$ {}^0\mathbf{w}_2 = \frac{1}{|{}^0\mathbf{x}|}\, {}^0\mathbf{x} \tag{6.123}$$

and the vector to the contact point,

$$ {}^0\mathbf{w}_3 = {}^0\mathbf{w}_1 \times {}^0\mathbf{w}_2 \tag{6.124}$$

The contact point C can be computed from

$$ {}^0\mathbf{p}_C = {}^0\mathbf{p}_{m1} + r \cdot {}^0\mathbf{w}_3 \tag{6.125}$$

The velocity of the contact point at the disc is computed from,

$$ {}^0\mathbf{v}_{Cm1} = {}^0\mathbf{v}_{m1} + {}^0\boldsymbol{\omega}_{m1} \times (r \cdot {}^0\mathbf{w}_3) \tag{6.126}$$

If marker 0 body is (moving) rigid body instead of a ground body, the contact point C is reconstructed in body of marker 0,

$$ {}^{m0}\mathbf{p}_C = {}^{m0,0}\mathbf{A}\,({}^0\mathbf{p}_C - {}^0\mathbf{p}_{m0}) \tag{6.127}$$

The velocity of the contact point at the marker 0 body reads

$$ {}^0\mathbf{v}_{Cm0} = {}^0\mathbf{v}_{m0} + {}^0\boldsymbol{\omega}_{m0} \times \left({}^{0,m0}\mathbf{A}\ {}^{m0}\mathbf{p}_C\right) \tag{6.128}$$

6.3.27.4 Connector constraint equations

`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)

$$\begin{bmatrix} {}^0\mathbf{v}_{Cm1,x} \\ {}^0\mathbf{v}_{Cm1,y} \\ {}^0\mathbf{v}_{Cm1,z} \end{bmatrix} - \begin{bmatrix} {}^0\mathbf{v}_{Cm0,x} \\ {}^0\mathbf{v}_{Cm0,y} \\ {}^0\mathbf{v}_{Cm0,z} \end{bmatrix} = \mathbf{0} \qquad (6.129)$$

`activeConnector = False`:

$$\mathbf{z} = \mathbf{0} \qquad (6.130)$$

For further examples on ObjectJointRollingDisc see TestModels:

- `rollingCoinTest.py`

6.3.28 ObjectJointRevolute2D

A revolute joint in 2D; constrains the absolute 2D position of two points given by PointMarkers or RigidMarkers

Additional information for ObjectJointRevolute2D:

- The Object has 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		[MAXINT, MAXINT]	list of markers used in connector
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temorarily) 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

---

6.3.28.1 DESCRIPTION of ObjectJointRevolute2D:

For further examples on ObjectJointRevolute2D see Examples:

- `sliderCrank3DwithANCFbeltDrive2.py`
- `ANCF_slidingJoint2D.py`
- `geneticOptimizationSliderCrank.py`
- `rigid_pendulum.py`
- `SliderCrank.py`
- `sliderCrank3DwithANCFbeltDrive.py`
- `slidercrankWithMassSpring.py`

- switchingConstraintsPendulum.py

For further examples on ObjectJointRevolute2D see TestModels:

- compareFullModifiedNewton.py
- modelUnitTests.py
- PARTS_ATEs_moving.py
- pendulumFriction.py
- scissorPrismaticRevolute2D.py
- sliderCrankFloatingTest.py

6.3.29 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

$$(\mathbf{p}_1 - \mathbf{p}_0)^T \cdot \mathbf{A}_1 \cdot \mathbf{n}_1 = 0 \tag{6.131}$$

$$(\mathbf{A}_0 \cdot \mathbf{t}_0)^T \cdot \mathbf{A}_1 \cdot \mathbf{n}_1 = 0 \tag{6.132}$$

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:

- The Object has 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		[MAXINT, MAXINT]	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 (temorarily) 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

6.3.29.1 DESCRIPTION of ObjectJointPrismatic2D:

For further examples on ObjectJointPrismatic2D see Examples:

- sliderCrank3DwithANCFbeltDrive2.py
- geneticOptimizationSliderCrank.py

For further examples on ObjectJointPrismatic2D see TestModels:

- PARTS_ATEs_moving.py
- scissorPrismaticRevolute2D.py
- sliderCrankFloatingTest.py

6.3.30 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:

- The Object has 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		[MAXINT, MAXINT]	marker m0: position-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		MAXINT	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	uses a formulation with 3 equations, including the force in sliding direction to be zero; forces in global coordinates, only index 3; alternatively: use local formulation, which only needs two equations and can be used with index 2 formulation
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temorarily) 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

---

6.3.30.1 DESCRIPTION of ObjectJointSliding2D:

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	$[d_{s0}, \ldots, d_{sn}]$	
nodeNumber	n_{GD}	

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)

6.3.30.2 Definition of quantities

intermediate variables	symbol	description
data node	$\mathbf{x} = [x_{data0},\ x_{data1}]^{\mathrm{T}}$	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
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	${}^0\mathbf{r}'_{ANCF} = \mathbf{S}'(s_{el})\mathbf{q}_{ANCF,m1} = [r'_0, r'_1]^\mathrm{T}$	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	$\mathbf{z} = [\lambda_0, \lambda_1, s]^\mathrm{T}$	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

6.3.30.3 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

$$s_{el} = s + x_{data1} - d_{m1} = s_g - d_{m1}. \tag{6.133}$$

The vector (=difference; error) between the marker $m0$ and the marker $m1$ (=$\mathbf{r}_{ANCF}$) positions reads

$${}^0\Delta\mathbf{p} = {}^0\mathbf{r}_{ANCF} - {}^0\mathbf{p}_{m0} \tag{6.134}$$

The vector (=difference; error) between the marker $m0$ and the marker $m1$ velocities reads

$$^0\Delta \mathbf{v} = {^0}\dot{\mathbf{r}}_{ANCF} - {^0}\mathbf{v}_{m0} \quad (6.135)$$

6.3.30.4 Connector constraint equations (classicalFormulation=True)

The 2D sliding joint is implemented having 3 equations, using the special algebraic coordinates $\mathbf{z}$. The algebraic equations read

$$\begin{aligned} {^0}\Delta\mathbf{p} &= \mathbf{0}, && \text{... 2 index 3 equations, ensuring the sliding body to stay at the cable} && (6.136)\\ [\lambda_0, \lambda_1] \cdot {^0}\mathbf{r}'_{ANCF} &= 0, && \text{... 1 index 1 equation, ensuring the force in sliding direction = 0} && (6.137)\\ &&&&& (6.138) \end{aligned}$$

No index 2 case exists, because no time derivative exists for s_{el}. The jacobian matrices for algebraic and ODE2 coordinates read

$$J_{AE} = \begin{bmatrix} 0 & 0 & r'_0 \\ 0 & 0 & r'_1 \\ r'_0 & r'_1 & r''_0\lambda_0 + r''_1\lambda_1 \end{bmatrix} \quad (6.139)$$

$$J_{ODE2} = \begin{bmatrix} -J_{pos,m0} & \mathbf{S}(s_{el}) \\ \mathbf{0}^\mathrm{T} & [\lambda_0,\, \lambda_1] \cdot \mathbf{S}'(s_{el}) \end{bmatrix} \quad (6.140)$$

if `activeConnector = False`, the algebraic equations are changed to:

$$\begin{aligned} \lambda_0 &= 0, && (6.141)\\ \lambda_1 &= 0, && (6.142)\\ s &= 0 && (6.143) \end{aligned}$$

6.3.30.5 Connector constraint equations (classicalFormulation=False)

The 2D sliding joint is implemented having 3 equations (first equation is dummy and could be eliminated), using the special algebraic coordinates $\mathbf{z}$. The algebraic equations read

$$\begin{aligned} \lambda_0 &= 0, && \text{... this equation is not necessary, but can be used for switching to other modes} && (6.144)\\ {^0}\Delta\mathbf{p}^\mathrm{T}\, {^0}\mathbf{n} &= 0, && \text{... equation ensures that sliding body stays at cable centerline; index3 equation} && (6.145)\\ {^0}\Delta\mathbf{p}^\mathrm{T}\, {^0}\mathbf{r}'_{ANCF} &= 0. && \text{... resolves the sliding coordinate } s\text{; index1 equation!} && (6.146) \end{aligned}$$

In the index 2 case, the second equation reads

$$^0\Delta\mathbf{v}^\mathrm{T}\, {^0}\mathbf{n} + {^0}\Delta\mathbf{p}^\mathrm{T}\, {^0}\dot{\mathbf{n}} = 0 \quad (6.147)$$

if `activeConnector = False`, the algebraic equations are changed to:

$$\begin{aligned} \lambda_0 &= 0, && (6.148)\\ \lambda_1 &= 0, && (6.149)\\ s &= 0 && (6.150) \end{aligned}$$

6.3.30.6 Post Newton Step

After the Newton solver has converged, a PostNewtonStep is performed for the element, which updates the marker $m1$ index if necessary.

$$\begin{aligned} s_{el} < 0 \quad &\rightarrow \quad x_{data0} \mathrel{-}= 1 \\ s_{el} > L \quad &\rightarrow \quad x_{data0} \mathrel{+}= 1 \end{aligned} \tag{6.151}$$

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,

$$x_{data1} \mathrel{+}= s. \tag{6.152}$$

For further examples on ObjectJointSliding2D see Examples:

- `ANCF_moving_rigidbody.py`
- `ANCF_slidingJoint2D.py`
- `ANCF_switchingSlidingJoint2D.py`

For further examples on ObjectJointSliding2D see TestModels:

- `modelUnitTests.py`

6.3.31 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:

- The Object has 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		[MAXINT, MAXINT]	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		[MAXINT, MAXINT]	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 (temorarily) 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

---

6.3.31.1 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	$[x_{data0},\ q_{ALE}]^{\mathrm{T}}$	provides two values: [0] = current sliding marker index, [1] = ALE sliding coordinate
Coordinates_t	$[\dot q_{ALE}]^{\mathrm{T}}$	provides ALE sliding velocity
Force	$\mathbf{f}$	joint force vector (3D)

6.3.31.2 Definition of quantities

intermediate variables	symbol	description
generic data node	$\mathbf{x} = [x_{data0}]^{\mathrm{T}}$	coordinates of node with node number n_{GD}
generic ODE2 node	$\mathbf{q} = [q_0]^{\mathrm{T}}$	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	$\mathbf{z} = [\lambda_0, \lambda_1]^\mathrm{T}$	algebraic variables (Lagrange multipliers) according to the algebraic equations

6.3.31.3 Geometric relations

The element sliding coordinate (in the local coordinates of the current sliding element) is computed from the ALE coordinate

$$s_{el} = q_{ALE} + s_{off} - d_{m1} = s_g - d_{m1}. \tag{6.153}$$

The vector (=difference; error) between the marker $m0$ and the marker $m1$ (=$\mathbf{r}_{ANCF}$) positions reads

$${}^0\Delta\mathbf{p} = {}^0\mathbf{r}_{ANCF} - {}^0\mathbf{p}_{m0} \tag{6.154}$$

The vector (=difference; error) between the marker $m0$ and the marker $m1$ velocities reads

$${}^0\Delta\mathbf{v} = {}^0\mathbf{v}_{ANCF} - {}^0\mathbf{v}_{m0} \tag{6.155}$$

6.3.31.4 Connector constraint equations

The 2D sliding joint is implemented having 2 equations, using the Lagrange multipliers $\mathbf{z}$. The algebraic (index 3) equations read

$${}^0\Delta\mathbf{p} = 0 \tag{6.156}$$

Note that the Lagrange multipliers $[\lambda_0, \lambda_1]^\mathrm{T}$are the global forces in the joint. In the index 2 case the algebraic equations read

$${}^0\Delta\mathbf{v} = 0 \tag{6.157}$$

If `usePenalty = True`, the algebraic equations are changed to:

$$^{0}\Delta\mathbf{p} - \frac{1}{k}\mathbf{z} = 0. \tag{6.158}$$

If `activeConnector = False`, the algebraic equations are changed to:

$$\lambda_0 = 0, \tag{6.159}$$
$$\lambda_1 = 0. \tag{6.160}$$

6.3.31.5 Post Newton Step

After the Newton solver has converged, a PostNewtonStep is performed for the element, which updates the marker $m1$ index if necessary.

$$\begin{aligned} s_{el} < 0 \quad &\rightarrow \quad x_{data0} \;-\!= 1 \\ s_{el} > L \quad &\rightarrow \quad x_{data0} \;+\!= 1 \end{aligned} \tag{6.161}$$

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,

$$x_{data1} \mathrel{+}= s. \tag{6.162}$$

For further examples on ObjectJointALEMoving2D see Examples:

- `ANCF_moving_rigidbody.py`

For further examples on ObjectJointALEMoving2D see TestModels:

- `ANCFmovingRigidBodyTest.py`

6.4 Markers

6.4.1 MarkerBodyMass

A marker attached to the body mass; use this marker to apply a body-load (e.g. gravitational force).

Additional information for MarkerBodyMass:

- The Marker has 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		MAXINT	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

---

6.4.1.1 DESCRIPTION of MarkerBodyMass:

For further examples on MarkerBodyMass see Examples:

- `ALEANCF_pipe.py`
- `ANCF_moving_rigidbody.py`
- `ANCF_slidingJoint2D.py`
- `ANCF_switchingSlidingJoint2D.py`
- `rigid3Dexample.py`
- `rigidBodyIMUtest.py`

For further examples on MarkerBodyMass see TestModels:

- `genericJointUserFunctionTest.py`
- `modelUnitTests.py`
- `sphericalJointTest.py`

6.4.2 MarkerBodyPosition

A position body-marker attached to local position (x,y,z) of the body.

Additional information for MarkerBodyPosition:

- The Marker has 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		MAXINT	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

6.4.2.1 DESCRIPTION of MarkerBodyPosition:

For further examples on MarkerBodyPosition see Examples:

- `ANCF_contact_circle.py`
- `ANCF_contact_circle2.py`
- `ANCF_moving_rigidbody.py`
- `ANCF_slidingJoint2D.py`
- `ANCF_switchingSlidingJoint2D.py`
- `cartesianSpringDamper.py`
- `coordinateSpringDamper.py`
- `flexibleRotor3Dtest.py`
- `geneticOptimizationSliderCrank.py`
- `lavalRotor2Dtest.py`
- ...

For further examples on MarkerBodyPosition see TestModels:

- `ACNFslidingAndALEjointTest.py`
- `ANCFcontactCircleTest.py`

- ANCFcontactFrictionTest.py
- ANCFmovingRigidBodyTest.py
- compareFullModifiedNewton.py
- explicitLieGroupIntegratorPythonTest.py
- explicitLieGroupIntegratorTest.py
- explicitLieGroupMBSTest.py
- genericODE2test.py
- heavyTop.py
- ...

6.4.3 MarkerBodyRigid

A rigid-body (position+orientation) body-marker attached to local position (x,y,z) of the body.

Additional information for MarkerBodyRigid:

- The Marker has 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		MAXINT	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

---

6.4.3.1 DESCRIPTION of MarkerBodyRigid:

For further examples on MarkerBodyRigid see Examples:

- `ANCF_contact_circle.py`
- `ANCF_contact_circle2.py`
- `ANCF_slidingJoint2D.py`
- `ANCF_tests2.py`
- `ANCF_test_halfcircle.py`
- `flexibleRotor3Dtest.py`
- `geneticOptimizationSliderCrank.py`
- `mouseInteractionExample.py`
- `rigid3Dexample.py`
- `rigidBodyIMUtest.py`
- ...

For further examples on MarkerBodyRigid see TestModels:

- `ACNFslidingAndALEjointTest.py`
- `ANCFcontactCircleTest.py`

- ANCFcontactFrictionTest.py
- ANCFmovingRigidBodyTest.py
- carRollingDiscTest.py
- connectorRigidBodySpringDamperTest.py
- driveTrainTest.py
- genericJointUserFunctionTest.py
- manualExplicitIntegrator.py
- mecanumWheelRollingDiscTest.py
- ...

6.4.4 MarkerNodePosition

A node-Marker attached to a position-based node.

Additional information for MarkerNodePosition:

- The Marker has 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		MAXINT	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

---

6.4.4.1 DESCRIPTION of MarkerNodePosition:

For further examples on MarkerNodePosition see Examples:

- `ALEANCF_pipe.py`
- `ANCF_cantilever_test.py`
- `ANCF_cantilever_test_dyn.py`
- `ANCF_contact_circle.py`
- `ANCF_contact_circle2.py`
- `ANCF_slidingJoint2D.py`
- `ANCF_tests2.py`
- `ANCF_test_halfcircle.py`
- `flexibleRotor3Dtest.py`
- `geneticOptimizationSliderCrank.py`
- ...

For further examples on MarkerNodePosition see TestModels:

- `ANCFcontactCircleTest.py`
- `ANCFcontactFrictionTest.py`
- `fourBarMechanismTest.py`
- `manualExplicitIntegrator.py`
- `modelUnitTests.py`

- objectFFRFTest.py
- pendulumFriction.py
- sliderCrankFloatingTest.py
- sphericalJointTest.py

6.4.5 MarkerNodeRigid

A rigid-body (position+orientation) node-marker attached to a rigid-body node.

Additional information for MarkerNodeRigid:

- The Marker has 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		MAXINT	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

---

6.4.5.1 DESCRIPTION of MarkerNodeRigid:

For further examples on MarkerNodeRigid see Examples:

- `ANCF_contact_circle.py`
- `ANCF_contact_circle2.py`
- `ANCF_slidingJoint2D.py`
- `ANCF_tests2.py`
- `ANCF_test_halfcircle.py`
- `objectFFRFreducedOrderNetgen.py`
- `sliderCrankCMSacme.py`
- `solverFunctionsTestEigenvalues.py`

For further examples on MarkerNodeRigid see TestModels:

- `ANCFcontactCircleTest.py`
- `connectorRigidBodySpringDamperTest.py`
- `manualExplicitIntegrator.py`
- `objectFFRFreducedOrderAccelerations.py`
- `objectFFRFreducedOrderStressModesTest.py`
- `objectFFRFreducedOrderTest.py`
- `objectFFRFTest.py`
- `objectFFRFTest2.py`
- `superElementRigidJointTest.py`

6.4.6 MarkerNodeCoordinate

A node-Marker attached to a ODE2 coordinate of a node; for other coordinates (ODE1,...) other markers need to be defined.

Additional information for MarkerNodeCoordinate:

- The Marker has 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		MAXINT	node number to which marker is attached to
coordinate	Index		MAXINT	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

---

6.4.6.1 DESCRIPTION of MarkerNodeCoordinate:

For further examples on MarkerNodeCoordinate see Examples:

- `ALEANCF_pipe.py`
- `ANCF_cantilever_test.py`
- `ANCF_cantilever_test_dyn.py`
- `ANCF_contact_circle.py`
- `ANCF_contact_circle2.py`
- `ANCF_moving_rigidbody.py`
- `ANCF_slidingJoint2D.py`
- `ANCF_switchingSlidingJoint2D.py`
- `ANCF_tests2.py`
- `ANCF_test_halfcircle.py`
- ...

For further examples on MarkerNodeCoordinate see TestModels:

- `ACNFslidingAndALEjointTest.py`
- `ANCFcontactCircleTest.py`

- ANCFcontactFrictionTest.py
- ANCFmovingRigidBodyTest.py
- computeODE2EigenvaluesTest.py
- driveTrainTest.py
- explicitLieGroupIntegratorPythonTest.py
- explicitLieGroupIntegratorTest.py
- explicitLieGroupMBSTest.py
- fourBarMechanismTest.py
- ...

6.4.7 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:

- The Marker has the following types = `Node`, `Orientation`, `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		MAXINT	node number to which marker is attached to
rotationCoordinate	Index		MAXINT	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

---

6.4.7.1 DESCRIPTION of MarkerNodeRotationCoordinate:

For further examples on MarkerNodeRotationCoordinate see Examples:

- `rigidRotor3DbasicBehaviour.py`
- `sliderCrank3DwithANCFbeltDrive2.py`

For further examples on MarkerNodeRotationCoordinate see TestModels:

- `driveTrainTest.py`

6.4.8 MarkerSuperElementPosition

A position marker attached to a SuperElement, such as ObjectFFRF, ObjectGenericODE2 and ObjectFFRFreducedOrder (for which it is 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:

- The Marker has 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		MAXINT	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
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)

6.4.8.1 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	${}^0\mathbf{p}_{\,m} = \sum_{i=0}^{n-1} w_i \cdot {}^0\mathbf{p}_i$	current global position which is provided by marker
marker velocity	${}^0\mathbf{v}_{\,m} = \sum_{i=0}^{n-1} w_i \cdot {}^0\mathbf{v}_i$	current global velocity which is provided by marker

6.4.8.2 Marker quantities

The marker provides a 'position' jacobian, which is the derivative of the marker velocity w.r.t. the object velocity coordinates $\dot{\mathbf{q}}_{n_b}$,

$$\mathbf{J}_{m,pos} = \frac{\partial\, {}^0\mathbf{v}_{\,m}}{\partial \dot{\mathbf{q}}_{n_b}} = \sum_{i=0}^{n-1} w_i \cdot \mathbf{J}_{i,pos} \tag{6.163}$$

in which $\mathbf{J}_{i,pos}$ denotes the position jacobian of mesh node i,

$$\mathbf{J}_{i,pos} = \frac{\partial\, {}^0\mathbf{v}_{\,i}}{\partial \dot{\mathbf{q}}_{n_b}} \tag{6.164}$$

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 `GetAccessFunctionSuperElement`.

6.4.8.3 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()

sims=exu.SimulationSettings()
sims.timeIntegration.generalizedAlpha.spectralRadius=1
SC.TimeIntegrationSolve(mbs, 'GeneralizedAlpha', sims)

#check result at default integration time
exudynTestGlobals.testResult = mbs.GetNodeOutput(nMass1, exu.OutputVariableType.
    Position)[0]
```

For further examples on MarkerSuperElementPosition see Examples:

- `NGsolvePistonEngine.py`
- `objectFFRFreducedOrderNetgen.py`
- `sliderCrankCMSacme.py`

For further examples on MarkerSuperElementPosition see TestModels:

- `objectFFRFreducedOrderAccelerations.py`
- `objectFFRFreducedOrderStressModesTest.py`
- `objectFFRFreducedOrderTest.py`
- `objectFFRFTest.py`
- `objectFFRFTest2.py`
- `objectGenericODE2Test.py`
- `superElementRigidJointTest.py`

6.4.9 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:

- The Marker has 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		MAXINT	body number to which marker is attached to
referencePosition	Vector3D	3	[0.,0.,0.]	local marker SuperElement reference position used to compute average displacement and average rotation; currently, this must be the center of weighted nodes of the marker
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
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)

6.4.9.1 DESCRIPTION of MarkerSuperElementRigid:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
bodyNumber	n_b	
referencePosition	${}^r\mathbf{p}_{0,ref}$	
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
mesh node local position	${}^r\mathbf{p}_{\,i}$	current local (within reference frame r) position of mesh node k_i in object n_b
mesh node local reference position	${}^r\mathbf{p}_{\,ref,i}$	local (within reference frame r) reference position of mesh node k_i in object n_b
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 velocity	${}^r\mathbf{v}_{\,i}$	current local (within reference frame r) velocity of mesh node k_i in object n_b
super element reference position	${}^0\mathbf{p}_{\,r}$	current reference position 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 reference position	${}^0\mathbf{p}_{\,0} = {}^0\mathbf{p}_{\,r} + {}^{0r}\mathbf{A}\,{}^r\mathbf{p}_{0,ref}$	current global marker reference position; note that ${}^0\mathbf{p}_{\,0} = {}^{0r}\mathbf{I}\,{}^r\mathbf{p}_{0,ref}$, if the object does not provide a reference frame (such as GenericODE2)
marker position	${}^0\mathbf{p}_{\,m} = {}^0\mathbf{p}_{\,0} + {}^{0r}\mathbf{A}\sum_{i=0}^{n-1} w_i \cdot {}^r\mathbf{u}_i$	current global position which is provided by marker
marker velocity	${}^0\mathbf{v}_{\,m} = {}^0\dot{\mathbf{p}}_{\,r} + {}^{0r}\mathbf{A}\,{}^r\tilde{\boldsymbol{\omega}}_r\,{}^r\mathbf{p}_{0,ref} + {}^{0r}\mathbf{A}\left(\sum_{i=0}^{n-1}(w_i \cdot {}^r\mathbf{v}_i\,) + {}^r\tilde{\boldsymbol{\omega}}_r\sum_{i=0}^{n-1}(w_i\,{}^r\mathbf{u}_i\,)\right)$	current global velocity which is provided by marker
marker local rotation	${}^r\boldsymbol{\theta}_{\,m} = \frac{\sum_{i=0}^{n-1} w_i\,{}^r\mathbf{p}_{ref,i} \times {}^r\mathbf{u}_i}{\sum_{i=0}^{n-1} w_i \vert {}^r\mathbf{p}_{i,ref}\vert^2}$	current local (within reference frame r) linearized rotation parameters

marker rotation matrix	${}^{0b}\mathbf{A}_m = {}^{0r}\mathbf{A} \begin{bmatrix} 1 & -\theta_2 & \theta_1 \\ \theta_2 & 1 & -\theta_0 \\ -\theta_1 & \theta_0 & 1 \end{bmatrix}$	current rotation matrix, which transforms the local marker coordinates and adds the rigid body transformation of floating frames ${}^{0r}\mathbf{A}$; only valid for small (linearized rotations)!
marker local angular velocity	${}^{r}\boldsymbol{\omega}_m = {}^{r}\dot{\boldsymbol{\theta}}_m = \frac{\sum_{i=0}^{n-1} w_i \, {}^{r}\tilde{\mathbf{p}}_{ref,i} \, {}^{r}\mathbf{v}_i}{\sum_{i=0}^{n-1} w_i \lvert {}^{r}\mathbf{p}_{i,ref} \rvert^2}$	local (within reference frame r) angular velocity due to mesh node velocity only
marker inertial angular velocity	${}^{0}\boldsymbol{\omega}_m = {}^{0}\boldsymbol{\omega}_r + {}^{0r}\mathbf{A} \, {}^{r}\boldsymbol{\omega}_m$	current inertial angular velocity

6.4.9.2 Marker quantities

The marker provides a 'position' jacobian, which is the derivative of the marker velocity w.r.t. the object velocity coordinates $\dot{\mathbf{q}}_{n_b}$,

$$\mathbf{J}_{m,pos} = \frac{\partial \, {}^{0}\mathbf{v}_m}{\dot{\mathbf{q}}_{n_b}} = ... \tag{6.165}$$

In ObjectFFRFreducedOrder, the jacobian may affect all reduced coordinates.

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 $\dot{\mathbf{q}}_{n_b}$,

$$\mathbf{J}_{m,rot} = \frac{\partial \, {}^{0}\boldsymbol{\omega}_m}{\partial \dot{\mathbf{q}}_{n_b}} = \frac{\partial \, {}^{0r}\mathbf{A} \, ({}^{r}\boldsymbol{\omega}_r + {}^{r}\boldsymbol{\omega}_m)}{\partial \dot{\mathbf{q}}_{n_b}} = {}^{0r}\mathbf{A} \, (\mathbf{G}_{local} + \frac{\sum_{i=0}^{n-1} w_i \, {}^{r}\tilde{\mathbf{p}}_{ref,i} \, \mathbf{J}_{i,pos}}{\sum_{i=0}^{n-1} w_i \lvert {}^{r}\mathbf{p}_{i,ref} \rvert^2}) \tag{6.166}$$

with the matrix $\mathbf{G}_{local} = \frac{\partial \, {}^{r}\boldsymbol{\omega}_r}{\partial \dot{\mathbf{q}}_{n_b}}$.

Alternative computation of rotation (under development):
In the alternative mode, where the weighting matrix $\mathbf{W}$ has the interpretation of an inertia tensor of built from nodes using weights = node mass, and the local angular momentum reads

$$\mathbf{W} \, {}^{r}\boldsymbol{\omega}_m = \sum_{i=0}^{n-1} w_i \, {}^{r}\tilde{\mathbf{p}}_{ref,i} \, {}^{r}\mathbf{v}_i = -\sum_{i=0}^{n-1} \left(w_i \, {}^{r}\tilde{\mathbf{p}}_{i,ref} \, {}^{r}\tilde{\mathbf{p}}_{i,ref} \right) {}^{r}\boldsymbol{\omega}_m \tag{6.167}$$

and the marker local rotations and marker local angular velocity are defined as

$${}^{r}\boldsymbol{\theta}_m = \mathbf{W}^{-1} \sum_{i=0}^{n-1} w_i \, {}^{r}\tilde{\mathbf{p}}_{ref,i} \, {}^{r}\mathbf{u}_i \, , \quad {}^{r}\boldsymbol{\omega}_m = \mathbf{W}^{-1} \sum_{i=0}^{n-1} w_i \, {}^{r}\tilde{\mathbf{p}}_{ref,i} \, {}^{r}\mathbf{v}_i \, , \tag{6.168}$$

with the weighting matrix (which must be invertable)

$$\mathbf{W} = -\sum_{i=0}^{n-1} w_i \, {}^{r}\tilde{\mathbf{p}}_{i,ref} \, {}^{r}\tilde{\mathbf{p}}_{i,ref} \tag{6.169}$$

and in the alternative mode, the angular velocity is defined as

$$\mathbf{J}_{m,rot} = \frac{\partial \, {}^{0}\boldsymbol{\omega}_m}{\partial \dot{\mathbf{q}}_{n_b}} = \frac{\partial \, {}^{0r}\mathbf{A} \, ({}^{r}\boldsymbol{\omega}_r + {}^{r}\boldsymbol{\omega}_m)}{\partial \dot{\mathbf{q}}_{n_b}} = {}^{0r}\mathbf{A} \, (\mathbf{G}_{local} + \mathbf{W}^{-1} \sum_{i=0}^{n-1} w_i \, {}^{r}\tilde{\mathbf{p}}_{ref,i} \, \mathbf{J}_{i,pos}) \tag{6.170}$$

Note that $\mathbf{J}_{m,rot}$ is computed by the ObjectSuperElement within the function GetAccessFunctionSuperElement.

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 further examples on MarkerSuperElementRigid see TestModels:

- `superElementRigidJointTest.py`

6.4.10 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:

- The Marker has 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		MAXINT	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

6.4.11 MarkerBodyCable2DShape

A special Marker attached to a 2D ANCF beam finite element with cubic interpolation and 8 coordinates.

Additional information for MarkerBodyCable2DShape:

- The Marker has 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		MAXINT	body number to which marker is attached to
numberOfSegments	Index		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
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

---

6.4.11.1 DESCRIPTION of MarkerBodyCable2DShape:

For further examples on MarkerBodyCable2DShape see Examples:

- `ANCF_contact_circle.py`
- `ANCF_contact_circle2.py`
- `ANCF_moving_rigidbody.py`
- `ANCF_slidingJoint2D.py`
- `sliderCrank3DwithANCFbeltDrive.py`
- `sliderCrank3DwithANCFbeltDrive2.py`

For further examples on MarkerBodyCable2DShape see TestModels:

- `ACNFslidingAndALEjointTest.py`
- `ANCFcontactCircleTest.py`
- `ANCFcontactFrictionTest.py`
- `ANCFmovingRigidBodyTest.py`

6.4.12 MarkerBodyCable2DCoordinates

A special Marker attached to the coordinates of a 2D ANCF beam finite element with cubic interpolation.

Additional information for MarkerBodyCable2DCoordinates:

- The Marker has 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		MAXINT	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

6.4.12.1 DESCRIPTION of MarkerBodyCable2DCoordinates:

For further examples on MarkerBodyCable2DCoordinates see Examples:

- `ANCF_moving_rigidbody.py`
- `ANCF_slidingJoint2D.py`
- `ANCF_switchingSlidingJoint2D.py`

For further examples on MarkerBodyCable2DCoordinates see TestModels:

- `ANCFmovingRigidBodyTest.py`
- `modelUnitTests.py`

6.5 Loads

6.5.1 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		MAXINT	marker's number to which load is applied
loadVector	Vector3D		[0.,0.,0.]	vector-valued load [SI:N]
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	PyFunctionVector3DmbsScalarVector3D		0	A python function which defines the time-dependent load
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

6.5.1.1 DESCRIPTION of LoadForceVector:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
loadVector	$\mathbf{f}$	

6.5.1.2 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 further examples on LoadForceVector see Examples:

- `interactiveTutorial.py`
- `SpringDamperMassUserFunction.py`
- `ANCF_cantilever_test.py`
- `ANCF_cantilever_test_dyn.py`
- `ANCF_contact_circle.py`
- `ANCF_contact_circle2.py`
- `ANCF_moving_rigidbody.py`
- `ANCF_slidingJoint2D.py`
- `ANCF_switchingSlidingJoint2D.py`
- `ANCF_tests2.py`
- ...

For further examples on LoadForceVector see TestModels:

- `ACNFslidingAndALEjointTest.py`
- `ANCFcontactCircleTest.py`
- `ANCFcontactFrictionTest.py`
- `ANCFmovingRigidBodyTest.py`
- `compareFullModifiedNewton.py`
- `explicitLieGroupIntegratorPythonTest.py`
- `explicitLieGroupIntegratorTest.py`
- `explicitLieGroupMBSTest.py`
- `fourBarMechanismTest.py`
- `genericODE2test.py`
- ...

6.5.2 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		MAXINT	marker's number to which load is applied
loadVector	Vector3D		[0.,0.,0.]	vector-valued load [SI:N]
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 with parameters (Real t, Vector3D load); the load represents the current value of the load; WARNING: this factor does not work in combination with static computation (loadFactor); 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

---

6.5.2.1 DESCRIPTION of LoadTorqueVector:

Information on input parameters:

input parameter	symbol	description see tables above
loadVector	τ	

6.5.2.2 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:

```python
from math import sin, cos, pi
def UFforce(mbs, t, loadVector):
    return [loadVector[0]*sin(t*10*2*pi),0,0]
```

For further examples on LoadTorqueVector see Examples:

- `sliderCrank3DwithANCFbeltDrive2.py`
- `ANCF_contact_circle.py`
- `ANCF_contact_circle2.py`
- `ANCF_slidingJoint2D.py`
- `ANCF_tests2.py`
- `ANCF_test_halfcircle.py`
- `flexibleRotor3Dtest.py`
- `rigid3Dexample.py`
- `rigidBodyIMUtest.py`
- `rigidRotor3DbasicBehaviour.py`
- ...

For further examples on LoadTorqueVector see TestModels:

- `ANCFcontactCircleTest.py`
- `ANCFcontactFrictionTest.py`
- `carRollingDiscTest.py`
- `manualExplicitIntegrator.py`
- `mecanumWheelRollingDiscTest.py`

- modelUnitTests.py
- objectFFRFreducedOrderAccelerations.py
- objectFFRFreducedOrderStressModesTest.py
- objectFFRFreducedOrderTest.py
- objectFFRFTest.py
- ...

6.5.3 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		MAXINT	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
loadVectorUserFunction	PyFunctionVector3DmbsScalarVector3D		0	A python function which defines the time-dependent loadVector.
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

---

6.5.3.1 DESCRIPTION of LoadMassProportional:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
loadVector	$\mathbf{b}$	

6.5.3.2 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`

6.5.3.3 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()
SC.TimeIntegrationSolve(mbs, 'GeneralizedAlpha', exu.SimulationSettings())

#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 further examples on LoadMassProportional see Examples:

- `ALEANCF_pipe.py`
- `ANCF_moving_rigidbody.py`
- `ANCF_slidingJoint2D.py`
- `ANCF_switchingSlidingJoint2D.py`
- `rigid3Dexample.py`
- `rigidBodyIMUtest.py`

For further examples on LoadMassProportional see TestModels:

- `genericJointUserFunctionTest.py`
- `modelUnitTests.py`
- `sphericalJointTest.py`

6.5.4 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		MAXINT	marker's number to which load is applied
load	Real		0.	scalar load [SI:N]
loadUserFunction	PyFunctionMbsScalar2		0	A python function which defines the time-dependent load; see description below
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

---

6.5.4.1 DESCRIPTION of LoadCoordinate:

Information on input parameters:

input parameter	**symbol**	**description see tables above**
load	f	

6.5.4.2 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 further examples on LoadCoordinate see Examples:

- coordinateSpringDamper.py
- geneticOptimizationExample.py
- geneticOptimizationSliderCrank.py
- lavalRotor2Dtest.py
- massSpringFrictionInteractive.py
- parameterVariationExample.py
- simulateInteractively.py
- slidercrankWithMassSpring.py
- springDamperTutorial.py

For further examples on LoadCoordinate see TestModels:

- ACNFslidingAndALEjointTest.py
- driveTrainTest.py
- geneticOptimizationTest.py
- modelUnitTests.py
- sliderCrankFloatingTest.py
- springDamperUserFunctionTest.py

6.6 Sensors

6.6.1 SensorNode

A sensor attached to a node. The sensor measures OutputVariables and outputs values into a file, showing per line [time, sensorValue[0], sensorValue[1], ...]. A user function can be attached to modify sensor values accordingly.

The item **SensorNode** with type = 'Node' has the following parameters:

Name	**type**	**size**	**default value**	**description**
name	String		''	marker's unique name
nodeNumber	NodeIndex		MAXINT	node number to which sensor is attached to
writeToFile	Bool		True	true: write sensor output to file
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
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

---

6.6.1.1 DESCRIPTION of SensorNode:

For further examples on SensorNode see Examples:

- geneticOptimizationSliderCrank.py
- massSpringFrictionInteractive.py
- mouseInteractionExample.py
- simulateInteractively.py
- sliderCrank3DwithANCFbeltDrive.py
- sliderCrankCMSacme.py
- stiffFlyballGovernor2.py

For further examples on SensorNode see TestModels:

- driveTrainTest.py
- explicitLieGroupIntegratorPythonTest.py
- explicitLieGroupIntegratorTest.py

- `explicitLieGroupMBSTest.py`
- `heavyTop.py`
- `objectFFRFreducedOrderAccelerations.py`
- `objectFFRFreducedOrderStressModesTest.py`
- `objectFFRFreducedOrderTest.py`
- `objectFFRFTest.py`
- `objectFFRFTest2.py`
- ...

6.6.2 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], ...]. A user function can be attached to postprocess sensor values accordingly.

The item **SensorObject** with type = 'Object' has the following parameters:

Name	type	size	default value	description
name	String		''	marker's unique name
objectNumber	ObjectIndex		MAXINT	object (e.g. connector) number to which sensor is attached to
writeToFile	Bool		True	true: write sensor output to file
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
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)!

6.6.2.1 DESCRIPTION of SensorObject:

For further examples on SensorObject see Examples:

- `geneticOptimizationExample.py`
- `parameterVariationExample.py`
- `sliderCrank3DwithANCFbeltDrive.py`
- `sliderCrankCMSacme.py`
- `springDamperTutorial.py`

For further examples on SensorObject see TestModels:

- `carRollingDiscTest.py`
- `geneticOptimizationTest.py`

- mecanumWheelRollingDiscTest.py
- objectFFRFTest.py
- rollingCoinPenaltyTest.py
- rollingCoinTest.py
- serialRobotTest.py

6.6.3 SensorBody

A sensor attached to a body-object with local position. As a difference to other ObjectSensors, the body sensor has 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], ...]. A user function can be attached to postprocess sensor values accordingly.

The item **SensorBody** with type = 'Body' has the following parameters:

Name	**type**	**size**	**default value**	**description**
name	String		''	marker's unique name
bodyNumber	ObjectIndex		MAXINT	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
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
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

---

6.6.3.1 DESCRIPTION of SensorBody:

For further examples on SensorBody see Examples:

- `rigidBodyIMUtest.py`
- `rigidRotor3DbasicBehaviour.py`
- `rigidRotor3DFWBW.py`
- `rigidRotor3Drunup.py`
- `sliderCrank3DwithANCFbeltDrive.py`
- `sliderCrank3DwithANCFbeltDrive2.py`

For further examples on SensorBody see TestModels:

- `carRollingDiscTest.py`
- `driveTrainTest.py`
- `explicitLieGroupIntegratorPythonTest.py`

- explicitLieGroupIntegratorTest.py
- explicitLieGroupMBSTest.py
- heavyTop.py
- mecanumWheelRollingDiscTest.py
- pendulumFriction.py
- rollingCoinPenaltyTest.py
- rollingCoinTest.py
- ...

6.6.4 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], ...]. A user function can be attached to postprocess sensor values accordingly.

The item **SensorSuperElement** with type = 'SuperElement' has the following parameters:

Name	**type**	**size**	**default value**	**description**
name	String		"	marker's unique name
bodyNumber	ObjectIndex		MAXINT	body (=object) number to which sensor is attached to
meshNodeNumber	Index		-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
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 6.3.10.3)
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

6.6.4.1 DESCRIPTION of SensorSuperElement:

For further examples on SensorSuperElement see Examples:

- `objectFFRFreducedOrderNetgen.py`
- `sliderCrankCMSacme.py`

For further examples on SensorSuperElement see TestModels:

- objectFFRFreducedOrderAccelerations.py
- objectFFRFreducedOrderStressModesTest.py
- objectFFRFreducedOrderTest.py
- objectFFRFTest.py
- objectFFRFTest2.py
- objectGenericODE2Test.py
- superElementRigidJointTest.py

6.6.5 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], ...].

The item **SensorLoad** with type = 'Load' has the following parameters:

Name	**type**	**size**	**default value**	**description**
name	String		''	marker's unique name
loadNumber	LoadIndex		MAXINT	load number to which sensor is attached to
writeToFile	Bool		True	true: write sensor output to file
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
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; CURRENTLY NOT AVAILABLE

6.6.5.1 DESCRIPTION of SensorLoad:

For further examples on SensorLoad see Examples:

- `simulateInteractively.py`
- `sliderCrank3DwithANCFbeltDrive2.py`

For further examples on SensorLoad see TestModels:

- `serialRobotTest.py`
- `springDamperUserFunctionTest.py`

Chapter 7

EXUDYN Settings and Solver Structures

This section includes the reference manual for settings which are available in the python interface, e.g. simulation settings, visualization settings, and structures for solvers. The data is auto-generated from the according interfaces in order to keep fully up-to-date with changes.

7.1 Simulation settings

This section includes hierarchical structures for simulation settings, e.g., time integration, static solver, Newton iteration and solution file export.

7.1.1 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**
writeSolutionToFile	bool		True	flag (true/false), which determines if (global) solution vector is written to file; standard quantities that are written are: solution is written as displacements and coordinatesODE1; for additional coordinates in the solution file, see the options below
appendToFile	bool		False	flag (true/false); if true, solution and solverInformation is appended to existing file (otherwise created)
writeFileHeader	bool		True	flag (true/false); if true, file header is written (turn off, e.g. for multiple runs of time integration)
writeFileFooter	bool		True	flag (true/false); if true, information at end of simulation is written: convergence, total solution time, statistics
solutionWritePeriod	UReal		0.01	time span (period), determines how often the solution is written during a simulation
exportVelocities	bool		True	add ODE2 velocities to solution file
exportAccelerations	bool		True	add ODE2 accelerations to solution file

exportODE1Velocities	bool		True	add coordinatesODE1_t to solution file
exportAlgebraicCoordinates	bool		True	add algebraicCoordinates (=Lagrange multipliers) to solution file
exportDataCoordinates	bool		True	add DataCoordinates to solution file
coordinatesSolutionFileName	FileName		'coordinatesSolution.txt'	filename and (relative) path of solution file containing all 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
sensorsAppendToFile	bool		False	flag (true/false); if true, sensor output is appended to existing file (otherwise created)
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 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
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 8.4, but for compatibility, it is recommended to use ASCII characters only (95 characters, see wiki)
outputPrecision	Index		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

7.1.2 NumericalDifferentiationSettings

Settings for numerical differentiation of a function (needed for computation of numerical jacobian e.g. in implizit integration); HOTINT1: relativeEpsilon * Maximum(minimumCoordinateSize, fabs(x(i))).
NumericalDifferentiationSettings has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
relativeEpsilon	UReal		1e-7	relative differentiation parameter epsilon; the numerical differentiation parameter ε follows from the formula ($\varepsilon = \varepsilon_\mathrm{relative} * max(q_{min}, \vert q_i + [q_i^{Ref}]\vert)$, with $\varepsilon_\mathrm{relative}$=relativeEpsilon, q_{min} =minimumCoordinateSize, q_i is the current coordinate which is differentiated, and $qRef_i$ is the reference coordinate of the current coordinate
minimumCoordinateSize	UReal		1e-2	minimum size of coordinates in relative 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
addReferenceCoordinatesToEpsilon	bool		False	true: for the size estimation of the differentiation parameter, the reference coordinate q_i^{Ref} is added to ODE2 coordinates –> see; false: only the current coordinate is used for size estimation of the differentiation parameter

7.1.3 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**
maxIterations	Index		5	maximum number of discontinuous (post Newton) iterations
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

7.1.4 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)
useNumericalDifferentiation	bool		False	flag (true/false); false = perform direct computation of jacobian, true = use numerical differentiation for jacobian
useNewtonSolver	bool		True	flag (true/false); false = linear computation, true = use Newton solver for nonlinear solution
relativeTolerance	UReal		1e-8	relative tolerance of residual for Newton (general goal of Newton is to decrease the residual by this factor)
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
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
newtonResidualMode	Index		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
adaptInitialResidual	bool		True	flag (true/false); false = standard; true: if initialResidual is very small (or zero), it may increas dramatically in first step; to achieve relativeTolerance, the initialResidual will by updated by a higher residual within the first Newton iteration
modifiedNewtonContractivity	UReal		0.5	maximum contractivity (=reduction of error in every Newton iteration) accepted by modified Newton; if contractivity is greater, a Jacobian update is computed
useModifiedNewton	bool		False	true: compute Jacobian only at first step; no Jacobian updates per step; false: Jacobian computed in every step

modifiedNewtonJacUpdatePerStep	bool		False	true: compute Jacobian at every time step, but not in every iteration (except for bad convergence ==> switch to full Newton)
maxIterations	Index		25	maximum number of iterations (including modified + restart Newton steps); after that iterations, the static/dynamic solver stops with error
maxModifiedNewtonIterations	Index		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	Index		7	maximum number of iterations for modified Newton after aJacobian update; after that number of iterations, the full Newton method is started for this step
maximumSolutionNorm	UReal		1e38	this is the maximum allowed value for solutionU.L2NormSquared() which is the square of the square norm (value=u_1^2+u_2^2+...), and solutionV/A...; if the norm of solution vectors are larger, Newton method is stopped; the default value is chosen such that it would still work for single precision numbers (float)

7.1.5 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**
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
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

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
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

7.1.6 ExplicitIntegrationSettings

Settings for generalized-alpha, implicit trapezoidal or Newmark time integration methods.
ExplicitIntegrationSettings has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
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, but also improves integration of other rigid body nodes. Only available for RK44 integrator.
dynamicSolverType	DynamicSolverType		DynamicSolverType::DOPRI5	selection of explicit solver type (DOPRI5, ExplicitEuler, ExplicitMidpoint, RK44, RK67, ...), for detailed description see DynamicSolverType, Section 4.5.3, but only referring to explicit solvers.

7.1.7 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**
newton	NewtonSettings			parameters for Newton method; used for implicit time integration methods only
discontinuous	DiscontinuousSettings			parameters for treatment of discontinuities
startTime	UReal		0	t_{start}: start time of time integration (usually set to zero)
endTime	UReal		1	t_{end}: end time of time integration
numberOfSteps	UInt		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, $h_{max} = h$
adaptiveStep	bool		True	True: the step size may be reduced if step fails; no automatic stepsize control
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 $h = h_{max} = \frac{t_{end}-t_{start}}{n_{steps}}$
minimumStepSize	UReal		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)
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.
absoluteTolerance	UReal		1e-8	a_{tol}: if automaticStepSize=True, absolute tolerance for the error control; must fulfill $a_{tol} > 0$; see Section 10.1
relativeTolerance	UReal		1e-8	r_{tol}: if automaticStepSize=True, relative tolerance for the error control; must fulfill $r_{tol} \geq 0$; see Section 10.1
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 10.1. Make this factor smaller if many steps are rejected.
stepSizeMaxIncrease	UReal		2	f_{maxInc}: if automaticStepSize=True, maximum increase of step size per step, see Section 10.1; make this factor smaller (but > 1) if too many rejected steps

preStepPyExecute	String		”	DEPRECATED, use mbs.SetPreStepUserFunction(...); Python code to be executed prior to every step and after last step, e.g. for postprocessing
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
realtimeFactor	UReal		1	if simulateInRealtime=True, this factor is used to make the simulation slower than realtime (factor < 1) or faster than realtime (factor > 1)
verboseMode	Index		0	0 ... no output, 1 ... show short step information every 2 seconds (error), 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	Index		0	same behaviour as verboseMode, but outputs all solver information to file
stepInformation	Index		2	0 ... only current step time, 1 ... show time to go, 2 ... show newton iterations (Nit) per step, 3 ... show discontinuous iterations (Dit) and newton jacobians (jac) per step
generalizedAlpha	GeneralizedAlphaSettings			parameters for generalized-alpha, implicit trapezoidal rule or Newmark (options only apply for these methods)
explicitIntegration	ExplicitIntegrationSettings			special parameters for explicit time integration

7.1.8 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**
newton	NewtonSettings			parameters for Newton method (e.g. in static solver or time integration)
discontinuous	DiscontinuousSettings			parameters for treatment of discontinuities
numberOfLoadSteps	Index		1	number of load steps; if numberOfLoadSteps=1, no load steps are used and full forces are applied at once
loadStepDuration	UReal		1	quasi-time for all load steps (added to current time in load steps)

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/numberOfLoad-Steps; loadStepTime = loadStepStart + i*loadStepDuration/numberOfLoadSteps, but loadStepStart untouched ==> increment by user
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	UReal		1000	if loadStepGeometric=true, the load steps are increased in a geometric series, see load-StepGeometric
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
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: $loadStepFactor = 1$ -> $stabilizer = 0$
adaptiveStep	bool		True	true: use step reduction if step fails; false: fixed step size
minimumStepSize	UReal		1e-8	lower limit of step size, before nonlinear solver stops
verboseMode	Index		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	Index		0	same behaviour as verboseMode, but outputs all solver information to file
stepInformation	Index		2	0 ... only current step time, 1 ... show time to go, 2 ... show newton iterations (Nit) per step, 3 ... show discontinuous iterations (Dit) and newton jacobians (jac) per step

preStepPyExecute	String		''	DEPRECATED, use mbs.SetPreStepUserFunction(...); Python code to be executed prior to every load step and after last step, e.g. for postprocessing

7.1.9 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**
timeIntegration	TimeIntegrationSettings			time integration parameters
solutionSettings	SolutionSettings			settings for solution files
staticSolver	StaticSolverSettings			static solver parameters
linearSolverType	LinearSolverType		LinearSolverType::EXUdense	selection of numerical linear solver: exu.LinearSolverType.EXUdense (dense matrix inverse), exu.LinearSolverType.EigenSparse (sparse matrix LU-factorization), ... (enumeration type)
cleanUpMemory	bool		False	true: solvers will free memory at exit (recommended for large systems); false: keep allocated memory for repeated computations to increase performance
displayStatistics	bool		False	display general computation information at end of time step (steps, iterations, function calls, step rejections, ...
displayComputationTime	bool		False	display computation time statistics at end of solving
pauseAfterEachStep	bool		False	pause after every time step or static load step(user press SPACE)
outputPrecision	Index		6	precision for floating point numbers written to console; e.g. values written by solver
numberOfThreads	Index		1	number of threads used for parallel computation (1 == scalar processing); not yet implemented (status: Nov 2019)

7.2 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.

7.2.1 VSettingsGeneral

General settings for visualization.
VSettingsGeneral has the following items:

Name	type / function return type	size	default value / function args	description
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
autoFitScene	bool		True	automatically fit scene within first second after StartRenderer()
textSize	float		12.	general text size (font size) in pixels if not overwritten; if useWindowsMonitorScaleFactor=True, the the textSize is multplied with the windows monitor scaling factor for larger texts on on high resolution monitors; 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)
textColor	Float4	4	[0.,0.,0.,1.0]	general text color (default); used for system texts in render window
useWindowsMonitorScaleFactor	bool		True	the windows monitor scaling is used for increased visibility of texts on high resolution monitors; based on GLFW glfwGetWindowContentScale
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
minSceneSize	float		0.1	minimum scene size for initial scene size and for autoFitScene, to avoid division by zero; SET GREATER THAN ZERO
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
useGradientBackground	bool		False	true = use vertical gradient for background;
coordinateSystemSize	float		5.	size of coordinate system relative to font size
drawCoordinateSystem	bool		True	false = no coordinate system shown
drawWorldBasis	bool		False	true = draw world basis coordinate system at (0,0,0)
worldBasisSize	float		1.0	size of world basis coordinate system

showComputationInfo	bool		True	true = show (hide) all computation information including EXUDYN and version
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
pointSize	float		0.01	global point size (absolute)
circleTiling	Index		16	global number of segments for circles; if smaller than 2, 2 segments are used (flat)
cylinderTiling	Index		16	global number of segments for cylinders; if smaller than 2, 2 segments are used (flat)
sphereTiling	Index		6	global number of segments for spheres; if smaller than 2, 2 segments are used (flat)
axesTiling	Index		12	global number of segments for drawing axes cylinders and cones (reduce this number, e.g. to 4, if many axes are drawn)

7.2.2 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**
outputVariableComponent	Index	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; if this component is not available by certain objects or nodes, no value is drawn
outputVariable	OutputVariableType		OutputVariableType::_None	selected contour plot output variable type; select OutputVariableType._None to deactivate contour plotting.
minValue	float	1	0	minimum value for contour plot; set manually, if automaticRange == False
maxValue	float	1	1	maximum value for contour plot; set manually, if automaticRange == False
automaticRange	bool		True	if true, the contour plot value range is chosen automatically to the maximum range

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)
showColorBar	bool		True	show the colour bar with minimum and maximum values for the contour plot
colorBarTiling	Index	1	12	number of tiles (segements) shown in the colorbar for the contour plot

7.2.3 VSettingsNodes

Visualization settings for nodes.
VSettingsNodes has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
show	bool		True	flag to decide, whether the nodes are shown
showNumbers	bool		False	flag to decide, whether the node number is shown
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
showBasis	bool		False	show basis (three axes) of coordinate system in 3D nodes
basisSize	float		0.2	size of basis for nodes
tiling	Index		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
defaultSize	float		-1.	global node size; if -1.f, node size is relative to openGL.initialMaxSceneSize
defaultColor	Float4	4	[0.2,0.2,1.,1.]	default cRGB olor for nodes; 4th value is alpha-transparency
showNodalSlopes	Index		False	draw nodal slope vectors, e.g. in ANCF beam finite elements

7.2.4 VSettingsBeams

Visualization settings for beam finite elements.
VSettingsBeams has the following items:

Name	type / function return type	size	default value / function args	description
axialTiling	Index		8	number of segments to discretise the beams axis

7.2.5 VSettingsBodies

Visualization settings for bodies.
VSettingsBodies has the following items:

Name	type / function return type	size	default value / function args	description
show	bool		True	flag to decide, whether the bodies are shown
showNumbers	bool		False	flag to decide, whether the body(=object) number is shown
defaultSize	Float3	3	[1.,1.,1.]	global body size of xyz-cube
defaultColor	Float4	4	[0.3,0.3,1.,1.]	default cRGB olor for bodies; 4th value is
deformationScaleFactor	float		1	global deformation scale factor; also applies to nodes, if drawn; used for scaled drawing of (linear) finite elements, beams, etc.
beams	VSettingsBeams			visualization settings for beams (e.g. ANCFCable or other beam elements)

7.2.6 VSettingsConnectors

Visualization settings for connectors.
VSettingsConnectors has the following items:

Name	type / function return type	size	default value / function args	description
show	bool		True	flag to decide, whether the connectors are shown
showNumbers	bool		False	flag to decide, whether the connector(=object) number is shown
defaultSize	float		0.1	global connector size; if -1.f, connector size is relative to maxSceneSize
showJointAxes	bool		False	flag to decide, whether contact joint axes of 3D joints are shown
jointAxesLength	float		0.2	global joint axes length
jointAxesRadius	float		0.02	global joint axes radius
showContact	bool		False	flag to decide, whether contact points, lines, etc. are shown

springNumberOfWindings	Index		8	number of windings for springs drawn as helical spring
contactPointsDefaultSize	float		0.02	global contact points size; if -1.f, connector size is relative to maxSceneSize
defaultColor	Float4	4	[0.2,0.2,1.,1.]	default cRGB olor for connectors; 4th value is alpha-transparency

7.2.7 VSettingsMarkers

Visualization settings for markers.
VSettingsMarkers has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
show	bool		True	flag to decide, whether the markers are shown
showNumbers	bool		False	flag to decide, whether the marker numbers are shown
drawSimplified	bool		True	draw markers with simplified symbols
defaultSize	float		-1.	global marker size; if -1.f, marker size is relative to maxSceneSize
defaultColor	Float4	4	[0.1,0.5,0.1,1.]	default cRGB olor for markers; 4th value is alpha-transparency

7.2.8 VSettingsLoads

Visualization settings for loads.
VSettingsLoads has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
show	bool		True	flag to decide, whether the loads are shown
showNumbers	bool		False	flag to decide, whether the load numbers are shown
defaultSize	float		0.2	global load size; if -1.f, load size is relative to maxSceneSize
defaultRadius	float		0.005	global radius of load axis if drawn in 3D
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
drawSimplified	bool		True	draw markers with simplified symbols
loadSizeFactor	float		0.1	if fixedLoadSize=false, then this scaling factor is used to draw the load vector

defaultColor	Float4	4	[0.7,0.1,0.1,1.]	default cRGB olor for loads; 4th value is alpha-transparency

7.2.9 VSettingsSensors

Visualization settings for sensors.
VSettingsSensors has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
show	bool		True	flag to decide, whether the sensors are shown
showNumbers	bool		False	flag to decide, whether the sensor numbers are shown
drawSimplified	bool		True	draw sensors with simplified symbols
defaultSize	float		-1.	global sensor size; if -1.f, sensor size is relative to maxSceneSize
defaultColor	Float4	4	[0.6,0.6,0.1,1.]	default cRGB olor for sensors; 4th value is alpha-transparency

7.2.10 VSettingsWindow

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**
renderWindowSize	Index2	2	[1024,768]	initial size of OpenGL render window in pixel
startupTimeout	Index		2500	OpenGL render window startup timeout in ms (change might be necessary if CPU is very slow)
alwaysOnTop	bool		False	true: OpenGL render window will be always on top of all other windows
maximize	bool		False	true: OpenGL render window will be maximized at startup
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)
keypressRotationStep	float		5.	rotation increment per keypress in degree (full rotation = 360 degree)

mouseMoveRotationFactor	float		1.	rotation increment per 1 pixel mouse movement in degree
keypressTranslationStep	float		0.1	translation increment per keypress relative to window size
zoomStepFactor	float		1.15	change of zoom per keypress (keypad +/-) or mouse wheel increment
keyPressUserFunction	KeyPressUserFunction		0	add a Python function f(key, action, mods) here, which is called every time a key is pressed; 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)				
showMouseCoordinates	bool		False	true: show OpenGL coordinates and distance to last left mouse button pressed position; switched on/off with key 'F3'
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'
ResetKeyPressUserFunction()	void			set keyPressUserFunction to zero (no function); because this cannot be assign to the variable itself

7.2.11 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**
initialCenterPoint	Float3	3	[0.,0.,0.]	centerpoint of scene (3D) at renderer startup; overwritten if autoFitScene = True
initialZoom	float		1.	initial zoom of scene; overwritten/ignored if autoFitScene = True

initialMaxSceneSize	float		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]]
multiSampling	Index	1	1	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!
lineWidth	float	1	1.	width of lines used for representation of lines, circles, points, etc.
lineSmooth	bool	1	True	draw lines smooth
textLineWidth	float	1	1.	width of lines used for representation of text
textLineSmooth	bool	1	False	draw lines for representation of text smooth
showFaces	bool	1	True	show faces of triangles, etc.; using the options showFaces=false and showFaceEdges=true gives are wire frame representation
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
showFaceEdges	bool	1	False	show edges of faces; using the options showFaces=false and showFaceEdges=true gives are wire frame representation
shadeModelSmooth	bool	1	True	true: turn on smoothing for shaders, which uses vertex normals to smooth surfaces
materialAmbientAndDiffuse	Float4	4	[0.6,0.6,0.6,1.]	4f ambient color of material
materialShininess	float	1	32.	shininess of material
materialSpecular	Float4	4	[0.6,0.6,0.6,1.]	4f specular color of material
enableLighting	bool	1	True	generally enable lighting (otherwise, colors of objects are used); OpenGL: glEnable(GL_LIGHTING)
lightModelLocalViewer	bool	1	False	select local viewer for light; maps to OpenGL glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER,...)
lightModelTwoSide	bool	1	True	enlighten also backside of object; maps to OpenGL glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,...)

lightModelAmbient	Float4	4	[0.,0.,0.,1.]	global ambient light; maps to OpenGL glLightModeli(GL_LIGHT_MODEL_AMBIENT,[r,g,b,a])
enableLight0	bool	1	True	turn on/off light0
light0position	Float4	4	[0.2,0.2,10.,0.]	4f 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
light0ambient	float	1	0.3	ambient value of GL_LIGHT0
light0diffuse	float	1	0.6	diffuse value of GL_LIGHT0
light0specular	float	1	0.5	specular 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
light0linearAttenuation	float	1	0.0	linear attenuation coefficient of GL_LIGHT0, this is a linear factor for attenuation of the light source with distance
light0quadraticAttenuation	float	1	0.0	quadratic attenuation coefficient of GL_LIGHT0, this is a quadratic factor for attenuation of the light source with distance
enableLight1	bool	1	True	turn on/off light1
light1position	Float4	4	[1.,1.,-10.,0.]	4f 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
light1ambient	float	1	0.0	ambient value of GL_LIGHT1
light1diffuse	float	1	0.5	diffuse value of GL_LIGHT1
light1specular	float	1	0.6	specular 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
light1linearAttenuation	float	1	0.0	linear attenuation coefficient of GL_LIGHT1, this is a linear factor for attenuation of the light source with distance

light1quadraticAttenuation	float	1	0.0	quadratic attenuation coefficient of GL_LIGHT1, this is a quadratic factor for attenuation of the light source with distance
drawFaceNormals	bool	1	False	draws triangle normals, e.g. at center of triangles; used for debugging of faces
drawVertexNormals	bool	1	False	draws vertex normals; used for debugging
drawNormalsLength	float	1	0.1	length of normals; used for debugging

7.2.12 VSettingsExportImages

Functionality to export images to files (.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**
saveImageTimeOut	Index		5000	timeout for safing a frame as image to disk; this is the amount of time waited for redrawing; increase for very complex scenes
saveImageFileName	FileName		'images/frame'	filename (without extension!) and (relative) path for image file(s) with consecutive numbering (e.g., frame0000.tga, frame0001.tga,...); ; directory will be created if it does not exist
saveImageFileCounter	Index		0	current value of the counter which is used to consecutively save frames (images) with consecutive numbers
saveImageSingleFile	bool		False	true: only save single files with given filename, not adding numbering; false: add numbering to files, see saveImageFileName

7.2.13 VisualizationSettings

Settings for visualization.
VisualizationSettings has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
general	VSettingsGeneral			general visualization settings
contour	VSettingsContour			contour plot visualization settings
nodes	VSettingsNodes			node visualization settings
bodies	VSettingsBodies			body visualization settings
connectors	VSettingsConnectors			connector visualization settings
markers	VSettingsMarkers			marker visualization settings

loads	VSettingsLoads			load visualization settings
sensors	VSettingsSensors			sensor visualization settings
window	VSettingsWindow			visualization window and interaction settings
openGL	VSettingsOpenGL			OpenGL rendering settings
exportImages	VSettingsExportImages			settings for exporting (saving) images to files in order to create animations

7.3 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.

7.3.1 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**
useTimer	bool		True	flag to decide, whether the timer is used (true) or not
total	Real		0.	total time measured between start and end of computation (static/dynamics)
factorization	Real		0.	solve or inverse
newtonIncrement	Real		0.	Jac^{-1} * RHS; backsubstitution
integrationFormula	Real		0.	time spent for evaluation of integration formulas
ODE2RHS	Real		0.	time for residual evaluation of ODE2 right-hand-side
ODE1RHS	Real		0.	time for residual evaluation of ODE1 right-hand-side
AERHS	Real		0.	time for residual evaluation of algebraic equations right-hand-side
totalJacobian	Real		0.	time for all jacobian computations
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)
jacobianAE	Real		0.	jacobian of algebraic equations (not counted in sum)
massMatrix	Real		0.	mass matrix computation

reactionForces	Real		0.	CqT * lambda
postNewton	Real		0.	post newton step
errorEstimator	Real		0.	for explicit solvers, additional evaluation
writeSolution	Real		0.	time for writing solution
overhead	Real		0.	overhead, such as initialization, copying and some matrix-vector multiplication
python	Real		0.	time spent for python functions
visualization	Real		0.	time spent for visualization in computation thread
Reset(...)	void		useSolverTimer	reset solver timings to initial state by assigning default values; useSolverTimer sets the useTimer flag
Sum()	Real			compute sum of all timers (except for those counted multiple, e.g., jacobians
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
ToString()	String			converts the current timings to a string

7.3.2 SolverLocalData

Solver local data structure for solution vectors, system matrices and temporary vectors and data structures.
SolverLocalData has the following items:

Name	**type / function return type**	**size**	**default value / function args**	**description**
nODE2	Index		0	number of second order ordinary diff. eq. coordinates
nODE1	Index		0	number of first order ordinary diff. eq. coordinates
nAE	Index		0	number of algebraic coordinates
nData	Index		0	number of data coordinates
nSys	Index		0	number of system (unknown) coordinates = nODE2+nODE1+nAE
startAE	Index		0	start of algebraic coordinates, but set to zero if nAE==0
systemResidual	ResizableVector			system residual vector (vectors will be linked to this vector!)
newtonSolution	ResizableVector			Newton decrement (computed from residual and jacobian)
tempODE2	ResizableVector			temporary vector for ODE2 quantities; use in initial accelerations and during Newton
temp2ODE2	ResizableVector			second temporary vector for ODE2 quantities; use in static computation
tempODE2F0	ResizableVector			temporary vector for ODE2 Jacobian
tempODE2F1	ResizableVector			temporary vector for ODE2 Jacobian

tempODE1F0	ResizableVector			temporary vector for ODE1 Jacobian
tempODE1F1	ResizableVector			temporary vector for ODE1 Jacobian
startOfStepStateAAlgorithmic	ResizableVector			additional term needed for generalized alpha (startOfStep state)
aAlgorithmic	ResizableVector			additional term needed for generalized alpha (current state)
CleanUpMemory()	void			if desired, temporary data is cleaned up to safe memory
SetLinearSolverType(...)	void		linearSolverType	set linear solver type and matrix version: links system matrices to according dense/sparse versions
GetLinearSolverType()	LinearSolverType			return current linear solver type (dense/sparse)

7.3.3 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**
maxStepSize	Real		0.	constant or maximum stepSize
minStepSize	Real		0.	minimum stepSize for static/dynamic solver; only used, if automaticStepSize is activated
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
currentStepSize	Real		0.	stepSize of current step
numberOfSteps	Index		0	number of time steps (if fixed size); n
currentStepIndex	Index		0	current step index; i
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)
currentTime	Real		0.	holds the current simulation time, copy of state.current.time; interval is [startTime,tEnd]; in static solver, duration is loadStepDuration
startTime	Real		0.	time at beginning of time integration
endTime	Real		0.	end time of static/dynamic solver

discontinuousIteration	Index		0	number of current discontinuous iteration
newtonSteps	Index		0	number of current newton steps
newtonStepsCount	Index		0	count total Newton steps
newtonJacobiCount	Index		0	count total Newton jacobian computations
rejectedModifiedNewtonSteps	Index		0	count the number of rejected modified Newton steps (switch to full Newton)
discontinuousIterationsCount	Index		0	count total number of discontinuous iterations (min. 1 per step)
rejectedAutomaticStepSizeSteps	Index		0	count the number of rejected steps in case of automatic step size control (rejected steps are repeated with smaller step size)
automaticStepSizeError	Real		0	estimated error (relative to atol + rtol*solution) of last step; must be $\leq$ 1 for a step to be accepted
ToString()	String			convert iteration statistics to string; used for displayStatistics option

7.3.4 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**
stepReductionFailed	bool		False	true, if iterations over time/static steps failed (finally, cannot be recovered)
discontinuousIterationSuccessful	bool		True	true, if last discontinuous iteration had success (failure may be recovered by adaptive step)
linearSolverFailed	bool		False	true, if linear solver failed to factorize
newtonConverged	bool		False	true, if Newton has (finally) converged
newtonSolutionDiverged	bool		False	true, if Newton diverged (may be recovered)
jacobianUpdateRequested	bool		True	true, if a jacobian update is requested in modified Newton (determined in previous step)
massMatrixNotInvertible	bool		True	true, if mass matrix is not invertable during initialization or solution (explicit solver)
discontinuousIterationError	Real		0.	error of discontinuous iterations (contact, friction, ...) outside of Newton iteration
residual	Real		0.	current Newton residual
lastResidual	Real		0.	last Newton residual to determine contractivity
contractivity	Real		0.	Newton contractivity = geometric decay of error in every step
errorCoordinateFactor	Real		1.	factor may include the number of system coordinates to reduce the residual

InitializeData()	void			initialize SolverConvergenceData by assigning default values

7.3.5 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**
finishedSuccessfully	bool		False	flag is false until solver finshed successfully (can be used as external trigger)
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
stepInformation	Index		0	this is a copy of the solvers stepInformation used for console output
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
sensorValuesTemp	ResizableVector			temporary vector for per sensor values (overwritten for every sensor; usually contains last sensor)
lastSolutionWritten	Real		0.	simulation time when last solution has been written
lastSensorsWritten	Real		0.	simulation time when last sensors have been written
lastImageRecorded	Real		0.	simulation time when last image has been recorded
cpuStartTime	Real		0.	CPU start time of computation (starts counting at computation of initial conditions)
cpuLastTimePrinted	Real		0.	CPU time when output has been printed last time
InitializeData()	void			initialize SolverOutputData by assigning default values

7.3.6 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**
timer	CSolverTimer			timer which measures the CPU time of solver sub functions
it	SolverIterationData			all information about iterations (steps, discontinuous iteration, newton,...)
conv	SolverConvergenceData			all information about tolerances, errors and residua
output	SolverOutputData			output modes and timers for exporting solver information and solution
newton	NewtonSettings			copy of newton settings from timeint or staticSolver
loadStepGeometricFactor	Real			multiplicative load step factor; this factor is computed from loadStepGeometric parameters in SolveSystem(...)
CheckInitialized(...)	bool		mainSystem	check if MainSolver and MainSystem are correctly initialized ==> otherwise raise SysError
ComputeLoadFactor(...)	Real		simulationSettings	for static solver, this is a factor in interval [0,1]; MUST be overwritten
GetSolverName()	std::string			get solver name - needed for output file header and visualization window
IsStaticSolver()	bool			return true, if static solver; needs to be overwritten in derived class
GetSimulationEndTime(...)	Real		simulationSettings	compute simulation end time (depends on static or time integration solver)
ReduceStepSize(...)	bool		mainSystem, simulationSettings, severity	reduce step size (1..normal, 2..severe problems); return true, if reduction was successful
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
PreInitializeSolverSpecific(...)	void		mainSystem, simulationSettings	pre-initialize for solver specific tasks; called at beginning of InitializeSolver, right after Solver data reset
InitializeSolverOutput(...)	void		mainSystem, simulationSettings	initialize output files; called from InitializeSolver()
InitializeSolverPreChecks(...)	bool		mainSystem, simulationSettings	check if system is solvable; initialize dense/sparse computation modes
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()
PostInitializeSolverSpecific(...)	void		mainSystem, simulationSettings	post-initialize for solver specific tasks; called at the end of InitializeSolver
SolveSystem(...)	bool		mainSystem, simulationSettings	solve System: InitializeSolver, SolveSteps, FinalizeSolver
FinalizeSolver(...)	void		mainSystem, simulationSettings	write concluding information (timer statistics, messages) and close files
SolveSteps(...)	bool		mainSystem, simulationSettings	main solver part: calls multiple InitializeStep(...)/ DiscontinuousIteration(...)/ FinishStep(...); do step reduction if necessary; return true if success, false else
UpdateCurrentTime(...)	void		mainSystem, simulationSettings	update currentTime (and load factor); MUST be overwritten in special solver class
InitializeStep(...)	void		mainSystem, simulationSettings	initialize static step / time step; python-functions; do some outputs, checks, etc.
FinishStep(...)	void		mainSystem, simulationSettings	finish static step / time step; write output of results to file
DiscontinuousIteration(...)	bool		mainSystem, simulationSettings	perform discontinuousIteration for static step / time step; CALLS ComputeNewtonResidual
Newton(...)	bool		mainSystem, simulationSettings	perform Newton method for given solver method
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
ComputeNewtonJacobian(...)	void		mainSystem, simulationSettings	compute jacobian for newton method of given solver method; store result in systemJacobian
WriteSolutionFileHeader(...)	void		mainSystem, simulationSettings	write unique file header, depending on static/ dynamic simulation
WriteCoordinatesToFile(...)	void		mainSystem, simulationSettings	write unique coordinates solution file
IsVerboseCheck(...)	bool		level	return true, if file or console output is at or above the given level
VerboseWrite(...)	void		level, str	write to console and/or file in case of level
GetODE2size()	Index			number of ODE2 equations in solver
GetODE1size()	Index			number of ODE1 equations in solver (not yet implemented)
GetAEsize()	Index			number of algebraic equations in solver
GetDataSize()	Index			number of data (history) variables in solver
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
GetNewtonSolution()	NumpyVector			get locally stored / last computed solution (=increment) of Newton
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
ComputeMassMatrix(...)	void		mainSystem, scalarFactor=1.	compute systemMassMatrix (multiplied with factor) in cSolver and return mass matrix
ComputeJacobianODE2RHS(...)	void		mainSystem, scalarFactor=1.	set systemJacobian to zero and add jacobian (multiplied with factor) of ODE2RHS to systemJacobian in cSolver
ComputeJacobianODE2RHS_t(...)	void		mainSystem, scalarFactor=1.	add jacobian of ODE2RHS_t (multiplied with factor) to systemJacobian in cSolver
ComputeJacobianAE(...)	void		mainSystem, scalarFactor_ODE2=1., scalarFactor_ODE2_t=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 and w.r.t. ODE2_t (velocity) coordinates; if velocityLevel == true, the constraints are evaluated at velocity level
ComputeODE2RHS(...)	void		mainSystem	compute the RHS of ODE2 equations in systemResidual in range(0,nODE2)
ComputeAlgebraicEquations(...)	void		mainSystem, velocityLevel=false	compute the algebraic equations in systemResidual in range(nODE2+nODE1, nODE2+nODE1+nAE)

7.3.7 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 functions, 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**
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.
it	SolverIterationData			all information about iterations (steps, discontinuous iteration, newton,...)
conv	SolverConvergenceData			all information about tolerances, errors and residua
output	SolverOutputData			output modes and timers for exporting solver information and solution
newton	NewtonSettings			copy of newton settings from timeint or staticSolver
useOldAccBasedSolver	bool		False	set this flag True, to use old (until 2021-02-05) accelerations based generalized alpha solver; this is outdated, but kept in order to ensure compatibility for some time (will be ERASED in FUTURE!)
newmarkBeta	Real			copy of parameter in timeIntegration.generalizedAlpha
newmarkGamma	Real			copy of parameter in timeIntegration.generalizedAlpha
alphaM	Real			copy of parameter in timeIntegration.generalizedAlpha
alphaF	Real			copy of parameter in timeIntegration.generalizedAlpha
spectralRadius	Real			copy of parameter in timeIntegration.generalizedAlpha
factJacAlgorithmic	Real			locally computed parameter from generalizedAlpha parameters
CheckInitialized(...)	bool		mainSystem	check if MainSolver and MainSystem are correctly initialized ==> otherwise raise SysError
ComputeLoadFactor(...)	Real		simulationSettings	for static solver, this is a factor in interval [0,1]; MUST be overwritten
GetAAlgorithmic()	NumpyVector			get locally stored / last computed algorithmic accelerations
GetStartOfStepStateAAlgorithmic()	NumpyVector			get locally stored / last computed algorithmic accelerations at start of step
SetUserFunctionUpdateCurrentTime(...)	void		mainSystem, userFunction	set user function

SetUserFunctionInitializeStep(...)	void		mainSystem, userFunction	set user function
SetUserFunctionFinishStep(...)	void		mainSystem, userFunction	set user function
SetUserFunctionDiscontinuousIteration(...)	void		mainSystem, userFunction	set user function
SetUserFunctionNewton(...)	void		mainSystem, userFunction	set user function
SetUserFunctionComputeNewtonUpdate(...)	void		mainSystem, userFunction	set user function
SetUserFunctionComputeNewtonResidual(...)	void		mainSystem, userFunction	set user function
SetUserFunctionComputeNewtonJacobian(...)	void		mainSystem, userFunction	set user function
GetSolverName()	std::string			get solver name - needed for output file header and visualization window
IsStaticSolver()	bool			return true, if static solver; needs to be overwritten in derived class
GetSimulationEndTime(...)	Real		simulationSettings	compute simulation end time (depends on static or time integration solver)
ReduceStepSize(...)	bool		mainSystem, simulationSettings, severity	reduce step size (1..normal, 2..severe problems); return true, if reduction was successful
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
PreInitializeSolverSpecific(...)	void		mainSystem, simulationSettings	pre-initialize for solver specific tasks; called at beginning of InitializeSolver, right after Solver data reset
InitializeSolverOutput(...)	void		mainSystem, simulationSettings	initialize output files; called from InitializeSolver()
InitializeSolverPreChecks(...)	bool		mainSystem, simulationSettings	check if system is solvable; initialize dense/sparse computation modes
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()
PostInitializeSolverSpecific(...)	void		mainSystem, simulationSettings	post-initialize for solver specific tasks; called at the end of InitializeSolver
SolveSystem(...)	bool		mainSystem, simulationSettings	solve System: InitializeSolver, SolveSteps, FinalizeSolver
FinalizeSolver(...)	void		mainSystem, simulationSettings	write concluding information (timer statistics, messages) and close files
SolveSteps(...)	bool		mainSystem, simulationSettings	main solver part: calls multiple InitializeStep(...)/ DiscontinuousIteration(...)/ FinishStep(...); do step reduction if necessary; return true if success, false else

UpdateCurrentTime(...)	void		mainSystem, simulationSettings	update currentTime (and load factor); MUST be overwritten in special solver class
InitializeStep(...)	void		mainSystem, simulationSettings	initialize static step / time step; python-functions; do some outputs, checks, etc.
FinishStep(...)	void		mainSystem, simulationSettings	finish static step / time step; write output of results to file
DiscontinuousIteration(...)	bool		mainSystem, simulationSettings	perform discontinuousIteration for static step / time step; CALLS ComputeNewtonResidual
Newton(...)	bool		mainSystem, simulationSettings	perform Newton method for given solver method
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
ComputeNewtonJacobian(...)	void		mainSystem, simulationSettings	compute jacobian for newton method of given solver method; store result in systemJacobian
WriteSolutionFileHeader(...)	void		mainSystem, simulationSettings	write unique file header, depending on static/ dynamic simulation
WriteCoordinatesToFile(...)	void		mainSystem, simulationSettings	write unique coordinates solution file
IsVerboseCheck(...)	bool		level	return true, if file or console output is at or above the given level
VerboseWrite(...)	void		level, str	write to console and/or file in case of level
GetODE2size()	Index			number of ODE2 equations in solver
GetODE1size()	Index			number of ODE1 equations in solver (not yet implemented)
GetAEsize()	Index			number of algebraic equations in solver
GetDataSize()	Index			number of data (history) variables in solver
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
GetNewtonSolution()	NumpyVector			get locally stored / last computed solution (=increment) of Newton
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

ComputeMassMatrix(...)	void		mainSystem, scalarFactor=1.	compute systemMassMatrix (multiplied with factor) in cSolver and return mass matrix
ComputeJacobianODE2RHS(...)	void		mainSystem, scalarFactor=1.	set systemJacobian to zero and add jacobian (multiplied with factor) of ODE2RHS to systemJacobian in cSolver
ComputeJacobianODE2RHS_t(...)	void		mainSystem, scalarFactor=1.	add jacobian of ODE2RHS_t (multiplied with factor) to systemJacobian in cSolver
ComputeJacobianAE(...)	void		mainSystem, scalarFactor_ODE2=1., scalarFactor_ODE2_t=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 and w.r.t. ODE2_t (velocity) coordinates; if velocityLevel == true, the constraints are evaluated at velocity level
ComputeODE2RHS(...)	void		mainSystem	compute the RHS of ODE2 equations in systemResidual in range(0,nODE2)
ComputeODE1RHS(...)	void		mainSystem	compute the RHS of ODE1 equations in systemResidual in range(0,nODE1)
ComputeAlgebraicEquations(...)	void		mainSystem, velocityLevel=false	compute the algebraic equations in systemResidual in range(nODE2+nODE1, nODE2+nODE1+nAE)

7.3.8 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**
timer	CSolverTimer			timer which measures the CPU time of solver sub functions
it	SolverIterationData			all information about iterations (steps, discontinuous iteration, newton,...)
conv	SolverConvergenceData			all information about tolerances, errors and residua

output	SolverOutputData			output modes and timers for exporting solver information and solution
ComputeLoadFactor(...)	Real		simulationSettings	for static solver, this is a factor in interval [0,1]; MUST be overwritten
GetNumberOfStages()	Index			return number of stages in current method
GetMethodOrder()	Index			return order of method (higher value in methods with automatic step size, e.g., DOPRI5=5)
GetSolverName()	std::string			get solver name - needed for output file header and visualization window
IsStaticSolver()	bool			return true, if static solver; needs to be overwritten in derived class
GetSimulationEndTime(...)	Real		simulationSettings	compute simulation end time (depends on static or time integration solver)
ReduceStepSize(...)	bool		mainSystem, simulationSettings, severity	reduce step size (1..normal, 2..severe problems); return true, if reduction was successful
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
PreInitializeSolverSpecific(...)	void		mainSystem, simulationSettings	pre-initialize for solver specific tasks; called at beginning of InitializeSolver, right after Solver data reset
InitializeSolverOutput(...)	void		mainSystem, simulationSettings	initialize output files; called from InitializeSolver()
InitializeSolverPreChecks(...)	bool		mainSystem, simulationSettings	check if system is solvable; initialize dense/sparse computation modes
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()
PostInitializeSolverSpecific(...)	void		mainSystem, simulationSettings	post-initialize for solver specific tasks; called at the end of InitializeSolver
SolveSystem(...)	bool		mainSystem, simulationSettings	solve System: InitializeSolver, SolveSteps, FinalizeSolver
FinalizeSolver(...)	void		mainSystem, simulationSettings	write concluding information (timer statistics, messages) and close files
SolveSteps(...)	bool		mainSystem, simulationSettings	main solver part: calls multiple InitializeStep(...)/ DiscontinuousIteration(...)/ FinishStep(...); do step reduction if necessary; return true if success, false else
UpdateCurrentTime(...)	void		mainSystem, simulationSettings	update currentTime (and load factor); MUST be overwritten in special solver class
InitializeStep(...)	void		mainSystem, simulationSettings	initialize static step / time step; python-functions; do some outputs, checks, etc.
FinishStep(...)	void		mainSystem, simulationSettings	finish static step / time step; write output of results to file

DiscontinuousIteration(...)	bool		mainSystem, simulationSettings	perform discontinuousIteration for static step / time step; CALLS ComputeNewtonResidual
Newton(...)	bool		mainSystem, simulationSettings	perform Newton method for given solver method
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
ComputeNewtonJacobian(...)	void		mainSystem, simulationSettings	compute jacobian for newton method of given solver method; store result in systemJacobian
WriteSolutionFileHeader(...)	void		mainSystem, simulationSettings	write unique file header, depending on static/ dynamic simulation
WriteCoordinatesToFile(...)	void		mainSystem, simulationSettings	write unique coordinates solution file
IsVerboseCheck(...)	bool		level	return true, if file or console output is at or above the given level
VerboseWrite(...)	void		level, str	write to console and/or file in case of level
GetODE2size()	Index			number of ODE2 equations in solver
GetODE1size()	Index			number of ODE1 equations in solver (not yet implemented)
GetAEsize()	Index			number of algebraic equations in solver
GetDataSize()	Index			number of data (history) variables in solver
GetSystemMassMatrix()	NumpyMatrix			get locally stored / last computed mass matrix of solver
GetSystemResidual()	NumpyVector			get locally stored / last computed system residual
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
ComputeMassMatrix(...)	void		mainSystem, scalarFactor=1.	compute systemMassMatrix (multiplied with factor) in cSolver and return mass matrix
ComputeODE2RHS(...)	void		mainSystem	compute the RHS of ODE2 equations in systemResidual in range(0,nODE2)
ComputeODE1RHS(...)	void		mainSystem	compute the RHS of ODE1 equations in systemResidual in range(0,nODE1)

Chapter 8

3D Graphics Visualization

The 3D graphics visualization 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`.

8.1 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
right mouse button	rotate model	keep right mouse button pressed to rotate model around current current X_1/X_2 axes
mouse wheel	zoom	use mouse wheel to zoom (on touch screens 'pinch-to-zoom' might work as well)

Note that current mouse coordinates can be obtained via `SystemContainer.GetCurrentMouseCoordinates()`.

8.2 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, which can be changed from 1=1 update per 20ms to 5=1 update per 100s
'.' 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)
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)
KEYPAD 2/8,4/6,1/9	rotate scene	about 1, 2 and 3-axis (use CTRL for small rotations)
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 body numbers	pressing this key switches the visibility of body numbers
L	show/hide loads	pressing this key switches the visibility of loads
CTRL+L	show/hide load numbers	pressing this key switches the visibility of load numbers
M	show/hide markers	pressing this key switches the visibility of markers
CTRL+M	show/hide marker numbers	pressing this key switches the visibility of marker numbers
N	show/hide nodes	pressing this key switches the visibility of nodes
CTRL+N	show/hide node numbers	pressing this key switches the visibility of node numbers
S	show/hide sensors	pressing this key switches the visibility of sensors
CTRL+S	show/hide sensor numbers	pressing this key switches the visibility of sensor numbers
T	faces transparent	add transparency to all faces (bodies, finite elements) in order to make node/marker/object numbers visible
Q	stop simulation	simulation is stopped and cannot be recovered
X	execute command	open dialog to enter a python command (in global python scope); dialog may appear behind the visualization window! User errors may lead to crash – be careful! Examples: '`print(mbs)`', '`x=5`', '`mbs.GetObject(0)`',etc.
V	visualization settings	open dialog to modify visualization settings; dialog may appear behind the visualization window!
ESCAPE	close render window	stops the simulation and closes the render window
SPACE	continue simulation	if simulation is paused, it can be continued by pressing space; use SHIFT+SPACE to continuously activate 'continue simulation'

8.3 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 operations may be speed up as data is stored internally and may be transformed by very efficient OpenGL commands in the future. However, note that the number of triangles to represent the object should be kept low (< 100000) in order to obtain a reasonably fast response of the renderer.

Many objects include a `GraphicsData` dictionary structure for definition of attached visualization of the object. 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 utilities module `exudyn.graphicsDataUtilities.p` see Section 5.4. `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':

Name	**type**	**default value**	**description**
type = 'Line':			*draws a polygonal line between all specified points*
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: data=[0,0,0, 1,0,0, 1,1,0, 0,1,0, 0,0,0] ... draws a rectangle with side length 1
type = 'Circle':			*draws a circle with center point, normal (defines plane of circle) and radius*
color	list	[0,0,0,1]	list of 4 floats to define RGB-color and transparency
radius	float	mandatory	radius
position	list	mandatory	list of float triples of x,y,z coordinates of center point of the circle
normal	list	[0,0,1]	list of float triples of x,y,z coordinates of normal to the plane of the circle; the default value gives a circle in the (x, y)-plane
type = 'Text':			*places the given text at position*
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 8.4)
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]
type = 'TriangleList':			*draws a flat triangle mesh for given points and connectivity*
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	empty	list [R0,G0,B0,A0, R1,G2,B1,A1, ...] containing $n \times 4$ floats to define RGB-color and transparency A, where n must be according to number of points; if field 'colors' does not exist, default colors will be used
normals	list	empty	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 $n_{trig} \times 3$ floats to define point indices of each vertex of the triangles (=connectivity); point indices start with index 0; the maximum index must be $\leq$ points.size()

Examples of `GraphicsData` can be found in the Python examples and in `exudynUtilities.py`.

8.4 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), see the following table 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	$\approx$
	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: ∞

Greek characters include: $\alpha, \beta, \gamma, \delta, \varepsilon, \zeta, \eta, \theta, \kappa, \lambda, \nu, \xi, \pi, \rho, \sigma, \varphi, \Delta, \Pi, \Sigma, \Omega$. Note, that unicode character codes are shown only for completeness, but they **cannot be encoded by EXUDYN**!

Chapter 9

Notation and System of equations of motion

The general idea of the code 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 follow partially the previous implementation in HOTINT [13, 9, 10].

9.1 LHS-RHS naming conventions in EXUDYN

Functions and variables contain the abbreviations LHS (*left-hand-side*) and RHS (*right-hand-side*), 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,

$$\underbrace{m \cdot \ddot{q} + d \cdot \dot{q} + k \cdot q}_{LHS} = \underbrace{f}_{RHS} \tag{9.1}$$

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]:

$$\underbrace{M_{sys} \cdot \ddot{q}_{sys}}_{LHS} = \underbrace{f_{sys}}_{RHS} \quad . \tag{9.2}$$

[1]except for the acceleration × mass matrix and constraint reaction forces, see Eq. (9.4)

In the case of the object equation

$$m \cdot \ddot{q} + d \cdot \dot{q} + k \cdot q = f\,, \tag{9.3}$$

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.

9.2 Nomenclature for system equations of motion and solvers

Using the basic notation for coordinates in Section 6.1, we use the following quantities and symbols for equations of motion and solvers:

quantity	symbol	description
number of ODE2 coordinates	n	ODE2 = 2nd order differential equations
number of ODE1 coordinates	n_y	ODE1 = 1st order differential equations
number of AE coordinates	m	AE = algebraic equations
number of system coordinates	n_s	system equations
ODE2 coordinates	$\mathbf{q} = [q_0, \ldots, q_{n_q}]^\mathrm{T}$	ODE2, displacement-based coordinates (could also be rotation or deformation coordinates)
ODE2 velocities	$\mathbf{v} = \dot{\mathbf{q}} = [\dot{q}_0, \ldots, \dot{q}_{n_q}]^\mathrm{T}$	ODE2 velocity coordinates
ODE2 accelerations	$\ddot{\mathbf{q}} = [\ddot{q}_0, \ldots, \ddot{q}_{n_q}]^\mathrm{T}$	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 × 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

9.2.1 System equations of motion

The system equations of motion in EXUDYN are represented as

$$\mathbf{M}\ddot{\mathbf{q}} + \frac{\partial \mathbf{g}}{\partial \mathbf{q}^{\mathrm{T}}}\boldsymbol{\lambda} = \mathbf{f}_q(\mathbf{q}, \dot{\mathbf{q}}, t) \quad (9.4)$$

$$\dot{\mathbf{y}} + \frac{\partial \mathbf{g}}{\partial \mathbf{y}^{\mathrm{T}}}\boldsymbol{\lambda} = \mathbf{f}_y(\mathbf{y}, t) \quad (9.5)$$

$$\mathbf{g}(\mathbf{q}, \dot{\mathbf{q}}, \mathbf{y}, \boldsymbol{\lambda}, t) = 0 \quad (9.6)$$

Note that the term $\frac{\partial \mathbf{g}}{\partial \mathbf{y}}\boldsymbol{\lambda}_y$ is not yet implemented, such that algebraic equations may not yet depend on 1st order differential equationscoordinates.

It is important to note, that for linear mechanical the term $\mathbf{f}_q$ becomes

$$\mathbf{f}_q^{lin} = \mathbf{f}_a - \mathbf{K}\mathbf{q} - \mathbf{D}\dot{\mathbf{q}} \quad (9.7)$$

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.

Chapter 10

Solvers

The description of solvers in this section follows the nomenclature given in Chapter 9. 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. 10.5, 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. 10.1 shows the basic loops for the solution process. The inner loops are shown in Fig. 10.3 andFig. 10.4. 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 7.3.1 – 7.3.5. The description of interfaces for solvers starts in Section 7.3.6.

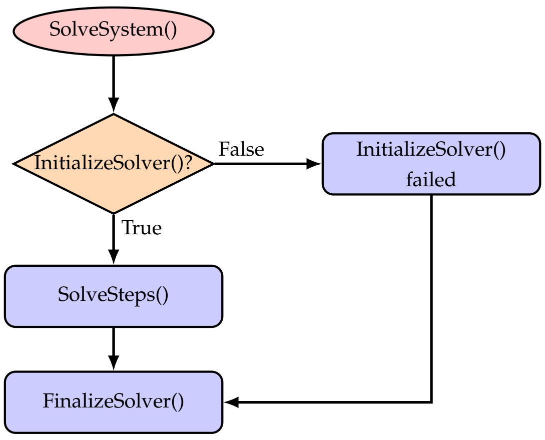


Figure 10.1: Basic solver flow chart for SolveSystem(). This flow chart is the same for static solver and for time integration.

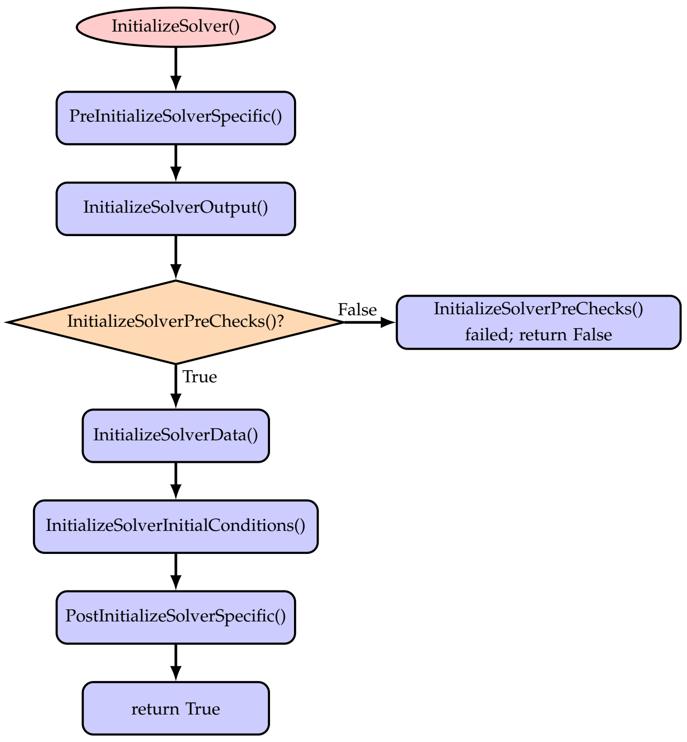

InitializeSolver()
PreInitializeSolverSpecific()
InitializeSolverOutput()
InitializeSolverPreChecks()?
False
InitializeSolverPreChecks() failed; return False
True
InitializeSolverData()
InitializeSolverInitialConditions()
PostInitializeSolverSpecific()
return True

Figure 10.2: Basic solver flow chart for function InitializeSolver().

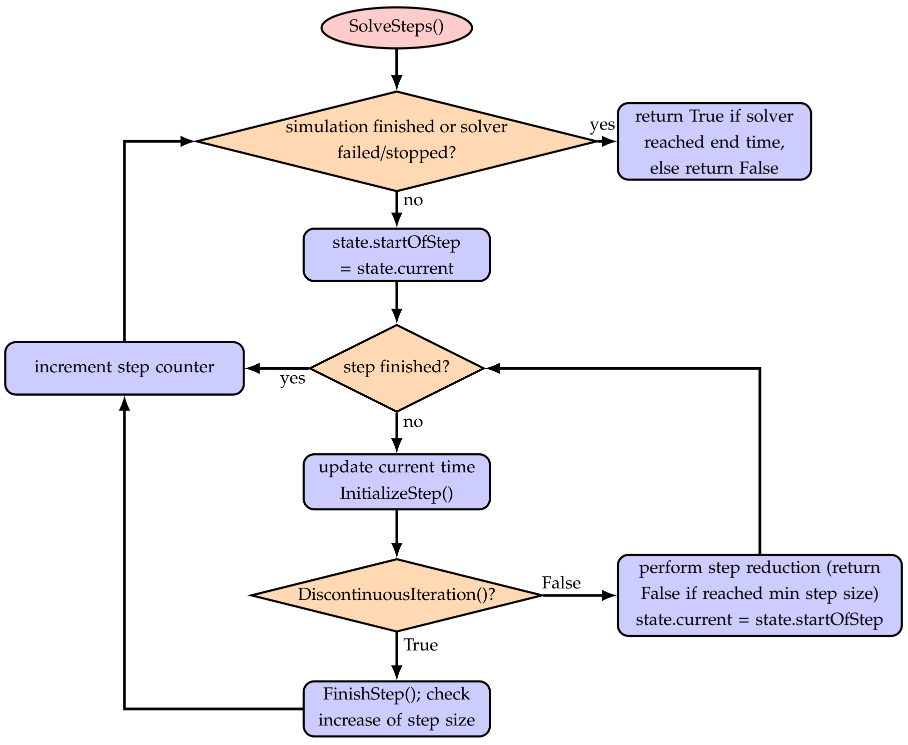
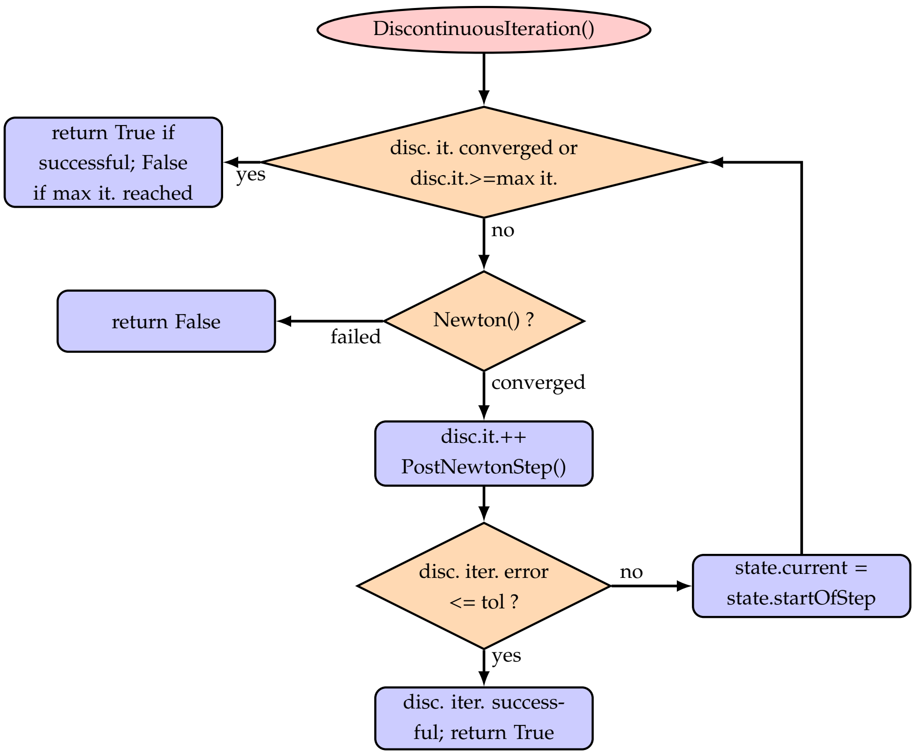
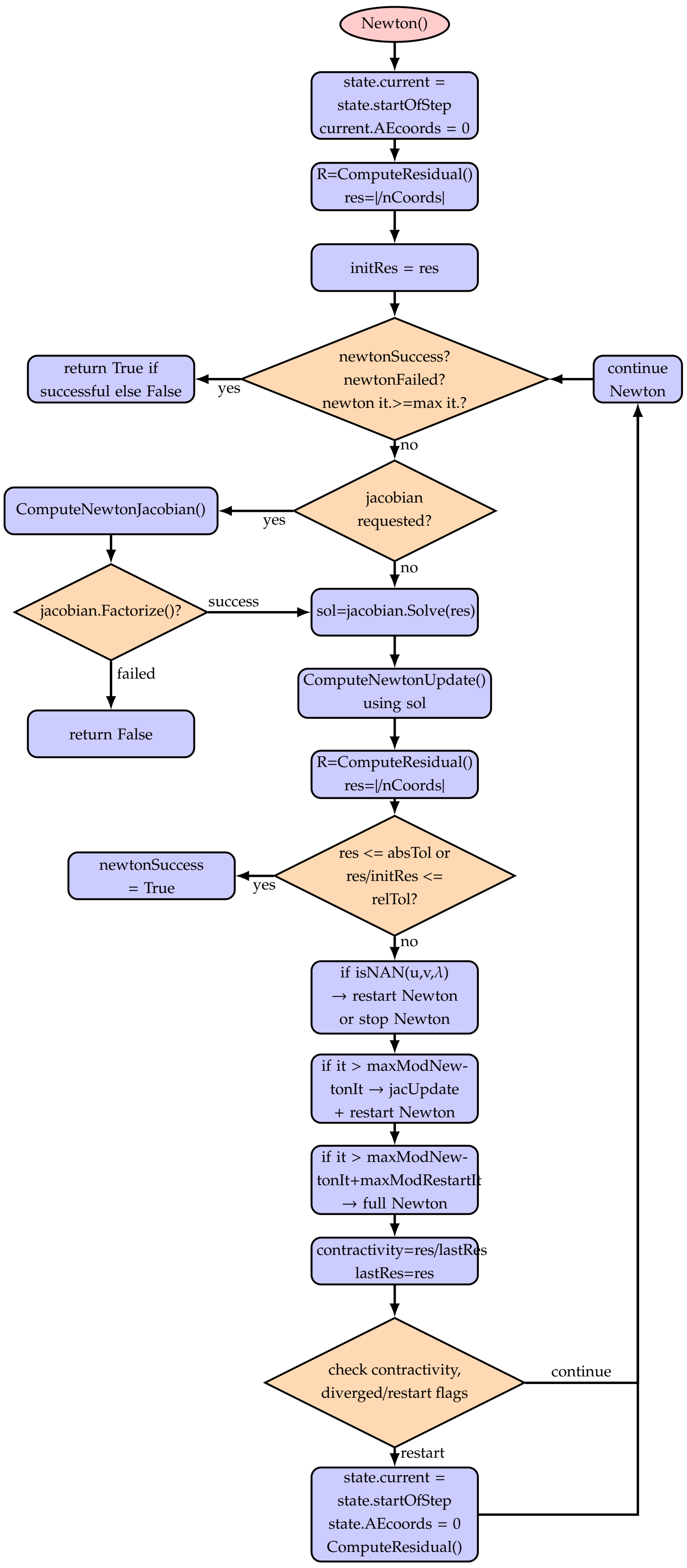

SolveSteps()
simulation finished or solver failed/stopped?
yes
return True if solver reached end time, else return False
no
state.startOfStep = state.current
step finished?
yes
increment step counter
no
update current time InitializeStep()
DiscontinuousIteration()?
False
perform step reduction (return False if reached min step size) state.current = state.startOfStep
True
FinishStep(); check increase of step size

Figure 10.3: Solver flow chart for SolveSteps(), which is the inner loop of the solver.



DiscontinuousIteration()
disc. it. converged or disc.it.>=max it.
yes
return True if successful; False if max it. reached
no
Newton() ?
failed
return False
converged
disc.it.++ PostNewtonStep()
disc. iter. error <= tol ?
no
state.current = state.startOfStep
yes
disc. iter. successful; return True

Figure 10.4: 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.



Newton()
state.current = state.startOfStep current.AEcoords = 0
R=ComputeResidual() res=|/nCoords|
initRes = res
newtonSuccess? newtonFailed? newton it.>=max it.?
yes
return True if successful else False
continue Newton
no
jacobian requested?
yes
ComputeNewtonJacobian()
no
jacobian.Factorize()?
success
failed
return False
sol=jacobian.Solve(res)
ComputeNewtonUpdate() using sol
R=ComputeResidual() res=|/nCoords|
res <= absTol or res/initRes <= relTol?
yes
newtonSuccess = True
no
if isNAN(u,v,λ) → restart Newton or stop Newton
if it > maxModNewtonIt → jacUpdate + restart Newton
if it > maxModNewtonIt+maxModRestartIt → full Newton
contractivity=res/lastRes lastRes=res
check contractivity, diverged/restart flags
continue
restart
state.current = state.startOfStep state.AEcoords = 0 ComputeResidual()

Figure 10.5: Solver flow chart for Newton(), which is run inside the DiscontinuousIteration(). The shown case is valid for newtonResidualMode = 0.

10.1 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 4.5.3, for an overview of all implicit and explicit solvers.

The solution vector ξ (denoted as y in the literature [14]), which is defined as

$$\xi = [\mathbf{q}^\mathrm{T} \;\; \dot{\mathbf{q}}^\mathrm{T} \;\; \mathbf{y}^\mathrm{T}]^\mathrm{T} \tag{10.1}$$

and which includes ODE2 coordinates and velocities and ODE1 coordinates. All coordinates are computed without reference values.

The ODE1 and ODE2 equations of Eq. (10.2), with $\lambda = 0$, are written in explicit form and converted to first order equations,

$$\begin{aligned} \dot{\mathbf{q}} &= \mathbf{v} \\ \dot{\mathbf{v}} &= \mathbf{M}^{-1}\mathbf{f}_q(\mathbf{q}, \mathbf{v}, t) \\ \dot{\mathbf{y}} &= \mathbf{f}_y(\mathbf{y}, t) \end{aligned} \tag{10.2}$$

(10.3)

The system first order differential equations for explicit solvers thus read

$$\dot{\xi} = \mathbf{f}_e(\xi, t) \tag{10.4}$$

10.1.1 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, $t = 0$),

$$\xi_{t+h} = \xi_t + \Delta\xi\,. \tag{10.5}$$

For any given Runge-Kutta method, the integration of one step with step size h is performed by an approximation

$$\Delta\xi = \int_t^{t+h} \mathbf{f}_e(\tau, \xi(\tau))d\tau \approx h\,[b_1\mathbf{f}_e(t, \xi(t)) + b_2\mathbf{f}_e(t + c_2 h, \xi(t + c_2 h)) + \ldots + b_s\mathbf{f}_e(t + \xi_s h, u(t + \xi_s h))] \tag{10.6}$$

in which $t + c_i h$ 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

$$\mathbf{g}_i \approx \xi(t + c_i h) \tag{10.7}$$

and computed by explicit integration (quadrature) formulas of lower order (g_i not to be mixed up with algebraic equations!),

$$\begin{aligned} \mathbf{g}_1 &= \xi_t \\ \mathbf{g}_2 &= \xi_t + h a_{21}\mathbf{f}_e(t, \mathbf{g}_1) \\ \mathbf{g}_3 &= \xi_t + h\,[a_{31}\mathbf{f}_e(t, \mathbf{g}_1) + a_{32}\mathbf{f}_e(t + c_2 h, \mathbf{g}_2)] \\ &\;\;\vdots \\ \mathbf{g}_s &= \xi_t + h\,[a_{s1}\mathbf{f}_e(t, \mathbf{g}_1) + a_{s2}\mathbf{f}_e(t + c_2 h, \mathbf{g}_2) + \ldots + a_{s,s-1}\mathbf{f}_e(t + c_{s-1}h, \mathbf{g}_{s-1})] \end{aligned} \tag{10.8}$$

Explicit Euler method, number of stages $s = 1$, order $p = 1$:

$$\begin{array}{c|c} 0 & \\ \hline & 1 \end{array}$$

Explicit midpoint rule, number of stages $s = 2$, order $p = 2$:

$$\begin{array}{c|cc} 0 & & \\ 1/2 & 1/2 & \\ \hline & 0 & 1 \end{array}$$

Classical explicit Runge-Kutta method (RK44) , number of stages $s = 4$, order $p = 4$:

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array}$$

Table 10.1: Several examples of tableaus for the implemented Runge-Kutta methods.

After all vectors $\mathbf{g}_i$ have been consecutively evaluated, the step is updated by Eq. (10.6).

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 10.1, using the structure

$$\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^\mathrm{T} \end{array}$$

with

$$\begin{array}{c|cccc} 0 & 0 & & & \\ c_2 & a_{21} & \ddots & & \\ \vdots & \vdots & \ddots & \ddots & \\ c_s & a_{s1} & \cdots & a_{s,s-1} & 0 \\ \hline & b_1 & \cdots & b_{s-1} & b_s \end{array}$$

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.

10.1.2 Automatic step size control

Advanced solvers, such as `ODE23` and `DOPRI5`, include automatic step size control[1].

[1]activated with `timeIntegration.automaticStepSize = True` in simulationSettings

We estimate the error of a time step with current step size h by using an embedded Runge-Kutta formula, which includes two approximations 10.6 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. [14].

The according apporximations ξ and $\hat{\xi}$ are used to estimate an error

$$e_j = |\xi_j - \hat{\xi}_j| \tag{10.9}$$

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:

$$s_j = a_{tol} + r_{tol} \cdot \max(|\xi_{0j}|, |\xi_{1j}|) \tag{10.10}$$

Then the relative, scaled, scalar error for the step, which needs to fulfill $err \leq 1$, is computed as

$$err = \sqrt{\frac{1}{n} \sum_{j=1}^{n} \left(\frac{\xi_{1j} - \hat{\xi}_{1j}}{s_j} \right)^2} \tag{10.11}$$

The optimal step size then reads

$$h_{opt} = h \cdot \left(\frac{1}{err} \right)^{(1/(q+1))} \tag{10.12}$$

Currently we use the suggested step size as

$$h_{new} = \min\Big(h_{max}, \min\Big(h \cdot f_{maxInc}, \max(h_{min}, f_{sfty} \cdot h_{opt})\Big)\Big) \tag{10.13}$$

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. [14].

10.1.3 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.

10.1.4 Explicit Lie group integrators

All explicit solvers including the automatic step size solvers (DOPRI5, ODE23) have been equiped with Lie group integration functionality – details will be published soon and some tests are made, but handle this with care.

Basically, the integration formulas, see Section 10.1.1 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`.

10.1.5 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 6.3.14ff., 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

$$g_j(\mathbf{q}) = 0 \quad \Leftrightarrow \quad q_j = 0 \tag{10.14}$$

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.

10.2 Implicit (trapezoidal rule-based, Newmark, generalized-α) 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 [19], 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** [19]
- **Generalized-α method** (= generalized Newmark method with additional parameters), see Chung and Hulbert [5] for the original method and Arnold and Brüls [1] for the application to multibody system dynamics.

10.2.1 Newmark and generalized-α method

Newmark's method has two parameters β and γ. The main ideas are

- Interpolate the displacements and the velocities linearly 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):

$$\begin{aligned} \mathbf{q}_T &= \mathbf{q}_0 + h\dot{\mathbf{q}}_0 + h^2(\frac{1}{2} - \beta)\mathbf{a}_0 + h^2\beta\mathbf{a}_T \\ \dot{\mathbf{q}}_T &= \dot{\mathbf{q}}_0 + h(1-\gamma)\mathbf{a}_0 + h\gamma\mathbf{a}_T \\ \mathbf{y}_T &= \mathbf{y}_0 + h(1-\gamma_y)\mathbf{v}_y^0 + h\gamma_y\mathbf{v}_y^T \end{aligned} \tag{10.15}$$

- Solve the system equations 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. (10.15). 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 [5], algorithmic accelerations $\mathbf{a}$ are used in Eq. (10.15). 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 $\mathbf{a} = \ddot{\mathbf{q}}$.
- For generalized-α, the algorithmic accelerations $\mathbf{a}$ are computed from the recurrence relation

$$(1-\alpha_m)\mathbf{a}_T + \alpha_m \mathbf{a}_0 = (1-\alpha_f)\ddot{\mathbf{u}}_T + \alpha_f \ddot{\mathbf{u}}_0 \tag{10.16}$$

which can be resolved for the unknown $\mathbf{a}_T$,

$$\mathbf{a}_T = \frac{(1-\alpha_f)\ddot{\mathbf{u}}_T + \alpha_f \ddot{\mathbf{u}}_0 - \alpha_m \mathbf{a}_0}{(1-\alpha_m)} \tag{10.17}$$

For the first step, one can simply use $\mathbf{a}_0 = \ddot{\mathbf{q}}_0$.

10.2.2 Parameter selection for generalized-α

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 [2], 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. 10.6 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. 10.6. 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 [5], regarding the αs

$$\alpha_m = \frac{2\rho_\infty - 1}{\rho_\infty + 1}, \quad \alpha_f = \frac{\rho_\infty}{\rho_\infty + 1} \tag{10.18}$$

and Newmarks's parameters,

$$\gamma = \frac{1}{2} - \alpha_m + \alpha_f, \quad \beta = \frac{1}{4}(1 - \alpha_m + \alpha_f)^2 \tag{10.19}$$

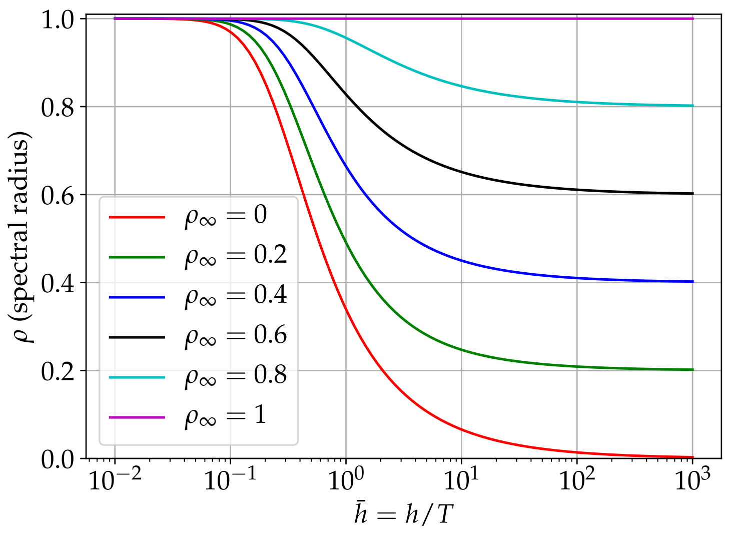


Figure 10.6: Spectral radius for generalized-α method depending on dimensionless step size $\bar{h} = h/T$, in which T is the period of an equivalent single DOF mass-spring-damper system.

10.2.3 Newton iteration

Thus, the residuals at the end of the time step (T) read (put all terms to LHS):

$$\mathbf{r}_q^{G\alpha} = \mathbf{M}\ddot{\mathbf{q}}_T + \frac{\partial \mathbf{g}}{\partial \mathbf{q}^{\mathrm{T}}}\boldsymbol{\lambda}_T - \mathbf{f}_q(\mathbf{q}_T, \dot{\mathbf{q}}_T, t) = 0 \tag{10.20}$$

$$\mathbf{r}_y^{G\alpha} = \dot{\mathbf{y}}_T + \frac{\partial \mathbf{g}}{\partial \mathbf{y}^{\mathrm{T}}}\boldsymbol{\lambda}_T - \mathbf{f}_y(\mathbf{y}_T, t) = 0 \tag{10.21}$$

$$\mathbf{r}_\lambda^{G\alpha} = \mathbf{g}(\mathbf{q}_T, \dot{\mathbf{q}}_T, \mathbf{y}_T, \boldsymbol{\lambda}_T, t) = 0 \tag{10.22}$$

We consider two options for 2nd order differential equations: (A) solve for unknown accelerations $\ddot{\mathbf{q}}_T$, or (B) for unknown displacements $\mathbf{q}_T$.

10.2.3.1 (A) Solve for unknown accelerations

The unknowns for the Newton method then are

$$\boldsymbol{\xi}_{k+1}^{G\alpha} = \begin{bmatrix} \ddot{\mathbf{q}}_T \\ \mathbf{y}_T \\ \boldsymbol{\lambda}_T \end{bmatrix} \tag{10.23}$$

and at the beginning of the step, we have

$$\boldsymbol{\xi}_{k}^{G\alpha} = \begin{bmatrix} \ddot{\mathbf{q}}_0 \\ \mathbf{y}_0 \\ \boldsymbol{\lambda}_0 \end{bmatrix} \tag{10.24}$$

For the Newton method, we need to compute an update for the unknowns Eq. (10.24), using the previous residual $\mathbf{r}_k$ and the inverse of the Jacobian $\mathbf{J}_k$ of Newton iteration k,

$$\boldsymbol{\xi}_{k+1}^{G\alpha} = \boldsymbol{\xi}_{k}^{G\alpha} - \mathbf{J}^{-1}\left(\boldsymbol{\xi}_{k}^{G\alpha}\right) \cdot \mathbf{r}^{G\alpha}\left(\boldsymbol{\xi}_{k}^{G\alpha}\right) \tag{10.25}$$

The Jacobian has the following 3×3 structure,

$$\mathbf{J} = \begin{bmatrix} \mathbf{J}_{qq} & \mathbf{J}_{qy} & \mathbf{J}_{q\lambda} \\ \mathbf{J}_{yq} & \mathbf{J}_{yy} & \mathbf{J}_{y\lambda} \\ \mathbf{J}_{\lambda q} & \mathbf{J}_{\lambda y} & \mathbf{J}_{\lambda\lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{J}_{qq} & \mathbf{0} & \mathbf{J}_{q\lambda} \\ \mathbf{0} & \mathbf{J}_{yy} & \mathbf{J}_{y\lambda} \\ \mathbf{J}_{\lambda q} & \mathbf{J}_{\lambda y} & \mathbf{J}_{\lambda\lambda} \end{bmatrix} \tag{10.26}$$

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 read

$$\begin{aligned}
\mathbf{J}_{qq} &= \frac{\partial \mathbf{r}_q^{G\alpha}}{\partial \ddot{\mathbf{q}}} = \frac{\partial \mathbf{r}_q^{G\alpha}}{\partial \mathbf{q}}\frac{\partial \mathbf{q}}{\partial \ddot{\mathbf{q}}} + \frac{\partial \mathbf{r}_q^{G\alpha}}{\partial \dot{\mathbf{q}}}\frac{\partial \dot{\mathbf{q}}}{\partial \ddot{\mathbf{q}}} = h^2\beta\mathbf{K} + h\gamma\mathbf{D} \\
\mathbf{J}_{q\lambda} &= \frac{\partial \mathbf{r}_q^{G\alpha}}{\partial \boldsymbol{\lambda}} = \frac{\partial \mathbf{g}}{\partial \mathbf{q}} \quad \left(\text{or } \frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}} \text{ for constraints at velocity level}\right) \\
\mathbf{J}_{yy} &= \frac{\partial \mathbf{r}_y^{G\alpha}}{\partial \mathbf{y}} \\
\mathbf{J}_{\lambda q} &= \frac{\partial \mathbf{r}_\lambda^{G\alpha}}{\partial \ddot{\mathbf{q}}} = \frac{\partial \mathbf{g}}{\partial \ddot{\mathbf{q}}} = \frac{\partial \mathbf{g}}{\partial \mathbf{q}}\frac{\partial \mathbf{q}}{\partial \ddot{\mathbf{q}}} + \frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}}\frac{\partial \dot{\mathbf{q}}}{\partial \ddot{\mathbf{q}}} = h^2\beta\frac{\partial \mathbf{g}}{\partial \mathbf{q}} + h\gamma\frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}} \\
\mathbf{J}_{\lambda y} &= \frac{\partial \mathbf{r}_\lambda^{G\alpha}}{\partial \mathbf{y}} \\
\mathbf{J}_{\lambda\lambda} &= \frac{\partial \mathbf{r}_\lambda^{G\alpha}}{\partial \boldsymbol{\lambda}} = \frac{\partial \mathbf{g}}{\partial \boldsymbol{\lambda}}
\end{aligned} \tag{10.27}$$

Once an update $\mathbf{q}_{k+1}^{\mathrm{Newton}}$ has been computed, the interpolation formulas (10.15) need to be evaluated before the next residual and Jacobian can be computed.

10.2.3.2 (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.

10.2.4 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. (9.4) are differentiated w.r.t. time,

$$\dot{\mathbf{g}}(\mathbf{q}_0, \dot{\mathbf{q}}_0, \mathbf{y}_0, \boldsymbol{\lambda}_0, t) = \frac{\partial \mathbf{g}}{\partial \mathbf{q}}\dot{\mathbf{q}}_0 + \frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}}\ddot{\mathbf{q}}_0 + \frac{\partial \mathbf{g}}{\partial \mathbf{y}}\dot{\mathbf{y}}_0 + \frac{\partial \mathbf{g}}{\partial \boldsymbol{\lambda}}\dot{\boldsymbol{\lambda}} + \frac{\partial \mathbf{g}}{\partial t} = 0\,. \tag{10.28}$$

Currently, we assume $\frac{\partial \mathbf{g}}{\partial \boldsymbol{\lambda}} = 0$ for all further derivations on initial accelerations. For velocity level constraints, Eq. (10.28) is used to extract initial accelerations $\ddot{\mathbf{q}}_0$,

$$\frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}}\ddot{\mathbf{q}}_0 = -\frac{\partial \mathbf{g}}{\partial \mathbf{q}}\dot{\mathbf{q}}_0 - \frac{\partial \mathbf{g}}{\partial \mathbf{y}}\dot{\mathbf{y}}_0 - \frac{\partial \mathbf{g}}{\partial t}\,. \tag{10.29}$$

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 $\mathbf{y}$,

$$\begin{bmatrix} \mathbf{M} & \mathbf{0} & \frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}^{\mathrm{T}}} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \ddot{\mathbf{q}}_0 \\ \mathbf{y}_0 \\ \boldsymbol{\lambda}_0 \end{bmatrix} = \begin{bmatrix} \mathbf{f}_q(\mathbf{q}_T, \dot{\mathbf{q}}_T, t) \\ \mathbf{y}_{init} \\ -\frac{\partial \mathbf{g}}{\partial \mathbf{q}}\dot{\mathbf{q}}_0 - \frac{\partial \mathbf{g}}{\partial \mathbf{y}}\dot{\mathbf{y}}_0 - \frac{\partial \mathbf{g}}{\partial t} \end{bmatrix}, \tag{10.30}$$

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. (10.28) and perform a second derivation w.r.t. time,

$$\ddot{\mathbf{g}}(\mathbf{q}_0, \dot{\mathbf{q}}_0, \mathbf{y}_0, \boldsymbol{\lambda}_0, t) = \frac{\partial^2 \mathbf{g}}{\partial \mathbf{q}^2}\dot{\mathbf{q}}_0^2 + 2\frac{\partial^2 \mathbf{g}}{\partial \mathbf{q}\partial t}\dot{\mathbf{q}}_0 + \frac{\partial \mathbf{g}}{\partial \mathbf{q}}\ddot{\mathbf{q}}_0 + \frac{\partial^2 \mathbf{g}}{\partial t^2} = 0 \,. \tag{10.31}$$

For position level constraints, Eq. (10.31) is used to extract initial accelerations $\ddot{\mathbf{q}}_0$,

$$\frac{\partial \mathbf{g}}{\partial \mathbf{q}}\ddot{\mathbf{q}}_0 = -2\frac{\partial^2 \mathbf{g}}{\partial \mathbf{q}\partial t}\dot{\mathbf{q}}_0 - \frac{\partial^2 \mathbf{g}}{\partial \mathbf{q}^2}\dot{\mathbf{q}}_0^2 - \frac{\partial^2 \mathbf{g}}{\partial t^2} \,. \tag{10.32}$$

Finally, the equations for the computation of the initial accelerations for position level constraints read

$$\begin{bmatrix} \mathbf{M} & \mathbf{0} & \frac{\partial \mathbf{g}}{\partial \mathbf{q}^\mathrm{T}} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \frac{\partial \mathbf{g}}{\partial \mathbf{q}} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \ddot{\mathbf{q}}_0 \\ \mathbf{y}_0 \\ \boldsymbol{\lambda}_0 \end{bmatrix} = \begin{bmatrix} \mathbf{f}_q(\mathbf{q}_T, \dot{\mathbf{q}}_T, t) \\ \mathbf{y}_{init} \\ -2\frac{\partial^2 \mathbf{g}}{\partial \mathbf{q}\partial t}\dot{\mathbf{q}}_0 - \frac{\partial^2 \mathbf{g}}{\partial \mathbf{q}^2}\dot{\mathbf{q}}_0^2 - \frac{\partial^2 \mathbf{g}}{\partial t^2} \end{bmatrix} , \tag{10.33}$$

The linear system of equations 10.30 or 10.33 is solved prior to an implicit time integration if

```
simulationSettings.timeIntegration.generalizedAlpha.computeInitialAccelerations = True
```

which is the default value.

Chapter 11

Issues and bugs

This section contains resolved issues per release and known bugs. Use this information to understand changes compared to previous versions. The mentioning of author is omitted if it was Johannes Gerstmayr (JG). BUG numbers refer to the according issue numbers. For major changes, also see Section 2.5. For details, see the `trackerlog.html` file.
General information on current version:

- EXUDYN version = 1.0.130,
- last change = 2021-02-10
- Number of issues = 580,
- Number of resolved issues = 498 (130 in current version),

11.1 Resolved issues and resolved bugs

The following list contains the issues which have been **RESOLVED** in the according version:

- **Version 1.0.130**: 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**: Issue 0569: **fix 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**: Issue 0568: **fix 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: Issue 0554: **explicit integrator** (extension)
  - description: add explicit integrator with automatic step size control (DOPRI5, ODE23); checkout Section 10.1 for description of explicit solvers and Section 4.5.3 for available solver types
  - date resolved: **2021-01-25 00:54**, date raised: 2021-01-24
- **Version 1.0.107**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: Issue 0385: **add solver eigenvalues example** (extension)
  - description: add Examples/solverFunctionsTestEigenvalues to test suite
  - **notes: added ComputeODE2EigenvaluesTest.py using new functionality exudyn.solver.ComputeODE2Eigenva**
  - date resolved: **2021-01-07 11:08**, date raised: 2020-05-06
- **Version 1.0.89**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: Issue 0524: **interpret UTF8** (change)
  - description: add conversion from UTF8 to unicode to interpret most central European characters + some important characters correctly (see Section 8.3)
  - date resolved: **2021-01-02 20:13**, date raised: 2020-12-29

- **Version 1.0.75**: 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**: 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**: BUG 0522: **fix openGl issues**
  - 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**: Issue 0521: **useWindowsMonitorScaleFactor** (extension)
  - description: add new option useWindowsMonitorScaleFactor in visualizationSettings.general to include monitor scaling factor for font sizes
  - date resolved: **2020-12-27 01:25**, date raised: 2020-12-27

- **Version 1.0.71**: 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**: 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**: 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**: Issue 0518: **windows monitor scaling** (extension)
  - description: include windows monitor (screen) scaling into drawing of texts to increase visibility on high dpi screens

  - **notes: added flag in visualizationSettings: general.useWindowsMonitorScaleFactor**
  - date resolved: **2020-12-24 00:22**, date raised: 2020-12-24

- **Version 1.0.67**: 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**: 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**: 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**: 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**: 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**: Issue 0505: **rigidBodyUtilities** (extension)
  - description: add description for RigidBodyInertia class
  - date resolved: **2020-12-17**, date raised: 2020-12-16
- **Version 1.0.61**: 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**: 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**: 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**: 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**: 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**: BUG 0499: **key V gives error**
  - description: keypress v for visualization dialog gives error
  - date resolved: **2020-12-10**, date raised: 2020-12-10
- **Version 1.0.55**: BUG 0498: **SensorObject position**

  - description: wrong position shown in sensor
  - date resolved: **2020-12-10**, date raised: 2020-12-10

- **Version 1.0.54**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: Issue 0457: **GetVersionString** (extension)
  - description: put into docu with pybindings
  - date resolved: **2020-11-29**, date raised: 2020-11-07
- **Version 1.0.26**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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: Stefan Holzinger)
- **Version 1.0.16**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: 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**: Issue 0439: **setuptools** (extension)
  - description: use setuptools for installation
  - date resolved: **2020-08-17**, date raised: 2020-08-13

- **Version 1.0.5**: 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**: 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**: 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**: 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**: 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**: BUG 0434: **CheckSystemIntegrity**
  - description: gives wrong node, marker, etc. numbers for some checks
  - date resolved: **2020-07-20**, date raised: 2020-07-20

11.2 Known open bugs

- open **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
  - date raised: 2021-02-08

- open **BUG 0456**: **ObjectFFRF bug with GenericJoint**

  - description: raises error: CSolverBase::SolveSteps CObjectSuperElement:GetAccessFunctionSuperElement: AngularVelocity_qt not implemented; cannot compute jacobian for orientation
  - date raised: 2020-10-13

- open **BUG 0448**: **ObjectGenericODE2 bug**
  - description: ObjectGenericODE2 crashes without message when initialized with invalid node numbers
  - date raised: 2020-09-09

- open **BUG 0423**: **fix MarkerSuperElementRigidBody**
  - description: fix velocity level for MarkerSuperElementRigidBody (check constraint equations)
  - date raised: 2020-06-09

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] O. A. Bauchau. *Flexible Multibody Dynamics*. Springer New York, Philadelphia, 2011.

[3] 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.

[4] 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.

[5] 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.

[6] P. Corke. *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*. Springer Publishing Company, Incorporated, 1st edition, 2013.

[7] P. Flores, J. Ambrosio, J. Pimenta Claro, and H. Lankarani. Kinematics and dynamics of multibody systems with imperfect joints. 2008.

[8] J. Gerstmayr. *EXUDYN github repository*.

[9] 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.

[10] J. Gerstmayr, A. Dorninger, R. Eder, P. Gruber, D. Reischl, M. Saxinger, M. Schï¿½rgenhumer, 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.

[11] 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.

[12] 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.

[13] 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.

[14] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving ordinary differential equations I, nonstiff problems*. Springer Berlin Heidelberg, 1987.

[15] D. Henderson. Euler angles, quaternions, and transformation matrices for space shuttle analysis. Technical report, NASA, 1977.

[16] S. Holzinger and J. Gerstmayr. Time integration of rigid bodies modelled with three rotation parameters. *Multibody System Dynamics , (currently under review)*, 2021.

[17] W. Jakob, J. Rhinelander, and D. Moldovan. pybind11 – seamless operability between c++11 and python, 2016. https://github.com/pybind/pybind11.

[18] A. Müller. Coordinate Mappings for Rigid Body Motions. *Journal of Computational and Nonlinear Dynamics*, 12(2):10, 2017.

[19] N. M. Newmark. A Method of Computation for Structural Dynamics. *ASCE Journal of the Engineering Mechanics Division*, 85(3):67–94, 1959.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] C. Woernle. *Mehrkörpersysteme: Eine Einführung in die Kinematik und Dynamik von Systemen starrer Körper*. Springer Berlin Heidelberg, 2016.

[27] T. Yoshikawa. Manipulability of robotic mechanisms. *The International Journal of Robotics Research*, 4(2):3–9, 1985.

[28] 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.

[29] A. Zwölfer and J. Gerstmayr. The nodal-based floating frame of reference formulation with modal reduction. *Acta Mechanica*, 2021.

Chapter 12

License

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one

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.
```