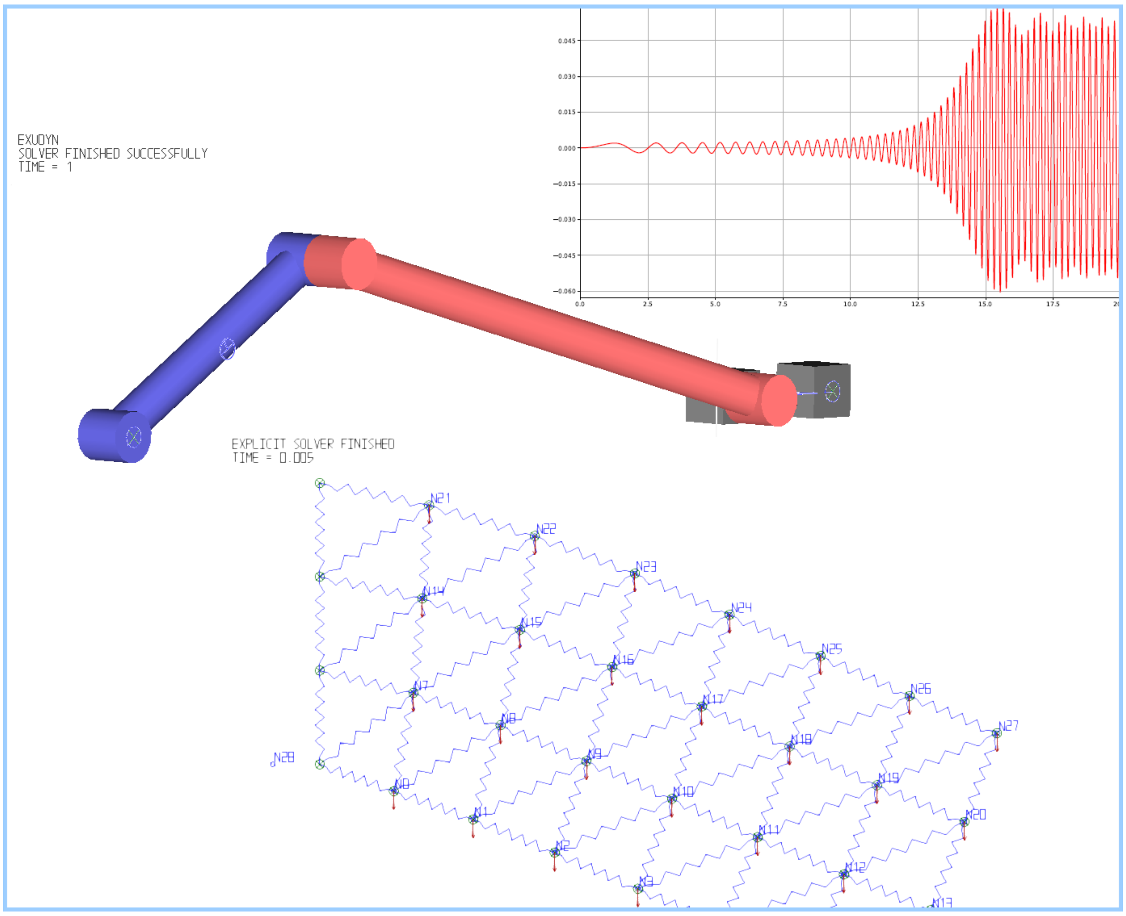
Flexible Multibody Dynamics Systems with Python and C++

EXUDYN

(flexible multibody dynamics)

User documentation



EXUDYN version = 0.1.304

University of Innsbruck, Department of Mechatronics, March 20, 2020,

Johannes Gerstmayr

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:

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

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 is a C++ based Python library for efficient simulation of flexible multibody dynamics systems. It is designed to easily set up complex multibody models, consisting of rigid and flexible bodies with joints, loads and other components.

The formulation is mostly 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 Who is developing EXUDYN?

EXUDYN is currently (3-2020) developed at the University of Innsbruck. In the first phase most of the core code has been (and still is) written by Johannes Gerstmayr, implementing ideas of earlier developments of HOTINT. 15 years of development led to a lot of lessions learned.

Some specific codes regarding pybind interface and parallelization have been written by Stefan Holzinger, who also supports the upload to GitLab. Important discussions with researchers from the community where important for the design and development of EXUDYN, where we like to mention Joachim Schöberl from TU-Vienna who influenced the design of the code. During a Comet-K2 cooperation project, several discussions with the TMECH/LCM group in Linz influenced the code development.

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

1.1.3 How to link EXUDYN with Python (recommended for beginners)?

In order to run EXUDYN, you need an appropriate Python installation. We recommend to use

- Anaconda, 32bit, Python 3.6.5 (alternatively 64bit)
- Spyder 3.2.8 with Python 3.6.5 32 bit (alternatively 64bit)[1]

If you plan to use 64bit and newer Python versions, we recommend to use VS2019 to compile your code, which offers Python 3.7 compatibility. However, you should know that Python versions and the version of the module must be identical (e.g., Python 3.6 32 **both** in the exudyn module and in Spyder).

The simplest way to start is, to copy the files (and possibly further files that are needed)

- `exudynUtilities.py`
- `itemInterface.py`
- `exudyn.pyd`

to your working directory and directly import the modules as described in tutorials and examples. The second way (**recommended**) is to use Python's `sys` module to link to your `WorkingRelease` directory, for example:

[1] It is important that Spyder, python and exudyn are either 32bit or 64bit. There will be a strange .DLL error, if you mix up 32/64bit. Furthermore, it is possible to install both, Anaconda 32bit and Anacondo 64bit; Anaconda 64bit with Python3.6 must be downloaded manually via Anaconda3-5.2.0-Windows-x86_64.exe, which can be found in repo archives.

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

The path `C:/DATA/cpp/EXUDYN_git/main/bin/WorkingRelease` needs to be adapted to the location of your `WorkingRelease` where the `exudyn.pyd` files are located. In case of 64bit, it must be changed to `.... /bin/WorkingRelease64`.

In the future, there will also be a possibility to install the module using pip commands – we are happy, if somebody could do this!

1.1.4 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 some systems, you might 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.

1.1.5 Goals of EXUDYN

After the first development phase (planned in Q4/2021), it will

- 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 $n_{DOF} < 10$),
- safe and widely accessible module for Python,
- allow to add user defined objects in C++,
- allow to add user defined solvers in Python).

1.1.6 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

Listing 1.1: My first example

```
from itemInterface import *     #conversion of data to exudyn dictionaries
import exudyn as exu           #C++ EXUDYN library
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()
SC.TimeIntegrationSolve(mbs, 'GeneralizedAlpha', simulationSettings)
```

- open Spyder and copy the example provided in Listing 1.1 into a new file, or
- open `myFirstExample.py` from your `WorkingRelease` 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/WorkingRelease/myFirstExample.py',
        wdir='C:/DATA/cpp/EXUDYN_git/main/bin/WorkingRelease')
Assemble nodes:
Set initial system coordinates (for ODE2, ODE1 and Data coordinates) ...
Time integration finished after 0.0011442 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
#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.

1.2 FAQ – 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. Why does type auto completion does not work for mbs (Main system)?

   → most python environments only have information up to the first sub-structure, e.g., `SC=exu.SystemContai` 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'.

3. 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.

4. 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.

5. 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()`

6. 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).

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

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
  - `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 5;

[1]note that 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.

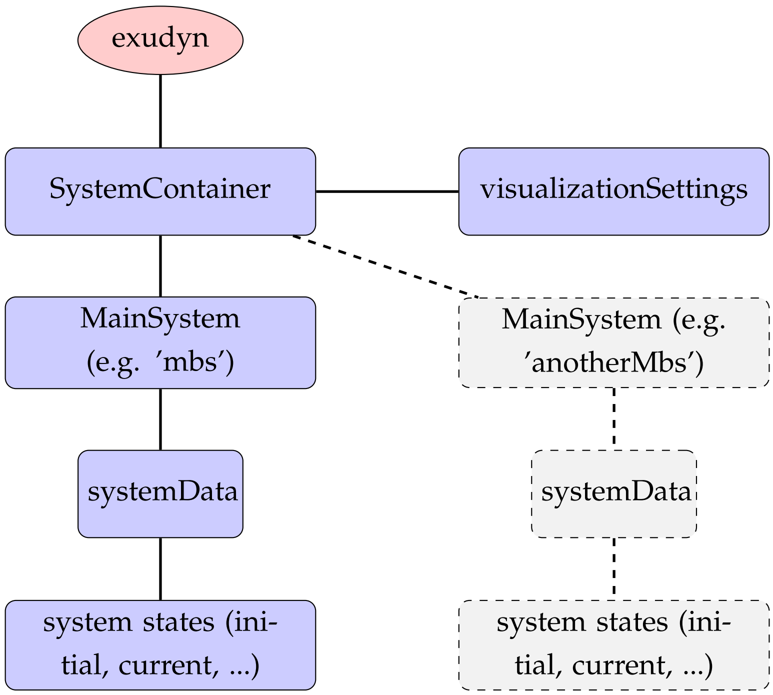


Figure 2.1: Overview of exudyn module.

– `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.

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`:

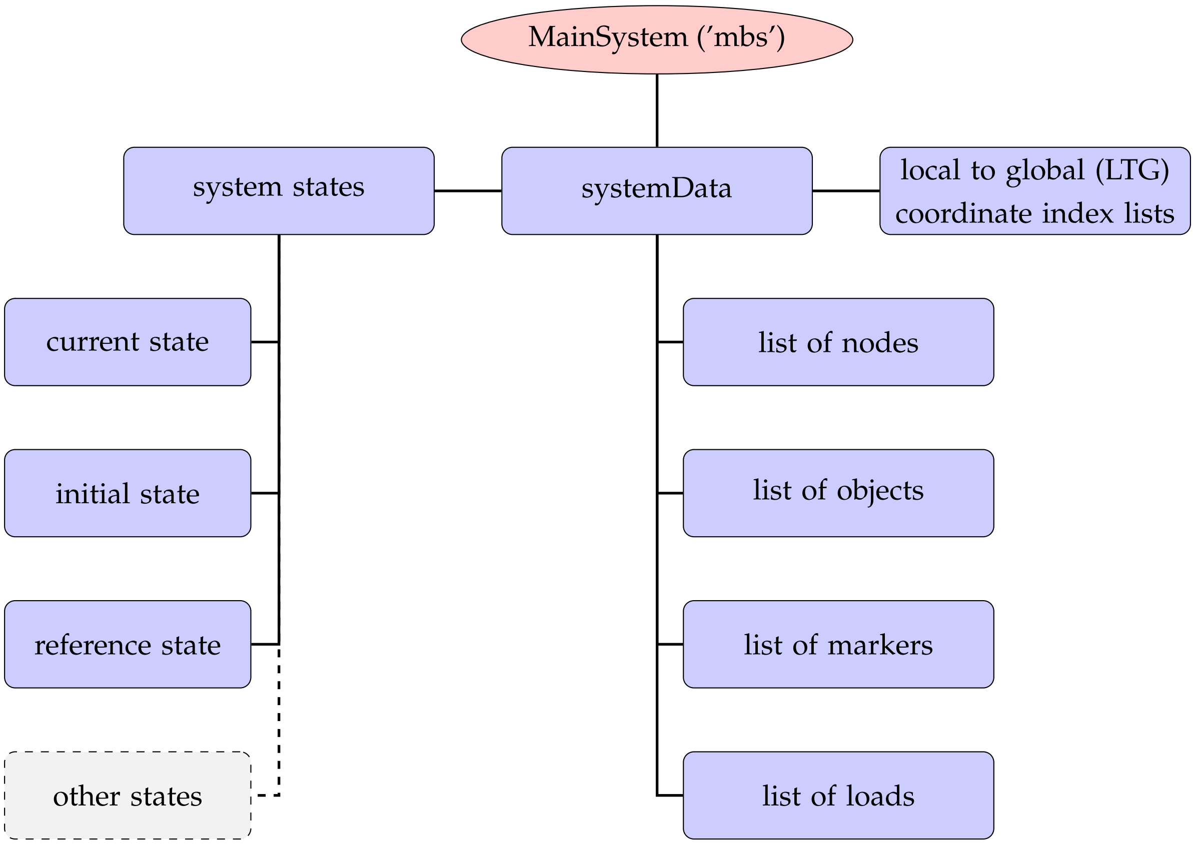


Figure 2.2: Overview of systemData, which connects items and states. Note that access to items is provided via functions in `system`.

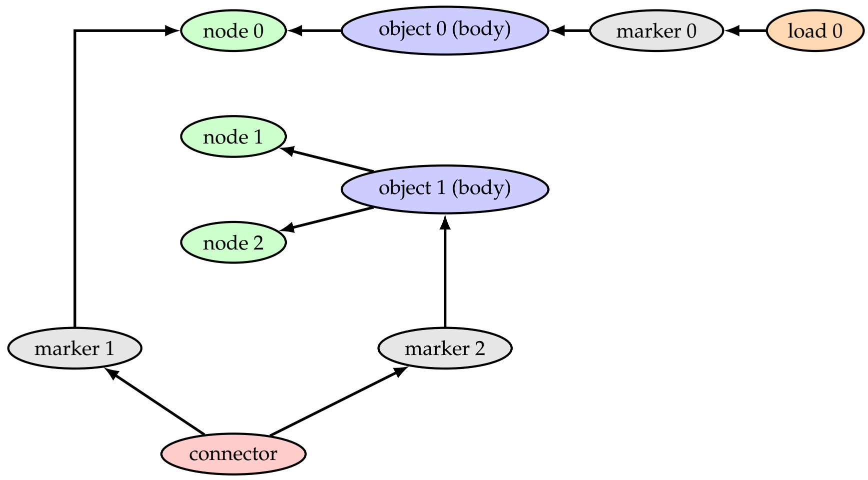


Figure 2.3: Typical interaction of items in a multibody system. Note that both, bodies and connectors/constraints are (computational) objects. The arrows indicate, that, e.g., object 1 has node 1 and node 2 (indices) and that marker 0 is attached to object 0, while load 0 uses marker 0 to apply the load.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

- `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 'preStepPyExecute' 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 6.1.1 – 6.1.7.

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 6.2.1 – 6.2.13.

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]
```

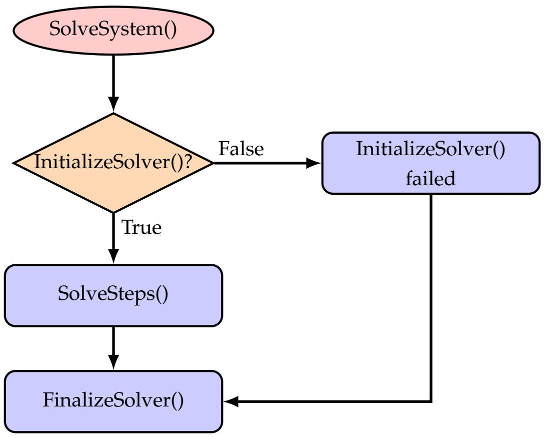


Figure 2.4: Basic solver flow chart for SolveSystem(). This flow chart is the same for static solver and for time integration.

```
#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
```

2.3.4 Solver

Both in the static as well as in the dynamic case, the solver runs in a loop to solve a nonlinear system of (differential and/or algebraic) equations over a given time or load interval. For the time integration (dynamic solver), Fig. 2.4 shows the basic loops for the solution process. The inner loops are shown in Fig. 2.6 andFig. 2.7. 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 6.3.1 – 6.3.5. The description of interfaces for solvers starts in Section 6.3.6.

2.3.5 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.

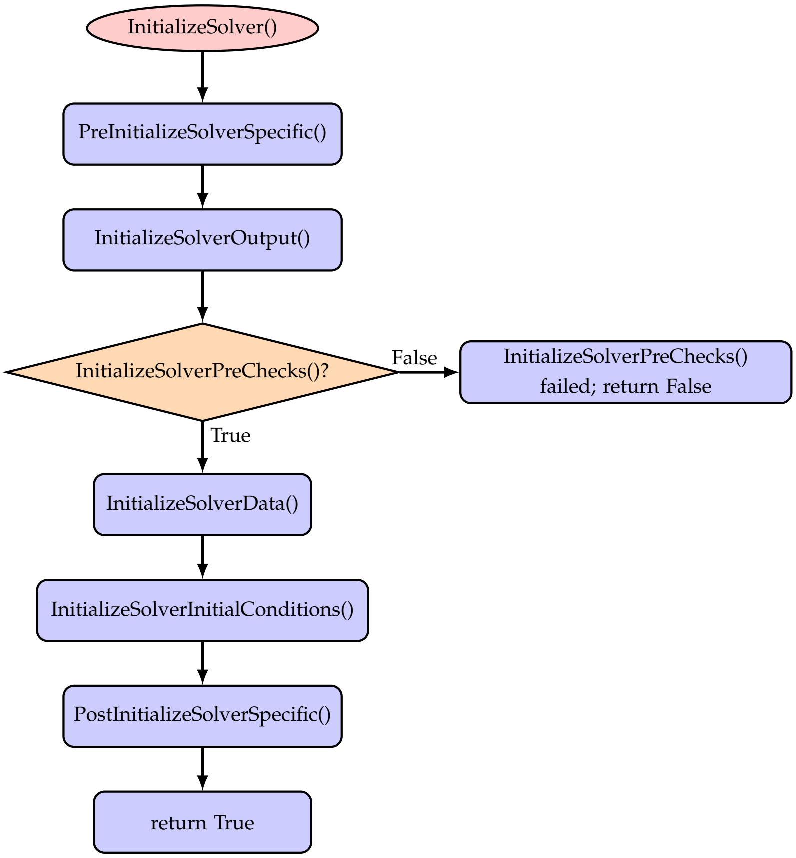


Figure 2.5: Basic solver flow chart for function InitializeSolver().

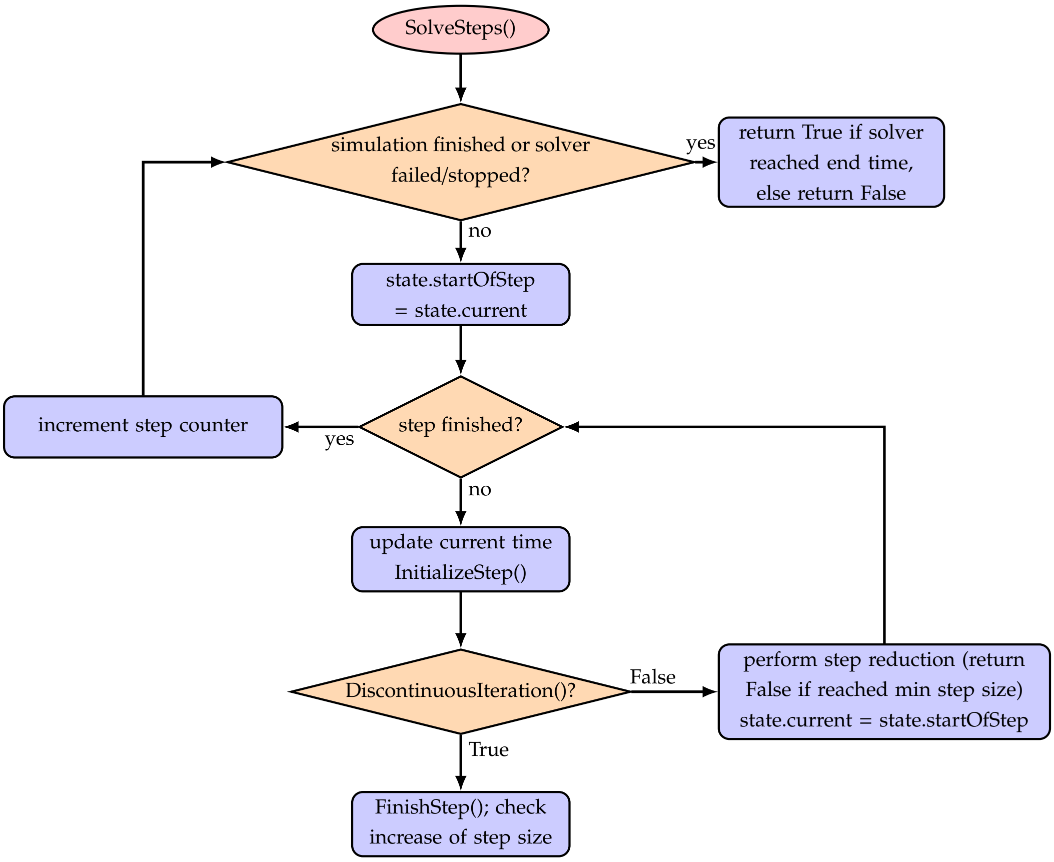


Figure 2.6: Solver flow chart for SolveSteps(), which is the inner loop of the solver.

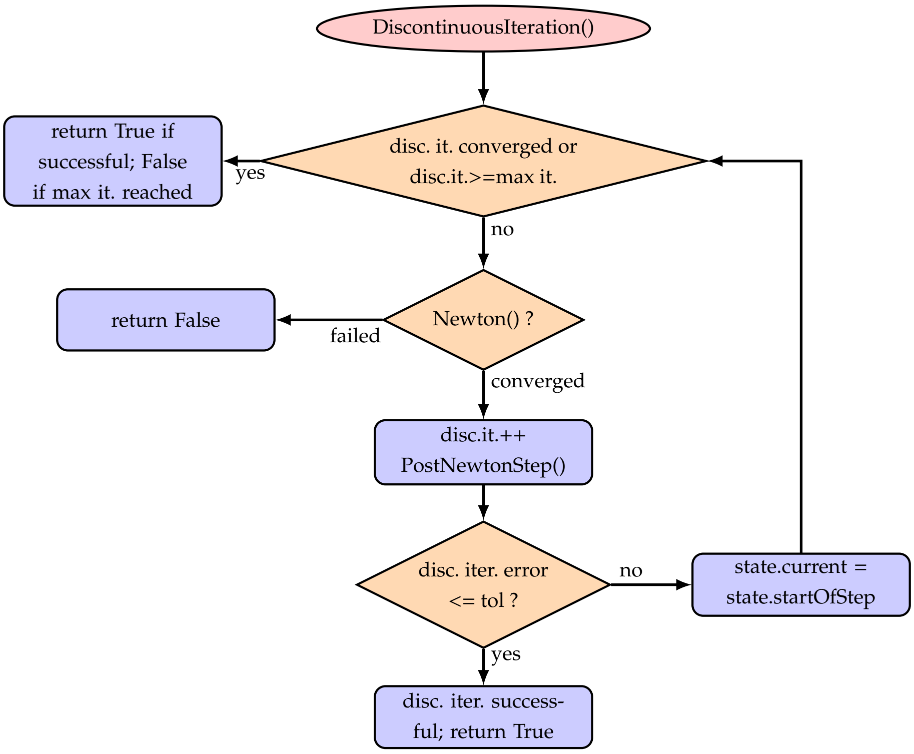


Figure 2.7: Solver flow chart for DiscontinuousIteration(), which is run for every solved step inside the static/dynamic solvers. If the DiscontinuousIteration() returns False, SolveSteps() will try to reduce the step size.

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
```

2.3.6 Graphics pipeline

The user cannot interact with the visualization part for now. There are basically two loops during simulation, which feed the graphics pipeline. The solver runs a loop:

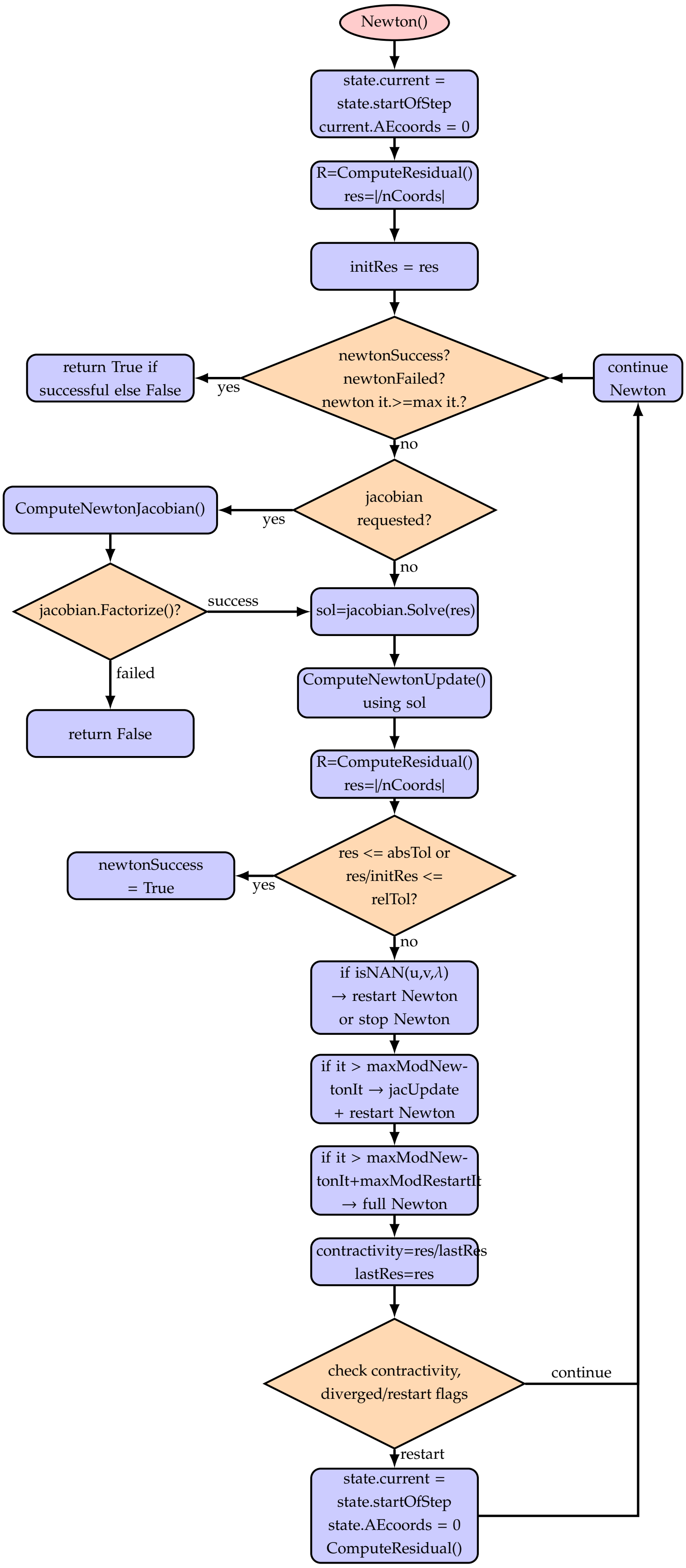


Figure 2.8: Solver flow chart for Newton(), which is run inside the DiscontinuousIteration(). The shown case is valid for newtonResidualMode = 0.

- 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 runs the following loop:

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

2.3.7 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.visulizationSettings.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 ’f́or 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, release_assert
- 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 0.1.288 → Version 0.1.289**

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

  - `Added time 't' as additional first argument in user functions: ObjectCoordinateSpringD` `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 consisten` `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.C`

- **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`

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

To start up, we set the system path to the directory of the library (needs to be adjusted; can be relative or absolute path):

```
import sys
sys.path.append('../../bin/WorkingRelease') #chose path of WorkingRelease!!!
```

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

```
import exudyn as exu
from 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`.

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`.

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`:

```
nC = 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)')
```

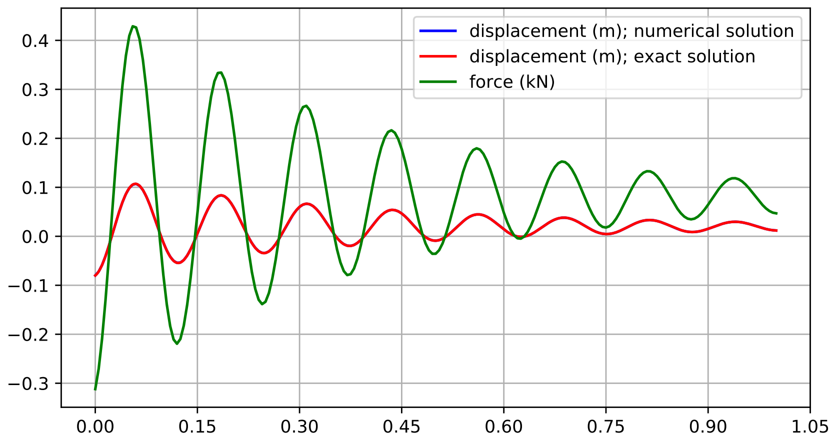
In order to get a nice plot within Spyder, the following options can be used[4]:

```
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
```

The exudyn module will usually hold one SystemContainer, which is a class that is initialized by assigning a system container to a variable, usually denoted as 'SC':

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

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:

```python
import exudyn as exu
SC = exu.SystemContainer()
exu.InfoStat()
exu.Go()
nInvalid = exu.InvalidIndex()
```

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

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

[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 the 'MainSystem' provided by SystemContainer, the following examples might help:

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

4.1 EXUDYN

These are the access functions to the EXUDYN module.

function/structure name	**description**
Go()	Creates a SystemContainer SC and a main system mbs
InfoStat()	Print some global (debug) information: linear algebra, memory allocation, threads, computational efficiency, etc.
StartRenderer()	Start OpenGL rendering engine (in separate thread)
StopRenderer()	Stop OpenGL rendering engine
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
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
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

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 by the system, e.g. for testsuite or solvers to store exudyn-wide data in order to avoid global python variables

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
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
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
GetRenderState()	Get dictionary with current render state (openGL zoom, modelview, etc.) **EXAMPLE**: SC = exu.SystemContainer() d = SC.GetRenderState() print(d['zoom'])
RedrawAndSaveImage()	Redraw openGL scene and save image (command waits until process is finished)
TimeIntegrationSolve(mainSystem, solverName, simulationSettings)	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)	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)
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

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)
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
__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 number of newly added node **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 index **EXAMPLE:** nodeDict = mbs.GetNode(0)
ModifyNode(nodeNumber, nodeDict)	modify node's dictionary by index **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 nodeNumber 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 nodeNumber 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 a object with objectDefinition from Python object class; returns (global) object number 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 index **EXAMPLE**: objectDict = mbs.GetObject(0)
ModifyObject(objectNumber, objectDict)	modify object's dictionary by index **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 objectNumber and OutputVariableType; can only be computed for exu.ConfigurationType.Current configuration!
GetObjectOutputBody(objectNumber, variableType, localPosition, configuration = ConfigurationType.Current)	get body's output variable from objectNumber and OutputVariableType **EXAMPLE**: u = mbs.GetObjectOutputBody(objectNumber = 1, variableType = exu.OutputVariableType.Position, localPosition=[1,0,0], configuration = exu.ConfigurationType.Initial)
GetObjectParameter(objectNumber, parameterName)	get objects's parameter from objectNumber 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 objectNumber 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 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 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 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())
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)
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.SetDataCoordinates(sysStateList, configurationType = 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
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 LinearSolverType

This section shows the LinearSolverType structure, which is used for selecting output values, e.g. for GetObjectOutput(...) or for selecting variables for contour plot.

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

Objects, nodes, markers, loads and sensors reference manual

This chapter includes the reference manual for all objects (bodies/constraints), nodes, markers, loads and sensors.

5.1 Notation for item equations

The following subscripts are used to define configurations of a quantity, e.g., $\mathbf{x}$:

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

As written in the introduction, the coordinates can have the types:

- ODE2 ... second order differential equations coordinates
- ODE1 ... first order differential equations coordinates; CURRENTLY NOT AVAILABLE
- AE ... algebraic equations coordinates
- Data ... data coordinates (history variables)

Finally, time is defined as 'time' or t.
The following table contains the common notation:

python name (or description)	symbol	description
displacement coordinates	$\mathbf{q} = [q_0, \ldots, q_n]^T$	vector of n displacement based coordinates in any configuration
rotation coordinates	$\boldsymbol{\psi} = [\psi_0, \ldots, \psi_\eta]^T$	vector of η **rotation based coordinates** in any configuration; this coordinates are added to reference rotation parameters to provide the current rotation parameters
algebraic coordinates	$\mathbf{z} = [z_0, \ldots, z_m]^T$	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
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
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	$\mathbf{c}_\mathrm{ref} = [c_0, \ldots, c_n]^\mathrm{T}_\mathrm{ref} = [c_{\mathrm{Ref},0}, \ldots, c_{\mathrm{Ref},n}]^\mathrm{T}_\mathrm{ref}$	n coordinates of reference configuration (can usually be set at initialization of nodes)
initialCoordinates	$\mathbf{c}_\mathrm{ini}$	initial coordinates with generalized or mixed displacement/rotation quantities (can usually be set at initialization of nodes)
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.

5.1.1 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.,

$$\mathbf{p}_{\mathrm{cur}} = \mathbf{p}_{\mathrm{ref}} + \mathbf{u}_{\mathrm{cur}} \tag{5.1}$$

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

$$\boldsymbol{\theta}_{\mathrm{cur}} = \boldsymbol{\theta}_{\mathrm{ref}} + \boldsymbol{\psi}_{\mathrm{cur}} \tag{5.2}$$

5.1.2 Coordinate Systems

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

$${}^{0}\mathbf{u} \tag{5.3}$$

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

$${}^{m1}\mathbf{u} \tag{5.4}$$

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

$${}^{0}\mathbf{u} = {}^{0,m0}\mathbf{A}\; {}^{m0}\mathbf{u} \tag{5.5}$$

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

5.2 Nodes

5.2.1 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).
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

Short name for Python: **Point**

Definition of quantities:

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

The following output parameters are available as OutputVariableType in sensors and other functions:

output parameters	symbol	description
Position	$\mathbf{p}_{\mathrm{config}} = [p_0, p_1, p_2]_{\mathrm{config}}^T = \mathbf{u}_{\mathrm{config}} + \mathbf{p}_{ref}$	global 3D position vector of node; $\mathbf{u}_{\mathrm{ref}} = 0$
Displacement	$\mathbf{u}_{\mathrm{config}} = [q_0, q_1, q_2]_{\mathrm{config}}^T$	global 3D displacement vector of node
Velocity	$\mathbf{v}_{\mathrm{config}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2]_{\mathrm{config}}^T$	global 3D velocity vector of node

Coordinates	$\mathbf{c}_{\text{config}} = \mathbf{u}_{\text{config}} = [q_0,\, q_1,\, q_2]^{\mathrm{T}}_{\text{config}}$	coordinate vector of node
Coordinates_t	$\dot{\mathbf{c}}_{\text{config}} = \mathbf{v}_{\text{config}} = [\dot{q}_0,\, \dot{q}_1,\, \dot{q}_2]^{\mathrm{T}}_{\text{config}}$	velocity coordinates vector of node

Description of Item: The node provides $n_c = 3$ displacement coordinates. According equations 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

5.2.2 NodePoint2D

A 2D point node for point masses or solid finite elements which has 2 displacement degrees of freedom for second order differential equations.
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

Short name for Python: **Point2D**

Definition of quantities:

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 parameters are available as OutputVariableType in sensors and other functions:

output parameters	**symbol**	**description**
Position	$\mathbf{p}_{\mathrm{config}} = [p_0,\, p_1,\, 0]_{\mathrm{config}}^{\mathrm{T}} = \mathbf{u}_{\mathrm{config}} + \mathbf{p}_{ref}$	global 3D position vector of node; $\mathbf{u}_{\mathrm{ref}} = 0$
Displacement	$\mathbf{u}_{\mathrm{config}} = [q_0,\, q_1,\, 0]_{\mathrm{config}}^{\mathrm{T}}$	global 3D displacement vector of node
Velocity	$\mathbf{v}_{\mathrm{config}} = [\dot{q}_0,\, \dot{q}_1,\, 0]_{\mathrm{config}}^{\mathrm{T}}$	global 3D velocity vector of node
Coordinates	$\mathbf{c}_{\mathrm{config}} = [q_0,\, q_1]_{\mathrm{config}}^{\mathrm{T}}$	coordinate vector of node
Coordinates_t	$\dot{\mathbf{c}}_{\mathrm{config}} = [\dot{q}_0,\, \dot{q}_1]_{\mathrm{config}}^{\mathrm{T}}$	velocity coordinates vector of node

Description of Item:

Note the difference of coordinate vectors and displacement or position vectors:

quantity	symbol	description
Coordinates	$\mathbf{c}_{\text{config}} = \mathbf{q}_{\text{config}} = [q_0,\, q_1]^{\mathrm{T}}_{\text{config}} = [u_0,\, u_1]^{\mathrm{T}}_{\text{config}} \ldots$	displacement coordinates
Displacement	$\mathbf{u}_{\text{config}} = [u_0,\, u_1,\, 0]^{\mathrm{T}}_{\text{config}}$	displacement vector, 0 in third component
Position	$\mathbf{p}_{\text{config}} = [p_0,\, p_1,\, 0]^{\mathrm{T}}_{\text{config}} = [u_0,\, u_1,\, 0]^{\mathrm{T}}_{\text{config}} + [r_0,\, r_1,\, 0]^{\mathrm{T}}_{\text{ref}}$	displacement vector, 0 in third component

The node provides $n_c = 2$ displacement coordinates. According equations 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.

Example for NodePoint: see ObjectMassPoint

5.2.3 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).
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

Short name for Python: **RigidEP**

Definition of quantities:

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	$\mathbf{q}_{\mathrm{ini}} = [q_0, q_1, q_2, \psi_0, \psi_1, \psi_2, \psi_3]^{\mathrm{T}}_{\mathrm{ini}} = [\mathbf{u}^{\mathrm{T}}_{\mathrm{ini}}, \boldsymbol{\psi}^{\mathrm{T}}_{\mathrm{ini}}]^{\mathrm{T}}$	
initialVelocities	$\dot{\mathbf{q}}_{\mathrm{ini}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2, \dot{\psi}_0, \dot{\psi}_1, \dot{\psi}_2, \dot{\psi}_3]^{\mathrm{T}}_{\mathrm{ini}} = [\dot{\mathbf{u}}^{\mathrm{T}}_{\mathrm{ini}}, \dot{\boldsymbol{\psi}}^{\mathrm{T}}_{\mathrm{ini}}]^{\mathrm{T}}$	

The following output parameters are available as OutputVariableType in sensors and other functions:

output parameters	symbol	description
Position	${}^{0}\mathbf{p}_{\mathrm{config}} = {}^{0}[p_0,\, p_1,\, p_2]^{\mathrm{T}}_{\mathrm{config}} = {}^{0}\mathbf{u}_{\mathrm{config}} + {}^{0}\mathbf{p}_{ref}$	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
Coordinates	$\mathbf{c}_{\mathrm{config}} = [q_0,\, q_1,\, q_2,\, \psi_0,\, \psi_1,\, \psi_2,\, \psi_3]^{\mathrm{T}}_{\mathrm{config}}$	coordinate vector of node, having 3 displacement coordinates and 4 Euler parameters
Coordinates_t	$\dot{\mathbf{c}}_{\mathrm{config}} = [\dot q_0,\, \dot q_1,\, \dot q_2,\, \dot\psi_0,\, \dot\psi_1,\, \dot\psi_2,\, \dot\psi_3]^{\mathrm{T}}_{\mathrm{config}}$	velocity 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	$[\varphi_0,\, \varphi_1,\, \varphi_2]^{\mathrm{T}}_{\mathrm{config}}$	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	${}^{0}\boldsymbol{\omega}_{\mathrm{config}} = {}^{0}[\omega_0,\, \omega_1,\, \omega_2]^{\mathrm{T}}_{\mathrm{config}}$	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

Description of Item: 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

$$1 - \sum_{i=0}^{3} \theta_i^2 = 0. \tag{5.6}$$

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,

$${}^{0}\mathbf{p}_{\mathrm{config}} = {}^{0b}\mathbf{A}_{\mathrm{config}}\, {}^{b}\mathbf{p} \tag{5.7}$$

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

$$\boldsymbol{\theta}_{\mathrm{cur}} = \boldsymbol{\psi}_{\mathrm{cur}} + \boldsymbol{\psi}_{\mathrm{ref}}. \tag{5.8}$$

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

$${}^{0b}\mathbf{A} = \begin{bmatrix} -2\theta_3^2 - 2\theta_2^2 + 1 & -2\theta_3\theta_0 + 2\theta_2\theta_1 & 2 * \theta_3\theta_1 + 2 * \theta_2\theta_0 \\ 2\theta_3\theta_0 + 2\theta_2\theta_1 & -2\theta_3^2 - 2\theta_1^2 + 1 & 2\theta_3\theta_2 - 2\theta_1\theta_0 \\ -2\theta_2\theta_0 + 2\theta_3\theta_1 & 2\theta_3\theta_2 + 2\theta_1\theta_0 & -2\theta_2^2 - 2\theta_1^2 + 1 \end{bmatrix} \tag{5.9}$$

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 $\mathbf{G}$ matrices, as used in the equations of motion for rigid bodies,

$${}^{0}\boldsymbol{\omega} = {}^{0}\mathbf{G}\, \dot{\boldsymbol{\theta}}, \tag{5.10}$$

$${}^{b}\boldsymbol{\omega} = {}^{b}\mathbf{G}\, \dot{\boldsymbol{\theta}}. \tag{5.11}$$

5.2.4 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 $-k \in \mathbb{N}$.

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

Short name for Python: **RigidRxyz**

Definition of quantities:

input parameter	symbol	description see tables above
referenceCoordinates	$\mathbf{q}_{\mathrm{ref}} = [q_0, q_1, q_2, \psi_0, \psi_1, \psi_2]^{\mathrm{T}}_{\mathrm{ref}} = [\mathbf{p}^{\mathrm{T}}_{\mathrm{ref}}, \boldsymbol{\psi}^{\mathrm{T}}_{\mathrm{ref}}]^{\mathrm{T}}$	
initialCoordinates	$\mathbf{q}_{\mathrm{ini}} = [q_0, q_1, q_2, \psi_0, \psi_1, \psi_2]^{\mathrm{T}}_{\mathrm{ini}} = [\mathbf{u}^{\mathrm{T}}_{\mathrm{ini}}, \boldsymbol{\psi}^{\mathrm{T}}_{\mathrm{ini}}]^{\mathrm{T}}$	
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 parameters are available as OutputVariableType in sensors and other functions:

output parameters	symbol	description
Position	${}^{0}\mathbf{p}_{\mathrm{config}} = {}^{0}[p_0,\, p_1,\, p_2]^{\mathrm{T}}_{\mathrm{config}} = {}^{0}\mathbf{u}_{\mathrm{config}} + {}^{0}\mathbf{p}_{ref}$	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
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
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

Description of Item: The node has 3 displacement coordinates $[q_0,\, q_1,\, q_2]^{\mathrm{T}}$ and three 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,

$$ {}^{0}\mathbf{p}_{\mathrm{config}} = {}^{0b}\mathbf{A}_{\mathrm{config}}\, {}^{b}\mathbf{p} \tag{5.12}$$

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

$$ \boldsymbol{\theta}_{\mathrm{cur}} = \boldsymbol{\psi}_{\mathrm{cur}} + \boldsymbol{\psi}_{\mathrm{ref}}. \tag{5.13}$$

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

$$ {}^{0b}\mathbf{A} = \mathbf{A}_0(\theta_0)\mathbf{A}_1(\theta_1)\mathbf{A}_2(\theta_2) \tag{5.14}$$

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 $\mathbf{G}$ matrices, as used in the equations of motion for rigid bodies,

$$ {}^{0}\boldsymbol{\omega} = {}^{0}\mathbf{G}\, \dot{\boldsymbol{\theta}}, \tag{5.15}$$

$$ {}^{b}\boldsymbol{\omega} = {}^{b}\mathbf{G}\, \dot{\boldsymbol{\theta}}. \tag{5.16}$$

5.2.5 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.

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

Short name for Python: **RigidRotVecLG**

Definition of quantities:

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

The following output parameters are available as OutputVariableType in sensors and other functions:

output parameters	symbol	description
Position	${}^0\mathbf{p}_{\mathrm{config}} = {}^0[p_0,\, p_1,\, p_2]_{\mathrm{config}}^{\mathrm{T}} = {}^0\mathbf{u}_{\mathrm{config}} + {}^0\mathbf{p}_{ref}$	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
Coordinates	$\mathbf{c}_{\mathrm{config}} = [q_0,\, q_1,\, q_2,\, \nu_0,\, \nu_1,\, \nu_2]^{\mathrm{T}}_{\mathrm{config}}$	coordinate vector of node, having 3 displacement coordinates and 3 Euler angles
Coordinates_t	$\dot{\mathbf{c}}_{\mathrm{config}} = [\dot q_0,\, \dot q_1,\, \dot q_2,\, \dot \nu_0,\, \dot \nu_1,\, \dot \nu_2]^{\mathrm{T}}_{\mathrm{config}}$	velocity 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	$[\varphi_0,\, \varphi_1,\, \varphi_2]^{\mathrm{T}}_{\mathrm{config}}$	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

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

$$\boldsymbol{\nu} = \varphi \mathbf{n} = \boldsymbol{\nu}_{\mathrm{config}} + \boldsymbol{\nu}_{\mathrm{ref}}, \tag{5.17}$$

with the rotation angle φ and the rotation axis $\mathbf{n}$. All coordinates $\mathbf{c}_{\mathrm{config}}$ lead to second order differential equations, however the rotation vector cannot be used as a conventional parameterization. It must be computed within a nonlinear update, using appropriate Lie group methods.

The 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,

$${}^{0}\mathbf{p}_{\mathrm{config}} = {}^{0b}\mathbf{A}_{\mathrm{config}}\, {}^{b}\mathbf{p} \tag{5.18}$$

Note that the rotation vector $\boldsymbol{\nu}_{\mathrm{cur}}$ are computed as sum of current coordinates plus reference coordinates,

$$\boldsymbol{\theta}_{\mathrm{cur}} = \boldsymbol{\nu}_{\mathrm{cur}} + \boldsymbol{\nu}_{\mathrm{ref}} \quad \text{with.} \tag{5.19}$$

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 $\mathbf{G}$ matrices, as used in the equations of motion for rigid bodies,

$${}^{0}\boldsymbol{\omega} = {}^{0}\mathbf{G}\, \dot{\boldsymbol{\theta}}, \tag{5.20}$$

$${}^{b}\boldsymbol{\omega} = {}^{b}\mathbf{G}\, \dot{\boldsymbol{\theta}}. \tag{5.21}$$

5.2.6 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.

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

Short name for Python: **Rigid2D**

Definition of quantities:

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

The following output parameters are available as OutputVariableType in sensors and other functions:

output parameters	**symbol**	**description**
Position	${}^0\mathbf{p}_{\,\rm config} = {}^0[p_0,\, p_1,\, 0]_{\,\rm config}^{\rm T} = {}^0\mathbf{u}_{\,\rm config} + {}^0\mathbf{p}_{\,ref}$	global 3D position vector of node; $\mathbf{u}_{\rm ref} = 0$
Displacement	${}^0\mathbf{u}_{\,\rm config} = [q_0,\, q_1,\, 0]_{\rm config}^{\rm T}$	global 3D displacement vector of node
Velocity	${}^0\mathbf{v}_{\,\rm config} = [\dot q_0,\, \dot q_1,\, 0]_{\rm config}^{\rm T}$	global 3D velocity vector of node
Coordinates	$\mathbf{c}_{\rm config} = [q_0,\, q_1,\, \psi_0]_{\rm config}^{\rm T}$	coordinate vector of node, having 2 displacement coordinates and 1 angle

Coordinates_t	$\dot{\mathbf{c}}_{\text{config}} = [\dot{q}_0,\, \dot{q}_1,\, \dot{\psi}_0]^{\mathrm{T}}_{\text{config}}$	velocity coordinates vector of node
RotationMatrix	$[A_{00},\, A_{01},\, A_{02},\, A_{10},\, \ldots,\, A_{21},\, A_{22}]^{\mathrm{T}}_{\text{config}}$	vector with 9 components of the rotation matrix ${}^{0b}\mathbf{A}_{\,\text{config}}$ in row-major format, in any configuration; the rotation matrix transforms local (b) to global (0) coordinates
Rotation	$[\theta_0]^{\mathrm{T}}_{\text{config}} = [\psi_0]^{\mathrm{T}}_{\text{ref}} + [\psi_0]^{\mathrm{T}}_{\text{config}}$	vector with 1 angle around out of plane axix
AngularVelocity	${}^{0}\boldsymbol{\omega}_{\,\text{config}} = {}^{0}[\omega_0,\, \omega_1,\, \omega_2]^{\mathrm{T}}_{\,\text{config}}$	global 3D angular velocity vector of node
AngularVelocityLocal	${}^{b}\boldsymbol{\omega}_{\,\text{config}} = {}^{b}[\omega_0,\, \omega_1,\, \omega_2]^{\mathrm{T}}_{\,\text{config}}$	local (body-fixed) 3D angular velocity vector of node

Description of Item: 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 $\theta_{0\text{config}} = \psi_{0ref} + \psi_{0\text{config}}$, the rotation matrix is defined as

$$
{}^{0b}\mathbf{A}_{\,\text{config}} = \begin{bmatrix} \cos(\theta_0) & -\sin(\theta_0) & 0 \\ \sin(\theta_0) & \cos(\theta_0) & 0 \\ 0 & 0 & 1 \end{bmatrix}_{\text{config}} \tag{5.22}
$$

Example for NodeRigidBody2D: see ObjectRigidBody2D

5.2.7 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 $\mathbf{r}' = [r'_x \ \ r'_y]^T = [1 \ \ 0]^T$.

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

Short name for Python: **Point2DS1**

Definition of quantities:

The following output parameters are available as OutputVariableType in sensors and other functions:

output parameters	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)

5.2.8 NodeGenericODE2

A node containing a number of ODE2 variables; use e.g. for scalar dynamic equations (Mass1D) or for the ALECable element.

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

Definition of quantities:

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	$\dot{\mathbf{q}}_{\mathrm{ini}} = [\dot{q}_0, \ldots, \dot{q}_{nc}]_{\mathrm{ini}}^{\mathrm{T}}$	
numberOfODE2Coordinates	n_c	

The following output parameters are available as OutputVariableType in sensors and other functions:

output parameters	**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

5.2.9 NodeGenericData

A node containing a number of data (history) variables; use e.g. for contact (active set), friction or plasticity (history variable).
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

Definition of quantities:
The following output parameters are available as OutputVariableType in sensors and other functions:

output parameters	**symbol**	**description**
Coordinates		data coordinates (history variables) vector of node

5.2.10 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.

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

Short name for Python: **PointGround**

Definition of quantities:

The following output parameters are available as OutputVariableType in sensors and other functions:

output parameters	**symbol**	**description**
Position		global 3D position vector of node (=reference position)
Displacement		zero 3D vector
Velocity		zero 3D vector
Coordinates		vector of length zero
Coordinates_t		vector of length zero

5.3 Objects

5.3.1 ObjectMassPoint

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

Short name for Python: **MassPoint**

Definition of quantities:

The following output parameters are available as OutputVariableType in sensors and other functions:

output parameters	symbol	description
Position		global position vector of translated local position
Displacement		global displacement vector of center point
Velocity		global velocity vector of center point

Description of Item:

Definition of quantities:

- $m \ldots$ physicsMass
- $n0 \ldots$ node number
- $\mathbf{c}_{\mathrm{ini}} = \mathbf{c}_{n0} (= [q_0,\ q_1,\ q_2]^{\mathrm{T}}) \ldots$ displacement coordinates of body (taken from NodePoint)
- $\mathbf{f} = [f_0,\ f_1,\ f_2]^{\mathrm{T}} \ldots$ residual of all forces (loads, constraints, springs, ...)
- $\mathbf{p}_{\mathrm{ref}} = \mathbf{c}_{\mathrm{ref}} = [q_0,\ q_1,\ q_2]^{\mathrm{T}}_{\mathrm{ref}} \ldots$ reference position = reference coordinates of node
- $\mathbf{p}_{\mathrm{config}} = \mathbf{u}_{\mathrm{config}} + \mathbf{p}_{\mathrm{ref}} \ldots$ position in any configuration ($\mathbf{u}_{\mathrm{ref}} = 0$)
- $\mathbf{p}_{\mathrm{cur}} = \mathbf{u}_{\mathrm{cur}} + \mathbf{p}_{\mathrm{ref}} \ldots$ current position, equals to node's reference position + current coordinates

Equations of motion:

$$\begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{bmatrix} \begin{bmatrix} \ddot{q}_0 \\ \ddot{q}_1 \\ \ddot{q}_2 \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ f_2 \end{bmatrix}. \tag{5.23}$$

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**

$$\mathbf{J}_{pos} = \partial \mathbf{p}_{\mathrm{cur}} / \partial \mathbf{c}_{\mathrm{cur}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{5.24}$$

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

$$\mathbf{Q} = \mathbf{J}_{pos}\, {}^0\mathbf{f} \,. \tag{5.25}$$

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

5.3.2 ObjectMassPoint2D

A 2D mass point which is attached to a position-based 2D node. Equations of motion with the displacements $[u_x \;\; u_y]^T$, the mass m and the residual of all forces $[R_x \;\; R_y]^T$ are given as

$$\begin{bmatrix} m \cdot \ddot{u}_x \\ m \cdot \ddot{u}_y \end{bmatrix} = \begin{bmatrix} R_x \\ R_y \end{bmatrix}. \tag{5.26}$$

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	Index		MAXINT	node number 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

Short name for Python: **MassPoint2D**

Definition of quantities:

The following output parameters are available as OutputVariableType in sensors and other functions:

output parameters	**symbol**	**description**
Position		global position vector of translated local position
Displacement		global displacement vector of center point
Velocity		global velocity vector of center point

5.3.3 ObjectRigidBody

A 3D rigid body which is attached to a 3D rigid body node. Equations of motion with the displacements $[u_x \quad u_y \quad u_z]^T$ 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 $[R_x \quad R_y \quad R_z \quad R_{q0} \quad R_{q1} \quad R_{q2} \quad R_{q3}]^T$ are given as ...

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 mass point
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. center of mass
nodeNumber	Index		MAXINT	node number 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
graphicsData	BodyGraphicsData			Structure contains data for body visualization; data is defined in special list / dictionary structure

Short name for Python: **RigidBody**

Definition of quantities:

The following output parameters are available as OutputVariableType in sensors and other functions:

output parameters	**symbol**	**description**
Position		global position vector of rotated and translated local position
Displacement		global displacement vector of local position
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
Velocity		global velocity vector of local position
AngularVelocity		angular velocity of body
AngularVelocityLocal		local (body-fixed) 3D velocity vector of node

5.3.4 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
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 mass point
physicsInertia	UReal		0.	inertia [SI:kgm^2] of rigid body w.r.t. center of mass
nodeNumber	Index		MAXINT	node number 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
graphicsData	BodyGraphicsData			Structure contains data for body visualization; data is defined in special list / dictionary structure

Short name for Python: **RigidBody2D**

Definition of quantities:

The following output parameters are available as OutputVariableType in sensors and other functions:

output parameters	**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
Rotation		scalar rotation angle of body
AngularVelocity		angular velocity of body
RotationMatrix		rotation matrix in vector form (stored in row-major order)

Description of Item:

Definition of quantities:

- m ... physicsMass: total body mass
- J ... physicsInertia: momentinertia w.r.t. axis 2
- $n0$ ... node number
- $\mathbf{c}_{\rm ini} = \mathbf{c}_{n0} (= [q_0,\ q_1,\ \psi_2]^{\rm T}_{n0})$ ... displacement coordinates of body taken from NodeRigidBody2D
- $\mathbf{Q} = [f_0,\ f_1,\ \tau_2]$ ... residual of all generalized forces (incl. torques), e.g., loads, constraints, springs, ...

Global position of a local body-fixed position, in any configuration

$$^{0}\mathbf{p}_{\,\mathrm{config}} = {}^{0}\mathbf{p}_{\,\mathrm{ref}} + {}^{0}\mathbf{u}_{\,\mathrm{config}} + {}^{0b}\mathbf{A}_{\,\mathrm{config}}\, {}^{b}\mathbf{p} \tag{5.27}$$

Equations of motion:

$$\begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & J \end{bmatrix} \begin{bmatrix} \ddot{q}_0 \\ \ddot{q}_1 \\ \ddot{\psi}_2 \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ \tau_2 \end{bmatrix} = \mathbf{Q}. \tag{5.28}$$

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 $\mathbf{p}_{\mathrm{config}}$. With the body rotation $\theta = \theta_{2,n0}$ and the localPosition ${}^{b}\mathbf{p} = [{}^{b}p_0\,, {}^{b}p_1\,, 0]^{\mathrm{T}}$, the **position jacobian** is defined as,

$$\mathbf{J}_{pos} = \partial \mathbf{p}_{\mathrm{cur}} / \partial \mathbf{c}_{\mathrm{cur}} = \begin{bmatrix} 1 & 0 & -\sin(\theta)\,{}^{b}p_0 - \cos(\theta)\,{}^{b}p_1 \\ 0 & 1 & \cos(\theta)\,{}^{b}p_0 - \sin(\theta)\,{}^{b}p_1 \\ 0 & 0 & 0 \end{bmatrix} \tag{5.29}$$

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

$$\mathbf{Q} = \mathbf{J}_{pos}\, {}^{0}\mathbf{f} \tag{5.30}$$

Orientation-based markers can measure the rotation matrix $\mathbf{A}_{\mathrm{config}}$. The **rotation jacobian**

$$\mathbf{J}_{rot} = \partial \mathbf{p}_{\mathrm{cur}} / \partial \mathbf{c}_{\mathrm{cur}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.31}$$

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

$$\mathbf{Q} = \mathbf{J}_{rot}\, {}^{0}\boldsymbol{\tau} \tag{5.32}$$

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
testError = mbs.GetNodeOutput(node, exu.OutputVariableType.Position)[0] - 2
testError+= mbs.GetNodeOutput(node, exu.OutputVariableType.Coordinates)[2] - 0.75*np.
    pi
#final x-coordinate of position shall be 2, angle theta shall be np.pi
```

5.3.5 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 $(n \times n)$.

The item **ObjectGenericODE2** with type = 'GenericODE2' has the following parameters:

Name	type	size	default value	description
name	String		''	objects"s unique name
nodeNumbers	ArrayIndex		[]	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	PyFunctionVectorScalar2Vector		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; Example for python function with numpy stiffness matrix K: def f(t, q, q_t): return np.dot(K, q)
massMatrixUserFunction	PyFunctionMatrixScalar2Vector		0	A python user function which computes the mass matrix instead of the constant mass matrix; The function takes the time, coordinates q (without reference values) and coordinate velocities q_t; Example (academic) for python function with numpy stiffness matrix M: def f(t, q, q_t): return (q[0]+1)*M
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

Definition of quantities:

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

massMatrix	$\mathbf{M} \in \mathbb{R}^{n \times n}$	
stiffnessMatrix	$\mathbf{K} \in \mathbb{R}^{n \times n}$	
dampingMatrix	$\mathbf{D} \in \mathbb{R}^{n \times n}$	
forceVector	$\mathbf{f} \in \mathbb{R}^{n}$	
forceUserFunction	$\mathbf{f}_{user} \in \mathbb{R}^{n}$	
massMatrixUserFunction	$\mathbf{M}_{user} \in \mathbb{R}^{n \times n}$	

The following output parameters are available as OutputVariableType in sensors and other functions:

output parameters	**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 ComputeODE2RHS)

Description of Item: 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

$$n = \sum_i n_{c_i}. \tag{5.33}$$

Equations of motion:

$$\mathbf{M}\ddot{\mathbf{q}} + \mathbf{D}\dot{\mathbf{q}} + \mathbf{K}\mathbf{q} = \mathbf{f} + \mathbf{f}_{user}(t, \mathbf{q}, \dot{\mathbf{q}}) \tag{5.34}$$

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

In case that a user mass matrix is specified, Eq. (5.34) is replaced with

$$\mathbf{M}_{user}(t, \mathbf{q}, \dot{\mathbf{q}})\ddot{\mathbf{q}} + \mathbf{D}\dot{\mathbf{q}} + \mathbf{K}\mathbf{q} = \mathbf{f} + \mathbf{f}_{user}(t, \mathbf{q}, \dot{\mathbf{q}}) \tag{5.35}$$

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

5.3.6 ObjectANCFCable2D

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.
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 $\varepsilon = \|\mathbf{r}'\| - 1$ 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
nodeNumbers	Index2		[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

Short name for Python: **Cable2D**

Definition of quantities:
The following output parameters are available as OutputVariableType in sensors and other functions:

output parameters	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)

5.3.7 ObjectALEANCFCable2D

A 2D cable finite element using 2 nodes of type NodePoint2DSlope1 and a axially moving coordinate of type NodeGenericODE2; the element has 8+1 coordinates and uses cubic polynomials for position interpolation; the element in addition to ANCFCable2D adds an Eulerian axial velocity by the GenericODE2 coordiante
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 $\rho A L$
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 $\varepsilon = \lvert\mathbf{r}'\rvert - 1$ 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
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	Index3		[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

Short name for Python: **ALECable2D**

Definition of quantities:
The following output parameters are available as OutputVariableType in sensors and other functions:

output parameters	**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)

5.3.8 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.

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
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

Definition of quantities:

The following output parameters are available as OutputVariableType in sensors and other functions:

output parameters	**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)

5.3.9 ObjectConnectorSpringDamper

An simple spring-damper element with additional force; connects to position-based markers.
Requested marker type = Marker::Position
The item **ObjectConnectorSpringDamper** with type = 'ConnectorSpringDamper' has the following parameters:

Name	type	size	default value	description
name	String		”	connector"s unique name
markerNumbers	ArrayIndex		[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
activeConnector	bool		True	flag, which determines, if the connector is active; used to deactivate (temorarily) a connector or constraint
springForceUserFunction	PyFunctionScalar6		0	A python function which defines the spring force with parameters (time, deltaL, deltaL_t, Real stiffness, Real damping, Real springForce); the parameters are provided to the function using the current values of the SpringDamper object; The python function will only be evaluated, if activeConnector is true, otherwise the SpringDamper is inactive; Example for python function: def f(t, u, v, k, d, F0): return k*u + d*v + F0
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.]	RGB connector color; if R==-1, use default color

Short name for Python: **SpringDamper**

Definition of quantities:

The following output parameters are available as OutputVariableType in sensors and other functions:

output parameters	symbol	description
Distance		distance between both points
Displacement		relative displacement between both points
Velocity		relative velocity between both points
Force		spring-damper force

Description of Item:

Definition of quantities:

input parameter	symbol	description
referenceLength	L_0	
stiffness	k	
damping	d	
force	f_a	additional force (e.g., actuator force)
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 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

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

$$\mathbf{v}_f = \frac{1}{L}\Delta^0\mathbf{p} \tag{5.36}$$

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

$$f_{SD} = k \cdot (L - L_0) + d \cdot \Delta^0\mathbf{v}^{\mathrm{T}}\mathbf{v}_f + f_a \tag{5.37}$$

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

$$f_{SD} = \mathrm{UF}(t, L - L_0, \Delta^0\mathbf{v}^{\mathrm{T}}\mathbf{v}_f, k, d, f_a) \tag{5.38}$$

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

$$\mathbf{f} = f_{SD}\mathbf{v}_f \tag{5.39}$$

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
testError = mbs.GetNodeOutput(node, exu.OutputVariableType.Position)[0] -
    0.9736596225944887
```

5.3.10 ObjectConnectorCartesianSpringDamper

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

Requested marker type = Marker::Position

The item **ObjectConnectorCartesianSpringDamper** with type = 'ConnectorCartesianSpringDamper' has the following parameters:

Name	type	size	default value	description
name	String		"	connector"s unique name
markerNumbers	ArrayIndex		[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	PyFunctionVector3DScalar5Vector3D		0	A python function which computes the 3D force vector between the two marker points, if activeConnector=True; The function takes the relative displacement (3D) vector (m1.position-m0.position, etc.) and the relative velocity vector (3D), the spring striffness vector 3D, damping and offset parameter vectors (3D): f(time, displacement, velocity, stiffness, damping, offset); Example for python function: def f(t, u, v, k, d, offset): return [u[0]*k[0],u[1]*k[1],u[2]*k[2]]
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.]	RGB connector color; if R==-1, use default color

Short name for Python: **CartesianSpringDamper**

Definition of quantities:

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 parameters are available as OutputVariableType in sensors and other functions:

output parameters	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

Description of Item:

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

Connector equations: Displacement between marker m0 to marker m1 positions,

$$\Delta^0\mathbf{p} = {}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0} \tag{5.40}$$

and relative velocity,

$$\Delta^0\mathbf{v} = {}^0\mathbf{v}_{m1} - {}^0\mathbf{v}_{m0} \tag{5.41}$$

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

$$\mathbf{f}_{SD} = \left(\mathbf{k} \cdot (\Delta^0\mathbf{p} - \mathbf{v}_{\mathrm{off}}) + \mathbf{d}\Delta^0\mathbf{v}\right) \tag{5.42}$$

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

$$\mathbf{f}_{SD} = \mathrm{UF}(t, \Delta^0\mathbf{p}, \Delta^0\mathbf{v}, \mathbf{k}, \mathbf{d}, \mathbf{v}_{\mathrm{off}}) \tag{5.43}$$

if `activeConnector = False`, $\mathbf{f}_{SD}$ is set to zero. **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
testError = mbs.GetNodeOutput(nMass, exu.OutputVariableType.Displacement)[1] -
    (-0.000999999999997058)
```

5.3.11 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.)
Requested marker type = (Marker::Type)((Index)Marker::Position + (Index)Marker::Orientation)
The item **ObjectConnectorRigidBodySpringDamper** with type = 'ConnectorRigidBodySpringDamper' has the following parameters:

Name	type	size	default value	description
name	String		"	connector"s unique name
markerNumbers	ArrayIndex		[MAXINT, MAXINT]	list of markers used in connector
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
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.]	RGB connector color; if R==-1, use default color

Short name for Python: **RigidBodySpringDamper**

Definition of quantities:

The following output parameters are available as OutputVariableType in sensors and other functions:

output parameters	symbol	description
DisplacementLocal		relative displacement in local marker0 coordinates
VelocityLocal		relative translational velocity in local marker0 coordinates
Rotation		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		relative angular velocity in local marker0 coordinates
ForceLocal		joint force in local marker0 coordinates
TorqueLocal		joint torque in in local marker0 coordinates

Description of Item:

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

output variables	**symbol**	**formula**
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	${}^{J0}\Delta\omega$	$\left({}^{0,m0}\mathbf{A}\;{}^{m0,J0}\mathbf{A}\right)^{\mathrm{T}}\left({}^{0,m1}\mathbf{A}\;{}^{m1}\omega - {}^{0,m0}\mathbf{A}\;{}^{m0}\omega\right)$
Rotation	${}^{J0}\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2]$	angles retrieved from relative rotation matrix, ...
ForceLocal	${}^{J0}\mathbf{f}$	see below
TorqueLocal	${}^{J0}\mathbf{m}$	see below

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

$$\begin{bmatrix} {}^{J0}\mathbf{f}_{SD} \\ {}^{J0}\mathbf{m}_{SD} \end{bmatrix} = \mathbf{k}\left(\begin{bmatrix} {}^{J0}\Delta\mathbf{p} \\ {}^{J0}\boldsymbol{\theta} \end{bmatrix} - {}^{J0}\mathbf{v}_{\mathrm{off}}\right) + \mathbf{d}\begin{bmatrix} {}^{J0}\Delta\mathbf{v} \\ {}^{J0}\Delta\omega \end{bmatrix} \tag{5.44}$$

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.

5.3.12 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.
Requested marker type = Marker::Coordinate
The item **ObjectConnectorCoordinateSpringDamper** with type = 'ConnectorCoordinateSpringDamper' has the following parameters:

Name	type	size	default value	description
name	String		''	connector''s unique name
markerNumbers	ArrayIndex		[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	PyFunctionScalar8		0	A python function which defines the spring force with 8 parameters, see equations section; Example for python function: def f(t, u, v, k, d, offset, frictionForce, frictionProportionalZone): return k*(u-offset) + d*v
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.]	RGB connector color; if R==-1, use default color

Short name for Python: **CoordinateSpringDamper**

Definition of quantities:

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

The following output parameters are available as OutputVariableType in sensors and other functions:

output parameters	symbol	description
Displacement	Δq	relative scalar displacement of marker coordinates
Velocity	Δv	difference of scalar marker velocity coordinates
Force	f_{SD}	scalar spring force

Description of Item:

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}	

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

$$\Delta q = q_{m1} - q_{m0} \tag{5.45}$$

and relative velocity,

$$\Delta v = v_{m1} - v_{m0} \tag{5.46}$$

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

$$f_{\mathrm{friction}} = \begin{cases} \mathrm{Sgn}(\Delta v) \cdot f_\mu & \text{if} \quad |\Delta v| \geq v_\mu \\ \dfrac{\Delta v}{v_\mu} f_\mu & \text{if} \quad |\Delta v| < v_\mu \end{cases} \tag{5.47}$$

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

$$f_{SD} = k\,(\Delta q - l_{\mathrm{off}}) + d \cdot \Delta v + f_{\mathrm{friction}} \tag{5.48}$$

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

$$f_{SD} = \mathrm{UF}(t, \Delta q, \Delta v, k, d, l_{\mathrm{off}}, f_\mu, v_\mu) \tag{5.49}$$

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

```
def springForce(t, u, v, k, d, offset, frictionForce, frictionProportionalZone):
    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
testError = mbs.GetNodeOutput(nMass, exu.OutputVariableType.Displacement)[0] -
    0.0019995158325691875
```

5.3.13 ObjectConnectorDistance

Connector which enforces constant or prescribed distance between two bodies/nodes.
Requested marker type = Marker::Position
The item **ObjectConnectorDistance** with type = 'ConnectorDistance' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints"s unique name
markerNumbers	ArrayIndex		[MAXINT, MAXINT]	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.]	RGB connector color; if R==-1, use default color

Short name for Python: **DistanceConstraint**

Definition of quantities:

input parameter	symbol	description see tables above
markerNumbers	$[m0, m1]^{\mathrm{T}}$	
distance	d_0	

The following output parameters are available as OutputVariableType in sensors and other functions:

output parameters	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

Description of Item: **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

Algebraic constraint equations: If `activeConnector = True`, the index 3 algebraic equation reads

$$\left|{}^0\Delta\mathbf{p}\right| - d_0 = 0 \tag{5.50}$$

The index 2 (velocity level) algebraic equation reads

$$\left(\frac{{}^0\Delta\mathbf{p}}{\left|{}^0\Delta\mathbf{p}\right|}\right)^{\mathrm{T}} \Delta\mathbf{v} = 0 \tag{5.51}$$

if `activeConnector = False`, the algebraic equation reads

$$\lambda_0 = 0 \tag{5.52}$$

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
testError = mbs.GetNodeOutput(nMass, exu.OutputVariableType.Position)[0] -
    (-0.9845225086606828)
```

5.3.14 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.
Requested marker type = Marker::Coordinate
The item **ObjectConnectorCoordinate** with type = 'ConnectorCoordinate' has the following parameters:

Name	**type**	**size**	**default value**	**description**
name	String		"	constraints"s unique name
markerNumbers	ArrayIndex		[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	PyFunctionScalar2		0	A python function which defines the time-dependent offset; it is highly RECOMMENDED to use sufficiently smooth functions, having consistent initial offsets with initial configuration of bodies, zero or compatible initial offset-velocity, and no accelerations; Example for python function: def UF(t, l_offset): return l_offset*(1-np.cos(t*10*2*np.pi))
offsetUserFunction_t	PyFunctionScalar2		0	time derivative of offsetUserFunction; needed for "velocityLevel=True", or for index2 time integration and for computation of initial accelerations in SecondOrderImplicit integrators
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.]	RGB connector color; if R==-1, use default color

Short name for Python: **CoordinateConstraint**

Definition of quantities:

input parameter	**symbol**	**description see tables above**
markerNumbers	$[m0, m1]^{\mathrm{T}}$	
offset	l_{off}	
factorValue1	k_{m1}	
offsetUserFunction	$\mathrm{UF}(t, l_{\mathrm{off}})$	
offsetUserFunction_t	$\mathrm{UF}_t(t, l_{\mathrm{off}})$	

The following output parameters are available as OutputVariableType in sensors and other functions:

output parameters	**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)

Description of Item: **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	$\Delta v = v_{m1} - v_{m0}$	

Algebraic constraint equations: If `activeConnector = True`, the index 3 algebraic equation reads

$$\mathbf{c}(q_{m0}, q_{m1}) = k_{m1} \cdot q_{m1} - q_{m0} - l_{\mathrm{off}} = 0 \tag{5.53}$$

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

$$\mathbf{c}(q_{m0}, q_{m1}) = k_{m1} \cdot q_{m1} - q_{m0} - \mathrm{UF}(t, l_{\mathrm{off}}) = 0 \tag{5.54}$$

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

$$\dot{\mathbf{c}}(\dot{q}_{m0}, \dot{q}_{m1}) = k_{m1} \cdot \dot{q}_{m1} - \dot{q}_{m0} - d = 0 \tag{5.55}$$

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

$$\dot{\mathbf{c}}(\dot{q}_{m0}, \dot{q}_{m1}) = k_{m1} \cdot \dot{q}_{m1} - \dot{q}_{m0} - \mathrm{UF}_t(t, l_{\mathrm{off}}) = 0 \tag{5.56}$$

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:

$$\mathbf{c}(\lambda_0) = \lambda_0 = 0 \tag{5.57}$$

Example for ObjectConnectorCoordinate:

```
def OffsetUF(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
testError = mbs.GetNodeOutput(nMass, exu.OutputVariableType.Displacement)[0] -
    0.049999999999272404
```

5.3.15 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.
Requested marker type = Marker::Coordinate
The item **ObjectContactCoordinate** with type = 'ContactCoordinate' has the following parameters:

Name	type	size	default value	description
name	String		"	connector"s unique name
markerNumbers	ArrayIndex		[MAXINT, MAXINT]	markers define contact gap
nodeNumber	Index		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.]	RGB connector color; if R==-1, use default color

Definition of quantities:

5.3.16 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.

Requested marker type = Marker::_None

The item **ObjectContactCircleCable2D** with type = 'ContactCircleCable2D' has the following parameters:

Name	**type**	**size**	**default value**	**description**
name	String		"	connector"s unique name
markerNumbers	ArrayIndex		[MAXINT, MAXINT]	markers define contact gap
nodeNumber	Index		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 length of the beam axis; 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 length of the beam axis; 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.]	RGB connector color; if R==-1, use default color

Definition of quantities:

5.3.17 ObjectContactFrictionCircleCable2D

A very specialized penalty-based contact/friction 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 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.

Requested marker type = Marker::_None

The item **ObjectContactFrictionCircleCable2D** with type = 'ContactFrictionCircleCable2D' has the following parameters:

Name	**type**	**size**	**default value**	**description**
name	String		''	connector''s unique name
markerNumbers	ArrayIndex		[MAXINT, MAXINT]	markers define contact gap
nodeNumber	Index		MAXINT	node number of a NodeGenericData with $3 \times$ 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 length of the beam axis; 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 length of the beam axis; 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.]	RGB connector color; if R==-1, use default color

Definition of quantities:

5.3.18 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.
Requested marker type = Marker::_None
The item **ObjectJointSliding2D** with type = 'JointSliding2D' has the following parameters:

Name	**type**	**size**	**default value**	**description**
name	String		"	constraints"s unique name
markerNumbers	ArrayIndex		[MAXINT, MAX-INT]	marker0: position-marker of mass point or rigid body; marker1: 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	ArrayIndex		[]	these markers are used to update marker1, if the sliding position exceeds the current cable"s range; the markers must be sorted such that marker(i) at x=cable.length is equal to marker(i+1) at x=0
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), ...
nodeNumber	Index		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.]	RGB connector color; if R==-1, use default color

Short name for Python: **SlidingJoint2D**

Definition of quantities:

The following output parameters are available as OutputVariableType in sensors and other functions:

output parameters	**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)

Description of Item:

Definition of quantities:

input parameter	**symbol**	**description**
nodeNumber	n_{GD}	node number of generic data node
markerNumbers[0]	$m0$	position-marker of mass point or rigid body
markerNumbers[1]	$m1$	marker to a Cable2D element, which is **updated** in every PostNewtonStep; if the sliding body ($m0$) is in the range of all sliding cable elements, $m1$ contains the current marker number, which is active for the sliding joint
slidingMarkerNumbers	$[m_{s0}, \ldots, m_{sn}]^{\mathrm{T}}$	a list of sn (global) marker numbers which are are used to update marker1
slidingMarkerOffsets	$[d_{s0}, \ldots, d_{sn}]$	a list of sn scalar offsets, which represent the (reference arc) length of all previous sliding cable elements

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

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

$$s_{el} = s + x_{data1} - d_{m1} = s_g - d_{m1}. \tag{5.58}$$

The vector (=difference; error) between the marker $m0$ and the marker $m1$ (=$\mathbf{r}_{ANCF}$) positions reads

$${}^{0}\Delta\mathbf{p} = {}^{0}\mathbf{r}_{ANCF} - {}^{0}\mathbf{p}_{m0} \tag{5.59}$$

The vector (=difference; error) between the marker $m0$ and the marker $m1$ velocities reads

$${}^{0}\Delta\mathbf{v} = {}^{0}\dot{\mathbf{r}}_{ANCF} - {}^{0}\mathbf{v}_{m0} \tag{5.60}$$

Algebraic constraint equations (classicalFormulation=True): The 2D sliding joint is implemented having 3 equations, using the special algebraic coordinates $\mathbf{z}$. The algebraic equations read

$$
\begin{aligned}
{}^{0}\Delta\mathbf{p}^{\mathrm{T}} &= \mathbf{0}, && \text{... 2 index 3 equations, ensuring the sliding body to stay at the cable} && (5.61)\\
[\lambda_0, \lambda_1] \cdot {}^{0}\mathbf{r}'_{ANCF} &= 0, && \text{... 1 index 1 equation, ensuring the force in sliding direction = 0} && (5.62)\\
& && && (5.63)
\end{aligned}
$$

No index 2 case exists, because no time derivative exists for s_{el}. The jacobian matrices for algebraic and ODE2 coordinates read

$$
J_{AE} = \begin{bmatrix} 0 & 0 & r'_0 \\ 0 & 0 & r'_1 \\ r'_0 & r'_1 & r''_0\lambda_0 + r''_1\lambda_1 \end{bmatrix} \quad (5.64)
$$

$$
J_{ODE2} = \begin{bmatrix} -J_{pos,m0} & \mathbf{S}(s_{el}) \\ \mathbf{0}^{\mathrm{T}} & [\lambda_0,\, \lambda_1] \cdot \mathbf{S}'(s_{el}) \end{bmatrix} \quad (5.65)
$$

if `activeConnector = False`, the algebraic equations are changed to:

$$
\begin{aligned}
\lambda_0 &= 0, && (5.66)\\
\lambda_1 &= 0, && (5.67)\\
s &= 0 && (5.68)
\end{aligned}
$$

Algebraic 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

$$
\begin{aligned}
\lambda_0 &= 0, && \text{... this equation is not necessary, but can be used for switching to other modes} && (5.69)\\
{}^{0}\Delta\mathbf{p}^{\mathrm{T}}\, {}^{0}\mathbf{n} &= 0, && \text{... equation ensures that sliding body stays at cable centerline; index3 equation} && (5.70)\\
{}^{0}\Delta\mathbf{p}^{\mathrm{T}}\, {}^{0}\mathbf{r}'_{ANCF} &= 0. && \text{... resolves the sliding coordinate } s\text{; index1 equation!} && (5.71)
\end{aligned}
$$

In the index 2 case, the second equation reads

$$
{}^{0}\Delta\mathbf{v}^{\mathrm{T}}\, {}^{0}\mathbf{n} + {}^{0}\Delta\mathbf{p}^{\mathrm{T}}\, {}^{0}\dot{\mathbf{n}} = 0 \quad (5.72)
$$

if `activeConnector = False`, the algebraic equations are changed to:

$$
\begin{aligned}
\lambda_0 &= 0, && (5.73)\\
\lambda_1 &= 0, && (5.74)\\
s &= 0 && (5.75)
\end{aligned}
$$

Post Newton Step: After the Newton solver has converged, a PostNewtonStep is performed for the element, which updates the marker $m1$ index if necessary.

$$
\begin{aligned}
s_{el} < 0 \quad &\rightarrow \quad x_{data0} \mathrel{-}= 1 \\
s_{el} > L \quad &\rightarrow \quad x_{data0} \mathrel{+}= 1 && (5.76)
\end{aligned}
$$

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,

$$
x_{data1} \mathrel{+}= s. \quad (5.77)
$$

Examples: see TestModels!

5.3.19 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.
Requested marker type = Marker::_None
The item **ObjectJointALEMoving2D** with type = 'JointALEMoving2D' has the following parameters:

Name	**type**	**size**	**default value**	**description**
name	String		"	constraints"s unique name
markerNumbers	ArrayIndex		[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	ArrayIndex		[]	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	ArrayIndex		[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.]	RGB connector color; if R==-1, use default color

Short name for Python: **ALEMovingJoint2D**

Definition of quantities:

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 parameters are available as OutputVariableType in sensors and other functions:

output parameters	**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)

Description of Item:

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 coordinates	$\mathbf{z} = [\lambda_0,\, \lambda_1]^{\mathrm{T}}$	algebraic coordinates composed of Lagrange multipliers λ_0 and λ_1, according to the algebraic equations

The element sliding coordinate (in the local coordinates of the current sliding element) is computed from the ALE coordinate

$$s_{el} = q_{ALE} + s_{off} - d_{m1} = s_g - d_{m1}. \tag{5.78}$$

The vector (=difference; error) between the marker $m0$ and the marker $m1$ (=$\mathbf{r}_{ANCF}$) positions reads

$${}^{0}\Delta\mathbf{p} = {}^{0}\mathbf{r}_{ANCF} - {}^{0}\mathbf{p}_{m0} \tag{5.79}$$

The vector (=difference; error) between the marker $m0$ and the marker $m1$ velocities reads

$${}^{0}\Delta\mathbf{v} = {}^{0}\mathbf{v}_{ANCF} - {}^{0}\mathbf{v}_{m0} \tag{5.80}$$

Algebraic constraint equations: The 2D sliding joint is implemented having 2 equations, using the Lagrange multipliers $\mathbf{z}$. The algebraic (index 3) equations read

$${}^{0}\Delta\mathbf{p} = 0 \tag{5.81}$$

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

$${}^{0}\Delta\mathbf{v} = 0 \tag{5.82}$$

If `usePenalty = True`, the algebraic equations are changed to:

$${}^{0}\Delta\mathbf{p} - \frac{1}{k}\mathbf{z} = 0. \tag{5.83}$$

If `activeConnector = False`, the algebraic equations are changed to:

$$\lambda_0 = 0, \tag{5.84}$$
$$\lambda_1 = 0. \tag{5.85}$$

Post Newton Step: After the Newton solver has converged, a PostNewtonStep is performed for the element, which updates the marker $m1$ index if necessary.

$$\begin{aligned} s_{el} < 0 \quad &\rightarrow \quad x_{data0} \mathrel{-}= 1 \\ s_{el} > L \quad &\rightarrow \quad x_{data0} \mathrel{+}= 1 \end{aligned} \tag{5.86}$$

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,

$$x_{data1} \mathrel{+}= s. \tag{5.87}$$

Examples: see TestModels!

5.3.20 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

Requested marker type = (Marker::Type)((Index)Marker::Position + (Index)Marker::Orientation)

The item **ObjectJointGeneric** with type = 'JointGeneric' has the following parameters:

Name	**type**	**size**	**default value**	**description**
name	String		''	constraints''s unique name
markerNumbers	ArrayIndex	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; 0=free, 1=constrained
rotationMarker0	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	local rotation matrix for marker 0; translation and rotation axes for marker0 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 1; translation and rotation axes for marker1 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	PyFunctionVector6DScalarVector6D		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 (t, offsetUserFunctionParameters); the offset represents the current value of the object; it is highly RECOMMENDED to use sufficiently smooth functions, having consistent initial offsets with initial configuration of bodies, zero or compatible initial offset-velocity, and no accelerations; Example for python function: def f(t, offsetUserFunctionParameters): return [offsetUserFunctionParameters[0]*(1 - np.cos(t*10*2*np.pi)), 0,0,0,0,0]
offsetUserFunction_t	PyFunctionVector6DScalarVector6D		0	time derivative of offsetUserFunction using the same parameters; needed for "velocityLevel=True", or for index2 time integration and for computation of initial accelerations in SecondOrderImplicit integrators
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.]	RGB connector color; if R==-1, use default color

Short name for Python: **GenericJoint**

Definition of quantities:

5.3.21 ObjectJointRevolute2D

A revolute joint in 2D; constrains the absolute 2D position of two points given by PointMarkers or RigidMarkers
Requested marker type = Marker::Position
The item **ObjectJointRevolute2D** with type = 'JointRevolute2D' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints"s unique name
markerNumbers	ArrayIndex		[MAXINT, MAX-INT]	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.]	RGB connector color; if R==-1, use default color

Short name for Python: **RevoluteJoint2D**

Definition of quantities:

5.3.22 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

$$(\mathbf{p}_1 - \mathbf{p}_0)^T \cdot \mathbf{A}_1 \cdot \mathbf{n}_1 = 0 \tag{5.88}$$

$$(\mathbf{A}_0 \cdot \mathbf{t}_0)^T \cdot \mathbf{A}_1 \cdot \mathbf{n}_1 = 0 \tag{5.89}$$

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.
Requested marker type = (Marker::Type)(Marker::Position + Marker::Orientation)
The item **ObjectJointPrismatic2D** with type = 'JointPrismatic2D' has the following parameters:

Name	**type**	**size**	**default value**	**description**
name	String		''	constraints''s unique name
markerNumbers	ArrayIndex		[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.]	RGB connector color; if R==-1, use default color

Short name for Python: **PrismaticJoint2D**

Definition of quantities:

5.4 Markers

5.4.1 MarkerBodyMass

A marker attached to the body mass; use this marker to apply a body-load (e.g. gravitational force).
The item **MarkerBodyMass** with type = 'BodyMass' has the following parameters:

Name	**type**	**size**	**default value**	**description**
name	String		''	marker''s unique name
bodyNumber	Index		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

Definition of quantities:

5.4.2 MarkerBodyPosition

A position body-marker attached to local position (x,y,z) of the body.
The item **MarkerBodyPosition** with type = 'BodyPosition' has the following parameters:

Name	**type**	**size**	**default value**	**description**
name	String		”	marker''s unique name
bodyNumber	Index		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

Definition of quantities:

5.4.3 MarkerBodyRigid

A rigid-body (position+orientation) body-marker attached to local position (x,y,z) of the body.
The item **MarkerBodyRigid** with type = 'BodyRigid' has the following parameters:

Name	**type**	**size**	**default value**	**description**
name	String		''	marker''s unique name
bodyNumber	Index		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

Definition of quantities:

5.4.4 MarkerNodePosition

A node-Marker attached to a position-based node.
The item **MarkerNodePosition** with type = 'NodePosition' has the following parameters:

Name	type	size	default value	description
name	String		''	marker''s unique name
nodeNumber	Index		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

Definition of quantities:

5.4.5 MarkerNodeRigid

A rigid-body (position+orientation) node-marker attached to a rigid-body node.
The item **MarkerNodeRigid** with type = 'NodeRigid' has the following parameters:

Name	**type**	**size**	**default value**	**description**
name	String		"	marker"s unique name
nodeNumber	Index		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

Definition of quantities:

5.4.6 MarkerNodeCoordinate

A node-Marker attached to a ODE2 coordinate of a node; for other coordinates (ODE1,...) other markers need to be defined.

The item **MarkerNodeCoordinate** with type = 'NodeCoordinate' has the following parameters:

Name	type	size	default value	description
name	String		”	marker''s unique name
nodeNumber	Index		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

Definition of quantities:

5.4.7 MarkerBodyCable2DShape

A special Marker attached to a 2D ANCF beam finite element with cubic interpolation and 8 coordinates.
The item **MarkerBodyCable2DShape** with type = 'BodyCable2DShape' has the following parameters:

Name	**type**	**size**	**default value**	**description**
name	String		''	marker''s unique name
bodyNumber	Index		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

Definition of quantities:

5.4.8 MarkerBodyCable2DCoordinates

A special Marker attached to the coordinates of a 2D ANCF beam finite element with cubic interpolation.

The item **MarkerBodyCable2DCoordinates** with type = 'BodyCable2DCoordinates' has the following parameters:

Name	type	size	default value	description
name	String		”	marker"s unique name
bodyNumber	Index		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

Definition of quantities:

5.5 Loads

5.5.1 LoadForceVector

Load with (3D) force vector; attached to position-based marker.
Requested marker type = Marker::Position
The item **LoadForceVector** with type = 'ForceVector' has the following parameters:

Name	type	size	default value	description
name	String		''	load''s unique name
markerNumber	Index		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	PyFunctionVector3DScalarVector3D		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(t, loadVector): return [loadVector[0]*np.sin(t*10*2*3.1415),0,0]
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

Short name for Python: **Force**

Definition of quantities:

5.5.2 LoadTorqueVector

Load with (3D) torque vector; attached to rigidbody-based marker.
Requested marker type = Marker::Orientation
The item **LoadTorqueVector** with type = 'TorqueVector' has the following parameters:

Name	**type**	**size**	**default value**	**description**
name	String		"	load''s unique name
markerNumber	Index		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	PyFunctionVector3DScalarVector3D		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(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

Short name for Python: **Torque**

Definition of quantities:

5.5.3 LoadMassProportional

Load attached to BodyMass-based 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).

Requested marker type = Marker::BodyMass

The item **LoadMassProportional** with type = 'MassProportional' has the following parameters:

Name	type	size	default value	description
name	String		''	load''s unique name
markerNumber	Index		MAXINT	marker''s number to which load is applied
loadVector	Vector3D		[0.,0.,0.]	vector-valued load [SI:N/kg = m/s^2]
loadVectorUserFunction	PyFunctionVector3DScalarVector3D		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(t, loadVector): return [loadVector[0]*np.sin(t*10*2*3.1415),0,0]
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

Short name for Python: **Gravity**

Definition of quantities:

5.5.4 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.

Requested marker type = Marker::Coordinate

The item **LoadCoordinate** with type = 'Coordinate' has the following parameters:

Name	type	size	default value	description
name	String		''	load''s unique name
markerNumber	Index		MAXINT	marker''s number to which load is applied
load	Real		0.	scalar load [SI:N]
loadUserFunction	PyFunctionScalar2		0	A python function which defines the time-dependent load with parameters (Real t, Real 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(t, load): return load*np.sin(t*10*2*3.1415)
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

Definition of quantities:

5.6 Sensors

5.6.1 SensorNode

A sensor attached to a node. The sensor measures OutputVariables and outputs values into a file, showing 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	Index		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
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

Definition of quantities:

5.6.2 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 time, sensorValue[0], sensorValue[1], ... A user function can be attached to post-process 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	Index		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
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

Definition of quantities:

5.6.3 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 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	Index		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
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)!

Definition of quantities:

5.6.4 SensorLoad

A sensor attached to a load. The sensor measures the load values and outputs values into a file, showing 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	Index		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
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

Definition of quantities:

5.7 GraphicsData

Some items may include a 'graphicsData' structure. GraphicsData contains a list of graphicsData items, i.e. graphicsData = [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
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 ≤ points.size()

Examples of `GraphicsData` can be found in the Python examples and in `exudynUtilities.py`.

Chapter 6

EXUDYN Settings

This section includes the reference manual for settings which are available in the python interface, e.g. simulation settings, visualization settings, and others.

6.1 Simulation settings

This section includes hierarchical structures for simulation settings, e.g., time integration, static solver, Newton iteration and solution file export.

6.1.1 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
appendToFile	bool		False	flag (true/false); if true, solution 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
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

exportVelocities	bool		True	solution is written as displacements, velocities[, accelerations] [,algebraicCoordinates] [,DataCoordinates]
exportAccelerations	bool		True	solution is written as displacements, [velocities,] accelerations [,algebraicCoordinates] [,DataCoordinates]
exportAlgebraicCoordinates	bool		True	solution is written as displacements, [velocities,] [accelerations,], algebraicCoordinates (=Lagrange multipliers) [,DataCoordinates]
exportDataCoordinates	bool		True	solution is written as displacements, [velocities,] [accelerations,] [,algebraicCoordinates (=Lagrange multipliers)] ,DataCoordinates
coordinatesSolutionFileName	FileName		'coordinatesSolution.txt'	filename and (relative) path of solution file containing all coordinates versus time
solverInformationFileName	FileName		'solverInormation.txt'	filename and (relative) path of text file showing detailed information during solving; detail level according to yourSolver.verboseModeFile
solutionInformation	String		''	special information added to header of solution file (e.g. parameters and settings, modes, ...)
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

6.1.2 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 ($\varepsilon = \varepsilon_{\mathrm{relative}} * max(q_{min}, \lvert q_i + [q_i^{Ref}]\rvert)$, 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

6.1.3 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 change of solution 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)
maxDiscontinuousIterations	Index		5	maximum number of discontinuous (post Newton) iterations

ignoreMaxDiscontinuousIterations	bool		True	continue solver if maximum number of discontinuous (post Newton) iterations is reached (ignore tolerance)
discontinuousIterationTolerance	UReal		1	absolute tolerance for discontinuous (post Newton) iterations; the errors represent absolute residuals and can be quite high
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

6.1.4 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

6.1.5 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
startTime	UReal		0	start time of time integration (usually set to zero)
endTime	UReal		1	end time of time integration
numberOfSteps	UInt		100	number of steps in time integration; stepsize is computed from (endTime-startTime)/numberOfSteps
adaptiveStep	bool		True	true: use step reduction if step fails; false: constant step size
minimumStepSize	UReal		1e-8	lower limit of time step size, before integrator stops
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
generalizedAlpha	GeneralizedAlphaSettings			parameters for generalized-alpha, implicit trapezoidal rule or Newmark (options only apply for these methods)
preStepPyExecute	String		”	Python code to be executed prior to every step and after last step, e.g. for postprocessing

6.1.6 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)
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 loadStepGeometric
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
preStepPyExecute	String		”	Python code to be executed prior to every load step and after last step, e.g. for postprocessing

6.1.7 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)

6.2 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.

6.2.1 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 if not overwritten
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.,1.,1.,1.]	red, green, blue and alpha values for background of render window (white=[1,1,1,1]; black = [0,0,0,1])
coordinateSystemSize	float		0.4	size of coordinate system relative to screen
drawCoordinateSystem	bool		True	false = no coordinate system shown
showComputationInfo	bool		True	false = no info about computation (current time, solver, etc.) shown
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		8	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)

6.2.2 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		5000	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

6.2.3 VSettingsOpenGL

OpenGL settings for 2D and 2D rendering.
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 show-FaceEdges=true gives are wire frame representation
showFaceEdges	bool	1	False	show edges of faces; using the options showFaces=false and show-FaceEdges=true gives are wire frame representation
shadeModelSmooth	bool	1	True	true: turn on smoothing for shaders, which uses vertex normals to smooth surfaces
materialSpecular	Float4	4	[1.,1.,1.,1.]	4f specular color of material
materialShininess	float	1	60.	shininess of material
enableLight0	bool	1	True	turn on/off light0
light0position	Float4	4	[1.,1.,-10.,0.]	4f position vector of GL light0; 4th value should be 0, otherwise the vector obtains a special interpretation, see opengl manuals
light0ambient	float	1	0.25	ambient value of GL light0
light0diffuse	float	1	0.4	diffuse value of GL light0
light0specular	float	1	0.4	specular value of GL light0
enableLight1	bool	1	True	turn on/off light1
light1position	Float4	4	[0.,3.,2.,0.]	4f position vector of GL light1; 4th value should be 0, otherwise the vector obtains a special interpretation, see opengl manuals
light1ambient	float	1	0.25	ambient value of GL light1
light1diffuse	float	1	0.4	diffuse value of GL light1
light1specular	float	1	0.	specular value of GL light1
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

6.2.4 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
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

6.2.5 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/rame'	filename (without extension!) and (relative) path for image file(s) with consecutive numbering (e.g., frame0000.tga, frame0001.tga,...); folders must already 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

6.2.6 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
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

6.2.7 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

6.2.8 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	[0.2,0.2,1.,1.]	default cRGB olor for bodies; 4th value is
beams	VSettingsBeams			visualization settings for beams (e.g. ANCFCable or other beam elements)

6.2.9 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
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
defaultSize	float		0.1	global connector size; if -1.f, connector size is relative to maxSceneSize
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

6.2.10 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
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

6.2.11 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
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

6.2.12 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
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

6.2.13 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
window	VSettingsWindow			visualization window and interaction settings
openGL	VSettingsOpenGL			OpenGL rendering settings
contour	VSettingsContour			contour plot visualization settings
exportImages	VSettingsExportImages			settings for exporting (saving) images to files in order to create animations

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

6.3 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.

6.3.1 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
AERHS	Real		0.	time for residual evaluation of algebraic equations right-hand-side
totalJacobian	Real		0.	time for all jacobian computations
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
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

6.3.2 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
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)

6.3.3 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 adaptive step 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	if true, the step size may be adaptively controlled
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)
ToString()	String			convert iteration statistics to string; used for displayStatistics option

6.3.4 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)
discontinuousIterationsFailed	bool		False	true, if discontinuous iterations failed (may be recovered if adaptive step is active)
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

6.3.5 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
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

6.3.6 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(...)	void		mainSystem, simulationSettings	compute residual for Newton method (e.g. static or time step); store result in systemResidual

ComputeNewtonUpdate(...)	void		mainSystem, simulationSettings	compute update for currentState from newtonSolution (decrement from residual and jacobian)
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)

6.3.7 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 (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.
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
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
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(...)	void		mainSystem, simulationSettings	compute residual for Newton method (e.g. static or time step); store result in systemResidual
ComputeNewtonUpdate(...)	void		mainSystem, simulationSettings	compute update for currentState from newtonSolution (decrement from residual and jacobian)
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)

Chapter 7

3D Graphics Visualization

The 3D graphics visualization window is kept simple, but useful to see the animated results of the multibody system.

7.1 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)

7.2 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)
0 or KEYPAD '0'	reset rotation	set rotation such that the scene is oriented in the x/y plane
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

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
Q	stop simulation	simulation is stopped and cannot be recovered
X	execute command	open dialog to enter a python command (in global python scope)
V	visualization settings	open dialog to modify visualization settings
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'

Chapter 8

Solver

8.1 Jacobian computation

The computation of the global jacobian matrix is time consuming for the static solver or implicit time integration. The equations are split into 2nd order differential equations, 1st order differential equationsand algebraic equationsparts. From this structure, in the general non-symmetric case, 3×3 submatrices result for the jacobian. Every submatrix of the jacobian has a certain meaning and needs to be computed individually. Specifically, in implicit time integration the 2nd order differential equations$\times$ 2nd order differential equationsterm includes the (tangent) stiffness matrix and the mass matrix.

For efficient computation purpose, the elements provide a list of flags, which determine the dependencies as well as available (analytical) functions to compute the local (object) jacobian:

- ODE2_ODE2 … derivative of ODE2 equations with respect to ODE2 variables
- ODE2_ODE2_t … derivative of ODE2 equations with respect to ODE2_t (velocity) variables
- ODE1_ODE1 … derivative of ODE1 equations with respect to ODE1 variables (NOT YET AVAILABLE)
- AE_ODE2 … derivative of AE (algebraic) equations with respect to ODE2 variables
- AE_ODE2_t … derivative of AE (algebraic) equations with respect to ODE2_t (velocity) variables (NOT YET AVAILABLE)
- AE_ODE1 … derivative of AE (algebraic) equations with respect to ODE1 variables (NOT YET AVAILABLE)
- AE_AE … derivative of AE (algebraic) equations with respect to AE variables

If one of these flags is set (binary; e.g.ODE2_ODE2 + ODE2_ODE2_t), then the according local jacobian is computed and assembled into the global jacobian in the static or implicit dynamic solver.

Jacobians can also be supplied in analytical (function) form, which is indicated by an additional flag with the same name but an additional term '_function', e.g. 'ODE2_ODE2_function' indicates that the derivative of ODE2 equations with respect to its ODE2 coordinates is provided in an analytical form (this is the tangent stiffness matrix).

Two **object** functions are used to compute the local jacobians:

- **ComputeJacobianODE2_ODE2(Matrix& jacobian, Matrix& jacobian_ODE2_t)**: computes the `ODE2_ODE2` and `ODE2_ODE2_t` jacobians
- **ComputeJacobianAE(Matrix& jacobian, Matrix& jacobian_AE)**: computes the `AE_ODE2` and `AE_AE` jacobians of the object ITSELF

Two **connector** functions are used to compute the local jacobians, using `MarkerData`:

- **ComputeJacobianODE2_ODE2(Matrix& jacobian, Matrix& jacobian_ODE2_t, const MarkerDataStructure& markerData)**: computes the `ODE2_ODE2` and `ODE2_ODE2_t` jacobians of the connector; e.g. for spring-damper
- **ComputeJacobianAE(Matrix& jacobian, Matrix& jacobian_AE, const MarkerDataStructure& markerData)**: computes the `AE_ODE2` and `AE_AE` jacobians of the connector; e.g. for coordinate constraint

The system jacobian has the structure (2= ODE2, 1= ODE1, λ = AE; $\bar{\mathbf{f}}_2$ = according system residual including dynamic (mass matrix) terms in time integration; $\mathbf{g}_\lambda$ = algebraic equations):

$$\begin{bmatrix} \frac{\partial \bar{\mathbf{f}}_2}{\partial \mathbf{q}_2} & 0 & \left(\frac{\partial \mathbf{g}_\lambda}{\partial \mathbf{q}_2}\right)^T \\ 0 & \frac{\partial \mathbf{f}_1}{\partial \mathbf{q}_1} & \left(\frac{\partial \mathbf{g}_\lambda}{\partial \mathbf{q}_1}\right)^T \\ \frac{\partial \mathbf{g}_\lambda}{\partial \mathbf{q}_2} & \frac{\partial \mathbf{g}_\lambda}{\partial \mathbf{q}_1} & \frac{\partial \mathbf{g}_\lambda}{\partial \mathbf{q}_\lambda} \end{bmatrix} \tag{8.1}$$

Two system jacobian functions are currently available:

- **JacobianODE2RHS(temp, newton, factorODE2, factorODE2_t, jacobian_ODE2, jacobian_ODE2_t)**: compute analytical/numerical differentiation of ODE2RHS w.r.t. ODE2 and ODE2_t coordinates; if analytical/-functional version of jacobian is available and Newton flag 'useNumericalDifferentiation'=false, then the according jacobian is computed by its according function; results are 2 jacobians; the factors 'factor_ODE2' and 'factor_ODE2_t' are used to scale the two jacobians; if a factor is zero, the according jacobian is not computed.
- **JacobianAE(temp, newton, jacobian, factorODE2, velocityLevel, fillIntoSystemMatrix)**: compute constraint jacobian of AE with respect to ODE2 ('fillIntoSystemMatrix'=true: also w.r.t. [ODE1] and AE) coordinates $\rightarrow$ direct computation given by access functions; 'factorODE2' is used to scale the ODE2-part of the jacobian (to avoid postmultiplication); velocityLevel = true: velocityLevel constraints are used, if available; 'fillIntoSystemMatrix'=true: fill in both $\frac{\partial \bar{\mathbf{f}}_\lambda}{\partial \mathbf{q}_2}$, $\frac{\partial \bar{\mathbf{f}}_\lambda}{\partial \mathbf{q}_2}^T$ AND $\frac{\partial \bar{\mathbf{f}}_\lambda}{\partial \mathbf{q}_\lambda}$ at according locations into system matrix; 'fillIntoSystemMatrix'=false: (this is a temporary/WORKAROUND function):

The system jacobian functions compute the local jacobians either by means of a provided function or numerically, using the 'NumericalDifferentiation' settings of 'Newton'.

8.2 Implicit trapezoidal rule solver

This solver represents a class of solvers, which are based on the implicit trapezoidal rule. This integration includes the start value and the end value of a time step for the interpolation, thus being a trapezoidal integration rule. In some specializations, e.g. the Newmark method, the interpolation might only depend on the start value or the end value.

Most important representations of this rule:

- Trapezoidal rule (= Newmark with $\beta = \frac{1}{4}$ and $\gamma = \frac{1}{2}$)
- Newmark method
- Generalized-α method (= generalized Newmark method with additional parameters

8.3 Representation of coordinates and equations of motion

Nomenclature:

- '2' ... second order equations (usually of a mechanical system)
- '1' ... first order equations (e.g. of a controller, fluid, etc.)
- 'λ' ... algebraic equations (usually of joints)
- $\mathbf{M}$ ... mass matrix
- $\mathbf{q}_2$ ... 'displacement' coordinates of ODE2 equations
- $\dot{\mathbf{q}}_2$ ... 'velocity' coordinates of ODE2 equations
- $\ddot{\mathbf{q}}_2$ ... 'acceleration' coordinates of ODE2 equations
- $\mathbf{q}_1$ ... coordinates of ODE1 equations
- $\dot{\mathbf{q}}_1$ ... 'velocity' coordinates of ODE1 equations
- $\mathbf{q}_\lambda$ ... Lagrange multipliers
- $\mathbf{f}_2$ ... right-hand-side of ODE2 equations (except for action of joint reaction forces)
- $\mathbf{f}_1$ ... right-hand-side of ODE1 equations
- $\mathbf{g}$ ... algebraic equations
- $\mathbf{K}$ ... (tangent) stiffness matrix
- $\mathbf{D}$ ... damping/gyroscopic matrix
- h ... step size of time integration method

The equations of motion in EXUDYN are represented as

$$\begin{aligned}\mathbf{M}\ddot{\mathbf{q}}_2 + \frac{\partial \mathbf{g}}{\partial \mathbf{q}_2^{\mathrm{T}}}\mathbf{q}_\lambda &= \mathbf{f}_2(\mathbf{q}_2, \dot{\mathbf{q}}_2, t) \qquad (8.2)\\ \dot{\mathbf{q}}_1 + \frac{\partial \mathbf{g}}{\partial \mathbf{q}_1^{\mathrm{T}}}\mathbf{q}_\lambda &= \mathbf{f}_1(\mathbf{q}_1, t) \qquad (8.3)\\ \mathbf{g}(\mathbf{q}_2, \dot{\mathbf{q}}_2, \mathbf{q}_1, \mathbf{q}_\lambda, t) &= 0 \qquad (8.4)\end{aligned}$$

Note that the term $\frac{\partial \mathbf{g}}{\partial \mathbf{q}_1}\mathbf{q}_\lambda$ 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 systems, $\mathbf{f}_2$ becomes

$$\mathbf{f}_2^{lin} = \mathbf{f}^a - \mathbf{K}\mathbf{q}_2 - \mathbf{D}\dot{\mathbf{q}}_1 \qquad (8.5)$$

in which $\mathbf{f}^a$ represents applied forces and $\mathbf{K}$ and $\mathbf{D}$ become part of the system Jacobian for time integration.

8.4 Newmark method

The Newmark method obtains two parameters β and γ. The main ideas are

- Interpolate the displacements and the velocities linearly using the accelerations of the beginning of the time step (subindex '0') and the end of the time step (subindex 'T').
- Solve the system equations at the end of the time step for the unknown accelerations as well as for 1st order differential equations and algebraic equations coordinates.

We abbreviate the unknown accelerations by $\ddot{\mathbf{q}} = \mathbf{a}$ and the unknown velocities $\dot{\mathbf{q}} = \mathbf{v}$. Thus, the equations at the end of the time step read (bring all terms to LHS):

$$\begin{aligned}\mathbf{f}_2^{\mathrm{Newmark}} &= \mathbf{M}\mathbf{a}_2^T + \frac{\partial \mathbf{g}}{\partial \mathbf{q}_2^{\mathrm{T}}}\mathbf{q}_\lambda^T - \mathbf{f}_2(\mathbf{q}_2^T, \dot{\mathbf{q}}_2^T, t) = 0 \qquad (8.6)\\ \mathbf{f}_1^{\mathrm{Newmark}} &= \mathbf{v}_1^T + \frac{\partial \mathbf{g}}{\partial \mathbf{q}_1^{\mathrm{T}}}\mathbf{q}_\lambda^T - \mathbf{f}_1(\mathbf{q}_1^T, t) = 0 \qquad (8.7)\\ \mathbf{f}_\lambda^{\mathrm{Newmark}} &= \mathbf{g}(\mathbf{q}_2^T, \dot{\mathbf{q}}_2^T, \mathbf{q}_1^T, \mathbf{q}_\lambda^T, t) = 0 \qquad (8.8)\end{aligned}$$

Within Eq. (8.6), the 2nd order differential equations displacements and velocities and for 1st order differential equations coordinates are given by

$$\begin{aligned}
\mathbf{q}_2^T &= \mathbf{q}_2^0 + h\dot{\mathbf{q}}_2^0 + h^2(\frac{1}{2} - \beta)\mathbf{a}_2^0 + h^2\beta\mathbf{a}_2^T \\
\dot{\mathbf{q}}_2^T &= \dot{\mathbf{q}}_2^0 + h(1-\gamma)\mathbf{a}_2^0 + h\gamma\mathbf{a}_2^T \\
\mathbf{q}_1^T &= \mathbf{q}_1^0 + h(1-\gamma)\mathbf{v}_1^0 + h\gamma\mathbf{v}_1^T
\end{aligned} \quad (8.9)$$

The unknowns for the Newton method are

$$\mathbf{q}^{\text{Newton}} = \begin{bmatrix} \mathbf{a}_2^T \\ \mathbf{v}_1^T \\ \mathbf{q}_\lambda^T \end{bmatrix} \quad (8.10)$$

For the Newton method, we need to compute an update for the unknowns Eq. (8.10), using the known residual $\mathbf{r}_{i-1}$ and the inverse of the Jacobian $\mathbf{J}_{i-1}$ of step $i-1$,

$$\mathbf{q}_i^{\text{Newton}} = \mathbf{q}_{i-1}^{\text{Newton}} - \mathbf{J}^{-1}\left(\mathbf{q}_{i-1}^{\text{Newton}}\right)\mathbf{r}\left(\mathbf{q}_{i-1}^{\text{Newton}}\right) \quad (8.11)$$

The Jacobian has the following 3×3 structure,

$$\mathbf{J} = \begin{bmatrix} \mathbf{J}_{22} & \mathbf{J}_{21} & \mathbf{J}_{2\lambda} \\ \mathbf{J}_{12} & \mathbf{J}_{11} & \mathbf{J}_{1\lambda} \\ \mathbf{J}_{\lambda 2} & \mathbf{J}_{\lambda 1} & \mathbf{J}_{\lambda\lambda} \end{bmatrix} \quad (8.12)$$

Note that currently, all terms related to '1' are not implemented. The other terms are only evaluated in the specific jacobian computation, if according flags are set in GetAvailableJacobian(). Otherwise, the constraint needs to be implemented as object which can employ all kinds of coordinates, which do not depend on coordinates of markers.

The available Jacobians need to be rewritten in terms of the Newton unkowns (8.10), and thus read

$$\begin{aligned}
\mathbf{J}_{22} &= \frac{\partial \mathbf{f}_2^{\text{Newmark}}}{\partial \mathbf{a}_2^{\text{T}}} = \frac{\partial \mathbf{f}_2^{\text{Newmark}}}{\partial \mathbf{q}_2^{\text{T}}}\frac{\mathbf{q}_2}{\mathbf{a}_2^{\text{T}}} + \frac{\partial \mathbf{f}_2^{\text{Newmark}}}{\partial \dot{\mathbf{q}}_2^{\text{T}}}\frac{\dot{\mathbf{q}}_2}{\mathbf{a}_2^{\text{T}}} = h^2\beta\mathbf{K} + h\gamma\mathbf{D} \\
\mathbf{J}_{2\lambda} &= \frac{\partial \mathbf{f}_2^{\text{Newmark}}}{\partial \mathbf{q}_\lambda^{\text{T}}} = \frac{\partial \mathbf{g}}{\partial \mathbf{q}_2^{\text{T}}} \\
\mathbf{J}_{\lambda 2} &= \frac{\partial \mathbf{f}_\lambda^{\text{Newmark}}}{\partial \mathbf{a}_2^{\text{T}}} = \frac{\partial \mathbf{g}}{\partial \mathbf{a}_2^{\text{T}}} = \frac{\partial \mathbf{g}}{\partial \mathbf{q}_2^{\text{T}}}\frac{\mathbf{q}_2}{\mathbf{a}_2^{\text{T}}} + \frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}_2^{\text{T}}}\frac{\dot{\mathbf{q}}_2}{\mathbf{a}_2^{\text{T}}} = h^2\beta\frac{\partial \mathbf{g}}{\partial \mathbf{q}_2^{\text{T}}} + h\gamma\frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}_2^{\text{T}}} \\
\mathbf{J}_{\lambda\lambda} &= \frac{\partial \mathbf{f}_\lambda^{\text{Newmark}}}{\partial \mathbf{q}_\lambda^{\text{T}}} = \frac{\partial \mathbf{g}}{\partial \mathbf{q}_\lambda^{\text{T}}}
\end{aligned} \quad (8.13)$$

Note that the derivative $\frac{\mathbf{q}_2}{\mathbf{a}_2^{\text{T}}}$ follows from the Newmark interpolation (8.9) using the relation between $\mathbf{q}_2^T$ and $\mathbf{a}_2^T$. The tangent stiffness matrix $\mathbf{K}$ must also include derivatives of applied forces $\mathbf{f}^a$, which is currently not implemented. Furthermore, the Jacobian is not symmetric, which could be obtained by according scaling.

Once an update $\mathbf{q}_i^{\text{Newton}}$ has been computed, the interpolation formulas (8.9) need to be evaluated before the next residual and Jacobian can be computed.

Chapter 9

References

coming soon!

Chapter 10

License

```
EXUDYN General License (Version 1.0)
======================================

Copyright (c) 2018-2019 Johannes Gerstmayr, Institute of Mechatronics, University of
    Innsbruck, Austria.

Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are
met:

- Redistributions of source code must retain the above copyright
  notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright
  notice, this list of conditions and the following disclaimer listed
  in this license in the documentation and/or other materials
  provided with the distribution.

- Neither the name of the copyright holders nor the names of its
  contributors may be used to endorse or promote products derived from
  this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

EXUDYN is making use of the following third party codes and libraries, which are mentioned in the following incl. the licenses:

OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This program contains SuperLU 5.0, ExtGL, BLAS 3.5.0, LAPACK 3.5.0, Spectra
and Eigen covered under the following licenses:

======================================

GLFW 3.3 – www.glfw.org

This software is provided 'as–is', without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would
   be appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not
   be misrepresented as being the original software.

3. This notice may not be removed or altered from any source
   distribution.

======================================

PYBIND11

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this
   list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.

=====================================

Eigen3 is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.
For more information go to http://eigen.tuxfamily.org/.
Eigen3 is used under the Mozilla Public License Version 2.0 (MPL2) license:

Mozilla Public License Version 2.0
==================================

1. Definitions

1.1. "Contributor"
means each individual or legal entity that creates, contributes to the creation of, or owns Covered Software.

1.2. "Contributor Version"
means the combination of the Contributions of others (if any) used by a Contributor and that particular Contributor's Contribution.

1.3. "Contribution"
means Covered Software of a particular Contributor.

1.4. "Covered Software"
means Source Code Form to which the initial Contributor has attached the notice in Exhibit A, the Executable Form of such Source Code Form, and Modifications of such Source Code Form, in each case including portions thereof.

1.5. "Incompatible With Secondary Licenses"
means

(a) that the initial Contributor has attached the notice described in Exhibit B to the Covered Software; or

(b) that the Covered Software was made available under the terms of version 1.1 or earlier of the License, but not also under the terms of a Secondary License.

1.6. "Executable Form"
means any form of the work other than Source Code Form.

```
1.7. "Larger Work"
    means a work that combines Covered Software with other material, in
    a separate file or files, that is not Covered Software.

1.8. "License"
    means this document.

1.9. "Licensable"
    means having the right to grant, to the maximum extent possible,
    whether at the time of the initial grant or subsequently, any and
    all of the rights conveyed by this License.

1.10. "Modifications"
    means any of the following:

    (a) any file in Source Code Form that results from an addition to,
        deletion from, or modification of the contents of Covered
        Software; or

    (b) any new file in Source Code Form that contains any Covered
        Software.

1.11. "Patent Claims" of a Contributor
    means any patent claim(s), including without limitation, method,
    process, and apparatus claims, in any patent Licensable by such
    Contributor that would be infringed, but for the grant of the
    License, by the making, using, selling, offering for sale, having
    made, import, or transfer of either its Contributions or its
    Contributor Version.

1.12. "Secondary License"
    means either the GNU General Public License, Version 2.0, the GNU
    Lesser General Public License, Version 2.1, the GNU Affero General
    Public License, Version 3.0, or any later versions of those
    licenses.

1.13. "Source Code Form"
    means the form of the work preferred for making modifications.

1.14. "You" (or "Your")
    means an individual or a legal entity exercising rights under this
    License. For legal entities, "You" includes any entity that
    controls, is controlled by, or is under common control with You. For
    purposes of this definition, "control" means (a) the power, direct
    or indirect, to cause the direction or management of such entity,
    whether by contract or otherwise, or (b) ownership of more than
    fifty percent (50%) of the outstanding shares or beneficial
```

ownership of such entity.

2. License Grants and Conditions

2.1. Grants

Each Contributor hereby grants You a world-wide, royalty-free, non-exclusive license:

(a) under intellectual property rights (other than patent or trademark) Licensable by such Contributor to use, reproduce, make available, modify, display, perform, distribute, and otherwise exploit its Contributions, either on an unmodified basis, with Modifications, or as part of a Larger Work; and

(b) under Patent Claims of such Contributor to make, use, sell, offer for sale, have made, import, and otherwise transfer either its Contributions or its Contributor Version.

2.2. Effective Date

The licenses granted in Section 2.1 with respect to any Contribution become effective for each Contribution on the date the Contributor first distributes such Contribution.

2.3. Limitations on Grant Scope

The licenses granted in this Section 2 are the only rights granted under this License. No additional rights or licenses will be implied from the distribution or licensing of Covered Software under this License. Notwithstanding Section 2.1(b) above, no patent license is granted by a Contributor:

(a) for any code that a Contributor has removed from Covered Software; or

(b) for infringements caused by: (i) Your and any other third party's modifications of Covered Software, or (ii) the combination of its Contributions with other software (except as part of its Contributor Version); or

(c) under Patent Claims infringed by Covered Software in the absence of its Contributions.

This License does not grant any rights in the trademarks, service marks, or logos of any Contributor (except as may be necessary to comply with the notice requirements in Section 3.4).

2.4. Subsequent Licenses

No Contributor makes additional grants as a result of Your choice to distribute the Covered Software under a subsequent version of this License (see Section 10.2) or under the terms of a Secondary License (if permitted under the terms of Section 3.3).

2.5. Representation

Each Contributor represents that the Contributor believes its Contributions are its original creation(s) or it has sufficient rights to grant the rights to its Contributions conveyed by this License.

2.6. Fair Use

This License is not intended to limit any rights You have under applicable copyright doctrines of fair use, fair dealing, or other equivalents.

2.7. Conditions

Sections 3.1, 3.2, 3.3, and 3.4 are conditions of the licenses granted in Section 2.1.

3. Responsibilities

3.1. Distribution of Source Form

All distribution of Covered Software in Source Code Form, including any Modifications that You create or to which You contribute, must be under the terms of this License. You must inform recipients that the Source Code Form of the Covered Software is governed by the terms of this License, and how they can obtain a copy of this License. You may not attempt to alter or restrict the recipients' rights in the Source Code Form.

3.2. Distribution of Executable Form

If You distribute Covered Software in Executable Form then:

(a) such Covered Software must also be made available in Source Code Form, as described in Section 3.1, and You must inform recipients of the Executable Form how they can obtain a copy of such Source Code Form by reasonable means in a timely manner, at a charge no more than the cost of distribution to the recipient; and

(b) You may distribute such Executable Form under the terms of this License, or sublicense it under different terms, provided that the license for the Executable Form does not attempt to limit or alter the recipients' rights in the Source Code Form under this License.

3.3. Distribution of a Larger Work

You may create and distribute a Larger Work under terms of Your choice, provided that You also comply with the requirements of this License for the Covered Software. If the Larger Work is a combination of Covered Software with a work governed by one or more Secondary Licenses, and the Covered Software is not Incompatible With Secondary Licenses, this License permits You to additionally distribute such Covered Software under the terms of such Secondary License(s), so that the recipient of the Larger Work may, at their option, further distribute the Covered Software under the terms of either this License or such Secondary License(s).

3.4. Notices

You may not remove or alter the substance of any license notices (including copyright notices, patent notices, disclaimers of warranty, or limitations of liability) contained within the Source Code Form of the Covered Software, except that You may alter any license notices to the extent required to remedy known factual inaccuracies.

3.5. Application of Additional Terms

You may choose to offer, and to charge a fee for, warranty, support, indemnity or liability obligations to one or more recipients of Covered Software. However, You may do so only on Your own behalf, and not on behalf of any Contributor. You must make it absolutely clear that any such warranty, support, indemnity, or liability obligation is offered by You alone, and You hereby agree to indemnify every Contributor for any liability incurred by such Contributor as a result of warranty, support, indemnity or liability terms You offer. You may include additional disclaimers of warranty and limitations of liability specific to any jurisdiction.

4. Inability to Comply Due to Statute or Regulation

If it is impossible for You to comply with any of the terms of this License with respect to some or all of the Covered Software due to statute, judicial order, or regulation then You must: (a) comply with the terms of this License to the maximum extent possible; and (b) describe the limitations and the code they affect. Such description must be placed in a text file included with all distributions of the Covered Software under this License. Except to the extent prohibited by statute or regulation, such description must be sufficiently detailed for a recipient of ordinary skill to be able to understand it.

5. Termination

5.1. The rights granted under this License will terminate automatically if You fail to comply with any of its terms. However, if You become compliant, then the rights granted under this License from a particular Contributor are reinstated (a) provisionally, unless and until such Contributor explicitly and finally terminates Your grants, and (b) on an ongoing basis, if such Contributor fails to notify You of the non-compliance by some reasonable means prior to 60 days after You have come back into compliance. Moreover, Your grants from a particular Contributor are reinstated on an ongoing basis if such Contributor notifies You of the non-compliance by some reasonable means, this is the first time You have received notice of non-compliance with this License from such Contributor, and You become compliant prior to 30 days after Your receipt of the notice.

5.2. If You initiate litigation against any entity by asserting a patent infringement claim (excluding declaratory judgment actions, counter-claims, and cross-claims) alleging that a Contributor Version directly or indirectly infringes any patent, then the rights granted to You by any and all Contributors for the Covered Software under Section 2.1 of this License shall terminate.

5.3. In the event of termination under Sections 5.1 or 5.2 above, all end user license agreements (excluding distributors and resellers) which have been validly granted by You or Your distributors under this License prior to termination shall survive termination.

```
************************************************************************
*                                                                      *
*  6. Disclaimer of Warranty                                           *
*  -------------------------                                           *
*                                                                      *
*  Covered Software is provided under this License on an "as is"       *
*  basis, without warranty of any kind, either expressed, implied, or  *
*  statutory, including, without limitation, warranties that the       *
*  Covered Software is free of defects, merchantable, fit for a        *
*  particular purpose or non-infringing. The entire risk as to the     *
*  quality and performance of the Covered Software is with You.        *
```

```
*  Should any Covered Software prove defective in any respect, You     *
*  (not any Contributor) assume the cost of any necessary servicing,   *
*  repair, or correction. This disclaimer of warranty constitutes an   *
*  essential part of this License. No use of any Covered Software is   *
*  authorized under this License except under this disclaimer.         *
*                                                                      *
************************************************************************

************************************************************************
*                                                                      *
*  7. Limitation of Liability                                          *
*  --------------------------                                          *
*                                                                      *
*  Under no circumstances and under no legal theory, whether tort      *
*  (including negligence), contract, or otherwise, shall any           *
*  Contributor, or anyone who distributes Covered Software as          *
*  permitted above, be liable to You for any direct, indirect,         *
*  special, incidental, or consequential damages of any character      *
*  including, without limitation, damages for lost profits, loss of    *
*  goodwill, work stoppage, computer failure or malfunction, or any    *
*  and all other commercial damages or losses, even if such party      *
*  shall have been informed of the possibility of such damages. This   *
*  limitation of liability shall not apply to liability for death or   *
*  personal injury resulting from such party's negligence to the       *
*  extent applicable law prohibits such limitation. Some               *
*  jurisdictions do not allow the exclusion or limitation of           *
*  incidental or consequential damages, so this exclusion and          *
*  limitation may not apply to You.                                    *
*                                                                      *
************************************************************************
```

8. Litigation

Any litigation relating to this License may be brought only in the courts of a jurisdiction where the defendant maintains its principal place of business and such litigation shall be governed by laws of that jurisdiction, without reference to its conflict-of-law provisions. Nothing in this Section shall prevent a party's ability to bring cross-claims or counter-claims.

9. Miscellaneous

This License represents the complete agreement concerning the subject matter hereof. If any provision of this License is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable. Any law or regulation which provides that the language of a contract shall be construed against the drafter shall not be used to construe this License against a Contributor.

10. Versions of the License

10.1. New Versions

Mozilla Foundation is the license steward. Except as provided in Section 10.3, no one other than the license steward has the right to modify or publish new versions of this License. Each version will be given a distinguishing version number.

10.2. Effect of New Versions

You may distribute the Covered Software under the terms of the version of the License under which You originally received the Covered Software, or under the terms of any subsequent version published by the license steward.

10.3. Modified Versions

If you create software not governed by this License, and you want to create a new license for such software, you may create and use a modified version of this License if you rename the license and remove any references to the name of the license steward (except to note that such modified license differs from this License).

10.4. Distributing Source Code Form that is Incompatible With Secondary Licenses

If You choose to distribute Source Code Form that is Incompatible With Secondary Licenses under the terms of this version of the License, the notice described in Exhibit B of this License must be attached.

Exhibit A — Source Code Form License Notice

  This Source Code Form is subject to the terms of the Mozilla Public
  License, v. 2.0. If a copy of the MPL was not distributed with this
  file, You can obtain one at http://mozilla.org/MPL/2.0/.

If it is not possible or desirable to put the notice in a particular file, then You may include the notice in a location (such as a LICENSE file in a relevant directory) where a recipient would be likely to look for such a notice.

You may add additional accurate notices of copyright ownership.

Exhibit B — "Incompatible With Secondary Licenses" Notice

```
This Source Code Form is "Incompatible With Secondary Licenses", as
defined by the Mozilla Public License, v. 2.0.
```