

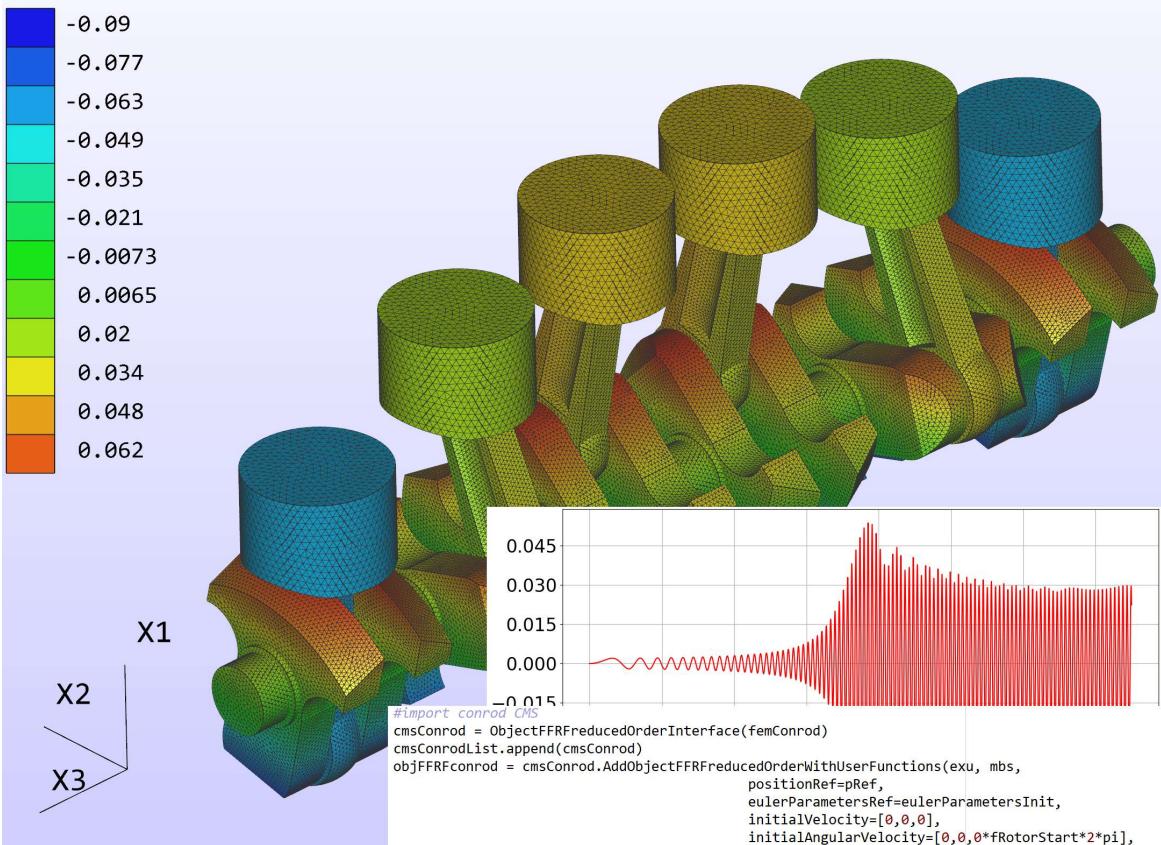
# Flexible Multibody Dynamics Systems with Python and C++

## Exudyn USER DOCUMENTATION

### EXUDYN

Solver finished successfully  
time = 0.0355

Displacement(0)  
min=-0.0904633, max=0.0758431



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

Exudyn version = 1.2.28 (Corea)  
(build date and time=2022-04-04 12:35 )  
CHECK [Section 12](#) for changes from previous versions!!!

University of Innsbruck, Department of Mechatronics, April 4, 2022,

Johannes Gerstmayr

# Table of Contents

<b>1</b>	<b>Installation and Getting Started</b>	<b>1</b>
1.1	Getting started . . . . .	1
1.1.1	What is EXUDYN? . . . . .	1
1.1.2	Developers of EXUDYN and thanks . . . . .	2
1.2	Installation instructions . . . . .	3
1.2.1	Requirements for EXUDYN? . . . . .	3
1.2.2	Install EXUDYN with PIP INSTALLER (pypi.org) . . . . .	4
1.2.3	DEPRECATED: Install with Windows MSI installer . . . . .	4
1.2.4	Install from Wheel (UBUNTU and Windows) . . . . .	5
1.2.5	Build and install EXUDYN under Windows 10? . . . . .	5
1.2.6	Build and install EXUDYN under Mac OS X? . . . . .	6
1.2.7	Build and install EXUDYN under UBUNTU? . . . . .	7
1.2.8	Uninstall EXUDYN . . . . .	9
1.2.9	How to install EXUDYN and use the C++ source code (advanced)? . . . . .	9
1.3	Further notes . . . . .	9
1.3.1	Goals of EXUDYN . . . . .	9
1.4	Run a simple example in Python . . . . .	10
1.5	Trouble shooting and FAQ . . . . .	13
1.5.1	Trouble shooting . . . . .	13
1.5.2	FAQ . . . . .	17
<b>2</b>	<b>Overview on EXUDYN</b>	<b>21</b>
2.1	Module structure . . . . .	21
2.1.1	Overview of modules . . . . .	21
2.1.2	Conventions: items, indexes, coordinates . . . . .	24
2.2	Items: Nodes, Objects, Loads, Markers, Sensors, ... . . . . .	24
2.2.1	Nodes . . . . .	24
2.2.2	Objects . . . . .	25
2.2.3	Markers . . . . .	25
2.2.4	Loads . . . . .	26
2.2.5	Sensors . . . . .	26
2.2.6	Reference coordinates and displacements . . . . .	26

2.3	Mapping between local and global coordinates . . . . .	26
2.4	Exudyn Basics . . . . .	27
2.4.1	Interaction with the EXUDYN module . . . . .	27
2.4.2	Simulation settings . . . . .	28
2.4.3	Generating output and results . . . . .	28
2.4.4	Visualization settings . . . . .	29
2.4.5	Graphics pipeline . . . . .	30
2.4.6	Graphics user Python functions . . . . .	31
2.4.7	Color and RGBA . . . . .	31
2.4.8	Camera following objects and interacting with model view . . . . .	32
2.4.9	Solution viewer . . . . .	32
2.4.10	Generating animations . . . . .	33
2.4.11	Contact problems . . . . .	34
2.4.12	Removing convergence problems and solver failures . . . . .	35
2.4.13	Performance and ways to speed up computations . . . . .	36
2.5	C++ Code . . . . .	37
2.5.1	Focus of the C++ code . . . . .	37
2.5.2	C++ Code structure . . . . .	38
2.5.3	C++ Code: Modules . . . . .	38
2.5.4	Code style and conventions . . . . .	39
2.5.5	Notation conventions . . . . .	40
2.5.6	No-abbreviations-rule . . . . .	41
2.6	Changes . . . . .	41
3	<b>Tutorial</b> . . . . .	43
3.1	Mass-Spring-Damper tutorial . . . . .	43
3.2	Rigid body and joints tutorial . . . . .	48
4	<b>Python-C++ command interface</b> . . . . .	57
4.1	General information on Python interface . . . . .	57
4.1.1	Item index . . . . .	58
4.2	EXUDYN . . . . .	58
4.3	SystemContainer . . . . .	60
4.4	MainSystem . . . . .	62
4.4.1	MainSystem: Node . . . . .	64
4.4.2	MainSystem: Object . . . . .	65
4.4.3	MainSystem: Marker . . . . .	67
4.4.4	MainSystem: Load . . . . .	68
4.4.5	MainSystem: Sensor . . . . .	69
4.5	SystemData . . . . .	70
4.5.1	SystemData: Access coordinates . . . . .	71
4.5.2	SystemData: Get object LTG coordinate mappings . . . . .	73

4.6	MatrixContainer . . . . .	74
4.7	GeneralContact . . . . .	74
4.8	VisuGeneralContact . . . . .	76
4.9	Type definitions . . . . .	77
4.9.1	OutputVariableType . . . . .	77
4.9.2	ConfigurationType . . . . .	78
4.9.3	ItemType . . . . .	79
4.9.4	NodeType . . . . .	79
4.9.5	DynamicSolverType . . . . .	80
4.9.6	KeyCode . . . . .	81
4.9.7	LinearSolverType . . . . .	81
5	<b>Python utility functions</b> . . . . .	83
5.1	Utility: ResultsMonitor . . . . .	83
5.2	Module: basicUtilities . . . . .	84
5.3	Module: beams . . . . .	88
5.4	Module: FEM . . . . .	92
5.4.1	CLASS MaterialBaseClass (in module FEM) . . . . .	97
5.4.2	CLASS KirchhoffMaterial(MaterialBaseClass) (in module FEM) . . . . .	97
5.4.3	CLASS FiniteElement (in module FEM) . . . . .	98
5.4.4	CLASS Tet4(FiniteElement) (in module FEM) . . . . .	98
5.4.5	CLASS ObjectFFRFInterface (in module FEM) . . . . .	98
5.4.6	CLASS ObjectFFRFreducedOrderInterface (in module FEM) . . . . .	99
5.4.7	CLASS HCBstaticModeSelection(Enum) (in module FEM) . . . . .	102
5.4.8	CLASS FEMInterface (in module FEM) . . . . .	102
5.5	Module: graphicsDataUtilities . . . . .	114
5.6	Module: GUI . . . . .	124
5.7	Module: interactive . . . . .	125
5.7.1	CLASS InteractiveDialog (in module interactive) . . . . .	127
5.8	Module: kinematicTree . . . . .	130
5.8.1	CLASS KinematicTree (in module kinematicTree) . . . . .	132
5.8.2	CLASS KinematicTree66 (in module kinematicTree) . . . . .	134
5.9	Module: lieGroupBasics . . . . .	135
5.10	Module: physics . . . . .	142
5.11	Module: plot . . . . .	144
5.12	Module: processing . . . . .	149
5.13	Module: rigidBodyUtilities . . . . .	155
5.13.1	CLASS RigidBodyInertia (in module rigidBodyUtilities) . . . . .	169
5.13.2	CLASS InertiaCuboid(RigidBodyInertia) (in module rigidBodyUtilities) . . . . .	170
5.13.3	CLASS InertiaRodX(RigidBodyInertia) (in module rigidBodyUtilities) . . . . .	171
5.13.4	CLASS InertiaMassPoint(RigidBodyInertia) (in module rigidBodyUtilities) . . . . .	171
5.13.5	CLASS InertiaSphere(RigidBodyInertia) (in module rigidBodyUtilities) . . . . .	171

5.13.6	CLASS InertiaHollowSphere(RigidBodyInertia) (in module rigidBodyUtilities)	171
5.13.7	CLASS InertiaCylinder(RigidBodyInertia) (in module rigidBodyUtilities)	172
5.14	Module: robotics	172
5.14.1	CLASS VRobotLink (in module robotics)	176
5.14.2	CLASS RobotLink (in module robotics)	176
5.14.3	CLASS VRobotTool (in module robotics)	177
5.14.4	CLASS RobotTool (in module robotics)	177
5.14.5	CLASS VRobotBase (in module robotics)	178
5.14.6	CLASS RobotBase (in module robotics)	178
5.14.7	CLASS Robot (in module robotics)	178
5.15	Module: robotics.future	181
5.16	Module: robotics.models	182
5.17	Module: robotics.mobile	183
5.18	Module: robotics.motion	184
5.18.1	CLASS ProfileConstantAcceleration (in module robotics.motion)	184
5.18.2	CLASS ProfileLinearAccelerationsList (in module robotics.motion)	184
5.18.3	CLASS ProfilePTP (in module robotics.motion)	185
5.18.4	CLASS Trajectory (in module robotics.motion)	186
5.19	Module: robotics.special	188
5.20	Module: robotics.utilities	192
5.21	Module: signalProcessing	192
5.22	Module: solver	194
5.23	Module: utilities	198
5.23.1	CLASS TCPIPdata (in module utilities)	210
<b>6</b>	<b>Theory and formulations</b>	<b>211</b>
6.1	Notation	211
6.1.1	Common types in item descriptions	211
6.1.2	MatrixContainer	212
6.1.3	States and coordinate attributes	212
6.1.4	Symbols in item equations	213
6.1.5	Reference and current coordinates	215
6.1.6	Reference point	215
6.1.7	Coordinate Systems	216
6.1.8	Integration Points	216
6.2	Model order reduction and component mode synthesis	218
6.2.1	Eigenmodes	218
6.2.2	Hurty-Craig-Bampton modes	220
6.3	Modeling of Contact in EXUDYN	225
6.3.1	Contact of meshed rigid bodies	225
6.3.2	Regularized friction	226
6.3.3	Sphere-sphere contact: Equations	226

6.3.4	Contact relations for ANCF cable $g_i$ (marker $m0$ ) and sphere $g_j$ (marker $m1$ ) . . .	232
<b>7</b>	<b>Objects, nodes, markers, loads and sensors reference manual</b>	<b>239</b>
7.1	Nodes . . . . .	240
7.1.1	NodePoint . . . . .	240
7.1.2	NodePoint2D . . . . .	242
7.1.3	NodeRigidBodyEP . . . . .	244
7.1.4	NodeRigidBodyRxyz . . . . .	247
7.1.5	NodeRigidBodyRotVecLG . . . . .	250
7.1.6	NodeRigidBody2D . . . . .	253
7.1.7	Node1D . . . . .	256
7.1.8	NodePoint2DSlope1 . . . . .	258
7.1.9	NodeGenericODE2 . . . . .	260
7.1.10	NodeGenericODE1 . . . . .	262
7.1.11	NodeGenericData . . . . .	263
7.1.12	NodePointGround . . . . .	265
7.2	Objects (Body) . . . . .	267
7.2.1	ObjectGround . . . . .	267
7.2.2	ObjectMassPoint . . . . .	270
7.2.3	ObjectMassPoint2D . . . . .	273
7.2.4	ObjectMass1D . . . . .	276
7.2.5	ObjectRotationalMass1D . . . . .	279
7.2.6	ObjectRigidBody . . . . .	282
7.2.7	ObjectRigidBody2D . . . . .	288
7.3	Objects (SuperElement) . . . . .	292
7.3.1	ObjectGenericODE2 . . . . .	292
7.3.2	ObjectFFRF . . . . .	300
7.3.3	ObjectFFRFreducedOrder . . . . .	308
7.4	Objects (FiniteElement) . . . . .	318
7.4.1	ObjectANCF Cable2D . . . . .	318
7.4.2	ObjectALEANCF Cable2D . . . . .	326
7.4.3	ObjectBeamGeometricallyExact2D . . . . .	329
7.5	Objects (Joint) . . . . .	331
7.5.1	ObjectJointGeneric . . . . .	331
7.5.2	ObjectJointRevoluteZ . . . . .	337
7.5.3	ObjectJointPrismaticX . . . . .	341
7.5.4	ObjectJointSpherical . . . . .	345
7.5.5	ObjectJointRollingDisc . . . . .	348
7.5.6	ObjectJointRevolute2D . . . . .	352
7.5.7	ObjectJointPrismatic2D . . . . .	354
7.5.8	ObjectJointSliding2D . . . . .	356
7.5.9	ObjectJointALEMoving2D . . . . .	361

7.6	Objects (Connector) . . . . .	366
7.6.1	ObjectConnectorSpringDamper . . . . .	366
7.6.2	ObjectConnectorCartesianSpringDamper . . . . .	372
7.6.3	ObjectConnectorRigidBodySpringDamper . . . . .	377
7.6.4	ObjectConnectorTorsionalSpringDamper . . . . .	383
7.6.5	ObjectConnectorCoordinateSpringDamper . . . . .	388
7.6.6	ObjectConnectorGravity . . . . .	392
7.6.7	ObjectConnectorRollingDiscPenalty . . . . .	396
7.6.8	ObjectContactConvexRoll . . . . .	401
7.6.9	ObjectContactCoordinate . . . . .	406
7.6.10	ObjectContactCircleCable2D . . . . .	408
7.6.11	ObjectContactFrictionCircleCable2D . . . . .	410
7.7	Objects (Constraint) . . . . .	420
7.7.1	ObjectConnectorDistance . . . . .	420
7.7.2	ObjectConnectorCoordinate . . . . .	423
7.7.3	ObjectConnectorCoordinateVector . . . . .	428
7.8	Objects (Object) . . . . .	433
7.8.1	ObjectGenericODE1 . . . . .	433
7.9	Markers . . . . .	437
7.9.1	MarkerBodyMass . . . . .	437
7.9.2	MarkerBodyPosition . . . . .	439
7.9.3	MarkerBodyRigid . . . . .	441
7.9.4	MarkerNodePosition . . . . .	443
7.9.5	MarkerNodeRigid . . . . .	444
7.9.6	MarkerNodeCoordinate . . . . .	445
7.9.7	MarkerNodeCoordinates . . . . .	447
7.9.8	MarkerNodeODE1Coordinate . . . . .	448
7.9.9	MarkerNodeRotationCoordinate . . . . .	449
7.9.10	MarkerSuperElementPosition . . . . .	450
7.9.11	MarkerSuperElementRigid . . . . .	453
7.9.12	MarkerObjectODE2Coordinates . . . . .	459
7.9.13	MarkerBodyCable2DShape . . . . .	460
7.9.14	MarkerBodyCable2DCoordinates . . . . .	462
7.10	Loads . . . . .	463
7.10.1	LoadForceVector . . . . .	463
7.10.2	LoadTorqueVector . . . . .	466
7.10.3	LoadMassProportional . . . . .	468
7.10.4	LoadCoordinate . . . . .	471
7.11	Sensors . . . . .	473
7.11.1	SensorNode . . . . .	473
7.11.2	SensorObject . . . . .	475

7.11.3	SensorBody	477
7.11.4	SensorSuperElement	479
7.11.5	SensorMarker	481
7.11.6	SensorLoad	483
7.11.7	SensorUserFunction	484
<b>8</b>	<b>ExUDYN Settings and Solver Structures</b>	<b>487</b>
8.1	Simulation settings	487
8.1.1	SolutionSettings	487
8.1.2	NumericalDifferentiationSettings	490
8.1.3	DiscontinuousSettings	491
8.1.4	NewtonSettings	492
8.1.5	GeneralizedAlphaSettings	493
8.1.6	ExplicitIntegrationSettings	494
8.1.7	TimeIntegrationSettings	494
8.1.8	StaticSolverSettings	497
8.1.9	LinearSolverSettings	499
8.1.10	Parallel	500
8.1.11	SimulationSettings	501
8.2	Visualization settings	501
8.2.1	VSettingsGeneral	501
8.2.2	VSettingsContour	503
8.2.3	VSettingsNodes	504
8.2.4	VSettingsBeams	505
8.2.5	VSettingsBodies	506
8.2.6	VSettingsConnectors	506
8.2.7	VSettingsMarkers	507
8.2.8	VSettingsLoads	507
8.2.9	VSettingsSensors	508
8.2.10	VSettingsContact	508
8.2.11	VSettingsWindow	509
8.2.12	VSettingsOpenGL	510
8.2.13	VSettingsExportImages	512
8.2.14	VSettingsInteractive	513
8.2.15	VisualizationSettings	514
8.3	Solver substructures	515
8.3.1	CSolverTimer	515
8.3.2	SolverLocalData	516
8.3.3	SolverIterationData	517
8.3.4	SolverConvergenceData	518
8.3.5	SolverOutputData	519
8.3.6	MainSolverStatic	520

8.3.7	MainSolverImplicitSecondOrder . . . . .	523
8.3.8	MainSolverExplicit . . . . .	526
<b>9</b>	<b>3D graphics visualization</b>	<b>531</b>
9.1	Mouse input . . . . .	531
9.1.1	6D mouse . . . . .	531
9.2	Keyboard input . . . . .	532
9.3	GraphicsData . . . . .	533
9.4	Character encoding: UTF-8 . . . . .	534
<b>10</b>	<b>Notation and System of equations of motion</b>	<b>537</b>
10.1	Left-Hand-Side (of equation) (LHS)–Right-Hand-Side (of equation) (RHS) naming conventions in EXUDYN . . . . .	537
10.2	System assembly . . . . .	538
10.3	Nomenclature for system equations of motion and solvers . . . . .	539
10.3.1	System equations of motion . . . . .	540
<b>11</b>	<b>Solvers</b>	<b>541</b>
11.1	Solvers in EXUDYN . . . . .	541
11.2	General solver structure . . . . .	542
11.3	Explicit solvers . . . . .	546
11.3.1	Explicit Runge-Kutta method . . . . .	546
11.3.2	Automatic step size control . . . . .	547
11.3.3	Stability limit . . . . .	548
11.3.4	Explicit Lie group integrators . . . . .	548
11.3.5	Constraints with explicit solvers . . . . .	549
11.4	Implicit (trapezoidal rule-based, Newmark, generalized- $\alpha$ ) solver . . . . .	549
11.4.1	Newmark and generalized- $\alpha$ method . . . . .	549
11.4.2	Parameter selection for generalized- $\alpha$ . . . . .	550
11.4.3	Newton iteration . . . . .	551
11.4.4	Initial accelerations . . . . .	552
11.5	Optimization and parameter variation . . . . .	554
11.5.1	Parameter Variation . . . . .	554
11.5.2	Genetic Optimization . . . . .	554
<b>12</b>	<b>Issues and bugs</b>	<b>557</b>
12.1	Resolved issues and resolved bugs . . . . .	557
12.2	Known open bugs . . . . .	606
<b>References</b>		<b>607</b>
<b>13</b>	<b>License</b>	<b>611</b>

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

<b>2D</b>	two dimensions or planar
<b>3D</b>	three dimensions or spatial
<b>abs</b>	absolute (e.g., absolute error), absolute value
<b>AE</b>	algebraic equations
<b>CMS</b>	component mode synthesis
<b>coeffs</b>	coefficients
<b>COM</b>	center of mass
<b>EOM</b>	equations of motion
<b>EP</b>	Euler parameters
<b>FFRF</b>	floating frame of reference formulation
<b>HT</b>	homogeneous transformation
<b>LHS</b>	Left-Hand-Side (of equation)
<b>LTG</b>	local-to-global
<b>min</b>	minimum
<b>max</b>	maximum
<b>ODE</b>	ordinary differential equation
<b>ODE1</b>	first order ordinary differential equations
<b>ODE2</b>	second order ordinary differential equations
<b>pos</b>	position
<b>quad</b>	quadrangle, polygon with 4 vertices
<b>rel</b>	relative (e.g., relative error)
<b>RHS</b>	Right-Hand-Side (of equation)
<b>Rot</b>	rotation
<b>Rxyz</b>	rotation parameterization: consecutive rotations around x, y and z-axis (Tait-Bryan)
<b>STL</b>	STereoLithography
<b>T66</b>	Plücker transformation
<b>trig</b>	triangle (in graphics)

# Chapter 1

# Installation and 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').

Tutorial videos can be found in the [youtube channel of Exudyn](#). ExUDYN is hosted on [GitHub](#) [13]:

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

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

- email: [reply.exudyn@gmail.com](mailto:reply.exudyn@gmail.com)

Thanks for your contribution!

## 1.1 Getting started

This section will show:

1. What is ExUDYN?
2. Who is developing ExUDYN?
3. How to install ExUDYN
4. How to link ExUDYN and Python
5. Goals of ExUDYN
6. Run a simple example in Python
7. FAQ – Frequently asked questions

### 1.1.1 What is ExUDYN?

ExUDYN – (fLEXible mUltibody DYNamics – EXtend yoUr DYNamics)

ExUDYN is a C++ based Python library for efficient simulation of flexible multibody dynamics systems. It is the follow up code of the previously developed multibody code HOTINT, which Johannes Gerstmayer started during his PhD-thesis. It seemed that the previous code HOTINT reached limits of

further (efficient) development and it seemed impossible to continue from this code as it was outdated regarding programming techniques and the numerical formulation at the time ExUDYN was started.

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

The multibody formulation is mainly based on redundant coordinates. This means that computational objects (rigid bodies, flexible bodies, ...) are added as independent bodies to the system. Hereafter, connectors (e.g., springs or constraints) are used to interconnect the bodies. The connectors are using Markers on the bodies as interfaces, in order to transfer forces and displacements. For details on the interaction of nodes, objects, markers and loads see [Section 2.2](#).

### 1.1.2 Developers of ExUDYN and thanks

ExUDYN is currently (4-2022) developed at the University of Innsbruck. In the first phase most of the core code is written by Johannes Gerstmayr, implementing ideas that followed out of the project HOTINT. 15 years of development led to a lot of lessons learned and after 20 years, a code must be re-designed.

Some important tests for the coupling between C++ and Python have been written by Stefan Holzinger. Stefan also helped to set up the previous upload to GitLab and to test parallelization features. For the interoperability between C++ and Python, we extensively use **Pybind11**[[25](#)], originally written by Jakob Wenzel, see <https://github.com/pybind/pybind11>. Without Pybind11 we couldn't have made this project – Thank's a lot!

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

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

The following people have contributed to Python and C++ library implementations:

- Joachim Schöberl (Providing specialized NGsolve core library with `taskmanager` for **multithreaded parallelization**; NGsolve mesh and FE-matrices import; highly efficient eigenvector computations)
- Stefan Holzinger (Lie group solvers in Python, Lie group node)
- Peter Manzl (ConvexRoll Python and C++ implementation / pip install on linux / wsl with graphics)
- Martin Sereinig (special robotics functionality)

The following people have contributed to examples and other parts:

- Stefan Holzinger, Michael Pieber, Manuel Schieferle, Martin Knapp, Lukas March, Dominik Sponring, David Wibmer, Andreas Zwölfer, Peter Manzl, Simon Scheiber  
– thanks a lot! –

## 1.2 Installation instructions

### 1.2.1 Requirements for EXUDYN?

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

- **Anaconda, 64bit, Python 3.7.7**<sup>1</sup> (but Python 3.8 and 3.9 (since 2021-11) are also working well!)
- **Spyder 4.1.3** (with Python 3.7.7, 64bit), which is included in the Anaconda installation<sup>2</sup>

Many alternative options exist:

- In case that you have an older CPU, which does not support AVX2, use: Anaconda, 32bit, Python 3.6.5, or compile without AVX flags.<sup>3</sup>
- Users report successful use of EXUDYN with **Visual Studio Code**. **Jupyter** has been tested with some examples; both environments should work with default settings.
- Anaconda 2020-11 with **Python 3.8** and Spyder 4.1.5: no problems up to now (2021-07), TestSuite runs without problems since EXUDYN version 1.0.182.
- Anaconda 2021-11 with **Python 3.9** and Spyder 5.1.5: Tested with current version (1.1.99), TestSuite runs without problems.
- Alternative option with more stable Spyder (as compared to Spyder 4.1.3): Anaconda, 64bit, Python 3.6.5)<sup>4</sup>

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

**Run without Anaconda:** If you do not install Anaconda (e.g., under Linux), make sure that you have the according Python packages installed:

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

---

<sup>1</sup>Anaconda3 64bit with Python3.7.7 can be downloaded via the repository archive <https://repo.anaconda.com/archive/> choosing Anaconda3-2020.02-Windows-x86\_64.exe

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

<sup>3</sup>Anaconda 32bit with Python3.6 can be downloaded via the repository archive <https://repo.anaconda.com/archive/> choosing Anaconda3-5.2.0-Windows-x86.exe.

<sup>4</sup>Anaconda 64bit with Python3.6 can be downloaded via the repository archive <https://repo.anaconda.com/archive/> choosing Anaconda3-5.2.0-Windows-x86\_64.exe for 64bit.

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

You can install most of these packages using `pip install numpy` (Windows) or `pip3 install numpy` (Linux).

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

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

see also common blogs for your operating system.

### 1.2.2 Install EXUDYN with PIP INSTALLER (pypi.org)

Pre-built versions of EXUDYN are hosted on [pypi.org](https://pypi.org), see the project

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

As with most other packages, in the regular case (if your binary has been pre-built) you just need to do<sup>6</sup>

```
pip install exudyn
```

On Linux (currently only pre-built for UBUNTU 18.04 and 20.04), **update pip to at least 20.3** and use

```
pip3 install exudyn
```

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

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

In some cases (e.g. for AppleM1), your pre-built binary will not work due to some incompatibilities. Then you need to build from source as described in the 'Build and install' sections, [Section 1.2.5](#).

### 1.2.3 DEPRECATED: Install with Windows MSI installer

A simple way to install EXUDYN on Windows 10 (and maybe also Windows 7) is to use `.msi` installers in the `main/dist` directory<sup>7</sup>:

- NOTE (2022-03-18): `.msi` installers are now only available for selected Python versions; however, wheels can be downloaded directly from <https://pypi.org/project/exudyn>, see below
- For the 64bits Python 3.7 version, double click on (version may differ):  
`exudyn-1.0.248.win-amd64-py3.7.msi`
- Follow the instructions of the installer
- If Python / Anaconda is not found by the installer, provide the 'python directory' as the installation directory of Anaconda3, which usually is installed in:  
`C:\ProgramData\Anaconda3`

<sup>6</sup>If the index of pypi is not updated, it may help to use `pip install -i https://pypi.org/project/ exudyn`

<sup>7</sup>It works better if you installed only one Python version and if you installed Anaconda with the option 'Register Anaconda as my default Python 3.x' or similar; in other cases you may to specify some installation directories, etc.

#### **1.2.4 Install from Wheel (UBUNTU and Windows)**

A way to install the Python package ExUDYN is to use the so-called ‘wheels’ (file ending .whl). Wheels can be downloaded directly from <https://pypi.org/project/exudyn/#files>, for many Python versions and architectures.

For UBUNTU18.04 (which by default uses Python 3.6) this may read (version number 1.0.20 may be different):

- Python 3.6, 64bit: pip3 install dist\exudyn-1.0.20-cp36-cp36-linux\_x86\_64.whl

For UBUNTU20.04 (which by default uses Python 3.8) this may read (version number 1.0.20 may be different):

- Python 3.8, 64bit: pip3 install dist\exudyn-1.0.20-cp38-cp38-linux\_x86\_64.whl

NOTE that your installation may have environments with different Python versions, so install that ExUDYN version appropriately! If the wheel installation does not work on UBUNTU, it is highly recommended to build ExUDYN for your specific system as given in [Section 1.2.7](#).

#### **Windows:**

First, open an Anaconda prompt:

- EITHER calling: START->Anaconda->... OR go to anaconda/Scripts folder and call activate.bat
- You can check your Python version then, by running `python8`, the output reads like:

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

- type `exit()` to close Python

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

- Python 3.6, 32bit: pip install dist\exudyn-1.0.20-cp36-cp36m-win32.whl
- Python 3.6, 64bit: pip install dist\exudyn-1.0.20-cp36-cp36m-win\_amd64.whl
- Python 3.7, 64bit: pip install dist\exudyn-1.0.20-cp37-cp37m-win\_amd64.whl

#### **1.2.5 Build and install ExUDYN under Windows 10?**

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

- you need an appropriate compiler (tested with Microsoft Visual Studio; recommended: VS2017)
- install your Anaconda distribution including Spyder
- close all Python programs (e.g. Spyder, Jupyter, ...)
- run an Anaconda prompt (may need to be run as administrator)

---

<sup>8</sup>python3 under UBUNTU 18.04

- if you cannot run Anaconda prompt directly, do:
  - open windows shell (cmd.exe) as administrator (START → search for cmd.exe → right click on app → ‘run as administrator’ if necessary) [may not be necessary]
  - go to your Scripts folder inside the Anaconda folder (e.g. C:\ProgramData\Anaconda\Scripts) [may not be necessary]
  - run ‘activate.bat’ [may not be necessary]
- go to ‘main’ of your cloned github folder of EXUDYN
- run: `python setup.py install`
- read the output; if there are errors, try to solve them by installing appropriate modules

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

```
python setup.py bdist_wheel
```

This will add a wheel in the dist folder.

### 1.2.6 Build and install EXUDYN under Mac OS X?

Installation and building on Mac OS X is rarely tested, but first successful compilation including GLFW has been achieved. Requirements are an according Anaconda installation.

#### Tested configurations:

- Mac OS 11.x ‘Big Sur’, Mac Mini (2021), Apple M1, 16GB Memory
- Anaconda (i368 based with Rosetta 2) with Python 3.8
- this configuration is currently evaluated but showed general compatibility  
→ some wheels are already available on pypi (you may need to download them manually)!

Alternatively, we tested on:

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

Note, that in all cases tkinter does not yet run properly (help appreciated), while otherwise we produced a stable version. The AppleM1 native version is approx. 10x faster than the Rosetta version!

For a compatible Mac OS X system some pre-built wheels will be available via pypi.org. Note that these may be built on an emulated AppleM1, thus being much slower than the Windows or Linux compliant. **Compile from source:**

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

- `python setup.py bdist_wheel`

→ this compiles and takes approx. 5 minutes, depending on your machine  
→ it may produce some errors, depending on your version; if there are some linker errors (saying that there is no '-framework Cocoa' and '-framework OpenGL', just go back in the terminal and copy everything from 'g++ ...' until the end of the last command '-mmacosx-verion-min...' and paste it into the terminal. Calling that again will finalize linking; then run again

```
python setup.py bdist_wheel
```

→ this now creates the wheel (if you want to distribute) in the dist folder  
alternatively just call

- `python setup.py install`

to install exudyn

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

```
python springDamperUserFunctionTest.py
```

If there are other issues, we are happy to receive your detailed bug reports.

Note that you need to run

```
exudyn.StartRenderer()  
exudyn.DoRendererIdleTasks(-1)
```

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

### 1.2.7 Build and install EXUDYN under UBUNTU?

Having a new UBUNTU 18.04 standard installation (e.g. using a VM virtual box environment), the following steps need to be done (Python 3.6 is already installed on UBUNTU18.04, otherwise use `sudo apt install python3`)<sup>9</sup>:

First update ...

---

```
sudo apt-get update
```

---

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

---

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

---

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

---

```
pip3 install pybind11
```

---

<sup>9</sup>see also the youtube video: [https://www.youtube.com/playlist?list=PLZduTa9mdcmOh5KVUqatD9GzVg\\_jtl6fx](https://www.youtube.com/playlist?list=PLZduTa9mdcmOh5KVUqatD9GzVg_jtl6fx)

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

---

```
sudo apt-get install freeglut3 freeglut3-dev  
sudo apt-get install mesa-common-dev  
sudo apt-get install libglfw3 libglfw3-dev  
sudo apt-get install libx11-dev xorg-dev libglew1.5 libglew1.5-dev libglu1-mesa libglu1-  
mesa-dev libgl1-mesa-glx libgl1-mesa-dev
```

---

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

---

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

---

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

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

You can also create a UBUNTU wheel which can be easily installed on the same machine (x64), same operating system (UBUNTU18.04) and with same Python version (e.g., 3.6):

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

### **Exudyn under Ubuntu / WSL:**

- Note that EXUDYN also nicely works under WSL (Windows subsystem for linux; tested for Ubuntu18.04) and an according xserver (VcXsrv).
- Just set the display variable in your `.bashrc` file accordingly and you can enjoy the OpenGL windows and settings.
- It shall be noted that WSL + xserver works better than on MacOS, even for tkinter, multitasking, etc.! So, if you have troubles with your Mac, use a virtual machine with ubuntu and a xserver, that may do better

### **Exudyn under Raspberry Pi 4b:**

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

### **KNOWN issues for linux builds:**

- Using **WSL2** (Windows subsystem for linux), there occur some conflicts during build because of incompatible windows and linux file systems and builds will not be copied to the `dist` folder; workaround: go to explorer, right click on 'build' directory and set all rights for authenticated user to 'full access'

- **compiler (gcc,g++) conflicts:** It seems that EXUDYN works well on UBUNTU18.04 with the original Python 3.6.9 and gcc-7.5.0 version as well as with UBUNTU20.04 with Python 3.8.5 and gcc-9.3.0. Upgrading gcc on a linux system with Python 3.6 to, e.g., gcc-8.2 showed us a linker error when loading the EXUDYN module in Python – there are some common restriction using gcc versions different from those with which the Python version has been built. Starting python or python3 on your linux machine shows you the gcc version it had been build with. Check your current gcc version with: `gcc -version`

### 1.2.8 Uninstall EXUDYN

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

```
pip uninstall exudyn
```

To uninstall under UBUNTU, run:

```
sudo pip3 uninstall exudyn
```

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

### 1.2.9 How to install EXUDYN and use the C++ source code (advanced)?

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

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

- Windows / Microsoft Visual Studio 2017 and above:
  - get the files from git
  - put them into a local directory (recommended: C:/DATA/cpp/EXUDYN\_git)
  - start `main.sln` with Visual Studio
  - compile the code and run `main/pythonDev/pytest.py` example code
  - adapt `pytest.py` for your applications
  - extend the C++ source code
  - link it to your own code
  - NOTE: on Linux systems, you mostly need to replace '/' with '\'
- Linux, etc.: not fully supported yet; however, all external libraries are Linux-compatible and thus should run with minimum adaptation efforts.

## 1.3 Further notes

### 1.3.1 Goals of EXUDYN

After the first development phase (2019-2021), it

- is a moderately large (2MB on windows!) multibody library, which can be easily linked to other projects,
- contains basic multibody rigid bodies, flexible bodies, joints, contact, etc.,
- includes a large Python utility library for convenient building and post processing of models,
- allows to efficiently simulate small scale systems (compute 100000s time steps per second for systems with  $n_{DOF} < 10$ ),
- allows to efficiently simulate medium scaled systems for problems with  $n_{DOF} < 1\,000\,000$ ,
- is a safe and widely accessible module for Python,
- allows to add user defined objects and solvers in C++,
- allows to add user defined objects and solvers in Python.

Future goals are:

- add more multi-threaded parallel computing techniques (first parts implemented, improvements planned during 2022),
- add vectorization,
- add specific and advanced connectors/constraints (extended wheels, contact, control connector)
- kinematical trees (with minimum coordinates),
- automatic step size selection for second order solvers,
- deeper integration of Lie groups,
- more interfaces for robotics,
- add 3D beams.

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

## 1.4 Run a simple example in Python

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

In order to start, run the Python interpreter Spyder (or any preferred Python environment). For the following example,

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

Hereafter, press the play button or F5 in Spyder.

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

---

```
runfile('C:/DATA/cpp/EXUDYN_git/main/pythonDev/Examples/myFirstExample.py',
      wdir='C:/DATA/cpp/EXUDYN_git/main/pythonDev/Examples')
+++++
EXUDYN V1.2.9 solver: implicit second order time integration
```

Listing 1.1: My first example

```

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

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

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

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

```

---

STEP100, t = 1 sec, timeToGo = 0 sec, Nit/step = 1  
solver finished after 0.0007824 seconds.

If you check your current directory (where `myFirstExample.py` lies), you will find a new file `coordinatesSolution.txt`, which contains the results of your computation (with default values for time integration). The beginning and end of the file should look like:

---

```

#Exudyn generalized alpha solver solution file
#simulation started=2019-11-14,20:35:12
#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]
#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 coordinates, their derivatives and Lagrange multipliers on system level. For relation of local to global coordinates, see [Section 2.3](#). As expected, the  $x$ -coordinate of the point mass has constant acceleration  $a = f/m = 0.001/10 = 0.0001$ , the velocity grows up to 0.0001 after 1 second and the point mass moves 0.00005 along the  $x$ -axis.

## 1.5 Trouble shooting and FAQ

### 1.5.1 Trouble shooting

Python import errors:

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

---

```
Traceback (most recent call last):
```

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

---

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

---

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

---

```
Traceback (most recent call last):
```

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

---

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

---

**There are several reasons and workarounds:**

- You mixed up 32 and 64 bits version (see below)
- You are using an exudyn version for Python  $x_1.y_1$  (e.g., 3.6.z<sub>1</sub>) different from the Python  $x_2.y_2$  version in your Anaconda (e.g., 3.7.z<sub>2</sub>); note that  $x_1 = x_2$  and  $y_1 = y_2$  must be obeyed while  $z_1$  and  $z_2$  may be different
- **ModuleNotFoundError: No module named 'exudynCPP':**

- A known reason is that your CPU **does not support AVX2**, while ExUDYN is compiled with the AVX2 option<sup>10</sup>.
- **workaround** to solve the AVX problem: use the Python 3.6 32bits version, which is compiled without AVX2; you can also compile for your specific Python version without AVX if you adjust the `setup.py` file in the main folder.

---

<sup>10</sup>modern Intel Core-i3, Core-i5 and Core-i7 processors as well as AMD processors, especially Zen and Zen-2 architectures should have no problems with AVX2; however, low-cost Celeron, Pentium and older AMD processors do **not** support AVX2, e.g., Intel Celeron G3900, Intel core 2 quad q6600, Intel Pentium Gold G5400T; check the system settings of your computer to find out the processor type; typical CPU manufacturer pages or Wikipedia provide information on this

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

### Typical Python errors:

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

---

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

---

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

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

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

---

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

---

- see installation instructions to install missing Python modules, [Section 1.2](#).

### Typical solver errors:

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

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

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

→ see also 'Convergence problems', [Section 2.4.12](#)

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

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

---

```
=====
SYSTEM ERROR [file 'C:\ProgramData\Anaconda3_64b37\lib\site-packages\exudyn\solver.py',
line 207]:
```

```

CSolverBase::Newton: System Jacobian seems to be singular / not invertible!
time/load step #1, time = 0.0002
causing system equation number (coordinate number) = 42
=====

```

---

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

```

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

```

---

- object 7 seems to be the reason, possibly there are too much (joint) constraints applied to your system, check this object.
- show typical REASONS and SOLUTIONS, by using `showHints=True` in `exu.SolveDynamic(...)` or `exu.SolveStatic(...)`
- You can also **highlight** object 7 by using the following code in the iPython console:

```

exu.StartRenderer()
HighlightItem(SC,mbs,7)

```

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

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

```

+++++
EXUDYN V1.0.200 solver: implicit second order time integration
Newton (time/load step #1): convergence failed after 25 iterations; relative error =
0.079958, time = 2
Newton (time/load step #1): convergence failed after 25 iterations; relative error =
0.0707764, time = 1
Newton (time/load step #1): convergence failed after 25 iterations; relative error =
0.0185745, time = 0.5
Newton (time/load step #2): convergence failed after 25 iterations; relative error =
0.332953, time = 0.5
Newton (time/load step #2): convergence failed after 25 iterations; relative error =
0.0783815, time = 0.375
Newton (time/load step #2): convergence failed after 25 iterations; relative error =
0.0879718, time = 0.3125
Newton (time/load step #2): convergence failed after 25 iterations; relative error =
2.84704e-06, time = 0.28125

```

```
Newton (time/load step #3): convergence failed after 25 iterations; relative error =
1.9894e-07, time = 0.28125
STEP348, t = 20 sec, timeToGo = 0 sec, Nit/step = 7.00575
solver finished after 0.258349 seconds.
```

---

- this solver error is caused, because the nonlinear system cannot be solved using Newton's method.
- the static or dynamic solver by default tries to reduce step size to overcome this problem, but may fail finally (at minimum step size).
- possible reasons are: too large time steps (reduce step size by using more steps/second), inappropriate initial conditions, or inappropriate joints or constraints (remove joints to see if they are the reason), usually within a singular configuration. Sometimes a system may be just unsolvable in the way you set it up.
- see also 'Convergence problems', [Section 2.4.12](#)

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

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

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

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

---

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

## 1.5.2 FAQ

**Some frequently asked questions:**

1. When **importing** ExUDYN in Python (windows) I get an error
  - see trouble shooting instructions above!
2. I do not understand the **Python errors** – how can I find the reason of the error or crash?
  - Read trouble shooting section above!
  - First, you should read all error messages and warnings: from the very first to the last message. Very often, there is a definite line number which shows the error. Note, that if you are executing a string (or module) as a Python code, the line numbers refer to the local line number inside the script or module.
  - If everything fails, try to execute only part of the code to find out where the first error occurs. By omitting parts of the code, you should find the according source of the error.
  - If you think, it is a bug: send an email with a representative code snippet, version, etc. to `reply.exudyn@gmail.com`
3. Spyder **console hangs** up, does not show error messages, ....:
  - very often a new start of Spyder helps; most times, it is sufficient to restart the kernel or to just press the 'x' in your IPython console, which closes the current session and restarts the kernel (this is much faster than restarting Spyder)
  - restarting the IPython console also brings back all error messages
4. Where do I find the '**.exe**' file?
  - ExUDYN is only available via the Python interface as a module '`exudyn`', the C++ code being inside of `exudynCPP.pyd`, which is located in the `exudyn` folder where you installed the package. This means that you need to **run Python** (best: Spyder) and import the ExUDYN module.
5. I get the error message 'check potential mixing of different (object, node, marker, ...) indices', what does it mean?
  - probably you used wrong item indexes, see beginning of command interface in [Section 4](#).
  - E.g., an object number `oNum = mbs.AddObject(...)` is used at a place where a `NodeIndex` is expected, e.g., `mbs.AddObject(MassPoint(nodeNumber=oNum, ...))`
  - Usually, this is an ERROR in your code, it does not make sense to mix up these indexes!

→ In the exceptional case, that you want to convert numbers, see beginning of [Section 4](#).

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

- UPDATE 2020-06-01: with Spyder 4, using Python 3.7, type auto completion works much better, but may find too many completions.
- most Python environments (e.g., with Spyder 3) only have information up to the first sub-structure, e.g., `SC=exu.SystemContainer()` provides full access to SC in the type completion, but `mbs=SC.AddSystem()` is at the second sub-structure of the module and is not accessible.
- WORKAROUND: type `mbs>MainSystem()` **before** the `mbs=SC.AddSystem()` command and the interpreter will know what type mbs is. This also works for settings, e.g., simulation settings 'Newton'.

## 7. How to add graphics?

- Graphics (lines, text, 3D triangular / [STL](#) mesh) can be added to all BodyGraphicsData items in objects. Graphics objects which are fixed with the background can be attached to a ObjectGround object. Moving objects must be attached to the BodyGraphicsData of a moving body. Other moving bodies can be realized, e.g., by adding a ObjectGround and changing its reference with time. Furthermore, ObjectGround allows to add fully user defined graphics.

## 8. In `GenerateStraightLineANCFCable2D`

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

## 9. What is the difference between MarkerBodyPosition and MarkerBodyRigid?

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

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

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

---

```
File "C:/DATA/cpp/EXUDYN\_git/main/pythonDev/...<file name>", line XXX, in <module>
    solver.SolveSystem(...)

SystemError: <built-in method SolveSystem of PyCapsule object at 0x0CC63590>
    returned a result with an error set
```

---

- The pre-checks, which are performed to enable a crash-free simulation are insufficient for your model
- As a first try, **restart the IPython console** in order to get all error messages, which may be blocked due to a previous run of EXUDYN.

- Very likely, you are using Python user functions inside ExUDYN : They lead to an internal Python error, which is not always caught by ExUDYN ; e.g., a load user function UFload(mbs, t, load), which tries to access component load[3] of a load vector with 3 components will fail internally;
  - Use the print(...) command in Python at many places to find a possible error in user functions (e.g., put `print("Start user function XYZ")` at the beginning of every user function; test user functions from iPython console
  - It is also possible, that you are using inconsistent data, which leads to the crash. In that case, you should try to change your model: omit parts and find out which part is causing your error
  - see also **I do not understand the Python errors – how can I find the cause?**
11. Why can't I get the focus of the simulation window on startup (render window hidden)?
- Starting ExUDYN out of Spyder might not bring the simulation window to front, because of specific settings in Spyder(version 3.2.8), e.g., Tools→Preferences→Editor→Advanced settings: uncheck 'Maintain focus in the Editor after running cells or selections'; Alternatively, set `SC.visualizationSettings.window.alwaysOnTop=True` before starting the renderer with `exu.StartRenderer()`



# Chapter 2

## Overview on EXUDYN

### 2.1 Module structure

This section will show:

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

For an introduction to the solvers, see [Section 11](#).

#### 2.1.1 Overview of modules

Currently, the module structure is simple:

- **C++ parts**, see Figs. [2.1](#) and [2.2](#):
  - `exudyn`: on this level, there are just very few functions: `SystemContainer()`, `StartRenderer()`, `StopRenderer()`, `GetVersionString()`, `SolveStatic(...)`, `SolveDynamic(...)`, ... as well as system and user variable dictionaries `exudyn.variables` and `exudyn.sys`
  - `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 be added to the system, see [Section 7](#);
  - `mbs.systemData`: contains the initial, current, visualization, ... states of the system and holds the items, see Fig. [2.2](#)
- **Python parts** (this list is continuously extended, see [Section 5](#)), sorted by importance:
  - `exudyn.utilities`: contains helper classes in Python and includes EXUDYN modules `basicUtilities`, `rigidBodyUtilities`, `graphicsDataUtilities`, and `itemInterface`, which is recommended to be loaded at beginning of your model file

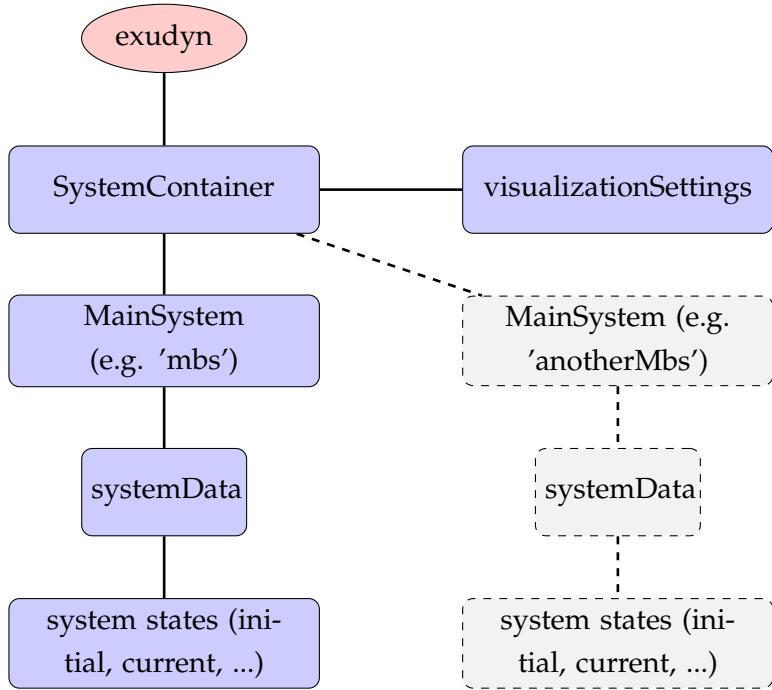


Figure 2.1: Overview of exudyn module.

- `exudyn.itemInterface`: contains the interface, which transfers python classes (e.g., of a `NodePoint`) to dictionaries that can be understood by the C++ module
- `exudyn.basicUtilities`: contains basic helper classes, without importing numpy
- `exudyn.rigidBodyUtilities`: contains important helper classes for creation of rigid body inertia, rigid bodies, and rigid body joints; includes helper functions for rotation parameterization, rotation matrices, homogeneous transformations, etc.
- `exudyn.graphicsDataUtilities`: provides some basic drawing utilities, definition of colors and basic drawing objects (including `STL` import); rotation/translation of `graphicsData` objects
- `exudyn.plot`: containing helper functions, which are physics related such as friction
- `exudyn.processing`: methods for optimization, parameter variation, sensitivity analysis, etc.
- `exudyn.FEM`: everything related to finite element import and creation of model order reduction flexible bodies
- `exudyn.robotics`: submodule containing several helper modules related to manipulators (`robotics`, `robotics.models`), mobile robots (`robotics.mobile`), trajectory generation (`robotics.motion`), etc.
- `exudyn.beams`: helper functions for creation of beams along straight lines and curves, sliding joints, etc.
- `exudyn.interactive`: helper classes to create interactive models (e.g. for teaching or demos)
- `exudyn.physics`: containing helper functions, which are physics related such as friction
- `exudyn.signalProcessing`: filters, FFT, etc.; interfaces to `scipy` and `numpy` methods
- `exudyn.solver`: functions imported when loading `exudyn`, containing main solvers

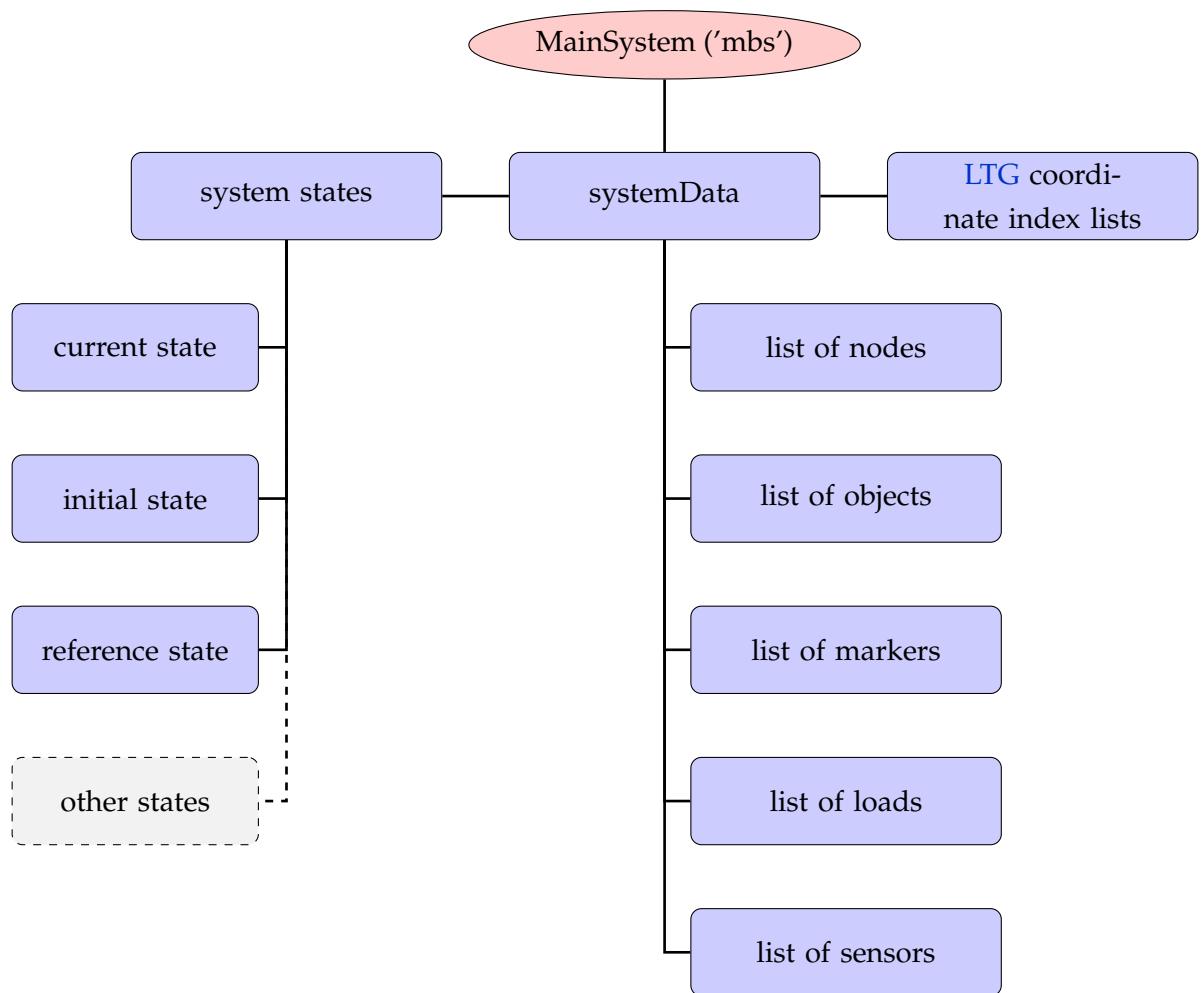


Figure 2.2: Overview of systemData, which connects items, states and stores the **LTG**. Note that access to items is provided via functions in **system**.

### 2.1.2 Conventions: items, indexes, coordinates

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

$$\text{item} \in \{\text{node, object, marker, load, sensor}\} \quad (2.1)$$

#### Indexes: arrays and vectors starting with 0:

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

#### Dimensionality of objects and vectors:

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

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

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

- `referenceCoordinates` is a 2D vector (but could be any dimension in general nodes)
- measuring the current position of `NodePoint2D` gives a 3D vector
- when attaching a `MarkerNodePosition` and a `LoadForceVector`, the force will be still a 3D vector

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

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

In this section, the most important part of ExUDYN are provided. An overview of the interaction of the items is given in Fig. 2.3

### 2.2.1 Nodes

Nodes provide the coordinates (and the degrees of freedom) to the system. They have no mass, stiffness or whatsoever assigned. Without nodes, the system has no unknown coordinates. Adding a node provides (for the system unknown) coordinates. In addition we also need equations for every nodal coordinate – otherwise the system cannot be computed (NOTE: this is currently not checked by the preprocessor).

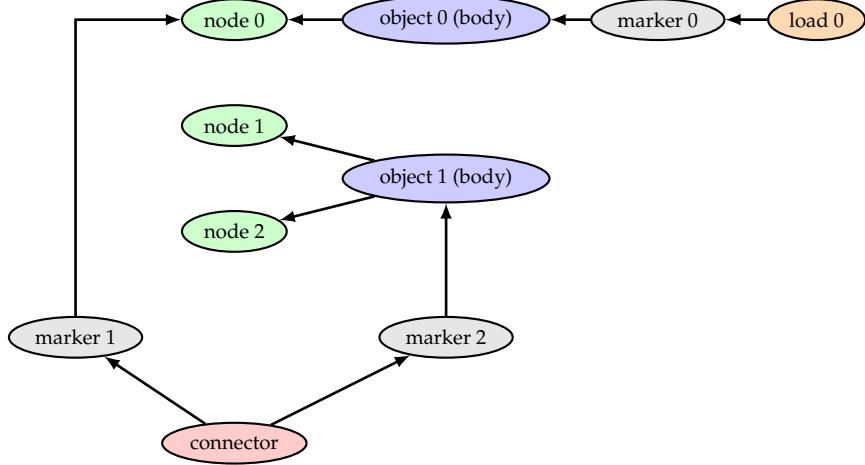


Figure 2.3: Typical interaction of items in a multibody system. Note that both, bodies and connectors/constraints are (computational) objects. The arrows indicate, that, e.g., object 1 has node 1 and node 2 (indexes) and that marker 0 is attached to object 0, while load 0 uses marker 0 to apply the load.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

## 2.2.2 Objects

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

Objects can be a:

- general object (e.g. a controller, user defined object, ...; no example yet)
- body: has a mass or mass distribution; markers can be placed on bodies; loads can be applied; constraints can be attached via markers; bodies can be:
  - ground object: has no nodes
  - simple body: has one node (e.g. mass point, rigid body)
  - finite element and more complicated body (e.g. FFRF-object): has more than one node
- connector: uses markers to connect nodes and/or bodies; adds additional terms to system equations either based on stiffness/damping or with constraints (and Lagrange multipliers). Possible connectors:
  - algebraic constraint (e.g. constrain two coordinates:  $q_1 = q_2$ )
  - classical joint
  - spring-damper or penalty constraint

## 2.2.3 Markers

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

Typical situations are:

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

#### 2.2.4 Loads

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

#### 2.2.5 Sensors

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

#### 2.2.6 Reference coordinates and displacements

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

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

### 2.3 Mapping between local and global coordinates

The local-to-global ([LTG](#))-mappings<sup>1</sup> between local coordinates, on node or object level, and global (=system) coordinates follows the following rules:

- [LTG](#)-mappings are computed during `mbs.Assemble()` and are not available before.
- Nodes own a global index which relates the local coordinates to global (system) coordinate. E.g., for a second order ordinary differential equations ([ODE2](#)) node with node number `i`, this index can be obtained via the function `mbs.GetNodeODE2Index(i)`.

---

<sup>1</sup>local-to-global coordinate mappings containing transformation from local object coordinate indices to global (system) coordinate indices

- The order of global coordinates is simply following the node numbering. If we add three nodes `NodePoint`, the system will contain 9 coordinates, where the first triple (starting index 0) belongs to node 0, the second triple (starting index 3) belongs to node 1 and the third triple (starting index 6) belongs to node 2. After `mbs.Assemble()`, you can access the system coordinates via `mbs.systemData.GetODE2Coordinates()`, which returns a numpy array with 9 coordinates, containing the initial values provided in `NodePoint` (default: zero).
- Objects have their own **LTG**-mappings for their respective coordinate types. The **ODE2** coordinates of an object `j` can be retrieved via `mbs.systemData.GetObjectLTGODE2(j)`. For a body, these are the global **ODE2** coordinates representing the body; for a connector, these are the coordinates to which the connector is linked (usually coordinates of two bodies); for a ground object, the **LTG**-mapping is empty; see also [Section 4.5.2](#).
- Constraints create algebraic variables (Lagrange multipliers) automatically. For a constraint with object number `k`, the global index to algebraic variables (of algebraic equations (**AE**)-type) can be accessed via `mbs.systemData.GetObjectLTGAE(k)`.

## 2.4 Exudyn Basics

This section will show:

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

### 2.4.1 Interaction with the EXUDYN module

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

#### Where do objects live?

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

## 2.4.2 Simulation settings

The simulation settings consists of a couple of substructures, e.g., for `solutionSettings`, `staticSolver`, `timeIntegration` as well as a couple of general options – for details see Sections 8.1.1 – 8.1.11.

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

The simulation settings structure is created with

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

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

```
tEnd = 10 #10 seconds of simulation time:  
h = 0.01 #step size (gives 1000 steps)  
simulationSettings.timeIntegration.endTime = tEnd  
#steps for time integration must be integer:  
simulationSettings.timeIntegration.numberOfSteps = int(tEnd/h)  
#assigns a new tolerance for Newton's method:  
simulationSettings.timeIntegration.newton.relativeTolerance = 1e-9  
#write some output while the solver is active (SLOWER):  
simulationSettings.timeIntegration.verboseMode = 2  
#write solution every 0.1 seconds:  
simulationSettings.solutionSettings.solutionWritePeriod = 0.1  
#use sparse matrix storage and solver (package Eigen):  
simulationSettings.linearSolverType = exu.LinearSolverType.EigenSparse
```

## 2.4.3 Generating output and results

The solvers provide a number of options in `solutionSettings` to generate a solution file. As a default, exporting solution to the solution file is activated with a writing period of 0.01 seconds.

Typical output settings are:

```
#create a new simulationSettings structure:  
simulationSettings = exu.SimulationSettings()  
  
#activate writing to solution file:  
simulationSettings.solutionSettings.writeSolutionToFile = True  
#write results every 1ms:  
simulationSettings.solutionSettings.solutionWritePeriod = 0.001  
  
#assign new filename to solution file  
simulationSettings.solutionSettings.coordinatesSolutionFileName= "myOutput.txt"  
  
#do not export certain coordinates:  
simulationSettings.solutionSettings.exportDataCoordinates = False
```

## 2.4.4 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 8.2.1 – 8.2.15.

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

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

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

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

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

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

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

### 2.4.4.1 Storing the model view

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

```
import exudyn as exu
SC=exu.SystemContainer()
SC.visualizationSettings.general.autoFitScene = False #prevent from autozoom
```

---

<sup>2</sup>note that `visualizationSettings.general.autoFitScene` should be set False if you want to use the stored zoom factor

```

exu.StartRenderer()
if 'renderState' in exu.sys:
    SC.SetRenderState(exu.sys['renderState'])
#####
#do simulation here and adjust model view settings with mouse
#####

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

```

---

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

```

In[1] : SC.GetRenderState()
Out[1]:
{'centerPoint': [1.0, 0.0, 0.0],
 'maxSceneSize': 2.0,
 'zoom': 1.0,
 'currentWindowSize': [1024, 768],
 'modelRotation': [[ 0.34202015, 0. , 0.9396926 ],
                   [-0.60402274, 0.76604444, 0.21984631],
                   [-0.7198463 , -0.6427876 , 0.26200265]]}

```

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

```

SC.visualizationSettings.general.autoFitScene = False #prevent from autozoom
exu.StartRenderer()
renderState={'centerPoint': [1.0, 0.0, 0.0],
             'maxSceneSize': 2.0,
             'zoom': 1.0,
             'currentWindowSize': [1024, 768],
             'modelRotation': [[ 0.34202015, 0. , 0.9396926 ],
                               [-0.60402274, 0.76604444, 0.21984631],
                               [-0.7198463 , -0.6427876 , 0.26200265]]}

SC.SetRenderState(renderState)
#.... further code for simulation here

```

---

## 2.4.5 Graphics pipeline

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

- compute new step
- finish computation step; results are in current state
- copy current state to visualization state (thread safe)

- signal graphics pipeline that new visualization data is available

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

- render openGL scene with a given graphicsData structure (containing lines, faces, text, ...)
- go idle for some milliseconds
- check if openGL rendering needs an update (e.g. due to user interaction)
  - 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 prescribed value, the graphicsData structure is updated with the new visualization state

#### 2.4.6 Graphics user Python functions

There are some user functions in order to customize drawing:

- You can assign graphicsData to the visualization to most bodies, such as rigid bodies in order to change the shape. Graphics can also be imported from files (`GraphicsDataFromSTLfileTxt`) using the established format STereoLithography (**STL**)<sup>3</sup>.
- Some objects, e.g., `ObjectGenericODE2` or `ObjectRigidBody`, provide customized a function `graphicsDataUserFunction`. This user function just returns a list of `GraphicsData`, see [Section 9.3](#). With this function you can change the shape of the body in every step of the computation.
- Specifically, the `graphicsDataUserFunction` in `ObjectGround` can be used to draw any moving background in the scene.

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

#### 2.4.7 Color and RGBA

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

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

E.g., red color with no transparency is obtained by the `color=[1,0,0,1]`. Color predefinitions are found in `exudynGraphicsDataUtilities.py`, e.g., `color4red` or `color4steelblue` as well a list of 10 colors `color4list`, which is convenient to be used in a loop creating objects.

---

<sup>3</sup>STereoLithography or Standard Triangle Language; file format available in nearly all CAD systems

## 2.4.8 Camera following objects and interacting with model view

For some models, it may be advantageous to track the translation and/or rotation of certain bodies, e.g., for cars, (wheeled) robots or bicycles. To do so, the current render state (`SC.GetRenderState()`, `SC.SetRenderState(...)`) can be obtained and modified, in order to always follow a certain position. As this needs to be done during redraw of every frame, it is conveniently done in a `graphicsUserFunction`, e.g., within the ground body. This is shown in the following example, in which `mbs.variables['nTrackNode']` is a node number to be tracked:

```
#mbs.variables['nTrackNode'] contains node number
def UFgraphics(mbs, objectNum):
    n = mbs.variables['nTrackNode']
    p = mbs.GetNodeOutput(n,exu.OutputVariableType.Position,
                          configuration=exu.ConfigurationType.Visualization)
    rs=SC.GetRenderState() #get current render state
    A = np.array(rs['modelRotation'])
    p = A.T @ p #transform point into model view coordinates
    rs['centerPoint']=[p[0],p[1],p[2]]
    SC.SetRenderState(rs) #modify render state
    return []

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

## 2.4.9 Solution viewer

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

- `utilities.AnimateSolution`, see Section 5.23
- `interactive.SolutionViewer`, see Section 5.7
- `interactive.AnimateModes`, lets you view the animation of computed modes, see Section 5.7

The function `AnimateSolution` allows to directly visualize the stored solution for according stored time frames. The `SolutionViewer` adds a `tkinter` interactive dialog, which lets you interact with the model (‘Player’). In both methods `AnimateSolution` and `SolutionViewer`, the solution needs to be loaded with `LoadSolutionFile('coordinatesSolution.txt')`, where ‘coordinatesSolution.txt’ represents the stored solution file, see

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

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

```

from exodyn.interactive import SolutionViewer
sol = LoadSolutionFile('coordinatesSolution.txt')
SolutionViewer(mbs, sol)

```

Alternatively, you can just reload the last stored solution (according to your `simulationSettings`):

```

from exodyn.interactive import SolutionViewer
SolutionViewer(mbs)

```

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

## 2.4.10 Generating animations

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

Animations are created based on a series of images (frames, snapshots) taken during simulation. It is important, that the current view is used to record these images – this means that the view should not be changed during the recording of images. To turn on recording of images during solving, set the following flag to a positive value

- `simulationSettings.solutionSettings.recordImagesInterval = 0.01`

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

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

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

To create animation files, an external tool FFmpeg is used to efficiently convert a series of images into an animation. 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 frame0000.png, frame0001.png, ... files to a video
REM for higher quality use crf option (standard: -crf 23, range: 0-51, lower crf value means higher
quality)

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

```

---

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

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

## 2.4.11 Contact problems

Since Q4 2021 a contact module is available in ExUDYN. This separate module `GeneralContact` is highly optimized and implemented with parallelization (multi-threaded) for certain types of contact elements.

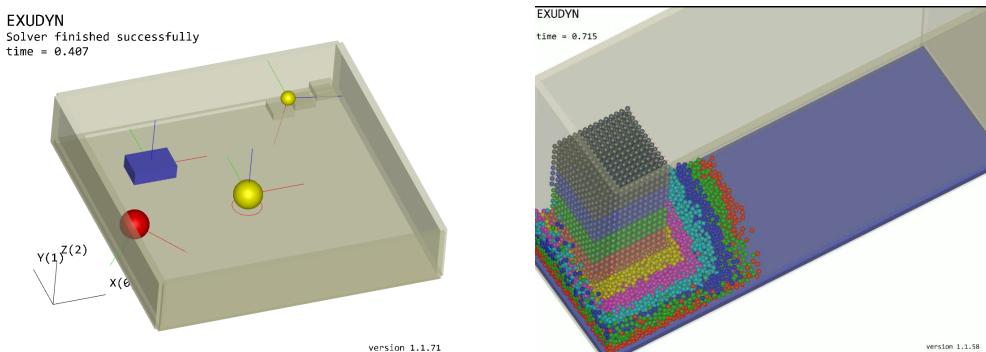


Figure 2.4: Some tests and examples using `GeneralContact`.

### Note:

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

Currently, `GeneralContact` includes:

- Sphere-Sphere contact (attached to any marker); may represent circle-circle contact in 2D
- Triangles mounted on rigid bodies, in contact with Spheres
- ANCF Cable 2D contacting with spheres (which then represent circles in 2D)

For details on the contact formulations, see [Section 6.3](#).

## 2.4.12 Removing convergence problems and solver failures

Nonlinear formulations (such as most multibody systems, especially nonlinear finite elements) cause problems and there is no general nonlinear solver which may reliably and accurately solve such problems. Tuning solver parameters is at hand of the user. In general, the Newton solver tries to reduce the error by the factor given in `simulationSettings.staticSolver.newton.relativeTolerance` (for static solver), which is not possible for very small (or zero) initial residuals. The absolute tolerance is helping out as a lower bound for the error, given in `simulationSettings.staticSolver.newton.absoluteTolerance` (for static solver), which is by default rather low ( $1e-10$ ) – in order to achieve accurate results for small systems or small motion (in mm or  $\mu\text{m}$  regime). Increasing this value helps to solve such problems. Nevertheless, you should usually set tolerances as low as possible because otherwise, your solution may become inaccurate.

The following hints shall be followed (also some solver hints).

- **static solver:** load steps are reduced even if the solution seems to be smooth and less steps are expected; larger number of steps may happen for finer discretization; you may adjust (increase) `.newton.relativeTolerance / .newton.absoluteTolerance` in static solver or in time integration to resolve such problems, but check if solution achieves according accuracy
- **static solver:** load steps are reduced significantly for highly nonlinear problems; solver repeatedly writes that steps are reduced → try to use `loadStepGeometric` and use a large `loadStepGeometricRange`: this allows to start with very small loads in which the system is nearly linear (e.g. for thin strings or belts under gravity).
- **static solver:** in case that your system is (nearly) kinematic, a static solution can be achieved using `stabilizerODE2term`, which adds mass-proportional stiffness terms during load steps  $< 1$ .
- very small loads or even **zero loads** do not converge: `SolveDynamic` or `SolveStatic terminated due to errors`
  - the reason is the nonlinearity of formulations (nonlinear kinematics, nonlinear beam, etc.) and round off errors, which restrict Newton to achieve desired tolerances
  - adjust (increase) `.newton.relativeTolerance / .newton.absoluteTolerance` in static solver or in time integration
  - in many cases, especially for static problems, the `.newton.newtonResidualMode = 1` evaluates the increments; the nonlinear problems is assumed to be converged, if increments are within given absolute/relative tolerances; this also works usually better for kinematic solutions
- for **discontinuous problems**: try to adjust solver parameters; especially the `discontinuous.iterationTolerance` and `discontinuous.maxIterations`; try to make smaller load or time steps in order to resolve switching points of contact or friction; generalized alpha solvers may cause troubles when reducing step sizes → use `TrapezoidalIndex2` solver
- if you see further problems, please post them (including relevant example) at the ExUDYN github page!

### 2.4.13 Performance and ways to speed up computations

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

The following examples refer to `simulationSettings = exu.SimulationSettings()`. In general, to see where CPU time is lost, use the option turn on `simulationSettings.displayComputationTime = True` to see which parts of the solver need most of the time (deactivated in current Python3.8 version!).

#### Possible speed ups in general:

- for models with more than 50 coordinates, switching to sparse solvers might greatly improve speed: `simulationSettings.linearSolverType = exu.LinearSolverType.EigenSparse`
- try to avoid Python function or try to speed up Python functions
- instead of user functions in objects or loads (computed in every iteration), some problems would also work if these parameters are only updated in `mbs.SetPreStepUserFunction(...)`
- for **discontinuous problems**, try to adjust solver parameters; especially the `discontinuous.iterationTolerance` which may be too tight and cause many iterations; iterations may be limited by `discontinuous.maxIterations`, which at larger values solely multiplies the computation time with a factor if all iterations are performed

#### Possible speed ups for dynamic simulations:

- for implicit integration, turn on **modified Newton**, which updates jacobians only if needed: `simulationSettings.timeIntegration.newton.useModifiedNewton = True`
- switch to Python3.8: this version excludes range checks and timings; usually 30% faster
- use **multi-threading**: `simulationSettings.parallel.numberOfThreads = ...`, depending on the number of cores (larger values usually do not help); improves greatly for contact problems, but also for some objects computed in parallel; will improve significantly in future
- decrease number of steps (`simulationSettings.timeIntegration.numberOfSteps = int(tEnd/h)`) by increasing the step size  $h$  if not needed for accuracy reasons; note that in general, the solver will reduce steps in case of divergence, but not for accuracy reasons, which may still lead to divergence if step sizes are too large
- switch off measuring computation time, if not needed: `simulationSettings.displayComputationTime = False`
- try to switch to **explicit solvers**, if problem has no constraints and if problem is not stiff
- try to have **constant mass matrices** (see according objects, which have constant mass matrices; e.g. rigid bodies using RotationVector Lie group node have constant mass matrix)
- for explicit integration, set `computeEndOfStepAccelerations = False`, if you do not need accurate evaluation of accelerations at end of time step (will then be taken from beginning)
- if you are sure that your mass matrix is constant, set `simulationSettings.timeIntegration.reuseConstantMassMatrix = True`; check results!

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

## 2.5 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.

### 2.5.1 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.

### 2.5.2 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)

### 2.5.3 C++ Code: Modules

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

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

The following main external libraries are linked to Exudyn:

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

#### 2.5.4 Code style and conventions

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

- write simple code (no complicated structures or uncommon coding)
- write readable code (e.g., variables and functions with names that represent the content or functionality; AVOID abbreviations)
- put a header in every file, according to Doxygen format
- put a comment to every (global) function, member function, data member, template parameter
- ALWAYS USE curly brackets for single statements in 'if', 'for', etc.; example: if ( $i < n$ ) { $i += 1$ };
- use Doxygen-style comments (use '//' Qt style and '@ date' with '@' instead of 'for commands')
- use Doxygen (with preceding '@') '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

## 2.5.5 Notation conventions

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

- use lowerCamelCase for names of variables (including class member variables), consts, c-define variables, ...; EXCEPTION: for algorithms following formulas, e.g.,  $f = M * q_{tt} + K * q$ , GBar, ...
- use UpperCamelCase for functions, classes, structs, ...
- Special cases for CamelCase (with some exceptions that happened in the past ...):
  - continue upper case after upper case abbreviations in case of **functions or classes**: 'ODESystem', 'Point2DClass', 'ANCF Cable2D', 'ANCFALE', 'ComputeODE1Equations', ... (this is not always nice to read, but has become a standard and will be further used!)
  - for variables and class member variables continue **lower case**: 'nODE1variables', 'dim2Dspecial', 'ANCFsize'
  - abbreviations at beginning of expressions: for functions or classes use ODEComputeCoords(), for variables avoid 'ODE' at beginning: use 'nODE' or write 'odeCoordinates'
- '[...]Init' ... in arguments, for initialization of variables; e.g. 'valueInit' for initialization of member variable 'value'
- use American English throughout: Visualization, etc.
- AVOID consecutive capitalized words, e.g., avoid 'ODEAE'
- do not use '\_' within variable or function names; exception: derivatives
- use name which exactly describes the function/variable: 'numberOfItems' instead of 'size' or 'l'
- examples for variable names: secondOrderSize, massMatrix, mThetaTheta
- examples for function/class names: SecondOrderSize, EvaluateMassMatrix, Position(const Vector3D& localPosition)
- use the Get/Set...() convention if data is retrieved from a class (Get) or something is set in a class (Set); Use const T& Get() / T& Get if direct access to variables is needed; Use Get/Set for pybind11
- example Get/Set: Real\* GetDataPointer(), Vector::SetAll(Real), GetTransposed(), SetRotationalParam, SetColor(...), ...
- use 'Real' instead of double or float: for compatibility, also for AVX with SP/DP

- use 'Index' for array/vector size and index instead of `size_t` or `int`
- item: object, node, marker, load: anything handled within the computational/visualization systems
- Do not use numbers (3 for 3D or any other number which represents, e.g., the number of rotation parameters). Use `const Index` or `constexpr` to define constants.

### 2.5.6 No-abbreviations-rule

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

Exceptions to the NO-ABBREVIATIONS-RULE, see also [List of Abbreviations](#):

- ordinary differential equation ([ODE](#))
- [ODE2](#): marks parts related to second order differential equations (SOS2, `EvalF2` in HOTINT)
- first order ordinary differential equations ([ODE1](#)): marks parts related to first order differential equations (ES, `EvalF` in HOTINT)
- [AE](#); note: using the term 'AEcoordinates' for 'algebraicEquationsCoordinates'
- 'C[...]' ... Computational, e.g. for `ComputationalNode` ==> use 'CNode'
- minimum ([min](#)), maximum ([max](#))
- absolute (e.g., absolute error), absolute value ([abs](#)), relative (e.g., relative error) ([rel](#))
- triangle (in graphics) ([trig](#))
- quadrangle, polygon with 4 vertices ([quad](#))
- [RHS](#)
- [LHS](#)
- Euler parameters ([EP](#))
- rotation parameterization: consecutive rotations around x, y and z-axis (Tait-Bryan) ([Rxyz](#))
- coefficients ([coeffs](#))
- position ([pos](#))
- write time derivatives with underscore: `_t`, `_tt`; example: `Position_t`, `Position_tt`, ...
- write space-wise derivatives ith underscore: `_x`, `_xx`, `_y`, ...
- if a scalar, write coordinate derivative with underscore: `_q`, `_v` (derivative w.r.t. velocity coordinates)
- for components, elements or entries of vectors, arrays, matrices: use 'item' throughout
- '[...]Init' ... in arguments, for initialization of variables; e.g. 'valueInit' for initialization of member variable 'value'

## 2.6 Changes

For continuous tracking of changes, see [Section 12](#).



# Chapter 3

## Tutorial

This section will show:

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

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

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

### 3.1 Mass-Spring-Damper tutorial

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

```
main/pythonDev/Examples/springDamperTutorial.py
```

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

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

```
import exudyn as exu  
from exudyn.itemInterface import *  
import numpy as np #for postprocessing
```

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

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

In order to check, which version you are using, you can printout the current EXUDYN version. 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<sup>1</sup>:

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

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

While `Point` adds 3 unknown coordinates to the system, which need to be solved, we also can add ground nodes, which can be used similar to nodes, but they do not have unknown coordinates – and therefore also have no initial displacements or velocities. The advantage of ground nodes (and ground bodies) is that no constraints are needed to fix these nodes. Such a ground node is added via:

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

In the next step, we add an object<sup>2</sup>, 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))
```

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

---

<sup>1</sup>Note: Point is an abbreviation for NodePoint, defined in `itemInterface.py`.

<sup>2</sup>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.

level, both for forces as well as for constraints. Markers are attached to the according ground and regular node number, additionally using a coordinate number (0 ... first coordinate):

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

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

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

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

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

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

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

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

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:

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

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

```
simulationSettings = exu.SimulationSettings()
simulationSettings.solutionSettings.solutionWritePeriod = 5e-3 #output interval general
simulationSettings.solutionSettings.sensorsWritePeriod = 5e-3 #output interval of sensors
simulationSettings.timeIntegration.numberOfSteps = int(tEnd/h) #must be integer
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 (in render
                           window!)
```

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:

```
exu.SolveDynamic(mbs, 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
steps = int(tEnd/h)                   #use same steps for reference solution

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

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

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

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

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

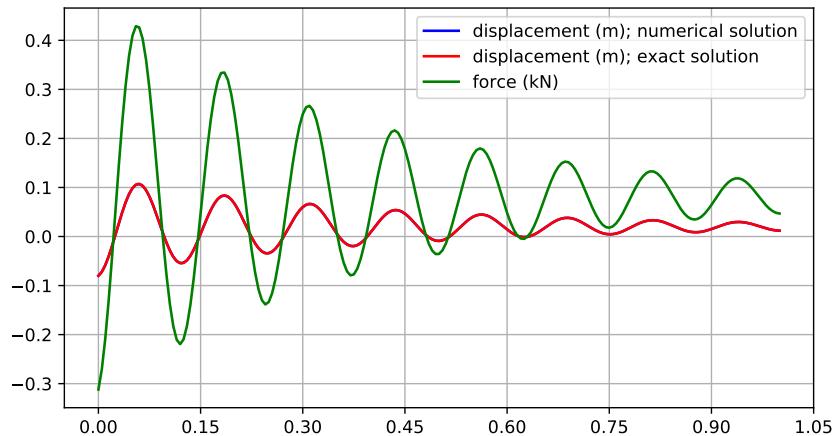
The sensor result can be loaded in the same way. The sensor output format contains time in the first column and sensor values in the remaining columns. The number of columns depends on the sensor and the output quantity (scalar, vector, ...):

```
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<sup>4</sup>:

```
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:




---

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

<sup>4</sup>note, in some environments you need finally the command plt.show()

## 3.2 Rigid body and joints tutorial

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

```
main/pythonDev/Examples/rigidBodyTutorial3.py
```

This tutorial will set up a multibody system containing a ground, two rigid bodies and two revolute joints driven by gravity, compare a 3D view of the example in Fig. 3.1.

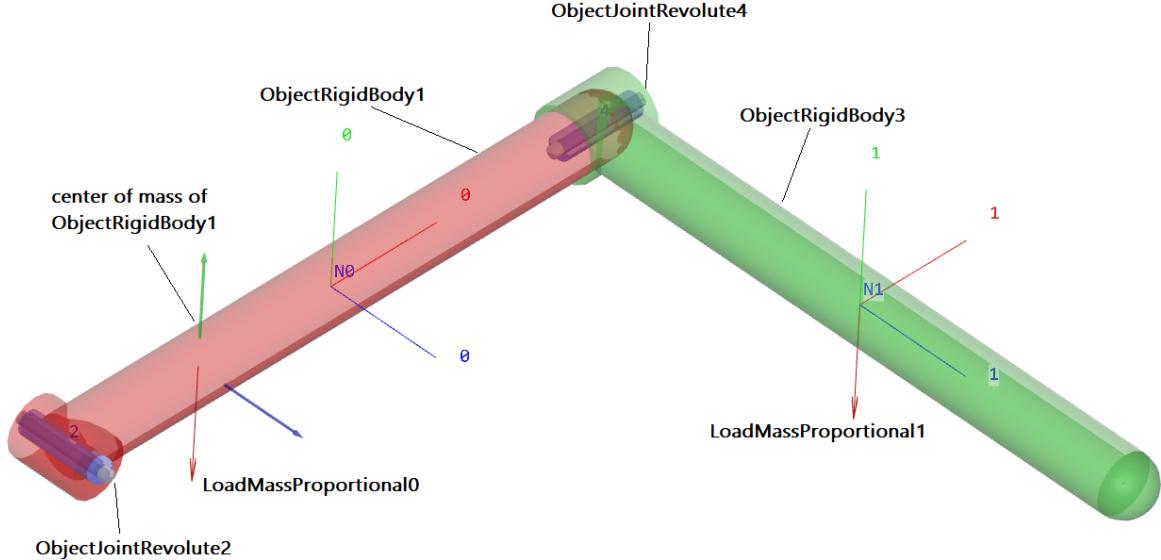


Figure 3.1: Render view of rigid body tutorial, showing objects, nodes ( $N_0, N_1$ ), and loads.

---

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

```
import exudyn as exu
from exudyn.itemInterface import *
from exudyn.utilities import *
import numpy as np #for postprocessing
```

The submodule `exudyn.utilities` contains helper functions for graphics representation, 3D rigid bodies and joints.

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

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

We define some geometrical parameters for lateron use.

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

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

```
#ground body
oGround = mbs.AddObject(ObjectGround())
```

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

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

Note that the COM is translated in axial direction, while it would be at the body's local position [0,0,0] by default!

For visualization, we need to add some graphics for the body defined as a 3D RigidLink object and we additionally draw a basis (three RGB-vectors) at the COM:

```
#graphics for body
graphicsBody0 = GraphicsDataRigidLink(p0=[-0.5*L, 0, 0], p1=[0.5*L, 0, 0],
                                       axis0=[0, 0, 1], axis1=[0, 0, 0], radius=[0.5*w, 0.5*w],
                                       thickness=w, width=[1.2*w, 1.2*w], color=color4red)
graphicsCOM0 = GraphicsDataBasis(origin=iCube0.com, length=2*w)
```

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

```
[n0,b0]=AddRigidBody(mainSys = mbs,
                      inertia = iCube0, #includes COM
                      nodeType = exu.NodeType.RotationEulerParameters,
                      position = pMid0,
                      rotationMatrix = np.diag([1,1,1]),
                      gravity = g,
                      graphicsDataList = [graphicsBody0, graphicsCOM0])
```

which also adds a gravity load and could also set initial velocities, if wanted. The `nodeType` specifies the underlying model for the rigid body node, see [Section 4.9.4](#). We can use

- `RotationEulerParameters`: for fast computation, but leads to an additional algebraic equation and thus needs an implicit solver
- `RotationRxyz`: contains a singularity if the second angle reaches +/- 90 degrees, but no algebraic equations
- `RotationRotationVector`: basically contains a singularity for 0 degrees, but if used in combination with Lie group integrators, singularities are bypassed

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

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

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

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

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

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

Additional transformation matrices (`rotationMarker0/1`) can be added in order to chose any rotation axis.

Note that an error in the definition of markers for the joints can be also detected in the render window (if you completed the example), e.g., if you change the following marker in the lines above,

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

→ you will see a misalignment of the two parts of the joint by  $0.1*L$ .

Due to the fact that the definition of markers for general joints is tedious, there is a utility function, which allows to attach revolute joints immediately to bodies and defining the rotation axis only once for the joint:

```
#revolute joint option 3:
AddRevoluteJoint(mbs, body0=oGround, body1=b0, point=[0,0,0],
                  axis=[0,0,1], useGlobalFrame=True, showJoint=True,
                  axisRadius=0.2*w, axisLength=1.4*w)
```

The second link and the according joint can be set up in a very similar way:

```
#second link:
graphicsBody1 = GraphicsDataRigidLink(p0=[0,0,-0.5*L],p1=[0,0,0.5*L],
                                         axis0=[1,0,0], axis1=[0,0,0], radius=[0.06,0.05],
```

```

        thickness = 0.1, width = [0.12,0.12],
        color=color4lightgreen)

iCube1 = InertiaCuboid(density=5000, sideLengths=[0.1,0.1,1])

pMid1 = np.array([L,0,0]) + np.array([0,0,0.5*L]) #center of mass, body1
[n1,b1]=AddRigidBody(mainSys = mbs,
                        inertia = iCube1,
                        nodeType = exu.NodeType.RotationEulerParameters,
                        position = pMid1,
                        rotationMatrix = np.diag([1,1,1]),
                        angularVelocity = [0,0,0],
                        gravity = g,
                        graphicsDataList = [graphicsBody1])

```

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

```

#revolute joint (free x-axis)
AddRevoluteJoint(mbs, body0=b0, body1=b1, point=[L,0,0],
                    axis=[1,0,0], useGlobalFrame=True, showJoint=True,
                    axisRadius=0.2*w, axisLength=1.4*w)

```

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

```

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

```

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

```
mbs.Assemble()
```

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

```

<systemData:
    Number of nodes= 2
    Number of objects = 5
    Number of markers = 8
    Number of loads = 2
    Number of sensors = 1
    Number of ODE2 coordinates = 14
    Number of ODE1 coordinates = 0
    Number of AE coordinates = 12
    Number of data coordinates = 0

```

For details see `mbs.systemData`, `mbs.sys` and `mbs.variables`

>

---

Note that there are 2 nodes for the two rigid bodies. The five objects are due to ground object, 2 rigid bodies and 2 revolute joints. The meaning of markers can be seen in the graphical representation described below.

Alternatively we can print the full internal information as a dictionary using:

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

which results in the following output (shortened):

---

```
node0:  
  {'nodeType': 'RigidBodyEP', 'referenceCoordinates': [0.5, 0.0, 0.0, 1.0, 0.0, 0.0,  
  0.0], 'addConstraintEquation': True, 'initialCoordinates': [0.0, 0.0, 0.0, 0.0, 0.0,  
  0.0], 'initialVelocities': [0.0, 0.0, 0.0, 0.0, 0.0, 0.0], 'name': 'node0', '  
  Vshow': True, 'VdrawSize': -1.0, 'Vcolor': [-1.0, -1.0, -1.0, -1.0]}  
node1:  
  {'nodeType': 'RigidBodyEP', 'referenceCoordinates': [1.0, 0.0, 0.5, 1.0, 0.0, 0.0,  
  0.0], 'addConstraintEquation': True, 'initialCoordinates': [0.0, 0.0, 0.0, 0.0, 0.0,  
  0.0], 'initialVelocities': [0.0, 0.0, 0.0, 0.0, 0.0, 0.0], 'name': 'node1', '  
  Vshow': True, 'VdrawSize': -1.0, 'Vcolor': [-1.0, -1.0, -1.0, -1.0]}  
object0:  
  {'objectType': 'Ground', 'referencePosition': [0.0, 0.0, 0.0], 'name': 'object0', '  
  Vshow': True, 'VgraphicsDataUserFunction': 0, 'Vcolor': [-1.0, -1.0, -1.0, -1.0], '  
  VgraphicsData': {'TODO': 'Get graphics data to be implemented'}}  
object1:  
  {'objectType': 'RigidBody', 'physicsMass': 50.0, 'physicsInertia':  
  [0.0833333333333336, 7.33333333333334, 7.33333333333334, 0.0, 0.0, 0.0], '  
  physicsCenterOfMass': [-0.25, 0.0, 0.0], 'nodeNumber': 0, 'name': 'object1', 'Vshow':  
  True, 'VgraphicsDataUserFunction': 0, 'VgraphicsData': {'TODO': 'Get graphics data to be  
  implemented'}}  
object2:  
  {'objectType': 'JointRevolute', 'markerNumbers': [3, 4], 'rotationMarker0': [[0.0,  
  1.0, 0.0], [-1.0, 0.0, 0.0], [0.0, 0.0, 1.0]], 'rotationMarker1': [[0.0, 1.0, 0.0],  
  [-1.0, 0.0, 0.0], [0.0, 0.0, 1.0]], 'activeConnector': True, 'name': 'object2', 'Vshow':  
  True, 'VaxisRadius': 0.01999999552965164, 'VaxisLength': 0.14000000059604645, 'Vcolor':  
  [-1.0, -1.0, -1.0, -1.0]}  
object3:  
  ...
```

---

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

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

For the output see Fig. 3.2. Note that obviously, markers are always needed to connect objects (or nodes) as well as loads. We can also see, that 2 markers MarkerBodyRigid1 and MarkerBodyRigid2 are unused, which is no further problem for the model and also does not require additional computational resources (except for some bytes of memory). Having isolated nodes or joints that are not connected

(or having too many connections) may indicate that you did something wrong in setting up your model.

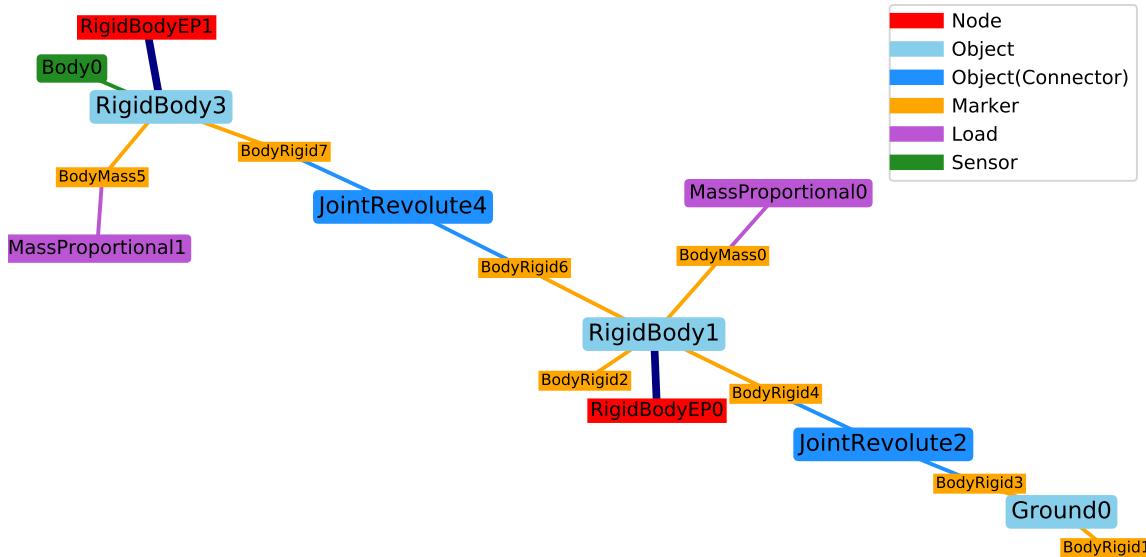


Figure 3.2: System graph for rigid body tutorial. Numbers are always related to the node number, object number, etc.; note that colors are used for nodes, objects, markers, etc. → the label names (which do not include Object . . . , Node . . . ) need to be read with the color information!

---

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

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

tEnd = 4 #simulation time
h = 1e-3 #step size
simulationSettings.timeIntegration.numberOfSteps = int(tEnd/h)
simulationSettings.timeIntegration.endTime = tEnd
simulationSettings.timeIntegration.verboseMode = 1
#simulationSettings.timeIntegration.simulateInRealtime = True
simulationSettings.solutionSettings.solutionWritePeriod = 0.005 #store every 5 ms
```

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

In order to improve visualization, there are hundreds of options, see Visualization settings in [Section 8.2.1](#), some of them used here:

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

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

The option `autoFitScene` is used in order to avoid zooming while loading the last saved render state, see below.

We can start the 3D visualization (Renderer) now:

```
exu.StartRenderer()
```

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

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

mbs.WaitForUserToContinue() #stop before simulating
```

the function `WaitForUserToContinue()` waits with simulation until we press SPACE bar. This allows us to make some pre-checks.

Finally, the **index 2** (velocity level) implicit time integration (simulation) is started with:

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

This solver is used in the present example, but should be considered with care as it leads to (small) drift of position constraints, linearly increasing in time. Using sufficiently small time steps, this effect is often negligible on the advantage of having a **energy-conserving integrator** (guaranteed for linear systems, but very often also for the nonlinear multibody system). Due to the velocity level, the integrator is less sensitive to consistent initial conditions on position level and compatible to frequent step size changes, however, initial jumps in velocities may never damp out in undamped systems. Alternatively, an **index 3** implicit time integration – the generalized- $\alpha$  method – is started with:

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

Note that the **generalized- $\alpha$  method** includes numerical damping (adjusted with the spectral radius) for stabilization of index 3 constraints. This leads to effects every time the integrator is (re-)started, e.g., when adapting time step sizes. For fixed step sizes, this is **the recommended integrator**.

After simulation, the library would immediately exit (and jump back to iPython or close the terminal window). In order to avoid this, we can use `WaitForRenderEngineStopFlag()` to wait until we press key 'Q'.

```
SC.WaitForRenderEngineStopFlag() #stop before closing
exu.StopRenderer() #safely close rendering window!
```

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

```
sol = LoadSolutionFile('coordinatesSolution.txt')
from exudyn.interactive import SolutionViewer
SolutionViewer(mbs, sol)
```

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

```
from exudyn.plot import PlotSensor
PlotSensor(mbs, [sens1],[1])
```

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



# Chapter 4

## Python-C++ command interface

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

### 4.1 General information on Python interface

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):

```
import exudyn as exu
```

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

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

Everything you work with is provided by the class `SystemContainer`, except for some very basic system functionality (which is inside the EXUDYN module).

You can create a new `SystemContainer`, which is a class that is initialized by assigning a system container to a variable, usually denoted as 'SC':

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

Note that creating a second `exu.SystemContainer()` will be independent of SC and therefore usually makes no sense.

Furthermore, there are a couple of commands available directly in the EXUDYN module, given in the following subsections. Regarding the **(basic) module access**, functions are related to the '`exudyn = exu`' module, see these examples:

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

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

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

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

#### 4.1.1 Item index

Many functions will work with node numbers ('NodeIndex'), object numbers ('ObjectIndex'), marker numbers ('MarkerIndex') and others. These numbers are special objects, which have been introduced in order to avoid mixing up, e.g., node and object numbers. For example, the command mbs.AddNode(...) returns a NodeIndex. For these indices, the following rules apply:

- mbs.Add[Node|Object|...](...) returns a specific NodeIndex, ObjectIndex, ...
- You can create any item index, e.g., using ni = NodeIndex(42) or oi = ObjectIndex(42)
- You can convert any item index, e.g., NodeIndex ni into an integer number using int(ni) or no.GetIndex()
- Still, you can use integers as initialization for item numbers, e.g.,
 

```
mbs.AddObject(MassPoint(nodeNumber=13, ...))
```

 However, it must be a pure integer type.
- You can also print item indices, e.g., print(ni) as it converts to string by default
- If you are unsure about the type of an index, use ni.GetTypeString() to show the index type

## 4.2 EXUDYN

These are the access functions to the EXUDYN module.

function/structure name	description
GetVersionString()	Get Exudyn module version as string
RequireVersion(requiredVersionString)	Checks if the installed version is according to the required version. Major, micro and minor version must agree the required level. Example: RequireVersion("1.0.31")
StartRenderer(verbose = false)	Start OpenGL rendering engine (in separate thread); use verbose=True to output information during OpenGL window creation; some of the information will only be seen in windows command (powershell) windows or linux shell, but not inside iPython or Spyder

StopRenderer()	Stop OpenGL rendering engine
DoRendererIdleTasks(waitSeconds = 0)	Call this function in order to interact with Renderer window; use waitSeconds in order to run this idle tasks while animating a model (e.g. waitSeconds=0.04), use waitSeconds=0 without waiting, or use waitSeconds=-1 to wait until window is closed
SolveStatic(mbs, simulationSettings = exudyn.SimulationSettings(), updateInitialValues = False, storeSolver = True)	Static solver function, mapped from module solver; for details on the Python interface see <a href="#">Section 5.22</a> ; for background on solvers, see <a href="#">Section 11</a>
SolveDynamic(mbs, simulationSettings = exudyn.SimulationSettings(), solverType = exudyn.DynamicSolverType.GeneralizedAlpha, updateInitialValues = False, storeSolver = True)	Dynamic solver function, mapped from module solver; for details on the Python interface see <a href="#">Section 5.22</a> ; for background on solvers, see <a href="#">Section 11</a>
ComputeODE2Eigenvalues(mbs, simulationSettings = exudyn.SimulationSettings(), useSparseSolver = False, numberEigenvalues = -1, setInitialValues = True, convert2Frequencies = False)	Simple interface to scipy eigenvalue solver for eigenvalue analysis of the second order differential equations part in mbs, mapped from module solver; for details on the Python interface see <a href="#">Section 5.22</a>
SetOutputPrecision(numberOfDigits)	Set the precision (integer) for floating point numbers written to console (reset when simulation is started!); NOTE: this affects only floats converted to strings inside C++ exudyn; if you print a float from Python, it is usually printed with 16 digits; if printing numpy arrays, 8 digits are used as standard, to be changed with numpy.set_printoptions(precision=16); alternatively convert into a list
SetLinalgOutputFormatPython(flagPythonFormat)	True: use Python format for output of vectors and matrices; False: use matlab format
SetWriteToConsole(flag)	set flag to write (True) or not write to console; default = True
SetWriteToFile(filename, flagWriteToFile = true, flagAppend = false)	set flag to write (True) or not write to console; default value of flagWriteToFile = False; flagAppend appends output to file, if set True; in order to finalize the file, write exu.SetWriteToFile("", False) to close the output file <b>EXAMPLE:</b> <code>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</code>
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
SuppressWarnings(flag)	set flag to suppress (=True) or enable (=False) warnings

InfoStat(writeOutput = true)	Retrieve list of global information on memory allocation and other counts as list:[array_new_counts, array_delete_counts, vector_new_counts, vector_delete_counts, matrix_new_counts, matrix_delete_counts, linkedDataVectorCast_counts]; May be extended in future; if writeOutput==True, it additionally prints the statistics; counts for new vectors and matrices should not depend on numberOfSteps, except for some objects such as ObjectGenericODE2 and for (sensor) output to files; Not available if code is compiled with __FAST_EXUDYN_LINALG flag
Go()	Creates a SystemContainer SC and a main system mbs
InvalidIndex()	This function provides the invalid index, which may depend on the kind of 32-bit, 64-bit signed or unsigned integer; e.g. node index or item index in list; currently, the InvalidIndex() gives -1, but it may be changed in future versions, therefore you should use this function
variables	this dictionary may be used by the user to store exudyn-wide data in order to avoid global Python variables; usage: exu.variables["myvar"] = 42
sys	this dictionary is used and reserved by the system, e.g. for testsuite, graphics or system function to store module-wide data in order to avoid global Python variables; the variable exu.sys['renderState'] contains the last render state after exu.StopRenderer() and can be used for subsequent simulations

## 4.3 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
Reset()	delete all systems and reset SystemContainer (including graphics); this also releases SystemContainer from the renderer, which requires SC.AttachToRenderEngine() to be called in order to reconnect to rendering; a safer way is to delete the current SystemContainer and create a new one (SC=SystemContainer() )
AddSystem()	add a new computational system

NumberOfSystems()	obtain number of systems available in system container
GetSystem(systemNumber)	obtain systems with index from system container
visualizationSettings	this structure is read/writeable and contains visualization settings, which are immediately applied to the rendering window.  EXAMPLE: <code>SC = exu.SystemContainer() SC.visualizationSettings.autoFitScene=False</code>
GetRenderState()	Get dictionary with current render state (openGL zoom, modelview, etc.); will have no effect if GLFW_GRAPHICS is deactivated  EXAMPLE: <code>SC = exu.SystemContainer() renderState = SC.GetRenderState() print(renderState['zoom'])</code>
SetRenderState(renderState)	Set current render state (openGL zoom, modelview, etc.) with given dictionary; usually, this dictionary has been obtained with GetRenderState; will have no effect if GLFW_GRAPHICS is deactivated  EXAMPLE: <code>SC = exu.SystemContainer() SC.SetRenderState(renderState)</code>
RedrawAndSaveImage()	Redraw openGL scene and save image (command waits until process is finished)
WaitForRenderEngineStopFlag()	Wait for user to stop render engine (Press 'Q' or Escape-key)
RenderEngineZoomAll()	Send zoom all signal, which will perform zoom all at next redraw request
RedrawAndSaveImage()	Redraw openGL scene and save image (command waits until process is finished)
AttachToRenderEngine()	Links the SystemContainer to the render engine, such that the changes in the graphics structure drawn upon updates, etc.; done automatically on creation of SystemContainer; return False, if no renderer exists (e.g., compiled without GLFW) or cannot be linked (if other SystemContainer already linked)
DetachFromRenderEngine()	Releases the SystemContainer from the render engine; return True if successfully released, False if no GLFW available or detaching failed
GetCurrentMouseCoordinates(useOpenGLcoordinates = False)	Get current mouse coordinates as list [x, y]; x and y being floats, as returned by GLFW, measured from top left corner of window; use GetCurrentMouseCoordinates(useOpenGLcoordinates=True) to obtain OpenGLcoordinates of projected plane

## 4.4 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 exdyn as exu
SC = exu.SystemContainer()
mbs = SC.AddSystem()
```

function/structure name	description
Assemble()	assemble items (nodes, bodies, markers, loads, ...); Calls CheckSystemIntegrity(...), AssembleCoordinates(), AssembleLTGLists(), AssembleInitializeSystemCoordinates(), and AssembleSystemInitialize()
AssembleCoordinates()	assemble coordinates: assign computational coordinates to nodes and constraints (algebraic variables)
AssembleLTGLists()	build <a href="#">LTG</a> coordinate lists for objects (used to build global ODE2RHS, MassMatrix, etc. vectors and matrices) and store special object lists (body, connector, constraint, ...)
AssembleInitializeSystemCoordinates()	initialize all system-wide coordinates based on initial values given in nodes
AssembleSystemInitialize()	initialize some system data, e.g., generalContact objects (searchTree, etc.)
Reset()	reset all lists of items (nodes, bodies, markers, loads, ...) and temporary vectors; deallocate memory
GetSystemContainer()	return the systemContainer where the mainSystem (mbs) was created
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
ActivateRendering(flag = true)	activate (flag=True) or deactivate (flag=False) rendering for this system

SetPreStepUserFunction()	<p>Sets a user function PreStepUserFunction(mbs, t) executed at beginning of every computation step; in normal case return True; return False to stop simulation after current step</p> <p><b>EXAMPLE:</b></p> <pre>def PreStepUserFunction(mbs, t):     print(mbs.systemData.NumberOfNodes())     if(t&gt;1):         return False     return True mbs.SetPreStepUserFunction(PreStepUserFunction)</pre>
SetPostNewtonUserFunction()	<p>Sets a user function PostNewtonUserFunction(mbs, t) executed after successful Newton iteration in implicit or static solvers and after step update of explicit solvers, but BEFORE PostNewton functions are called by the solver; function returns list [discontinuousError, recommendedStepSize], containing a error of the PostNewtonStep, which is compared to [solver].discontinuous.iterationTolerance. The recommendedStepSize shall be negative, if no recommendation is given, 0 in order to enforce minimum step size or a specific value to which the current step size will be reduced and the step will be repeated; use this function, e.g., to reduce step size after impact or change of data variables</p> <p><b>EXAMPLE:</b></p> <pre>def PostNewtonUserFunction(mbs, t):     if(t&gt;1):         return [0, 1e-6]     return [0,0] mbs.SetPostNewtonUserFunction(PostNewtonUserFunction)</pre>
AddGeneralContact()	add a new general contact, used to enable efficient contact computation between objects (nodes or markers)
GetGeneralContact(generalContactNumber)	get read/write access to GeneralContact with index 'generalContactNumber' stored in mbs
DeleteGeneralContact(generalContactNumber)	delete GeneralContact with index 'generalContactNumber' in mbs; other general contacts are resorted (index changes!)
__repr__()	<p>return the representation of the system, which can be, e.g., printed</p> <p><b>EXAMPLE:</b></p> <pre>print(mbs)</pre>
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,...)

#### 4.4.1 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)	<p>add a node with nodeDefinition from Python node class; returns (global) node index (type NodeIndex) of newly added node; use int(nodeIndex) to convert to int, if needed (but not recommended in order not to mix up index types of nodes, objects, markers, ...)</p> <p><b>EXAMPLE:</b></p> <pre>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)</pre>
GetNodeNumber(nodeName)	<p>get node's number by name (string)</p> <p><b>EXAMPLE:</b></p> <pre>n = mbs.GetNodeNumber('example node')</pre>

GetNode(nodeNumber)	get node's dictionary by node number (type NodeIndex) <b>EXAMPLE:</b> <code>mbs.GetNode(0)</code>
ModifyNode(nodeNumber, nodeDict)	modify node's dictionary by node number (type NodeIndex) <b>EXAMPLE:</b> <code>mbs.ModifyNode(nodeNumber, nodeDict)</code>
GetNodeDefaults(typeName)	get node's default values for a certain nodeType as (dictionary) <b>EXAMPLE:</b> <code>nodeType = 'Point' nodeDict = mbs.GetNodeDefaults(nodeType)</code>
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) <b>EXAMPLE:</b> <code>mbs.GetNodeOutput(nodeNumber=0, variableType=exu.OutputVariableType.Displacement)</code>
GetNodeODE2Index(nodeNumber)	get index in the global ODE2 coordinate vector for the first node coordinate of the specified node <b>EXAMPLE:</b> <code>mbs.GetNodeODE2Index(nodeNumber=0)</code>
GetNodeParameter(nodeNumber, parameterName)	get nodes's parameter from node number (type NodeIndex) and parameterName; parameter names can be found for the specific items in the reference manual; for visualization parameters, use a 'V' as a prefix <b>EXAMPLE:</b> <code>mbs.GetNodeParameter(0, 'referenceCoordinates')</code>
SetNodeParameter(nodeNumber, parameterName, value)	set parameter 'parameterName' of node with node number (type NodeIndex) to value; parameter names can be found for the specific items in the reference manual; for visualization parameters, use a 'V' as a prefix <b>EXAMPLE:</b> <code>mbs.SetNodeParameter(0, 'Vshow', True)</code>

#### 4.4.2 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)	<p>add an object with objectDefinition from Python object class; returns (global) object number (type ObjectIndex) of newly added object</p> <p><b>EXAMPLE:</b></p> <pre>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)</pre>
GetObjectNumber(objectName)	<p>get object's number by name (string)</p> <p><b>EXAMPLE:</b></p> <pre>n = mbs.GetObjectNumber('heavy object')</pre>
GetObject(objectNumber)	<p>get object's dictionary by object number (type ObjectIndex); NOTE: visualization parameters have a prefix 'V'</p> <p><b>EXAMPLE:</b></p> <pre>objectDict = mbs.GetObject(0)</pre>
ModifyObject(objectNumber, objectDict)	<p>modify object's dictionary by object number (type ObjectIndex); NOTE: visualization parameters have a prefix 'V'</p> <p><b>EXAMPLE:</b></p> <pre>mbs.ModifyObject(objectNumber, objectDict)</pre>
GetObjectDefaults(typeName)	<p>get object's default values for a certain objectType as (dictionary)</p> <p><b>EXAMPLE:</b></p> <pre>objectType = 'MassPoint' objectDict = mbs.GetObjectDefaults(objectType)</pre>
GetObjectOutput(objectNumber, variableType)	<p>get object's current output variable from object number (type ObjectIndex) and OutputVariableType; can only be computed for exu.ConfigurationType.Current configuration!</p>
GetObjectOutputBody(objectNumber, variableType, localPosition, configuration = ConfigurationType.Current)	<p>get body's output variable from object number (type ObjectIndex) and OutputVariableType, using the localPosition as defined in the body, and as used in MarkerBody and SensorBody</p> <p><b>EXAMPLE:</b></p> <pre>u = mbs.GetObjectOutputBody(objectNumber = 1, variableType = exu.OutputVariableType.Position, localPosition=[1,0,0], configuration = exu.ConfigurationType.Initial)</pre>

GetObjectOutputSuperElement(objectNumber, variableType, meshNodeNumber, configuration = ConfigurationType.Current)	get output variable from mesh node number of object with type SuperElement (GenericODE2, FFRE, FFRReduced - CMS) with specific OutputVariableType; the meshNodeNumber is the object's local node number, not the global node number! <b>EXAMPLE:</b> <code>u = mbs.GetObjectOutputSuperElement(objectNumber = 1, variableType = exu.OutputVariableType.Position, meshNodeNumber = 12, configuration = exu.ConfigurationType.Initial)</code>
GetObjectParameter(objectNumber, parameterName)	get objects's parameter from object number (type ObjectIndex) and parameterName; parameter names can be found for the specific items in the reference manual; for visualization parameters, use a 'V' as a prefix <b>EXAMPLE:</b> <code>mbs.GetObjectParameter(objectNumber = 0, parameterName = 'nodeNumber')</code>
SetObjectParameter(objectNumber, parameterName, value)	set parameter 'parameterName' of object with object number (type ObjectIndex) to value;; parameter names can be found for the specific items in the reference manual; for visualization parameters, use a 'V' as a prefix <b>EXAMPLE:</b> <code>mbs.SetObjectParameter(objectNumber = 0, parameterName = 'Vshow', value=True)</code>

#### 4.4.3 MainSystem: Marker

This section provides functions for adding, reading and modifying markers. Markers define how to measure primal kinematical quantities on objects or nodes (e.g., position, orientation or coordinates themselves), and how to act on the quantities which are dual to the kinematical quantities (e.g., force, torque and generalized forces). Markers provide unique interfaces for loads, sensors and constraints in order to address these quantities independently of the structure of the object or node (e.g., rigid or flexible body).

function/structure name	description
AddMarker(pyObject)	add a marker with markerDefinition from Python marker class; returns (global) marker number (type MarkerIndex) of newly added marker <b>EXAMPLE:</b> <code>item = MarkerNodePosition(name='my marker', nodeNumber=1) mbs.AddMarker(item) markerDict = {'markerType': 'NodePosition', 'nodeNumber': 0, 'name': 'position0'} mbs.AddMarker(markerDict)</code>

GetMarkerNumber(markerName)	get marker's number by name (string) <b>EXAMPLE:</b> <code>n = mbs.GetMarkerNumber('my marker')</code>
GetMarker(markerNumber)	get marker's dictionary by index <b>EXAMPLE:</b> <code>markerDict = mbs.GetMarker(0)</code>
ModifyMarker(markerNumber, markerDict)	modify marker's dictionary by index <b>EXAMPLE:</b> <code>mbs.ModifyMarker(markerNumber, markerDict)</code>
GetMarkerDefaults(typeName)	get marker's default values for a certain markerType as (dictionary) <b>EXAMPLE:</b> <code>markerType = 'NodePosition' markerDict = mbs.GetMarkerDefaults(markerType)</code>
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

#### 4.4.4 MainSystem: Load

This section provides functions for adding, reading and modifying operating loads. Loads are used to act on the quantities which are dual to the primal kinematic quantities, such as displacement and rotation. Loads represent, e.g., forces, torques or generalized forces.

function/structure name	description
AddLoad(pyObject)	add a load with loadDefinition from Python load class; returns (global) load number (type LoadIndex) of newly added load <b>EXAMPLE:</b> <code>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)</code>
GetLoadNumber(loadName)	get load's number by name (string) <b>EXAMPLE:</b> <code>n = mbs.GetLoadNumber('heavy load')</code>
GetLoad(loadNumber)	get load's dictionary by index <b>EXAMPLE:</b> <code>loadDict = mbs.GetLoad(0)</code>

ModifyLoad(loadNumber, loadDict)	modify load's dictionary by index <b>EXAMPLE:</b> <code>mbs.ModifyLoad(loadNumber, loadDict)</code>
GetLoadDefaults(typeName)	get load's default values for a certain loadType as (dictionary) <b>EXAMPLE:</b> <code>loadType = 'ForceVector'</code> <code>loadDict = mbs.GetLoadDefaults(loadType)</code>
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

#### 4.4.5 MainSystem: Sensor

This section provides functions for adding, reading and modifying operating sensors. Sensors are used to measure information in nodes, objects, markers, and loads for output in a file.

function/structure name	description
AddSensor(pyObject)	add a sensor with sensor definition from Python sensor class; returns (global) sensor number (type SensorIndex) of newly added sensor <b>EXAMPLE:</b> <code>item = mbs.AddSensor(SensorNode(sensorType=exu.SensorType.Node, nodeNumber=0, name='test sensor'))</code> <code>mbs.AddSensor(item)</code> <code>sensorDict = {'sensorType': 'Node', 'nodeNumber': 0, 'fileName': 'sensor.txt', 'name': 'test sensor'}</code> <code>mbs.AddSensor(sensorDict)</code>
GetSensorNumber(sensorName)	get sensor's number by name (string) <b>EXAMPLE:</b> <code>n = mbs.GetSensorNumber('test sensor')</code>
GetSensor(sensorNumber)	get sensor's dictionary by index <b>EXAMPLE:</b> <code>sensorDict = mbs.GetSensor(0)</code>
ModifySensor(sensorNumber, sensorDict)	modify sensor's dictionary by index <b>EXAMPLE:</b> <code>mbs.ModifySensor(sensorNumber, sensorDict)</code>

GetSensorDefaults(typeName)	get sensor's default values for a certain sensorType as (dictionary) <b>EXAMPLE:</b> <code>sensorType = 'Node'</code> <code>sensorDict = mbs.GetSensorDefaults(sensorType)</code>
GetSensorValues(sensorNumber, configuration = ConfigurationType.Current)	get sensors's values for configuration; can be a scalar or vector-valued return value!
GetSensorStoredData(sensorNumber)	get sensors's internally stored data as matrix (all time points stored); rows are containing time and sensor values as obtained by sensor (e.g., time, and x, y, and z value of position)
GetSensorParameter(sensorNumber, parameterName)	get sensors's parameter from sensorNumber and parameterName; parameter names can be found for the specific items in the reference manual
SetSensorParameter(sensorNumber, parameterName, value)	set parameter 'parameterName' of sensor with sensorNumber to value; parameter names can be found for the specific items in the reference manual

## 4.5 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 <b>EXAMPLE:</b> <code>print(mbs.systemData.NumberOfLoads())</code>
NumberOfMarkers()	return number of markers in system <b>EXAMPLE:</b> <code>print(mbs.systemData.NumberOfMarkers())</code>
NumberOfNodes()	return number of nodes in system <b>EXAMPLE:</b> <code>print(mbs.systemData.NumberOfNodes())</code>
NumberOfObjects()	return number of objects in system <b>EXAMPLE:</b> <code>print(mbs.systemData.NumberOfObjects())</code>
NumberOfSensors()	return number of sensors in system <b>EXAMPLE:</b> <code>print(mbs.systemData.NumberOfSensors())</code>

ODE2Size(configurationType exu.ConfigurationType.Current)	=	get size of ODE2 coordinate vector for given configuration (only works correctly after mbs.Assemble() ) <b>EXAMPLE:</b> <code>print('ODE2 size=',mbs.systemData.ODE2Size())</code>
ODE1Size(configurationType exu.ConfigurationType.Current)	=	get size of ODE1 coordinate vector for given configuration (only works correctly after mbs.Assemble() ) <b>EXAMPLE:</b> <code>print('ODE1 size=',mbs.systemData.ODE1Size())</code>
AESize(configurationType exu.ConfigurationType.Current)	=	get size of AE coordinate vector for given configuration (only works correctly after mbs.Assemble() ) <b>EXAMPLE:</b> <code>print('AE size=',mbs.systemData.AESize())</code>
DataSize(configurationType exu.ConfigurationType.Current)	=	get size of Data coordinate vector for given configuration (only works correctly after mbs.Assemble() ) <b>EXAMPLE:</b> <code>print('Data size=',mbs.systemData.DataSize())</code>
SystemSize(configurationType exu.ConfigurationType.Current)	=	get size of System coordinate vector for given configuration (only works correctly after mbs.Assemble() ) <b>EXAMPLE:</b> <code>print('System size=',mbs.systemData.SystemSize())</code>
GetTime(configurationType exu.ConfigurationType.Current)	=	get configuration dependent time. <b>EXAMPLE:</b> <code>mbs.systemData.GetTime(exu.ConfigurationType.Initial)</code>
SetTime(newTime, configurationType exu.ConfigurationType.Current)	=	set configuration dependent time; use this access with care, e.g. in user-defined solvers. <b>EXAMPLE:</b> <code>mbs.systemData.SetTime(10., exu.ConfigurationType.Initial)</code>
Info()		print detailed system information for every item; for short information use print(mbs) <b>EXAMPLE:</b> <code>mbs.systemData.Info()</code>

#### 4.5.1 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) <b>EXAMPLE:</b> <code>uCurrent = mbs.systemData.GetODE2Coordinates()</code>
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! <b>EXAMPLE:</b> <code>mbs.systemData.SetODE2Coordinates(uCurrent)</code>

GetODE2Coordinates_t(configuration exu.ConfigurationType.Current)	=	get ODE2 system coordinates (velocities) for given configuration (default: exu.Configuration.Current)  <b>EXAMPLE:</b> <code>vCurrent = mbs.systemData.GetODE2Coordinates_t()</code>
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!  <b>EXAMPLE:</b> <code>mbs.systemData.SetODE2Coordinates_t(vCurrent)</code>
GetODE2Coordinates_tt(configuration exu.ConfigurationType.Current)	=	get ODE2 system coordinates (accelerations) for given configuration (default: exu.Configuration.Current)  <b>EXAMPLE:</b> <code>vCurrent = mbs.systemData.GetODE2Coordinates_tt()</code>
SetODE2Coordinates_tt(coordinates, configuration exu.ConfigurationType.Current)	=	set ODE2 system coordinates (accelerations) for given configuration (default: exu.Configuration.Current); invalid vector size may lead to system crash!  <b>EXAMPLE:</b> <code>mbs.systemData.SetODE2Coordinates_tt(aCurrent)</code>
GetODE1Coordinates(configuration exu.ConfigurationType.Current)	=	get ODE1 system coordinates (displacements) for given configuration (default: exu.Configuration.Current)  <b>EXAMPLE:</b> <code>qCurrent = mbs.systemData.GetODE1Coordinates()</code>
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!  <b>EXAMPLE:</b> <code>mbs.systemData.SetODE1Coordinates(qCurrent)</code>
GetAECoordinates(configuration exu.ConfigurationType.Current)	=	get algebraic equations (AE) system coordinates for given configuration (default: exu.Configuration.Current)  <b>EXAMPLE:</b> <code>lambdaCurrent = mbs.systemData.GetAECoordinates()</code>
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!  <b>EXAMPLE:</b> <code>mbs.systemData.SetAECoordinates(lambdaCurrent)</code>
GetDataCoordinates(configuration exu.ConfigurationType.Current)	=	get system data coordinates for given configuration (default: exu.Configuration.Current)  <b>EXAMPLE:</b> <code>dataCurrent = mbs.systemData.GetDataCoordinates()</code>
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!  <b>EXAMPLE:</b> <code>mbs.systemData.SetDataCoordinates(dataCurrent)</code>

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] <b>EXAMPLE:</b> <code>sysStateList = mbs.systemData.GetSystemState()</code>
SetSystemState(systemStateList, exu.ConfigurationType.Current)	configuration	= 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] <b>EXAMPLE:</b> <code>mbs.systemData.SetSystemState(sysStateList, configuration = exu.ConfigurationType.Initial)</code>

#### 4.5.2 SystemData: Get object LTG coordinate mappings

This section provides access functions the **LTG**-lists for every object (body, constraint, ...) in the system. For details on the **LTG** mapping, see [Section 2.3](#)

function/structure name	description
GetObjectLTGODE2(objectNumber)	get local-to-global coordinate mapping (list of global coordinate indices) for ODE2 coordinates; only available after Assemble() <b>EXAMPLE:</b> <code>ltgObject4 = mbs.systemData.GetObjectLTGODE2(4)</code>
GetObjectLTGODE1(objectNumber)	get local-to-global coordinate mapping (list of global coordinate indices) for ODE1 coordinates; only available after Assemble() <b>EXAMPLE:</b> <code>ltgObject4 = mbs.systemData.GetObjectLTGODE1(4)</code>
GetObjectLTGAE(objectNumber)	get local-to-global coordinate mapping (list of global coordinate indices) for algebraic equations (AE) coordinates; only available after Assemble() <b>EXAMPLE:</b> <code>ltgObject4 = mbs.systemData.GetObjectLTGODE2(4)</code>
GetObjectLTGData(objectNumber)	get local-to-global coordinate mapping (list of global coordinate indices) for data coordinates; only available after Assemble() <b>EXAMPLE:</b> <code>ltgObject4 = mbs.systemData.GetObjectLTGData(4)</code>

## 4.6 MatrixContainer

The MatrixContainer is a versatile representation for dense and sparse matrices.

Usage:

- Create empty MatrixContainer with `mc = MatrixContainer()`
- Create MatrixContainer with dense matrix `mc = MatrixContainer(matrix)`, where matrix can be a list of lists of a numpy array
- Set with dense pyArray (a numpy array): `mc.SetWithDenseMatrix(pyArray, bool useDenseMatrix = True)`
- Set with sparse pyArray (a numpy array), which has 3 columns and according rows containing the sparse triplets (`row, col, value`) describing the sparse matrix

function/structure name	description
<code>SetWithDenseMatrix(pyArray, useDenseMatrix = false)</code>	set MatrixContainer with dense numpy array; array (=matrix) contains values and matrix size information; if <code>useDenseMatrix=True</code> , matrix will be stored internally as dense matrix, otherwise it will be converted and stored as sparse matrix (which may speed up computations for larger problems)
<code>SetWithSparseMatrixCSR(numberOfRowsInit, numberOfRowsColumnsInit, pyArrayCSR, useDenseMatrix = true)</code>	set with sparse CSR matrix format: numpy array 'pyArrayCSR' contains sparse triplet (row, col, value) per row; <code>numberOfRows</code> and <code>numberOfColumns</code> given extra; if <code>useDenseMatrix=True</code> , matrix will be converted and stored internally as dense matrix, otherwise it will be stored as sparse matrix
<code>GetPyObject()</code>	convert MatrixContainer to numpy array (dense) or dictionary (sparse): containing nr. of rows, nr. of columns, numpy matrix with sparse triplets
<code>Convert2DenseMatrix()</code>	convert MatrixContainer to dense numpy array (SLOW and may fail for too large sparse matrices)
<code>UseDenseMatrix()</code>	returns True if dense matrix is used, otherwise False

## 4.7 GeneralContact

Structure to define general and highly efficient contact functionality in multibody systems<sup>1</sup>. For further explanations and theoretical backgrounds, see [Section 6.3](#).

Usage:

- Add GeneralContact to mbs `gContact = mbs.AddGeneralContact()`
- Add contact elements, e.g., `gContact.AddSphereWithMarker(...)`, using appropriate arguments
- Call `SetFrictionPairings(...)` to set friction pairings and adjust searchTree if needed.

---

<sup>1</sup>Note that GeneralContact is still developed, use with care.

function/structure name	description
GetPyObject()	convert member variables of GeneralContact into dictionary; use this for debug only!
Reset(freeMemory = true)	remove all contact objects and reset contact parameters
isActive	default = True (compute contact); if isActive=False, no contact computation is performed for this contact set
verboseMode	default = 0; verboseMode = 1 or higher outputs useful information on the contact creation and computation
visualization	access visualization data structure
sphereSphereContact	activate/deactivate contact between spheres
sphereSphereFrictionRecycle	False: compute static friction force based on tangential velocity; True: recycle friction from previous PostNewton step, which greatly improves convergence, but may lead to unphysical artifacts; will be solved in future by step reduction
minRelDistanceSpheresTriangles	(default=1e-10) tolerance (relative to sphere radii) below which the contact between triangles and spheres is ignored; used for spheres directly attached to triangles
frictionProportionalZone	(default=0.001) velocity $v_{\mu,\text{reg}}$ upon which the dry friction coefficient is interpolated linearly (regularized friction model); must be greater 0; very small values cause oscillations in friction force
frictionVelocityPenalty	(default=1e3) regularization factor for friction [N/(m <sup>2</sup> ·m/s)] $k_{\mu,\text{reg}}$ , multiplied with tangential velocity to compute friction force as long as it is smaller than $\mu$ times contact force; large values cause oscillations in friction force
excludeOverlappingTrigSphereContacts	(default=True) for consistent, closed meshes, we can exclude overlapping contact triangles (which would cause holes if mesh is overlapping and not consistent!!!)
excludeDuplicatedTrigSphereContactPoints	(default=False) run additional checks for double contacts at edges or vertices, being more accurate but can cause additional costs if many contacts
ancfCableUseExactMethod	(default=True) if true, uses exact computation of intersection of 3rd order polynomials and contacting circles
ancfCableNumberOfContactSegments	(default=1) number of segments to be used in case that ancfCableUseExactMethod=False; maximum number of segments=3
SetFrictionPairings(frictionPairings)	set Coulomb friction coefficients for pairings of materials (e.g., use material 0,1, then the entries (0,1) and (1,0) define the friction coefficients for this pairing); matrix should be symmetric! <b>EXAMPLE:</b> <code>#set 3 surface friction types, all being 0.1: gContact.SetFrictionPairings(0.1*np.ones((3,3)))</code>
SetFrictionProportionalZone(frictionProportionalZone)	regularization for friction (m/s); used for all contacts
SetSearchTreeCellSize(numberOfCells)	set number of cells of search tree (boxed search) in x, y and z direction <b>EXAMPLE:</b> <code>gContact.SetSearchTreeInitSize([10, 10, 10])</code>

SetSearchTreeBox(pMin, pMax)	set geometric dimensions of searchTreeBox (point with minimum coordinates and point with maximum coordinates); if this box becomes smaller than the effective contact objects, contact computations may slow down significantly <b>EXAMPLE:</b> <code>gContact.SetSearchTreeBox(pMin=[-1,-1,-1], Max=[1,1,1])</code>
AddSphereWithMarker(markerIndex, radius, contactStiffness, contactDamping, frictionMaterialIndex)	add contact object using a marker (Position or Rigid), radius and contact/friction parameters; frictionMaterialIndex refers to frictionPairings in GeneralContact; contact is possible between spheres (circles in 2D) (if intraSphereContact = True), spheres and triangles and between sphere (=circle) and ANCF Cable2D; contactStiffness is computed as serial spring between contacting objects, while damping is computed as a parallel damper (otherwise the smaller damper would always dominate)!
AddANCFCable(objectIndex, halfHeight, contactStiffness, contactDamping, frictionMaterialIndex)	add contact object for an ANCF cable element, using the objectIndex of the cable element and the cable's half height as an additional distance to contacting objects (currently not causing additional torque in case of friction); currently only contact with spheres (circles in 2D) possible; contact computed using exact geometry of elements, finding max 3 intersecting contact regions
AddTrianglesRigidBodyBased(rigidBodyMarkerIndex, contactStiffness, contactDamping, frictionMaterialIndex, pointList, triangleList)	add contact object using a rigidBodyMarker (of a body), contact/friction parameters, a list of points (as 3D numpy arrays or lists; coordinates relative to rigidBodyMarker) and a list of triangles (3 indices as numpy array or list) according to a mesh attached to the rigidBodyMarker; mesh can be produced with GraphicsData2TrigsAndPoints(...); contact is possible between sphere (circle) and Triangle but yet not between triangle and triangle; frictionMaterialIndex refers to frictionPairings in GeneralContact; contactStiffness is computed as serial spring between contacting objects, while damping is computed as a parallel damper (otherwise the smaller damper would always dominate); the triangle normal must point outwards, with the normal of a triangle given with local points (p0,p1,p2) defined as n=(p1-p0) x (p2-p0), see function ComputeTriangleNormal(...)
<code>__repr__()</code>	return the string representation of the GeneralContact, containing basic information and statistics

## 4.8 VisuGeneralContact

Data structure for visualization inside GeneralContact.

Usage:

- `gContact.visualization.drawSpheres = True`

function/structure name	description
<code>Reset()</code>	reset visualization parameters to default values

## 4.9 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)
```

### 4.9.1 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 interpretation of the output variable can be found at the object definitions. The `OutputVariableType` does not provide information about the size of the output variable, which can be either scalar or a list (vector). For vector output quantities, the contour plot option offers an additional parameter for selection of the component of the `OutputVariableType`. The components are usually out of {0,1,2}, representing {x,y,z} components (e.g., of displacements, velocities, ...), or {0,1,2,3,4,5} representing {xx,yy,zz,yz,xz,xy} components (e.g., of strain or stress). In order to compute a norm, chose component=-1, which will result in the quadratic norm for other vectors and to a norm specified for stresses (if no norm is defined for an `outputVariable`, it does not compute anything)

function/structure name	description
<code>_None</code>	no value; used, e.g., to select no output variable in contour plot
<code>Distance</code>	e.g., measure distance in spring damper connector
<code>Position</code>	measure 3D position, e.g., of node or body
<code>Displacement</code>	measure displacement; usually difference between current position and reference position
<code>DisplacementLocal</code>	measure local displacement, e.g. in local joint coordinates
<code>Velocity</code>	measure (translational) velocity of node or object
<code>VelocityLocal</code>	measure local (translational) velocity, e.g. in local body or joint coordinates
<code>Acceleration</code>	measure (translational) acceleration of node or object
<code>RotationMatrix</code>	measure rotation matrix of rigid body node or object
<code>Rotation</code>	measure, e.g., scalar rotation of 2D body, Euler angles of a 3D object or rotation within a joint
<code>AngularVelocity</code>	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
Coordinates	measure the coordinates of a node or object; coordinates usually just contain displacements, but not the position values
Coordinates_t	measure the time derivative of coordinates (= velocity coordinates) of a node or object
Coordinates_tt	measure the second time derivative of coordinates (= acceleration coordinates) of a node or object
SlidingCoordinate	measure sliding coordinate in sliding joint
Director1	measure a director (e.g. of a rigid body frame), or a slope vector in local 1 or x-direction
Director2	measure a director (e.g. of a rigid body frame), or a slope vector in local 2 or y-direction
Director3	measure a director (e.g. of a rigid body frame), or a slope vector in local 3 or z-direction
Force	measure global force, e.g., in joint or beam (resultant force), or generalized forces; see description of according object
ForceLocal	measure local force, e.g., in joint or beam (resultant force)
Torque	measure torque, e.g., in joint or beam (resultant couple/moment)
TorqueLocal	measure local torque, e.g., in joint or beam (resultant couple/moment)
StrainLocal	measure local strain, e.g., axial strain in cross section frame of beam or Green-Lagrange strain
StressLocal	measure local stress, e.g., axial stress in cross section frame of beam or Second Piola-Kirchoff stress; choosing component==1 will result in the computation of the Mises stress
CurvatureLocal	measure local curvature; may be scalar or vectorial: twist and curvature of beam in cross section frame
ConstraintEquation	evaluates constraint equation (=current deviation or drift of constraint equation)

#### 4.9.2 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

#### 4.9.3 ItemType

This section shows the ItemType structure, which is used for defining types of indices, e.g., in render window and will be also used in item dictionaries in future.

function/structure name	description
_None	item has no type
Node	item or index is of type Node
Object	item or index is of type Object
Marker	item or index is of type Marker
Load	item or index is of type Load
Sensor	item or index is of type Sensor

#### 4.9.4 NodeType

This section shows the NodeType structure, which is used for defining node types for 3D rigid bodies.

function/structure name	description
_None	node has no type
Ground	ground node
Position2D	2D position node
Orientation2D	node with 2D rotation
Point2DSlope1	2D node with 1 slope vector
Position	3D position node
Orientation	3D orientation node
RigidBody	node that can be used for rigid bodies
RotationEulerParameters	node with 3D orientations that are modelled with Euler parameters (unit quaternions)

RotationRxyz	node with 3D orientations that are modelled with Tait-Bryan angles
RotationRotationVector	node with 3D orientations that are modelled with the rotation vector
RotationLieGroup	node intended to be solved with Lie group methods
GenericODE2	node with general ODE2 variables
GenericODE1	node with general ODE1 variables
GenericAE	node with general algebraic variables
GenericData	node with general data variables

#### 4.9.5 DynamicSolverType

This section shows the DynamicSolverType structure, which is used for selecting dynamic solvers for simulation.

function/structure name	description
GeneralizedAlpha	an implicit solver for index 3 problems; intended to be used for solving directly the index 3 constraints using the spectralRadius sufficiently small (usually 0.5 .. 1)
TrapezoidalIndex2	an implicit solver for index 3 problems with index2 reduction; uses generalized alpha solver with settings for Newmark with index2 reduction
ExplicitEuler	an explicit 1st order solver (generally not compatible with constraints)
ExplicitMidpoint	an explicit 2nd order solver (generally not compatible with constraints)
RK33	an explicit 3 stage 3rd order Runge-Kutta method, aka "Heun"; (generally not compatible with constraints)
RK44	an explicit 4 stage 4th order Runge-Kutta method, aka "classical Runge Kutta" (generally not compatible with constraints), compatible with Lie group integration and elimination of CoordinateConstraints
RK67	an explicit 7 stage 6th order Runge-Kutta method, see 'On Runge-Kutta Processes of High Order', J. C. Butcher, J. Austr Math Soc 4, (1964); can be used for very accurate (reference) solutions, but without step size control!
ODE23	an explicit Runge Kutta method with automatic step size selection with 3rd order of accuracy and 2nd order error estimation, see Bogacki and Shampine, 1989; also known as ODE23 in MATLAB
DOPRI5	an explicit Runge Kutta method with automatic step size selection with 5th order of accuracy and 4th order error estimation, see Dormand and Prince, 'A Family of Embedded Runge-Kutta Formulae.', J. Comp. Appl. Math. 6, 1980
DVERK6	[NOT IMPLEMENTED YET] an explicit Runge Kutta solver of 6th order with 5th order error estimation; includes adaptive step selection

#### 4.9.6 KeyCode

This section shows the KeyCode structure, which is used for special key codes in keyPressUserFunction.

function/structure name	description
SPACE	space key
ENTER	enter (return) key
TAB	
BACKSPACE	
RIGHT	cursor right
LEFT	cursor left
DOWN	cursor down
UP	cursor up
F1	function key F1
F2	function key F2
F3	function key F3
F4	function key F4
F5	function key F5
F6	function key F6
F7	function key F7
F8	function key F8
F9	function key F9
F10	function key F10

#### 4.9.7 LinearSolverType

This section shows the LinearSolverType structure, which is used for selecting linear solver types, which are dense or sparse solvers.

function/structure name	description
_None	no value; used, e.g., if no solver is selected
EXUdense	use dense matrices and according solvers for densely populated matrices (usually the CPU time grows cubically with the number of unknowns)
EigenSparse	use sparse matrices and according solvers; additional overhead for very small systems; specifically, memory allocation is performed during a factorization process



# Chapter 5

## Python utility functions

This chapter describes in every subsection the functions and classes of the utility modules. These modules help to create multibody systems with the EXUDYN core module. Functions are implemented in Python and can be easily changed, extended and also verified by the user. **Check the source code** by entering these functions in Sypder and pressing **CTRL + left mouse button**. These Python functions are much slower than the functions available in the C++ core. Some matrix computations with larger matrices implemented in numpy and scipy, however, are parallelized and therefore very efficient.

Note that in usually functions accept lists and numpy arrays. If not, an error will occur, which is easily tracked. Furthermore, angles are generally provided in radian ( $2\pi$  equals  $360^\circ$ ) and no units are used for distances, but it is recommended to use SI units (m, kg, s) throughout.

Functions have been implemented, if not otherwise mentioned, by Johannes Gerstmayr.

### 5.1 Utility: ResultsMonitor

The results monitor is a special tool, which allows to monitor results during simulation. This is intended, e.g., to show results for long-term simulations or to visualize results for teaching. The tool can visualize time dependent data (e.g., from sensors or solution files) or data from optimization. The tool automatically detects the type of file and visualizes all given columns (default) or selected columns of the file.

For running the results monitor, start a terminal (linux) or an Anaconda prompt (windows). Either you copy the `resultsLoader.py`, located in the `main/pythonDev/exudyn` subfolder of the repository, to your desired/current directory or you call it from a relative path from your current directory. Usage is described by typing `python resultsMonitor.py -h`, as given in the following listing:

---

```
usage for resultsLoader:  
  python resultsLoader.py file.txt  
options:  
  -xcols i,j,...: comma-separated columns (NO SPACES!) to be plotted on x-axis  
  -ycols i,j,...: comma-separated columns (NO SPACES!) to be plotted on y-axis  
  -logx: use log scale for x-axis  
  -logy: use log scale for y-axis  
  -sizex float: float = x-size of one subplot in inches (default=5)  
  -sizey float: float = y-size of one subplot in inches (default=5)  
  -update float: float = update period in seconds  
  -color char: char = line color code according to pyplot, default=b (blue)  
  -style char: char = line symbol according to pyplot, default="-"
```

```
example: (to be called from windows Anaconda prompt or in linux terminal in the directory  
where file.txt lies)  
python resultsMonitor.py file.txt -logy -xcols 0,1 -ycols 2,3 -update 0.2
```

---

## 5.2 Module: basicUtilities

Basic utility functions and constants, not depending on numpy or other python modules.

Author: Johannes Gerstmayr

Date: 2020-03-10 (created)

Notes: Additional constants are defined:

pi = 3.1415926535897932

sqrt2 = 2\*\*0.5

g=9.81

eye2D (2x2 diagonal matrix)

eye3D (3x3 diagonal matrix)

Two variables 'gaussIntegrationPoints' and 'gaussIntegrationWeights' define integration points and weights for function GaussIntegrate(...)

```
def ClearWorkspace ()
```

– **function description:**

clear all workspace variables except for system variables with '\_' at beginning,  
'func' or 'module' in name

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

– **example:**

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

---

```
def DiagonalMatrix (rowsColumns, value= 1)
```

– **function description:** create a diagonal or identity matrix; used for interface.py, avoiding the need for numpy

– **input:**

*rowsColumns*: provides the number of rows and columns

*value*: initialization value for diagonal terms

- **output**: list of lists representing a matrix
- 

```
def NormL2 (vector)
```

- **function description**: compute L2 norm for vectors without switching to numpy or math module
- **input**: vector as list or in numpy format
- **output**: L2-norm of vector

For examples on NormL2 see Examples (Ex) and TestModels (TM):

- [bicycleIftommBenchmark.py](#) (Ex), [NGsolvePistonEngine.py](#) (Ex), [springsDeactivateConnectors.py](#) (Ex),  
[explicitLieGroupIntegratorTest.py](#) (TM), [fourBarMechanismRedundant.py](#) (TM), [genericODE2test.py](#) (TM), [NGsolveCrankShaftTest.py](#) (TM), [pendulumFriction.py](#) (TM), ...
- 

```
def VSum (vector)
```

- **function description**: compute sum of all values of vector
- **input**: vector as list or in numpy format
- **output**: sum of all components of vector

For examples on VSum see Examples (Ex) and TestModels (TM):

- [serialRobotOldTests.py](#) (Ex), [serialRobotTestDH2.py](#) (Ex), [serialRobotTSD.py](#) (Ex),  
[serialRobotTest.py](#) (TM)
- 

```
def VAdd (v0, v1)
```

- **function description**: add two vectors instead using numpy
- **input**: vectors v0 and v1 as list or in numpy format
- **output**: component-wise sum of v0 and v1

For examples on VAdd see Examples (Ex) and TestModels (TM):

- [NGsolvePistonEngine.py](#) (Ex), [carRollingDiscTest.py](#) (TM), [mecanumWheelRollingDiscTest.py](#) (TM),  
[NGsolveCrankShaftTest.py](#) (TM), [rigidBodyCOMtest.py](#) (TM), [sliderCrank3Dbenchmark.py](#) (TM)
-

```
def VSub (v0, v1)
```

- **function description:** subtract two vectors instead using numpy: result = v0-v1
- **input:** vectors v0 and v1 as list or in numpy format
- **output:** component-wise difference of v0 and v1

For examples on VSub see Examples (Ex) and TestModels (TM):

- [NGsolveCMSTutorial.py](#) (Ex), [NGsolvePistonEngine.py](#) (Ex), [ObjectFFRFconvergenceTestHinge.py](#) (Ex),  
[NGsolveCrankShaftTest.py](#) (TM), [rigidBodyCOMtest.py](#) (TM)
- 

```
def VMult (v0, v1)
```

- **function description:** scalar multiplication of two vectors instead using numpy: result = v0'\*v1
  - **input:** vectors v0 and v1 as list or in numpy format
  - **output:** sum of all component wise products: c0[0]\*v1[0] + v0[1]\*v1[0] + ...
- 

```
def ScalarMult (scalar, v)
```

- **function description:** multiplication vectors with scalar: result = s\*v
- **input:** value *scalar* and vector *v* as list or in numpy format
- **output:** scalar multiplication of all components of *v*: [scalar\*v[0], scalar\*v[1], ...]

For examples on ScalarMult see Examples (Ex) and TestModels (TM):

- [pendulumFriction.py](#) (TM), [sliderCrank3Dbenchmark.py](#) (TM)
- 

```
def Normalize (v)
```

- **function description:** take a 3D vector and return a normalized 3D vector (L2Norm=1)
- **input:** vector *v* as list or in numpy format
- **output:** vector *v* multiplied with scalar such that L2-norm of vector is 1

For examples on Normalize see Examples (Ex) and TestModels (TM):

- [NGsolveCMSTutorial.py](#) (Ex), [NGsolvePistonEngine.py](#) (Ex), [ObjectFFRFconvergenceTestHinge.py](#) (Ex),  
[NGsolveCrankShaftTest.py](#) (TM)
-

```
def Vec2Tilde (v)
```

- **function description:** apply tilde operator (skew) to 3D-vector and return skew matrix
- **input:** 3D vector v as list or in numpy format

– **output:** matrix as list of lists containing the skew-symmetric matrix computed from v: 
$$\begin{bmatrix} 0 & -v[2] & v[1] \\ v[2] & 0 & -v[0] \\ -v[1] & v[0] & 0 \end{bmatrix}$$

For examples on Vec2Tilde see Examples (Ex) and TestModels (TM):

- [explicitLieGroupMBSTest.py](#) (TM)
- 

```
def Tilde2Vec (m)
```

- **function description:** take skew symmetric matrix and return vector (inverse of Skew(...))
  - **input:** list of lists containing a skew-symmetric matrix (3x3)
  - **output:** list containing the vector v (inverse function of Vec2Tilde(...))
- 

```
def GaussIntegrate (functionOfX, integrationOrder, a, b)
```

- **function description:** compute numerical integration of functionOfX in interval [a,b] using Gaussian integration
  - **input:**
    - functionOfX*: scalar, vector or matrix-valued function with scalar argument (X or other variable)
    - integrationOrder*: odd number in {1,3,5,7,9}; currently maximum order is 9
    - a*: integration range start
    - b*: integration range end
  - **output:** (scalar or vectorized) integral value
- 

```
def LobattoIntegrate (functionOfX, integrationOrder, a, b)
```

- **function description:** compute numerical integration of functionOfX in interval [a,b] using Lobatto integration
- **input:**
  - functionOfX*: scalar, vector or matrix-valued function with scalar argument (X or other variable)
  - integrationOrder*: even number in {2,4,6}; currently maximum order is 6
  - a*: integration range start
  - b*: integration range end
- **output:** (scalar or vectorized) integral value

## 5.3 Module: beams

Beam utility functions, e.g. for creation of sequences of straight or curved beams.

Author: Johannes Gerstmayr

Date: 2022-01-30 (created)

Notes: For a list of plot colors useful for matplotlib, see also utilities.PlotLineCode(...)

```
def GenerateStraightLineANCF Cable2D (mbs, positionOfNode0, positionOfNode1, numberOfWorkElements,  
cableTemplate, massProportionalLoad= [0,0,0], fixedConstraintsNode0= [0,0,0,0], fixedConstraintsNode1= [0,0,0,0],  
nodeNumber0= -1, nodeNumber1= -1)
```

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

– **input:**

*mbs*: the system where ANCF cables are added

*positionOfNode0*: 3D position (list or np.array) for starting point of line

*positionOfNode1*: 3D position (list or np.array) for end point of line

*numberOfWorkElements*: for discretization of line

*cableTemplate*: a ObjectANCF Cable2D object, containing the desired cable properties; cable length and node numbers are set automatically

*massProportionalLoad*: a 3D list or np.array, containing the gravity vector or zero

*fixedConstraintsNode0*: a list of 4 binary values, indicating the coordinate constraints on the first node (x,y-position and x,y-slope)

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

*nodeNumber0*: if set other than -1, this node number defines the node that shall be used at positionOfNode0

*nodeNumber1*: if set other than -1, this node number defines the node that shall be used at positionOfNode1

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

– **example:**

see Examples/ANCF\_cantilever\_test.py

For examples on GenerateStraightLineANCF Cable2D see Examples (Ex) and TestModels (TM):

- [ANCF ALE test .py](#) (Ex), [ANCF\\_cantilever\\_test .py](#) (Ex), [beltDriveReevingSystem .py](#) (Ex), [finiteSegmentMethod .py](#) (Ex), [reevingSystem .py](#) (Ex), ... , [ANCFslidingAndALEjointTest .py](#) (TM), [ANCFbeltDrive .py](#) (TM), [ANCFgeneralContactCircle .py](#) (TM), ...

---

```
def GenerateCircularArcANCF Cable2D (mbs, positionOfNode0, radius, startAngle, arcAngle, numberOfWorkElements,  
cableTemplate, massProportionalLoad= [0,0,0], fixedConstraintsNode0= [0,0,0,0], fixedConstraintsNode1= [0,0,0,0],  
nodeNumber0= -1, nodeNumber1= -1, setCurvedReferenceConfiguration= True, verboseMode= False)
```

- **function description:** generate cable elements along circular arc with given start point, radius, start angle (measured relative to  $x$ -axis, in positive rotation sense) and angle of arc
- **input:**
  - mbs*: the system where ANCF cables are added
  - positionOfNode0*: 3D position (list or np.array) for starting point of line
  - radius*: radius of arc
  - startAngle*: start angle of arc in radians ( $0 \dots 2\pi$ ), defines the direction of the slope vector, measured relative to  $x$ -axis, in positive rotation sense
  - arcAngle*: total angle of arc in radians ( $0 \dots 2\pi$ ), measured in positive rotation sense (negative angle reverts curvature and center point of circle)
  - numberOfElements*: for discretization of arc
  - cableTemplate*: a ObjectANCFCable2D object, containing the desired cable properties; cable length and node numbers are set automatically
  - massProportionalLoad*: a 3D list or np.array, containing the gravity vector or zero
  - fixedConstraintsNode0*: a list of 4 binary values, indicating the coordinate constraints on the first node ( $x,y$ -position and  $x,y$ -slope)
  - fixedConstraintsNode1*: a list of 4 binary values, indicating the coordinate constraints on the last node ( $x,y$ -position and  $x,y$ -slope)
  - nodeNumber0*: if set other than -1, this node number defines the node that shall be used at *positionOfNode0*
  - nodeNumber1*: if set other than -1, this node number defines the node that shall be used at *positionOfNode1*
  - setCurvedReferenceConfiguration*: if True, the curvature  $+/- (1/radius)$  is set as a reference configuration (sign depends on *arcAngle*); if False, the reference configuration is straight
  - verboseMode*: if True, prints out information on created nodes
- **output:** returns a list [cableNodeList, cableObjectList, loadList, cableNodePositionList, cableCoordinateConstraintList]

For examples on GenerateCircularArcANCFCable2D see Examples (Ex) and TestModels (TM):

- [ANCFbeltDrive.py](#) (TM), [ANCFgeneralContactCircle.py](#) (TM)
- 

```
def CreateReevingCurve (circleList, drawingLinesPerCircle= 64, numberOfANCFnodes= -1, removeLastLine=False, removeFirstLine= False, radialOffset= 0., closedCurve= False, graphicsElementsPerCircle= 64, graphicsNodeSize= 0, colorCircles= [0.,0.5,1.,1.], colorLines= [1.,0.5,0.,1.])
```

- **function description:** CreateReevingCurve for creating the geometry of a reeving system based on circles with radius and left/right side of passing the circles; left/right is seen in the direction passing from one to the next circle
- **input:**
  - circleList*: list containing center position, radius and 'L' (left) or 'R' (right) passing of circle

- radialOffset*: additional offset added to circles to account for half height of rope or beam
- closedCurve*: if True, the system adds circleList[0] and circleList[1] at end of list and sets removeLastLine=True and removeFirstLine=False, in order to generate a closed curve according to given circles; furthermore, the number of nodes becomes equal to the number of elements in this case
- drawingLinesPerCircle*: number of lines in lineData per one revolution
- numberOfANCFnodes*: if not -1, function also generates nodes with equidistant distribution along curve!
- graphicsElementsPerCircle*: number of drawing lines generated in graphicsDataLines per circle revolution (larger generates better approximation of circles)
- graphicsNodeSize*: if not 0, addes graphics representation of nodes generated; for check if mesh is correct
- removeFirstLine*: removes first line generated, which may be unwanted
- removeLastLine*: removes last line generated, which may be unwanted
- colorCircles*: RGBA color for circles
- colorLines*: RGBA color for lines
- **output**: return a dictionary with 'ancfPointsSlopes':ancfPointsSlopes, 'elementLengths':elementLengths, 'elementCurvatures':elementCurvatures, 'totalLength':totalLength, 'circleData':circle2D, 'graphicsDataLines':graphicsDataLines, 'graphicsDataCircles':graphicsDataCircles ; 'ancfPointsSlopes' denotes 4-dimensional vector with (x/y) position and (x/y) slope coordinates in a row; 'elementLengths' is the list of curved lengths for elements between nodes (size is 1 smaller than number of nodes), 'elementCurvatures' is the list of scalar curvatures between nodes (according to list of elementLengths), 'totalLength' is the total length of the reeving line, 'circleData' represents the lines and arcs calculated for the reeving system, 'graphicsDataLines' is the graphicsData for the lines and 'graphicsDataCircles' represents the graphicsData for the circles

- **example:**

```
#list with circle center, radius and side at which rope runs
circleList = [[[0,0],0.2,'L'],
              [[0,1],0.2,'L'],
              [[0.8,0.8],0.4,'L'],
              [[1,0],0.2,'L'],
              [[0,0],0.2,'L'],
              [[0,1],0.2,'L'],
              ]
[] = CreateReevingCurve(circleList,
                        removeLastLine=True, #allows closed curve
                        numberOfANCFnodes=50)
```

For examples on CreateReevingCurve see Examples (Ex) and TestModels (TM):

- [beltDriveReevingSystem.py](#) (Ex), [reevingSystem.py](#) (Ex)
- 

```
def PointsAndSlopes2ANCFCable2D (mbs, ancfPointsSlopes, elementLengths, cableTemplate,
massProportionalLoad= [0,0,0], fixedConstraintsNode0= [0,0,0,0], fixedConstraintsNode1= [0,0,0,0],
firstNodeIsLastNode= True, elementCurvatures= [], graphicsSizeConstraints= -1)
```

- **function description:** Create nodes and ANCF Cable2D elements in MainSystem mbs from a given set of nodes, elements lengths and a template for the cable, based on output of function CreateReevingCurve(...); function works similar to GenerateStraightLineANCF Cable2D, but for arbitrary geometry (curved elements); optionally add loads and constraints
- **input:**
  - mbs*: the system where ANCF elements and nodes are added
  - ancfPointsSlopes*: list of position and slopes for nodes, provided as 4D numpy arrays, as returned by CreateReevingCurve(...)
  - elementLengths*: list of element lengths per element, as returned by CreateReevingCurve(...)
  - cableTemplate*: a ObjectANCF Cable2D object, containing the desired cable properties; cable length and node numbers are set automatically
  - massProportionalLoad*: a 3D list or np.array, containing the gravity vector to be applied to all elements or zero
  - fixedConstraintsNode0*: a list of 4 binary values, indicating the coordinate constraints on the first node (x,y-position and x,y-slope)
  - fixedConstraintsNode1*: a list of 4 binary values, indicating the coordinate constraints on the last node (x,y-position and x,y-slope)
  - firstNodeIsLastNode*: if True, then the last node is using the node number of the first node and the curve is closed; otherwise, the first and last nodes are different, and the curve is open
  - elementCurvatures*: optional list of pre-curvatures of elements, used to override the cableTemplate entry 'physicsReferenceCurvature'; use 0. for straight lines!
  - graphicsSizeConstraints*: if set other than -1, it will be used as the size for drawing applied coordinate constraints
- **output:** returns a list [cableNodeList, cableObjectList, loadList, cableNodePositionList, cableCoordinateConstraintList]

For examples on PointsAndSlopes2ANCF Cable2D see Examples (Ex) and TestModels (TM):

- [beltDriveReevingSystem.py](#) (Ex), [reevingSystem.py](#) (Ex)
- 

```
def GenerateSlidingJoint (mbs, cableObjectList, markerBodyPositionOfSlidingBody,
localMarkerIndexOfStartCable= 0, slidingCoordinateStartPosition= 0)
```

- **function description:** generate a sliding joint from a list of cables, marker to a sliding body, etc.

For examples on GenerateSlidingJoint see Examples (Ex) and TestModels (TM):

- [ACNFslidingAndALEjointTest.py](#) (TM)
- 

```
def GenerateAleSlidingJoint (mbs, cableObjectList, markerBodyPositionOfSlidingBody, AleNode,
localMarkerIndexOfStartCable= 0, AleSlidingOffset= 0, activeConnector= True, penaltyStiffness= 0)
```

- **function description:** generate an ALE sliding joint from a list of cables, marker to a sliding body, etc.

For examples on GenerateAleSlidingJoint see Examples (Ex) and TestModels (TM):

- [ACNFslidingAndALEjointTest.py](#) (TM)

## 5.4 Module: FEM

Support functions and helper classes for import of meshes, finite element models (ABAQUS, ANSYS, NETGEN) and for generation of FFRF (floating frame of reference) objects.

Author: Johannes Gerstmayr; Stefan Holzinger (Abaqus and Ansys import utilities); Joachim Schöberl (support for NGsolve import and eigen computations)

Date: 2020-03-10 (created)

Notes: internal CSR matrix storage format contains 3 float numbers per row: [row, column, value], can be converted to scipy csr sparse matrices with function CSRtoScipySparseCSR(...)

def [CompressedRowSparseToDenseMatrix](#) (*sparseData*)

- **function description:** convert zero-based sparse matrix data to dense numpy matrix
  - **input:** *sparseData*: format (per row): [row, column, value] ==> converted into dense format
  - **output:** a dense matrix as np.array
- 

def [MapSparseMatrixIndices](#) (*matrix, sorting*)

- **function description:** resort a sparse matrix (internal CSR format) with given sorting for rows and columns; changes matrix directly! used for ANSYS matrix import
- 

def [VectorDiadicUnitMatrix3D](#) (*v*)

- **function description:** compute diadic product of vector v and a 3D unit matrix = diadic(v,I<sub>3x3</sub>); used for ObjectFFRF and CMS implementation
- 

def [CyclicCompareReversed](#) (*list1, list2*)

- **function description:** compare cyclic two lists, reverse second list; return True, if any cyclic shifted lists are same, False otherwise

---

```
def AddEntryToCompressedRowSparseArray (sparseData, row, column, value)
```

- **function description:**

add entry to compressedRowSparse matrix, avoiding duplicates

value is either added to existing entry (avoid duplicates) or a new entry is appended

---

```
def CSRtoRowsAndColumns (sparseMatrixCSR)
```

- **function description:** compute rows and columns of a compressed sparse matrix and return as tuple: (rows,columns)

---

```
def CSRtoScipySparseCSR (sparseMatrixCSR)
```

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

---

```
def ScipySparseCSRtoCSR (scipyCSR)
```

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

---

```
def ResortIndicesOfCSRmatrix (mXXYYZZ, numberOfRows)
```

- **function description:**

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

needed for import from NGsolve

---

```
def ResortIndicesOfNGvector (vXXYYZZ)
```

- **function description:** resort indices of given NGsolve vector in XXXYYYZZZ format to XYZXYZXYZ format

---

```
def ResortIndicesExudyn2NGvector (vXYZXYZ)
```

- **function description:** resort indices of given Exudyun vector XYZXYZXYZ to NGsolve vector in XXXYYYYZZZ format
- 

```
def ConvertHexToTrigs (nodeNumbers)
```

- **function description:**

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

For examples on ConvertHexToTrigs see Examples (Ex) and TestModels (TM):

- [objectFFRFTest.py](#) (TM)
- 

```
def ConvertDenseToCompressedRowMatrix (denseMatrix)
```

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

```
def ReadMatrixFromAnsysMMF (fileName, verbose= False)
```

- **function description:**

This function reads either the mass or stiffness matrix from an Ansys Matrix Market Format (MMF). The corresponding matrix can either be exported as dense matrix or sparse matrix.

- **input:** *fileName* of MMF file

- **output:** internal compressed row sparse matrix (as (nrows x 3) numpy array)

- **author:** Stefan Holzinger

- **notes:**

A MMF file can be created in Ansys by placing the following APDL code inside *the solution tree in Ansys Workbench*:

!!!!!!!!!!!!!!

! APDL code that exports sparse stiffness and mass matrix in MMF format. If ! the dense matrix is needed, replace \*SMAT with \*DMAT in the following

```

! APDL code.

! Export the stiffness matrix in MMF format
*SMAT,MatKD,D,IMPORT,FULL,file.full,STIFF

*EXPORT,MatKD,MMF,fileNameStiffnessMatrix,,,
! Export the mass matrix in MMF format
*SMAT,MatMD,D,IMPORT,FULL,file.full,MASS
*EXPORT,MatMD,MMF,fileNameMassMatrix,,,
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

```

In case a lumped mass matrix is needed, place the following APDL Code inside *the Modal Analysis Tree*:

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

! APDL code to force Ansys to use a lumped mass formulation (if available for  
! used elements)

```
LUMPM, ON, , 0
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

---

def [ReadMatrixDOFmappingVectorFromAnsysTxt](#) (*fileName*)

- **function description:**

read sorting vector for ANSYS mass and stiffness matrices and return sorting vector as np.array  
the file contains sorting for nodes and applies this sorting to the DOF (assuming 3 DOF per node!)  
the resulting sorted vector is already converted to 0-based indices

---

def [ReadNodalCoordinatesFromAnsysTxt](#) (*fileName*, *verbose*= False)

- **function description:** This function reads the nodal coordinates exported from Ansys.
- **input:** *fileName* (file name ending must be .txt!)
- **output:** nodal coordinates as numpy array
- **author:** Stefan Holzinger
- **notes:**

The nodal coordinates can be exported from Ansys by creating a named selection of the body whos mesh should to exported by choosing its geometry. Next, create a second named selction by using a worksheet. Add the named selection that was created first into the worksheet of the second named selection.

---

Inside the working sheet, choose 'convert' and convert the first created named selection to 'mesh node' (Netzknoten in german) and click on generate to create the second named selection. Next, right click on the second named selection tha was created and choose 'export' and save the nodal coordinates as .txt file.

---

```
def ReadElementsFromAnsysTxt (fileName, verbose= False)
```

- **function description:** This function reads the nodal coordinates exported from Ansys.
- **input:** fileName (file name ending must be .txt!)
- **output:** element connectivity as numpy array
- **author:** Stefan Holzinger
- **notes:**

The elements can be exported from Ansys by creating a named selection of the body whos mesh should to exported by choosing its geometry. Next, create a second named selction by using a worksheet. Add the named selection that was created first into the worksheet of the second named selection.

Inside the worksheet, choose 'convert' and convert the first created named selection to 'mesh element' (Netzelement in german) and click on generate to create the second named selection. Next, right click on the second named selection tha was created and choose 'export' and save the elements as .txt file.

---

```
def CMSObjectComputeNorm (mbs, objectNumber, outputVariableType, norm= 'max', nodeNumberList= [])
```

- **function description:** compute current (max, min, ...) value for chosen ObjectFFRFreducedOrder object (CMSobject) with exu.OutputVariableType. The function operates on nodal values. This is a helper function, which can be used to conveniently compute output quantities of the CMSobject efficiently and to use it in sensors
- **input:**

*mbs*: MainSystem of *objectNumber*

*objectNumber*: number of ObjectFFRFreducedOrder in *mbs*

*outputVariableType*: a exu.OutputVariableType out of [StressLocal, DisplacementLocal, VelocityLocal]

*norm*: string containing chosen norm to be computed, out of 'Mises', 'maxNorm', 'min', 'max'; 'max' will return maximum of all components (component wise), 'min' does same but for minimum; 'maxNorm' computes np.linalg.norm for every node and then takes maximum of all norms; Mises computes von-Mises stress for every node and then takes maximum of all nodes

*nodeNumberList*: list of mesh node numbers (from FEMinterface); if empty [], all nodes are used; otherwise, only given nodes are evaluated

- **output**: return value or list of values according to chosen norm as np.array

For examples on CMSObjectComputeNorm see Examples (Ex) and TestModels (TM):

- [NGsolveCMSTutorial.py](#) (Ex)

#### 5.4.1 CLASS MaterialBaseClass (in module FEM)

**class description**: material base class, e.g., for FiniteElement

#### 5.4.2 CLASS KirchhoffMaterial(MaterialBaseClass) (in module FEM)

**class description**: class for representation of Kirchhoff (linear elastic, 3D and 2D) material

- **notes**: use planeStress=False for plane strain

---

`def Strain2Stress (self, strain)`

- **classFunction**: convert strain tensor into stress tensor using elasticity tensor

---

`def StrainVector2StressVector (self, strainVector)`

- **classFunction**: convert strain vector into stress vector

---

`def StrainVector2StressVector2D (self, strainVector2D)`

- **classFunction**: compute 2D stress vector from strain vector

---

`def LameParameters (self)`

- **classFunction**: compute Lame parameters from internal Young's modulus and Poisson ratio

- **output:** return vector [mu, lam] of Lame parameters

For examples on KirchhoffMaterial see Examples (Ex) and TestModels (TM):

- [CMSexampleCourse.py](#) (Ex), [NGsolveCMTutorial.py](#) (Ex), [NGsolveCraigBampton.py](#) (Ex), [NGsolvePistonEngine.py](#) (Ex), [NGsolvePostProcessingStresses.py](#) (Ex), ...

### 5.4.3 CLASS FiniteElement (in module FEM)

**class description:** finite element base class for lateron implementations of other finite elements

### 5.4.4 CLASS Tet4(FiniteElement) (in module FEM)

**class description:** simplistic 4-noded tetrahedral interface to compute strain/stress at nodal points

### 5.4.5 CLASS ObjectFFRFinterface (in module FEM)

**class description:** compute terms necessary for ObjectFFRF class used internally in FEMinterface to compute ObjectFFRF object this class holds all data for ObjectFFRF user functions

```
def __init__(self, femInterface)
```

- **classFunction:**

initialize ObjectFFRFinterface with FEMinterface class

initializes the ObjectFFRFinterface with nodes, modes, surface description and systemmatrices from FEMinterface

data is then transferred to mbs object with classFunction AddObjectFFRF(...)

```
def AddObjectFFRF(self, exu, mbs, positionRef= [0,0,0], eulerParametersRef= [1,0,0,0], initialVelocity= [0,0,0],  
initialAngularVelocity= [0,0,0], gravity= [0,0,0], constrainRigidBodyMotion= True, massProportionalDamping= 0,  
stiffnessProportionalDamping= 0, color= [0.1,0.9,0.1,1.])
```

- **classFunction:** add according nodes, objects and constraints for FFRF object to MainSystem mbs; only implemented for Euler parameters

- **input:**

*exu:* the exudyn module

*mbs:* a MainSystem object

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

*eulerParametersRef:* reference euler parameters of created ObjectFFRF (set in rigid body node underlying to ObjectFFRF)

*initialVelocity*: initial velocity of created ObjectFFRF (set in rigid body node underlying to ObjectFFRF)  
*initialAngularVelocity*: initial angular velocity of created ObjectFFRF (set in rigid body node underlying to ObjectFFRF)  
*gravity*: set [0,0,0] if no gravity shall be applied, or to the gravity vector otherwise  
*constrainRigidBodyMotion*: set True in order to add constraint (Tisserand frame) in order to suppress rigid motion of mesh nodes  
*color*: provided as list of 4 RGBA values  
add object to mbs as well as according nodes

---

`def UFforce (self, exu, mbs, t, q, q_t)`

- **classFunction**: optional forceUserFunction for ObjectFFRF (per default, this user function is ignored)
- 

`def UFmassGenericODE2 (self, exu, mbs, t, q, q_t)`

- **classFunction**: optional massMatrixUserFunction for ObjectFFRF (per default, this user function is ignored)

For examples on ObjectFFRFinterface see Examples (Ex) and TestModels (TM):

- [objectFFRFTest2.py](#) (TM)

#### 5.4.6 CLASS ObjectFFRFreducedOrderInterface (in module FEM)

**class description**: compute terms necessary for ObjectFFRFreducedOrder class used internally in FEMinterface to compute ObjectFFRFreducedOrder dictionary this class holds all data for ObjectFFRFreducedOrder user functions

`def __init__ (self, femInterface, rigidBodyNodeType= 'NodeType.RotationEulerParameters', roundMassMatrix= 1e-13, roundStiffnessMatrix= 1e-13)`

- **classFunction**:  
initialize ObjectFFRFreducedOrderInterface with FEMinterface class  
initializes the ObjectFFRFreducedOrderInterface with nodes, modes, surface description and reduced system matrices from FEMinterface  
data is then transferred to mbs object with classFunction AddObjectFFRFreducedOrderWithUserFunctions(...)

- **input**:

*femInterface*: must provide nodes, surfaceTriangles, modeBasis, massMatrix, stiffness

*roundMassMatrix*: use this value to set entries of reduced mass matrix to zero which are below the threshold

*roundStiffnessMatrix*: use this value to set entries of reduced stiffness matrix to zero which are below the threshold

---

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

- **classFunction**: add according nodes, objects and constraints for ObjectFFRFreducedOrder object to Main-System mbs; use this function with userfunctions=0 in order to use internal C++ functionality, which is approx. 10x faster; implementation of userfunctions also available for rotation vector (Lie group formulation), which needs further testing

– **input**:

*exu*: the exudyn module

*mbs*: a MainSystem object

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

*initialVelocity*: initial velocity of created ObjectFFRFreducedOrder (set in rigid body node underlying to ObjectFFRFreducedOrder)

*rotationMatrixRef*: reference rotation of created ObjectFFRFreducedOrder (set in rigid body node underlying to ObjectFFRFreducedOrder); if [], it becomes the unit matrix

*initialAngularVelocity*: initial angular velocity of created ObjectFFRFreducedOrder (set in rigid body node underlying to ObjectFFRFreducedOrder)

*eulerParametersRef*: DEPRECATED, use rotationParametersRef or rotationMatrixRef in future: reference euler parameters of created ObjectFFRFreducedOrder (set in rigid body node underlying to ObjectFFRFreducedOrder)

*gravity*: ONLY available if user functions are applied; otherwise use LoadMassProportional and add to ObjectFFRFreducedOrder; set [0,0,0] if no gravity shall be applied, or to the gravity vector otherwise

*UFforce*: (OPTIONAL, computation is slower) provide a user function, which computes the quadratic velocity vector and applied forces; usually this function reads like:

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

*UFmassMatrix*: (OPTIONAL, computation is slower) provide a user function, which computes the quadratic velocity vector and applied forces; usually this function reads like:

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

*massProportionalDamping*: Rayleigh damping factor for mass proportional damping (multiplied with reduced mass matrix), added to floating frame/modal coordinates only

*stiffnessProportionalDamping*: Rayleigh damping factor for stiffness proportional damping, added to floating frame/modal coordinates only (multiplied with reduced stiffness matrix)  
*color*: provided as list of 4 RGBA values

---

`def UFmassFFRFreducedOrder (self, exu, mbs, t, qReduced, qReduced_t)`

- **classFunction**: CMS mass matrix user function; qReduced and qReduced\_t contain the coordinates of the rigid body node and the modal coordinates in one vector!
- 

`def UFForceFFRFreducedOrder (self, exu, mbs, t, qReduced, qReduced_t)`

- **classFunction**: CMS force matrix user function; qReduced and qReduced\_t contain the coordinates of the rigid body node and the modal coordinates in one vector!
- 

`def AddObjectFFRFreducedOrder (self, mbs, positionRef= [0,0,0], initialVelocity= [0,0,0], rotationMatrixRef= [], initialAngularVelocity= [0,0,0], massProportionalDamping= 0, stiffnessProportionalDamping= 0, color= [0.1,0.9,0.1,1.])`

- **classFunction**: add according nodes, objects and constraints for ObjectFFRFreducedOrder object to Main-System mbs; use this function in order to use internal C++ functionality, which is approx. 10x faster than AddObjectFFRFreducedOrderWithUserFunctions(...)

- **input**:

*exu*: the exudyn module

*mbs*: a MainSystem object

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

*initialVelocity*: initial velocity of created ObjectFFRFreducedOrder (set in rigid body node underlying to ObjectFFRFreducedOrder)

*rotationMatrixRef*: reference rotation of created ObjectFFRFreducedOrder (set in rigid body node underlying to ObjectFFRFreducedOrder); if [], it becomes the unit matrix

*initialAngularVelocity*: initial angular velocity of created ObjectFFRFreducedOrder (set in rigid body node underlying to ObjectFFRFreducedOrder)

*massProportionalDamping*: Rayleigh damping factor for mass proportional damping, added to floating frame/modal coordinates only

*stiffnessProportionalDamping*: Rayleigh damping factor for stiffness proportional damping, added to floating frame/modal coordinates only

*color*: provided as list of 4 RGBA values

For examples on ObjectFFRFreducedOrderInterface see Examples (Ex) and TestModels (TM):

- [CMSexampleCourse.py](#) (Ex), [NGsolveCMSTutorial.py](#) (Ex), [NGsolveCraigBampton.py](#) (Ex), [NGsolvePistonEngine.py](#) (Ex), [NGsolvePostProcessingStresses.py](#) (Ex), ... , [NGsolveCrankShaftTest.py](#) (TM), [objectFFRFreducedOrderAccelerations.py](#) (TM), [objectFFRFreducedOrderShowModes.py](#) (TM), ...

#### 5.4.7 CLASS HCBstaticModeSelection(Enum) (in module FEM)

**class description:** helper calss for function ComputeHurtyCraigBamptonModes, declaring some computation options. It offers the following options:

- allBoundaryNodes: compute a single static mode for every boundary coordinate
- RBE2: static modes only for rigid body motion at boundary nodes
- RBE3: [yet NOT AVAILABLE] averaged rigid body interfaces; for future implementations
- noStaticModes: do not compute static modes, only eigen modes (not recommended; usually only for tests)

#### 5.4.8 CLASS FEMinterface (in module FEM)

**class description:** general interface to different FEM / mesh imports and export to EXUDYN functions use this class to import meshes from different meshing or FEM programs (NETGEN/NGsolve, ABAQUS, ANSYS, ..) and store it in a unique format do mesh operations, compute eigenmodes and reduced basis, etc. load/store the data efficiently with LoadFromFile(...), SaveToFile(...) if import functions are slow export to EXUDYN objects

```
def __init__(self)
    - classFunction: initalize all data of the FEMinterface by, e.g., fem = FEMinterface()
    - example:
        """
        #***** this is not an example, just a description for internal variables *****
        #default values for member variables stored internally in FEMinterface fem and
        #typical structure:
        fem.nodes = {}                      # {'Position':[[x0,y0,z0],...], 'RigidBodyRxyz':[[x0,
        y0,z0],...], ...}
                                                #dictionary of different node lists
        fem.elements = []                    # [{Name:'identifier', 'Tet4':[[n0,n1,n2,n3],...],
        'Hex8':[[n0,...,n7],...], ...}       #there may be several element sets
        fem.massMatrix = np.zeros((0,0))     # np.array([[r0,c0,value0],[r1,c1,value1], ...
        ])]                                 #currently only in SparseCSR format allowed!
        fem.stiffnessMatrix=np.zeros((0,0))  # np.array([[r0,c0,value0],[r1,c1,value1], ...
        ])]                                 #currently only in SparseCSR format allowed!
        fem.surface = []                   # [{Name:'identifier', 'Trigs':[[n0,n1,n2],...], 'Quads':[[n0,...,n3],...], ...}]
                                                #surface with faces
        fem.nodeSets = []                  # [{Name:'identifier', 'NodeNumbers':[n_0,...,n_ns
        ], 'NodeWeights':[w_0,...,w_ns]},...]
                                                #for boundary conditions, etc.
        fem.elementSets = []              # [{Name:'identifier', 'ElementNumbers':[n_0,...,
        n_ns]},...]
                                                #for different volumes, etc.
```

```

fem.modeBasis = {}                      # {'matrix':[[Psi_00,Psi_01, ..., Psi_0m],..., [Psi_n0
    ,Psi_n1, ..., Psi_nm]],'type':'NormalModes'} #'NormalModes' are eigenmodes, '
    HCBmodes' are Craig-Bampton modes including static modes
fem.eigenValues = []                   # [ev0, ev1, ...]
                                         #eigenvalues according to eigenvectors
    in mode basis
fem.postProcessingModes = {}   # {'matrix':<matrix containing stress components (xx,
    yy,zz,yz,zx,xy) in each column, rows are for every mesh node>,'outputVariableType
    ':exodyn.OutputVariableType.StressLocal}

```

---

`def SaveToFile (self, fileName)`

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

`def LoadFromFile (self, fileName)`

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

`def ImportFromAbaqusInputFile (self, fileName, typeName= 'Part', name= 'Part-1', verbose= False)`

- **classFunction:**  
import nodes and elements from Abaqus input file and create surface elements  
node numbers in elements are converted from 1-based indices to python's 0-based indices  
only works for Hex8, Hex20, Tet4 and Tet10 (C3D4, C3D8, C3D10, C3D20) elements  
return node numbers as numpy array
- 

`def ReadMassMatrixFromAbaqus (self, fileName, type= 'SparseRowColumnValue')`

- **classFunction:**  
*read mass matrix from compressed row text format (exported from Abaqus); in order to export system matrices,  
write the following lines in your Abaqus input file:*

```
*STEP  
*MATRIX GENERATE, STIFFNESS, MASS  
*MATRIX OUTPUT, STIFFNESS, MASS, FORMAT=COORDINATE  
*End Step
```

---

```
def ReadStiffnessMatrixFromAbaqus (self, fileName, type= 'SparseRowColumnValue')
```

- **classFunction:** read stiffness matrix from compressed row text format (exported from Abaqus)
- 

```
def ImportMeshFromNGsolve (self, mesh, density, youngsModulus, poissonsRatio, verbose= False,  
computeEigenmodes= False, meshOrder= 1, **kwargs)
```

- **classFunction:** import mesh from NETGEN/NGsolve and setup mechanical problem
  - **input:**
    - mesh:* a previously created ngs.mesh (NGsolve mesh, see examples)
    - youngsModulus:* Young's modulus used for mechanical model
    - poissonsRatio:* Poisson's ratio used for mechanical model
    - density:* density used for mechanical model
    - meshOrder:* use 1 for linear elements and 2 for second order elements (recommended to use 2 for much higher accuracy!)
    - verbose:* set True to print out some status information
  - **output:** creates according nodes, elements, in FEM and returns [bfM, bfK, fes] which are the (mass matrix M, stiffness matrix K) bilinear forms and the finite element space fes
  - **author:** Johannes Gerstmayr, Joachim Schöberl
  - **notes:** setting ngsolve.SetNumThreads(nt) you can select the number of threads that are used for assemble or other functionality with NGsolve functionality
- 

```
def ComputeEigenmodesNGsolve (self, bfM, bfK, nModes, maxEigensolveIterations= 40,  
excludeRigidBodyModes= 0, verbose= False)
```

- **classFunction:** compute nModes smallest eigenvalues and eigenmodes from mass and stiffnessMatrix; store mode vectors in modeBasis, but exclude a number of 'excludeRigidBodyModes' rigid body modes from modeBasis; uses scipy for solution of generalized eigenvalue problem
- **input:**

`nModes`: prescribe the number of modes to be computed; total computed modes are (`nModes+excludeRigidBodyMode`) but only `nModes` with smallest absolute eigenvalues are considered and stored  
`excludeRigidBodyModes`: if rigid body modes are expected (in case of free-free modes), then this number specifies the number of eigenmodes to be excluded in the stored basis (usually 6 modes in 3D)  
`maxEigensolveIterations`: maximum number of iterations for iterative eigensolver; default=40  
`verbose`: if True, output some relevant information during solving

- **output**: eigenmodes are stored internally in FEMinterface as 'modeBasis' and eigenvalues as 'eigenValues'
- **author**: Johannes Gerstmayr, Joachim Schöberl

---

```
def ComputeHurtyCraigBamptonModesNGsolve (self, bfM, bfK, boundaryNodesList, nEigenModes,
maxEigensolveIterations= 40, verbose= False)

    – classFunction: compute static and eigen modes based on Hurty-Craig-Bampton, for details see theory part Section 6.2. This function uses internal computational functionality of NGsolve and is often much faster than the scipy variant

    – input:
        bfM: bilinearform for mass matrix as retured in ImportMeshFromNGsolve(...)
        bfK: bilinearform for stiffness matrix as retured in ImportMeshFromNGsolve(...)
        boundaryNodesList: [nodeList0, nodeList1, ...] a list of node lists, each of them representing a set of 'Position' nodes for which a rigid body interface (displacement/rotation and force/torque) is created; NOTE THAT boundary nodes may not overlap between the different node lists (no duplicated node indices!)
        nEigenModes: number of eigen modes in addition to static modes (may be zero for RBE2 computation-Mode); eigen modes are computed for the case where all rigid body motions at boundaries are fixed; only smallest nEigenModes absolute eigenvalues are considered
        maxEigensolveIterations: maximum number of iterations for iterative eigensolver; default=40
        verbose: if True, output some relevant information during solving
    – output: stores computed modes in self.modeBasis and abs(eigenvalues) in self.eigenValues
    – author: Johannes Gerstmayr, Joachim Schöberl
```

---

```
def ComputePostProcessingModesNGsolve (self, fes, material= 0, outputVariableType=
'OutputVariableType.StressLocal', verbose= False)

    – classFunction: compute special stress or strain modes in order to enable visualization of stresses and strains in ObjectFFRFreducedOrder; takes a NGsolve fes as input and uses internal NGsolve methods to efficiently compute stresses or strains

    – input:
```

*fes*: finite element space as returned in ImportMeshFromNGsolve(...)

*material*: specify material properties for computation of stresses, using a material class, e.g. material = KirchhoffMaterial(Emodulus, nu, rho); not needed for strains (material = 0)

*outputVariableType*: specify either exudyn.OutputVariableType.StressLocal or exudyn.OutputVariableType.StrainLocal as the desired output variables

– **output**: post processing modes are stored in FEMinterface in local variable postProcessingModes as a dictionary, where 'matrix' represents the modes and 'outputVariableType' stores the type of mode as a OutputVariableType

– **author**: Johannes Gerstmayer, Joachim Schöberl

– **notes**: This function is implemented in Python and rather slow for larger meshes; for NGsolve / Netgen meshes, see the according ComputePostProcessingModesNGsolve function, which is usually much faster

---

def GetMassMatrix (*self*, *sparse*= True)

– **classFunction**: get sparse mass matrix in according format

---

def GetStiffnessMatrix (*self*, *sparse*= True)

– **classFunction**: get sparse stiffness matrix in according format

---

def NumberOfNodes (*self*)

– **classFunction**: get total number of nodes

---

def GetNodePositionsAsArray (*self*)

– **classFunction**: get node points as array; only possible, if there exists only one type of Position nodes

– **notes**: in order to obtain a list of certain node positions, see example

– **example**:

```
p=GetNodePositionsAsArray(self)[42] #get node 42 position
nodeList=[1,13,42]
pArray=GetNodePositionsAsArray(self)[nodeList] #get np.array with positions of node
    indices
```

---

```
def GetNodePositionsMean (self, nodeNumberList)
```

- **classFunction:** get mean (average) position of nodes defined by list of node numbers
- 

```
def NumberOfCoordinates (self)
```

- **classFunction:** get number of total nodal coordinates
- 

```
def GetNodeAtPoint (self, point, tolerance= 1e-5, raiseException= True)
```

- **classFunction:**

get node number for node at given point, e.g. p=[0.1,0.5,-0.2], using a tolerance (+/-) if coordinates are available only with reduced accuracy

if not found, it returns an invalid index

---

```
def GetNodesInPlane (self, point, normal, tolerance= 1e-5)
```

- **classFunction:**

get node numbers in plane defined by point p and (normalized) normal vector n using a tolerance for the distance to the plane

if not found, it returns an empty list

---

```
def GetNodesInCube (self, pMin, pMax)
```

- **classFunction:** get node numbers in cube, given by pMin and pMax, containing the minimum and maximum x, y, and z coordinates

- **output:** returns list of nodes; if no nodes found, return an empty list

- **example:**

```
nList = GetNodesInCube([-1, -0.2, 0], [1, 0.5, 0.5])
```

---

```
def GetNodesOnLine (self, p1, p2, tolerance= 1e-5)
```

- **classFunction:** get node numbers lying on line defined by points p1 and p2 and tolerance, which is accepted for points slightly outside the surface
- 

```
def GetNodesOnCylinder (self, p1, p2, radius, tolerance= 1e-5)
```

- **classFunction:**

get node numbers lying on cylinder surface; cylinder defined by cylinder axes (points p1 and p2), cylinder radius and tolerance, which is accepted for points slightly outside the surface  
if not found, it returns an empty list

---

```
def GetNodesOnCircle (self, point, normal, r, tolerance= 1e-5)
```

- **classFunction:**

get node numbers lying on a circle, by point p, (normalized) normal vector n (which is the axis of the circle) and radius r  
using a tolerance for the distance to the plane  
if not found, it returns an empty list

---

```
def GetSurfaceTriangles (self)
```

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

```
def VolumeToSurfaceElements (self, verbose= False)
```

- **classFunction:**

generate surface elements from volume elements  
stores the surface in self.surface  
only works for one element list and one type ('Hex8') of elements

---

```
def GetGyroscopicMatrix (self, rotationAxis= 2, sparse= True)
```

- **classFunction:** get gyroscopic matrix in according format; rotationAxis=[0,1,2] = [x,y,z]
- 

```
def ScaleMassMatrix (self, factor)
```

- **classFunction:** scale (=multiply) mass matrix with factor
- 

```
def ScaleStiffnessMatrix (self, factor)
```

- **classFunction:** scale (=multiply) stiffness matrix with factor
- 

```
def AddElasticSupportAtNode (self, nodeNumber, springStiffness= [1e8,1e8,1e8])
```

- **classFunction:**

modify stiffness matrix to add elastic support (joint, etc.) to a node; nodeNumber zero based (as everywhere in the code...)

springStiffness must have length according to the node size

---

```
def AddNodeMass (self, nodeNumber, addedMass)
```

- **classFunction:** modify mass matrix by adding a mass to a certain node, modifying directly the mass matrix
- 

```
def CreateLinearFEMObjectGenericODE2 (self, mbs, color= [0.9,0.4,0.4,1.])
```

- **classFunction:** create GenericODE2 object out of (linear) FEM model; uses always the sparse matrix mode, independent of the solver settings; this model can be directly used inside the multibody system as a static or dynamic FEM subsystem undergoing small deformations; computation is several magnitudes slower than ObjectFFRFreducedOrder

- **input:** mbs: multibody system to which the GenericODE2 is added

- **output:** return list [oGenericODE2, nodeList] containing object number of GenericODE2 as well as the list of mbs node numbers of all NodePoint nodes

---

```
def CreateNonlinearFEMObjectGenericODE2NGsolve (self, mbs, mesh, density, youngsModulus, poissonsRatio, meshOrder= 1, color= [0.9,0.4,0.4,1.])
```

- **classFunction:** create GenericODE2 object fully nonlinear FEM model using NGsolve; uses always the sparse matrix mode, independent of the solver settings; this model can be directly used inside the multi-body system as a static or dynamic nonlinear FEM subsystem undergoing large deformations; computation is several magnitudes slower than ObjectFFRFreducedOrder

- **input:**

*mbs*: multibody system to which the GenericODE2 is added

*mesh*: a previously created ngs.mesh (NGsolve mesh, see examples)

*youngsModulus*: Young's modulus used for mechanical model

*poissonsRatio*: Poisson's ratio used for mechanical model

*density*: density used for mechanical model

*meshOrder*: use 1 for linear elements and 2 for second order elements (recommended to use 2 for much higher accuracy!)

- **output:** return list [oGenericODE2, nodeList] containing object number of GenericODE2 as well as the list of mbs node numbers of all NodePoint nodes

- **author:** Johannes Gerstmayr, Joachim Schöberl

- **notes:**

The interface to NETGEN/NGsolve has been created together with Joachim Schöberl, main developer of NETGEN/NGsolve; Thank's a lot!

download NGsolve at: <https://ngsolve.org/>

NGsolve needs Python 3.7 (64bit) ==> use according EXUDYN version!

note that node/element indices in the NGsolve mesh are 1-based and need to be converted to 0-base!

---

```
def ComputeEigenmodes (self, nModes, excludeRigidBodyModes= 0, useSparseSolver= True)
```

- **classFunction:** compute nModes smallest eigenvalues and eigenmodes from mass and stiffnessMatrix; store mode vectors in modeBasis, but exclude a number of 'excludeRigidBodyModes' rigid body modes from modeBasis; uses scipy for solution of generalized eigenvalue problem

- **input:**

*nModes*: prescribe the number of modes to be computed; total computed modes are (nModes+excludeRigidBodyMode) but only nModes with smallest absolute eigenvalues are considered and stored

*excludeRigidBodyModes*: if rigid body modes are expected (in case of free-free modes), then this number specifies the number of eigenmodes to be excluded in the stored basis (usually 6 modes in 3D)

*useSparseSolver*: for larger systems, the sparse solver needs to be used, which iteratively solves the problem and uses a random number generator (internally in ARPACK); therefore, results are not fully repeatable!!!

- **output:** eigenmodes are stored internally in FEMinterface as 'modeBasis' and eigenvalues as 'eigenValues'
  - **notes:** for NGsolve / Netgen meshes, see the according ComputeEigenmodesNGsolve function, which is usually much faster
- 

`def ComputeEigenModesWithBoundaryNodes (self, boundaryNodes, nEigenModes, useSparseSolver= True)`

- **classFunction:** compute eigenmodes, using a set of boundary nodes that are all fixed; very similar to ComputeEigenmodes, but with additional definition of (fixed) boundary nodes.
  - **input:**
    - boundaryNodes*: a list of boundary node indices, referring to 'Position' type nodes in FEMinterface; all coordinates of these nodes are fixed for the computation of the modes
    - nEigenModes*: prescribe the number of modes to be computed; only nEigenModes with smallest abs(eigenvalues) are considered and stored
    - useSparseSolver*: [yet NOT IMPLEMENTED] for larger systems, the sparse solver needs to be used, which iteratively solves the problem and uses a random number generator (internally in ARPACK); therefore, results are not fully repeatable!!!
  - **output:** eigenmodes are stored internally in FEMinterface as 'modeBasis' and eigenvalues as 'eigenValues'
- 

`def ComputeHurtyCraigBamptonModes (self, boundaryNodesList, nEigenModes, useSparseSolver= True, computationMode= HCBstaticModeSelection.RBE2)`

- **classFunction:** compute static and eigen modes based on Hurty-Craig-Bampton, for details see theory part [Section 6.2](#). Note that this function may need significant time, depending on your hardware, but 50.000 nodes will require approx. 1-2 minutes and more nodes typically raise time more than linearly.
- **input:**
  - boundaryNodesList*: [nodeList0, nodeList1, ...] a list of node lists, each of them representing a set of 'Position' nodes for which a rigid body interface (displacement/rotation and force/torque) is created; NOTE THAT boundary nodes may not overlap between the different node lists (no duplicated node indices!)
  - nEigenModes*: number of eigen modes in addition to static modes (may be zero for RBE2 computation-Mode); eigen modes are computed for the case where all rigid body motions at boundaries are fixed; only smallest nEigenModes absolute eigenvalues are considered
  - useSparseSolver*: for more than approx. 500 nodes, it is recommended to use the sparse solver
  - computationMode*: see class HCBstaticModeSelection for available modes; select RBE2 as standard, which is both efficient and accurate and which uses rigid-body-interfaces (6 independent modes) per boundary
- **output:** stores computed modes in self.modeBasis and abs(eigenvalues) in self.eigenValues

- **notes:** for NGsolve / Netgen meshes, see the according ComputeHurtyCraigBamptonModesNGsolve function, which is usually much faster
- 

```
def GetEigenFrequenciesHz (self)
```

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

```
def ComputePostProcessingModes (self, material= 0, outputVariableType= 'OutputVariableType.StressLocal',  
numberOfThreads= 1)
```

- **classFunction:** compute special stress or strain modes in order to enable visualization of stresses and strains in ObjectFFRFreducedOrder;
  - **input:**
    - material:* specify material properties for computation of stresses, using a material class, e.g. material = KirchhoffMaterial(Emodulus, nu, rho); not needed for strains
    - outputVariableType:* specify either exudyn.OutputVariableType.StressLocal or exudyn.OutputVariableType.StrainLocal as the desired output variables
    - numberOfThreads:* if numberOfThreads=1, it uses single threaded computation; if numberOfThreads>1, it uses the multiprocessing pools functionality, which requires that all code in your main file must be encapsulated within an if clause "if \_\_name\_\_ == '\_\_main\_\_':", see examples; if numberOfThreads==1, it uses all threads/CPUs available
  - **output:** post processing modes are stored in FEMinterface in local variable postProcessingModes as a dictionary, where 'matrix' represents the modes and 'outputVariableType' stores the type of mode as a OutputVariableType
  - **notes:** This function is implemented in Python and rather slow for larger meshes; for NGsolve / Netgen meshes, see the according ComputePostProcessingModesNGsolve function, which is usually much faster
- 

```
def ComputeCampbellDiagram (self, terminalFrequency, nEigenfrequencies= 10, frequencySteps= 25,  
rotationAxis= 2, plotDiagram= False, verbose= False, useCorotationalFrame= False, useSparseSolver= False)
```

- **classFunction:**
  - compute Campbell diagram for given mechanical system
  - create a first order system  $Ax + Bx = 0$  with  $x = [q, \dot{q}]'$  and compute eigenvalues
  - takes mass M, stiffness K and gyroscopic matrix G from FEMinterface
  - currently only uses dense matrices, so it is limited to approx. 5000 unknowns!

- **input:**

*terminalFrequency*: frequency in Hz, up to which the campbell diagram is computed  
*nEigenfrequencies*: gives the number of computed eigenfrequencies(modes), in addition to the rigid body mode 0  
*frequencySteps*: gives the number of increments (gives frequencySteps+1 total points in campbell diagram)  
*rotationAxis*:[0,1,2] = [x,y,z] provides rotation axis  
*plotDiagram*: if True, plots diagram for nEigenfrequencies before terminating  
*verbose*: if True, shows progress of computation; if verbose=2, prints also eigenfrequencies  
*useCorotationalFrame*: if False, the classic rotor dynamics formulation for rotationally-symmetric rotors is used, where the rotor can be understood in a Lagrangian-Eulerian manner: the rotation is represented by an additional (Eulerian) velocity in rotation direction; if True, the corotational frame is used, which gives a factor 2 in the gyroscopic matrix and can be used for non-symmetric rotors as well  
*useSparseSolver*: for larger systems, the sparse solver needs to be used for creation of system matrices and for the eigenvalue solver (uses a random number generator internally in ARPACK, therefore, results are not fully repeatable!!!)

– **output:**

[listFrequencies, campbellFrequencies]  
*listFrequencies*: list of computed frequencies  
*campbellFrequencies*: array of campbell frequencies per eigenfrequency of system

---

`def CheckConsistency (self)`

- **classFunction:** perform some consistency checks
- 

`def ReadMassMatrixFromAnsys (self, fileName, dofMappingVectorFile, sparse= True, verbose= False)`

- **classFunction:** read mass matrix from CSV format (exported from Ansys)
- 

`def ReadStiffnessMatrixFromAnsys (self, fileName, dofMappingVectorFile, sparse= True, verbose= False)`

- **classFunction:** read stiffness matrix from CSV format (exported from Ansys)
-

```
def ReadNodalCoordinatesFromAnsys (self, fileName, verbose= False)
```

- **classFunction:** read nodal coordinates (exported from Ansys as .txt-File)
- 

```
def ReadElementsFromAnsys (self, fileName, verbose= False)
```

- **classFunction:** read elements (exported from Ansys as .txt-File)

For examples on FEMinterface see Examples (Ex) and TestModels (TM):

- [CMExampleCourse.py](#) (Ex), [NGsolveCMTutorial.py](#) (Ex), [NGsolveCraigBampton.py](#) (Ex),  
[NGsolvePistonEngine.py](#) (Ex), [NGsolvePostProcessingStresses.py](#) (Ex), ..., [compareAbaqusAnsysRotorEigenfrequencies.py](#) (TM), [NGsolveCrankShaftTest.py](#) (TM), [objectFFRFreducedOrderAccelerations.py](#) (TM), ...

## 5.5 Module: graphicsDataUtilities

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

Author: Johannes Gerstmayr

Date: 2020-07-26 (created)

Notes: Some useful colors are defined, using RGBA (Red, Green, Blue and Alpha = opacity) channels in the range [0,1], e.g., red = [1,0,0,1].

Available colors are: color4red, color4green, color4blue, color4cyan, color4magenta, color4yellow, color4orange, color4pink, color4lawngreen, color4violet, color4springgreen, color4dodgerblue, color4grey, color4darkgrey, color4lightgrey, color4lightred, color4lightgreen, color4steelblue, color4brown, color4black, color4darkgrey2, color4lightgrey2, color4white

Additionally, a list of 16 colors 'color4list' is available, which is intended to be used, e.g., for creating n bodies with different colors

```
def SwitchTripletOrder (vector)
```

- **function description:** helper function to switch order of three items in a list; mostly used for reverting normals in triangles
  - **input:** 3D vector as list or as np.array
  - **output:** interchanged 2nd and 3rd component of list
- 

```
def ComputeTriangleNormal (p0, p1, p2)
```

- **function description:** compute normalized normal for 3 triangle points

- **input:** 3D vector as list or as np.array
  - **output:** normal as np.array
- 

```
def GraphicsData2PointsAndTrigs (g)
```

- **function description:** convert graphics data into list of points and list of triangle indices (triplets)
- **input:** g contains a GraphicsData with type TriangleList
- **output:** returns [points, triangles], with points as list of np.array with 3 floats per point and triangles as a list of np.array with 3 int per triangle (0-based indices to points)

For examples on GraphicsData2PointsAndTrigs see Examples (Ex) and TestModels (TM):

- [particleClusters.py](#) (Ex), [tippeTop.py](#) (Ex), [generalContactFrictionTests.py](#) (TM)
- 

```
def GraphicsDataFromPointsAndTrigs (points, triangles, color= [0.,0.,0.,1.])
```

- **function description:** convert triangles and points as returned from GraphicsData2TrigsAndPoints(...)
- **input:**
  - points:* list of np.array with 3 floats per point
  - triangles:* list of np.array with 3 int per triangle (0-based indices to triangles)
  - color:* provided as list of 4 RGBA values
- **output:** returns GraphicsData with type TriangleList

For examples on GraphicsDataFromPointsAndTrigs see Examples (Ex) and TestModels (TM):

- [generalContactFrictionTests.py](#) (TM)
- 

```
def RefineMesh (points, triangles)
```

- **function description:** refine triangle mesh; every triangle is subdivided into 4 triangles
- **input:**
  - points:* list of np.array with 3 floats per point
  - triangles:* list of np.array with 3 int per triangle (0-based indices to triangles)
- **output:** returns [points2, triangles2] containing the refined mesh; if the original mesh is consistent, no points are duplicated; if the mesh is not consistent, some mesh points are duplicated!
- **notes:** becomes slow for meshes with more than 5000 points

For examples on RefineMesh see Examples (Ex) and TestModels (TM):

- [particleClusters.py](#) (Ex), [tippeTop.py](#) (Ex), [generalContactFrictionTests.py](#) (TM)
- 

```
def ShrinkMeshNormalToSurface (points, triangles, distance)
```

- **function description:** shrink mesh using triangle normals; every point is at least moved a distance '*distance*' normal from boundary
- **input:**
  - points*: list of np.array with 3 floats per point
  - triangles*: list of np.array with 3 int per triangle (0-based indices to triangles)
  - distance*: float value of minimum distance
- **output:** returns [*points2*, *triangles2*] containing the refined mesh; currently the points of the subdivided triangles are duplicated!
- **notes:** ONLY works for consistent meshes (no duplicated points!)

For examples on ShrinkMeshNormalToSurface see Examples (Ex) and TestModels (TM):

- [generalContactFrictionTests.py](#) (TM)
- 

```
def MoveGraphicsData (g, pOff, Aoff)
```

- **function description:** add rigid body transformation to GraphicsData, using position offset (global) *pOff* (list or np.array) and rotation *Aoff* (transforms local to global coordinates; list of lists or np.array)

For examples on MoveGraphicsData see Examples (Ex) and TestModels (TM):

- [rigidBodyAsUserFunctionTest.py](#) (TM)
- 

```
def MergeGraphicsDataTriangleList (g1, g2)
```

- **function description:** merge 2 different graphics data with triangle lists
- **input:** graphicsData dictionaries *g1* and *g2* obtained from GraphicsData functions
- **output:** one graphicsData dictionary with single triangle lists and compatible points and normals, to be used in visualization of EXUDYN objects

For examples on MergeGraphicsDataTriangleList see Examples (Ex) and TestModels (TM):

- [particleClusters.py](#) (Ex), [generalContactFrictionTests.py](#) (TM)
-

```
def GraphicsDataRectangle (xMin, yMin, xMax, yMax, color= [0.,0.,0.,1.])
```

- **function description:** generate graphics data for 2D rectangle
- **input:** minimal and maximal cartesian coordinates in (x/y) plane; color provided as list of 4 RGBA values
- **output:** graphicsData dictionary, to be used in visualization of EXUDYN objects

For examples on GraphicsDataRectangle see Examples (Ex) and TestModels (TM):

- [ANCF\\_switchingSlidingJoint2D.py](#) (Ex), [lavalRotor2Dtest.py](#) (Ex), [particleClusters.py](#) (Ex), [particlesTest.py](#) (Ex), [particlesTest3D.py](#) (Ex), ... , [ACNFslidingAndALEjointTest.py](#) (TM), [ANCFcontactFrictionTest.py](#) (TM), [ANCFmovingRigidBodyTest.py](#) (TM), ...
- 

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

- **function description:** generate graphics data for orthogonal cube drawn with lines
- **input:** minimal and maximal cartesian coordinates for orthogonal cube; color provided as list of 4 RGBA values
- **output:** graphicsData dictionary, to be used in visualization of EXUDYN objects

For examples on GraphicsDataOrthoCubeLines see Examples (Ex) and TestModels (TM):

- [rigid3Dexample.py](#) (Ex), [genericJointUserFunctionTest.py](#) (TM), [rigidBodyCOMtest.py](#) (TM), [sphericalJointTest.py](#) (TM)
- 

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

- **function description:** generate graphics data for orthogonal 3D cube with min and max dimensions
- **input:** minimal and maximal cartesian coordinates for orthogonal cube; color provided as list of 4 RGBA values
- **output:** graphicsData dictionary, to be used in visualization of EXUDYN objects

For examples on GraphicsDataOrthoCube see Examples (Ex) and TestModels (TM):

- [geneticOptimizationSliderCrank.py](#) (Ex), [massSpringFrictionInteractive.py](#) (Ex), [mouseInteractionExample.py](#) (Ex), [performanceMultiThreadingNG.py](#) (Ex), [rigidBodyIMUTest.py](#) (Ex), ... , [driveTrainTest.py](#) (TM), [explicitLieGroupIntegratorPy](#) (TM), [explicitLieGroupIntegratorTest.py](#) (TM), ...
- 

```
def GraphicsDataOrthoCubePoint (centerPoint= [0,0,0], size= [0.1,0.1,0.1], color= [0.,0.,0.,1.], addNormals= False)
```

- **function description:** generate graphics data for orthogonal 3D cube with center point and size

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

For examples on GraphicsDataOrthoCubePoint see Examples (Ex) and TestModels (TM):

- [addPrismaticJoint.py](#) (Ex), [addRevoluteJoint.py](#) (Ex), [bicycleIftommBenchmark.py](#) (Ex), [finiteSegmentMethod.py](#) (Ex), [leggedRobot.py](#) (Ex), ... , [carRollingDiscTest.py](#) (TM), [connectorRigidBodySpringDamperTest.py](#) (TM), [contactCoordinateTest.py](#) (TM), ...
- 

```
def GraphicsDataCube (pList, color= [0.,0.,0.,1.], faces= [1,1,1,1,1,1], addNormals= False)
```

- **function description:** generate graphics data for general cube with endpoints, according to given vertex definition
  - **input:**
    - pList*: is a list of points [[x0,y0,z0],[x1,y1,z1],...]
    - color*: provided as list of 4 RGBA values
    - faces*: includes the list of six binary values (0/1), denoting active faces (value=1); set index to zero to hide face
    - addNormals*: if True, normals are added and there are separate points for every triangle
  - **output:** graphicsData dictionary, to be used in visualization of EXUDYN objects
- 

```
def GraphicsDataSphere (point= [0,0,0], radius= 0.1, color= [0.,0.,0.,1.], nTiles= 8)
```

- **function description:** generate graphics data for a sphere with point p and radius
- **input:**
  - point*: center of sphere (3D list or np.array)
  - radius*: positive value
  - color*: provided as list of 4 RGBA values
  - nTiles*: used to determine resolution of sphere >=3; use larger values for finer resolution
- **output:** graphicsData dictionary, to be used in visualization of EXUDYN objects

For examples on GraphicsDataSphere see Examples (Ex) and TestModels (TM):

- [bicycleIftommBenchmark.py](#) (Ex), [lugreFrictionTest.py](#) (Ex), [nMassOscillatorInteractive.py](#) (Ex), [particleClusters.py](#) (Ex), [particlesTest3D.py](#) (Ex), ... , [connectorGravityTest.py](#) (TM), [contactCoordinateTest.py](#) (TM), [coordinateVectorConstraint.py](#) (TM), ...
-

```

def GraphicsDataCylinder (pAxis= [0,0,0], vAxis= [0,0,1], radius= 0.1, color= [0.,0.,0.,1.], nTiles= 16,
angleRange= [0,2*np.pi], lastFace= True, cutPlain= True, **kwargs)

```

- **function description:** generate graphics data for a cylinder with given axis, radius and color; nTiles gives the number of tiles (minimum=3)
- **input:**
  - pAxis*: axis point of one face of cylinder (3D list or np.array)
  - vAxis*: vector representing the cylinder's axis (3D list or np.array)
  - radius*: positive value representing radius of cylinder
  - color*: provided as list of 4 RGBA values
  - nTiles*: used to determine resolution of cylinder >=3; use larger values for finer resolution
  - angleRange*: given in rad, to draw only part of cylinder (halfcylinder, etc.); for full range use [0..2 \* pi]
  - lastFace*: if angleRange != [0,2\*pi], then the faces of the open cylinder are shown with lastFace = True
  - cutPlain*: only used for angleRange != [0,2\*pi]; if True, a plane is cut through the part of the cylinder; if False, the cylinder becomes a cake shape ...
  - alternatingColor*: if given, optionally another color in order to see rotation of solid; only works, if angleRange=[0,2\*pi]
- **output:** graphicsData dictionary, to be used in visualization of EXUDYN objects

For examples on GraphicsDataCylinder see Examples (Ex) and TestModels (TM):

- [beltDriveReevingSystem.py](#) (Ex), [bicycleIftommBenchmark.py](#) (Ex), [flexibleRotor3Dtest.py](#) (Ex), [gyroStability.py](#) (Ex), [mouseInteractionExample.py](#) (Ex), ..., [ANCFbeltDrive.py](#) (TM), [ANCFgeneralContactCircle.py](#) (TM), [driveTrainTest.py](#) (TM), ...
- 

```

def GraphicsDataRigidLink (p0, p1, axis0= [0,0,0], axis1= [0,0,0], radius= [0.1,0.1], thickness= 0.05, width=
[0.05,0.05], color= [0.,0.,0.,1.], nTiles= 16)

```

- **function description:** generate graphics data for a planar Link between the two joint positions, having two axes
- **input:**
  - p0*: joint0 center position
  - p1*: joint1 center position
  - axis0*: direction of rotation axis at p0, if drawn as a cylinder; [0,0,0] otherwise
  - axis1*: direction of rotation axis of p1, if drawn as a cylinder; [0,0,0] otherwise
  - radius*: list of two radii [radius0, radius1], being the two radii of the joints drawn by a cylinder or sphere
  - width*: list of two widths [width0, width1], being the two widths of the joints drawn by a cylinder; ignored for sphere
  - thickness*: the thickness of the link (shaft) between the two joint positions; thickness in z-direction or diameter (cylinder)
  - color*: provided as list of 4 RGBA values

*nTiles*: used to determine resolution of cylinder  $\geq 3$ ; use larger values for finer resolution

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

For examples on GraphicsDataRigidLink see Examples (Ex) and TestModels (TM):

- [fourBarMechanism3D.py](#) (Ex), [geneticOptimizationSliderCrank.py](#) (Ex), [multiMbsTest.py](#) (Ex), [rigidBodyTutorial.py](#) (Ex), [rigidBodyTutorial2.py](#) (Ex), ... , [fourBarMechanismRedundant.py](#) (TM), [sliderCrank3Dbenchmark.py](#) (TM), [sliderCrankFloatingTest.py](#) (TM), ...
- 

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

- **function description:** generate graphics data from STL file (text format!) and use color for visualization
  - **input:**
    - fileName*: string containing directory and filename of STL-file (in text / SCII format) to load
    - color*: provided as list of 4 RGBA values
    - verbose*: if True, useful information is provided during reading
  - **output:** interchanged 2nd and 3rd component of list
- 

```
def GraphicsDataSolidOfRevolution (pAxis, vAxis, contour, color= [0.,0.,0.,1.], nTiles= 16, smoothContour= False, **kwargs)
```

- **function description:** generate graphics data for a solid of revolution with given 3D point and axis, 2D point list for contour, (optional)2D normals and color;
- **input:**
  - pAxis*: axis point of one face of solid of revolution (3D list or np.array)
  - vAxis*: vector representing the solid of revolution's axis (3D list or np.array)
  - contour*: a list of 2D-points, specifying the contour ( $x=$ axis,  $y=$ radius), e.g.:  $[[0,0],[0,0.1],[1,0.1]]$
  - color*: provided as list of 4 RGBA values
  - nTiles*: used to determine resolution of solid; use larger values for finer resolution
  - smoothContour*: if True, the contour is made smooth by auto-computing normals to the contour
  - alternatingColor*: add a second color, which enables to see the rotation of the solid
- **output:** graphicsData dictionary, to be used in visualization of EXUDYN objects
- **example:**

```
#simple contour, using list of 2D points:  
contour=[[0,0.2],[0.3,0.2],[0.5,0.3],[0.7,0.4],[1,0.4],[1,0.]]  
rev1 = GraphicsDataSolidOfRevolution(pAxis=[0,0.5,0], vAxis=[1,0,0],  
                                     contour=contour, color=color4red,  
                                     alternatingColor=color4grey)
```

```

#draw torus:
contour=[]
r = 0.2 #small radius of torus
R = 0.5 #big radius of torus
nc = 16 #discretization of torus
for i in range(nc+3): #+3 in order to remove boundary effects
    contour+=[[r*cos(i/nc*pi*2),R+r*sin(i/nc*pi*2)]]
#use smoothContour to make torus looking smooth
rev2 = GraphicsDataSolidOfRevolution(pAxis=[0,0.5,0], vAxis=[1,0,0],
                                      contour=contour, color=color4red,
                                      nTiles = 64, smoothContour=True)

```

For examples on GraphicsDataSolidOfRevolution see Examples (Ex) and TestModels (TM):

- [ConvexContactTest.py](#) (TM)
- 

```
def GraphicsDataArrow (pAxis, vAxis, radius, color= [0.,0.,0.,1.], headFactor= 2, headStretch= 4, nTiles= 12)
```

- **function description:** generate graphics data for an arrow with given origin, axis, shaft radius, optional size factors for head and color; nTiles gives the number of tiles (minimum=3)
- **input:**
  - pAxis*: axis point of the origin (base) of the arrow (3D list or np.array)
  - vAxis*: vector representing the vector pointing from the origin to the tip (head) of the error (3D list or np.array)
  - radius*: positive value representing radius of shaft cylinder
  - headFactor*: positive value representing the ratio between head's radius and the shaft radius
  - headStretch*: positive value representing the ratio between the head's radius and the head's length
  - color*: provided as list of 4 RGBA values
  - nTiles*: used to determine resolution of arrow (of revolution object) >=3; use larger values for finer resolution
- **output:** graphicsData dictionary, to be used in visualization of EXUDYN objects

For examples on GraphicsDataArrow see Examples (Ex) and TestModels (TM):

- [beltDriveReevingSystem.py](#) (Ex), [reevingSystem.py](#) (Ex), [ANCFbeltDrive.py](#) (TM),  
[ANCFgeneralContactCircle.py](#) (TM)
- 

```
def GraphicsDataBase (origin= [0,0,0], length= 1, colors= [color4red, color4green, color4blue], headFactor= 2,
headStretch= 4, nTiles= 12, **kwargs)
```

- **function description:** generate graphics data for three arrows representing an orthogonal basis with point of origin, shaft radius, optional size factors for head and colors; nTiles gives the number of tiles (minimum=3)

– **input:**

*origin*: point of the origin of the base (3D list or np.array)  
*length*: positive value representing lengths of arrows for basis  
*colors*: provided as list of 3 colors (list of 4 RGBA values)  
*headFactor*: positive value representing the ratio between head's radius and the shaft radius  
*headStretch*: positive value representing the ratio between the head's radius and the head's length  
*nTiles*: used to determine resolution of arrows of basis (of revolution object)  $>= 3$ ; use larger values for finer resolution  
*radius*: positive value representing radius of arrows; default: radius =  $0.01 * \text{length}$

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

For examples on GraphicsDataBasis see Examples (Ex) and TestModels (TM):

- [fourBarMechanism3D.py](#) (Ex), [gyroStability.py](#) (Ex), [rigidBodyTutorial3.py](#) (Ex),  
[explicitLieGroupIntegratorTest.py](#) (TM)
- 

```
def GraphicsDataQuad(pList, color=[0.,0.,0.,1.], **kwargs)
```

– **function description:**

generate graphics data for simple quad with option for checkerboard pattern;  
*points are arranged counter-clock-wise*, e.g.: p0=[0,0,0], p1=[1,0,0], p2=[1,1,0], p3=[0,1,0]

– **input:**

*pList*: list of 4 quad points [[x0,y0,z0],[x1,y1,z1],...]

*color*: provided as list of 4 RGBA values

*alternatingColor*: second color; if defined, a checkerboard pattern (default: 10x10) is drawn with color and alternatingColor

*nTiles*: number of tiles for checkerboard pattern (default: 10)

*nTilesY*: if defined, use number of tiles in y-direction different from x-direction (=nTiles)

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

– **example:**

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

For examples on GraphicsDataQuad see Examples (Ex) and TestModels (TM):

- [massSpringFrictionInteractive.py](#) (Ex), [nMassOscillatorInteractive.py](#) (Ex), [simulateInteractively.py](#) (Ex),  
[TCPIPexudynMatlab.py](#) (Ex)
-

```
def GraphicsDataCheckerBoard (point= [0,0,0], normal= [0,0,1], size= 1, color= color4lightgrey,  
alternatingColor= color4lightgrey2, nTiles= 10, **kwargs)
```

– **function description:**

function to generate checkerboard background;

*points are arranged counter-clock-wise, e.g.:*

– **input:**

*point:* midpoint of pattern provided as list or np.array

*normal:* normal to plane provided as list or np.array

*size:* dimension of first side length of quad

*size2:* dimension of second side length of quad

*color:* provided as list of 4 RGBA values

*alternatingColor:* second color; if defined, a checkerboard pattern (default: 10x10) is drawn with color  
and alternatingColor

*nTiles:* number of tiles for checkerboard pattern in first direction

*nTiles2:* number of tiles for checkerboard pattern in second direction; default: nTiles

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

– **example:**

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

For examples on GraphicsDataCheckerBoard see Examples (Ex) and TestModels (TM):

- [bicycleIftommBenchmark.py](#) (Ex), [finiteSegmentMethod.py](#) (Ex), [leggedRobot.py](#) (Ex),  
[tippeTop.py](#) (Ex), [ANCFoutputTest.py](#) (TM), [connectorGravityTest.py](#) (TM), [coordinateVectorConstraint.py](#) (TM),  
[coordinateVectorConstraintGenericODE2.py](#) (TM), ...
- 

```
def ComputeTriangularMesh (vertices, segments)
```

– **function description:**

helper function to compute triangular mesh from list of vertices (=points) and segments;

computes triangular meshes for non-convex case. In order to make it efficient, it first computes

neighbors and then defines triangles at segments to be inside/outside. Finally neighboring

relations are used to define all triangles inside/outside

finally only returns triangles that are inside the segments

– **input:**

*vertices:* list of pairs of coordinates of vertices in mesh [x,y]

*segments:* list of segments, which are pairs of node numbers [i,j], defining the boundary of the mesh;

the ordering of the nodes is such that left triangle = inside, right triangle = outside, compare example with segment [V1,V2]:

inside  
V1 V2  
O——O  
outside

– **output:** triangulation structure of Delaunay(...), see `scipy.spatial.Delaunaystructure`, containing all simplices (=triangles)

– **example:**

```
points = np.array([[0, 0], [0, 2], [2, 2], [2, 1], [1, 1], [0, 1], [1, 0]])
segments = [len(points)-1,0]
for i in range(len(points)-1):
    segments += [i,i+1]
tri = ComputeTriangularMesh(points, segments)
print(tri.simplices)
```

---

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

– **function description:**

create graphicsData for solid extrusion based on 2D points and segments;

additional transformations are possible to translate and rotate

– **input:**

*vertices*: list of pairs of coordinates of vertices in mesh [x,y], see `ComputeTriangularMesh(...)`

*segments*: list of segments, which are pairs of node numbers [i,j], defining the boundary of the mesh; the ordering of the nodes is such that left triangle = inside, right triangle = outside; see `ComputeTriangularMesh(...)`

*height*: height of extruded object

*rot*: rotation matrix, which the extruded object point coordinates are multiplied with before adding offset

*pOff*: 3D offset vector added to extruded coordinates; the z-coordinate of the extrusion object obtains 0 for the base plane, z=height for the top plane

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

## 5.6 Module: GUI

Helper functions and classes for graphical interaction with Exudyn

Author: Johannes Gerstmayer

Date: 2020-01-25

Notes: This is an internal library, which is only used inside Exudyn for modifying settings.

```
def EditDictionaryWithTypeInfo (dictionaryData, exu= None, dictionaryName= 'edit')
```

- **function description:** edit dictionaryData and return modified (new) dictionary
- **input:**
  - dictionaryData*: dictionary obtained from SC.visualizationSettings.GetDictionaryWithTypeInfo()
  - exu*: exudyn module
  - dictionaryName*: name displayed in dialog
- **output:** returns modified dictionary, which can be used, e.g., for SC.visualizationSettings.SetDictionary(...)

## 5.7 Module: interactive

Utilities for interactive simulation and results monitoring

Author: Johannes Gerstmayr

Date: 2021-01-17 (created)

```
def AnimateModes (systemContainer, mainSystem, nodeNumber, period= 0.04, stepsPerPeriod= 30, showTime= True, renderWindowText= "", runOnStart= False, runMode= 0, scaleAmplitude= 1)
```

- **function description:** animate modes of ObjectFFRFreducedOrder and other objects (changes periodically one nodal coordinate); for creating snapshots, press 'Static' and 'Record animation' and press 'Run' to save one figure in the image subfolder; for creating animations for one mode, use the same procedure but use 'One Cycle'
- **input:**
  - systemContainer*: system container (usually SC) of your model, containing visualization settings
  - mainSystem*: system (usually mbs) containing your model
  - nodeNumber*: node number of which the coordinates shall be animated. In case of ObjectFFRFreducedOrder, this is the generic node, e.g., 'nGenericODE2' in the dictionary returned by the function AddObjectFFRFreducedOrderWithUserFunctions(...)
  - period*: delay for animation of every frame; the default of 0.04 results in approximately 25 frames per second
  - stepsPerPeriod*: number of steps into which the animation of one cycle of the mode is split into
  - showTime*: show a virtual time running from 0 to  $2\pi$  during one mode cycle
  - renderWindowText*: additional text written into renderwindow before 'Mode X' (use \n to add line breaks)
  - runOnStart*: immediately go into 'Run' mode
  - runMode*: 0=continuous run, 1=one cycle, 2=static (use slider/mouse to vary time steps)
  - scaleAmplitude*: additional scaling for amplitude if necessary
- **output:** opens interactive dialog with further settings
- **notes:**

Uses class `InteractiveDialog` in the background, which can be used to adjust animation creation.

Press 'Run' to start animation; Chose 'Mode shape', according component for contour plot; to record one cycle for animation, choose 'One cycle', run once to get the according range in the contour plot, press 'Record animation' and press 'Run', now images can be found in subfolder 'images' (for further info on animation creation see [Section 2.4.10](#)); now deactivate 'Record animation' by pressing 'Off' and chose another mode

For examples on `AnimateModes` see Examples (Ex) and TestModels (TM):

- [`CMSexampleCourse.py`](#) (Ex), [`NGsolveCMSTutorial.py`](#) (Ex), [`NGsolveCraigBampton.py`](#) (Ex),  
[`NGsolvePistonEngine.py`](#) (Ex), [`ObjectFFRFconvergenceTestHinge.py`](#) (Ex), ... , [`objectFFRFreducedOrderShowModes.py`](#) (TM)
- 

```
def SolutionViewer (mainSystem, solution= [], rowIncrement= 1, timeout= 0.04, runOnStart= True, runMode= 2)
```

– **function description:** open interactive dialog and visualization (animate) solution loaded with `LoadSolutionFile(...)`; Change slider 'Increment' to change the automatic increment of time frames; Change mode between continuous run, one cycle (fits perfect for animation recording) or 'Static' (to change Solution steps manually with the mouse); update period also lets you change the speed of animation; Press Run / Stop button to start/stop interactive mode (updating of graphics)

– **input:**

`mainSystem`: the system used for animation

`solution`: solution dictionary previously loaded with `exudyn.utilities.LoadSolutionFile(...)`; will be played from first to last row; if `solution==''`, it tries to load the file `coordinatesSolutionFileName` as stored in `mbs.sys['simulationSettings']`, which are the `simulationSettings` of the previous simulation

`rowIncrement`: can be set larger than 1 in order to skip solution frames: e.g. `rowIncrement=10` visualizes every 10th row (frame)

`timeout`: in seconds is used between frames in order to limit the speed of animation; e.g. use `timeout=0.04` to achieve approximately 25 frames per second

`runOnStart`: immediately go into 'Run' mode

`runMode`: 0=continuous run, 1=one cycle, 2=static (use slider/mouse to vary time steps)

– **output:** updates current visualization state, renders the scene continuously (after pressing button 'Run')

– **example:**

```
#HERE, mbs must contain same model as solution stored in coordinatesSolution.txt  
#adjust autoFitScene, otherwise it may lead to unwanted fit to scene  
SC.visualizationSettings.general.autoFitScene = False  
from exudyn.interactive import SolutionViewer #import function  
sol = LoadSolutionFile('coordinatesSolution.txt') #load solution: adjust to your file  
name
```

```
SolutionViewer(mbs, sol)
```

For examples on `SolutionViewer` see Examples (Ex) and TestModels (TM):

- [`addPrismaticJoint.py`](#) (Ex), [`addRevoluteJoint.py`](#) (Ex), [`beltDriveReevingSystem.py`](#) (Ex),  
[`CMSexampleCourse.py`](#) (Ex), [`fourBarMechanism3D.py`](#) (Ex), ... , [`ANCFbeltDrive.py`](#) (TM), [`ANCFgeneralContactCircle.py`](#) (TM), [`ConvexContactTest.py`](#) (TM), ...

### 5.7.1 CLASS InteractiveDialog (in module interactive)

**class description:** create an interactive dialog, which allows to interact with simulations the dialog has a 'Run' button, which initiates the simulation and a 'Stop' button which stops/pauses simulation; 'Quit' closes the simulation model for examples, see `simulateInteractively.py` and `massSpringFrictionInteractive.py` use `__init__` method to setup this class with certain buttons, edit boxes and sliders

- **example:**

```
#the following example is only demonstrating the structure of dialogItems and plots
#dialogItems structure:
#general items:
#    'type' can be out of:
#        'label' (simple text),
#        'button' (button with callback function),
#        'radio' (a radio button with several alternative options),
#        'slider' (with an adjustable range to choose a value)
#    'grid': (row, col, colspan) specifies the row, column and (optionally) the span
#    of columns the item is placed at;
#        exception in 'radio', where grid is a list of (row, col) for every
#        choice
#    'options': text options, where 'L' means flush left, 'R' means flush right
#suboptions of 'label':
#    'text': a text to be drawn
#suboptions of 'button':
#    'text': a text to be drawn on button
#    'callFunction': function which is called on button-press
#suboptions of 'radio':
#    'textValueList': [('text1',0),('text2',1)] a list of texts with
#    according values
#    'value': default value (choice) of radio buttons
#    'variable': according variable in mbs.variables, which is set to
#    current radio button value
#suboptions of 'slider':
#    'range': (min, max) a tuple containing minimum and maximum value of
#    slider
#    'value': default value of slider
#    'steps': number of steps in slider
#    'variable': according variable in mbs.variables, which is set to
#    current slider value
#example:
dialogItems = [{"type": "label", "text": "Nonlinear oscillation simulator", "grid": :(0,0,2), "options": ["L"]},
    {"type": "button", "text": "test button", "callFunction": ButtonCall, "grid": :(1,0,2)},
    {"type": "radio", "textValueList": [("linear",0),("nonlinear",1)], "value": 0, "variable": "mode", "grid": [(2,0),(2,1)]},
    {"type": "label", "text": "excitation frequency (Hz):", "grid": :(5,0)},
    {"type": "slider", "range": (3*f1/800, 3*f1), "value": omegaInit/(2*pi)},
```

```

'steps':800, 'variable':'frequency', 'grid':(5,1)},
    {'type':'label', 'text':'damping:', 'grid':(6,0)},
    {'type':'slider', 'range': (0, 40), 'value':damper, 'steps':800, 'variable':'damping', 'grid':(6,1)},
    {'type':'label', 'text':'stiffness:', 'grid':(7,0)},
    {'type':'slider', 'range':(0, 10000), 'value':spring, 'steps':800, 'variable':'stiffness', 'grid':(7,1)}]
#plots structure:
plots={'nPoints':500,                                     #number of stored points in subplots (higher means slower drawing)
       'subplots':(2,1),                                #(rows, columns) arrangement of subplots (for every sensor)
       #sensors defines per subplot (sensor, coordinate), xlabel and ylabel; if coordinate=0, time is used:
       'sensors':[[[(sensPos,0),(sensPos,1),'time','mass position'],
                    [(sensFreq,0),(sensFreq,1),'time','excitation frequency']],
                  'limitsX':[(0,2),(-5,5)],      #x-range per subplot; if not provided, autoscale is applied
                  'limitsY':[(-5,5),(0,10)],   #y-range per subplot; if not provided, autoscale is applied
                  'fontSize':16,              #custom font size for figure
                  'subplots':False,           #if not specified, subplots are created; if False, all plots go into one window
                  'lineStyles':['r-','b-'],    #if not specified, uses default '-b', otherwise define list of line styles [string for matplotlib.pyplot.plot] per sensor
                  'sizeInches':(12,12)}       #specific x and y size of figure in inches (using 100 dpi)

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

```

– **classFunction**: initialize an InteractiveDialog

– **input**:

*mbs*: a multibody system to be simulated

*simulationSettings*: exudyn.SimulationSettings() according to user settings

*simulationFunction*: a function which is called before a simulation for the short period is started (e.g., assign special values, etc.)

*dialogItems*: a list of dictionaries, which describe the contents of the interactive items, where every dict has the structure 'type':[label, entry, button, slider, check] ... according to tkinter widgets, 'callFunction': a function to be called, if item is changed/button pressed, 'grid': (row,col) of item to be placed, 'rowSpan': number of rows to be used, 'columnSpan': number of columns to be used; for special item options see notes

*plots*: list of dictionaries to specify a sensor to be plotted live, see example

*period*: a simulation time span in seconds which is simulated with the simulationFunction in every iteration

- realtimeFactor*: if 1, the simulation is nearly performed in realtime (except for computation time); if > 1, it runs faster than realtime, if < 1, than it is slower
- userStartFunction*: a function F(flag) which is called every time after Run/Stop is pressed. The argument flag = False if button "Run" has been pressed, flag = True, if "Stop" has been pressed
- title*: title text for interactive dialog
- showTime*: shows current time in dialog
- fontSize*: adjust font size for all dialog items
- doTimeIntegration*: performs internal time integration with given parameters
- runOnStart*: immediately activate 'Run' button on start
- **notes:** detailed description of dialogItems and plots list/dictionary is given in commented the example below
- 

`def OnQuit (self, event= None)`

- **classFunction:** function called when pressing escape or closing dialog
- 

`def StartSimulation (self, event= None)`

- **classFunction:** function called on button 'Run'
- 

`def ProcessWidgetStates (self)`

- **classFunction:** assign current values of radio buttons and sliders to mbs.variables
- 

`def ContinuousRunFunction (self, event= None)`

- **classFunction:** function which is repeatedly called when button 'Run' is pressed
- 

`def InitializePlots (self)`

- **classFunction:** initialize figure and subplots for plots structure

---

```
def UpdatePlots (self)
```

- **classFunction:** update all subplots with current sensor values
- 

```
def InitializeSolver (self)
```

- **classFunction:** function to initialize solver for repeated calls
- 

```
def FinalizeSolver (self)
```

- **classFunction:** stop solver (finalize correctly)
- 

```
def RunSimulationPeriod (self)
```

- **classFunction:** function which performs short simulation for given period

For examples on InteractiveDialog see Examples (Ex) and TestModels (TM):

- [massSpringFrictionInteractive.py](#) (Ex), [nMassOscillatorInteractive.py](#) (Ex), [simulateInteractively.py](#) (Ex)

## 5.8 Module: kinematicTree

A library for preparation of minimum coordinates (kinematic tree) formulation. This library follows mostly the algorithms of Roy Featherstone, see <http://royfeatherstone.org/> His code is available in MATLAB as well as described in the Springer Handbook of Robotics [34]. The main formalisms are based on the 6x6 Plücker coordinate system as denoted by Featherstone.

Author: Johannes Gerstmayr

Date: 2021-06-22

```
def MassCOMinertia2T66 (mass, centerOfMass, inertia)
```

- **function description:** convert mass, COM and inertia into 6x6 inertia matrix
- **input:**

*mass:* scalar mass

*centerOfMass:* 3D vector (list/array)

*inertia*: 3x3 matrix (list of lists / 2D array) w.r.t. center of mass

- **output**: 6x6 numpy array for further use in minimum coordinates formulation
- 

```
def Inertia2T66 (inertia)
```

- **function description**: convert inertia as produced with RigidBodyInertia class into 6x6 inertia matrix (as used in KinematicTree, Featherstone / Handbook of robotics [34])
  - **output**: 6x6 numpy array for further use in minimum coordinates formulation
  - **notes**: within the 6x6 matrix, the inertia tensor is defined w.r.t. the center of mass, while RigidBodyInertia defines the inertia tensor w.r.t. the reference point; however, this function correctly transforms all quantities of inertia.
- 

```
def Inertia66toMassCOMinertia (inertia66)
```

- **function description**: convert 6x6 inertia matrix into mass, COM and inertia
  - **input**: 6x6 numpy array containing rigid body inertia according to Featherstone / Handbook of robotics [34]
  - **output**:
    - [mass, centerOfMass, inertia]
    - mass*: scalar mass
    - centerOfMass*: 3D vector (list/array)
    - inertia*: 3x3 matrix (list of lists / 2D array) w.r.t. center of mass
- 

```
def JointTransformMotionSubspace66 (jointType, q)
```

- **function description**: return 6x6 Plücker joint transformation matrix evaluated for scalar joint coordinate *q* and motion subspace ('free modes' in Table 2.6 in Handbook of robotics [34])
- 

```
def JointTransformMotionSubspace (jointType, q)
```

- **function description**: return list containing rotation matrix, translation vector, rotation axis and translation axis for joint transformation

---

```
def CRM(v)
```

- **function description:** computes cross product operator for motion from 6D vector v; CRM(v) @ m computes the cross product of v and motion m
- 

```
def CRF(v)
```

- **function description:** computes cross product operator for force from 6D vector v; CRF(v) @ f computes the cross product of v and force f

### 5.8.1 CLASS KinematicTree (in module kinematicTree)

**class description:** class to define a kinematic tree, which can be used for building serial or tree-structured multibody systems (or robots) with a minimum coordinates formulation, using efficient rotation matrices and 3D offsets

- **notes:** The formulation and structures widely follows the more efficient formulas with 3D vectors and rotation matrices as proposed in Handbook of robotics [34], Chapter 3, but with the rotation matrices (`listOfRotations`) being transposed in the Python implementation as compared to the description in the book, being thus compliant with other Exudyn functions; the 3D vector/matrix Python implementation does not offer advantages as compared to the formulation with Plücker coordinates, BUT it reflects the formulas of the C++ implementation and is used for testing

```
def __init__(self, listOfJointTypes, listOfRotations, listOfOffsets, listOfInertia3D, listOfCOM, listOfMass, listOfParents= [], gravity= [0,0,-9.81], )
```

- **classFunction:** initialize kinematic tree

- **input:**

*listOfJointTypes*: mandatory list of joint types 'Rx', 'Ry', 'Rz' denoting revolute joints; 'Px', 'Py', 'Pz', denoting prismatic joints

*listOfRotations*: per link rotation matrix, transforming coordinates of the joint coordinate system w.r.t. the previous coordinate system (this is the inverse of Plücker coordinate transforms (6x6))

*listOfOffsets*: per link offset vector from previous coordinate system to the joint coordinate system

*listOfInertia3D*: per link 3D inertia matrix, w.r.t. reference point (not COM!)

*listOfCOM*: per link vector from reference point to center of mass (COM), in link coordinates

*listOfMass*: mass per link

*listOfParents*: list of parent object indices (int), according to the index in jointTypes and transformations; use empty list for kinematic chain and use -1 if no parent exists (parent=base or world frame)

*gravity*: a 3D list/array containing the gravity applied to the kinematic tree (in world frame)

---

```
def Size (self)
```

- **classFunction:** return number of joints, defined by size of jointTypes
- 

```
def XL (self, i)
```

- **classFunction:** return [A, p] containing rotation matrix and offset for joint j
- 

```
def ForwardDynamicsCRB (self, q= [], q_t= [], torques= [], forces= [])
```

- **classFunction:** compute forward dynamics using composite rigid body algorithm

- **input:**

*q:* joint space coordinates for the model at which the forward dynamics is evaluated

*q\_t:* joint space velocity coordinates for the model at which the forward dynamics is evaluated

*torques:* a vector of torques applied at joint coordinates or list/array with zero length

*forces:* forces acting on the bodies using special format

- **output:** returns acceleration vector q\_tt of joint coordinates
- 

```
def ComputeMassMatrixAndForceTerms (self, q, q_t, externalForces= [])
```

- **classFunction:**

compute generalized mass matrix M and generalized force terms for kinematic tree, using current state (joint) variables q and joint velocities q\_t. The generalized force terms f = fGeneralized contain Coriolis and gravity if given in the kinematicTree.

- **input:**

*q:* current joint coordinates

*q\_t:* current joint velocities

*externalForces:* list of torque/forces in global (world) frame per joint; may be empty list, containing 6D vectors or matrices with 6D vectors in columns that are summed up for each link

- **output:** mass matrix **M** and RHS vector **f<sub>RHS</sub>** for equations of motion  $M(q) \cdot q_tt + f(q, q_t, externalForces) = \tau$ ; RHS is  $f_{RHS} = \tau - f(q, q_t, externalForces)$ ;  $\tau$  can be added outside of **ComputeMassMatrixAndForceTerms**

## 5.8.2 CLASS KinematicTree66 (in module kinematicTree)

**class description:** class to define a kinematic tree, which can be used for building serial or tree-structured multibody systems (or robots) with a minimum coordinates formulation, using Plücker coordinate transforms (6x6)

- **notes:** The formulation and structures widely follow Roy Featherstone (<http://royfeatherstone.org/>) / Handbook of robotics [34]

```
def __init__(self, listOfJointTypes, listOfTransformations, listOfInertias, listOfParents= [], gravity= [0,0,-9.81], )
```

- **classFunction:** initialize kinematic tree
- **input:**

*listOfJointTypes*: mandatory list of joint types 'Rx', 'Ry', 'Rz' denoting revolute joints; 'Px', 'Py', 'Pz', denoting prismatic joints

*listOfTransformations*: provide a list of Plücker coordinate transforms (6x6 numpy matrices), describing the (constant) link transformation from the link coordinate system (previous/parent joint) to this joint coordinate system

*listOfInertias*: provide a list of inertias as (6x6 numpy matrices), as produced by the function Mass-COMInertia2T66

*listOfParents*: list of parent object indices (int), according to the index in jointTypes and transformations; use empty list for kinematic chain and use -1 if no parent exists (parent=base or world frame)

*gravity*: a 3D list/array containing the gravity applied to the kinematic tree (in world frame)

---

```
def Size(self)
```

- **classFunction:** return number of joints, defined by size of jointTypes
- 

---

```
def XL(self, i)
```

- **classFunction:** return 6D transformation of joint i, given by transformation
- 

---

```
def ForwardDynamicsCRB(self, q= [], q_t= [], torques= [], forces= [])
```

- **classFunction:** compute forward dynamics using composite rigid body algorithm
- **input:**

*q*: joint space coordinates for the model at which the forward dynamics is evaluated

$q_t$ : joint space velocity coordinates for the model at which the forward dynamics is evaluated

*torques*: a vector of torques applied at joint coordinates or list/array with zero length

*forces*: forces acting on the bodies using special format

- **output**: returns acceleration vector  $q_{tt}$  of joint coordinates
- 

```
def ComputeMassMatrixAndForceTerms (self, q, q_t, externalForces= [])
```

- **classFunction**:

compute generalized mass matrix  $M$  and generalized force terms for kinematic tree, using current state (joint) variables  $q$  and joint velocities  $q_t$ . The generalized force terms  $f = f_{Generalized}$  contain Coriolis and gravity if given in the kinematicTree.

- **input**:

$q$ : current joint coordinates

$q_t$ : current joint velocities

*externalForces*: list of torque/forces in global (world) frame per joint; may be empty list, containing 6D vectors or matrices with 6D vectors in columns that are summed up for each link

- **output**: mass matrix  $M$  and RHS vector  $\mathbf{f}_{RHS}$  for equations of motion  $M(q) \cdot q_{tt} + f(q, q_t, externalForces) = \tau$ ; RHS is  $\mathbf{f}_{RHS} = \tau - f(q, q_t, externalForces)$ ;  $\tau$  can be added outside of `ComputeMassMatrixAndForceTerms`
- 

```
def AddExternalForces (self, Xup, fvp, externalForces= [])
```

- **classFunction**: add action of external forces to forces  $fvp$  and return new composed vector of forces  $fvp$

- **input**:

$Xup$ : 6x6 transformation matrices per joint; as computed in `ComputeMassMatrixAndForceTerms`

$fvp$ : force (torque) per joint, as computed in `ComputeMassMatrixAndForceTerms`

*externalForces*: list of torque/forces in global (world) frame per joint; may be empty list, containing 6D vectors or matrices with 6D vectors in columns that are summed up for each link

## 5.9 Module: lieGroupBasics

Lie group methods and formulas for Lie group integration.

Author: Stefan Holzinger

Date: 2020-09-11

References:

For details on Lie group methods used here, see the references [22, 35, 5, 37, 36, 38, 29]. Lie group methods for rotation vector are described in Holzinger and Gerstmayr [16, 23].

def Sinc ( $x$ )

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

def Cot ( $x$ )

- **function description:** compute the cotangent function  $\cot(x)=1/\tan(x)$  in radians
  - **input:** scalar float or int value
  - **output:** float value in radians
- 

def R3xSO3Matrix2RotationMatrix ( $G$ )

- **function description:** computes  $3 \times 3$  rotation matrix from  $7 \times 7$   $R3xSO(3)$  matrix, see [5]
  - **input:**  $G$ :  $7 \times 7$  matrix as np.array
  - **output:**  $3 \times 3$  rotation matrix as np.array
- 

def R3xSO3Matrix2Translation ( $G$ )

- **function description:** computes translation part of  $R3xSO(3)$  matrix, see [5]
  - **input:**  $G$ :  $7 \times 7$  matrix as np.array
  - **output:** 3D vector as np.array containg translational part of  $R3xSO(3)$
- 

def R3xSO3Matrix ( $x, R$ )

- **function description:** builds  $7 \times 7$  matrix as element of the Lie group  $R3xSO(3)$ , see [5]
- **input:**
  - $x$ : 3D vector as np.array representing the translation part corresponding to  $R3$
  - $R$ :  $3 \times 3$  rotation matrix as np.array

- 
- **output:** 7x7 matrix as np.array

---

```
def ExpSO3 (Omega)
```

- **function description:** compute the matrix exponential map on the Lie group SO(3), see [29]
  - **input:** 3D rotation vector as np.array
  - **output:** 3x3 matrix as np.array
- 

---

```
def ExpS3 (Omega)
```

- **function description:** compute the quaternion exponential map on the Lie group S(3), see [38, 29]
  - **input:** 3D rotation vector as np.array
  - **output:**
    - 4D vector as np.array containing four Euler parameters
    - entry zero of output represent the scalar part of Euler parameters
- 

---

```
def LogSO3 (R)
```

- **function description:** compute the matrix logarithmic map on the Lie group SO(3), see [37, 36]
  - **input:** 3x3 rotation matrix as np.array
  - **output:** 3x3 skew symmetric matrix as np.array
- 

---

```
def TExpSO3 (Omega)
```

- **function description:** compute the tangent operator corresponding to ExpSO3, see [5]
  - **input:** 3D rotation vector as np.array
  - **output:** 3x3 matrix as np.array
- 

---

```
def TExpSO3Inv (Omega)
```

- **function description:**  
compute the inverse of the tangent operator TExpSO3, see [37]  
this function was improved, see coordinateMaps.pdf by Stefan Holzinger
  - **input:** 3D rotation vector as np.array
  - **output:** 3x3 matrix as np.array
- 

```
def ExpSE3 (x)
```

- **function description:** compute the matrix exponential map on the Lie group SE(3), see [5]
  - **input:** 6D incremental motion vector as np.array
  - **output:** 4x4 homogeneous transformation matrix as np.array
- 

```
def LogSE3 (H)
```

- **function description:** compute the matrix logarithm on the Lie group SE(3), see [37]
  - **input:** 4x4 homogeneous transformation matrix as np.array
  - **output:** 4x4 skew symmetric matrix as np.array
- 

```
def TExpSE3 (x)
```

- **function description:** compute the tangent operator corresponding to ExpSE3, see [5]
  - **input:** 6D incremental motion vector as np.array
  - **output:** 6x6 matrix as np.array
- 

```
def TExpSE3Inv (x)
```

- **function description:** compute the inverse of tangent operator TExpSE3, see [37]
  - **input:** 6D incremental motion vector as np.array
  - **output:** 6x6 matrix as np.array
-

```
def ExpR3xSO3 ( $x$ )
```

- **function description:** compute the matrix exponential map on the Lie group R3xSO(3), see [5]
  - **input:** 6D incremental motion vector as np.array
  - **output:** 7x7 matrix as np.array
- 

```
def TExpR3xSO3 ( $x$ )
```

- **function description:** compute the tangent operator corresponding to ExpR3xSO3, see [5]
  - **input:** 6D incremental motion vector as np.array
  - **output:** 6x6 matrix as np.array
- 

```
def TExpR3xSO3Inv ( $x$ )
```

- **function description:** compute the inverse of tangent operator TExpR3xSO3
  - **input:** 6D incremental motion vector as np.array
  - **output:** 6x6 matrix as np.array
- 

```
def CompositionRuleDirectProductR3AndS3 ( $q_0, incrementalMotionVector$ )
```

- **function description:** compute composition operation for pairs in the Lie group R3xS3
  - **input:**
    - $q_0$ : 7D vector as np.array containing position coordinates and Euler parameters
    - $incrementalMotionVector$ : 6D incremental motion vector as np.array
  - **output:** 7D vector as np.array containing composed position coordinates and composed Euler parameters
- 

```
def CompositionRuleSemiDirectProductR3AndS3 ( $q_0, incrementalMotionVector$ )
```

- **function description:** compute composition operation for pairs in the Lie group R3 semiTimes S3 (corresponds to SE(3))
- **input:**
  - $q_0$ : 7D vector as np.array containing position coordinates and Euler parameters
  - $incrementalMotionVector$ : 6D incremental motion vector as np.array

- **output:** 7D vector as np.array containing composed position coordinates and composed Euler parameters
- 

```
def CompositionRuleDirectProductR3AndR3RotVec (q0, incrementalMotionVector)
```

- **function description:**

compute composition operation for pairs in the group obtained from the direct product of R3 and R3,  
see [16]

the rotation vector is used as rotation parametrizations

this composition operation can be used in formulations which represent the translational velocities in  
the global (inertial) frame

- **input:**

*q0*: 6D vector as np.array containing position coordinates and rotation vector

*incrementalMotionVector*: 6D incremental motion vector as np.array

- **output:** 7D vector as np.array containing composed position coordinates and composed rotation vector
- 

```
def CompositionRuleSemiDirectProductR3AndR3RotVec (q0, incrementalMotionVector)
```

- **function description:**

compute composition operation for pairs in the group obtained from the direct product of R3 and R3.

the rotation vector is used as rotation parametrizations

this composition operation can be used in formulations which represent the translational velocities in  
the local (body-attached) frame

- **input:**

*q0*: 6D vector as np.array containing position coordinates and rotation vector

*incrementalMotionVector*: 6D incremental motion vector as np.array

- **output:** 6D vector as np.array containing composed position coordinates and composed rotation vector
- 

```
def CompositionRuleDirectProductR3AndR3RotXYZAngles (q0, incrementalMotionVector)
```

- **function description:**

compute composition operation for pairs in the group obtained from the direct product of R3 and R3.

Cardan-Tait/Bryan (CTB) angles are used as rotation parametrizations

this composition operation can be used in formulations which represent the translational velocities in  
the global (inertial) frame

– **input:**

$q0$ : 6D vector as np.array containing position coordinates and Cardan-Tait/Bryan angles  
 $incrementalMotionVector$ : 6D incremental motion vector as np.array

– **output:** 6D vector as np.array containing composed position coordinates and composed Cardan-Tait/Bryan angles

---

def **CompositionRuleSemiDirectProductR3AndR3RotXYZAngles** ( $q0, incrementalMotionVector$ )

– **function description:**

compute composition operation for pairs in the group obtained from the direct product of R3 and R3.  
Cardan-Tait/Bryan (CTB) angles are used as rotation parametrizations  
this composition operation can be used in formulations which represent the translational velocities in  
the local (body-attached) frame

– **input:**

$q0$ : 6D vector as np.array containing position coordinates and Cardan-Tait/Bryan angles  
 $incrementalMotionVector$ : 6D incremental motion vector as np.array

– **output:** 6D vector as np.array containing composed position coordinates and composed Cardan-Tait/Bryan angles

---

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

– **function description:**

compute composition operation for Euler parameters (unit quaternions)  
this composition operation is quaternion multiplication, see [38]

– **input:**

$q$ : 4D vector as np.array containing Euler parameters  
 $p$ : 4D vector as np.array containing Euler parameters

– **output:** 4D vector as np.array containing composed (multiplied) Euler parameters

---

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

– **function description:** compute composition operation for rotation vectors  $v0$  and  $\Omega$ , see [23]

– **input:**

$v0$ : 3D rotation vector as np.array

*Omega*: 3D (incremental) rotation vector as np.array

- **output**: 3D vector as np.array containing composed rotation vector v
- 

```
def CompositionRuleRotXYZAnglesRotationVector (alpha0, Omega)
```

- **function description**: compute composition operation for RotXYZ angles, see [23]

- **input**:

*alpha0*: 3D vector as np.array containing RotXYZ angles

*Omega*: 3D vector as np.array containing the (incremental) rotation vector

- **output**: 3D vector as np.array containing composed RotXYZ angles

## 5.10 Module: physics

The physics library includes helper functions and data related to physics models and parameters; for rigid body inertia, see rigidBodyUtilities

Date: 2021-01-20

```
def StribeckFunction (vel, muDynamic, muStaticOffset, muViscous= 0, expVel= 1e-3, regVel= 1e-3)
```

- **function description**:

describes regularized Stribeck function with optial viscous part for given velocity,

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

- **input**:

*vel*: input velocity *v*

*muDynamic*: dynamic friction coefficient  $\mu_d$

*muStaticOffset*:  $\mu_{s_{off}}$ , offset to dynamic friction, which gives *muStaticFriction* = *muDynamic* + *muStaticOffset*

*muViscous*:  $\mu_v$ , viscous part, acting proportional to velocity except for *regVel*

*regVel*:  $v_{reg}$ , small regularization velocity in which the friction is linear around zero velocity (e.g., to get Newton converged)

*expVel*:  $v_{exp}$ , velocity (relative to *regVel*, at which the *muStaticOffset* decreases exponentially, at *vel=expVel*, the factor to *muStaticOffset* is  $e^{-1} = 36.8\%$ )

- **output**: returns velocity dependent friction coefficient (if *muDynamic* and *muStaticOffset* are friction coefficients) or friction force (if *muDynamic* and *muStaticOffset* are on force level)

For examples on StribeckFunction see Examples (Ex) and TestModels (TM):

- [massSpringFrictionInteractive.py](#) (Ex)

---

```
def RegularizedFrictionStep (x, x0, h0, x1, h1)
```

- **function description:** helper function for RegularizedFriction(...)
- 

```
def RegularizedFriction (vel, muDynamic, muStaticOffset, velStatic, velDynamic, muViscous= 0)
```

- **function description:** describes regularized friction function, with increased static friction, dynamic friction and optional viscous part
- **input:**
  - vel*: input velocity
  - muDynamic*: dynamic friction coefficient
  - muStaticOffset*: offset to dynamic friction, which gives muStaticFriction = muDynamic + muStaticOffset
  - muViscous*: viscous part, acting proportional to velocity for velocities larger than velDynamic; extension to mentioned references
  - velStatic*: small regularization velocity at which exactly the staticFriction is reached; for smaller velocities, the friction is smooth and zero-crossing (unphysical!) (e.g., to get Newton converged)
  - velDynamic*: velocity at which muDynamic is reached for first time
- **output:** returns velocity dependent friction coefficient (if muDynamic and muStaticOffset are friction coefficients) or friction force (if muDynamic and muStaticOffset are on force level)
- **notes:** see references: Flores et al. [11], Qian et al. [32]

For examples on RegularizedFriction see Examples (Ex) and TestModels (TM):

- [massSpringFrictionInteractive.py](#) (Ex)
- 

```
def VonMisesStress (stress6D)
```

- **function description:** compute equivalent von-Mises stress given 6 stress components or list of stress6D (or stress6D in rows of np.array)
  - **input:** stress6D: 6 stress components as list or np.array, using ordering  $[\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{yz}, \sigma_{xz}, \sigma_{xy}]$
  - **output:** returns scalar equivalent von-Mises stress or np.array of von-Mises stresses for all stress6D
- 

```
def UFvonMisesStress (mbs, t, sensorNumbers, factors, configuration)
```

- **function description:** Sensor user function to compute equivalent von-Mises stress from sensor with Stress or StressLocal OutputVariableType; if more than 1 sensor is given in sensorNumbers, then the maximum stress is computed
- **input:** arguments according to SensorUserFunction; factors are ignored
- **output:** returns scalar (maximum) equivalent von-Mises stress
- **example:**

```
#assuming s0, s1, s2 being sensor numbers with StressLocal components
sUser = mbs.AddSensor(SensorUserFunction(sensorNumbers=[s0,s1,s2],
                                         fileName='solution/sensorMisesStress.txt',
                                         sensorUserFunction=UFvonMisesStress))
```

## 5.11 Module: plot

Plot utility functions based on matplotlib, including plotting of sensors and FFT.

Author: Johannes Gerstmayr

Date: 2020-09-16 (created)

Notes: For a list of plot colors useful for matplotlib, see also utilities.PlotLineCode(...)

def [ParseOutputFileHeader](#) (*lines*)

- **function description:** parse header of output file (solution file, sensor file, genetic optimization output, ...) given in file.readlines() format
  - **output:** return dictionary with 'type'=['sensor','solution','geneticOptimization','parameterVariation'], 'variableType',
- 

def [PlotSensorDefaults](#) ()

- **function description:** returns structure with default values for PlotSensor which can be modified once to be set for all later calls of PlotSensor
- **example:**

```
#change one parameter:
plot.PlotSensorDefaults().fontSize = 12
#==>now PlotSensor(...) will use fontSize=12
#==>now PlotSensor(..., fontSize=10) will use fontSize=10
#==>BUT PlotSensor(..., fontSize=16) will use fontSize=12, BECAUSE 16 is the original
      default value!!!
#see which parameters are available:
print(PlotSensorDefaults())
```

For examples on PlotSensorDefaults see Examples (Ex) and TestModels (TM):

- [coordinateVectorConstraintGenericODE2.py](#) (TM)

---

```
def PlotSensor (mbs, sensorNumbers= [], components= 0, xLabel= 'time (s)', yLabel= None, labels= [],
colorCodeOffset= 0, newFigure= True, closeAll= False, componentsX= [], title= "", figureName= "", fontSize= 16,
colors= [], lineStyles= [], lineWidths= [], markerStyles= [], markerSizes= [], markerDensity= 0.08, rangeX= [],
rangeY= [], majorTicksX= 10, majorTicksY= 10, offsets= [], factors= [], subPlot= [], sizeInches= [6.4,4.8], fileName=
", useXYZcomponents= True, **kwargs)
```

- **function description:** Helper function for direct and easy visualization of sensor outputs, without need for loading text files, etc.; PlotSensor can be used to simply plot, e.g., the measured x-Position over time in a figure. PlotSensor provides an interface to matplotlib (which needs to be installed). Default values of many function arguments can be changed using the exudyn.plot function PlotSensorDefaults(), see there for usage.

- **input:**

*mbs*: must be a valid MainSystem (*mbs*)

*sensorNumbers*: consists of one or a list of sensor numbers (type SensorIndex or int) as returned by the mbs function AddSensor(...); sensors need to set writeToFile=True and/or storeInternal=True for PlotSensor to work; alternatively, it may contain FILENAMES (incl. path) to stored sensor or solution files OR a numpy array instead of sensor numbers; the format of data (file or numpy array) must contain per row the time and according solution values in columns; if components is a list and sensorNumbers is a scalar, sensorNumbers is adjusted automatically to the components

*components*: consists of one or a list of components according to the component of the sensor to be plotted at y-axis; if components is a list and sensorNumbers is a scalar, sensorNumbers is adjusted automatically to the components; as always, components are zero-based, meaning 0=X, 1=Y, etc.

*componentsX*: default componentsX=[] uses time in files; otherwise provide componentsX as list of components (or scalar) representing x components of sensors in plotted curves; DON'T forget to change xLabel accordingly! Using componentsX=[...] with a list of column indices specifies the respective columns used for the x-coordinates in all sensors; by default, values are plotted against the first column in the files, which is time; according to counting in PlotSensor, this represents componentX=-1; plotting y over x in a position sensor thus reads: components=[1], componentsX=[0]; plotting time over x reads: components=[-1], componentsX=[0]; the default value reads componentsX=[-1,-1,...]

*xLabel*: string for text at x-axis

*yLabel*: string for text at y-axis (default: None==> label is automatically computed from sensor value types)

*labels*: string (for one sensor) or list of strings (according to number of sensors resp. components) representing the labels used in legend; if labels=[], automatically generated legend is used

*rangeX*: default rangeX=[]: computes range automatically; otherwise use rangeX to set range (limits) for x-axis provided as sorted list of two floats, e.g., rangeX=[0,4]

*rangeY*: default rangeY=[]: computes range automatically; otherwise use rangeY to set range (limits) for y-axis provided as sorted list of two floats, e.g., rangeY=[-1,1]

*figureName*: optional name for figure, if newFigure=True

*fontSize*: change general fontsize of axis, labels, etc. (matplotlib default is 12, default in PlotSensor: 16)

*title*: optional string representing plot title

*offsets*: provide as scalar, list of scalars (per sensor) or list of 2D numpy.arrays (per sensor, having same rows/columns as sensor data) to add offset to each sensor output; for an original value fOrig, the new value reads fNew = factor\*(fOrig+offset); for offset provided as numpy array (with same time values), the 'time' column is ignored in the offset computation; can be used to compute difference of sensors; if offsets=[], no offset is used

*factors*: provide as scalar or list (per sensor) to add factor to each sensor output; for an original value fOrig, the new value reads fNew = factor\*(fOrig+offset); if factor=[], no factor is used

*majorTicksX*: number of major ticks on x-axis; default: 10

*majorTicksY*: number of major ticks on y-axis; default: 10

*colorCodeOffset*: int offset for color code, color codes going from 0 to 27 (see PlotLineCode(...)); automatic line/color codes are used if no colors and lineStyles are used

*colors*: color is automatically selected from colorCodeOffset if colors=[]; otherwise chose from 'b', 'g', 'r', 'c', 'm', 'y', 'k' and many other colors see [https://matplotlib.org/stable/gallery/color/named\\_colors.html](https://matplotlib.org/stable/gallery/color/named_colors.html)

*lineStyles*: line style is automatically selected from colorCodeOffset if lineStyles=[]; otherwise define for all lines with string or with list of strings, choosing from '-', '--', '-.', ':', or ''

*lineWidths*: float to define line width by float (default=1); either use single float for all sensors or list of floats with length >= number of sensors

*markerStyles*: if different from [], marker styles are defined as list of marker style strings or single string for one sensor; chose from '.', 'o', 'x', '+', '\*', 'd', 'D', 's', 'v', '^', '<', '>', 'p', 'h', 'H', 'X', 'P', and many others, see [https://matplotlib.org/stable/api/markers\\_api.html](https://matplotlib.org/stable/api/markers_api.html); ADD a space to markers to make them empty (transparent), e.g. 'o' will create an empty circle

*markerSizes*: float to define marker size by float (default=6); either use single float for all sensors or list of floats with length >= number of sensors

*markerDensity*: if int, it defines approx. the total number of markers used along each graph; if float, this defines the distance of markers relative to the diagonal of the plot (default=0.08); if None, it adds a marker to every data point if marker style is specified for sensor

*newFigure*: if True, a new matplotlib.pyplot figure is created; otherwise, existing figures are overwritten

*subPlot*: given as list [nx, ny, position] with nx, ny being the number of subplots in x and y direction (nx=cols, ny=rows), and position in [1, ..., nx\*ny] gives the position in the subplots; use the same structure for first PlotSensor (with newFigure=True) and all subsequent PlotSensor calls with newFigure=False, which creates the according subplots; default=[](no subplots)

*sizeInches*: given as list [sizeX, sizeY] with the sizes per (sub)plot given in inches; default: [6.4, 4.8]; in case of sub plots, the total size of the figure is computed from nx\*sizeInches[0] and ny\*sizeInches[1]

*fileName*: if this string is non-empty, figure will be saved to given path and filename (use figName.pdf to save as PDF or figName.png to save as PNG image); use matplotlib.use('Agg') in order not to open figures if you just want to save them

*useXYZcomponents*: of True, it will use X, Y and Z for sensor components, e.g., measuring Position, Velocity, etc. wherever possible

*closeAll*: if True, close all figures before opening new one (do this only in first PlotSensor command!)

*[\*kwargs]*:

*minorTicksXon*: if True, turn minor ticks for x-axis on

*minorTicksYon*: if True, turn minor ticks for y-axis on

*fileCommentChar*: if exists, defines the comment character in files (#,  
*fileDelimiterChar*: if exists, defines the character indicating the columns for data (',', ',', ';', ...)

- **output**: plots the sensor data
- **notes**: adjust default values by modifying the variables exudyn.plot.plotSensorDefault..., e.g., exudyn.plot.plotSensorDefaultFontSize
- **example**:

```
#assume to have some position-based nodes 0 and 1:
s0=mbs.AddSensor(SensorNode(nodeNumber=0, fileName='s0.txt',
                             outputVariableType=exu.OutputVariableType.Position))
s1=mbs.AddSensor(SensorNode(nodeNumber=1, fileName='s1.txt',
                             outputVariableType=exu.OutputVariableType.Position))

PlotSensor(mbs, s0, 0) #plot x-coordinate
#plot x for s0 and z for s1:
PlotSensor(mbs, sensorNumbers=[s0,s1], components=[0,2], xLabel='time in seconds')
PlotSensor(mbs, sensorNumbers=s0, components=[0,1,2], factors=1000., title='Answers
to the big questions')
PlotSensor(mbs, sensorNumbers=s0, components=[0,1,2,3],
           yLabel='Coordantes with offset 1\nand scaled with $\\frac{1}{1000}$',
           factors=1e-3, offsets=1, fontSize=12, closeAll=True)
#assume to have body sensor sBody, marker sensor sMarker:
PlotSensor(mbs, sensorNumbers=[sBody]*3+[sMarker]*3, components=[0,1,2,0,1,2],
           colorCodeOffset=3, newFigure=False, fontSize=10,
           yLabel='Rotation $\\alpha, \\beta, \\gamma$ and\n Position $x,y,z$',
           title='compare marker and body sensor')
#assume having file plotSensorNode.txt:
PlotSensor(mbs, sensorNumbers=[s0]*3+ [filedir+'plotSensorNode.txt']*3,
           components=[0,1,2]*2)
#plot y over x:
PlotSensor(mbs, sensorNumbers=s0, componentsX=[0], components=[1], xLabel='x-Position
', yLabel='y-Position')
#for further examples, see also Examples/plotSensorExamples.py
```

For examples on PlotSensor see Examples (Ex) and TestModels (TM):

- [ANCFALEtest.py](#) (Ex), [beltDriveReevingSystem.py](#) (Ex), [bicycleIftommBenchmark.py](#) (Ex),  
[CMSEXampleCourse.py](#) (Ex), [finiteSegmentMethod.py](#) (Ex), ... , [ANCFbeltDrive.py](#) (TM), [ANCFgeneralContactCircle.py](#) (TM), [ANCFoutputTest.py](#) (TM), ...
- 

```
def PlotFFT (frequency, data, xLabel= 'frequency', yLabel= 'magnitude', label= "", freqStart= 0, freqEnd= -1,
logScaleX= True, logScaleY= True, majorGrid= True, minorGrid= True)
```

- **function description**: plot fft spectrum of signal
- **input**:
  - frequency*: frequency vector (Hz, if time is in SECONDS)
  - data*: magnitude or phase as returned by ComputeFFT() in exudyn.signalProcessing

*xLabel*: label for x-axis, default=frequency  
*yLabel*: label for y-axis, default=magnitude  
*label*: either empty string ("") or name used in legend  
*freqStart*: starting range for frequency  
*freqEnd*: end of range for frequency; if freqEnd== -1 (default), the total range is plotted  
*logScaleX*: use log scale for x-axis  
*logScaleY*: use log scale for y-axis  
*majorGrid*: if True, plot major grid with solid line  
*minorGrid*: if True, plot minor grid with dotted line

- **output:** creates plot and returns plot (plt) handle

---

```
def FileStripSpaces (filename, outputFilename, fileCommentChar= "", removeDoubleChars= "")
```

- **function description:** strip spaces at beginning / end of lines; this may be sometimes necessary when reading solutions from files that are space-separated
- **input:**
  - filename*: name of file to process
  - outputFilename*: name of file to which text without leading/trailing spaces is written
  - fileCommentChar*: if not equal "", lines starting with this character will not be processed
  - removeDoubleChars*: if not equal "", this double characters (especial multiple spaces) will be removed; '1.0 3.0' will be converted into '1.0 3.0'
- **output:** new file written

---

```
def DataArrayFromSensorList (mbs, sensorNumbers, positionList= [], time= "")
```

- **function description:** helper function to create data array from outputs defined by sensorNumbers list [+optional positionList which must have, e.g., local arc-length of beam according to sensor numbers]; if time== "", current sensor values will be used; if time!=[], evaluation will be based on loading values from file or sensor internal data and evaluate at that time
- **input:**
  - mbs*: a MainSystem where the sensors are given
  - sensorNumbers*: a list of sensor numbers, which shall be evaluated
  - positionList*: an optional list of positions per sensor (e.g., axial positions at beam)
  - time*: optional time at which the sensor values are evaluated (currently not implemented)
- **output:** returns data as numpy array, containing per row the number or position (positionList) in the first column and all sensor values in the remaining columns

## 5.12 Module: processing

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

Author: Johannes Gerstmayr, Stefan Holzinger

Date: 2020-11-17 (2022-02-04 modified by Stefan Holzinger)

Notes: Parallel processing, which requires multiprocessing library, can lead to considerable speedup (measured speedup factor > 50 on 80 core machine). The progress bar during multiprocessing requires the library tqdm.

```
def ProcessParameterList (parameterFunction, parameterList, addComputationIndex, useMultiProcessing, **kwargs)
```

– **function description:** processes parameterFunction for given parameters in parameterList, see ParameterVariation

– **input:**

*parameterFunction*: function, which takes the form parameterFunction(parameterDict) and which returns any values that can be stored in a list (e.g., a floating point number)

*parameterList*: list of parameter sets (as dictionaries) which are fed into the parameter variation, e.g.,  
[{'mass': 10, 'mass':20, ...}]

*addComputationIndex*: if True, key 'computationIndex' is added to every parameterDict in the call to parameterFunction(), which allows to generate independent output files for every parameter, etc.

*useMultiProcessing*: if True, the multiprocessing lib is used for parallelized computation; WARNING: be aware that the function does not check if your function runs independently; DO NOT use GRAPHICS and DO NOT write to same output files, etc.!

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

*resultsFile*: if provided, output is immediately written to resultsFile during processing

– **output:** returns values containing the results according to parameterList

– **notes:** options are passed from ParameterVariation

---

```
def ParameterVariation (parameterFunction, parameters, useLogSpace= False, debugMode= False,  
addComputationIndex= False, useMultiProcessing= False, showProgress= True, parameterFunctionData= , **kwargs)
```

– **function description:**

calls successively the function parameterFunction(parameterDict) with variation of parameters in given range; parameterDict is a dictionary, containing the current values of parameters,

e.g., *parameterDict*=[{'mass':13, 'stiffness':12000} to be computed and returns a value or a list of values which is then stored for each parameter

– **input:**

*parameterFunction*: function, which takes the form parameterFunction(parameterDict) and which returns any values that can be stored in a list (e.g., a floating point number)

*parameters*: given as a dictionary, consist of name and tuple of (begin, end, numberOfValues) same as in np.linspace(...), e.g. 'mass':(10,50,10), for a mass varied from 10 to 50, using 10 steps OR a list of values [v0, v1, v2, ...], e.g. 'mass':[10,15,25,50]

*useLogSpace*: (optional) if True, the parameters are varied at a logarithmic scale, e.g., [1, 10, 100] instead linear [1, 50.5, 100]

*debugMode*: if True, additional print out is done

*addComputationIndex*: if True, key 'computationIndex' is added to every parameterDict in the call to parameterFunction(), which allows to generate independent output files for every parameter, etc.

*useMultiProcessing*: if True, the multiprocessing lib is used for parallelized computation; WARNING: be aware that the function does not check if your function runs independently; DO NOT use GRAPHICS and DO NOT write to same output files, etc.!

*showProgress*: if True, shows for every iteration the progress bar (requires tqdm library)

*resultsFile*: if provided, output is immediately written to resultsFile during processing

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

*parameterFunctionData*: dictionary containing additional data passed to the parameterFunction inside the parameters with dict key 'functionData'; use this e.g. for passing solver parameters or other settings

- **output:**

*returns* [parameterList, values], containing, e.g., parameterList='mass':[1,1,1,2,2,3,3,3], 'stiffness':[4,5,6,4,5,6] and the result values of the parameter variation accoring to the parameterList, values=[7,8,9 ,3,4,5, 6,7,8] (depends on solution of problem ..., can also contain tuples, etc.)

- **example:**

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

For examples on ParameterVariation see Examples (Ex) and TestModels (TM):

- [parameterVariationExample.py](#) (Ex), [geneticOptimizationTest.py](#) (TM)
- 

```
def GeneticOptimization(objectiveFunction, parameters, populationSize= 100, numberOfWorkers= 10,
elitistRatio= 0.1, crossoverProbability= 0.25, crossoverAmount= 0.5, rangeReductionFactor= 0.7, distanceFactor= 0.1,
childDistribution= "uniform", distanceFactorGenerations= -1, debugMode= False, addComputationIndex= False,
useMultiProcessing= False, showProgress= True, **kwargs)
```

- **function description:** compute minimum of given objectiveFunction

- **input:**

*objectiveFunction*: function, which takes the form parameterFunction(parameterDict) and which returns a value or list (or numpy array) which reflects the size of the objective to be minimized

*parameters*: given as a dictionary, consist of name and tuple containing the search range for this parameter (begin, end), e.g. 'mass':(10,50)

*populationSize*: individuals in every generation

*initialPopulationSize*: number of random initial individuals; default: population size

*numberOfGenerations*: number of generations; NOTE: it is required that elitistRatio\*populationSize >= 1  
*elitistRatio*: the number of surviving individuals in every generation is equal to the previous population times the elitistRatio  
*crossoverProbability*: if > 0: children are generated from two (randomly selected) parents by gene-crossover; if 0, no crossover is used  
*crossoverAmount*: if crossoverProbability > 0, then this amount is the probability of genes to cross; 0.1: small amount of genes cross, 0.5: 50% of genes cross  
*rangeReductionFactor*: reduction of mutation range (boundary) relative to range of last generation; helps algorithm to converge to more accurate values  
*distanceFactor*: children only survive at a certain relative distance of the current range; must be small enough (< 0.5) to allow individuals to survive; ignored if distanceFactor=0; as a rule of thumb, the distanceFactor should be zero in case that there is only one significant minimum, but if there are many local minima, the distanceFactor should be used to search at several different local minima  
*childDistribution*: string with name of distribution for producing childs: "normal" (Gaussian, with sigma defining range), "uniform" (exactly in range of childs)  
*distanceFactorGenerations*: number of generations (populations) at which the distance factor is active; the distance factor is used to find several local minima; finally, convergence is speed up without the distance factor  
*randomizerInitialization*: initialize randomizer at beginning of optimization in order to get reproducible results, provide any integer in the range between 0 and  $2^{32} - 1$  (default: no initialization)  
*debugMode*: if True, additional print out is done  
*addComputationIndex*: if True, key 'computationIndex' is added to every parameterDict in the call to parameterFunction(), which allows to generate independent output files for every parameter, etc.  
*useMultiProcessing*: if True, the multiprocessing lib is used for parallelized computation; WARNING: be aware that the function does not check if your function runs independently; DO NOT use GRAPHICS and DO NOT write to same output files, etc.!  
*showProgress*: if True, shows for every iteration the progress bar (requires tqdm library)  
*numberOfThreads*: default: same as number of cpus (threads); used for multiprocessing lib;  
*resultsFile*: if provided, the results are stored columnwise into the given file and written after every generation; use resultsMonitor.py to track results in realtime  
*numberOfChildren*: (DEPRECATED, UNUSED) number childrens of surviving population  
*survivingIndividuals*: (DEPRECATED) number of surviving individuals after children are born

– **output:**

returns [optimumParameter, optimumValue, parameterList, valueList], containing the optimum parameter set 'optimumParameter', optimum value 'optimumValue', the whole list of parameters parameterList with according objective values 'valueList'  
 values=[7,8,9 ,3,4,5, 6,7,8] (depends on solution of problem ..., can also contain tuples, etc.)

– **notes:** This function is still under development and shows an experimental state!

– **example:**

```
GeneticOptimization(objectiveFunction = f0pt, parameters={'mass':(1,10), 'stiffness':(1000,10000)})
```

For examples on GeneticOptimization see Examples (Ex) and TestModels (TM):

- [geneticOptimizationSliderCrank.py](#) (Ex), [geneticOptimizationTest.py](#) (TM)
- 

```
def Minimize (objectiveFunction, parameters, initialGuess= [], method= 'Nelder-Mead', tol= 1e-4, options= , enforceBounds= True, debugMode= False, showProgress= True, addComputationIndex= False, storeFunctionValues= True, **kwargs)
```

– **function description:** Compute minimum of given objectiveFunction. This function is based on `scipy.optimize.minimize` and it provides the same interface as `GeneticOptimization()`.

– **input:**

*objectiveFunction*: function, which takes the form `parameterFunction(parameterDict)` and which returns a value or list (or numpy array) which reflects the size of the objective to be minimized

*parameters*: given as a dictionary, consist of name and tuple containing the search range for this parameter (begin, end), e.g. 'mass':(10,50)

*storeFunctionValues*: if True, objectiveFunction values are computed (additional costs!) and stored in every iteration into `valueList`

*initialGuess*: initial guess. Array of real elements of size (n,), where 'n' is the number of independent variables. If not provided by the user, `initialGuess` is computed from bounds provided in `parameterDict`.

*method*: solver that should be used, e.g. 'Nelder-Mead', 'Powell', 'CG' etc. A list of available solvers can be found in the documentation of `scipy.optimize.minimize()`.

*tol*: tolerance for termination. When `tol` is specified, the selected minimization algorithm sets some relevant solver-specific tolerance(s) equal to `tol`. For detailed control, use solver-specific options using the 'options' variable.

*options*: dictionary of solver options. Can be used to set absolute and relative error tolerances. Detailed information can be found in the documentation of `scipy.optimize.minimize()`.

*enforceBounds*: if True, ensures that only parameters within the bounds specified in `ParameterDict` are used for minimization; this may help to avoid, e.g., negative values, but may lead to non-convergence

*verbose*: prints solver information into console, e.g. number of iterations 'nit', number of function evaluations 'nfev', status etc.

*showProgress*: if True, shows for every iteration objective function value, number of current iteration, time needed for current iteration, maximum number of iterations until solver option 'maxiter' is reached.

*addComputationIndex*: if True, key 'computationIndex' is added for consistency reasons with `GeneticOptimization` to every `parameterDict` in the call to `parameterFunction()`; however, the value is always 0, because no multi threading is used in `Minimize(...)`

*resultsFile*: if provided, the results are stored columnwise into the given file and written after every generation; use `resultsMonitor.py` to track results in realtime

*useScipyBounds*: if True, use `scipy.optimize.minimize()` option 'bounds' to apply bounds on variable specified in `ParameterDict`. Note, this option is only used by some specific methods of `scipy.optimize.minimize()`! `method='Nelder-Mead'` ignores this option for example! if False, option 'enforceBounds' will be set to False!

*args*: extra arguments passed to the objective function and its derivatives (fun, jac and hess functions).  
*jac*: method for computing the gradient vector.  
*hess*: method for computing the Hessian matrix.  
*hessp*: hessian of objective function times an arbitrary vector p.  
*constraints*: constraints definition (only for COBYLA, SLSQP and trust-constr).

- **output**: returns [optimumParameter, optimumValue, parameterList, valueList], containing the optimum parameter set 'optimumParameter', optimum value 'optimumValue', the whole list of parameters parameterList with according objective values 'valueList'
- **author**: Stefan Holzinger, Johannes Gerstmayr
- **notes**: This function is still under development and shows an experimental state! There are currently unused arguments of `scipy.optimize.minimize()`: Detailed information can be found in the documentation of `scipy.optimize.minimize()`.

For examples on Minimize see Examples (Ex) and TestModels (TM):

- [minimizeExample.py](#) (Ex)
- 

```
def ComputeSensitivities (parameterFunction, parameters, scaledByReference= False, debugMode= False,
addComputationIndex= False, useMultiProcessing= False, showProgress= True, parameterFunctionData= dict(),
**kwargs)
```

- **function description**:

Perform a sensitivity analysis by successively calling the function `parameterFunction(parameterList[i])` with a one at a time variation of parameters in the defined increments.

e.g., `parameterList[0] =['mass':13, 'stiffness':12000]` to be computed and returns a value or a list of values which is then stored for each parameter

- **input**:

*parameterFunction*: function, which takes the form `parameterFunction(parameterDict)` and which returns one or more output values for which the sensitivity is calculated

*parameters*: given as a dictionary, consist of name and tuple of (begin, Variation steps, numberOfValues)  
e.g. 'mass':(10,0.01,5), for a reference mass of 10, incremented by 0.01\*10 and using 5 steps in negative and positive, doing 10 steps in total

*scaledByReference*: if true multiplies the sensitivities with the corresponding reference parameters, so that the sensitivity resembles a change relative to the reference value

*debugMode*: if True, additional information is shown

*addComputationIndex*: if True, key 'computationIndex' is added to every parameterDict in the call to `parameterFunction()`, which allows to generate independent output files for every parameter etc.

*useMultiProcessing*: if True, the multiprocessing lib is used for parallelized computation; WARNING: be aware that the function does not check if your function runs independently; DO NOT use GRAPHICS and DO NOT write to same output files, etc.!

*showProgress*: if True, shows for every iteration the progress bar (requires tqdm library)

- resultsFile*: if provided, output is immediately written to resultsFile during processing
- numberOfThreads*: default: same as number of cpus (threads); used for multiprocessing lib;
- parameterFunctionData*: dictionary containing additional data passed to the parameterFunction inside the parameters with dict key 'functionData'; use this e.g. for passing solver parameters or other settings
- **output**: returns [parameterList, valRef, valuesSorted, sensitivity], parameterList containing the list of dictionaries processed. valRef is the Solution for the reference values paramList[0], valuesSorted contains the results sorted by the dictionary key that was varied in the simulation. The sensitivity contains the calculated sensitivity, where the rows are the corresponding outputparameters, while the columns are the input parameters, thereby the index sensitivity[1,0] is the sensitivity of output parameter 1 with respect to the input parameter 0.
- **author**: Peter Manzl
- **example**:

```
ComputeSensitivities(parameterFunction=ParameterFunction, parameters = {'mass': (mRef
    , 0.01, 3), 'spring': (1000, 0.01, 10),}, multiprocessing=True)
```

For examples on ComputeSensitivities see Examples (Ex) and TestModels (TM):

- [ComputeSensitivitiesExample.py](#) (Ex)
- 

```
def PlotOptimizationResults2D (parameterList, valueList, xLogScale= False, yLogScale= False)
    – function description: visualize results of optimization for every parameter (2D plots)
    – input:
        parameterList: taken from output parameterList of GeneticOptimization, containing a dictionary with lists of parameters
        valueList: taken from output valueList of GeneticOptimization; containing a list of floats that result from the objective function
        xLogScale: use log scale for x-axis
        yLogScale: use log scale for y-axis
    – output: return [figList, axList] containing the corresponding handles; creates a figure for every parameter in parameterList
```

For examples on PlotOptimizationResults2D see Examples (Ex) and TestModels (TM):

- [geneticOptimizationSliderCrank.py](#) (Ex), [minimizeExample.py](#) (Ex), [geneticOptimizationTest.py](#) (TM)
- 

```
def PlotSensitivityResults (valRef, valuesSorted, sensitivity, fVar= None, strYAxis= None)
    – function description: visualize results of Sensitivityanalyis for every parameter (2D plots)
    – input:
```

*valRef*: The output values of the reference solution

*valuesSorted*: The output values of the analysed function sorted by the parameter which was varied

*sensitivity*: The sensitivity Matrix calculated by the function `ComputeSensitivities()`

*fVar*: The list of variation stepsizes. It is assumed to be 1e-3 if not defined.

*strYAxis*: A list of strings to label the plots *yAxis*

- **output:** return [fig, axs] containing the corresponding handles; creates a subplot for every row in the sensitivity matrix
- **author:** Peter Manzl

For examples on `PlotSensitivityResults` see Examples (Ex) and TestModels (TM):

- [ComputeSensitivitiesExample.py](#) (Ex)

## 5.13 Module: rigidBodyUtilities

Advanced utility/mathematical functions for reference frames, rigid body kinematics and dynamics. Useful Euler parameter and Tait-Bryan angle conversion functions are included. A class for rigid body inertia creating and transformation is available.

Author: Johannes Gerstmayr, Stefan Holzinger (rotation vector and Tait-Bryan angles)

Date: 2020-03-10 (created)

def [ComputeOrthonormalBasisVectors](#) (*vector0*)

- **function description:** compute orthogonal basis vectors (normal1, normal2) for given vector0 (non-unique solution!); the length of vector0 must not be 1; if vector0 == [0,0,0], then any normal basis is returned
  - **output:** returns [vector0normalized, normal1, normal2], in which vector0normalized is the normalized vector0 (has unit length); all vectors in numpy array format
- 

def [ComputeOrthonormalBasis](#) (*vector0*)

- **function description:** compute orthogonal basis, in which the normalized vector0 is the first column and the other columns are normals to vector0 (non-unique solution!); the length of vector0 must not be 1; if vector0 == [0,0,0], then any normal basis is returned
  - **output:** returns A, a rotation matrix, in which the first column is parallel to vector0; A is a 2D numpy array
- 

def [GramSchmidt](#) (*vector0, vector1*)

- **function description:** compute Gram-Schmidt projection of given 3D vector 1 on vector 0 and return normalized triad (vector0, vector1, vector0 x vector1)

For examples on GramSchmidt see Examples (Ex) and TestModels (TM):

- [sliderCrank3Dbenchmark.py](#) (TM)
- 

```
def Skew (vector)
```

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

For examples on Skew see Examples (Ex) and TestModels (TM):

- [leggedRobot.py](#) (Ex), [stiffFlyballGovernor2.py](#) (Ex), [carRollingDiscTest.py](#) (TM),  
[explicitLieGroupIntegratorPythonTest.py](#) (TM), [explicitLieGroupIntegratorTest.py](#) (TM), [heavyTop.py](#) (TM),  
[LieGroupIntegrationUnitTests.py](#) (TM), [mecanumWheelRollingDiscTest.py](#) (TM), ...
- 

```
def Skew2Vec (skew)
```

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

```
def ComputeSkewMatrix (v)
```

- **function description:** compute  $(3 \times 3^*n)$  skew matrix from  $(3^*n)$  vector

For examples on ComputeSkewMatrix see Examples (Ex) and TestModels (TM):

- [objectFFRFTest.py](#) (TM)
- 

```
def EulerParameters2G (eulerParameters)
```

- **function description:** convert Euler parameters (ep) to G-matrix ( $=\partial \omega / \partial \mathbf{p}_t$ )
  - **input:** vector of 4 eulerParameters as list or np.array
  - **output:** 3x4 matrix G as np.array
- 

```
def EulerParameters2GLocal (eulerParameters)
```

- **function description:** convert Euler parameters (ep) to local G-matrix ( $=\partial^b \omega / \partial \mathbf{p}_t$ )
- **input:** vector of 4 eulerParameters as list or np.array

- **output:** 3x4 matrix G as np.array

For examples on EulerParameters2GLocal see Examples (Ex) and TestModels (TM):

- [objectFFRFTTest.py](#) (TM), [rigidBodyAsUserFunctionTest.py](#) (TM)
- 

```
def EulerParameters2RotationMatrix (eulerParameters)
```

- **function description:** compute rotation matrix from eulerParameters
- **input:** vector of 4 eulerParameters as list or np.array
- **output:** 3x3 rotation matrix as np.array

For examples on EulerParameters2RotationMatrix see Examples (Ex) and TestModels (TM):

- [stiffFlyballGovernor2.py](#) (Ex), [stiffFlyballGovernor.py](#) (TM)
- 

```
def RotationMatrix2EulerParameters (rotationMatrix)
```

- **function description:** compute Euler parameters from given rotation matrix
- **input:** 3x3 rotation matrix as list of lists or as np.array
- **output:** vector of 4 eulerParameters as np.array

For examples on RotationMatrix2EulerParameters see Examples (Ex) and TestModels (TM):

- [mouseInteractionExample.py](#) (Ex), [NGsolvePistonEngine.py](#) (Ex), [stiffFlyballGovernor2.py](#) (Ex),  
[driveTrainTest.py](#) (TM), [perf3DRigidBodies.py](#) (TM), [stiffFlyballGovernor.py](#) (TM)
- 

```
def AngularVelocity2EulerParameters_t (angularVelocity, eulerParameters)
```

- **function description:**

compute time derivative of Euler parameters from (global) angular velocity vector

note that for Euler parameters  $\mathbf{p}$ , we have  $\boldsymbol{\omega} = \mathbf{G}\dot{\mathbf{p}} \implies \mathbf{G}^T\boldsymbol{\omega} = \mathbf{G}^T \cdot \mathbf{G} \cdot \dot{\mathbf{p}} \implies \mathbf{G}^T\mathbf{G} = 4(\mathbf{I}_{4 \times 4} - \mathbf{p} \cdot \mathbf{p}^T)\dot{\mathbf{p}} = 4(\mathbf{I}_{4 \times 4})\dot{\mathbf{p}}$

- **input:**

*angularVelocity*: 3D vector of angular velocity in global frame, as lists or as np.array

*eulerParameters*: vector of 4 eulerParameters as np.array or list

- **output:** vector of time derivatives of 4 eulerParameters as np.array

```
def RotationVector2RotationMatrix (rotationVector)
```

- **function description:** rotation matrix from rotation vector, see appendix B in [35]
- **input:** 3D rotation vector as list or np.array
- **output:** 3x3 rotation matrix as np.array

For examples on RotationVector2RotationMatrix see Examples (Ex) and TestModels (TM):

- [stiffFlyballGovernor2.py](#) (Ex), [explicitLieGroupMBSTest.py](#) (TM), [stiffFlyballGovernor.py](#) (TM)
- 

```
def RotationMatrix2RotationVector (rotationMatrix)
```

- **function description:** compute rotation vector from rotation matrix
- **input:** 3x3 rotation matrix as list of lists or as np.array
- **output:** vector of 3 components of rotation vector as np.array

For examples on RotationMatrix2RotationVector see Examples (Ex) and TestModels (TM):

- [explicitLieGroupMBSTest.py](#) (TM)
- 

```
def ComputeRotationAxisFromRotationVector (rotationVector)
```

- **function description:** compute rotation axis from given rotation vector
- **input:** 3D rotation vector as np.array
- **output:** 3D vector as np.array representing the rotation axis

For examples on ComputeRotationAxisFromRotationVector see Examples (Ex) and TestModels (TM):

- [LieGroupIntegrationUnitTests.py](#) (TM)
- 

```
def RotationVector2G (rotationVector)
```

- **function description:** convert rotation vector (parameters) (*v*) to G-matrix ( $=\partial\omega/\partial\dot{v}$ )
- **input:** vector of rotation vector (len=3) as list or np.array
- **output:** 3x3 matrix *G* as np.array

---

```
def RotationVector2GLocal (eulerParameters)
```

- **function description:** convert rotation vector (parameters) ( $\mathbf{v}$ ) to local G-matrix ( $=\partial^b \boldsymbol{\omega} / \partial \mathbf{v}_t$ )
  - **input:** vector of rotation vector (len=3) as list or np.array
  - **output:** 3x3 matrix G as np.array
- 

```
def RotXYZ2RotationMatrix (rot)
```

- **function description:** compute rotation matrix from consecutive xyz rotations (Rots) (Tait-Bryan angles);  $A = A_x * A_y * A_z$ ; rot=[rotX, rotY, rotZ]
- **input:** 3D vector of Tait-Bryan rotation parameters [X,Y,Z] in radiant
- **output:** 3x3 rotation matrix as np.array

For examples on RotXYZ2RotationMatrix see Examples (Ex) and TestModels (TM):

- stiffFlyballGovernor2.py (Ex), explicitLieGroupMBSTest.py (TM), stiffFlyballGovernor.py (TM)
- 

```
def RotationMatrix2RotXYZ (rotationMatrix)
```

- **function description:** convert rotation matrix to xyz Euler angles (Tait-Bryan angles);  $A = A_x * A_y * A_z$ ;
  - **input:** 3x3 rotation matrix as list of lists or np.array
  - **output:** vector of Tait-Bryan rotation parameters [X,Y,Z] (in radiant) as np.array
- 

```
def RotXYZ2G (rot)
```

- **function description:** compute (global-frame) G-matrix for xyz Euler angles (Tait-Bryan angles) ( ${}^0\mathbf{G} = \partial {}^0 \boldsymbol{\omega} / \partial \dot{\theta}$ )
  - **input:** 3D vector of Tait-Bryan rotation parameters [X,Y,Z] in radiant
  - **output:** 3x3 matrix G as np.array
- 

```
def RotXYZ2G_t (rot, rot_t)
```

- **function description:** compute time derivative of (global-frame) G-matrix for xyz Euler angles (Tait-Bryan angles) ( ${}^0\mathbf{G} = \partial {}^0 \boldsymbol{\omega} / \partial \dot{\theta}$ )
- **input:**  
*rot*: 3D vector of Tait-Bryan rotation parameters [X,Y,Z] in radiant

*rot\_t*: 3D vector of time derivative of Tait-Bryan rotation parameters [X,Y,Z] in radiant/s

- **output:** 3x3 matrix  $G_t$  as np.array
- 

```
def RotXYZ2GLocal (rot)
```

- **function description:** compute local (body-fixed) G-matrix for xyz Euler angles (Tait-Bryan angles) ( ${}^bG = \partial {}^b\omega / \partial \theta_i$ )
  - **input:** 3D vector of Tait-Bryan rotation parameters [X,Y,Z] in radiant
  - **output:** 3x3 matrix  $GLocal$  as np.array
- 

```
def RotXYZ2GLocal_t (rot, rot_t)
```

- **function description:** compute time derivative of (body-fixed) G-matrix for xyz Euler angles (Tait-Bryan angles) ( ${}^bG = \partial {}^b\omega / \partial \theta_i$ )
  - **input:**
    - rot*: 3D vector of Tait-Bryan rotation parameters [X,Y,Z] in radiant
    - rot\_t*: 3D vector of time derivative of Tait-Bryan rotation parameters [X,Y,Z] in radiant/s
  - **output:** 3x3 matrix  $GLocal_t$  as np.array
- 

```
def AngularVelocity2RotXYZ_t (angularVelocity, rotation)
```

- **function description:** compute time derivatives of angles RotXYZ from (global) angular velocity vector and given rotation
  - **input:**
    - angularVelocity*: global angular velocity vector as list or np.array
    - rotation*: 3D vector of Tait-Bryan rotation parameters [X,Y,Z] in radiant
  - **output:** time derivative of vector of Tait-Bryan rotation parameters [X,Y,Z] (in radiant) as np.array
- 

```
def RotXYZ2EulerParameters (alpha)
```

- **function description:** compute four Euler parameters from given RotXYZ angles, see [22]
- **input:** *alpha*: 3D vector as np.array containing RotXYZ angles

- **output:**  
4D vector as np.array containing four Euler parameters  
entry zero of output represent the scalar part of Euler parameters
- 

```
def RotationMatrixX (angleRad)
```

- **function description:** compute rotation matrix w.r.t. X-axis (first axis)
- **input:** angle around X-axis in radiant
- **output:** 3x3 rotation matrix as np.array

For examples on RotationMatrixX see Examples (Ex) and TestModels (TM):

- [addPrismaticJoint.py](#) (Ex), [addRevoluteJoint.py](#) (Ex), [particleClusters.py](#) (Ex),  
[solutionViewerTest.py](#) (Ex), [generalContactFrictionTests.py](#) (TM), [mecanumWheelRollingDiscTest.py](#) (TM), [perf3DRigidBody.py](#) (TM), [revoluteJointPrismaticJointTest.py](#) (TM), ...
- 

```
def RotationMatrixY (angleRad)
```

- **function description:** compute rotation matrix w.r.t. Y-axis (second axis)
- **input:** angle around Y-axis in radiant
- **output:** 3x3 rotation matrix as np.array

For examples on RotationMatrixY see Examples (Ex) and TestModels (TM):

- [addPrismaticJoint.py](#) (Ex), [addRevoluteJoint.py](#) (Ex), [bicycleIftommBenchmark.py](#) (Ex),  
[leggedRobot.py](#) (Ex), [stiffFlyballGovernor2.py](#) (Ex), ... , [ConvexContactTest.py](#) (TM), [revoluteJointPrismaticJointTest.py](#) (TM), [rollingCoinPenaltyTest.py](#) (TM), ...
- 

```
def RotationMatrixZ (angleRad)
```

- **function description:** compute rotation matrix w.r.t. Z-axis (third axis)
- **input:** angle around Z-axis in radiant
- **output:** 3x3 rotation matrix as np.array

For examples on RotationMatrixZ see Examples (Ex) and TestModels (TM):

- [addPrismaticJoint.py](#) (Ex), [addRevoluteJoint.py](#) (Ex), [bicycleIftommBenchmark.py](#) (Ex),  
[fourBarMechanism3D.py](#) (Ex), [mouseInteractionExample.py](#) (Ex), ... , [carRollingDiscTest.py](#) (TM), [driveTrainTest.py](#) (TM), [generalContactFrictionTests.py](#) (TM), ...

---

```
def HomogeneousTransformation (A, r)
```

- **function description:** compute homogeneous transformation ([HT](#)) matrix from rotation matrix A and translation vector r
- 

```
def HTtranslate (r)
```

- **function description:** [HT](#) for translation with vector r

For examples on HTtranslate see Examples (Ex) and TestModels (TM):

- [serialRobotOldTests.py](#) (Ex), [serialRobotTestDH2.py](#) (Ex), [serialRobotTSD.py](#) (Ex),  
[serialRobotTest.py](#) (TM)
- 

```
def HTtranslateX (x)
```

- **function description:** [HT](#) for translation along x axis with value x
- 

```
def HTtranslateY (y)
```

- **function description:** [HT](#) for translation along y axis with value y
- 

```
def HTtranslateZ (z)
```

- **function description:** [HT](#) for translation along z axis with value z
- 

```
def HT0 ()
```

- **function description:** identity [HT](#):

For examples on HT0 see Examples (Ex) and TestModels (TM):

- [serialRobotOldTests.py](#) (Ex), [serialRobotTestDH2.py](#) (Ex)

---

```
def HTrotateX (angle)
– function description: HT for rotation around axis X (first axis)
```

---

```
def HTrotateY (angle)
– function description: HT for rotation around axis X (first axis)
```

---

```
def HTrotateZ (angle)
– function description: HT for rotation around axis X (first axis)
```

---

```
def HT2translation (T)
– function description: return translation part of HT
```

For examples on HT2translation see Examples (Ex) and TestModels (TM):

- [serialRobotOldTests.py](#) (Ex), [serialRobotTestDH2.py](#) (Ex), [serialRobotTSD.py](#) (Ex),  
[serialRobotTest.py](#) (TM)
- 

```
def HT2rotationMatrix (T)
– function description: return rotation matrix of HT
```

For examples on HT2rotationMatrix see Examples (Ex) and TestModels (TM):

- [serialRobotTestDH2.py](#) (Ex)
- 

```
def InverseHT (T)
– function description: return inverse HT such that inv(T)*T = np.eye(4)
```

For examples on InverseHT see Examples (Ex) and TestModels (TM):

- [serialRobotTestDH2.py](#) (Ex)
- 

```
def RotationX2T66 (angle)
```

- **function description:** compute 6x6 coordinate transformation matrix for rotation around X axis; output: first 3 components for rotation, second 3 components for translation! See Featherstone / Handbook of robotics [34]; note that Plücker transformation ([T66](#)) represents coordinate transforms, which is the transposed of rotation matrices used in Exudyn
- 

```
def RotationY2T66 (angle)
```

- **function description:** compute 6x6 transformation matrix for rotation around Y axis; output: first 3 components for rotation, second 3 components for translation; note that T66 represents coordinate transforms, which is the transposed of rotation matrices used in Exudyn
- 

```
def RotationZ2T66 (angle)
```

- **function description:** compute 6x6 transformation matrix for rotation around Z axis; output: first 3 components for rotation, second 3 components for translation; note that T66 represents coordinate transforms, which is the transposed of rotation matrices used in Exudyn
- 

```
def Translation2T66 (translation3D)
```

- **function description:** compute 6x6 transformation matrix for translation according to 3D vector translation3D; output: first 3 components for rotation, second 3 components for translation!
- 

```
def TranslationX2T66 (translation)
```

- **function description:** compute 6x6 transformation matrix for translation along X axis; output: first 3 components for rotation, second 3 components for translation!
- 

```
def TranslationY2T66 (translation)
```

- 
- **function description:** compute 6x6 transformation matrix for translation along Y axis; output: first 3 components for rotation, second 3 components for translation!

```
def TranslationZ2T66 (translation)
```

- 
- **function description:** compute 6x6 transformation matrix for translation along Z axis; output: first 3 components for rotation, second 3 components for translation!

---

```
def T66toRotationTranslation (T66)
```

- **function description:** convert 6x6 coordinate transformation (Plücker transform) into rotation and translation
  - **input:** T66 given as 6x6 numpy array
  - **output:** [A, v] with 3x3 rotation matrix A and 3D translation vector v
- 

```
def RotationTranslation2T66 (A, v)
```

- **function description:** convert rotation and translation int 6x6 coordinate transformation (Plücker transform)
  - **input:**
    - A: 3x3 rotation matrix A
    - v: 3D translation vector v
  - **output:** return 6x6 transformation matrix 'T66'
- 

```
def T66toHT (T66)
```

- **function description:** convert 6x6 coordinate transformation (Plücker transform) into 4x4 homogeneous transformation; NOTE that the homogeneous transformation is the inverse of what is computed in function pluho() of Featherstone
  - **input:** T66 given as 6x6 numpy array
  - **output:** homogeneous transformation (4x4 numpy array)
-

```
def HT2T66 (T)
```

- **function description:** convert 4x4 homogeneous transformation into 6x6 coordinate transformation (Plücker transform); NOTE that the homogeneous transformation is the inverse of what is computed in function pluho() of Featherstone
  - **output:** input: T66 (6x6 numpy array)
- 

```
def InertiaTensor2Inertia6D (inertiaTensor)
```

- **function description:** convert a 3x3 matrix (list or numpy array) into a list with 6 inertia components, sorted as J00, J11, J22, J12, J02, J01

For examples on InertiaTensor2Inertia6D see Examples (Ex) and TestModels (TM):

- [serialRobotTestDH2.py](#) (Ex)
- 

```
def Inertia6D2InertiaTensor (inertia6D)
```

- **function description:** convert a list or numpy array with 6 inertia components (sorted as [J00, J11, J22, J12, J02, J01]) (list or numpy array) into a 3x3 matrix (np.array)

For examples on Inertia6D2InertiaTensor see Examples (Ex) and TestModels (TM):

- [serialRobotTestDH2.py](#) (Ex), [rigidBodyAsUserFunctionTest.py](#) (TM)
- 

```
def GetRigidBodyNode (nodeType, position= [0,0,0], velocity= [0,0,0], rotationMatrix= [], rotationParameters= [],  
angularVelocity= [0,0,0])
```

- **function description:** get node item interface according to nodeType, using initialization with position, velocity, angularVelocity and rotationMatrix

- **input:**

*nodeType*: a node type according to exudyn.NodeType, or a string of it, e.g., 'NodeType.RotationEulerParameters' (fastest, but additional algebraic constraint equation), 'NodeType.RotationRxyz' (Tait-Bryan angles, singularity for second angle at +/- 90 degrees), 'NodeType.RotationRotationVector' (used for Lie group integration)

*position*: reference position as list or numpy array with 3 components (in global/world frame)

*velocity*: initial translational velocity as list or numpy array with 3 components (in global/world frame)

*rotationMatrix*: 3x3 list or numpy matrix to define reference rotation; use EITHER *rotationMatrix*=[[...],[...],[...]] (while *rotationParameters*=[[]]) or *rotationParameters*=[...] (while *rotationMatrix*=[[]])

*rotationParameters*: reference rotation parameters; use EITHER *rotationMatrix*=[[...],[...],[...]] (while *rotationParameters*=[[]]) or *rotationParameters*=[...] (while *rotationMatrix*=[[]])

*angularVelocity*: initial angular velocity as list or numpy array with 3 components (in global/world frame)

- **output:** returns list containing node number and body number: [nodeNumber, bodyNumber]
- 

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

- **function description:** adds a node (with str(exu.NodeType. ...)) and body for a given rigid body; all quantities (esp. velocity and angular velocity) are given in global coordinates!

- **input:**

*inertia*: an inertia object as created by class RigidBodyInertia; containing mass, COM and inertia

*nodeType*: a node type according to exudyn.NodeType, or a string of it, e.g., 'NodeType.RotationEulerParameters' (fastest, but additional algebraic constraint equation), 'NodeType.RotationRxyz' (Tait-Bryan angles, singularity for second angle at +/- 90 degrees), 'NodeType.RotationRotationVector' (used for Lie group integration)

*position*: reference position as list or numpy array with 3 components (in global/world frame)

*velocity*: initial translational velocity as list or numpy array with 3 components (in global/world frame)

*rotationMatrix*: 3x3 list or numpy matrix to define reference rotation; use EITHER rotationMatrix=[[...],[...],[...]] (while rotationParameters=[]) or rotationParameters=[...] (while rotationMatrix=[])

*rotationParameters*: reference rotation parameters; use EITHER rotationMatrix=[[...],[...],[...]] (while rotationParameters=[]) or rotationParameters=[...] (while rotationMatrix=[])

*angularVelocity*: initial angular velocity as list or numpy array with 3 components (in global/world frame)

*gravity*: if provided as list or numpy array with 3 components, it adds gravity force to the body at the COM, i.e., fAdd = m\*gravity

*graphicsDataList*: list of graphicsData objects to define appearance of body

- **output:** returns list containing node number and body number: [nodeNumber, bodyNumber]

For examples on AddRigidBody see Examples (Ex) and TestModels (TM):

- [addPrismaticJoint.py](#) (Ex), [addRevoluteJoint.py](#) (Ex), [bicycleIftommBenchmark.py](#) (Ex), [fourBarMechanism3D.py](#) (Ex), [gyroStability.py](#) (Ex), ... , [carRollingDiscTest.py](#) (TM), [ConvexContactTest.py](#) (TM), [driveTrainTest.py](#) (TM), ...
- 

```
def AddRevoluteJoint (mbs, body0, body1, point, axis, useGlobalFrame= True, showJoint= True, axisRadius= 0.1,  
axisLength= 0.4)
```

- **function description:** add revolute joint between two bodies; definition of joint position and axis in global coordinates (alternatively in body0 local coordinates) for reference configuration of bodies; all markers, markerRotation and other quantities are automatically computed

– **input:**

*mbs*: the MainSystem to which the joint and markers shall be added

*body0*: a object number for body0, must be rigid body or ground object

*body1*: a object number for body1, must be rigid body or ground object

*point*: a 3D vector as list or np.array containing the global center point of the joint in reference configuration

*axis*: a 3D vector as list or np.array containing the global rotation axis of the joint in reference configuration

*useGlobalFrame*: if False, the point and axis vectors are defined in the local coordinate system of body0

– **output:** returns list [oJoint, mBody0, mBody1], containing the joint object number, and the two rigid body markers on body0/1 for the joint

For examples on AddRevoluteJoint see Examples (Ex) and TestModels (TM):

- [addRevoluteJoint.py](#) (Ex), [fourBarMechanism3D.py](#) (Ex), [rigidBodyTutorial3.py](#) (Ex),  
[solutionViewerTest.py](#) (Ex), [perf3DRigidBodies.py](#) (TM)
- 

```
def AddPrismaticJoint (mbs, body0, body1, point, axis, useGlobalFrame= True, showJoint= True, axisRadius= 0.1, axisLength= 0.4)
```

– **function description:** add prismatic joint between two bodies; definition of joint position and axis in global coordinates (alternatively in body0 local coordinates) for reference configuration of bodies; all markers, markerRotation and other quantities are automatically computed

– **input:**

*mbs*: the MainSystem to which the joint and markers shall be added

*body0*: a object number for body0, must be rigid body or ground object

*body1*: a object number for body1, must be rigid body or ground object

*point*: a 3D vector as list or np.array containing the global center point of the joint in reference configuration

*axis*: a 3D vector as list or np.array containing the global translation axis of the joint in reference configuration

*useGlobalFrame*: if False, the point and axis vectors are defined in the local coordinate system of body0

– **output:** returns list [oJoint, mBody0, mBody1], containing the joint object number, and the two rigid body markers on body0/1 for the joint

For examples on AddPrismaticJoint see Examples (Ex) and TestModels (TM):

- [addPrismaticJoint.py](#) (Ex)

### 5.13.1 CLASS RigidBodyInertia (in module rigidBodyUtilities)

**class description:** helper class for rigid body inertia (see also derived classes Inertia...). Provides a structure to define mass, inertia and center of mass (com) of a rigid body. The inertia tensor and center of mass must correspond when initializing the body!

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

– **example:**

```
i0 = RigidBodyInertia(10,np.diag([1,2,3]))
i1 = i0.Rotated(RotationMatrixX(np.pi/2))
i2 = i1.Translated([1,0,0])
```

```
def __init__(self, mass= 0, inertiaTensor= np.zeros([3,3]), com= np.zeros(3))
```

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

---

```
def __add__(self, otherBodyInertia)
```

– **classFunction:**

add (+) operator allows adding another inertia information with SAME local coordinate system  
only inertias with same center of rotation can be added!

– **example:**

```
J = InertiaSphere(2,0.1) + InertiaRodX(1,2)
```

---

```
def Translated(self, vec)
```

– **classFunction:** returns a RigidBodyInertia with center of mass com shifted by vec; → transforms the returned inertiaTensor to the new center of rotation

---

```
def Inertia(self)
```

– **classFunction:** returns 3x3 inertia tensor with respect to chosen reference point (not necessarily COM)

---

```
def InertiaCOM(self)
```

– **classFunction:** returns 3x3 inertia tensor with respect to chosen reference point (not necessarily COM)

---

```
def COM (self)
```

- **classFunction:** returns center of mass (COM) w.r.t. chosen reference point
- 

```
def Mass (self)
```

- **classFunction:** returns mass
- 

```
def Rotated (self, rot)
```

- **classFunction:** returns a RigidBodyInertia rotated by 3x3 rotation matrix *rot*, such that for a given *J*, the new inertia tensor reads  $J_{\text{new}} = \text{rot}^* J \text{rot.T}$
  - **notes:** only allowed if COM=0 !
- 

```
def GetInertia6D (self)
```

- **classFunction:** get vector with 6 inertia components ( $J_{xx}, J_{yy}, J_{zz}, J_{yz}, J_{xz}, J_{xy}$ ) as needed in ObjectRigid-Body

For examples on RigidBodyInertia see Examples (Ex) and TestModels (TM):

- [bicycleIftommBenchmark.py](#) (Ex), [sliderCrank3DwithANCFbeltDrive2.py](#) (Ex), [rigidBodyCOMtest.py](#) (TM),  
[rollingCoinPenaltyTest.py](#) (TM), [rollingCoinTest.py](#) (TM), [sliderCrank3Dbenchmark.py](#) (TM)

### 5.13.2 CLASS InertiaCuboid(RigidBodyInertia) (in module rigidBodyUtilities)

**class description:** create RigidBodyInertia with moment of inertia and mass of a cuboid with density and side lengths sideLengths along local axes 1, 2, 3; inertia w.r.t. center of mass, com=[0,0,0]

- **example:**

```
InertiaCuboid(density=1000, sideLengths=[1, 0.1, 0.1])
```

```
def __init__ (self, density, sideLengths)
```

- **classFunction:** initialize inertia

For examples on InertiaCuboid see Examples (Ex) and TestModels (TM):

- [addPrismaticJoint.py](#) (Ex), [addRevoluteJoint.py](#) (Ex), [fourBarMechanism3D.py](#) (Ex),  
[leggedRobot.py](#) (Ex), [mouseInteractionExample.py](#) (Ex), ... , [carRollingDiscTest.py](#) (TM), [driveTrainTest.py](#) (TM), [generalContactFrictionTests.py](#) (TM), ...

### 5.13.3 CLASS InertiaRodX(RigidBodyInertia) (in module rigidBodyUtilities)

**class description:** create RigidBodyInertia with moment of inertia and mass of a rod with mass m and length L in local 1-direction (x-direction); inertia w.r.t. center of mass, com=[0,0,0]

```
def __init__(self, mass, length)
    – classFunction: initialize inertia with mass and length of rod
```

For examples on InertiaRodX see Examples (Ex) and TestModels (TM):

- [fourBarMechanismRedundant.py](#) (TM)

### 5.13.4 CLASS InertiaMassPoint(RigidBodyInertia) (in module rigidBodyUtilities)

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

```
def __init__(self, mass)
    – classFunction: initialize inertia with mass of point
```

For examples on InertiaMassPoint see Examples (Ex) and TestModels (TM):

- [stiffFlyballGovernor2.py](#) (Ex), [stiffFlyballGovernor.py](#) (TM)

### 5.13.5 CLASS InertiaSphere(RigidBodyInertia) (in module rigidBodyUtilities)

**class description:** create RigidBodyInertia with moment of inertia and mass of sphere with mass and radius; inertia w.r.t. center of mass, com=[0,0,0]

```
def __init__(self, mass, radius)
    – classFunction: initialize inertia with mass and radius of sphere
```

For examples on InertiaSphere see Examples (Ex) and TestModels (TM):

- [particleClusters.py](#) (Ex), [tippeTop.py](#) (Ex), [generalContactFrictionTests.py](#) (TM)

### 5.13.6 CLASS InertiaHollowSphere(RigidBodyInertia) (in module rigidBodyUtilities)

**class description:** create RigidBodyInertia with moment of inertia and mass of hollow sphere with mass (concentrated at circumference) and radius; inertia w.r.t. center of mass, com=0

```
def __init__(self, mass, radius)
    – classFunction: initialize inertia with mass and (inner==outer) radius of hollow sphere
```

### 5.13.7 CLASS InertiaCylinder(RigidBodyInertia) (in module rigidBodyUtilities)

**class description:** create RigidBodyInertia with moment of inertia and mass of cylinder with density, length and outerRadius; axis defines the orientation of the cylinder axis (0=x-axis, 1=y-axis, 2=z-axis); for hollow cylinder use innerRadius != 0; inertia w.r.t. center of mass, com=[0,0,0]

```
def __init__(self, density, length, outerRadius, axis, innerRadius= 0)
```

- **classFunction:** initialize inertia with density, length, outer radius, axis (0=x-axis, 1=y-axis, 2=z-axis) and optional inner radius (for hollow cylinder)

For examples on InertiaCylinder see Examples (Ex) and TestModels (TM):

- [gyroStability.py](#) (Ex), [leggedRobot.py](#) (Ex), [ANCFbeltDrive.py](#) (TM),  
[ANCFgeneralContactCircle.py](#) (TM), [carRollingDiscTest.py](#) (TM), [ConvexContactTest.py](#) (TM), [driveTrainTest.py](#) (TM), [mecanumWheelRollingDiscTest.py](#) (TM), ...

## 5.14 Module: robotics

A library which includes support functions for robotics; the library is built on standard Denavit-Hartenberg Parameters and Homogeneous Transformations (HT) to describe transformations and coordinate systems; import this library e.g. with import exudyn.robotics as robotics

Author: Johannes Gerstmayr

Date: 2020-04-14

Example: New robot model uses the class Robot with class RobotLink; the old dictionary structure is defined in the example in ComputeJointHT for the definition of the 'robot' dictionary.

```
def StdDH2HT (DHparameters)
```

- **function description:** compute homogeneous transformation matrix HT from standard DHparameters=[theta, d, a, alpha]

For examples on StdDH2HT see Examples (Ex) and TestModels (TM):

- [serialRobotTestDH2.py](#) (Ex), [serialRobotTSD.py](#) (Ex), [serialRobotTest.py](#) (TM)
- 

```
def ModDHKK2HT (DHparameters)
```

- **function description:** compute pre- and post- homogeneous transformation matrices from modified Denavit-Hartenberg DHparameters=[alpha, d, theta, r]; returns [HTpre, HTpost]; HTpre is transformation before axis rotation, HTpost includes axis rotation and everything hereafter; modified DH-Parameters according to Khalil and Kleinfinger, 1986

For examples on ModDHKK2HT see Examples (Ex) and TestModels (TM):

- [serialRobotTestDH2.py](#) (Ex)
- 

```
def ConstantAccelerationParameters (duration, distance)
```

- **function description:** Compute parameters for optimal trajectory using given duration and distance
  - **input:** duration in seconds and distance in meters or radians
  - **output:** returns [vMax, accMax] with maximum velocity and maximum acceleration to achieve given trajectory
  - **notes:** DEPRECATED, DO NOT USE - moved to robotics.motion
- 

```
def ConstantAccelerationProfile (t, tStart, sStart, duration, distance)
```

- **function description:** Compute angle / displacement s, velocity v and acceleration a
- **input:**
  - t*: current time to compute values
  - tStart*: start time of profile
  - sStart*: start offset of path
  - duration*: duration of profile
  - distance*: total distance (of path) of profile
- **output:** [s, v, a] with path s, velocity v and acceleration a for constant acceleration profile; before tStart, solution is [0,0,0] while after duration, solution is [sStart+distance, 0, 0]
- **notes:** DEPRECATED, DO NOT USE - moved to robotics.motion

For examples on ConstantAccelerationProfile see Examples (Ex) and TestModels (TM):

- [serialRobotOldTests.py](#) (Ex)
- 

```
def MotionInterpolator (t, robotTrajectory, joint)
```

- **function description:** Compute joint value, velocity and acceleration for given robotTrajectory['PTP'] of point-to-point type, evaluated for current time t and joint number
- **input:**
  - t*: time to evaluate trajectory
  - robotTrajectory*: dictionary to describe trajectory; in PTP case, either use 'time' points, or 'time' and 'duration', or 'time' and 'maxVelocity' and 'maxAccelerations' in all consecutive points; 'maxVelocities' and 'maxAccelerations' must be positive nonzero values that limit velocities and accelerations;
  - joint*: joint number for which the trajectory shall be evaluated

- **output:** for current time t it returns [s, v, a] with path s, velocity v and acceleration a for current acceleration profile; outside of profile, it returns [0,0,0] !
- **notes:** DEPRECATED, DO NOT USE - moved to robotics.motion
- **example:**

```

q0 = [0,0,0,0,0,0] #initial configuration
q1 = [8,5,2,0,2,1] #other configuration
PTP = []
PTP+=[{'q':q0,
        'time':0}]
PTP+=[{'q':q1,
        'time':0.5}]
PTP+=[{'q':q1,
        'time':1e6}] #forever
RT={'PTP':PTP}
[u,v,a] = MotionInterpolator(t=0.5, robotTrajectory=RT, joint=1)

```

For examples on MotionInterpolator see Examples (Ex) and TestModels (TM):

- [serialRobotOldTests.py](#) (Ex), [serialRobotTestDH2.py](#) (Ex), [serialRobotTSD.py](#) (Ex),  
[serialRobotTest.py](#) (TM)
- 

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

- **function description:**

DEPRECATED function, use Robot.CreateRedundantCoordinateMBS(...); add items to existing mbs from the robot structure, a baseMarker (can be ground object or body)

and the user function list for the joints; there are options that can be passed as args / kwargs, which can contains options as described below. For details, see the python file and [serialRobotTest.py](#) in TestModels

- **input:**

*mbs*: the multibody system, which will be extended

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

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

*baseMarker*: a rigid body marker, at which the robot will be placed (usually ground); note that the local coordinate system of the base must be in accordance with the DH-parameters, i.e., the z-axis must be the first rotation axis. For correction of the base coordinate system, use *rotationMarkerBase*

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

*showCOM*: a scalar d, which if nonzero it causes to draw the center of mass (COM) as rectangular block with size [d,d,d]

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

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

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

*rotationMarkerBase*: add a numpy 3x3 matrix for rotation of the base, in order that the robot can be attached to any rotated base marker; the rotationMarkerBase is according to the definition in GenericJoint

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

For examples on SerialRobot2MBS see Examples (Ex) and TestModels (TM):

- [serialRobotOldTests.py](#) (Ex), [serialRobotTestDH2.py](#) (Ex), [serialRobotTest.py](#) (TM)
- 

```
def ComputeJointHT (robot, configuration)
```

- **function description**: DEPRECATED: compute list of homogeneous transformations HT from base to every joint (more precisely of every link!) for given configuration
- **example**:

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

For examples on ComputeJointHT see Examples (Ex) and TestModels (TM):

- [serialRobotOldTests.py](#) (Ex)
- 

```
def ComputeCOMHT (robot, HT)
```

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

For examples on ComputeCOMHT see Examples (Ex) and TestModels (TM):

- [serialRobotOldTests.py](#) (Ex)
- 

```
def ComputeStaticTorques (robot, HT)
```

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

For examples on ComputeStaticTorques see Examples (Ex) and TestModels (TM):

- [serialRobotOldTests.py](#) (Ex)
- 

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

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

For examples on Jacobian see Examples (Ex) and TestModels (TM):

- [serialRobotOldTests.py](#) (Ex), [serialRobotTestDH2.py](#) (Ex), [solverFunctionsTestEigenvalues.py](#) (Ex), [manualExplicitIntegrator.py](#) (TM), [scissorPrismaticRevolute2D.py](#) (TM)

### 5.14.1 CLASS VRobotLink (in module robotics)

**class description:** class to define visualization of RobotLink

```
def __init__ (self, jointRadius= 0.06, jointWidth= 0.05, showMBSjoint= True, linkWidth= 0.05, linkColor= [0.4,0.4,0.4,1], showCOM= True)
```

- **classFunction:** initialize robot link with parameters, being self-explaining

For examples on VRobotLink see Examples (Ex) and TestModels (TM):

- [serialRobotTestDH2.py](#) (Ex), [serialRobotTSD.py](#) (Ex), [serialRobotTest.py](#) (TM)

### 5.14.2 CLASS RobotLink (in module robotics)

**class description:** class to define one link of a robot

```
def __init__ (self, mass, COM, inertia, localHT= HT0(), jointType= 'Rz', parent= -2, preHT= HT0(), visualization= VRobotLink())
```

- **classFunction:** initialize robot link

- **input:**

*mass*: mass of robot link

*COM*: center of mass in link coordinate system

*inertia*: 3x3 matrix (list of lists / numpy array) containing inertia tensor in link coordinates, with respect to center of mass

*localHT*: 4x4 matrix (list of lists / numpy array) containing homogeneous transformation from local joint to link coordinates; default = identity

*preHT*: 4x4 matrix (list of lists / numpy array) containing homogeneous transformation from previous link to this joint; default = identity

*jointType*: string containing joint type, out of: 'Rx', 'Ry', 'Rz' for revolute joints and 'Px', 'Py', 'Pz' for prismatic joints around/along the respective local axes

*parent*: for building robots as kinematic tree; use '-2' to automatically set parents for serial robot (on fixed base), use '-1' for ground-parent and any other 0-based index for connection to parent link

*visualization*: VRobotLink structure containing options for drawing of link and joints; see class VRobotLink

For examples on RobotLink see Examples (Ex) and TestModels (TM):

- [serialRobotTestDH2.py](#) (Ex), [serialRobotTSD.py](#) (Ex), [serialRobotTest.py](#) (TM)

### 5.14.3 CLASS VRobotTool (in module robotics)

**class description:** class to define visualization of RobotTool

```
def __init__(self, graphicsData= [])
```

- **classFunction**: initialize robot tool with parameters; currently only graphicsData, which is a list of GraphicsData same as in mbs Objects

For examples on VRobotTool see Examples (Ex) and TestModels (TM):

- [serialRobotTestDH2.py](#) (Ex), [serialRobotTSD.py](#) (Ex), [serialRobotTest.py](#) (TM)

### 5.14.4 CLASS RobotTool (in module robotics)

**class description:** define tool of robot: containing graphics and HT (may add features in future)

```
def __init__(self, HT= HT0(), visualization= VRobotTool())
```

- **classFunction**: initialize robot tool

- **input:**

*HT*: 4x4 matrix (list of lists / numpy array) containing homogeneous transformation to transform from last link to tool

*graphicsData*: dictionary containing a list of GraphicsData, same as in exudyn Objects

For examples on RobotTool see Examples (Ex) and TestModels (TM):

- [serialRobotTestDH2.py](#) (Ex), [serialRobotTSD.py](#) (Ex), [serialRobotTest.py](#) (TM)

### 5.14.5 CLASS VRobotBase (in module robotics)

**class description:** class to define visualization of RobotBase

```
def __init__(self, graphicsData= [])
```

- **classFunction:** initialize robot base with parameters; currently only graphicsData, which is a list of GraphicsData same as in mbs Objects

For examples on VRobotBase see Examples (Ex) and TestModels (TM):

- [serialRobotTestDH2.py](#) (Ex), [serialRobotTSD.py](#) (Ex), [serialRobotTest.py](#) (TM)

### 5.14.6 CLASS RobotBase (in module robotics)

**class description:** define base of robot: containing graphics and HT (may add features in future)

```
def __init__(self, HT= HT0(), visualization= VRobotBase())
```

- **classFunction:** initialize robot base

- **input:**

*HT*: 4x4 matrix (list of lists / numpy array) containing homogeneous transformation to transform from world coordinates to base coordinates (changes orientation and position of robot)

*graphicsData*: dictionary containing a list of GraphicsData, same as in exudyn Objects

For examples on RobotBase see Examples (Ex) and TestModels (TM):

- [serialRobotTestDH2.py](#) (Ex), [serialRobotTSD.py](#) (Ex), [serialRobotTest.py](#) (TM)

### 5.14.7 CLASS Robot (in module robotics)

**class description:** class to define a robot

```
def __init__(self, gravity= [0,0,-9.81], base= RobotBase(), tool= RobotTool(), referenceConfiguration= [])
```

- **classFunction:** initialize robot class

- **input:** asfd: asdf

---

```
def AddLink(self, robotLink)
```

- **classFunction:** add a link to serial robot

---

```
def GetLink (self, i)
```

- **classFunction:** return Link object of link i
- 

```
def NumberOfLinks (self)
```

- **classFunction:** return number of links
- 

```
def GetBaseHT (self)
```

- **classFunction:** return base as homogeneous transformation
- 

```
def GetToolHT (self)
```

- **classFunction:** return base as homogeneous transformation
- 

```
def LinkHT (self, q)
```

- **classFunction:** compute list of homogeneous transformations for every link, using current joint coordinates q; leads to different results for standard and modified DH parameters because link coordinates are different!
- 

```
def JointHT (self, q)
```

- **classFunction:** compute list of homogeneous transformations for every joint (after rotation), using current joint coordinates q
- 

```
def COMHT (self, HT)
```

- 
- **classFunction:** compute list of homogeneous transformations HT from base to every COM using HT list from Robot.JointHT(...)

---

```
def StaticTorques (self, HT)
```

- **classFunction:** compute list of joint torques for serial robot due to gravity (gravity and mass as given in robot), taking HT from Robot.JointHT()
- 

---

```
def Jacobian (self, HT, toolPosition= [], mode= 'all')
```

- **classFunction:** compute jacobian for translation and rotation at toolPosition using joint HT; this is using the Robot functions, but is inefficient for simulation purposes

- **input:**

*HT:* list of homogeneous transformations per joint , as computed by Robot.JointHT(...)

*toolPosition:* global position at which the jacobian is evaluated (e.g., COM); if empty [], it uses the origin of the last link

*mode:* 'all'...translation and rotation jacobian, 'trans'...only translation part, 'rot': only rotation part

---

---

```
def CreateRedundantCoordinateMBS (self, mbs, baseMarker, jointLoadUserFunctionList= [],  
createJointTorqueLoads= True, *args, **kwargs)
```

- **classFunction:**

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

an optional user function list for joint loads; there are options that can be passed as args / kwargs, which can contains options as described below. For details, see the python file and `serialRobotTest.py` in TestModels;

the robot is built as rigid bodies (containing rigid body nodes), bodies represent the links which are connected by joints; joint torques need to be applied to bodies, applying a torque always with opposite direction to previous (=left) and next (=right) links (=bodies)

- **input:**

*mbs:* the multibody system, which will be extended

*baseMarker:* a rigid body marker, at which the robot will be placed (usually ground); note that the local coordinate system of the base must be in accordance with the DH-parameters, i.e., the z-axis must be the first rotation axis. For correction of the base coordinate system, use rotationMarkerBase

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

*createJointTorqueLoads*: if True, independently of jointLoadUserFunctionList, joint loads are created; the load numbers are stored in lists jointTorque0List/ jointTorque1List; the loads contain zero torques and need to be updated in every computation step, e.g., using a preStepUserFunction; unitTorque0List/ unitTorque1List contain the unit torque vector for the according body(link) which needs to be applied on both bodies attached to the joint

*rotationMarkerBase*: add a numpy 3x3 matrix for rotation of the base, in order that the robot can be attached to any rotated base marker; the rotationMarkerBase is according to the definition in GenericJoint; note, that for moving base, the static compensation does not work (base rotation must be updated)

- **output**: the function returns a dictionary containing per link nodes and object (body) numbers, 'nodeList', 'bodyList', the object numbers for joints, 'jointList', list of load numbers for joint torques (jointTorque0List, jointTorque1List), and unit torque vectors in local coordinates of the bodies to which the torques are applied (unitTorque0List, unitTorque1List)
- 

```
def GetKinematicTree66 (self)
```

- **classFunction**: export kinematicTree
- 

```
def GetLinkGraphicsData (self, i, p0, p1, axis0, axis1, linkVisualization)
```

- **classFunction**: create link GraphicsData (list) for link i; internally used in CreateRedundantCoordinateMBS(...); linkVisualization contains visualization dict of link
- 

```
def BuildFromDictionary (self, robotDict)
```

- **classFunction**: build robot structre from dictionary; this is a DEPRECATED function, which is used in older models; DO NOT USE

For examples on Robot see Examples (Ex) and TestModels (TM):

- [serialRobotOldTests.py](#) (Ex), [serialRobotTestDH2.py](#) (Ex), [serialRobotTSD.py](#) (Ex),  
[serialRobotTest.py](#) (TM)

## 5.15 Module: robotics.future

The future module contains functionality which is currently under development and will be moved in other robotics libraries in future

Date: 2021-12-02

## 5.16 Module: robotics.models

This module contains robotics models; They can be imported by simply calling the functions, which return the according robot dictionary; the library is built on Denavit-Hartenberg Parameters and Homogeneous Transformations (HT) to describe transformations and coordinate systems

Author: Martin Sereinig

Date: 2021-01-10

```
def Manipulator4Rsimple ()
```

- **function description:** generate simple 4R manipulator as myRobot dictionary, settings are done in function
  - **input:** nothing
  - **output:** myRobot dictionary
  - **notes:** the 4th joint is used to simulate a paralell kinematics manipulator
- 

```
def Manipulator3RSimple ()
```

- **function description:** generate simple 3R manipulator as myRobot dictionary, settings are done in function
  - **input:** nothing
  - **output:** myRobot dictionary
- 

```
def ManipulatorPANDA ()
```

- **function description:** generate simple Franka Emika Panda manipulator as myRobot dictionary, settings are done in function
  - **input:** nothing
  - **output:** myRobot dictionary
- 

```
def ManipulatorUR5 ()
```

- **function description:** generate UR5 manipulator as myRobot dictionary, settings are done in function
- **input:** nothing
- **output:** myRobot dictionary
- **notes:**
  - define myRobot kinematics, UR5 Universal Robotics,

*Standard DH-parameters:* [theta, d, a, alpha], according to P. Corke,  
Links modelled as cylindrical tubes, Inertia from Parham M. Kebria2016 / Kuefeta2014

---

```
def ManipulatorPuma560 ()
```

- **function description:** generate puma560 manipulator as myRobot dictionary, settings are done in function
  - **input:**
  - **output:** myRobot dictionary
  - **notes:** std DH-parameters: [theta, d, a, alpha], according to P. Corke page 138, puma p560 limits, taken from Corke Visual Control of Robots
- 

```
def LinkList2Robot (robotClass, robotLinkList, dhmode)
```

- **function description:** generate serial manipulator as robotClass object from linklist settings are done in function
- **input:**
  - robotClass:* robot class object generated before by robotics module
  - robotlinkList:* list of robot links generated by manipulator import for individual robot dictionary
  - dhmode:* 'std' for standard Denavit Hartenberg parameter, 'mod' for modified Denavit Hartenberg parameter
- **output:** myRobot dictionary
- **notes:**
  - DH Parameter Information
  - stdH* = [theta, d, a, alpha] with  $Rz(\theta) * Tz(d) * Tx(a) * Rx(\alpha)$
  - modDH* = [alpha, dx, theta, rz] with
    - used by Corke and Lynch:*  $Rx(\alpha) * Tx(a) * Rz(\theta) * Tz(d)$
    - used by Khali:*  $Rx(\alpha) * Tx(d) * Rz(\theta) * Tz(r)$
  - Important note:*  $d(khali)=a(corke)$  and  $r(khali)=d(corke)$

## 5.17 Module: robotics.mobile

The utilities contains functionality for mobile robots

Date: 2021-12-02

## 5.18 Module: robotics.motion

functionality for motion including generation of trajectories with acceleration profiles, path planning and motion

Author: Johannes Gerstmayr

Date: 2022-02-16

### 5.18.1 CLASS ProfileConstantAcceleration (in module robotics.motion)

**class description:** class to create a constant acceleration (optimal) PTP trajectory; trajectory ignores global max. velocities and accelerations

– **input:**

*finalCoordinates*: list or numpy array with final coordinates for profile

*duration*: duration (time) for profile

– **output:** returns profile object, which is then used to compute interpolated trajectory

---

```
def __init__(self, finalCoordinates, duration)
```

– **classFunction:** initialize ProfileConstantAcceleration with vector of final coordinates and duration (time span)

---

```
def GetBasicProfile(self, initialTime, initialCoordinates, globalMaxVelocities, globalMaxAccelerations)
```

– **classFunction:** return a class representing profile which is used in Trajectory

For examples on ProfileConstantAcceleration see Examples (Ex) and TestModels (TM):

- [serialRobotTSD.py](#) (Ex), [serialRobotTest.py](#) (TM)

### 5.18.2 CLASS ProfileLinearAccelerationsList (in module robotics.motion)

**class description:** class to create a linear acceleration PTP profile, using a list of accelerations to define the profile; the (joint) coordinates and velocities are computed relative to values of previous profiles; ignores global max. accelerations and velocities of Trajectory

– **input:** accelerationList: list of tuples (relativeTime, accelerationVector) in which relativeTime is the time relative to the start of the profile (first time must be zero!) and accelerationVector is the list of accelerations of this time point, which is then linearly interpolated

– **output:** returns profile object, which is then used to compute interpolated trajectory in class Trajectory

– **example:**

```
profile = ProfileLinearAccelerationsList([(0,[0.,1.,2]), (0,[1.,1.,-2])])
```

```
def __init__(self, accelerationList)
```

- **classFunction**: initialize ProfileLinearAccelerationsList with a list of tuples containing time and acceleration vector
- 

```
def GetBasicProfile(self, initialTime, initialCoordinates, globalMaxVelocities, globalMaxAccelerations)
```

- **classFunction**: return a class representing profile which is used in Trajectory

### 5.18.3 CLASS ProfilePTP (in module robotics.motion)

**class description:** class to create a synchronous motion PTP trajectory, using max. accelerations and max velocities; duration automatically computed

- **input:**

*finalCoordinates*: list or numpy array with final coordinates for profile

*maxVelocities*: list or numpy array with maximum velocities; may be empty list []; used if smaller than *globalMaxVelocities*

*maxAccelerations*: list or numpy array with maximum accelerations; may be empty list []; used if smaller than *globalMaxAccelerations*

- **output**: returns profile object, which is then used to compute interpolated trajectory

---

```
def __init__(self, finalCoordinates, syncAccTimes= True, maxVelocities= [], maxAccelerations= [])
```

- **classFunction**: initialize ProfilePTP with final coordinates of motion, optionally max. velocities and accelerations just for this profile (overrides global settings)
- 

```
def GetBasicProfile(self, initialTime, initialCoordinates, globalMaxVelocities, globalMaxAccelerations)
```

- **classFunction**: return a class representing profile which is used in Trajectory

For examples on ProfilePTP see Examples (Ex) and TestModels (TM):

- [serialRobotTSD.py](#) (Ex)

#### 5.18.4 CLASS Trajectory (in module robotics.motion)

**class description:** class to define (PTP) trajectories for robots and multibody systems; trajectories are defined for a set of coordinates (e.g. joint angles or other coordinates which need to be interpolated over time)

- **example:**

```
#create simple trajectory for two joint coordinates:  
traj = Trajectory(initialCoordinates=[1,1], initialTime=1)  
#add optimal trajectory with max. accelerations:  
traj.Add(ProfileConstantAcceleration([2.,3.],2.))  
traj.Add(ProfileConstantAcceleration([3.,-1.],2.))  
#add profile with limited velocities and accelerations:  
traj.Add(ProfilePTP([1,1],syncAccTimes=False, maxVelocities=[1,1], maxAccelerations  
=[5,5]))  
#now evaluate trajectory at certain time point (this could be now applied in a user  
function)  
[s,v,a] = traj.Evaluate(t=0.5)  
  
def __init__(self, initialCoordinates, initialTime= 0, maxVelocities= [], maxAccelerations= [])  
  
- classFunction: initialize robot link with parameters, being self-explaining  
- input:  
  initialTime: initial time for initial coordinates  
  initialCoordinates: initial coordinates for profile  
  maxVelocities: list or numpy array to describe global maximum velocities per coordinate  
  maxAccelerations: list or numpy array to describe global maximum accelerations per coordinate
```

---

```
def GetFinalCoordinates (self)
```

- **classFunction:** returns the coordinates at the end of the (currently) Final profile
- 

```
def Add (self, profile)
```

- **classFunction:** add successively profiles, using MotionProfile class
- 

```
def GetTimes (self)
```

- **classFunction:** return vector of times of start/end of profiles

---

```
def Initialize (self)
```

- **classFunction**: initialize some parameters for faster evaluation
- 

```
def Evaluate (self, t)
```

- **classFunction**: return interpolation of trajectory for coordinates, velocities and accelerations at given time
  - **output**: [s, v, a] as numpy arrays representing coordinates, velocities and accelerations
- 

```
def EvaluateCoordinate (self, t, coordinate)
```

- **classFunction**: return interpolation of trajectory for coordinate, including velocity and acceleration coordinate at given time
  - **output**: [s, v, a] being scalar position, velocity and acceleration
  - **notes**: faster for single coordinate than Evaluate(...)
- 

```
def __iter__ (self)
```

- **classFunction**: iterator allows to use for x in trajectory: ... constructs
- 

```
def __getitem__ (self, key)
```

- **classFunction**: access to profiles via operator [], allowing trajectory[0], etc.
- 

```
def __len__ (self)
```

- **classFunction**: allow using len(trajectory)
-

```
def __repr__(self)
```

- **classFunction:** representation of Trajectory is given a list of profiles, allowing easy inspection of data

For examples on Trajectory see Examples (Ex) and TestModels (TM):

- [serialRobotTSD.py](#) (Ex), [serialRobotTest.py](#) (TM)

## 5.19 Module: robotics.special

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

Author: Martin Sereinig

Date: 2021-22-09

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

- **function description:** compute velocity manipulability measure for given pose (homogenous transformation)
- **input:**
  - robot:* robot class
  - HT:* actual pose as hoogenous transformaton matrix
  - mode:* rotational or translational part of the movement
  - singularWeight:* Weighting of singular configurations where the value would be infinity,default value=100
- **output:** velocity manipulability measure as scalar value, defined as  $\sqrt{\det(JJ^T)}$
- **notes:** compute velocity dependent manipulability definded by Yoshikawa, see [41]

---

```
def ForceManipulability (robot, HT, mode, singular_weight= 100)
```

- **function description:** compute force manipulability measure for given pose (homogenous transformation)
- **input:**
  - robot:* robot class
  - HT:* actual pose as hoogenous transformaton matrix
  - singularWeight:* Weighting of singular configurations where the value would be infinity,default value=100
  - mode:* rotational or translational part of the movement
- **output:** force manipulability measure as scalar value, defined as  $\sqrt{(\det(JJ^T))^{-1}}$
- **notes:** compute force dependent manipulability definded by Yoshikawa, see [41]

---

```
def StiffnessManipulability (robot, JointStiffness, HT, mode, singularWeight= 1000)
```

- **function description:** compute cartesian stiffness measure for given pose (homogenous transformation)
- **input:**
  - robot*: robot class
  - JointStiffness*: joint stiffness matrix
  - HT*: actual pose as hoogenous transformaton matrix
  - mode*: rotational or translational part of the movement
  - singularWeight*: Weighting of singular configurations where the value would be infinity,default value=1000
- **output:**
  - stiffness manipulability measure as scalar value, defined as minimum Eigenvaluae of the Cartesian stiffness matrix
  - Cartesian stiffness matrix
- **notes:**
  - **status:** this function is **currently under development** and under testing!

---

```
def JointJacobian (robot, HTJoint, HTLink)
```

- **function description:** compute joint jacobian for each frame for given pose (homogenous transformation)
- **input:**
  - robot*: robot class
  - HT*: actual pose as hoogenous transformaton matrix
- **output:** Link(body)-Jacobi matrix  $JJ: {}^iJJ_i = [{}^iJ_{Ri}, {}^iJ_{Ti}]$  for each link  $i$ , seperated in rotational ( $J_R$ ) and translational ( $J_T$ ) part of Jacobian matrix located in the  $i^{th}$  coordiante system, see [40]
- **notes:** runs over number of HTs given in HT (may be less than number of links), caclulations in link coordinate system located at the end of each link regarding Standard Denavid-Hartenberg parameters, see [8]

---

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

- **function description:** compute mass matrix from jointJacobian
- **input:**
  - robot*: robot structure
  - HT*: actual pose as hoogenous transformaton matrix

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

- **output:** MM: Mass matrix

- **notes:**

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

\*\* Dynamic equations in minimal coordinates as described in MehrkÃ¶rpersysteme by Woernle, [40], p206, eq6.90.

\*\* Calculations in link coordinate system at the end of each link

For examples on MassMatrix see Examples (Ex) and TestModels (TM):

- [solverFunctionsTestEigenvalues.py](#) (Ex), [manualExplicitIntegrator.py](#) (TM), [objectFFRFreducedOrderAccelerations.py](#) (TM),  
[objectFFRFreducedOrderShowModes.py](#) (TM), [objectFFRFreducedOrderStressModesTest.py](#) (TM), [objectFFRFreducedOrderTest.py](#) (TM), [objectFFRFTest.py](#) (TM), [objectFFRFTest2.py](#) (TM), ...
- 

```
def DynamicManipulability (robot, HT, MassMatrix, Tmax, mode, singularWeight= 1000)
```

- **function description:** compute dynamic manipulability measure for given pose (homogenous transformation)

- **input:**

*robot*: robot structure

*HT*: actual pose as homogenous transformaton matrix

*Tmax*: maximum joint torques

*mode*: rotational or translational part of the movement

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

*singularWeight*: Weighting of singular configurations where the value would be infinity, default value=1000

- **output:**

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

dynamic manipulability matrix

- **notes:** acceleration dependent manipulability definded by Chiacchio, see [6], eq.32. The eigenvectors and eigenvalues of N ([eigenvec eigenval]=eig(N))gives the direction and value of minimal and maximal accaleration )

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

```
def ComputeIK3R (robot, HT)
```

- **function description:** calculates the analytical inverse kinematics for 3R elbow type serial robot manipulator
  - **input:**  
 $robot$ : robot structure  
 $HT$ : desired position and orientation for the end effector as 4x4 homogeneous transformation matrix as list of lists or np.array
  - **output:**  
solutions, list of lists with possible joint angles [q1,q2,q3] (in radiant)  
to achieve the desired position and orientation (4 possible solutions, shoulder left/right, elbow up/down )  
left/down, left/up, right/up, right/down
  - **notes:** only applicable for standard Denavit-Hartenberg parameters
  - **status:** tested with various configurations and joint angles
- 

```
def ComputeIKPuma560 (robot, HT)
```

- **function description:** calculates the analytical inverse kinematics for Puma560 serial 6R robot manipulator
  - **input:**  
 $robot$ : robot structure  
 $HT$ : desired position and orientation for the end effector as 4x4 homogeneous transformation matrix as list of lists or np.array
  - **output:**  
solutions, list of lists with possible joint angles [q1,q2,q3,q4,q5,q6] (in radiant)  
to achieve the desired position and orientation (8 possible solutions, shoulder left/right, elbow up/down, wrist flipped/notflipped (rotated by pi) )  
left/down/notflipped, left/down/flipped, left/up/notflipped, left/up/flipped, right/up/notflipped, right/up/flipped, right/down/notflipped, right/down/flipped
  - **notes:** Usage for different manipulators with spherical wrist possible, only applicable for standard Denavit-Hartenberg parameters
  - **status:** tested (compared with Robotcs, Vision and Control book of P. Corke)
- 

```
def ComputeIKUR (robot, HTdes)
```

- **function description:** calculates the analytical inverse kinematics for UR type serial 6R robot manipulator without spherical wrist
- **input:**

*robot*: robot structure

*HT*: desired position and orientation for the end effector as 4x4 homogeneous transformation matrix as list of lists or np.array

– **output:**

solutions, list of lists with possible joint angles [q1,q2,q3,q4,q5,q6] (in radiant)

to achieve the desired position and orientation (8 possible solutions, shoulder left/right, elbow up/down, wrist flipped/notflipped (rotated by pi) )

[left/down/notflipped, left/down/flipped, left/up/notflipped, left/up/flipped, right/up/notflipped, right/up/flipped, right/down/notflipped, right/down/flipped]

- **notes:** Usage for different manipulators without spherical wrist possible UR3,UR5,UR10, only applicable for standard Denavit-Hartenberg parameters
- **status:** under development, works for most configurations, singularities not checked -> ZeroConfiguration not working

## 5.20 Module: robotics.utilities

The utilities contains general helper functions for the robotics module

Date: 2021-12-02

## 5.21 Module: signalProcessing

The signal library supports processing of signals for import (e.g. measurement data) and for filtering result data.

Date: 2020-12-10

Notes: This module is still under construction and should be used with care!

def FilterSensorOutput (*signal*, *filterWindow*= 5, *polyOrder*= 3, *derivative*= 0, *centralDifferentiate*= True)

- **function description:** filter output of sensors (using numpy savgol filter) as well as numerical differentiation to compute derivative of signal

– **input:**

*signal*: numpy array (2D array with column-wise storage of signals, as exported by EXUDYN position, displacement, etc. sensors); first column = time, other columns = signals to operate on; note that it is assumed, that time deviated in almost constant steps!

*derivative*: 0=no derivative, 1=first derivative, 2=second derivative, etc. (>2 only possible with filter)

*polyOrder*: order of polynomial for interpolation filtering

*filterWindow*: if zero: produces unfiltered derivative; if positive, must be ODD integer 1,3,5,... and > polyOrder; filterWindow determines the length of the filter window (e.g., to get rid of noise)

*centralDifferentiate*: if True, it uses a central differentiation for first order, unfiltered derivatives; leads to less phase shift of signal!

- **output:** numpy array containing same columns, but with filtered signal and according derivatives

For examples on FilterSensorOutput see Examples (Ex) and TestModels (TM):

- [ANCFOutputTest.py](#) (TM), [objectFFRFreducedOrderAccelerations.py](#) (TM)
- 

```
def FilterSignal (signal, samplingRate= -1, filterWindow= 5, polyOrder= 3, derivative= 0, centralDifferentiate= True)
```

- **function description:** filter 1D signal (using numpy savgol filter) as well as numerical differentiation to compute derivative of signal
- **input:**
  - signal:* 1D numpy array
  - samplingRate:* (time increment) of signal values, needed for derivatives
  - derivative:* 0=no derivative, 1=first derivative, 2=second derivative, etc. (>2 only possible with filter)
  - polyOrder:* order of polynomial for interpolation filtering
  - filterWindow:* if zero: produces unfiltered derivative; if positive, must be ODD integer 1,3,5,... and > polyOrder; filterWindow determines the length of the filter window (e.g., to get rid of noise)
  - centralDifferentiate:* if True, it uses a central differentiation for first order, unfiltered derivatives; leads to less phase shift of signal!
- **output:** numpy array containing same columns, but with filtered signal and according derivatives

For examples on FilterSignal see Examples (Ex) and TestModels (TM):

- [objectFFRFreducedOrderAccelerations.py](#) (TM)
- 

```
def ComputeFFT (time, data)
```

- **function description:** computes fast-fourier-transform (FFT) resulting in frequency, magnitude and phase of signal data using numpy.fft of numpy
- **input:**
  - time ... time vector in SECONDS in numpy format, having constant sampling rate (not checked!)
  - data ... data vector in numpy format
- **output:**
  - frequency ... frequency vector (Hz, if time is in SECONDS)
  - magnitude ... magnitude vector
  - phase ... phase vector (in radiant)
- **author:** Stefan Holzinger
- **date:** 02.04.2020

---

```
def GetInterpolatedSignalValue (time, dataArray, timeArray= [], timeArrayIndex= -1, dataArrayIndex= -1, rangeWarning= True, tolerance= 1e-6)
```

– **function description:** Interpolate signal having time values with constant sampling rate in timeArray and according data in dataArray

– **input:**

*time*: time at which the data should be evaluated

*dataArray*: 1D numpy array containing data values to be interpolated [alternatively: 2D numpy array, rows containg the data of the according time point; use dataArrayColumnIndex to specify the column of requested data]

*timeArray*: 1D numpy array containing time values with CONSTANT SAMPLING RATE to be interpolated [alternatively: 2D numpy array, rows containg the time and data of the according time point; use timeArrayColumnIndex to specify the column representing time]; if timeArray is empty list [], dataArray is used instead!

*rangeWarning*: print warning if resulting index gets out of range

*dataArrayColumnIndex*: in case of 2D arrays, this represents the column of the requested data

*timeArrayColumnIndex*: in case of 2D arrays, this represents the column of time values

*tolerance*: this tolerance is used to check, if the timeArray has equidistant interpolation and if the found indices are correct; use e.g. 1e10 in order to ignore this tolerance

– **output:** interpolated value

– **notes:** for interpolation of data WITHOUT constant data rate, use numpy.interp(time, timeArray, dataArray) in case that timeArray and dataArray are 1D arrays

## 5.22 Module: solver

The solver module provides interfaces to static, dynamic and eigenvalue solvers. Most of the solvers are implemented inside the C++ core.

Author: Johannes Gerstmayr

Date: 2020-12-02

Notes: Solver functions are included directly in exudyn and can be used with exu.SolveStatic(...)

```
def SolverErrorMessage (solver, mbs, isStatic= False, showCausingObjects= True, showCausingNodes= True, showHints= True)
```

– **function description:** helper function for unique error and helper messages

```

def SolveStatic (mbs, simulationSettings= exudyn.SimulationSettings(), updateInitialValues= False, storeSolver= True, showHints= False, showCausingItems= True, )
    - function description: solves the static mbs problem using simulationSettings; check theDoc.pdf for Main-SolverStatic for further details of the static solver; NOTE that this function is directly available from exudyn (using exudyn.SolveStatic(...))

    - input:
        mbs: the MainSystem containing the assembled system; note that mbs may be changed upon several runs of this function
        simulationSettings: specific simulation settings out of exu.SimulationSettings(), as described in Section 8.1.1; use options for newton, discontinuous settings, etc., from staticSolver sub-items
        updateInitialValues: if True, the results are written to initial values, such at a consecutive simulation uses the results of this simulation as the initial values of the next simulation
        storeSolver: if True, the staticSolver object is stored in the mbs.sys dictionary as mbs.sys['staticSolver'], and simulationSettings are stored as mbs.sys['simulationSettings']

    - output: returns True, if successful, False if fails; if storeSolver = True, mbs.sys contains staticSolver, which allows to investigate solver problems (check theDoc.pdf Section 8.3 and the items described in Section 8.3.6)
    - example:
        import exudyn as exu
        from exudyn.itemInterface import *
        SC = exu.SystemContainer()
        mbs = SC.AddSystem()
        #create simple system:
        ground = mbs.AddObject(ObjectGround())
        mbs.AddNode(NodePoint())
        body = mbs.AddObject(MassPoint(physicsMass=1, nodeNumber=0))
        m0 = mbs.AddMarker(MarkerBodyPosition(bodyNumber=ground))
        m1 = mbs.AddMarker(MarkerBodyPosition(bodyNumber=body))
        mbs.AddObject(CartesianSpringDamper(markerNumbers=[m0,m1], stiffness=[100,100,100]))
        mbs.AddLoad(LoadForceVector(markerNumber=m1, loadVector=[10,10,10]))
        mbs.Assemble()
        sims = exu.SimulationSettings()
        sims.timeIntegration.endTime = 10
        success = exu.SolveStatic(mbs, sims, storeSolver = True)
        print("success =", success)
        print("iterations = ", mbs.sys['staticSolver'].it)
        print("pos=", mbs.GetObjectOutputBody(body, localPosition=[0,0,0], variableType=exu.OutputVariableType.Position))

```

For examples on SolveStatic see Examples (Ex) and TestModels (TM):

- [3SpringsDistance.py](#) (Ex), [ALEANCF\\_pipe.py](#) (Ex), [ANCFALEtest.py](#) (Ex), [ANCF\\_cantilever\\_test.py](#) (Ex), [ANCF\\_contact\\_circle.py](#) (Ex), ..., [ANCFslidingAndALEjointTest.py](#) (TM), [ANCFbeltDrive.py](#) (TM), [ANCFcontactCircleTest.py](#) (TM), ...

---

```

def SolveDynamic (mbs, simulationSettings= exodyn.SimulationSettings(), solverType=
exodyn.DynamicSolverType.GeneralizedAlpha, updateInitialValues= False, storeSolver= True, showHints=
False, showCausingItems= True, )

```

- **function description:** solves the dynamic mbs problem using simulationSettings and solver type; check theDoc.pdf for MainSolverImplicitSecondOrder for further details of the dynamic solver; NOTE that this function is directly available from exodyn (using exodyn.SolveDynamic(...))
- **input:**
  - mbs*: the MainSystem containing the assembled system; note that mbs may be changed upon several runs of this function
  - simulationSettings*: specific simulation settings out of exu.SimulationSettings(), as described in [Section 8.1.1](#); use options for newton, discontinuous settings, etc., from timeIntegration; therein, implicit second order solvers use settings from generalizedAlpha and explicit solvers from explicitIntegration; be careful with settings, as the influence accuracy (step size!), convergence and performance (see special [Section 2.4.13](#))
  - solverType*: use exodyn.DynamicSolverType to set specific solver (default=generalized alpha)
  - updateInitialValues*: if True, the results are written to initial values, such at a consecutive simulation uses the results of this simulation as the initial values of the next simulation
  - storeSolver*: if True, the staticSolver object is stored in the mbs.sys dictionary as mbs.sys['staticSolver'], and simulationSettings are stored as mbs.sys['simulationSettings']
  - showHints*: show additional hints, if solver fails
  - showCausingItems*: if linear solver fails, this option helps to identify objects, etc. which are related to a singularity in the linearized system matrix
- **output:** returns True, if successful, False if fails; if storeSolver = True, mbs.sys contains staticSolver, which allows to investigate solver problems (check theDoc.pdf [Section 8.3](#) and the items described in [Section 8.3.6](#))
- **example:**

```

import exodyn as exu
from exodyn.itemInterface import *
SC = exu.SystemContainer()
mbs = SC.AddSystem()
#create simple system:
ground = mbs.AddObject(ObjectGround())
mbs.AddNode(NodePoint())
body = mbs.AddObject(MassPoint(physicsMass=1, nodeNumber=0))
m0 = mbs.AddMarker(MarkerBodyPosition(bodyNumber=ground))
m1 = mbs.AddMarker(MarkerBodyPosition(bodyNumber=body))
mbs.AddObject(CartesianSpringDamper(markerNumbers=[m0,m1], stiffness=[100,100,100]))
mbs.AddLoad(LoadForceVector(markerNumber=m1, loadVector=[10,10,10]))
mbs.Assemble()
sims = exu.SimulationSettings()
sims.timeIntegration.endTime = 10
success = exu.SolveDynamic(mbs, sims, storeSolver = True)
print("success =", success)

```

```

print("iterations = ", mbs.sys['dynamicSolver'].it)
print("pos=", mbs.GetObjectOutputBody(body, localPosition=[0,0,0],
    variableType=exu.OutputVariableType.Position))

```

For examples on SolveDynamic see Examples (Ex) and TestModels (TM):

- [3SpringsDistance.py](#) (Ex), [addPrismaticJoint.py](#) (Ex), [addRevoluteJoint.py](#) (Ex), [ALEANCF\\_pipe.py](#) (Ex), [ANCFALEtest.py](#) (Ex), ... , [ANCFslidingAndALEjointTest.py](#) (TM), [ANCFbeltDrive.py](#) (TM), [ANCFcontactFrictionTest.py](#) (TM), ...
- 

```
def ComputeLinearizedSystem (mbs, simulationSettings= exudyn.SimulationSettings(), useSparseSolver=False)
```

– **function description:** compute linearized system of equations for ODE2 part of mbs, not considering the effects of algebraic constraints

– **input:**

*mbs*: the MainSystem containing the assembled system

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

*useSparseSolver*: if False (only for small systems), all eigenvalues are computed in dense mode (slow for large systems!); if True, only the *numberOfEigenvalues* are computed (*numberOfEigenvalues* must be set!); Currently, the matrices are exported only in DENSE MODE from mbs! NOTE that the sparsesolver accuracy is much less than the dense solver

– **output:** [M, K, D]; list containing numpy mass matrix M, stiffness matrix K and damping matrix D

– **example:**

```
#take any example from the Examples or TestModels folder, e.g., 'cartesianSpringDamper.py' and run it
#then execute the following commands in the console (or add it to the file):
[M, K, D] = exu.ComputeLinearizedSystem(mbs)
print("M=", M)
print("K=", K)
```

---

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

– **function description:** compute eigenvalues for unconstrained ODE2 part of mbs, not considering the effects of algebraic constraints; the computation is done for the initial values of the mbs, independently of previous computations. If you would like to use the current state for the eigenvalue computation, you need to copy the current state to the initial state (using GetSystemState, SetSystemState, see [Section 4.5.1](#)).

– **input:**

*mbs*: the MainSystem containing the assembled system

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

*useSparseSolver*: if False (only for small systems), all eigenvalues are computed in dense mode (slow for large systems!); if True, only the *numberOfEigenvalues* are computed (*numberOfEigenvalues* must be set!); Currently, the matrices are exported only in DENSE MODE from mbs! NOTE that the sparsesolver accuracy is much less than the dense solver

*numberOfEigenvalues*: number of eigenvalues and eivenvectors to be computed; if *numberOfEigenvalues*==0, all eigenvalues will be computed (may be impossible for larger problems!)

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

- **output**: [eigenValues, eigenVectors]; eigenValues being a numpy array of eigen values ( $\omega_i^2$ , being the squared eigen frequencies in ( $\omega_i$  in rad/s!)), eigenVectors a numpy array containing the eigenvectors in every column

- **example**:

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

For examples on ComputeODE2Eigenvalues see Examples (Ex) and TestModels (TM):

- [nMassOscillatorInteractive.py](#) (Ex), [computeODE2EigenvaluesTest.py](#) (TM)
- 

```
def CheckSolverInfoStatistics (solverName, infoStat, numberOfEvaluations)
```

- **function description**:

helper function for solvers to check e.g. if high number of memory allocations happened during simulation

This can happen, if large amount of sensors are attached and output is written in every time step

- **input**: stat=exudyn.InfoStat() from previous step, *numberOfEvaluations* is a counter which is proportional to number of RHS evaluations in method

## 5.23 Module: utilities

Basic support functions for simpler creation of Exudyn models. Advanced functions for loading and animating solutions and for drawing a graph of the mbs system. This library requires numpy (as well as time and copy)

Author: Johannes Gerstmayr

Date: 2019-07-26 (created)

```
def PlotLineCode (index)
```

- **function description**: helper functions for matplotlib, returns a list of 28 line codes to be used in plot, e.g. 'r-' for red solid line

- **input:** index in range(0:28)
- **output:** a color and line style code for matplotlib plot

For examples on PlotLineCode see Examples (Ex) and TestModels (TM):

- [serialRobotOldTests.py](#) (Ex), [serialRobotTestDH2.py](#) (Ex), [serialRobotTSD.py](#) (Ex), [serialRobotTest.py](#) (TM)
- 

```
def FillInSubMatrix (subMatrix, destinationMatrix, destRow, destColumn)
```

- **function description:** fill submatrix into given destinationMatrix; all matrices must be numpy arrays
- **input:**
  - subMatrix:* input matrix, which is filled into destinationMatrix
  - destinationMatrix:* the subMatrix is entered here
  - destRow:* row destination of subMatrix
  - destColumn:* column destination of subMatrix
- **output:** destinationMatrix is changed after function call
- **notes:** may be erased in future!

For examples on FillInSubMatrix see Examples (Ex) and TestModels (TM):

- [objectFFRFTest.py](#) (TM)
- 

```
def SweepSin (t, t1, f0, f1)
```

- **function description:** compute sin sweep at given time t
- **input:**
  - t:* evaluate of sweep at time t
  - t1:* end time of sweep frequency range
  - f0:* start of frequency interval [f0,f1] in Hz
  - f1:* end of frequency interval [f0,f1] in Hz
- **output:** evaluation of sin sweep (in range -1..+1)

For examples on SweepSin see Examples (Ex) and TestModels (TM):

- [objectGenericODE2Test.py](#) (TM)
- 

```
def SweepCos (t, t1, f0, f1)
```

- **function description:** compute cos sweep at given time t
- **input:**
  - $t$ : evaluate of sweep at time t
  - $t1$ : end time of sweep frequency range
  - $f0$ : start of frequency interval [ $f0, f1$ ] in Hz
  - $f1$ : end of frequency interval [ $f0, f1$ ] in Hz
- **output:** evaluation of cos sweep (in range -1..+1)

For examples on SweepCos see Examples (Ex) and TestModels (TM):

- [rigidRotor3DbasicBehaviour.py](#) (Ex), [rigidRotor3Drunup.py](#) (Ex), [objectGenericODE2Test.py](#) (TM)
- 

```
def FrequencySweep ( $t, t1, f0, f1$ )
```

- **function description:** frequency according to given sweep functions SweepSin, SweepCos
- **input:**
  - $t$ : evaluate of frequency at time t
  - $t1$ : end time of sweep frequency range
  - $f0$ : start of frequency interval [ $f0, f1$ ] in Hz
  - $f1$ : end of frequency interval [ $f0, f1$ ] in Hz
- **output:** frequency in Hz

For examples on FrequencySweep see Examples (Ex) and TestModels (TM):

- [objectGenericODE2Test.py](#) (TM)
- 

```
def SmoothStep ( $x, x0, x1, value0, value1$ )
```

- **function description:** step function with smooth transition from value0 to value1; transition is computed with cos function
- **input:**
  - $x$ : argument at which function is evaluated
  - $x0$ : start of step ( $f(x) = value0$ )
  - $x1$ : end of step ( $f(x) = value1$ )
  - $value0$ : value before smooth step
  - $value1$ : value at end of smooth step
- **output:** returns  $f(x)$

For examples on SmoothStep see Examples (Ex) and TestModels (TM):

- [leggedRobot .py](#) (Ex)
- 

```
def SmoothStepDerivative (x, x0, x1, value0, value1)
```

- **function description:** derivative of SmoothStep using same arguments
- **input:**
  - x*: argument at which function is evaluated
  - x0*: start of step ( $f(x) = \text{value0}$ )
  - x1*: end of step ( $f(x) = \text{value1}$ )
  - value0*: value before smooth step
  - value1*: value at end of smooth step
- **output:** returns  $d/dx(f(x))$

For examples on SmoothStepDerivative see Examples (Ex) and TestModels (TM):

- [leggedRobot .py](#) (Ex)
- 

```
def IndexFromValue (data, value, tolerance= 1e-7, assumeConstantSampleRate= False, rangeWarning= True)
```

- **function description:** get index from value in given data vector (numpy array); usually used to get specific index of time vector; this function is slow (linear search), if sampling rate is non-constant; otherwise set assumeConstantSampleRate=True!
- **input:**
  - data*: containing (almost) equidistant values of time
  - value*: e.g., time to be found in data
  - tolerance*: tolerance, which is accepted (default: tolerance=1e-7)
  - rangeWarning*: warn, if index returns out of range; if warning is deactivated, function uses the closest value
- **output:** index
- **notes:** to obtain the interpolated value of a time-signal array, use GetInterpolatedSignalValue() in exu-dyn.signalProcessing

---

```
def RoundMatrix (matrix, threshold= 1e-14)
```

- **function description:** set all entries in matrix to zero which are smaller than given threshold; operates directly on matrix

- **input:** matrix as np.array, threshold as positive value
  - **output:** changes matrix
- 

```
def ComputeSkewMatrix (v)
```

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

For examples on ComputeSkewMatrix see Examples (Ex) and TestModels (TM):

- [objectFFRFTest.py](#) (TM)
- 

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

- **function description:** check if input is list or array with according length; if *length*=-1, the length is not checked; raises Exception if the check fails
  - **input:**
    - vector*: a vector in np.array or list format
    - length*: desired length of vector; if *length*=-1, it is ignored
  - **output:** None
- 

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

- **function description:** check if input is list or array with according length and positive indices; if *length*=-1, the length is not checked; raises Exception if the check fails
  - **input:**
    - indexArray*: a vector in np.array or list format
    - length*: desired length of vector; if *length*=-1, it is ignored
  - **output:** None
- 

```
def FindObjectIndex (i, globalVariables)
```

- **function description:** simple function to find object index i within the local or global scope of variables
- **input:** i, the integer object number and globalVariables=locals()
- **example:**

```
FindObjectIndex(2, locals() ) #usually sufficient
FindObjectIndex(2, globals() ) #wider search
```

---

def **FindNodeIndex** (*i, globalVariables*)

- **function description:** simple function to find node index i within the local or global scope of variables
- **input:** i, the integer node number and globalVariables=locals()
- **example:**

```
FindObjectIndex(2, locals() ) #usually sufficient
FindObjectIndex(2, globals() ) #wider search
```

---

def **ShowOnlyObjects** (*mbs, objectNumbers= [], showOthers= False*)

- **function description:** function to hide all objects in mbs except for those listed in objectNumbers
  - **input:**
    - mbs*: mbs containing object
    - objectNumbers*: integer object number or list of object numbers to be shown; if empty list [], then all objects are shown
    - showOthers*: if True, then all other objects are shown again
  - **output:** changes all colors in mbs, which is NOT reversible
- 

def **HighlightItem** (*SC, mbs, itemNumber, itemType= exodyn.ItemType.Object, showNumbers= True*)

- **function description:** highlight a certain item with number itemNumber; set itemNumber to -1 to show again all objects
  - **input:**
    - mbs*: mbs containing object
    - itemNumbers*: integer object/node/etc number to be highlighted
    - itemType*: type of items to be highlighted
    - showNumbers*: if True, then the numbers of these items are shown
-

```
def UfsensorRecord (mbs, t, sensorNumbers, factors, configuration)
```

- **function description:** Internal SensorUserFunction, used in function AddSensorRecorder
- 

```
def AddSensorRecorder (mbs, sensorNumber, endTime, sensorsWritePeriod, sensorOutputSize= 3)
```

- **function description:** Add a SensorUserFunction object in order to record sensor output internally; this avoids creation of files for sensors, which can speedup and simplify evaluation in ParameterVariation and GeneticOptimization; values are stored internally in mbs.variables['sensorRecord'+str(sensorNumber)] where sensorNumber is the mbs sensor number

- **input:**

*mbs*: mbs containing object

*sensorNumber*: integer sensor number to be recorded

*endTime*: end time of simulation, as given in simulationSettings.timeIntegration.endTime

*sensorsWritePeriod*: as given in simulationSettings.solutionSettings.sensorsWritePeriod

*sensorOutputSize*: size of sensor data: 3 for Displacement, Position, etc. sensors; may be larger for RotationMatrix or Coordinates sensors; check this size by calling mbs.GetSensorValues(sensorNumber)

- **output:** adds an according SensorUserFunction sensor to mbs; returns new sensor number; during initialization a new numpy array is allocated in mbs.variables['sensorRecord'+str(sensorNumber)] and the information is written row-wise: [time, sensorValue1, sensorValue2, ...]
- **notes:** This sensor usually just passes through values of an existing sensor, while recording the values to a numpy array row-wise (time in first column, data in remaining columns)

For examples on AddSensorRecorder see Examples (Ex) and TestModels (TM):

- [ComputeSensitivitiesExample.py](#) (Ex)
- 

```
def LoadSolutionFile (fileName, safeMode= False, maxRows= -1, verbose= True)
```

- **function description:** read coordinates solution file (exported during static or dynamic simulation with option exu.SimulationSettings().solutionSettings.coordinatesSolutionFileName='...') into dictionary:

- **input:**

*fileName*: string containing directory and filename of stored coordinatesSolutionFile

*saveMode*: if True, it loads lines directly to load inconsistent lines as well; use this for huge files (>2GB); is slower but needs less memory!

*verbose*: if True, some information is written when importing file (use for huge files to track progress)

*maxRows*: maximum number of data rows loaded, if saveMode=True; use this for huge files to reduce loading time; set -1 to load all rows

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

For examples on LoadSolutionFile see Examples (Ex) and TestModels (TM):

- [addRevoluteJoint.py](#) (Ex), [beltDriveReevingSystem.py](#) (Ex), [fourBarMechanism3D.py](#) (Ex), [gyroStability.py](#) (Ex), [NGsolvePistonEngine.py](#) (Ex), ... , [ANCFbeltDrive.py](#) (TM), [ANCFgeneralContactCircle.py](#) (TM), [ConvexContactTest.py](#) (TM), ...
- 

---

def [\*\*NumpyInt8ArrayToString\*\*](#) (*npArray*)

- **function description**: simple conversion of int8 arrays into strings (not highly efficient, so use only for short strings)
- 

---

def [\*\*BinaryReadIndex\*\*](#) (*file, intType*)

- **function description**: read single Index from current file position in binary solution file
- 

---

def [\*\*BinaryReadReal\*\*](#) (*file, realType*)

- **function description**: read single Real from current file position in binary solution file
- 

---

def [\*\*BinaryReadString\*\*](#) (*file, intType*)

- **function description**: read string from current file position in binary solution file
- 

---

def [\*\*BinaryReadArrayIndex\*\*](#) (*file, intType*)

- **function description**: read Index array from current file position in binary solution file
- 

---

def [\*\*BinaryReadRealVector\*\*](#) (*file, intType, realType*)

- 
- **function description:** read Real vector from current file position in binary solution file
  - **output:** return data as numpy array, or False if no data read

---

```
def LoadBinarySolutionFile (fileName, maxRows= -1, verbose= True)
```

- **function description:** read BINARY coordinates solution file (exported during static or dynamic simulation with option exu.SimulationSettings().solutionSettings.coordinatesSolutionFileName='...') into dictionary
- **input:**
  - fileName*: string containing directory and filename of stored coordinatesSolutionFile
  - verbose*: if True, some information is written when importing file (use for huge files to track progress)
  - maxRows*: maximum number of data rows loaded, if saveMode=True; use this for huge files to reduce loading time; set -1 to load all rows
- **output:** dictionary with 'data': the matrix of stored solution vectors, 'columnsExported': a list with integer values showing the exported sizes [nODE2, nVel2, nAcc2, nODE1, nVel1, nAlgebraic, nData], 'nColumns': the number of data columns and 'nRows': the number of data rows

---

```
def RecoverSolutionFile (fileName, newFileName, verbose= 0)
```

- **function description:** recover solution file with last row not completely written (e.g., if crashed, interrupted or no flush file option set)
- **input:**
  - fileName*: string containing directory and filename of stored coordinatesSolutionFile
  - newFileName*: string containing directory and filename of new coordinatesSolutionFile
  - verbose*: 0=no information, 1=basic information, 2=information per row
- **output:** writes only consistent rows of file to file with name newFileName

---

```
def InitializeFromRestartFile (mbs, simulationSettings, restartFileName, verbose= True)
```

- **function description:** recover initial coordinates, time, etc. from given restart file
- **input:**
  - mbs*: MainSystem to be operated with
  - simulationSettings*: simulationSettings which is updated and shall be used afterwards for SolveDynamic(...) or SolveStatic(...)
  - restartFileName*: string containing directory and filename of stored restart file, as given in solutionSettings.restartFileName

*verbose*: False=no information, True=basic information

- **output**: modifies simulationSettings and sets according initial conditions in mbs
- 

```
def SetSolutionState (mbs, solution, row, configuration= exudyn.ConfigurationType.Current, sendRedrawSignal= True)
```

- **function description**: load selected row of solution dictionary (previously loaded with LoadSolutionFile) into specific state; flag sendRedrawSignal is only used if configuration = exudyn.ConfigurationType.Visualization
- 

```
def AnimateSolution (mbs, solution, rowIncrement= 1, timeout= 0.04, createImages= False, runLoop= False)
```

- **function description**: consecutively load the rows of a solution file and visualize the result

- **input**:

*mbs*: the system used for animation

*solution*: solution dictionary previously loaded with LoadSolutionFile; will be played from first to last row

*rowIncrement*: can be set larger than 1 in order to skip solution frames: e.g. rowIncrement=10 visualizes every 10th row (frame)

*timeout*: in seconds is used between frames in order to limit the speed of animation; e.g. use timeout=0.04 to achieve approximately 25 frames per second

*createImages*: creates consecutively images from the animation, which can be converted into an animation

*runLoop*: if True, the animation is played in a loop until 'q' is pressed in render window

- **output**: renders the scene in mbs and changes the visualization state in mbs continuously

For examples on AnimateSolution see Examples (Ex) and TestModels (TM):

- [NGsolvePistonEngine.py](#) (Ex), [rigidRotor3Dnutation.py](#) (Ex), [SliderCrank.py](#) (Ex), [slidercrankWithMassSpring.py](#) (Ex), [switchingConstraintsPendulum.py](#) (Ex), ... , [sliderCrankFloatingTest.py](#) (TM)
- 

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

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

- **input**:

*mbs*: MainSystem to be operated with

*showLoads*: toggle appearance of loads in mbs  
*showSensors*: toggle appearance of sensors in mbs  
*useItemNames*: if True, object names are shown instead of basic object types (Node, Load, ...)  
*useItemTypes*: if True, object type names (MassPoint, JointRevolute, ...) are shown instead of basic object types (Node, Load, ...); Note that Node, Object, is omitted at the beginning of itemName (as compared to theDoc.pdf); item classes become clear from the legend

- **output:** [nx, G, items] with nx being networkx, G the graph and item what is returned by nx.draw\_networkx\_labels(...)

For examples on DrawSystemGraph see Examples (Ex) and TestModels (TM):

- [fourBarMechanism3D.py](#) (Ex), [rigidBodyTutorial2.py](#) (Ex), [rigidBodyTutorial3.py](#) (Ex)
- 

```
def CreateTCPIPconnection (sendSize, receiveSize, IPaddress= '127.0.0.1', port= 52421, bigEndian= False, verbose= False)
```

– **function description:**

function which has to be called before simulation to setup TCP/IP socket (server) for sending and receiving data; can be used to communicate with other Python interpreters or for communication with MATLAB/Simulink

– **input:**

*sendSize*: number of double values to be sent to TCPIP client  
*receiveSize*: number of double values to be received from TCPIP client  
*IPaddress*: string containing IP address of client (e.g., '127.0.0.1')  
*port*: port for communication with client  
*bigEndian*: if True, it uses bigEndian, otherwise littleEndian is used for byte order

- **output:** returns information (TCPIPdata class) on socket; recommended to store this in mbs.sys['TCPIPObjecT']

– **example:**

```
mbs.sys['TCPIPObjecT'] = CreateTCPIPconnection(sendSize=3, receiveSize=2,
                                                bigEndian=True, verbose=True)
sampleTime = 0.01 #sample time in MATLAB! must be same!
mbs.variables['tLast'] = 0 #in case that exudyn makes finer steps than sample time
def PreStepUserFunction(mbs, t):
    if t >= mbs.variables['tLast'] + sampleTime:
        mbs.variables['tLast'] += sampleTime
        tcp = mbs.sys['TCPIPObjecT']
        y = TCPIPsendReceive(tcp, np.array([t, np.sin(t), np.cos(t)])) #time, torque
        tau = y[1]
        print('tau=', tau)
    return True
try:
    mbs.SetPreStepUserFunction(PreStepUserFunction)
```

```

#%+-----+
mbs.Assemble()
[...] #start renderer; simulate model
finally: #use this to always close connection, even in case of errors
    CloseTCPIPconnection(mbs.sys['TCPIPObj'])
#*****-----*
#the following settings work between Python and MATLAB-Simulink (client), and gives
    stable results(with only delay of one step):
# TCP/IP Client Send:
#   priority = 2 (in properties)
#   blocking = false
#   Transfer Delay on (but off also works)
# TCP/IP Client Receive:
#   priority = 1 (in properties)
#   blocking = true
#   Sourec Data type = double
#   data size = number of double in packer
#   Byte order = BigEndian
#   timeout = 10

```

For examples on CreateTCPIPconnection see Examples (Ex) and TestModels (TM):

- [TCPPIPxudynMatlab.py](#) (Ex)
- 

**def TCPIPsSendReceive** (*TCPIPObj*, *sendData*)

– **function description:**

call this function at every simulation step at which you intend to communicate with other programs via TCPIP; e.g., call this function in preStepUserFunction of a mbs model

– **input:**

*TCPIPObj*: the object returned by CreateTCPIPconnection(...)

*sendData*: numpy array containing data (double array) to be sent; must agree with sendSize

– **output:** returns array as received from TCPIP

– **example:**

```

mbs.sys['TCPIPObj']=CreateTCPIPconnection(sendSize=2, receiveSize=1, IPAddress='
    127.0.0.1')
y = TCPIPsSendReceive(mbs.sys['TCPIPObj'], np.array([1.,2.]))
print(y)

```

For examples on TCPIPsSendReceive see Examples (Ex) and TestModels (TM):

- [TCPPIPxudynMatlab.py](#) (Ex)
- 

**def CloseTCPIPconnection** (*TCPIPObj*)

- **function description:** close a previously created TCPIP connection

For examples on CloseTCPIPconnection see Examples (Ex) and TestModels (TM):

- [TCPIPexudynMatlab.py](#) (Ex)

### 5.23.1 CLASS TCPIPdata (in module utilities)

**class description:** helper class for CreateTCPIPconnection and for TCPIPsndReceive

# Chapter 6

## Theory and formulations

This section includes some material on notation, formulations and general theoretical backgrounds for EXUDYN. Note that the formulation of nodes, objects, markers, etc. is presented right at the subsection of each object, see [Section 7](#). Furthermore, overview on the system assembly and system equations of motion is given in [Section 10](#), solvers are described in [Section 11](#).

### 6.1 Notation

#### 6.1.1 Common types in item descriptions

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

- `float` ... a single-precision floating point number (note: in Python, 'float' is used also for double precision numbers; in EXUDYN, internally floats are single precision numbers especially for graphics objects and OpenGL)
- `Real` ... a double-precision floating point number (note: in Python this is also of type 'float')
- `UReal` ... same as `Real`, but may not be negative
- `PReal` ... same as `Real`, but must be positive, non-zero (e.g., step size may never be zero)
- `Index` ... deprecated, represents unsined integer, `UInt`
- `Int` ... a (signed) integer number, which converts to 'int' in Python, 'int' in C++
- `UInt` ... an unsigned integer number, which converts to 'int' in Python
- `PInt` ... an positive integer number ( $> 0$ ), which converts to 'int' in Python
- `NodeIndex`, `MarkerIndex`, ... ... a special (non-negative) integer type to represent indices of nodes, markers, ...; specifically, an unintentional conversion from one index type to the other is not possible (e.g., to convert `NodeIndex` to `MarkerIndex`); see [Section 4.1.1](#)
- `String` ... a string
- `ArrayIndex` ... a list of integer numbers (either list or in some cases numpy arrays may be allowed)
- `ArrayNodeIndex` ... a list of node indices
- `Bool` ... a boolean parameter: either `True` or `False` ('bool' in Python)
- `VObjectMassPoint`, `VObjectRigidBody`, `VObjectGround`, etc. ... represents the visualization object of the underlying object; 'V' is put in front of object name
- `BodyGraphicsData` ... see [Section 9.3](#)
- `Vector2D` ... a list or numpy array of 2 real numbers
- `Vector3D` ... a list or numpy array of 3 real numbers

- `Vector'X'D` ... a list or numpy array of 'X' real numbers
- `Float4` ... a list of 4 float numbers
- `Vector` ... a list or numpy array of real numbers (length given by according object)
- `NumpyVector` ... a 1D numpy array with real numbers (size given by according object); similar as `Vector`, but not accepting list
- `Matrix3D` ... a list of lists or numpy array with  $3 \times 3$  real numbers
- `Matrix6D` ... a list of lists or numpy array with  $6 \times 6$  real numbers
- `NumpyMatrix` ... a 2D numpy array (matrix) with real numbers (size given by according object)
- `NumpyMatrixI` ... a 2D numpy array (matrix) with integer numbers (size given by according object)
- `MatrixContainer` ... a versatile representation for dense and sparse matrices, see [Section 6.1.2](#) below
- `PyFunctionGraphicsData` ... a user function providing `GraphicsData`, see the user function description of the according object
- `PyFunctionMbsScalar` ... a user function for the according object; the name is chosen according to the interface (arguments containing scalars, vectors, etc.); see the according user function description

## 6.1.2 MatrixContainer

The `MatrixContainer` is a versatile representation for dense and sparse matrices. Some functionality:

- Create empty `MatrixContainer` with `mc = MatrixContainer()`
- Set with dense `pyArray` (a numpy array) (if `useDenseMatrix` is set False, it converts into a sparse matrix, which may speed up further computations for sparse matrices!): `mc.SetWithDenseMatrix(pyArray, bool useDenseMatrix = True)`
- Set with sparse `pyArray` (a numpy array), which has 3 columns and according rows containing the sparse triplets (`row, col, value`) describing the sparse matrix; Use `CSRtoScipySparseCSR(...)` and `ScipySparseCSRtoCSR(...)` to convert between this format and the scipy csr format; if `useDenseMatrix` is set True, it converts into a dense matrix, which may slow down computations: `mc.SetWithSparseMatrixCSR(numberOfRowsInit, numberOfRowsColumnsInit, pyArray, useDenseMatrix = False)`
- Convert matrix into dense format (slow, but helpful for debug): `mc.Convert2DenseMatrix()`
- Get internal stored object: `mc.GetPythonObject()`
- **Automatic type conversion:** if a function or class requests a `MatrixContainer`, such as `ObjectGenericODE2`, there are automatic type conversions from:
  - empty list: []
  - numpy array, e.g.: `np.array([[1,2],[3,4]])`
  - list of lists describing matrix, e.g. (slow for larger matrices!): `[[1,2],[3,4]]`

## 6.1.3 States and coordinate attributes

The following subscripts are used to define configurations of a quantity, e.g., for a vector of displacement coordinates `q`:

- `qconfig` ... `q` in any configuration
- `qref` ... `q` in reference configuration, e.g., reference coordinates: `cref`
- `qini` ... `q` in initial configuration, e.g., initial displacements: `uini`
- `qcur` ... `q` in current configuration
- `qvis` ... `q` in visualization configuration
- `qstart of step` ... `q` in start of step configuration

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

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

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

$$\mathbf{a} \times \mathbf{b} = \tilde{\mathbf{a}} \mathbf{b} = -\tilde{\mathbf{b}} \mathbf{a}, \quad (6.1)$$

in which  $\sim$  transforms a vector  $\mathbf{a}$  into a skew-symmetric matrix  $\tilde{\mathbf{a}}$ . The inverse operation is denoted as  $\text{vec}$ , resulting in  $\text{vec}(\tilde{\mathbf{a}}) = \mathbf{a}$ .

For the length of a vector we often use the abbreviation

$$\|\mathbf{a}\| = \sqrt{\mathbf{a}^T \mathbf{a}}. \quad (6.2)$$

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

$$\mathbf{A} = \text{diag}(\mathbf{a}) = \begin{bmatrix} x & 0 & 0 \\ 0 & y & 0 \\ 0 & 0 & z \end{bmatrix} \quad (6.3)$$

#### 6.1.4 Symbols in item equations

The following table contains the common notation:

python name (or description)	symbol	description
displacement coordinates ( <b>ODE2</b> )	$\mathbf{q} = [q_0, \dots, q_n]^T$	vector of $n$ displacement based coordinates in any configuration; used for second order differential equations
rotation coordinates ( <b>ODE2</b> )	$\psi = [\psi_0, \dots, \psi_\eta]^T$	vector of $\eta$ <b>rotation based coordinates</b> in any configuration; these coordinates are added to reference rotation parameters to provide the current rotation parameters; used for second order differential equations
coordinates ( <b>ODE1</b> )	$\mathbf{y} = [y_0, \dots, y_n]^T$	vector of $n$ coordinates for first order ordinary differential equations ( <b>ODE1</b> ) in any configuration
algebraic coordinates	$\mathbf{z} = [z_0, \dots, z_m]^T$	vector of $m$ algebraic coordinates if not Lagrange multipliers in any configuration
Lagrange multipliers	$\lambda = [\lambda_0, \dots, \lambda_m]^T$	vector of $m$ Lagrange multipliers (=algebraic coordinates) in any configuration
data coordinates	$\mathbf{x} = [x_0, \dots, x_l]^T$	vector of $l$ data coordinates in any configuration
python name: OutputVariable	symbol	description
Coordinate	$\mathbf{c} = [c_0, \dots, c_n]^T$	coordinate vector with $n$ generalized coordinates $c_i$ in any configuration; the letter $c$ is used both for <b>ODE1</b> and <b>ODE2</b> coordinates
Coordinate_t	$\dot{\mathbf{c}} = [c_0, \dots, c_n]^T$	time derivative of coordinate vector

Displacement	${}^0\mathbf{u} = [u_0, u_1, u_2]^T$	global displacement vector with 3 displacement coordinates $u_i$ in any configuration; in 1D or 2D objects, some of there coordinates may be zero
Rotation	$[\varphi_0, \varphi_1, \varphi_2]_{\text{config}}^T$	vector with 3 components of the Euler angles in xyz-sequence ( ${}^{0b}\mathbf{A}_{\text{config}} =: \mathbf{A}_0(\varphi_0) \cdot \mathbf{A}_1(\varphi_1) \cdot \mathbf{A}_2(\varphi_2)$ ), recomputed from rotation matrix
Identity matrix	$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & 1 \end{bmatrix}$	the identity matrix, very often $\mathbf{I} = \mathbf{I}_{3 \times 3}$ , the $3 \times 3$ identity matrix
Identity transformation	${}^{0b}\mathbf{I}_{3 \times 3} = \mathbf{I}_{3 \times 3}$	converts body-fixed into global coordinates, e.g., ${}^0\mathbf{x} = {}^{0b}\mathbf{I}_{3 \times 3} {}^b\mathbf{x}$ , thus resulting in ${}^0\mathbf{x} = {}^b\mathbf{x}$ in this case
RotationMatrix	${}^{0b}\mathbf{A} = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix}$	a 3D rotation matrix, which transforms local (e.g., body $b$ ) to global coordinates (0): ${}^0\mathbf{x} = {}^{0b}\mathbf{A} {}^b\mathbf{x}$
RotationMatrixX	${}^{01}\mathbf{A}_0(\theta_0) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_0) & -\sin(\theta_0) \\ 0 & \sin(\theta_0) & \cos(\theta_0) \end{bmatrix}$	rotation matrix for rotation around X axis (axis 0), transforming a vector from frame 1 to frame 0
RotationMatrixY	${}^{01}\mathbf{A}_1(\theta_1) = \begin{bmatrix} \cos(\theta_0) & 0 & \sin(\theta_0) \\ 0 & 1 & 0 \\ -\sin(\theta_0) & 0 & \cos(\theta_0) \end{bmatrix}$	rotation matrix for rotation around X axis (axis 0), transforming a vector from frame 1 to frame 0
RotationMatrixZ	${}^{01}\mathbf{A}_2(\theta_2) = \begin{bmatrix} \cos(\theta_0) & -\sin(\theta_0) & 0 \\ \sin(\theta_0) & \cos(\theta_0) & 0 \\ 0 & 0 & 1 \end{bmatrix}$	rotation matrix for rotation around X axis (axis 0), transforming a vector from frame 1 to frame 0
Position	${}^0\mathbf{p} = [p_0, p_1, p_2]^T$	global position vector with 3 position coordinates $p_i$ in any configuration
Velocity	${}^0\mathbf{v} = {}^0\dot{\mathbf{u}} = [v_0, v_1, v_2]^T$	global velocity vector with 3 displacement coordinates $v_i$ in any configuration
AngularVelocity	${}^0\boldsymbol{\omega} = [\omega_0, \dots, \omega_2]^T$	global angular velocity vector with 3 coordinates $\omega_i$ in any configuration
Acceleration	${}^0\mathbf{a} = {}^0\ddot{\mathbf{u}} = [a_0, a_1, a_2]^T$	global acceleration vector with 3 displacement coordinates $a_i$ in any configuration
AngularAcceleration	${}^0\boldsymbol{\alpha} = {}^0\dot{\boldsymbol{\omega}} = [\alpha_0, \dots, \alpha_2]^T$	global angular acceleration vector with 3 coordinates $\alpha_i$ in any configuration
VelocityLocal	${}^b\mathbf{v} = [v_0, v_1, v_2]^T$	local (body-fixed) velocity vector with 3 displacement coordinates $v_i$ in any configuration
AngularVelocityLocal	${}^b\boldsymbol{\omega} = [\omega_0, \dots, \omega_2]^T$	local (body-fixed) angular velocity vector with 3 coordinates $\omega_i$ in any configuration
Force	${}^0\mathbf{f} = [f_0, \dots, f_2]^T$	vector of 3 force components in global coordinates
Torque	${}^0\boldsymbol{\tau} = [\tau_0, \dots, \tau_2]^T$	vector of 3 torque components in global coordinates

python name: input to nodes, markers, etc.	symbol	description
referenceCoordinates	$\mathbf{c}_{\text{ref}} = [c_0, \dots, c_n]_{\text{ref}}^T = [c_{\text{Ref},0}, \dots, c_{\text{Ref},n}]_{\text{ref}}^T$	$n$ coordinates of reference configuration (can usually be set at initialization of nodes)
initialCoordinates	$\mathbf{c}_{\text{ini}}$	initial coordinates with generalized or mixed displacement/rotation quantities (can usually be set at initialization of nodes)
reference point	${}^0\mathbf{r} = [r_0, r_1, r_2]^T$	reference point of body, e.g., for rigid bodies or floating frame of reference formulation ( <a href="#">FFRF</a> ) bodies, in any configuration; NOTE: for ANCF elements, ${}^0\mathbf{r}$ is used for the position vector to the beam centerline
localPosition	${}^b\mathbf{b} = [{}^b b_0, {}^b b_1, {}^b b_2]^T$	local (body-fixed) position vector with 3 position coordinates $b_i$ in any configuration, measured relative to reference point; NOTE: for rigid bodies, ${}^0\mathbf{p} = {}^0\mathbf{r} + {}^{0b}\mathbf{A} {}^b\mathbf{b}$ ; localPosition is used for definition of body-fixed local position of markers, sensors, COM, etc.

### 6.1.5 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. The current position vector of a point node is computed from the reference position plus the current displacement, reading

$$\mathbf{p}_{\text{cur}} = \mathbf{p}_{\text{ref}} + \mathbf{u}_{\text{cur}} \quad (6.4)$$

In the same way rotation parameters are computed from,

$$\theta_{\text{cur}} = \theta_{\text{ref}} + \psi_{\text{cur}} \quad (6.5)$$

which are based on reference quantities plus displacements or *changes*. Note that these changes are additive, even for rotation parameters. Needless to say,  $\psi_{\text{cur}}$  do not represent rotation parameters, while  $\theta_{\text{ref}}$  should be chosen such that they represent the orientation of a node in reference configuration. The necessity for reference coordinates originates from finite elements, which usually split nodal position into displacements and reference position. However, we also use the reference position here in order to define joints, e.g., using the utility function `AddRevoluteJoint(...)`.

Against to the splitting of positions, displacements and velocities (and most other quantities) are not having this reference part!

### 6.1.6 Reference point

In contrast to the reference position or reference coordinates, the reference point is mainly used for objects, e.g., rigid bodies. The reference point is the position of the underlying (rigid body) node, while we can compute the position of any point on the body (or on the mass point). The reference point is also the origin of the co-rotating (reference) frame with the body-fixed coordinate system.

The same concept is also used for [FFRF](#) objects. In most cases, reference points are denoted by  $\mathbf{r}$ .

### 6.1.7 Coordinate Systems

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

$${}^0\mathbf{u} \quad (6.6)$$

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

$${}^{m1}\mathbf{u} \quad (6.7)$$

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  $m1$
- ${}^{J0}\mathbf{u}$  ... local coordinates of joint 0, related to marker  $m0$
- ${}^{J1}\mathbf{u}$  ... local coordinates of joint 1, related to marker  $m1$

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

$${}^0\mathbf{u} = {}^{0,m0}\mathbf{A} {}^{m0}\mathbf{u} \quad (6.8)$$

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

### 6.1.8 Integration Points

For several tasks, especially for finite elements and contact, different integration rules are used, which are summarized here. The interval of all integration rules is  $\in [-1, 1]$ , thus giving a total sum for integration weights of 2. The points  $\xi_{ip}$  and weights  $w_{ip}$  for Gauss rules read:

type/order	point 0	point 1	point 2	point 3
Gauss 1	0			
Gauss 3	$-\sqrt{1/3}$	$\sqrt{1/3}$		
Gauss 5	$-\sqrt{3/5}$	0	$\sqrt{3/5}$	
Gauss 7	$-\sqrt{3/7 + \sqrt{120}/35}$	$-\sqrt{3/7 - \sqrt{120}/35}$	$\sqrt{3/7 - \sqrt{120}/35}$	$\sqrt{3/7 + \sqrt{120}/35}$
type/order	weight 0	weight 1	weight 2	weight 3
Gauss 1	2			
Gauss 3	1	1		
Gauss 5	5/9	8/9	5/9	
Gauss 7	$1/2 - 5/(3\sqrt{120})$	$1/2 + 5/(3 * \sqrt{120})$	$1/2 + 5/(3 * \sqrt{120})$	$1/2 - 5/(3 * \sqrt{120})$

The points  $\xi_{ip}$  and weights  $w_{ip}$  for Lobatto rules read:

type/order	point 0	point 1	point 2	point 3
Lobatto 2	-1	1		
Lobatto 4	-1	0	1	
Lobatto 6	-1	$-\sqrt{1/5}$	$\sqrt{1/5}$	1

type/order	weight 0	weight 1	weight 2	weight 3
Lobatto 2	1	1		
Lobatto 4	1/3	4/3	1/3	
Lobatto 6	1/6	5/6	5/6	1/6

Further integration rules can be found in the C++ code of `ExUDYN`, see file `BasicLinalg.h`.

## 6.2 Model order reduction and component mode synthesis

This section describes the process how to create general flexible multibody system models using the floating frame of reference formulation with model order reduction (here also denoted as component mode synthesis ([CMS](#))). The according object `ObjectFFRFreducedOrder` is described in [Section 7.3.3](#).

### 6.2.1 Eigenmodes

This section will describe the computation of eigenmodes using FEMinterface.

The FEMinterface in the module FEM has various functionality to import finite element meshes from finite element software. We create a FEMinterface by means of

```
fem = FEMinterface()
```

which allows us to use the variable `fem` from now.

Meshes can be imported from NETGEN/NGsolve ([Section 5.4.8](#)), Abaqus (see [Section 5.4.8](#) and other sections related to ABAQUS), ANSYS (see [Section 5.4.8](#) and other sections related to ANSYS). The import procedure, which can also be done manually, needs to include `massMatrix M` and `stiffnessMatrix K` from any finite element model. Note that many functions are based on the requirement that nodes are 3D displacement-based nodes, without rotation or other coordinates.

For any functionality with `ObjectFFRFreducedOrder` and for the computation of Hurty-Craig-Bampton modes as described in the next section, `nodes` are required. Finally, `elements` need to be included for visualization, and a surface needs to be reconstructed from the element connectivity, which is available for tetrahedral and hexahedral elements for most import functions.

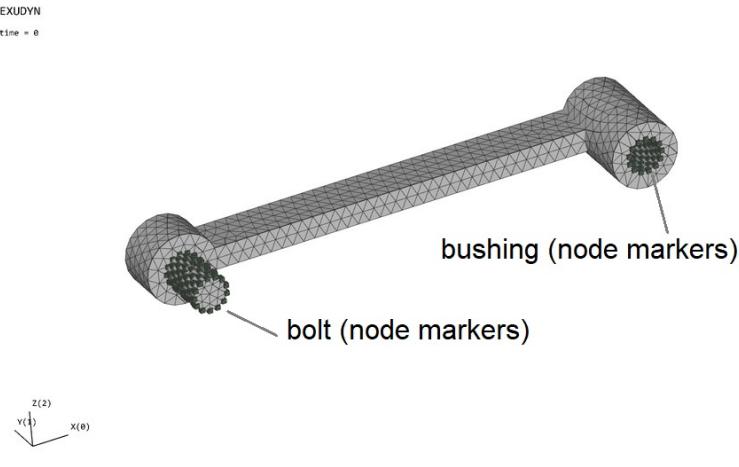


Figure 6.1: Test model and mesh for hinge created with Netgen (linear tetrahedral elements).

As an example, we consider a part denoted as 'hinge' in the following, see Fig. 6.1. The test example can be found in `Examples/NGsolveCMSTutorial.py` with lots of additional features.

After import of mass and stiffness matrix, eigenmodes and eigenfrequencies can be computed using `fem.ComputeEigenFrequencies(...)`, which computes the quantities `fem.modeBasis` and `fem.eigenValues`. The eigenvalues in Hz can be retrieved also with the function `fem.GetEigenFrequenciesHz()`. The function `fem.ComputeEigenFrequencies(...)` is available for dense and sparse matrices, and uses `scipy.linalg` to compute eigenvalues of the linear, undamped mechanical system

$$\mathbf{M}\ddot{\mathbf{q}}(t) + \mathbf{K}\dot{\mathbf{q}}(t) = \mathbf{f}(t) . \quad (6.9)$$

Here, the total number of coordinates of the system is  $n$ , thus having the vector of system coordinates  $\mathbf{q} \in \mathbb{R}^n$ , vector of applied forces  $\mathbf{f} \in \mathbb{R}^n$ , mass matrix  $\mathbf{M} \in \mathbb{R}^{n \times n}$  and stiffness matrix  $\mathbf{K} \in \mathbb{R}^{n \times n}$ . If we are interested in free vibrations of the system, without any boundary conditions or interconnections to other bodies, Eq. (6.9) can be converted to a generalized eigenvalue problem. Using the approach  $\dot{\mathbf{q}}(t) = \mathbf{v}e^{i\omega t}$  in Eq. (6.9), and thus  $\ddot{\mathbf{q}}(t) = -\omega^2\mathbf{q}(t)$ , we obtain

$$[(-\omega^2\mathbf{M} + \mathbf{K})\mathbf{v}]e^{i\omega t} = 0. \quad (6.10)$$

Assuming that Eq. (6.10) is valid for all times, the **generalized eigenvalue problem** follows that

$$(-\omega^2\mathbf{M} + \mathbf{K})\mathbf{v} = 0, \quad (6.11)$$

which can be rewritten as

$$\det(-\omega^2\mathbf{M} + \mathbf{K}) = 0, \quad (6.12)$$

and which defines the eigenvalues  $\omega_i^2$  of the linear system, where  $i \in \{0, \dots, n-1\}$ . Note that in this case, the eigenvalues are the squared eigenfrequencies (in rad/s). We can use eigenvalue algorithms to compute the eigenvalues  $\omega_i^2$  and according eigenvectors  $\mathbf{v}_i$  from Python. The function `fem.ComputeEigenmodes(...)` uses `eigh(...)` from `scipy.linalg` in the dense matrix mode, and in the sparse mode `eigsh(...)` from `scipy.sparse.linalg`, the latter being restricted to pure symmetric matrices. Using special shift-inverted techniques in `eigsh(...)`, it performs much better than standard settings. However, you may tune your specific eigenvalue problem by modifying the solver procedure (just copy that function and adjust to your needs). As an output, we obtain the smallest `nModes` eigenvectors (=eigenmodes)<sup>1</sup> of the system. Here, we will also use synonymously the terms ‘eigenmodes’ and ‘normal modes’, which result from an eigenvalue/eigenvector computation using certain (or even no) boundary conditions.

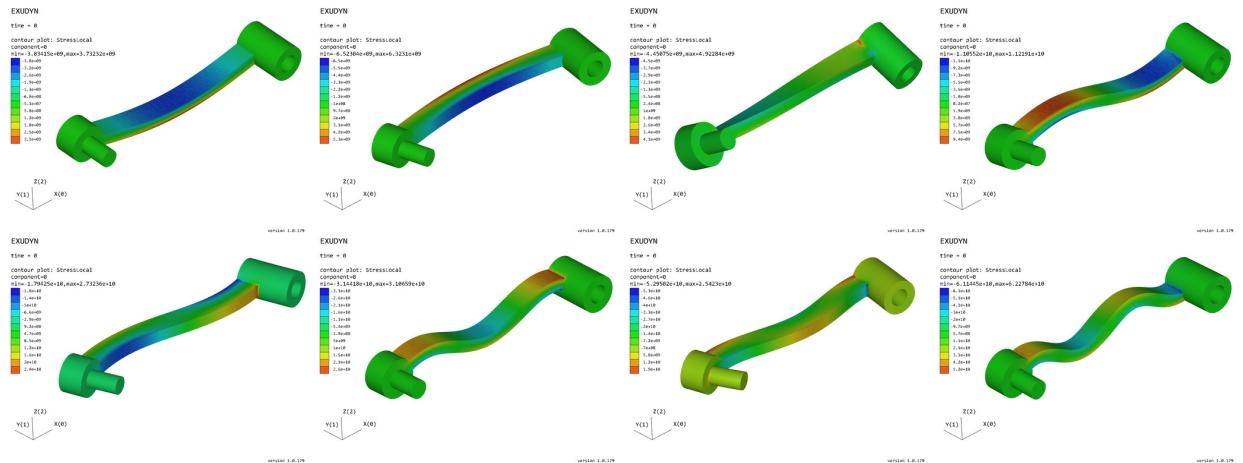


Figure 6.2: Lowest 8 free-free modes for hinge finite element model, contour plot for  $xx$ -stress component.

Clearly, if there are no supports included in the stiffness matrix, the resulting eigenmodes will contain 6 rigid body modes and we will also call this case for the computation of eigenmodes the ‘free-free’ case, in analogy to a simply supported beam. This rigid body modes, which are usually not needed (=unwanted) in the succeeding computation, can be excluded with an according option in

```
fem.ComputeEigenFrequencies(excludeRigidBodyModes = ...)
```

<sup>1</sup>Eigenvectors are the result of the eigenvalue algorithm, such as the QR algorithm. The mechanical interpretation of eigenvectors are eigenmodes, that can be visualized as shown in the figures of this section.

For our test example, 8 eigenmodes are shown in Fig. 6.2, where the 6 rigid body modes have been excluded (so in total, 14 eigenvectors were computed). The 8 eigenfrequencies for the chosen coarse mesh with mesh size  $h = 0.01$  and 1216 nodes result as

$$f_{0.7} = [671.59, 707.17, 1298.50, 1929.97, 1971.76, 3141.47, 3595.34, 4317.51] \text{Hz} \quad (6.13)$$

Note, that a computation with a finer mesh, using mesh size  $h = 0.002$  and 100224 nodes, leads to significantly different eigenfrequencies, starting with  $f_0 = 371.50 \text{ Hz}$ . This shows that quadratic finite elements would be more appropriate for this case.

After the computation of modes, it is always a good idea to visualize and/or animate these modes. We can do this, using the function `AnimateModes(...)` available in `exudyn.interactive`, which allows us to inspect and animate modes and to create animations for these modes, see the mentioned example.

Clearly, the free-free modes in Fig. 6.2 are not well suited for the modeling of the deformations within the hinge, if the bolt and the bushing shall be fixed to ground or to another part. Therefore, we can use modes based on ideas of Hurty [24] and Craig-Bampton [2], as shown in the following.

### 6.2.2 Hurty-Craig-Bampton modes

This section will describe the computation of static and eigen (normal) modes using FEMinterface. The theory is based on Hurty [24] and Craig-Bampton [2], but often only attributed to Craig-Bampton. Furthermore, boundaries are also called interfaces<sup>2</sup>, as they either represent surface sections of our finite element model which are connected to the ground or they represent interfaces to joints and are connected to other bodies.

The computation of so-called static and normal modes follows a simple concept based on finite element mass and stiffness matrices. The final goal of the computation of modes is to approximate the solution  $\mathbf{q} \in \mathbb{R}^n$  by means of a reduction basis  $\Psi \in \mathbb{R}^{n \times m}$  and a reduced set of coordinates  $\mathbf{p} \in \mathbb{R}^m$ , for which we assume  $m \ll n$ .

In order to include boundary/interface effects, we separate our nodes and the nodal coordinates into

- a) boundary nodes  $\mathbf{q}_b \in \mathbb{R}^{n_b}$  and
- b) internal or inner nodes  $\mathbf{q}_i \in \mathbb{R}^{n_i}$ .

We assume that internal nodes are not exposed to boundary/interface conditions or to forces.

Therefore, we may rewrite Eq. (6.9) as follows

$$\begin{bmatrix} \mathbf{M}_{bb} & \mathbf{M}_{bi} \\ \mathbf{M}_{ib} & \mathbf{M}_{ii} \end{bmatrix} \begin{bmatrix} \ddot{\mathbf{q}}_b \\ \ddot{\mathbf{q}}_i \end{bmatrix} + \begin{bmatrix} \mathbf{K}_{bb} & \mathbf{K}_{bi} \\ \mathbf{K}_{ib} & \mathbf{K}_{ii} \end{bmatrix} \begin{bmatrix} \mathbf{q}_b \\ \mathbf{q}_i \end{bmatrix} = \begin{bmatrix} \mathbf{f}_b \\ \mathbf{0} \end{bmatrix} \quad (6.14)$$

or, equivalently,

$$\mathbf{M}_{bb}\ddot{\mathbf{q}}_b + \mathbf{M}_{bi}\ddot{\mathbf{q}}_i + \mathbf{K}_{bb}\mathbf{q}_b + \mathbf{K}_{bi}\mathbf{q}_i = \mathbf{f}_b \quad (6.15)$$

$$\mathbf{M}_{ib}\ddot{\mathbf{q}}_b + \mathbf{M}_{ii}\ddot{\mathbf{q}}_i + \mathbf{K}_{ib}\mathbf{q}_b + \mathbf{K}_{ii}\mathbf{q}_i = \mathbf{0}. \quad (6.16)$$

A pure static condensation follows from Eq. (6.16) with the assumption that inertia terms are neglected, leading to the static result for internal nodes,

$$\mathbf{q}_{i,\text{stat}} = -\mathbf{K}_{ii}^{-1}\mathbf{K}_{ib}\mathbf{q}_b. \quad (6.17)$$

A pure static condensation, also denoted as Guyan-Irons method, keeps boundary coordinates but removes all internal modes, using the approximation

$$\begin{bmatrix} \mathbf{q}_b \\ \mathbf{q}_i \end{bmatrix} \approx \begin{bmatrix} \mathbf{I} \\ -\mathbf{K}_{ii}^{-1}\mathbf{K}_{ib} \end{bmatrix} \mathbf{q}_b = \Psi^{GI} \mathbf{q}_b, \quad (6.18)$$

---

<sup>2</sup>Here, and in the description of various Python functions, we will use boundary and interface often synonymously, as flexible bodies can be either connected to ground in the sense of a classical ‘support-type’ boundary condition, or they can represent the boundary of the flexible body as an interface to joints (via markers).

which leads to no approximations ('exact') results for the static case, but poor performance in highly dynamic problems.

Significant improvement result from the Hurty-Craig-Bampton method, which adds eigenmodes of the internal coordinates (internal nodes). We assume that  $\Psi_{ii}$  is the matrix of eigenvectors as a solution to the eigenvalue problem

$$(-\omega^2 \mathbf{M}_{ii} + \mathbf{K}_{ii}) \mathbf{v} = \mathbf{0}, \quad (6.19)$$

Hereafter, we will only keep the lowest (or other appropriate)  $m$  eigenmodes in a reduced eigenmode matrix,

$$\Psi_{ii}^{(red)} = [\Psi_{ii,0}, \dots, \Psi_{ii,m-1}] \quad (6.20)$$

Combining these 'fixed-fixed' eigenvectors with the Guyan-Irons reduction (6.18), we obtain the Hurty-Craig-Bampton modes as

$$\begin{bmatrix} \mathbf{q}_b \\ \mathbf{q}_i \end{bmatrix} \approx \begin{bmatrix} \mathbf{I} \\ -\mathbf{K}_{ii}^{-1} \mathbf{K}_{ib} \end{bmatrix} \mathbf{q}_b + \begin{bmatrix} \mathbf{0} \\ \Psi_{r,i} \end{bmatrix} \mathbf{p}_r, \quad (6.21)$$

or in matrix form

$$\begin{bmatrix} \mathbf{q}_b \\ \mathbf{q}_i \end{bmatrix} \approx \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{K}_{ii}^{-1} \mathbf{K}_{ib} & \Psi_{r,i} \end{bmatrix} \begin{bmatrix} \mathbf{q}_b \\ \mathbf{p}_r \end{bmatrix} = \Psi^{HCB} \mathbf{p}^{HCB}. \quad (6.22)$$

The disadvantage of Eq. (6.22) is evident by the fact that there may be a large number of boundary/interface nodes, leading to a huge number of static modes (100s or 1000s) and thus making the model reduction inefficient. Therefore, we can switch to other interfaces, as described in the following.

### 6.2.2.1 Definition of RBE2 interfaces

A powerful extension, which is available in many finite element as well as flexible multibody codes, is the definition of special boundary/interface conditions, based on pure rigid body motion. The so-called RBE2 boundaries are defined such that they are firmly connected to a rigid frame, thus the boundary or interface can only undergo rigid body motion. The advantage of this procedure is that, in comparison to Eq. (6.22), the number of boundary/interface modes is given by 6 *rigid body* modes, which allow simple integration into standard joints of multibody systems, e.g., the `GenericJoint`. The disadvantage is that such modes usually lead to artificial stiffening and stresses close to the boundary.

### 6.2.2.2 Computation of Hurty-Craig-Bampton modes with RBE2 interfaces

In the following section, we show the procedure for the computation of static modes for the RBE2 rigid-body interfaces.

First, we use the index  $j$  here as a node index, having the clear correspondence to the coordinate index  $i$ , that node  $j$  has coordinates  $[3 \cdot j, 3 \cdot j + 1, 3 \cdot j + 2]$ . Furthermore, nodes are split into boundary and internal nodes, which then leads to according internal and boundary coordinates. We shall note that this sorting is never done in the finite element model or matrices, but just some indexing (referencing) lists are generated and used throughout, using valuable features of `numpy.linalg` and `scipy.sparse`.

For a certain boundary node set  $B = [j_0, j_1, j_2, \dots] \in \mathbb{N}^{n_b}$  with certain  $n_b$  node indices  $j_0, \dots$ , we define one boundary set. The following transformations need to be performed for every set of boundary node lists. We also assume that weighting of all boundary nodes is equal, which may not be appropriate in all cases.

If we assume that there may only occur rigid body translation and rotation for the whole boundary node set, which is according to the idea of so-called RBE2 boundary conditions, it follows that the translation of all boundary nodes is given by

$$\mathbf{T}_t = [\mathbf{I} \ \mathbf{I} \ \dots \ \mathbf{I}]^T \in \mathbb{R}^{3n_b \times 3} \quad (6.23)$$

with  $\mathbf{I} \in \mathbb{R}^{3 \times 3}$  identity matrices. The nodal translation coordinates on boundary  $B$  are denoted as  $\mathbf{q}_{B,t} \in \mathbb{R}^3$ . The translation of the boundary/interface is mapped to the boundary coordinates as follows (assuming only one boundary  $B$ ),

$$\mathbf{q}_{b,t} = \mathbf{T}_t \mathbf{q}_{B,t} \quad (6.24)$$

The nodal rotation coordinates on boundary  $B$  are denoted as  $\mathbf{q}_{B,r} \in \mathbb{R}^3$ . The rotation of the boundary/interface is mapped to the boundary coordinates as follows (assuming only one boundary  $B$ ),

$$\mathbf{q}_{b,r} = \mathbf{T}_r \mathbf{q}_{B,r} \quad (6.25)$$

The computation of matrix  $\mathbf{T}_r$  is more involved. It is based on nodal (reference) position vectors  $\mathbf{r}_j^{(0)}$ ,  $j \in B$ , the midpoint of all boundary nodes,

$$\mathbf{r}^{(m)} = \frac{1}{n_b} \sum_{j=0}^{n_b-1} \mathbf{r}_j^{(0)} \quad (6.26)$$

and the position relative to the midpoint, denoted as

$$\mathbf{r}_j = \mathbf{r}_j^{(0)} - \mathbf{r}^{(m)} . \quad (6.27)$$

The transformation for rotation follows from

$$\mathbf{T}_r = [\tilde{\boldsymbol{\Omega}}_x \mathbf{r}_{j_0} \ \tilde{\boldsymbol{\Omega}}_y \mathbf{r}_{j_0} \ \tilde{\boldsymbol{\Omega}}_z \mathbf{r}_{j_0} \ \tilde{\boldsymbol{\Omega}}_x \mathbf{r}_{j_1} \ \tilde{\boldsymbol{\Omega}}_y \mathbf{r}_{j_1} \ \tilde{\boldsymbol{\Omega}}_z \mathbf{r}_{j_1} \dots]^T \in \mathbb{R}^{3n_b \times 3} \quad (6.28)$$

with the special tensors, representing rotation about (x,y,z)-axes,

$$\tilde{\boldsymbol{\Omega}}_x = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}, \quad \tilde{\boldsymbol{\Omega}}_y = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix}, \quad \tilde{\boldsymbol{\Omega}}_z = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} . \quad (6.29)$$

The total nodal coordinates at the boundary, representing translations and rotations, follow as

$$\mathbf{q}_B = \begin{bmatrix} \mathbf{q}_{B,t} \\ \mathbf{q}_{B,r} \end{bmatrix} , \quad (6.30)$$

and the transformation matrix for the translation and rotation simply reads

$$\mathbf{T} = [\mathbf{T}_t \ \mathbf{T}_r] \in \mathbb{R}^{3n_b \times 6} , \quad (6.31)$$

which provides the total mapping of boundary rigid body motion

$$\mathbf{q}_b = \mathbf{T} \mathbf{q}_B , \quad (6.32)$$

which is the sum of translation and rotation.

As an example, having the boundary nodes sorted for two boundary node set  $B_0$  and  $B_1$ , we obtain the following transformation for the Hurty-Craig-Bampton method with only 6 modes per boundary node set,

$$\begin{bmatrix} \mathbf{q}_b \\ \mathbf{q}_i \end{bmatrix} \approx \begin{bmatrix} \mathbf{T}_0 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{T}_1 & \mathbf{0} \\ -\mathbf{K}_{ii}^{-1} \mathbf{K}_{ib} \begin{bmatrix} \mathbf{T}_0 \\ \mathbf{0} \end{bmatrix} & -\mathbf{K}_{ii}^{-1} \mathbf{K}_{ib} \begin{bmatrix} \mathbf{0} \\ \mathbf{T}_1 \end{bmatrix} & \mathbf{\Psi}_{r,i} \end{bmatrix} \begin{bmatrix} \mathbf{q}_{B_0} \\ \mathbf{q}_{B_1} \\ \mathbf{p}_r \end{bmatrix} . \quad (6.33)$$

with the new boundary node vector  $\mathbf{q}_b = [\mathbf{q}_{B_0}^T \ \mathbf{q}_{B_1}^T]^T$ .

**Notes:**

- The inverse  $\mathbf{K}_{ii}^{-1}$  is not computed, but this matrix is LU-factorized using sparse techniques.
- The factorization only needs to be applied to six vectors for every relevant boundary node set.
- One set of boundary nodes can be omitted from the final static modes in Eq. (6.33), because keeping all boundary modes, would introduce six rigid body motions to our mode basis, what is usually not wanted nor needed.

Using again the examples given in Fig. 6.1, we now obtain a set of modified modes using the function `fem.ComputeHurtyCraigBamptonModes(...)`. Fig. 6.3 shows the first 6 rigid body modes. Note that these modes would be automatically removed in the function `fem.ComputeHurtyCraigBamptonModes(...)`. Fig. 6.4 shows the second set of 6 rigid body modes. Finally, 8 eigenmodes have been computed for the fixed-fixed case (where all boundary/interfaces nodes are fixed), see Fig. 6.5. The eigenfrequencies for this case now are significantly higher than in the free-free case, reading

$$f_{0.7} = [1277.35, 1469.86, 3336.91, 3584.28, \dots] \quad (6.34)$$

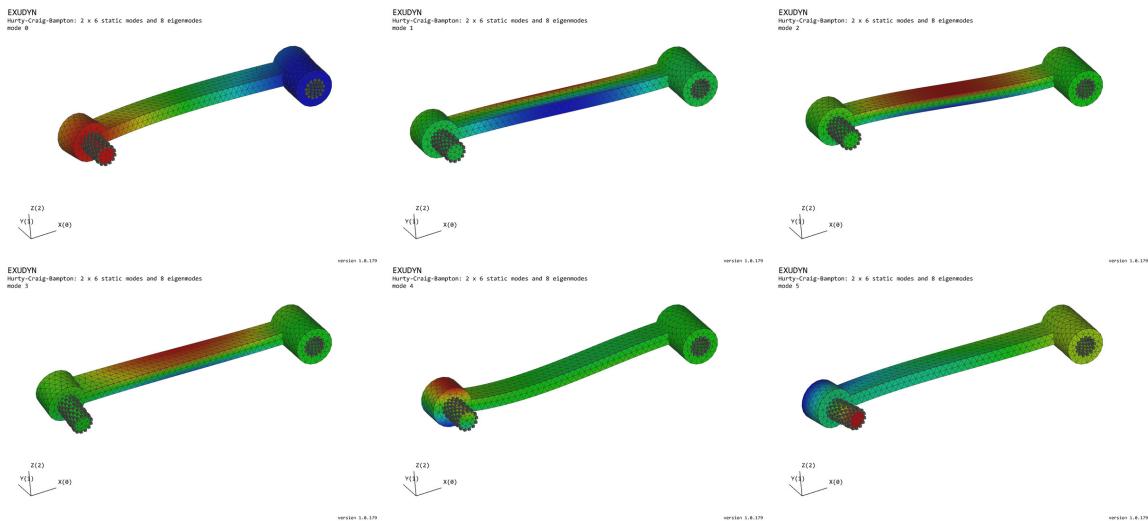


Figure 6.3: Static modes for bolt rigid body interface, using Hurty-Craig-Bampton method; top three images show (x,y,z)-translation modes, bottom three images show (x,y,z)-rotation modes; contour color represents norm of displacements.

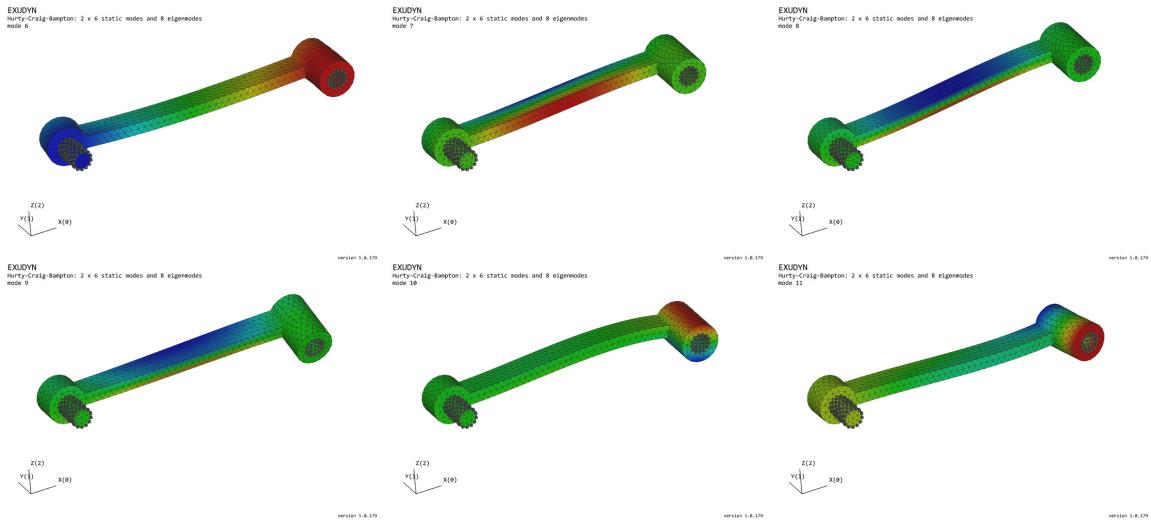


Figure 6.4: Static modes for bushing rigid body interface, using Hurty-Craig-Bampton method; top three images show  $(x,y,z)$ -translation modes, bottom three images show  $(x,y,z)$ -rotation modes; contour color represents norm of displacements.

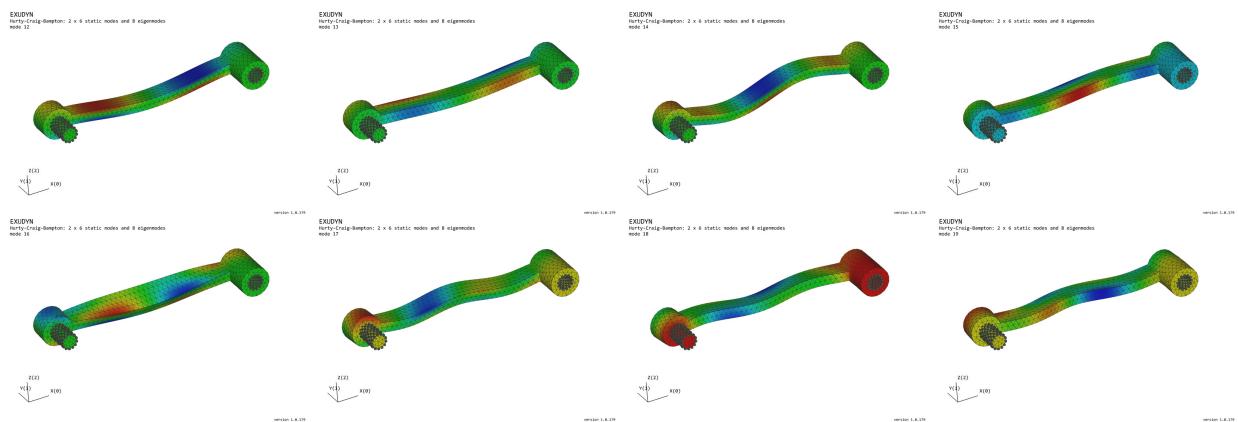


Figure 6.5: Eigenmodes for fixed-fixed case, resulting from Hurty-Craig-Bampton method; contour color represents norm of displacements.

## 6.3 Modeling of Contact in EXUDYN

The GeneralContact module, see [Section 4.7](#), provides a simple, efficient and versatile interface to a general contact module. The motivation for this module is based on the need for simple contact modeling in robotics, but also for the efficient modeling of beam-cylinder or beam-beam contact, as well as contact between deformable meshes (not yet available).

Note that there are currently only simplistic contact models, such as linear contact and simple damping, which are not representing realistic Hertzian contact (which will be implemented in near future). Furthermore, read the notes in GeneralContact carefully, how stiffness and damping is realized – e.g., stiffness may be a serial spring against the other object, while damping is implemented as parallel damper.

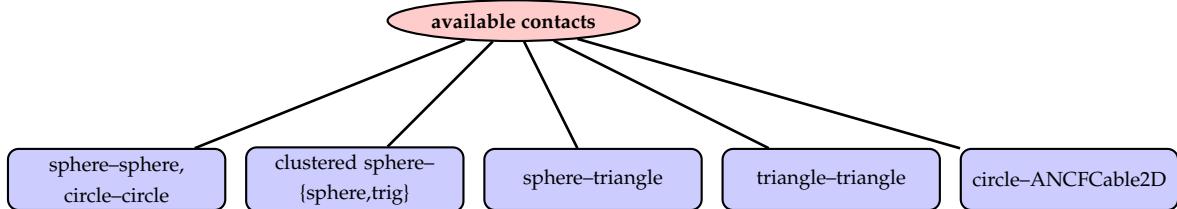


Figure 6.6: Contact: possible coupling of geometrical objects in EXUDYN

Fig. 6.6 shows the implemented / possible coupling of contact objects:

- 1) simulate spherical particles; in 2D, spheres are represented as circles
- 2) simulate clustered spherical [circular] particles which consist of rigid bodies made of a cluster of spheres; several contact spheres are attached to one rigid body by using rigid body markers; in 2D, spheres are represented as circles
- 3) simulate the contact of spheres with triangular meshes, e.g., in order to provide some limitations of the range of motion for your objects
- 4) simulate the contact between arbitrarily shaped rigid bodies
- 5) simulate the contact between rolls (spheres) and ANCF cable elements; to enable cable-cable contact, spheres must be attached and distributed along the cable elements

In all use cases, explicit integrators are much faster and they are recommended, as long as your problem allows to do so.

### 6.3.1 Contact of meshed rigid bodies

Case 4) is more involved and needs further explanations (which more or less also applies to case 4). The geometry is approximated by a mesh consisting of flat triangles, which are attached to a rigid body (marker). The triangles obtain a contact stiffness against spheres. There are special cases, depending if the sphere gets in contact with the triangle plane or with the triangle edge. Having contact with edges, usually involves several triangles at the same time (specifically at vertices), which leads to higher contact stiffness as compared to a planar contact. Nevertheless, for flat planes, contact computation takes care, that contact stiffness is constant in the whole plane, independently of the number of involved triangles. In order to realize contact between meshed rigid bodies, both body-attached meshes are added via the GeneralContact function

- `AddTrianglesRigidBodyBased(...)`

In addition, all mesh vertices are added as spheres with markers using

- `AddSphereWithMarker(...)`

However, as we need a certain finite radius of the spheres, the mesh must be shrunk for this purpose (and it needs to have according thickness). Shrinking of the (consistent) triangular mesh can be done by the utility function

- `ShrinkMeshNormalToSurface(...);`
- in order to reduce artifacts at object edges, it is recommended to refine the mesh, using the utility function `RefineMesh(...)`

According examples can be found in test models, but there will be a more convenient function for contact of meshes attached to rigid bodies in the future.

All contacts can be created in a `GeneralContact` object – which is not a regular object in mbs – created by

- `gContact = mbs.AddGeneralContact()`

Note that one can create several, independent contact objects. Hereafter, spheres, triangles, ... are added with appropriate functions, see [Section 4.7](#). Note that triangles need to be correctly numbered (see correct normals in Fig. 9.1), which defines inside/outside of a triangular mesh.

### 6.3.2 Regularized friction

Within a regularized friction law, similar to a well known law attributed to Haff-Werner, the friction force  $\mathbf{f}_f$  is computed from static (dry) friction coefficient  $\mu_s$  and the friction regularization velocity<sup>3</sup>  $v_{\mu,reg}$

$$v_t = |{}^0\mathbf{v}_t|,$$

$$\mathbf{f}_f = \begin{cases} \frac{\mu_s \cdot |f_c|}{v_{\mu,reg}} {}^0\mathbf{v}_t, & \text{if } v_t < v_{\mu,reg} \\ \mu_s \cdot |f_c| \frac{{}^0\mathbf{v}_t}{v_t}, & \text{else} \end{cases} \quad (6.35)$$

**Note** that the following equations represent the computed contact relations in high detail, but minor cases and flags, such as the `intraSpheresContact` are not described here, but must be carefully considered in the description of `GeneralContact`, see [Section 4.7](#).

### 6.3.3 Sphere-sphere contact: Equations

The equations for sphere-sphere contact in contact normal direction are very similar to the `ObjectConnectorSpringDamper`, see [Section 7.6.1](#). Every sphere is attached to a position-based marker<sup>4</sup>. In C++, the sphere attached to marker 0 is denoted as `sphereI` and the sphere attached to marker 1 is denoted as `sphereJ`. Input parameters for this contact model are

intermediate variables	symbol	description
sphere $i$ radius	$r_i$	radius of sphere $i$ , attached to marker 0
sphere $j$ radius	$r_j$	radius of sphere $j$ , attached to marker 1
sphere $i$ contact stiffness	$k_i$	N/m
sphere $j$ contact stiffness	$k_j$	N/m
sphere $i$ contact damping	$d_i$	N/m
sphere $j$ contact damping	$d_j$	N/m
friction pairing coefficient	$\mu_{ij}$	the friction coefficient stored in the friction pairings matrix, resulting from the friction indices of spheres $i$ and $j$

<sup>3</sup>global regularization coefficient stored in `GeneralContact.frictionProportionalZone`

<sup>4</sup>which can be attached itself to position nodes, rigid body nodes, point masses, rigid bodies as well as flexible bodies.  
NOTE, that in case of implicit integration, flexible bodies are not fully implemented!

Marker positions and velocities are given by the relations:

intermediate variables	symbol	description
sphere $i$ position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker $m0$
sphere $j$ position	${}^0\mathbf{p}_{m1}$	current global position which is provided by marker $m1$
marker $m0$ position Jacobian	${}^0\mathbf{J}_{pos,m0}$	with interpretation as variation of the position $\delta {}^0\mathbf{p}_{m0} = {}^0\mathbf{J}_{pos,m0} \delta \mathbf{q}_{m0}$ ; assuming that $\mathbf{q}_{m0}$ represents the generalized coordinates of marker $m0$
marker $m1$ position Jacobian	${}^0\mathbf{J}_{pos,m1}$	with interpretation as variation of the position $\delta {}^0\mathbf{p}_{m1} = {}^0\mathbf{J}_{pos,m1} \delta \mathbf{q}_{m1}$ ; assuming that $\mathbf{q}_{m1}$ represents the generalized coordinates of marker $m1$
sphere $i$ velocity	${}^0\mathbf{v}_i$	current global velocity which is provided by marker $m0$
sphere $j$ velocity	${}^0\mathbf{v}_j$	
relative position	${}^0\mathbf{n}$	${}^0\mathbf{p}_j - {}^0\mathbf{p}_i$
Distance*	$L =  {}^0\mathbf{n} $	
unit vector*	${}^0\mathbf{n}_0 = \frac{1}{L} {}^0\mathbf{n}$	vector in contact normal direction
gap*	$g = L - (r_i + r_j)$	
penetration*	$p = -g = r_i + r_j - L$	

In case of rigid bodies and non-zero friction, we also compute angular velocities and orientation,

intermediate variables: rigid bodies and friction	symbol	description
sphere $i$ angular velocity	${}^{m0}\boldsymbol{\omega}_i$	current local angular velocity provided by marker $m0$
marker $m0$ orientation	${}^{0,m0}\mathbf{A}$	transformation from marker $m0$ (body-fixed) to global coordinates
sphere $j$ angular velocity	${}^{m1}\boldsymbol{\omega}_j$	current local angular velocity provided by marker $m1$
marker $m1$ orientation	${}^{0,m1}\mathbf{A}$	transformation from marker $m1$ (body-fixed) to global coordinates
marker $m0$ rotation Jacobian	${}^0\mathbf{J}_{rot,m0}$	with interpretation as derivative of the global angular velocity ${}^0\boldsymbol{\omega}_{m0} = {}^0\mathbf{J}_{rot,m0} \dot{\mathbf{q}}_{m0}$ ; assuming that $\dot{\mathbf{q}}_{m0}$ represents the generalized velocities of marker $m0$
marker $m1$ rotation Jacobian	${}^0\mathbf{J}_{rot,m1}$	with interpretation as derivative of the global angular velocity ${}^0\boldsymbol{\omega}_{m1} = {}^0\mathbf{J}_{rot,m1} \dot{\mathbf{q}}_{m1}$ ; assuming that $\dot{\mathbf{q}}_{m1}$ represents the generalized velocities of marker $m1$

Contact between spheres with global index  $g_i$  and another sphere with global index  $g_j$  is active, if

- Bounding box of sphere  $g_j$  intersects with a box in the searchtree which also intersects with bounding box of sphere  $g_i$  AND
- if Bounding box of sphere  $g_j$  intersects with bounding box of sphere  $g_i$  AND
- if the condition  $L^2 < (r_i + r_j)^2$  holds OR if  $g_j$  belongs to the active set of  $g_i$  (computed in PostNewtonStep).

Note that quantities \* are only computed if contact is active.

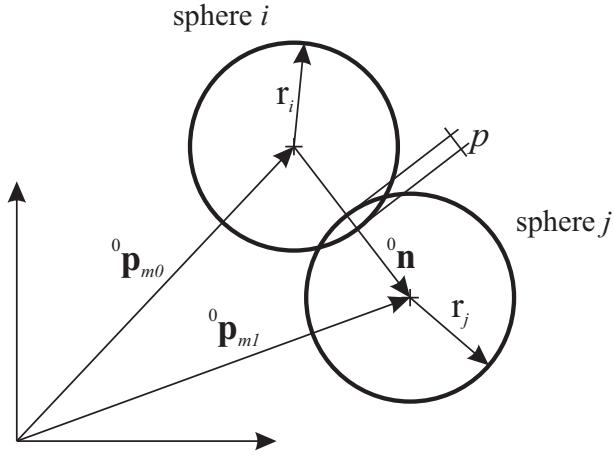


Figure 6.7: Geometrical relations for contact of two spheres  $i$  and  $j$  with according markers  $m0$  and  $m1$ .

### 6.3.3.1 Contact relations for sphere $g_i$ (marker $m0$ ) and sphere $g_j$ (marker $m1$ )

If contact is active, we compute the global position of the contact point<sup>5</sup>, see Fig. 6.7

$${}^0\mathbf{p}_c = {}^0\mathbf{p}_{m0} + r_i \cdot \mathbf{n}_0 , \quad (6.36)$$

the velocities of the spheres at the contact point<sup>6</sup>,

$$\begin{aligned} {}^0\mathbf{v}_{c,i} &= {}^0\mathbf{v}_i + ({}^{0,m0}\mathbf{A} {}^{m0}\boldsymbol{\omega}_i) \times ({}^0\mathbf{p}_c - {}^0\mathbf{p}_i) = {}^0\mathbf{v}_i + r_i \cdot ({}^{0,m0}\mathbf{A} {}^{m0}\boldsymbol{\omega}_i) \times {}^0\mathbf{n}_0 , \\ {}^0\mathbf{v}_{c,j} &= {}^0\mathbf{v}_j + ({}^{0,m1}\mathbf{A} {}^{m1}\boldsymbol{\omega}_j) \times ({}^0\mathbf{p}_c - {}^0\mathbf{p}_j) = {}^0\mathbf{v}_j - r_j \cdot ({}^{0,m1}\mathbf{A} {}^{m1}\boldsymbol{\omega}_j) \times {}^0\mathbf{n}_0 , \end{aligned} \quad (6.37)$$

the velocity in contact normal direction, which can be computed from sphere's center points,

$$v_n = {}^0\mathbf{n}_0^T ({}^0\mathbf{v}_j - {}^0\mathbf{v}_i) = {}^0\mathbf{n}_0^T ({}^0\mathbf{v}_{c,j} - {}^0\mathbf{v}_{c,i}) , \quad (6.38)$$

the velocity in tangential direction, considering the tangential velocities,

$$\begin{aligned} {}^0\mathbf{v}_t &= {}^0\mathbf{v}_{c,j} - {}^0\mathbf{v}_{c,i} - v_n \cdot {}^0\mathbf{n}_0 = (\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0) ({}^0\mathbf{v}_{c,j} - {}^0\mathbf{v}_{c,i}) , \\ &= (\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0) ({}^0\mathbf{v}_j + r_j \cdot {}^0\tilde{\mathbf{n}}_0 {}^{0,m1}\mathbf{A} {}^{m1}\boldsymbol{\omega}_j - {}^0\mathbf{v}_i + r_i \cdot {}^0\tilde{\mathbf{n}}_0 {}^{0,m0}\mathbf{A} {}^{m0}\boldsymbol{\omega}_i) \\ &= (\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0) ({}^0\mathbf{v}_j + r_j \cdot {}^0\tilde{\mathbf{n}}_0 {}^0\boldsymbol{\omega}_j - {}^0\mathbf{v}_i + r_i \cdot {}^0\tilde{\mathbf{n}}_0 {}^0\boldsymbol{\omega}_i) , \end{aligned} \quad (6.39)$$

the effective contact stiffness coefficient based on the stiffness  $k_i$  of sphere  $i$  and stiffness  $k_j$  of sphere  $j$ ,

$$k_c = \frac{k_i \cdot k_j}{k_i + k_j} , \quad (6.40)$$

and the effective contact damping coefficient<sup>7</sup> based on the damping  $k_i$  of sphere  $i$  and damping  $k_j$  of sphere  $j$

$$d_c = d_i + d_j . \quad (6.41)$$

<sup>5</sup>considering also penetration, a more consistent contact point would be  ${}^0\mathbf{p}_c^* = {}^0\mathbf{p}_{m0} + (r_i - \frac{p}{2}) \cdot \mathbf{n}_0$ , subtracting the half penetration.

<sup>6</sup>In case of no friction, the angular velocities are not included in these relations

<sup>7</sup>Note that this simplicial damping law is used according to the idea of parallel dampers, because serial dampers would not allow to adjust damping for different particles

The contact force (negative contact pressure) is computed from gap  $g$  and normal velocity  $v_n$ ,

$$f_c = k_c \cdot g + d_c \cdot v_n , \quad (6.42)$$

and the total vectorial contact force is computed with the help of Eq. (6.35), defining the friction force  $\mathbf{f}_f$ ,

$${}^0\mathbf{f}_c = f_c \cdot \mathbf{n}_0 + \mathbf{f}_f . \quad (6.43)$$

The torque due to friction for sphere  $i$  and sphere  $j$  results into<sup>8</sup>

$${}^0\boldsymbol{\tau}_{f,i} = (-r_i \cdot \mathbf{n}_0) \times {}^0\mathbf{f}_f , \quad {}^0\boldsymbol{\tau}_{f,j} = (-r_j \cdot \mathbf{n}_0) \times {}^0\mathbf{f}_f . \quad (6.44)$$

### 6.3.3.2 Generalized forces due to contact

Based on the contact pressure and the friction forces, forces and torques are applied via the markers' Jacobians, resulting in generalized forces to whatever the marker is attached to.

The generalized forces to the marker  $m0$  and  $m1$  (sphere  $i$  and  $j$ ) are computed as

$$\begin{aligned} \mathbf{f}_{m0,LHS} &= -{}^0\mathbf{J}_{pos,m0}^T {}^0\mathbf{f}_c + {}^0\mathbf{J}_{rot,m0}^T {}^0\boldsymbol{\tau}_{f,i} , \\ \mathbf{f}_{m1,LHS} &= {}^0\mathbf{J}_{pos,m1}^T {}^0\mathbf{f}_c + {}^0\mathbf{J}_{rot,m1}^T {}^0\boldsymbol{\tau}_{f,j} . \end{aligned} \quad (6.45)$$

### 6.3.3.3 Jacobi matrix for sphere $g_i$ and sphere $g_j$

For implicit time integration, the (contact) Jacobian<sup>9</sup> represents the derivative of the generalized forces

$$\mathbf{f}_{LHS} = \begin{bmatrix} \mathbf{f}_{m0,LHS} \\ \mathbf{f}_{m1,LHS} \end{bmatrix} \quad (6.46)$$

with respect to the generalized coordinates affected by the two markers,

$$\mathbf{q} = \begin{bmatrix} \mathbf{q}_{m0} \\ \mathbf{q}_{m1} \end{bmatrix} . \quad (6.47)$$

The Jacobian thus reads<sup>10</sup>

$$\mathbf{J}_c = \frac{\partial \mathbf{f}_{LHS}}{\partial \mathbf{q}} = \begin{bmatrix} \frac{\partial (-{}^0\mathbf{J}_{pos,m0}^T {}^0\mathbf{f}_c + {}^0\mathbf{J}_{rot,m0}^T {}^0\boldsymbol{\tau}_{f,i})}{\partial \mathbf{q}_{m0}} & \frac{\partial (-{}^0\mathbf{J}_{pos,m0}^T {}^0\mathbf{f}_c + {}^0\mathbf{J}_{rot,m0}^T {}^0\boldsymbol{\tau}_{f,i})}{\partial \mathbf{q}_{m1}} \\ \frac{\partial ({}^0\mathbf{J}_{pos,m1}^T {}^0\mathbf{f}_c + {}^0\mathbf{J}_{rot,m1}^T {}^0\boldsymbol{\tau}_{f,j})}{\partial \mathbf{q}_{m0}} & \frac{\partial ({}^0\mathbf{J}_{pos,m1}^T {}^0\mathbf{f}_c + {}^0\mathbf{J}_{rot,m1}^T {}^0\boldsymbol{\tau}_{f,j})}{\partial \mathbf{q}_{m1}} \end{bmatrix} \quad (6.48)$$

The single terms may be expressed as

$$\begin{aligned} \frac{\partial (-{}^0\mathbf{J}_{pos,m0}^T {}^0\mathbf{f}_c + {}^0\mathbf{J}_{rot,m0}^T {}^0\boldsymbol{\tau}_{f,i})}{\partial \mathbf{q}_{m0,1}} &= \\ -\frac{\partial {}^0\mathbf{J}_{pos,m0}^T {}^0\mathbf{f}_c}{\partial \mathbf{q}_{m0,1}} - {}^0\mathbf{J}_{pos,m0}^T \frac{\partial {}^0\mathbf{f}_c}{\partial \mathbf{q}_{m0,1}} + \frac{\partial {}^0\mathbf{J}_{rot,m0}^T {}^0\boldsymbol{\tau}_{f,i}}{\partial \mathbf{q}_{m0,1}} + {}^0\mathbf{J}_{rot,m0}^T \frac{\partial {}^0\boldsymbol{\tau}_{f,i}}{\partial \mathbf{q}_{m0,1}} , \end{aligned} \quad (6.49)$$

<sup>8</sup>note that both signs are the same and that  ${}^0\mathbf{f}_f$  could be replaced by  $\mathbf{f}_c$

<sup>9</sup>Here, we only consider the local Jacobian related to the coordinates underlying the two markers  $m0$  and  $m1$ ; in the implementation, the parts of the Jacobian are added to the sparse system

<sup>10</sup>NOTE that only terms marked in green are currently fully implemented and terms in blue are approximated, while other terms are neglected

and similar for  $m_1$ . In order to simplify implementation (avoiding arrays with 3 indices) and improve computational efficiency, derivatives of Jacobians are realized as

$$\frac{\partial {}^0\mathbf{J}_{pos,m0}^T}{\partial \mathbf{q}_{m0}} {}^0\mathbf{f}_c = \frac{\partial {}^0\mathbf{J}_{pos,m0}^T}{\partial \mathbf{q}_{m0}} {}^0\tilde{\mathbf{f}}_c, \quad (6.50)$$

in which  ${}^0\tilde{\mathbf{f}}_c = {}^0\mathbf{f}_c$ , but assumed to be a constant and not depending on  $\mathbf{q}$  in the computation of derivatives. Note that derivatives for position Jacobians, e.g.,  $\frac{\partial {}^0\mathbf{J}_{pos,m0}^T}{\partial \mathbf{q}_{m0}}$  or rotation Jacobians are provided by the according markers (will be described there in the near future).

For the jacobians, we need to compute the derivatives of the following terms<sup>1112</sup>:

- $L = ({}^0\mathbf{n}^T {}^0\mathbf{n})^{\frac{1}{2}}$ :

$$\frac{\partial L}{\partial \mathbf{q}_{m0,m1}} = \frac{\partial ({}^0\mathbf{n}^T {}^0\mathbf{n})^{\frac{1}{2}}}{\partial \mathbf{q}_{m0,m1}} = \frac{1}{L} \left( {}^0\mathbf{n}^T \frac{\partial {}^0\mathbf{n}}{\partial \mathbf{q}_{m0,m1}} \right) = \frac{1}{L} ({}^0\mathbf{n}^T {}^0\mathbf{J}_{pos}) = ({}^0\mathbf{n}_0^T {}^0\mathbf{J}_{pos}) \quad (6.51)$$

- $L^{-1} = ({}^0\mathbf{n}^T {}^0\mathbf{n})^{-\frac{1}{2}}$ :

$$\frac{\partial L^{-1}}{\partial \mathbf{q}_{m0,m1}} = \frac{\partial ({}^0\mathbf{n}^T {}^0\mathbf{n})^{-\frac{1}{2}}}{\partial \mathbf{q}_{m0,m1}} = -\frac{1}{L^3} \left( {}^0\mathbf{n}^T \frac{\partial {}^0\mathbf{n}}{\partial \mathbf{q}_{m0,m1}} \right) = -\frac{1}{L^2} ({}^0\mathbf{n}_0^T {}^0\mathbf{J}_{pos}) \quad (6.52)$$

- ${}^0\mathbf{n} = {}^0\mathbf{p}_j - {}^0\mathbf{p}_i$ :

$$\frac{\partial {}^0\mathbf{n}}{\partial \mathbf{q}_{m0,m1}} = {}^0\mathbf{J}_{pos} \quad (6.53)$$

- ${}^0\mathbf{n}_0 = \frac{1}{L} {}^0\mathbf{n}$ :<sup>13</sup>

$$\frac{\partial {}^0\mathbf{n}_0}{\partial \mathbf{q}_{m0,m1}} = -\frac{1}{L^3} ({}^0\mathbf{n} \otimes {}^0\mathbf{n}) {}^0\mathbf{J}_{pos} + \frac{1}{L} {}^0\mathbf{J}_{pos} = \frac{1}{L} (\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0) {}^0\mathbf{J}_{pos} \quad (6.54)$$

- $g = L - r_i + r_j$ :

$$\frac{\partial g}{\partial \mathbf{q}_{m0,m1}} = {}^0\mathbf{n}_0^T {}^0\mathbf{J}_{pos} \quad (6.55)$$

- $v_n = ({}^0\mathbf{v}_j - {}^0\mathbf{v}_i)^T {}^0\mathbf{n}_0$  (**NOTE:** only valid in case that markers are attached to node or body reference point!!!):

$$\frac{\partial v_n}{\partial \mathbf{q}_{m0,m1}} = ({}^0\mathbf{v}_j - {}^0\mathbf{v}_i)^T \left( \frac{1}{L} (\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0) {}^0\mathbf{J}_{pos} \right) \quad (6.56)$$

$$\frac{\partial v_n}{\partial \dot{\mathbf{q}}_{m0,m1}} = {}^0\mathbf{n}_0^T {}^0\mathbf{J}_{pos} \quad (6.57)$$

- $f_c = k_c \cdot g + d_c \cdot v_n$ :

$$\frac{\partial f_c}{\partial \mathbf{q}_{m0,m1}} = k_c \frac{\partial g}{\partial \mathbf{q}_{m0,m1}} + d_c \frac{\partial v_n}{\partial \mathbf{q}_{m0,m1}} = k_c \cdot {}^0\mathbf{n}_0^T {}^0\mathbf{J}_{pos} + d_c \cdot ({}^0\mathbf{v}_j - {}^0\mathbf{v}_i)^T \left( \frac{1}{L} (\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0) {}^0\mathbf{J}_{pos} \right) \quad (6.58)$$

$$\frac{\partial f_c}{\partial \dot{\mathbf{q}}_{m0,m1}} = d_c \frac{\partial v_n}{\partial \dot{\mathbf{q}}_{m0,m1}} = d_c {}^0\mathbf{n}_0^T {}^0\mathbf{J}_{pos} \quad (6.59)$$

<sup>11</sup>terms that are implemented are marked in green; black terms are not implemented or unused

<sup>12</sup>to keep derivations short, we use  ${}^0\mathbf{J}_{pos}$ , which represents  $-{}^0\mathbf{J}_{pos,m0}$  in case of  $\frac{\partial}{\partial \mathbf{q}_{m0}}$  and  ${}^0\mathbf{J}_{pos,m1}$  in case of  $\frac{\partial}{\partial \mathbf{q}_{m1}}$

<sup>13</sup>NOTE: dyadic product  $\otimes$

- ${}^0\mathbf{v}_t = {}^0\mathbf{v}_{c,j} - {}^0\mathbf{v}_{c,i} - v_n \cdot {}^0\mathbf{n}_0 = (\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0)({}^0\mathbf{v}_j - {}^0\mathbf{v}_i) + r_j \cdot {}^0\tilde{\mathbf{n}}_0 {}^0\boldsymbol{\omega}_j + r_i \cdot {}^0\tilde{\mathbf{n}}_0 {}^0\boldsymbol{\omega}_i;$

$$\begin{aligned} \frac{\partial {}^0\mathbf{v}_t}{\partial \dot{\mathbf{q}}_{m0,m1}} &= -{}^0\mathbf{n}_0 \otimes ({}^0\mathbf{v}_j - {}^0\mathbf{v}_i) \left( \frac{1}{L} (\mathbf{I} - {}^0\mathbf{n} \otimes {}^0\mathbf{n}) {}^0\mathbf{J}_{pos} \right) \\ &\quad - v_n \left( \frac{1}{L} (\mathbf{I} - {}^0\mathbf{n} \otimes {}^0\mathbf{n}) {}^0\mathbf{J}_{pos} \right) - v_n \cdot \left( \frac{1}{L} (\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0) {}^0\mathbf{J}_{pos} \right) \\ &\quad + r_{i,j} \cdot \left( -\frac{1}{L} {}^0\tilde{\boldsymbol{\omega}}_{i,j} (\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0) + {}^0\tilde{\mathbf{n}}_0 \frac{\partial {}^0\boldsymbol{\omega}_{i,j}}{\partial \dot{\mathbf{q}}_{m0,m1}} \right) \end{aligned} \quad (6.60)$$

- velocity coordinate derivatives for  ${}^0\mathbf{v}_t$ :

$$\frac{\partial {}^0\mathbf{v}_t}{\partial \dot{\mathbf{q}}_{m0}} = (\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0) (-{}^0\mathbf{J}_{pos,m0}) + r_i \cdot {}^0\tilde{\mathbf{n}}_0 {}^0\mathbf{J}_{rot,m0}, \quad (6.61)$$

$$\frac{\partial {}^0\mathbf{v}_t}{\partial \dot{\mathbf{q}}_{m1}} = (\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0) ({}^0\mathbf{J}_{pos,m1}) + r_j \cdot {}^0\tilde{\mathbf{n}}_0 {}^0\mathbf{J}_{rot,m1} \quad (6.62)$$

The contact force reads (note that because  $f_c$  is always negative, the sign of regularization term is negative),

$${}^0\mathbf{f}_c = f_c \cdot {}^0\mathbf{n}_0 + \mathbf{f}_f = \begin{cases} f_c \left( {}^0\mathbf{n}_0 - \frac{\mu_s}{v_{\mu,reg}} {}^0\mathbf{v}_t \right), & \text{if } |{}^0\mathbf{v}_t| < v_{\mu,reg} \\ f_c \cdot {}^0\mathbf{n}_0 + \mathbf{f}_f, & \text{else with } \mathbf{f}_f = const. \end{cases} \quad (6.63)$$

Thus we introduce a factor  $\delta_f$ , which is  $\delta_f = 1$  in the regularized small velocity state, and in the saturated (constant) friction force we use  $\delta_f = 0$ . Thus, the jacobian of the contact force  ${}^0\mathbf{f}_c$  reads (note the diadic product  $\otimes$ ),

$$\begin{aligned} \frac{\partial {}^0\mathbf{f}_c}{\partial \dot{\mathbf{q}}_{m0,m1}} &= \left( {}^0\mathbf{n}_0 - \delta_f \frac{\mu_s}{v_{\mu,reg}} {}^0\mathbf{v}_t \right) \otimes \frac{\partial f_c}{\partial \dot{\mathbf{q}}_{m0,m1}} + f_c \left( \frac{\partial {}^0\mathbf{n}_0}{\partial \dot{\mathbf{q}}_{m0,m1}} - \delta_f \frac{\mu_s}{v_{\mu,reg}} \frac{\partial {}^0\mathbf{v}_t}{\partial \dot{\mathbf{q}}_{m0,m1}} \right) \\ &= \left( {}^0\mathbf{n}_0 - \delta_f \frac{\mu_s}{v_{\mu,reg}} {}^0\mathbf{v}_t \right) \left( k_c \frac{\partial g}{\partial \dot{\mathbf{q}}_{m0,m1}} + d_c \frac{\partial v_n}{\partial \dot{\mathbf{q}}_{m0,m1}} \right) + f_c \left( \frac{\partial {}^0\mathbf{n}_0}{\partial \dot{\mathbf{q}}_{m0,m1}} - \delta_f \frac{\mu_s}{v_{\mu,reg}} \frac{\partial {}^0\mathbf{v}_t}{\partial \dot{\mathbf{q}}_{m0,m1}} \right) \\ &= \left( {}^0\mathbf{n}_0 - \delta_f \frac{\mu_s}{v_{\mu,reg}} {}^0\mathbf{v}_t \right) \otimes \left( k_c \cdot {}^0\mathbf{n}_0^\top + d_c \cdot ({}^0\mathbf{v}_j - {}^0\mathbf{v}_i)^\top \left( \frac{1}{L} (\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0) \right) \right) {}^0\mathbf{J}_{pos} + f_c \cdot (\dots) \end{aligned} \quad (6.64)$$

The jacobian for the contact force  ${}^0\mathbf{f}_c$  w.r.t. velocity marker coordinates reads, note that  ${}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0 - \mathbf{I} = -(\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0)$ ,

$$\frac{\partial {}^0\mathbf{f}_c}{\partial \dot{\mathbf{q}}_{m0,1}} = \left( {}^0\mathbf{n}_0 - \delta_f \frac{\mu_s}{v_{\mu,reg}} {}^0\mathbf{v}_t \right) \otimes \frac{\partial f_c}{\partial \dot{\mathbf{q}}_{m0,1}} - f_c \cdot \left( \delta_f \frac{\mu_s}{v_{\mu,reg}} \frac{\partial {}^0\mathbf{v}_t}{\partial \dot{\mathbf{q}}_{m0,1}} \right) \quad (6.65)$$

$$\begin{aligned} \frac{\partial {}^0\mathbf{f}_c}{\partial \dot{\mathbf{q}}_{m0}} &= -d_c \left( {}^0\mathbf{n}_0 - \delta_f \frac{\mu_s}{v_{\mu,reg}} {}^0\mathbf{v}_t \right) \otimes ({}^0\mathbf{n}_0^\top {}^0\mathbf{J}_{pos,m0}) - \\ &\quad f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}} \left( -(\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0) {}^0\mathbf{J}_{pos,m0} + r_i \cdot {}^0\tilde{\mathbf{n}}_0 {}^0\mathbf{J}_{rot,m0} \right) \\ \frac{\partial {}^0\mathbf{f}_c}{\partial \dot{\mathbf{q}}_{m1}} &= d_c \left( {}^0\mathbf{n}_0 - \delta_f \frac{\mu_s}{v_{\mu,reg}} {}^0\mathbf{v}_t \right) \otimes ({}^0\mathbf{n}_0^\top {}^0\mathbf{J}_{pos,m1}) - \\ &\quad f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}} \left( (\mathbf{I} - {}^0\mathbf{n}_0 \otimes {}^0\mathbf{n}_0) {}^0\mathbf{J}_{pos,m1} + r_j \cdot {}^0\tilde{\mathbf{n}}_0 {}^0\mathbf{J}_{rot,m1} \right) \end{aligned} \quad (6.66)$$

The jacobians for torques are computed for the case that friction is in the regularized small velocity state ( $\delta_f = 1$ ), while otherwise derivatives of  ${}^0\boldsymbol{\tau}_{f,(i,j)}$  are zero,

$${}^0\boldsymbol{\tau}_{f,i} = (-r_i \cdot {}^0\mathbf{n}_0) \times \left( -f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}} {}^0\mathbf{v}_t \right), \quad {}^0\boldsymbol{\tau}_{f,j} = (-r_j \cdot {}^0\mathbf{n}_0) \times \left( -f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}} {}^0\mathbf{v}_t \right) \quad (6.67)$$

in case of no friction or constant friction forces,  $\delta_f = 0$ . The jacobians follow from (accordingly for  $\tau_{f,i}$ ,  $\tau_{f,j}$  and derivatives w.r.t  $\mathbf{q}_{m0,1}$ ):

$$\begin{aligned} \frac{\partial^0 \tau_{f,(i,j)}}{\partial \mathbf{q}_{m0,1}} &= \frac{\partial (-r_{(i,j)} \cdot {}^0\mathbf{n}_0) \times \left( -f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}} {}^0\mathbf{v}_t \right)}{\partial \mathbf{q}_{m0,1}} \\ &= \left( -r_{(i,j)} \frac{\partial {}^0\mathbf{n}_0}{\partial \mathbf{q}_{m0,1}} \right) \times \left( -f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}} {}^0\mathbf{v}_t \right) + \left( -r_{(i,j)} \cdot {}^0\mathbf{n}_0 \right) \times \left( -f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}} \frac{\partial {}^0\mathbf{v}_t}{\partial \mathbf{q}_{m0,1}} \right) + \frac{\partial f_c}{\partial \mathbf{q}_{m0,1}} (...) \\ &= \left( -f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}} {}^0\tilde{\mathbf{v}}_t \right) \left( r_{(i,j)} \frac{\partial {}^0\mathbf{n}_0}{\partial \mathbf{q}_{m0,1}} \right) + \left( -r_{(i,j)} \cdot {}^0\tilde{\mathbf{n}}_0 \right) \left( -f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}} \frac{\partial {}^0\mathbf{v}_t}{\partial \mathbf{q}_{m0,1}} \right) + \frac{\partial f_c}{\partial \mathbf{q}_{m0,1}} (...) \end{aligned} \quad (6.68)$$

and (note that  ${}^0\tilde{\mathbf{n}}_0 \cdot {}^0\mathbf{n}_0 = 0$ ),

$$\begin{aligned} \frac{\partial^0 \tau_{f,(i,j)}}{\partial \dot{\mathbf{q}}_{m0}} &= \left( -r_{(i,j)} \cdot {}^0\tilde{\mathbf{n}}_0 \right) \left( -f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}} \frac{\partial {}^0\mathbf{v}_t}{\partial \dot{\mathbf{q}}_{m0}} - \frac{\partial f_c}{\partial \dot{\mathbf{q}}_{m0}} \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}} {}^0\mathbf{v}_t \right) \\ &= \left( r_{(i,j)} \cdot {}^0\tilde{\mathbf{n}}_0 \right) \left( f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}} \left( -{}^0\mathbf{J}_{pos,m0} + r_i \cdot {}^0\tilde{\mathbf{n}}_0 {}^0\mathbf{J}_{rot,m0} \right) - d_c \delta_f \frac{\mu_s}{v_{\mu,reg}} \cdot {}^0\mathbf{v}_t \otimes ({}^0\mathbf{n}_0 {}^0\mathbf{J}_{pos,m0}) \right) \end{aligned} \quad (6.69)$$

$$\begin{aligned} \frac{\partial^0 \tau_{f,(i,j)}}{\partial \dot{\mathbf{q}}_{m1}} &= \left( -r_{(i,j)} \cdot \tilde{\mathbf{n}}_0 \right) \left( -f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}} \frac{\partial {}^0\mathbf{v}_t}{\partial \dot{\mathbf{q}}_{m1}} - \frac{\partial f_c}{\partial \dot{\mathbf{q}}_{m1}} \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}} {}^0\mathbf{v}_t \right) \\ &= \left( r_{(i,j)} \cdot \tilde{\mathbf{n}}_0 \right) \left( f_c \cdot \delta_f \frac{\mu_s}{v_{\mu,reg}} \left( {}^0\mathbf{J}_{pos,m1} + r_j \cdot {}^0\tilde{\mathbf{n}}_0 {}^0\mathbf{J}_{rot,m1} \right) + d_c \delta_f \frac{\mu_s}{v_{\mu,reg}} \cdot {}^0\mathbf{v}_t \otimes ({}^0\mathbf{n}_0 {}^0\mathbf{J}_{pos,m1}) \right) \end{aligned} \quad (6.70)$$

### 6.3.4 Contact relations for ANCF cable $g_i$ (marker $m0$ ) and sphere $g_j$ (marker $m1$ )

If contact is active, we have two relative axial reference coordinates  $s_0$  and  $s_1$ , which define start and end location at the beam, for which the span in between intersects with the circle, see Fig. 6.8. The intersection points are either computed based on the exact 6th order polynomial equations or using a set of linear segments for interpolation. In this model, due to the active set strategy, the reference coordinates spanning  $[s_0, s_1]$  are kept fixed, even though that they would change during Newton iterations.

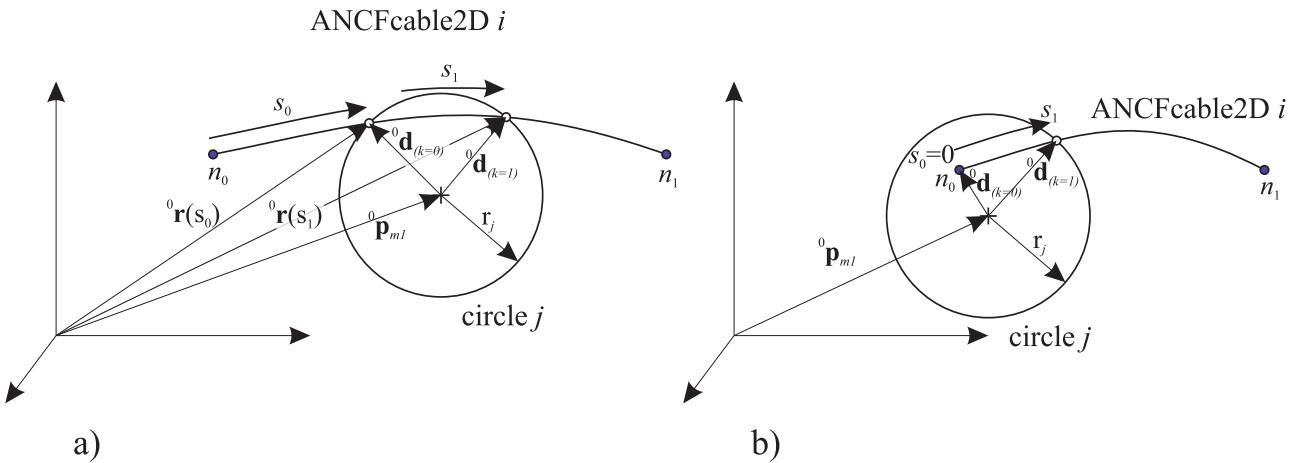


Figure 6.8: Geometrical relations for contact of ANCFcable2D  $i$  and circle  $j$  with according marker  $m1$ ; case a) shows a cable with nodes  $n_0$  and  $n_1$ , partially penetrating at the midspan of the cable; case b) shows the case of a cable where node  $n_0$  is inside the cable.

Normal contact and tangential friction forces are then computed based on integrals over the coordinates  $[s_0, s_1]$ . The integration is performed over  $n_{ip}$  integration points  $x_k \in [x_{i0}, x_{i1}, \dots]$ . In case of a 3 point Lobatto integration, we chose the integration points

$$x_k \in [s_0, (s_0 + s_1)/2, s_1] . \quad (6.71)$$

According weights are

$$w_k \in [1/3, 4/3, 1/3] . \quad (6.72)$$

The ANCF cable provides the global position of an integration point  $k \in \{i0, i1, \dots\}$  via

$${}^0\mathbf{r}(x_k) = {}^0\mathbf{S}(x_k) \mathbf{q} \quad (6.73)$$

with ANCF shape function matrix  $\mathbf{S}$  and current ANCF coordinates  $\mathbf{q}$ . Note that in the simplified case with linear segments,  ${}^0\mathbf{r}(x_k)$  is computed from linear interpolation of the segment which is attached to the cable. The velocity is computed in the same way,

$${}^0\dot{\mathbf{r}}(x_k) = {}^0\mathbf{S}(x_k) \dot{\mathbf{q}} \quad (6.74)$$

again using linear interpolation of the velocities along the straight segment, if linear segments are used.

In order to perform the integration of contact forces due to penetration as well as tangential (friction) forces, we iterate over all integration points, and sum up the according generalized forces on the cable and the circle marker object.

The integration factor for integration point  $k$  follows from

$$f_k = \frac{s_1 - s_0}{2} w_k , \quad (6.75)$$

assuming axial stretch of the cable element being moderately small. The vector  ${}^0\mathbf{d}_k$  which points from the center of the circle to the cable (integration) point reads

$${}^0\mathbf{d}_k = {}^0\mathbf{r}(x_k) - {}^0\mathbf{p}_j \quad (6.76)$$

The velocity of the circle at the contact integration point  $k$  follows as

$${}^0\mathbf{v}_{c,k} = {}^0\mathbf{v}_j + ({}^{0,m0}\mathbf{A} {}^{m0}\boldsymbol{\omega}_j) \times {}^0\mathbf{d}_k \quad (6.77)$$

The distance  $L$  between cable and circle center point, gap  $g$  and the contact normal vector read

$$L_k = |{}^0\mathbf{d}_k|, \quad g = L_k - (r + h_{1/2}), \quad {}^0\mathbf{d}_{0,k} = \frac{1}{L_k} {}^0\mathbf{d}_k \quad (6.78)$$

with the half height of the ANCF element  $h_{1/2}$ , which gives additional penetration. Note that this height is added on the side of the circle, which virtually represents a larger circle, behaving slightly different from a cable with thickness  $h$ .

The velocity in contact normal direction reads (note that we use the velocity of the circle's center point),

$$v_n = {}^0\mathbf{d}_{0,k} ({}^0\dot{\mathbf{r}}(x_k) - {}^0\mathbf{v}_j) \quad (6.79)$$

The contact force (tension! is always negative) follows in the simplistic case of a linear contact model as

$$f_{c,k} = k_c \cdot g + d \cdot v_n \quad (6.80)$$

with contact stiffness  $k_c$  and contact normal damping  $d_c$ .

In case of tangential friction, the tangential velocity reads

$$\begin{aligned}\mathbf{v}_{t,k} &= {}^0\dot{\mathbf{r}}(x_k) - {}^0\mathbf{v}_{c,k} - v_n \cdot {}^0\mathbf{d}_{0,k} = {}^0\dot{\mathbf{r}}(x_k) - {}^0\mathbf{v}_j - \left({}^{0,m0}\mathbf{A} {}^{m0}\boldsymbol{\omega}_j\right) \times {}^0\mathbf{d}_k - v_n \cdot {}^0\mathbf{d}_{0,k} \\ &= -\left({}^0\mathbf{d}_{0,k} \otimes {}^0\mathbf{d}_{0,k} - \mathbf{I}\right)\left({}^0\dot{\mathbf{r}}(x_k) - {}^0\mathbf{v}_{c,k}\right) + {}^0\tilde{\mathbf{d}}_k {}^{0,m0}\mathbf{A} {}^{m0}\boldsymbol{\omega}_j\end{aligned}\quad (6.81)$$

and the friction force is computed from Eq. (6.35) using the contact pressure  $-f_{c,k}$  from Eq. (6.80), while otherwise  ${}^0\mathbf{f}_f = \mathbf{0}$ .

The force vector for the contact point for integration point  $k$ , including integration weight  $f_k$ <sup>14</sup> thus reads

$${}^0\mathbf{f}_k = f_k \cdot \left(f_{c,k} \cdot {}^0\mathbf{d}_{0,k} + {}^0\mathbf{f}_f\right) \quad (6.82)$$

The total force and torque on the circle  $j$  is found by summation over all integration points  $k$ ,

$${}^0\mathbf{f}_{circ} = \sum_k {}^0\mathbf{f}_{circ,k} = \sum_k {}^0\mathbf{f}_k, \quad {}^0\mathbf{t}_{circ} = \sum_k {}^0\mathbf{t}_{circ,k} = \sum_k \left(r_j \cdot {}^0\mathbf{d}_{0,k}\right) \times {}^0\mathbf{f}_k \quad (6.83)$$

and the contribution to the generalized forces of the ANCF cable element (with generalized coordinates  $\mathbf{q}_{ANCF}$ ) read

$$\mathbf{f}_{ANCF} = \sum_k \mathbf{f}_{ANCF,k} = \sum_k {}^0\mathbf{S}(x_k)^T \cdot {}^0\mathbf{f}_k \quad (6.84)$$

The generalized LHS forces for marker  $m1$  (with generalized coordinates  $\mathbf{q}_{m1}$ ) thus read

$$\mathbf{f}_{m1,LHS} = {}^0\mathbf{J}_{pos,m1}^T {}^0\mathbf{f}_{circ} + {}^0\mathbf{J}_{rot,m1}^T {}^0\mathbf{t}_{circ}. \quad (6.85)$$

The Jacobian matrix for the circle-ANCF contact on position level thus reads<sup>15</sup>,

$$\mathbf{J}_c = \begin{bmatrix} \frac{\partial {}^0\mathbf{f}_{ANCF}}{\partial \mathbf{q}_{ANCF}} & \frac{\partial {}^0\mathbf{f}_{ANCF}}{\partial \mathbf{q}_{m1}} \\ -\frac{\partial \left({}^0\mathbf{J}_{pos,m1}^T {}^0\mathbf{f}_{circ} + {}^0\mathbf{J}_{rot,m1}^T {}^0\mathbf{t}_{circ}\right)}{\partial \mathbf{q}_{ANCF}} & \frac{\partial \left({}^0\mathbf{J}_{pos,m1}^T {}^0\mathbf{f}_{circ} + {}^0\mathbf{J}_{rot,m1}^T {}^0\mathbf{t}_{circ}\right)}{\partial \mathbf{q}_{m1}} \end{bmatrix} \quad (6.86)$$

and on velocity level, it follows as

$$\mathbf{J}_c = \begin{bmatrix} \frac{\partial {}^0\mathbf{f}_{ANCF}}{\partial \dot{\mathbf{q}}_{ANCF}} & \frac{\partial {}^0\mathbf{f}_{ANCF}}{\partial \dot{\mathbf{q}}_{m1}} \\ -\frac{\partial \left({}^0\mathbf{J}_{pos,m1}^T {}^0\mathbf{f}_{circ} + {}^0\mathbf{J}_{rot,m1}^T {}^0\mathbf{t}_{circ}\right)}{\partial \dot{\mathbf{q}}_{ANCF}} & \frac{\partial \left({}^0\mathbf{J}_{pos,m1}^T {}^0\mathbf{f}_{circ} + {}^0\mathbf{J}_{rot,m1}^T {}^0\mathbf{t}_{circ}\right)}{\partial \dot{\mathbf{q}}_{m1}} \end{bmatrix} \quad (6.87)$$

For the calculation of the jacobian, the derivatives of the following terms are needed:

- ${}^0\mathbf{d}_k = {}^0\mathbf{r}(x_k) - {}^0\mathbf{p}_j$ :

$$\frac{\partial {}^0\mathbf{d}_k}{\partial \mathbf{q}_{ANCF}} = \frac{\partial {}^0\mathbf{r}(x_k) - {}^0\mathbf{p}_j}{\partial \mathbf{q}_{ANCF}} = {}^0\mathbf{S}(x_k) \quad (6.88)$$

$$\frac{\partial {}^0\mathbf{d}_k}{\partial \mathbf{q}_{m1}} = \frac{\partial {}^0\mathbf{r}(x_k) - {}^0\mathbf{p}_j}{\partial \mathbf{q}_{m1}} = -{}^0\mathbf{J}_{pos,m1} \quad (6.89)$$

- ${}^0\dot{\mathbf{d}}_k = {}^0\dot{\mathbf{r}}(x_k) - {}^0\dot{\mathbf{v}}_j$ :

$$\frac{\partial {}^0\dot{\mathbf{d}}_k}{\partial \mathbf{q}_{ANCF}} = \frac{\partial {}^0\dot{\mathbf{r}}(x_k) - {}^0\mathbf{v}_j}{\partial \mathbf{q}_{ANCF}} = {}^0\mathbf{S}(x_k) \quad (6.90)$$

$$\frac{\partial {}^0\dot{\mathbf{d}}_k}{\partial \mathbf{q}_{m1}} = \frac{\partial {}^0\dot{\mathbf{r}}(x_k) - {}^0\mathbf{v}_j}{\partial \mathbf{q}_{m1}} = -{}^0\mathbf{J}_{pos,m1} \quad (6.91)$$

<sup>14</sup>this is done, because all further terms are proportional to  $\mathbf{f}_k$ .

<sup>15</sup>terms that are implemented are marked in green; black terms are not implemented or unused

- $L_k = |{}^0\mathbf{d}_k| = \left({}^0\mathbf{d}_k^\top {}^0\mathbf{d}_k\right)^{1/2}$ :

$$\frac{\partial |{}^0\mathbf{d}_k|}{\partial \mathbf{q}_{ANCF,m1}} = \frac{1}{L_k} \left( {}^0\mathbf{d}_k^\top \frac{\partial {}^0\mathbf{d}_k}{\partial \mathbf{q}_{ANCF,m1}} \right) = {}^0\mathbf{d}_{0,k}^\top \frac{\partial {}^0\mathbf{d}_k}{\partial \mathbf{q}_{ANCF,m1}} \quad (6.92)$$

- $L_k^{-1} = \left({}^0\mathbf{d}_k^\top {}^0\mathbf{d}_k\right)^{-1/2}$  (note different sign as in  $L$ -term due to  $-1/2$ ):

$$\frac{\partial L_k^{-1}}{\partial \mathbf{q}_{ANCF,m1}} = \frac{\partial \left({}^0\mathbf{d}_k^\top {}^0\mathbf{d}_k\right)^{-1/2}}{\partial \mathbf{q}_{ANCF,m1}} = -\frac{1}{L_k} \left( {}^0\mathbf{d}_{0,k}^\top \frac{\partial {}^0\mathbf{d}_k}{\partial \mathbf{q}_{ANCF,m1}} \right) \quad (6.93)$$

- ${}^0\mathbf{d}_{0,k} = \frac{1}{L_k} {}^0\mathbf{d}_k$ :

$$\frac{\partial {}^0\mathbf{d}_{0,k}}{\partial \mathbf{q}_{ANCF,m1}} = \frac{\partial \frac{1}{L_k} {}^0\mathbf{d}_k}{\partial \mathbf{q}_{ANCF,m1}} = -\frac{1}{L_k^2} {}^0\mathbf{d}_k \otimes \left( {}^0\mathbf{d}_{0,k}^\top \frac{\partial {}^0\mathbf{d}_k}{\partial \mathbf{q}_{ANCF,m1}} \right) + \frac{1}{L_k} \frac{\partial {}^0\mathbf{d}_k}{\partial \mathbf{q}_{ANCF,m1}} = \frac{1}{L_k} \left( \mathbf{I} - {}^0\mathbf{d}_{0,k} \otimes {}^0\mathbf{d}_{0,k} \right) \frac{\partial {}^0\mathbf{d}_k}{\partial \mathbf{q}_{ANCF,m1}} \quad (6.94)$$

- $g = L_k - (r + h_{1/2})$ :

$$\frac{\partial g}{\partial \mathbf{q}_{ANCF,m1}} = \frac{\partial L_k}{\partial \mathbf{q}_{ANCF,m1}} = {}^0\mathbf{d}_{0,k}^\top \frac{\partial {}^0\mathbf{d}_k}{\partial \mathbf{q}_{ANCF,m1}} \quad (6.95)$$

- velocity at circle contact point  $k$ :  ${}^0\mathbf{v}_{c,k} = {}^0\mathbf{v}_j + \left({}^{0,m1}\mathbf{A} {}^{m1}\boldsymbol{\omega}_j\right) \times {}^0\mathbf{d}_k = {}^0\mathbf{v}_j - {}^0\tilde{\mathbf{d}}_k {}^{0,m1}\mathbf{A} {}^{m1}\boldsymbol{\omega}_j$ :

$$\frac{\partial {}^0\mathbf{v}_{c,k}}{\partial \dot{\mathbf{q}}_{m1}} = {}^0\mathbf{J}_{pos,m1} - {}^0\tilde{\mathbf{d}}_k {}^0\mathbf{J}_{rot,m1} \quad (6.96)$$

- $v_n = {}^0\mathbf{d}_{0,k} \left( {}^0\dot{\mathbf{r}}(x_k) - {}^0\mathbf{v}_{c,k} \right)$ <sup>16</sup>

$$\frac{\partial v_n}{\partial \mathbf{q}_{ANCF,m1}} = {}^0\mathbf{d}_{0,k} \frac{\partial \left( {}^0\dot{\mathbf{r}}(x_k) - {}^0\mathbf{v}_{c,k} \right)}{\partial \dot{\mathbf{q}}_{ANCF}} \approx {}^0\mathbf{d}_{0,k}^\top \frac{\partial {}^0\mathbf{d}_k}{\partial \mathbf{q}_{ANCF,m1}} \quad (6.97)$$

- $\mathbf{v}_{t,k} = {}^0\dot{\mathbf{r}}(x_k) - {}^0\mathbf{v}_j - v_n \cdot {}^0\mathbf{d}_{0,k} = \left( \mathbf{I} - {}^0\mathbf{d}_{0,k} \otimes {}^0\mathbf{d}_{0,k} \right) \left( {}^0\dot{\mathbf{r}}(x_k) - {}^0\mathbf{v}_j \right) + {}^0\tilde{\mathbf{d}}_k {}^0\boldsymbol{\omega}_j$ :

$$\frac{\partial \mathbf{v}_{t,k}}{\partial \dot{\mathbf{q}}_{ANCF}} = \left( \mathbf{I} - {}^0\mathbf{d}_{0,k} \otimes {}^0\mathbf{d}_{0,k} \right) {}^0\mathbf{S}(x_k) \quad (6.98)$$

$$\frac{\partial \mathbf{v}_{t,k}}{\partial \dot{\mathbf{q}}_{m1}} = -\left( \mathbf{I} - {}^0\mathbf{d}_{0,k} \otimes {}^0\mathbf{d}_{0,k} \right) {}^0\mathbf{J}_{pos,m1} + {}^0\tilde{\mathbf{d}}_k {}^0\mathbf{J}_{rot,m1} \quad (6.99)$$

- $f_{c,k} = k_c \cdot g + d_c \cdot v_n$ :

$$\frac{\partial f_{c,k}}{\partial \mathbf{q}_{ANCF}} = k_c \cdot \frac{\partial g}{\partial \mathbf{q}_{ANCF}} + d_c \cdot \frac{\partial v_n}{\partial \mathbf{q}_{ANCF}} \approx k_c \cdot {}^0\mathbf{d}_{k,0}^\top {}^0\mathbf{S}(x_k) \quad (6.100)$$

$$\frac{\partial f_{c,k}}{\partial \dot{\mathbf{q}}_{m1}} = k_c \cdot \frac{\partial p}{\partial \dot{\mathbf{q}}_{m1}} + d_c \cdot \frac{\partial v_n}{\partial \dot{\mathbf{q}}_{m1}} \approx -k_c \cdot {}^0\mathbf{d}_{k,0}^\top {}^0\mathbf{J}_{pos,m1} \quad (6.101)$$

$$\frac{\partial f_{c,k}}{\partial \dot{\mathbf{q}}_{ANCF}} = d_c \cdot \frac{\partial v_n}{\partial \dot{\mathbf{q}}_{ANCF}} = d_c \cdot {}^0\mathbf{d}_{0,k} {}^0\mathbf{S}(x_k) \quad (6.102)$$

$$\frac{\partial f_{c,k}}{\partial \dot{\mathbf{q}}_{m1}} = d_c \cdot \frac{\partial v_n}{\partial \dot{\mathbf{q}}_{m1}} = -d_c \cdot {}^0\mathbf{d}_{0,k} {}^0\mathbf{J}_{pos,m1} \quad (6.103)$$

<sup>16</sup>the approximate sign is used, because  ${}^0\mathbf{v}_{c,k}$  includes a normal component if ANCF cable is not fully tangential, which is not considered here.

The contact force reads (note that because  $f_c$  is always negative, the sign of regularization term is negative),

$${}^0\mathbf{f}_k = f_k \cdot (f_{c,k} \cdot {}^0\mathbf{d}_{0,k} + \mathbf{f}_f) = \begin{cases} f_k \cdot f_{c,k} \left( {}^0\mathbf{d}_{0,k} - \frac{\mu_s}{v_{\mu,reg}} {}^0\mathbf{v}_t \right), & \text{if } |{}^0\mathbf{v}_t| < v_{\mu,reg} \\ f_k \cdot (f_{c,k} \cdot {}^0\mathbf{d}_{0,k} + \mathbf{f}_f), & \text{else with } \mathbf{f}_f = \text{const.} \end{cases} \quad (6.104)$$

We introduce a factor  $\delta_f$ , which is  $\delta_f = 1$  in the regularized small velocity state, and in the saturated (constant) friction force we use  $\delta_f = 0$ . Thus, the derivative of  $\mathbf{f}_f$ , using  $|f_{c,k}| = -f_{c,k}$ , reads:

$$\begin{aligned} \frac{\partial \mathbf{f}_f}{\partial \dot{\mathbf{q}}_{ANCF,m1}} &= \delta_f \frac{\mu_s}{v_{\mu,reg}} \frac{\partial (-f_{c,k} {}^0\mathbf{v}_t)}{\partial \dot{\mathbf{q}}_{ANCF,m1}} \approx -\delta_f f_{c,k} \frac{\mu_s}{v_{\mu,reg}} \frac{\partial {}^0\mathbf{v}_t}{\partial \dot{\mathbf{q}}_{ANCF,m1}} \\ \frac{\partial \mathbf{f}_f}{\partial \dot{\mathbf{q}}_{ANCF}} &\approx -\delta_f f_{c,k} \frac{\mu_s}{v_{\mu,reg}} (\mathbf{I} - {}^0\mathbf{d}_{0,k} \otimes {}^0\mathbf{d}_{0,k}) {}^0\mathbf{S}(x_k) \\ \frac{\partial \mathbf{f}_f}{\partial \dot{\mathbf{q}}_{m1}} &\approx -\delta_f f_{c,k} \frac{\mu_s}{v_{\mu,reg}} \left( -(\mathbf{I} - {}^0\mathbf{d}_{0,k} \otimes {}^0\mathbf{d}_{0,k}) {}^0\mathbf{J}_{pos,m1} + {}^0\tilde{\mathbf{d}}_k {}^0\mathbf{J}_{rot,m1} \right) \end{aligned} \quad (6.105)$$

The term  ${}^0\mathbf{f}_k$  gives:

$$\begin{aligned} \frac{\partial {}^0\mathbf{f}_k}{\partial \mathbf{q}_{ANCF,m1}} &= f_k \cdot \left( \left( {}^0\mathbf{d}_{0,k} - \frac{\mu_s}{v_{\mu,reg}} {}^0\mathbf{v}_t \right) \otimes \frac{\partial f_{c,k}}{\partial \mathbf{q}_{ANCF,m1}} + f_{c,k} \cdot \frac{\partial {}^0\mathbf{d}_{0,k}}{\partial \mathbf{q}_{ANCF,m1}} + \frac{\partial {}^0\mathbf{f}_f}{\partial \mathbf{q}_{ANCF,m1}} \right) \\ &\approx f_k \cdot k_c \cdot \left( \left( {}^0\mathbf{d}_{0,k} - \frac{\mu_s}{v_{\mu,reg}} {}^0\mathbf{v}_t \right) \otimes {}^0\mathbf{d}_{0,k} \frac{\partial {}^0\mathbf{d}_k}{\partial \mathbf{q}_{ANCF,m1}} \right) \end{aligned} \quad (6.106)$$

and the velocity terms yield

$$\begin{aligned} \frac{\partial {}^0\mathbf{f}_k}{\partial \dot{\mathbf{q}}_{ANCF}} &= f_k \cdot \left( d_c \cdot \left( {}^0\mathbf{d}_{0,k} - \frac{\mu_s}{v_{\mu,reg}} {}^0\mathbf{v}_t \right) \otimes \frac{\partial v_n}{\partial \dot{\mathbf{q}}_{ANCF}} + \frac{\partial {}^0\mathbf{f}_f}{\partial \dot{\mathbf{q}}_{ANCF}} \right) \\ &\approx f_k \cdot \left( d_c \cdot \left( {}^0\mathbf{d}_{0,k} - \frac{\mu_s}{v_{\mu,reg}} {}^0\mathbf{v}_t \right) \otimes {}^0\mathbf{d}_{0,k} - \delta_f f_{c,k} \frac{\mu_s}{v_{\mu,reg}} (\mathbf{I} - {}^0\mathbf{d}_{0,k} \otimes {}^0\mathbf{d}_{0,k}) \right) \frac{\partial {}^0\dot{\mathbf{d}}_k}{\partial \dot{\mathbf{q}}_{ANCF}} \end{aligned} \quad (6.107)$$

$$\begin{aligned} \frac{\partial {}^0\mathbf{f}_k}{\partial \dot{\mathbf{q}}_{m1}} &= f_k \cdot \left( d_c \cdot \left( {}^0\mathbf{d}_{0,k} - \frac{\mu_s}{v_{\mu,reg}} {}^0\mathbf{v}_t \right) \otimes \frac{\partial v_n}{\partial \dot{\mathbf{q}}_{m1}} + \frac{\partial {}^0\mathbf{f}_f}{\partial \dot{\mathbf{q}}_{m1}} \right) \\ &\approx f_k \cdot \left( \left( d_c \cdot \left( {}^0\mathbf{d}_{0,k} - \frac{\mu_s}{v_{\mu,reg}} {}^0\mathbf{v}_t \right) \otimes {}^0\mathbf{d}_{0,k} - \delta_f f_{c,k} \frac{\mu_s}{v_{\mu,reg}} (\mathbf{I} - {}^0\mathbf{d}_{0,k} \otimes {}^0\mathbf{d}_{0,k}) \right) \frac{\partial {}^0\dot{\mathbf{d}}_k}{\partial \dot{\mathbf{q}}_{m1}} \right. \\ &\quad \left. + \delta_f f_{c,k} \frac{\mu_s}{v_{\mu,reg}} {}^0\tilde{\mathbf{d}}_k {}^0\mathbf{J}_{rot,m1} \right) \end{aligned} \quad (6.108)$$

The single jacobian terms w.r.t.  $\mathbf{q}_{ANCF}$  and  $\mathbf{q}_{m1}$  may be expressed as

$$\begin{aligned} \frac{\partial \left( {}^0\mathbf{J}_{pos,m1}^T {}^0\mathbf{f}_{circ} + {}^0\mathbf{J}_{rot,m1}^T {}^0\mathbf{t}_{circ} \right)}{\partial \mathbf{q}_{ANCF,m1}} &= \\ \frac{\partial {}^0\mathbf{J}_{pos,m1}^T {}^0\mathbf{f}_{circ}}{\partial \mathbf{q}_{ANCF,m1}} + {}^0\mathbf{J}_{pos,m1}^T \frac{\partial {}^0\mathbf{f}_{circ}}{\partial \mathbf{q}_{ANCF,m1}} + \frac{\partial {}^0\mathbf{J}_{rot,m1}^T {}^0\mathbf{t}_{circ}}{\partial \mathbf{q}_{m1}} + {}^0\mathbf{J}_{rot,m1}^T \frac{\partial {}^0\mathbf{t}_{circ}}{\partial \mathbf{q}_{ANCF,m1}} & \end{aligned} \quad (6.109)$$

Note that similar relations follow for the time derivatives  $\frac{\partial}{\partial \dot{\mathbf{q}}_{ANCF,m1}}$ .

The derivatives of  ${}^0\mathbf{f}_{ANCF}$ ,  ${}^0\mathbf{f}_{circ}$ , and

$${}^0\mathbf{t}_{circ} = \sum_k r_j \cdot {}^0\mathbf{d}_{0,k} \times {}^0\mathbf{f}_k = \sum_k r_j \cdot {}^0\tilde{\mathbf{d}}_{0,k} {}^0\mathbf{f}_k \quad (6.110)$$

follow from

$$\begin{aligned}
 \frac{\partial^0 \mathbf{f}_{ANCF}}{\partial \mathbf{q}_{ANCF,m1}} &= \sum_k {}^0 \mathbf{S}(x_k)^T \frac{\partial^0 \mathbf{f}_k}{\partial \mathbf{q}_{ANCF,m1}}, \\
 \frac{\partial^0 \mathbf{f}_{circ}}{\partial \mathbf{q}_{ANCF,m1}} &= \sum_k \frac{\partial^0 \mathbf{f}_k}{\partial \mathbf{q}_{ANCF,m1}}, \\
 \frac{\partial^0 \mathbf{t}_{circ}}{\partial \mathbf{q}_{ANCF,m1}} &\approx \sum_k \left( r_j \cdot {}^0 \tilde{\mathbf{d}}_{0,k} \frac{\partial^0 \mathbf{f}_k}{\partial \mathbf{q}_{ANCF,m1}} - r_j \cdot {}^0 \tilde{\mathbf{f}}_k \frac{\partial^0 \mathbf{d}_{0,k}}{\partial \mathbf{q}_{ANCF,m1}} \right)
 \end{aligned} \tag{6.111}$$



## Chapter 7

# Objects, nodes, markers, loads and sensors reference manual

This chapter includes the reference manual for all objects (bodies/constraints), nodes, markers, loads and sensors (=**items**). For description of types (e.g., the meaning of `Vector3D` or `NumpyMatrix`), see [Section 6.1.1](#).

## 7.1 Nodes

Nodes provide coordinates for objects. Loads can be applied and Markers or Sensors can be attached to Nodes. The sorting of Nodes in the system (the order they are added to mbs) defines the order of system coordinates.

### 7.1.1 NodePoint

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

**Additional information for NodePoint:**

- The Node has the following types = **Position**
- **Short name for Python** = **Point**
- **Short name for Python (visualization object)** = **VPoint**

The item **NodePoint** with type = 'Point' has the following parameters:

Name	type	size	default value	description
name	String		"	node's unique name
referenceCoordinates	Vector3D	3	[0.,0.,0.]	reference coordinates of node, e.g. ref. coordinates for finite elements; global position of node without displacement
initialCoordinates	Vector3D	3	[0.,0.,0.]	initial displacement coordinate
initialVelocities	Vector3D	3	[0.,0.,0.]	initial velocity coordinate
visualization	VNodePoint			parameters for visualization of item

The item **VNodePoint** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size (diameter, dimensions of underlying cube, etc.) for item; size == -1.f means that default size is used
color	Float4	4	[-1.,-1.,-1.,-1.]	Default RGBA color for nodes; 4th value is alpha-transparency; R=-1.f means, that default color is used

---

#### 7.1.1.1 DESCRIPTION of NodePoint:

**Information on input parameters:**

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

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

output variable	symbol	description
Position	$\mathbf{p}_{\text{config}} = [p_0, p_1, p_2]_{\text{config}}^T = \mathbf{u}_{\text{config}} + \mathbf{p}_{\text{ref}}$	global 3D position vector of node; $\mathbf{u}_{\text{ref}} = 0$
Displacement	$\mathbf{u}_{\text{config}} = [q_0, q_1, q_2]_{\text{config}}^T$	global 3D displacement vector of node
Velocity	$\mathbf{v}_{\text{config}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2]_{\text{config}}^T$	global 3D velocity vector of node
Acceleration	$\mathbf{a}_{\text{config}} = \ddot{\mathbf{q}}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, \ddot{q}_2]_{\text{config}}^T$	global 3D acceleration vector of node
Coordinates	$\mathbf{c}_{\text{config}} = \mathbf{u}_{\text{config}} = [q_0, q_1, q_2]_{\text{config}}^T$	coordinate vector of node
Coordinates_t	$\dot{\mathbf{c}}_{\text{config}} = \mathbf{v}_{\text{config}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2]_{\text{config}}^T$	velocity coordinates vector of node
Coordinates_tt	$\ddot{\mathbf{c}}_{\text{config}} = \mathbf{a}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, \ddot{q}_2]_{\text{config}}^T$	acceleration coordinates vector of node

**Detailed information:** The node provides  $n_c = 3$  displacement coordinates. Equations of motion need to be provided by an according object (e.g., MassPoint, finite elements, ...). Usually, the nodal coordinates are provided in the global frame. However, the coordinate system is defined by the object (e.g. MassPoint uses global coordinates, but floating frame of reference objects use local frames). Note that for this very simple node, coordinates are identical to the nodal displacements, same for time derivatives. This is not the case, e.g. for nodes with orientation.

**Example** for NodePoint: see ObjectMassPoint, [Section 7.2.2](#)

For examples on NodePoint see Examples and TestModels:

- [interactiveTutorial.py](#) (Examples/)
- [particlesTest.py](#) (Examples/)
- [particlesTest3D.py](#) (Examples/)
- [particlesTest3D2.py](#) (Examples/)
- [plotSensorExamples.py](#) (Examples/)
- [Spring\\_with\\_constraints.py](#) (Examples/)
- [cartesianSpringDamper.py](#) (Examples/)
- [ComputeSensitivitiesExample.py](#) (Examples/)
- [coordinateSpringDamper.py](#) (Examples/)
- [massSpringFrictionInteractive.py](#) (Examples/)
- [minimizeExample.py](#) (Examples/)
- [nMassOscillatorInteractive.py](#) (Examples/)
- ...
- [connectorGravityTest.py](#) (TestModels/)
- [generalContactSpheresTest.py](#) (TestModels/)
- [objectGenericODE2Test.py](#) (TestModels/)
- ...

## 7.1.2 NodePoint2D

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

**Additional information for NodePoint2D:**

- The Node has the following types = Position2D, Position
- **Short name for Python** = **Point2D**
- **Short name for Python (visualization object)** = **VPoint2D**

The item **NodePoint2D** with type = 'Point2D' has the following parameters:

Name	type	size	default value	description
name	String		"	node's unique name
referenceCoordinates	Vector2D	2	[0.,0.]	reference coordinates of node ==> e.g. ref. coordinates for finite elements; global position of node without displacement
initialCoordinates	Vector2D	2	[0.,0.]	initial displacement coordinate
initialVelocities	Vector2D	2	[0.,0.]	initial velocity coordinate
visualization	VNodePoint2D			parameters for visualization of item

The item VNodePoint2D has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size (diameter, dimensions of underlying cube, etc.) for item; size == -1.f means that default size is used
color	Float4	4	[-1.,-1.,-1.,-1.]	Default RGBA color for nodes; 4th value is alpha-transparency; R=-1.f means, that default color is used

### 7.1.2.1 DESCRIPTION of NodePoint2D:

**Information on input parameters:**

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

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

output variable	symbol	description
Position	$\mathbf{p}_{\text{config}} = [p_0, p_1, 0]^T_{\text{config}} = \mathbf{u}_{\text{config}} + \mathbf{p}_{\text{ref}}$	global 3D position vector of node; $\mathbf{u}_{\text{ref}} = 0$
Displacement	$\mathbf{u}_{\text{config}} = [q_0, q_1, 0]^T_{\text{config}}$	global 3D displacement vector of node
Velocity	$\mathbf{v}_{\text{config}} = [\dot{q}_0, \dot{q}_1, 0]^T_{\text{config}}$	global 3D velocity vector of node
Acceleration	$\mathbf{a}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, 0]^T_{\text{config}}$	global 3D acceleration vector of node
Coordinates	$\mathbf{c}_{\text{config}} = [q_0, q_1]^T_{\text{config}}$	coordinate vector of node
Coordinates_t	$\dot{\mathbf{c}}_{\text{config}} = [\dot{q}_0, \dot{q}_1]^T_{\text{config}}$	velocity coordinates vector of node
Coordinates_tt	$\ddot{\mathbf{c}}_{\text{config}} = \mathbf{a}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1]^T_{\text{config}}$	acceleration coordinates vector of node

**Detailed information:** The node provides  $n_c = 2$  displacement coordinates. Equations of motion need to be provided by an according object (e.g., MassPoint2D). Coordinates are identical to the nodal displacements, except for the third coordinate  $u_2$ , which is zero, because  $q_2$  does not exist.

Note that for this very simple node, coordinates are identical to the nodal displacements, same for time derivatives. This is not the case, e.g. for nodes with orientation.

Example for NodePoint2D: see ObjectMassPoint2D, [Section 7.2.3](#)

For examples on NodePoint2D see Examples and TestModels:

- [myFirstExample.py](#) (Examples/)
- [pendulum2Dconstraint.py](#) (Examples/)
- [sliderCrank3DwithANCFbeltDrive2.py](#) (Examples/)
- [SpringDamperMassUserFunction.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- [ANCF\\_slidingJoint2Drigid.py](#) (Examples/)
- [geneticOptimizationSliderCrank.py](#) (Examples/)
- [SliderCrank.py](#) (Examples/)
- [slidercrankWithMassSpring.py](#) (Examples/)
- [switchingConstraintsPendulum.py](#) (Examples/)
- [modelUnitTests.py](#) (TestModels/)
- [sparseMatrixSpringDamperTest.py](#) (TestModels/)
- [coordinateVectorConstraint.py](#) (TestModels/)
- [coordinateVectorConstraintGenericODE2.py](#) (TestModels/)
- [sliderCrankFloatingTest.py](#) (TestModels/)
- ...

### 7.1.3 NodeRigidBodyEP

A 3D rigid body node based on Euler parameters for rigid bodies or beams; the node has 3 displacement coordinates (representing displacement of reference point  ${}^0\mathbf{r}$ ) and four rotation coordinates (Euler parameters = unit quaternions).

**Additional information for NodeRigidBodyEP:**

- The Node has the following types = Position, Orientation, RigidBody, RotationEulerParameters
- **Short name for Python** = **RigidEP**
- **Short name for Python** (visualization object) = **VRigidEP**

The item **NodeRigidBodyEP** with type = 'RigidBodyEP' has the following parameters:

Name	type	size	default value	description
name	String		"	node's unique name
referenceCoordinates	Vector7D	7	[0.,0.,0., 0.,0.,0.]	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.]	initial displacement coordinates and 4 Euler parameters relative to reference coordinates
initialVelocities	Vector7D	7	[0.,0.,0., 0.,0.,0.]	initial velocity coordinates: time derivatives of initial displacements and Euler parameters
addConstraintEquation	Bool		True	True: automatically add Euler parameter constraint for node; False: Euler parameter constraint is not added, must be done manually (e.g., with CoordinateVectorConstraint)
visualization	VNodeRigidBodyEP			parameters for visualization of item

The item **VNodeRigidBodyEP** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size (diameter, dimensions of underlying cube, etc.) for item; size == -1.f means that default size is used
color	Float4	4	[-1.,-1.,-1.,-1.]	Default RGBA color for nodes; 4th value is alpha-transparency; R=-1.f means, that default color is used

---

### 7.1.3.1 DESCRIPTION of NodeRigidBodyEP:

#### Information on input parameters:

input parameter	symbol	description see tables above
referenceCoordinates	$\mathbf{q}_{\text{ref}} = [q_0, q_1, q_2, \psi_0, \psi_1, \psi_2, \psi_3]_{\text{ref}}^T = [\mathbf{p}_{\text{ref}}^T, \boldsymbol{\psi}_{\text{ref}}^T]^T$	
initialCoordinates	$\mathbf{q}_{\text{ini}} = [q_0, q_1, q_2, \psi_0, \psi_1, \psi_2, \psi_3]_{\text{ini}}^T = [\mathbf{u}_{\text{ini}}^T, \boldsymbol{\psi}_{\text{ini}}^T]^T$	
initialVelocities	$\dot{\mathbf{q}}_{\text{ini}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2, \dot{\psi}_0, \dot{\psi}_1, \dot{\psi}_2, \dot{\psi}_3]_{\text{ini}}^T = [\dot{\mathbf{u}}_{\text{ini}}^T, \dot{\boldsymbol{\psi}}_{\text{ini}}^T]^T$	

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

output variable	symbol	description
Position	${}^0\mathbf{p}_{\text{config}} = {}^0[p_0, p_1, p_2]_{\text{config}}^T = {}^0\mathbf{u}_{\text{config}} + {}^0\mathbf{p}_{\text{ref}}$	global 3D position vector of node; $\mathbf{u}_{\text{ref}} = 0$
Displacement	${}^0\mathbf{u}_{\text{config}} = [q_0, q_1, q_2]_{\text{config}}^T$	global 3D displacement vector of node
Velocity	${}^0\mathbf{v}_{\text{config}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2]_{\text{config}}^T$	global 3D velocity vector of node
Acceleration	${}^0\mathbf{a}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, \ddot{q}_2]_{\text{config}}^T$	global 3D acceleration vector of node
Coordinates	$\mathbf{c}_{\text{config}} = [q_0, q_1, q_2, \psi_0, \psi_1, \psi_2, \psi_3]_{\text{config}}^T$	coordinate vector of node, having 3 displacement coordinates and 4 Euler parameters
Coordinates_t	$\dot{\mathbf{c}}_{\text{config}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2, \dot{\psi}_0, \dot{\psi}_1, \dot{\psi}_2, \dot{\psi}_3]_{\text{config}}^T$	velocity coordinates vector of node
Coordinates_tt	$\ddot{\mathbf{c}}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, \ddot{q}_2, \ddot{\psi}_0, \ddot{\psi}_1, \ddot{\psi}_2, \ddot{\psi}_3]_{\text{config}}^T$	acceleration coordinates vector of node
RotationMatrix	$[A_{00}, A_{01}, A_{02}, A_{10}, \dots, A_{21}, A_{22}]_{\text{config}}^T$	vector with 9 components of the rotation matrix ${}^0b\mathbf{A}_{\text{config}}$ in row-major format, in any configuration; the rotation matrix transforms local ( $b$ ) to global ( $0$ ) coordinates
Rotation	$[\varphi_0, \varphi_1, \varphi_2]_{\text{config}}^T$	vector with 3 components of the Euler angles in xyz-sequence ( ${}^0b\mathbf{A}_{\text{config}} =: \mathbf{A}_0(\varphi_0) \cdot \mathbf{A}_1(\varphi_1) \cdot \mathbf{A}_2(\varphi_2)$ ), recomputed from rotation matrix
AngularVelocity	${}^0\boldsymbol{\omega}_{\text{config}} = {}^0[\omega_0, \omega_1, \omega_2]_{\text{config}}^T$	global 3D angular velocity vector of node
AngularVelocityLocal	${}^b\boldsymbol{\omega}_{\text{config}} = {}^b[\omega_0, \omega_1, \omega_2]_{\text{config}}^T$	local (body-fixed) 3D angular velocity vector of node
AngularAcceleration	${}^0\boldsymbol{\alpha}_{\text{config}} = {}^0[\alpha_0, \alpha_1, \alpha_2]_{\text{config}}^T$	global 3D angular acceleration vector of node

**Detailed information:** All coordinates  $\mathbf{c}_{\text{config}}$  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

$$1 - \sum_{i=0}^3 \theta_i^2 = 0. \quad (7.1)$$

The rotation matrix  ${}^0b\mathbf{A}_{\text{config}}$  transforms a local (body-fixed) 3D position  ${}^b\mathbf{b} = {}^b[b_0, b_1, b_2]^T$  to global 3D positions,

$${}^0\mathbf{b}_{\text{config}} = {}^0b\mathbf{A}_{\text{config}} {}^b\mathbf{b} \quad (7.2)$$

Note that the Euler parameters  $\theta_{\text{cur}}$  are computed as sum of current coordinates plus reference coordinates,

$$\theta_{\text{cur}} = \psi_{\text{cur}} + \psi_{\text{ref}}. \quad (7.3)$$

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

$${}^0b\mathbf{A} = \begin{bmatrix} -2\theta_3^2 - 2\theta_2^2 + 1 & -2\theta_3\theta_0 + 2\theta_2\theta_1 & 2 * \theta_3\theta_1 + 2 * \theta_2\theta_0 \\ 2\theta_3\theta_0 + 2\theta_2\theta_1 & -2\theta_3^2 - 2\theta_1^2 + 1 & 2\theta_3\theta_2 - 2\theta_1\theta_0 \\ -2\theta_2\theta_0 + 2\theta_3\theta_1 & 2\theta_3\theta_2 + 2\theta_1\theta_0 & -2\theta_2^2 - 2\theta_1^2 + 1 \end{bmatrix} \quad (7.4)$$

The derivatives of the angular velocity vectors w.r.t. the rotation velocity coordinates  $\dot{\theta} = [\dot{\theta}_0, \dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_3]^T$  lead to the  $\mathbf{G}$  matrices, as used in the equations of motion for rigid bodies,

$${}^0\boldsymbol{\omega} = {}^0\mathbf{G} \dot{\theta}, \quad (7.5)$$

$${}^b\boldsymbol{\omega} = {}^b\mathbf{G} \dot{\theta}. \quad (7.6)$$

For creating a `NodeRigidBodyEP`, there is a `rigidBodyUtilities` function `AddRigidBody`, see [Section 5.13](#), which simplifies the setup of a rigid body significantly!

For examples on `NodeRigidBodyEP` see Examples and TestModels:

- [`rigid3Dexample.py`](#) (Examples/)
- [`rigidBodyIMUtest.py`](#) (Examples/)
- [`rigidRotor3DbasicBehaviour.py`](#) (Examples/)
- [`rigidRotor3DFWBW.py`](#) (Examples/)
- [`rigidRotor3Dnutation.py`](#) (Examples/)
- [`rigidRotor3Drunup.py`](#) (Examples/)
- [`addPrismaticJoint.py`](#) (Examples/)
- [`addRevoluteJoint.py`](#) (Examples/)
- [`bicycleIftommBenchmark.py`](#) (Examples/)
- [`fourBarMechanism3D.py`](#) (Examples/)
- [`gyroStability.py`](#) (Examples/)
- [`leggedRobot.py`](#) (Examples/)
- ...
- [`explicitLieGroupIntegratorPythonTest.py`](#) (TestModels/)
- [`explicitLieGroupIntegratorTest.py`](#) (TestModels/)
- [`explicitLieGroupMBSTest.py`](#) (TestModels/)
- ...

### 7.1.4 NodeRigidBodyRxyz

A 3D rigid body node based on Euler / Tait-Bryan angles for rigid bodies or beams; all coordinates lead to second order differential equations; NOTE that this node has a singularity if the second rotation parameter reaches  $\psi_1 = (2k - 1)\pi/2$ , with  $k \in \mathbb{N}$  or  $-k \in \mathbb{N}$ .

**Additional information for NodeRigidBodyRxyz:**

- The Node has the following types = Position, Orientation, RigidBody, RotationRxyz
- **Short name for Python** = **RigidRxyz**
- **Short name for Python (visualization object)** = **VRigidRxyz**

The item **NodeRigidBodyRxyz** with type = 'RigidBodyRxyz' has the following parameters:

Name	type	size	default value	description
name	String		"	node's unique name
referenceCoordinates	Vector6D	6	[0.,0.,0., 0.,0.,0.]	reference coordinates (3 position and 3 xyz Euler angles) of node ==> e.g. ref. coordinates for finite elements or reference position of rigid body (e.g. for definition of joints)
initialCoordinates	Vector6D	6	[0.,0.,0., 0.,0.,0.]	initial displacement coordinates: ux,uy,uz and 3 Euler angles (xyz) relative to reference coordinates
initialVelocities	Vector6D	6	[0.,0.,0., 0.,0.,0.]	initial velocity coordinate: time derivatives of ux,uy,uz and of 3 Euler angles (xyz)
visualization	VNodeRigidBodyRxyz			parameters for visualization of item

The item **VNodeRigidBodyRxyz** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size (diameter, dimensions of underlying cube, etc.) for item; size == -1.f means that default size is used
color	Float4	4	[-1.,-1.,-1.,-1.]	Default RGBA color for nodes; 4th value is alpha-transparency; R=-1.f means, that default color is used

---

#### 7.1.4.1 DESCRIPTION of NodeRigidBodyRxyz:

**Information on input parameters:**

input parameter	symbol	description see tables above
referenceCoordinates	$\mathbf{q}_{\text{ref}} = [q_0, q_1, q_2, \psi_0, \psi_1, \psi_2]_{\text{ref}}^T = [\mathbf{p}_{\text{ref}}^T, \boldsymbol{\psi}_{\text{ref}}^T]^T$	
initialCoordinates	$\mathbf{q}_{\text{ini}} = [q_0, q_1, q_2, \psi_0, \psi_1, \psi_2]_{\text{ini}}^T = [\mathbf{u}_{\text{ini}}^T, \boldsymbol{\psi}_{\text{ini}}^T]^T$	
initialVelocities	$\dot{\mathbf{q}}_{\text{ini}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2, \dot{\psi}_0, \dot{\psi}_1, \dot{\psi}_2]_{\text{ini}}^T = [\dot{\mathbf{u}}_{\text{ini}}^T, \dot{\boldsymbol{\psi}}_{\text{ini}}^T]^T$	

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

output variable	symbol	description
Position	${}^0\mathbf{p}_{\text{config}} = {}^0[p_0, p_1, p_2]_{\text{config}}^T = {}^0\mathbf{u}_{\text{config}} + {}^0\mathbf{p}_{\text{ref}}$	global 3D position vector of node; $\mathbf{u}_{\text{ref}} = 0$
Displacement	${}^0\mathbf{u}_{\text{config}} = [q_0, q_1, q_2]_{\text{config}}^T$	global 3D displacement vector of node
Velocity	${}^0\mathbf{v}_{\text{config}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2]_{\text{config}}^T$	global 3D velocity vector of node
Acceleration	${}^0\mathbf{a}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, \ddot{q}_2]_{\text{config}}^T$	global 3D acceleration vector of node
Coordinates	$\mathbf{c}_{\text{config}} = [q_0, q_1, q_2, \psi_0, \psi_1, \psi_2]_{\text{config}}^T$	coordinate vector of node, having 3 displacement coordinates and 3 Euler angles
Coordinates_t	$\dot{\mathbf{c}}_{\text{config}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2, \dot{\psi}_0, \dot{\psi}_1, \dot{\psi}_2]_{\text{config}}^T$	velocity coordinates vector of node
Coordinates_tt	$\ddot{\mathbf{c}}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, \ddot{q}_2, \ddot{\psi}_0, \ddot{\psi}_1, \ddot{\psi}_2]_{\text{config}}^T$	acceleration coordinates vector of node
RotationMatrix	$[A_{00}, A_{01}, A_{02}, A_{10}, \dots, A_{21}, A_{22}]_{\text{config}}^T$	vector with 9 components of the rotation matrix ${}^{0b}\mathbf{A}_{\text{config}}$ in row-major format, in any configuration; the rotation matrix transforms local ( $b$ ) to global ( $0$ ) coordinates
Rotation	$[\varphi_0, \varphi_1, \varphi_2]_{\text{config}}^T = [\psi_0, \psi_1, \psi_2]_{\text{ref}}^T + [\psi_0, \psi_1, \psi_2]_{\text{config}}^T$	vector with 3 components of the Euler / Tait-Bryan angles in xyz-sequence ( ${}^{0b}\mathbf{A}_{\text{config}} =: \mathbf{A}_0(\varphi_0) \cdot \mathbf{A}_1(\varphi_1) \cdot \mathbf{A}_2(\varphi_2)$ ), recomputed from rotation matrix
AngularVelocity	${}^0\boldsymbol{\omega}_{\text{config}} = {}^0[\omega_0, \omega_1, \omega_2]_{\text{config}}^T$	global 3D angular velocity vector of node
AngularVelocityLocal	${}^b\boldsymbol{\omega}_{\text{config}} = {}^b[\omega_0, \omega_1, \omega_2]_{\text{config}}^T$	local (body-fixed) 3D angular velocity vector of node
AngularAcceleration	${}^0\boldsymbol{\alpha}_{\text{config}} = {}^0[\alpha_0, \alpha_1, \alpha_2]_{\text{config}}^T$	global 3D angular acceleration vector of node

**Detailed information:** The node has 3 displacement coordinates  $[q_0, q_1, q_2]^T$  and 3 rotation coordinates  $[\psi_0, \psi_1, \psi_2]^T$  for consecutive rotations around the 0, 1 and 2-axis ( $x$ ,  $y$  and  $z$ ). All coordinates  $\mathbf{c}_{\text{config}}$  lead to second order differential equations. The rotation matrix  ${}^{0b}\mathbf{A}_{\text{config}}$  transforms a local (body-fixed) 3D position  ${}^b\mathbf{b} = {}^b[b_0, b_1, b_2]^T$  to global 3D positions,

$${}^0\mathbf{b}_{\text{config}} = {}^{0b}\mathbf{A}_{\text{config}} {}^b\mathbf{b} \quad (7.7)$$

Note that the Euler angles  $\theta_{\text{cur}}$  are computed as sum of current coordinates plus reference coordinates,

$$\theta_{\text{cur}} = \boldsymbol{\psi}_{\text{cur}} + \boldsymbol{\psi}_{\text{ref}}. \quad (7.8)$$

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

$${}^{0b}\mathbf{A} = {}^{01}\mathbf{A}_0(\theta_0) {}^{12}\mathbf{A}_1(\theta_1) {}^{2b}\mathbf{A}_2(\theta_2) \quad (7.9)$$

see [Section 6.1.4](#) for definition of rotation matrices  $\mathbf{A}_0$ ,  $\mathbf{A}_1$  and  $\mathbf{A}_2$ .

The derivatives of the angular velocity vectors w.r.t. the rotation velocity coordinates  $\dot{\theta} = [\dot{\theta}_0, \dot{\theta}_1, \dot{\theta}_2]^T$  lead to the  $\mathbf{G}$  matrices, as used in the equations of motion for rigid bodies,

$${}^0\omega = {}^0\mathbf{G} \dot{\theta}, \quad (7.10)$$

$${}^b\omega = {}^b\mathbf{G} \dot{\theta}. \quad (7.11)$$

For creating a `NodeRigidBodyRxyz`, there is a `rigidBodyUtilities` function `AddRigidBody`, see [Section 5.13](#), which simplifies the setup of a rigid body significantly!

---

For examples on `NodeRigidBodyRxyz` see Examples and TestModels:

- [`performanceMultiThreadingNG.py`](#) (Examples/)
- [`explicitLieGroupIntegratorPythonTest.py`](#) (TestModels/)
- [`explicitLieGroupIntegratorTest.py`](#) (TestModels/)
- [`explicitLieGroupMBSTest.py`](#) (TestModels/)
- [`heavyTop.py`](#) (TestModels/)
- [`connectorRigidBodySpringDamperTest.py`](#) (TestModels/)

### 7.1.5 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.

Authors: Gerstmayr Johannes, Holzinger Stefan

**Additional information for NodeRigidBodyRotVecLG:**

- The Node has the following types = `Position`, `Orientation`, `RigidBody`, `RotationRotationVector`, `RotationLieGroup`
- **Short name for Python** = `RigidRotVecLG`
- **Short name for Python (visualization object)** = `VRigidRotVecLG`

The item `NodeRigidBodyRotVecLG` with type = 'RigidBodyRotVecLG' has the following parameters:

Name	type	size	default value	description
name	String		"	node's unique name
referenceCoordinates	Vector6D	3	[0.,0.,0., 0.,0.,0.]	reference coordinates (position and rotation vector $\nu$ ) 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 $\nu$ 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

---

#### 7.1.5.1 DESCRIPTION of NodeRigidBodyRotVecLG:

**Information on input parameters:**

input parameter	symbol	description see tables above
referenceCoordinates	$\mathbf{q}_{\text{ref}} = [q_0, q_1, q_2, v_0, v_1, v_2]_{\text{ref}}^T = [\mathbf{p}_{\text{ref}}^T, \mathbf{v}_{\text{ref}}^T]^T$	
initialCoordinates	$\mathbf{q}_{\text{ini}} = [q_0, q_1, q_2, v_0, v_1, v_2]_{\text{ini}}^T = [\mathbf{u}_{\text{ini}}^T, \mathbf{v}_{\text{ini}}^T]^T$	
initialVelocities	$\dot{\mathbf{q}}_{\text{ini}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2, \dot{v}_0, \dot{v}_1, \dot{v}_2]_{\text{ini}}^T = [\dot{\mathbf{u}}_{\text{ini}}^T, \dot{\mathbf{v}}_{\text{ini}}^T]^T$	

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

output variable	symbol	description
Position	${}^0\mathbf{p}_{\text{config}} = {}^0[p_0, p_1, p_2]_{\text{config}}^T = {}^0\mathbf{u}_{\text{config}} + {}^0\mathbf{p}_{\text{ref}}$	global 3D position vector of node; $\mathbf{u}_{\text{ref}} = 0$
Displacement	${}^0\mathbf{u}_{\text{config}} = [q_0, q_1, q_2]_{\text{config}}^T$	global 3D displacement vector of node
Velocity	${}^0\mathbf{v}_{\text{config}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2]_{\text{config}}^T$	global 3D velocity vector of node
Coordinates	$\mathbf{c}_{\text{config}} = [q_0, q_1, q_2, v_0, v_1, v_2]_{\text{config}}^T$	coordinate vector of node, having 3 displacement coordinates and 3 Euler angles
Coordinates_t	$\dot{\mathbf{c}}_{\text{config}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2, \dot{v}_0, \dot{v}_1, \dot{v}_2]_{\text{config}}^T$	velocity coordinates vector of node
RotationMatrix	$[A_{00}, A_{01}, A_{02}, A_{10}, \dots, A_{21}, A_{22}]_{\text{config}}^T$	vector with 9 components of the rotation matrix ${}^{0b}\mathbf{A}_{\text{config}}$ in row-major format, in any configuration; the rotation matrix transforms local ( $b$ ) to global ( $0$ ) coordinates
Rotation	$[\varphi_0, \varphi_1, \varphi_2]_{\text{config}}^T$	vector with 3 components of the Euler/Tait-Bryan angles in xyz-sequence ( ${}^{0b}\mathbf{A}_{\text{config}} := \mathbf{A}_0(\varphi_0) \cdot \mathbf{A}_1(\varphi_1) \cdot \mathbf{A}_2(\varphi_2)$ ), recomputed from rotation matrix
AngularVelocity	${}^0\boldsymbol{\omega}_{\text{config}} = {}^0[\omega_0, \omega_1, \omega_2]_{\text{config}}^T$	global 3D angular velocity vector of node
AngularVelocityLocal	${}^b\boldsymbol{\omega}_{\text{config}} = {}^b[\omega_0, \omega_1, \omega_2]_{\text{config}}^T$	local (body-fixed) 3D angular velocity vector of node

**Detailed information:** For a detailed description on the rigid body dynamics formulation using this node, see Holzinger and Gerstmayr [16].

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

$$\boldsymbol{\nu} = \varphi \mathbf{n} = \boldsymbol{\nu}_{\text{config}} + \boldsymbol{\nu}_{\text{ref}}, \quad (7.12)$$

with the rotation angle  $\varphi$  and the rotation axis  $\mathbf{n}$ . All coordinates  $\mathbf{c}_{\text{config}}$  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  ${}^{0b}\mathbf{A}(\boldsymbol{\nu})_{\text{config}}$  transforms a local (body-fixed) 3D position  ${}^b\mathbf{b} = {}^b[b_0, b_1, b_2]^T$  to global 3D positions,

$${}^0\mathbf{b}_{\text{config}} = {}^{0b}\mathbf{A}(\boldsymbol{\nu})_{\text{config}} {}^b\mathbf{b} \quad (7.13)$$

Note that  $\mathbf{A}(\boldsymbol{\nu})$  is defined in function `RotationVector2RotationMatrix`, see [Section 5.13](#).

A Lie group integrator must be used with this node, which is why the is used, the rotation parameter velocities are identical to the local angular velocity  ${}^b\boldsymbol{\omega}$  and thus the matrix  ${}^b\mathbf{G}$  becomes the identity matrix.

For creating a `NodeRigidBodyRotVecLG`, there is a `rigidBodyUtilities` function `AddRigidBody`, see [Section 5.13](#), which simplifies the setup of a rigid body significantly!

---

For examples on NodeRigidBodyRotVecLG see Examples and TestModels:

- [explicitLieGroupIntegratorPythonTest.py](#) (TestModels/)
- [explicitLieGroupIntegratorTest.py](#) (TestModels/)
- [explicitLieGroupMBSTest.py](#) (TestModels/)

## 7.1.6 NodeRigidBody2D

A 2D rigid body node for rigid bodies or beams; the node has 2 displacement degrees of freedom and one rotation coordinate (rotation around z-axis: uphi). All coordinates are [ODE2](#), used for second order differential equations.

**Additional information for NodeRigidBody2D:**

- The Node has the following types = Position2D, Orientation2D, Position, Orientation, RigidBody
- **Short name for Python** = **Rigid2D**
- **Short name for Python** (visualization object) = **VRigid2D**

The item **NodeRigidBody2D** with type = 'RigidBody2D' has the following parameters:

Name	type	size	default value	description
name	String		"	node's unique name
referenceCoordinates	Vector3D	3	[0.,0.,0.]	reference coordinates (x-pos,y-pos and rotation) of node ==> e.g. ref. coordinates for finite elements; global position of node without displacement
initialCoordinates	Vector3D	3	[0.,0.,0.]	initial displacement coordinates and angle (relative to reference coordinates)
initialVelocities	Vector3D	3	[0.,0.,0.]	initial velocity coordinates
visualization	VNodeRigidBody2D			parameters for visualization of item

The item VNodeRigidBody2D has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size (diameter, dimensions of underlying cube, etc.) for item; size == -1.f means that default size is used
color	Float4	4	[-1.,-1.,-1.,-1.]	Default RGBA color for nodes; 4th value is alpha-transparency; R=-1.f means, that default color is used

---

### 7.1.6.1 DESCRIPTION of NodeRigidBody2D:

**Information on input parameters:**

input parameter	symbol	description see tables above
referenceCoordinates	$\mathbf{q}_{\text{ref}} = [q_0, q_1, \psi_0]^T_{\text{ref}}$	

initialCoordinates	$\mathbf{q}_{\text{ini}} = [q_0, q_1, \psi_0]^T_{\text{ini}}$	
initialVelocities	$\dot{\mathbf{q}}_{\text{ini}} = [\dot{q}_0, \dot{q}_1, \dot{\psi}_0]^T_{\text{ini}} = [v_0, v_1, \omega_2]^T_{\text{ini}}$	

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

output variable	symbol	description
Position	${}^0\mathbf{p}_{\text{config}} = {}^0[p_0, p_1, 0]^T_{\text{config}} = {}^0\mathbf{u}_{\text{config}} + {}^0\mathbf{p}_{\text{ref}}$	global 3D position vector of node; $\mathbf{u}_{\text{ref}} = 0$
Displacement	${}^0\mathbf{u}_{\text{config}} = [q_0, q_1, 0]^T_{\text{config}}$	global 3D displacement vector of node
Velocity	${}^0\mathbf{v}_{\text{config}} = [\dot{q}_0, \dot{q}_1, 0]^T_{\text{config}}$	global 3D velocity vector of node
Acceleration	${}^0\mathbf{a}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, 0]^T_{\text{config}}$	global 3D acceleration vector of node
AngularVelocity	${}^0\boldsymbol{\omega}_{\text{config}} = [0, 0, \dot{\psi}_0]^T_{\text{config}}$	global 3D angular velocity vector of node
Coordinates	$\mathbf{c}_{\text{config}} = [q_0, q_1, \psi_0]^T_{\text{config}}$	coordinate vector of node, having 2 displacement coordinates and 1 angle
Coordinates_t	$\dot{\mathbf{c}}_{\text{config}} = [\dot{q}_0, \dot{q}_1, \dot{\psi}_0]^T_{\text{config}}$	velocity coordinates vector of node
Coordinates_tt	$\ddot{\mathbf{c}}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, \ddot{\psi}_0]^T_{\text{config}}$	acceleration coordinates vector of node
RotationMatrix	$[A_{00}, A_{01}, A_{02}, A_{10}, \dots, A_{21}, A_{22}]^T_{\text{config}}$	vector with 9 components of the rotation matrix ${}^0{}^b\mathbf{A}_{\text{config}}$ in row-major format, in any configuration; the rotation matrix transforms local ( $b$ ) to global (0) coordinates
Rotation	$[0, 0, \theta_0]^T_{\text{config}} = [0, 0, \psi_0]^T_{\text{ref}} + [0, 0, \psi_0]^T_{\text{config}}$	vector with 3rd angle around out of plane axis
AngularVelocityLocal	${}^b\boldsymbol{\omega}_{\text{config}} = {}^b[0, 0, \dot{\psi}_0]^T_{\text{config}}$	local (body-fixed) 3D angular velocity vector of node
AngularAcceleration	${}^0\boldsymbol{\alpha}_{\text{config}} = {}^0[0, 0, \ddot{\psi}_0]^T_{\text{config}}$	global 3D angular acceleration vector of node

**Detailed information:** The node provides 2 displacement coordinates (displacement of center of mass (**COM**),  $(q_0, q_1)$ ) and 1 rotation parameter ( $\theta_0$ ). According equations need to be provided by an according object (e.g., `RigidBody2D`). Using the rotation parameter  $\theta_{0\text{config}} = \psi_{0\text{ref}} + \psi_{0\text{config}}$ , the rotation matrix is defined as

$${}^0{}^b\mathbf{A}_{\text{config}} = \begin{bmatrix} \cos(\theta_0) & -\sin(\theta_0) & 0 \\ \sin(\theta_0) & \cos(\theta_0) & 0 \\ 0 & 0 & 1 \end{bmatrix}_{\text{config}} \quad (7.14)$$

**Example** for `NodeRigidBody2D`: see `ObjectRigidBody2D`

For examples on `NodeRigidBody2D` see Examples and TestModels:

- [`beltDriveReevingSystem.py`](#) (Examples/)
- [`reevingSystem.py`](#) (Examples/)
- [`sliderCrank3DwithANCFbeltDrive2.py`](#) (Examples/)
- [`ANCF\_moving\_rigidbody.py`](#) (Examples/)
- [`ANCF\_slidingJoint2D.py`](#) (Examples/)
- [`ANCF\_slidingJoint2Drigid.py`](#) (Examples/)
- [`ANCF\_switchingSlidingJoint2D.py`](#) (Examples/)
- [`finiteSegmentMethod.py`](#) (Examples/)

- [geneticOptimizationSliderCrank.py](#) (Examples/)
- [lavalRotor2Dtest.py](#) (Examples/)
- [rigid\\_pendulum.py](#) (Examples/)
- [SliderCrank.py](#) (Examples/)
- ...
- [ANCFbeltDrive.py](#) (TestModels/)
- [ANCFgeneralContactCircle.py](#) (TestModels/)
- [fourBarMechanismRedundant.py](#) (TestModels/)
- ...

### 7.1.7 Node1D

A node with one [ODE2](#) coordinate for one dimensional (1D) problems; use e.g. for scalar dynamic equations (Mass1D) and mass-spring-damper mechanisms, representing either translational or rotational degrees of freedom: in most cases, Node1D is equivalent to NodeGenericODE2 using one coordinate, however, it offers a transformation to 3D translational or rotational motion and allows to couple this node to 2D or 3D bodies.

#### Additional information for Node1D:

- The Node has the following types = GenericODE2

The item **Node1D** with type = '1D' has the following parameters:

Name	type	size	default value	description
name	String		"	node's unique name
referenceCoordinates	Vector		[0.]	reference coordinate of node (in vector form)
initialCoordinates	Vector		[0.]	initial displacement coordinate (in vector form)
initialVelocities	Vector		[0.]	initial velocity coordinate (in vector form)
visualization	VNode1D			parameters for visualization of item

The item VNode1D has the following parameters:

Name	type	size	default value	description
show	Bool		False	set true, if item is shown in visualization and false if it is not shown; The node1D is represented as reference position and displacement along the global x-axis, which must not agree with the representation in the object using the Node1D

#### 7.1.7.1 DESCRIPTION of Node1D:

##### Information on input parameters:

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

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

output variable	symbol	description
Coordinates	$\mathbf{q}_{\text{config}} = [q_0]_{\text{config}}^T$	<a href="#">ODE2</a> coordinate of node (in vector form)
Coordinates_t	$\dot{\mathbf{q}}_{\text{config}} = [\dot{q}_0]_{\text{config}}^T$	<a href="#">ODE2</a> velocity coordinate of node (in vector form)

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

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

$$p_0 = q_{0\text{ref}} + q_{0\text{cur}} \quad (7.15)$$

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

$$p_{\text{config}} = \begin{bmatrix} p_{0\text{config}} \\ 0 \\ 0 \end{bmatrix} \quad (7.16)$$

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

For examples on Node1D see Examples and TestModels:

- [lugreFrictionTest.py](#) (Examples/)
- [multiprocessingTest.py](#) (Examples/)
- [nMassOscillatorInteractive.py](#) (Examples/)
- [driveTrainTest.py](#) (TestModels/)

### 7.1.8 NodePoint2DSlope1

A 2D point/slope vector node for planar Bernoulli-Euler ANCF (absolute nodal coordinate formulation) beam elements; the node has 4 displacement degrees of freedom (2 for displacement of point node and 2 for the slope vector 'slopex'); all coordinates lead to second order differential equations; the slope vector defines the directional derivative w.r.t the local axial (x) coordinate, denoted as ()'; in straight configuration aligned at the global x-axis, the slope vector reads  $\mathbf{r}' = [r'_x \ r'_y]^T = [1 \ 0]^T$ .

#### Additional information for NodePoint2DSlope1:

- The Node has the following types = Position2D, Orientation2D, Point2DSlope1, Position, Orientation
- Short name for Python = **Point2DS1**
- Short name for Python (visualization object) = **VPoint2DS1**

The item **NodePoint2DSlope1** with type = 'Point2DSlope1' has the following parameters:

Name	type	size	default value	description
name	String		"	node's unique name
referenceCoordinates	Vector4D	4	[0.,0.,1.,0.]	reference coordinates (x-pos,y-pos; x-slopex, y-slopex) of node; global position of node without displacement
initialCoordinates	Vector4D	4	[0.,0.,0.,0.]	initial displacement coordinates: ux, uy and x/y 'displacements' of slopex
initialVelocities	Vector4D	4	[0.,0.,0.,0.]	initial velocity coordinates
visualization	VNodePoint2DSlope1			parameters for visualization of item

The item **VNodePoint2DSlope1** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size (diameter, dimensions of underlying cube, etc.) for item; size == -1.f means that default size is used
color	Float4	4	[-1.,-1.,-1.,-1.]	Default RGBA color for nodes; 4th value is alpha-transparency; R=-1.f means, that default color is used

---

#### 7.1.8.1 DESCRIPTION of NodePoint2DSlope1:

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

<b>output variable</b>	<b>symbol</b>	<b>description</b>
Position		global 3D position vector of node (=displacement+reference position)
Displacement		global 3D displacement vector of node
Velocity		global 3D velocity vector of node
Coordinates		coordinates vector of node (2 displacement coordinates + 2 slope vector coordinates)
Coordinates_t		velocity coordinates vector of node (derivative of the 2 displacement coordinates + 2 slope vector coordinates)
Coordinates_tt		acceleration coordinates vector of node (derivative of the 2 displacement coordinates + 2 slope vector coordinates)

---

For examples on NodePoint2DSlope1 see Examples and TestModels:

- [sliderCrank3DwithANCFbeltDrive2.py](#) (Examples/)
- [ALEANCF\\_pipe.py](#) (Examples/)
- [ANCF\\_cantilever\\_test\\_dyn.py](#) (Examples/)
- [ANCF\\_contact\\_circle.py](#) (Examples/)
- [ANCF\\_contact\\_circle2.py](#) (Examples/)
- [ANCF\\_moving\\_rigidbody.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- [ANCF\\_slidingJoint2Drigid.py](#) (Examples/)
- [ANCF\\_switchingSlidingJoint2D.py](#) (Examples/)
- [ANCF\\_tests2.py](#) (Examples/)
- [ANCF\\_test\\_halfcircle.py](#) (Examples/)
- [sliderCrank3DwithANCFbeltDrive.py](#) (Examples/)
- ...
- [ANCFcontactCircleTest.py](#) (TestModels/)
- [ANCFcontactFrictionTest.py](#) (TestModels/)
- [computeODE2EigenvaluesTest.py](#) (TestModels/)
- ...

### 7.1.9 NodeGenericODE2

A node containing a number of [ODE2](#) variables; use e.g. for scalar dynamic equations (Mass1D) or for the ALECable element. Note that referenceCoordinates and all initialCoordinates(\_t) must be initialized, because no default values exist.

#### Additional information for NodeGenericODE2:

- The Node has the following types = GenericODE2

The item **NodeGenericODE2** with type = 'GenericODE2' has the following parameters:

Name	type	size	default value	description
name	String		''	node's unique name
referenceCoordinates	Vector		[]	generic reference coordinates of node; must be consistent with numberOfODE2Coordinates
initialCoordinates	Vector		[]	initial displacement coordinates; must be consistent with numberOfODE2Coordinates
initialCoordinates_t	Vector		[]	initial velocity coordinates; must be consistent with numberOfODE2Coordinates
numberOfODE2Coordinates	PInt		0	number of generic <a href="#">ODE2</a> 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

#### 7.1.9.1 DESCRIPTION of NodeGenericODE2:

##### Information on input parameters:

input parameter	symbol	description see tables above
referenceCoordinates	$\mathbf{q}_{\text{ref}} = [q_0, \dots, q_{nc}]^T_{\text{ref}}$	
initialCoordinates	$\mathbf{q}_{\text{ini}} = [q_0, \dots, q_{nc}]^T_{\text{ini}}$	
initialCoordinates_t	$\dot{\mathbf{q}}_{\text{ini}} = [\dot{q}_0, \dots, \dot{q}_{nc}]^T_{\text{ini}}$	
numberOfODE2Coordinates	$n_c$	

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

output variable	symbol	description
Coordinates	$\mathbf{q}_{\text{config}} = [q_0, \dots, q_{nc}]^T_{\text{config}}$	coordinates vector of node
Coordinates_t	$\dot{\mathbf{q}}_{\text{config}} = [\dot{q}_0, \dots, \dot{q}_{nc}]^T_{\text{config}}$	velocity coordinates vector of node
Coordinates_tt	$\ddot{\mathbf{q}}_{\text{config}} = [\ddot{q}_0, \dots, \ddot{q}_{nc}]^T_{\text{config}}$	acceleration coordinates vector of node

---

For examples on NodeGenericODE2 see Examples and TestModels:

- [ALEANCF\\_pipe.py](#) (Examples/)
- [ANCFALEtest.py](#) (Examples/)
- [ANCF\\_moving\\_rigidbody.py](#) (Examples/)
- [nMassOscillatorInteractive.py](#) (Examples/)
- [simulateInteractively.py](#) (Examples/)
- [ACNFslidingAndALEjointTest.py](#) (TestModels/)
- [ANCFmovingRigidBodyTest.py](#) (TestModels/)
- [solverExplicitODE10ODE2test.py](#) (TestModels/)

### 7.1.10 NodeGenericODE1

A node containing a number of [ODE1](#) variables; use e.g. linear state space systems. Note that referenceCoordinates and initialCoordinates must be initialized, because no default values exist.

The item **NodeGenericODE1** with type = 'GenericODE1' has the following parameters:

Name	type	size	default value	description
name	String		"	node's unique name
referenceCoordinates	Vector		[]	generic reference coordinates of node; must be consistent with <a href="#">numberOfODE1Coordinates</a>
initialCoordinates	Vector		[]	initial displacement coordinates; must be consistent with <a href="#">numberOfODE1Coordinates</a>
numberOfODE1Coordinates	PInt		0	number of generic <a href="#">ODE1</a> coordinates
visualization	VNodeGenericODE1			parameters for visualization of item

The item VNodeGenericODE1 has the following parameters:

Name	type	size	default value	description
show	Bool		False	set true, if item is shown in visualization and false if it is not shown

#### 7.1.10.1 DESCRIPTION of NodeGenericODE1:

**Information on input parameters:**

input parameter	symbol	description see tables above
referenceCoordinates	$y_{ref} = [y_0, \dots, y_{nc}]^T_{ref}$	
initialCoordinates	$y_{ini} = [y_0, \dots, y_{nc}]^T_{ini}$	
numberOfODE1Coordinates	$n_c$	

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

output variable	symbol	description
Coordinates	$y_{config} = [y_0, \dots, y_{nc}]^T_{config}$	<a href="#">ODE1</a> coordinates vector of node
Coordinates_t	$\dot{y}_{config} = [\dot{y}_0, \dots, \dot{y}_{nc}]^T_{config}$	<a href="#">ODE1</a> velocity coordinates vector of node

For examples on NodeGenericODE1 see Examples and TestModels:

- [lugreFrictionODE1.py](#) (Examples/)
- [lugreFrictionTest.py](#) (Examples/)
- [solverExplicitODE1ODE2test.py](#) (TestModels/)

### 7.1.11 NodeGenericData

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

**Additional information for NodeGenericData:**

- The Node has the following types = GenericData

The item **NodeGenericData** with type = 'GenericData' has the following parameters:

Name	type	size	default value	description
name	String		"	node's unique name
initialCoordinates	Vector		[]	initial data coordinates
numberOfDataCoordinates	UInt		0	number of generic data coordinates (history variables)
visualization	VNodeGenericData			parameters for visualization of item

The item **VNodeGenericData** has the following parameters:

Name	type	size	default value	description
show	Bool		False	set true, if item is shown in visualization and false if it is not shown

#### 7.1.11.1 DESCRIPTION of NodeGenericData:

**Information on input parameters:**

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

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

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

For examples on NodeGenericData see Examples and TestModels:

- [ANCF\\_contact\\_circle.py](#) (Examples/)
- [ANCF\\_contact\\_circle2.py](#) (Examples/)
- [ANCF\\_moving\\_rigidbody.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- [ANCF\\_slidingJoint2Drigid.py](#) (Examples/)
- [ANCF\\_switchingSlidingJoint2D.py](#) (Examples/)

- [beltDriveReevingSystem.py](#) (Examples/)
- [bicycleIftommBenchmark.py](#) (Examples/)
- [leggedRobot.py](#) (Examples/)
- [lugreFrictionTest.py](#) (Examples/)
- [serialRobotTSD.py](#) (Examples/)
- [sliderCrank3DwithANCFbeltDrive.py](#) (Examples/)
- ...
- [ANCFslidingAndALEjointTest.py](#) (TestModels/)
- [ANCFcontactCircleTest.py](#) (TestModels/)
- [ANCFcontactFrictionTest.py](#) (TestModels/)
- ...

## 7.1.12 NodePointGround

A 3D point node fixed to ground. The node can be used as NodePoint, but it does not generate coordinates. Applied or reaction forces do not have any effect. This node can be used for 'blind' or 'dummy' [ODE2](#) and [ODE1](#) coordinates to which CoordinateSpringDamper or CoordinateConstraint objects are attached to.

**Additional information for NodePointGround:**

- The Node has the following types = Ground, Position2D, Position, Orientation, GenericODE2
- **Short name for Python = PointGround**
- **Short name for Python (visualization object) = VPointGround**

The item **NodePointGround** with type = 'PointGround' has the following parameters:

Name	type	size	default value	description
name	String		"	node's unique name
referenceCoordinates	Vector3D	3	[0.,0.,0.]	reference coordinates of node ==> e.g. ref. coordinates for finite elements; global position of node without displacement
visualization	VNodePointGround			parameters for visualization of item

The item **VNodePointGround** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size (diameter, dimensions of underlying cube, etc.) for item; size == -1.f means that default size is used
color	Float4	4	[-1.,-1.,-1.,-1.]	Default RGBA color for nodes; 4th value is alpha-transparency; R=-1.f means, that default color is used

### 7.1.12.1 DESCRIPTION of NodePointGround:

**Information on input parameters:**

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

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

output variable	symbol	description

Position	$\mathbf{p}_{\text{config}} = [p_0, p_1, p_2]^T_{\text{config}} = \mathbf{p}_{\text{ref}}$	global 3D position vector of node (=reference position)
Displacement	$\mathbf{u}_{\text{config}} = [0, 0, 0]^T_{\text{config}}$	zero 3D vector
Velocity	$\mathbf{v}_{\text{config}} = [0, 0, 0]^T_{\text{config}}$	zero 3D vector
Coordinates	$\mathbf{c}_{\text{config}} = []$	vector of length zero
Coordinates_t	$\dot{\mathbf{c}}_{\text{config}} = []$	vector of length zero

---

For examples on NodePointGround see Examples and TestModels:

- [ALEANCF\\_pipe.py](#) (Examples/)
- [ANCFALTest.py](#) (Examples/)
- [ANCF\\_cantilever\\_test.py](#) (Examples/)
- [ANCF\\_cantilever\\_test\\_dyn.py](#) (Examples/)
- [ANCF\\_contact\\_circle.py](#) (Examples/)
- [ANCF\\_contact\\_circle2.py](#) (Examples/)
- [ANCF\\_moving\\_rigidbody.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- [ANCF\\_slidingJoint2Drigid.py](#) (Examples/)
- [ANCF\\_switchingSlidingJoint2D.py](#) (Examples/)
- [ANCF\\_tests2.py](#) (Examples/)
- [ANCF\\_test\\_halfcircle.py](#) (Examples/)
- ...
- [ANCFslidingAndALEjointTest.py](#) (TestModels/)
- [ANCFbeltDrive.py](#) (TestModels/)
- [ANCFcontactCircleTest.py](#) (TestModels/)
- ...

## 7.2 Objects (Body)

A Body is a special Object, which has physical properties such as mass. A localPosition can be measured w.r.t. the reference point of the body

### 7.2.1 ObjectGround

A ground object behaving like a rigid body, but having no degrees of freedom; used to attach body-connectors without an action. For examples see spring dampers and joints.

**Additional information for ObjectGround:**

- The Object has the following types = Ground, Body

The item **ObjectGround** with type = 'Ground' has the following parameters:

Name	type	size	default value	description
name	String		"	objects's unique name
referencePosition	Vector3D	3	[0.,0.,0.]	reference point = reference position for ground object; local position is added on top of reference position for a ground object
visualization	VObjectGround			parameters for visualization of item

The item VObjectGround has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
graphicsDataUserFunction	PyFunctionGraphicsData		0	A Python function which returns a body-GraphicsData object, which is a list of graphics data in a dictionary computed by the user function
color	Float4		[-1.,-1.,-1.,-1.]	RGB node color; if R== -1, use default color
graphicsData	BodyGraphicsData			Structure contains data for body visualization; data is defined in special list / dictionary structure

#### 7.2.1.1 DESCRIPTION of ObjectGround:

**Information on input parameters:**

input parameter	symbol	description see tables above
referencePosition	${}^0\mathbf{r}$	

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

output variable	symbol	description
Position	${}^0\mathbf{p} = {}^0\mathbf{r} + {}^{0b}\mathbf{I}_{3\times 3} {}^b\mathbf{b}$	global position vector of translated local position
Displacement	$\mathbf{0}$	global displacement vector of local position
Velocity	$\mathbf{0}$	global velocity vector of local position
AngularVelocity	$\mathbf{0}$	angular velocity of body
RotationMatrix	$\mathbf{I}$	rotation matrix in vector form (stored in row-major order)

### 7.2.1.2 Equations

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

In combination with markers, the `localPosition`  ${}^b\mathbf{b}$  is transformed by the `ObjectGround` to a global point  ${}^0\mathbf{p}$  using the reference point  ${}^0\mathbf{r}$ ,

$${}^0\mathbf{p} = {}^0\mathbf{r} + {}^{0b}\mathbf{I}_{3\times 3} {}^b\mathbf{b} \quad (7.17)$$

in which  ${}^{0b}\mathbf{I}_{3\times 3}$  is the identity transformation, leading to  ${}^0\mathbf{b} = {}^{0b}\mathbf{I}_{3\times 3} {}^b\mathbf{b} = {}^b\mathbf{b}$ .

---

### Userfunction: `graphicsDataUserFunction(mbs, itemNumber)`

A user function, which is called by the visualization thread in order to draw user-defined objects. The function can be used to generate any `BodyGraphicsData`, see Section 9.3. Use `graphicsDataUtilities` functions, see Section 5.5, to create more complicated objects. Note that `graphicsDataUserFunction` needs to copy lots of data and is therefore inefficient and only designed to enable simpler tests, but not large scale problems.

arguments / return	type or size	description
<code>mbs</code>	<code>MainSystem</code>	provides reference to <code>mbs</code> , which can be used in user function to access all data of the object
<code>itemNumber</code>	<code>Index</code>	integer number of the object in <code>mbs</code> , allowing easy access
<code>return value</code>	<code>BodyGraphicsData</code>	list of <code>GraphicsData</code> dictionaries, see Section 9.3

---

### User function example:

```
import exudyn as exu
from math import sin, cos, pi
from exudyn.itemInterface import *
from exudyn.graphicsDataUtilities import *
SC = exu.SystemContainer()
mbs = SC.AddSystem()
#create simple system:
mbs.AddNode(NodePoint())
body = mbs.AddObject(MassPoint(physicsMass=1, nodeNumber=0))
```

```

#user function for moving graphics:
def UFgraphics(mbs, objectNum):
    t = mbs.systemData.GetTime(exu.ConfigurationType.Visualization) #get time if needed
    #draw moving sphere on ground
    graphics1=GraphicsDataSphere(point=[sin(t*2*pi), cos(t*2*pi), 0],
                                   radius=0.1, color=color4red, nTiles=32)
    return [graphics1]

#add object with graphics user function
ground = mbs.AddObject(ObjectGround(visualization=VObjectGround(graphicsDataUserFunction
=UFgraphics)))
mbs.Assemble()
sims=exu.SimulationSettings()
sims.timeIntegration.numberOfSteps = 10000000 #many steps to see graphics
exu.StartRenderer() #perform zoom all (press 'a' several times) after startup to see the
sphere
exu.SolveDynamic(mbs, sims)
exu.StopRenderer()

```

For examples on ObjectGround see Examples and TestModels:

- [addPrismaticJoint.py](#) (Examples/)
- [addRevoluteJoint.py](#) (Examples/)
- [ALEANCF\\_pipe.py](#) (Examples/)
- [ANCF\\_cantilever\\_test.py](#) (Examples/)
- [ANCF\\_cantilever\\_test\\_dyn.py](#) (Examples/)
- [ANCF\\_contact\\_circle.py](#) (Examples/)
- [ANCF\\_contact\\_circle2.py](#) (Examples/)
- [ANCF\\_moving\\_rigidbody.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- [ANCF\\_slidingJoint2Drigid.py](#) (Examples/)
- [ANCF\\_switchingSlidingJoint2D.py](#) (Examples/)
- [ANCF\\_tests2.py](#) (Examples/)
- ...
- [ANCFslidingAndALEjointTest.py](#) (TestModels/)
- [ANCFbeltDrive.py](#) (TestModels/)
- [ANCFcontactCircleTest.py](#) (TestModels/)
- ...

## 7.2.2 ObjectMassPoint

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

**Additional information for ObjectMassPoint:**

- The Object has the following types = Body, SingleNoded
- Requested node type = Position
- Short name for Python = **MassPoint**
- Short name for Python (visualization object) = **VMassPoint**

The item **ObjectMassPoint** with type = 'MassPoint' has the following parameters:

Name	type	size	default value	description
name	String		"	objects's unique name
physicsMass	UReal		0.	mass [SI:kg] of mass point
nodeNumber	NodeIndex		MAXINT	node number (type NodeIndex) for mass point
visualization	VObjectMassPoint			parameters for visualization of item

The item VObjectMassPoint has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
graphicsData	BodyGraphicsData			Structure contains data for body visualization; data is defined in special list / dictionary structure

### 7.2.2.1 DESCRIPTION of ObjectMassPoint:

**Information on input parameters:**

input parameter	symbol	description see tables above
physicsMass	$m$	
nodeNumber	$n_0$	

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

output variable	symbol	description
Position	${}^0\mathbf{p}_{\text{config}}({}^b\mathbf{b}) = {}^0\mathbf{r}_{\text{config}} + {}^0\mathbf{r}_{\text{ref}} + {}^{0b}\mathbf{I}_{3 \times 3} {}^b\mathbf{b}$	global position vector of translated local position; local (body) coordinate system = global coordinate system
Displacement	${}^0\mathbf{u}_{\text{config}} = [q_0, q_1, q_2]^T_{\text{config}}$	global displacement vector of mass point
Velocity	${}^0\mathbf{v}_{\text{config}} = {}^0\dot{\mathbf{u}}_{\text{config}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2]^T_{\text{config}}$	global velocity vector of mass point
Acceleration	${}^0\mathbf{a}_{\text{config}} = {}^0\ddot{\mathbf{u}}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, \ddot{q}_2]^T_{\text{config}}$	global acceleration vector of mass point

### 7.2.2.2 Definition of quantities

intermediate variables	symbol	description
node position	${}^0\mathbf{r}_{\text{config}} + {}^0\mathbf{r}_{\text{ref}} = {}^0\mathbf{p}(n_0)_{\text{config}}$	position of mass point which is provided by node $n_0$ in any configuration
node displacement	${}^0\mathbf{u}_{\text{config}} = {}^0\mathbf{r}_{\text{config}} = [q_0, q_1, q_2]^T_{\text{config}} = {}^0\mathbf{u}(n_0)_{\text{config}}$	displacement of mass point which is provided by node $n_0$ in any configuration
node velocity	${}^0\mathbf{v}_{\text{config}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2]^T_{\text{config}} = {}^0\mathbf{v}(n_0)_{\text{config}}$	velocity of mass point which is provided by node $n_0$ in any configuration
transformation matrix	${}^{0b}\mathbf{A} = \mathbf{I}_{3 \times 3}$	transformation of local body ( $b$ ) coordinates to global (0) coordinates; this is the constant unit matrix, because local = global coordinates for the mass point
residual forces	${}^0\mathbf{f} = [f_0, f_1, f_2]^T$	residual of all forces on mass point
applied forces	${}^0\mathbf{f}_a = [f_0, f_1, f_2]^T$	applied forces (loads, connectors, joint reaction forces, ...)

### 7.2.2.3 Equations of motion

$$\begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{bmatrix} \begin{bmatrix} \ddot{q}_0 \\ \ddot{q}_1 \\ \ddot{q}_2 \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ f_2 \end{bmatrix}. \quad (7.18)$$

For example, a LoadCoordinate on coordinate 1 of the node would add a term in  $f_1$  on the RHS.

Position-based markers can measure position  $\mathbf{p}_{\text{config}}$ . The **position jacobian**

$$\mathbf{J}_{\text{pos}} = \partial \mathbf{p}_{\text{cur}} / \partial \mathbf{c}_{\text{cur}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (7.19)$$

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

$$\mathbf{Q} = \mathbf{J}_{\text{pos}} {}^0\mathbf{f}_a. \quad (7.20)$$

### 7.2.2.4 MINI EXAMPLE for ObjectMassPoint

```

node = mbs.AddNode(NodePoint(referenceCoordinates = [1,1,0],
                             initialCoordinates=[0.5,0,0],
                             initialVelocities=[0.5,0,0]))
mbs.AddObject(MassPoint(nodeNumber = node, physicsMass=1))

#assemble and solve system for default parameters
mbs.Assemble()
exu.SolveDynamic(mbs)

#check result

```

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

---

For examples on ObjectMassPoint see Examples and TestModels:

- [interactiveTutorial.py](#) (Examples/)
- [cartesianSpringDamper.py](#) (Examples/)
- [ComputeSensitivitiesExample.py](#) (Examples/)
- [coordinateSpringDamper.py](#) (Examples/)
- [massSpringFrictionInteractive.py](#) (Examples/)
- [minimizeExample.py](#) (Examples/)
- [nMassOscillatorInteractive.py](#) (Examples/)
- [parameterVariationExample.py](#) (Examples/)
- [particleClusters.py](#) (Examples/)
- [particlesTest.py](#) (Examples/)
- [particlesTest3D.py](#) (Examples/)
- [particlesTest3D2.py](#) (Examples/)
- ...
- [connectorGravityTest.py](#) (TestModels/)
- [contactCoordinateTest.py](#) (TestModels/)
- [fourBarMechanismTest.py](#) (TestModels/)
- ...

### 7.2.3 ObjectMassPoint2D

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

**Additional information for ObjectMassPoint2D:**

- The Object has the following types = Body, SingleNoded
- Requested node type = Position2D + Position
- **Short name for Python** = **MassPoint2D**
- **Short name for Python (visualization object)** = **VMassPoint2D**

The item **ObjectMassPoint2D** with type = 'MassPoint2D' has the following parameters:

Name	type	size	default value	description
name	String		"	objects's unique name
physicsMass	UReal		0.	mass [SI:kg] of mass point
nodeNumber	NodeIndex		MAXINT	node number (type NodeIndex) for mass point
visualization	VObjectMassPoint2D			parameters for visualization of item

The item VObjectMassPoint2D has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
graphicsData	BodyGraphicsData			Structure contains data for body visualization; data is defined in special list / dictionary structure

#### 7.2.3.1 DESCRIPTION of ObjectMassPoint2D:

**Information on input parameters:**

input parameter	symbol	description see tables above
physicsMass	$m$	
nodeNumber	$n_0$	

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

output variable	symbol	description
Position	${}^0\mathbf{p}_{\text{config}}({}^b\mathbf{b}) = {}^0\mathbf{r}_{\text{config}} + {}^0\mathbf{r}_{\text{ref}} + {}^{0b}\mathbf{I}_{2 \times 2} {}^b\mathbf{b}$	global position vector of translated local position; local (body) coordinate system = global coordinate system
Displacement	${}^0\mathbf{u}_{\text{config}} = [q_0, q_1, 0]^T_{\text{config}}$	global displacement vector of mass point
Velocity	${}^0\mathbf{v}_{\text{config}} = {}^0\dot{\mathbf{u}}_{\text{config}} = [\dot{q}_0, \dot{q}_1, 0]^T_{\text{config}}$	global velocity vector of mass point
Acceleration	${}^0\mathbf{a}_{\text{config}} = {}^0\ddot{\mathbf{u}}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, 0]^T_{\text{config}}$	global acceleration vector of mass point

### 7.2.3.2 Definition of quantities

intermediate variables	symbol	description
node position	${}^0\mathbf{r}_{\text{config}} + {}^0\mathbf{r}_{\text{ref}} = {}^0\mathbf{p}(n_0)_{\text{config}}$	position of mass point which is provided by node $n_0$ in any configuration (except reference)
node displacement	${}^0\mathbf{u}_{\text{config}} = {}^0\mathbf{r}_{\text{config}} = [q_0, q_1, 0]^T_{\text{config}} = {}^0\mathbf{u}(n_0)_{\text{config}}$	displacement of mass point which is provided by node $n_0$ in any configuration
node velocity	${}^0\mathbf{v}_{\text{config}} = [\dot{q}_0, \dot{q}_1, 0]^T_{\text{config}} = {}^0\mathbf{v}(n_0)_{\text{config}}$	velocity of mass point which is provided by node $n_0$ in any configuration
transformation matrix	${}^{0b}\mathbf{A} = \mathbf{I}_{3 \times 3}$	transformation of local body ( $b$ ) coordinates to global (0) coordinates; this is the constant unit matrix, because local = global coordinates for the mass point
residual forces	${}^0\mathbf{f} = [f_0, f_1]^T$	residual of all forces on mass point
applied forces	${}^0\mathbf{f}_a = [f_0, f_1, f_2]^T$	applied forces (loads, connectors, joint reaction forces, ...)

### 7.2.3.3 Equations of motion

$$\begin{bmatrix} m & 0 \\ 0 & m \end{bmatrix} \begin{bmatrix} \ddot{q}_0 \\ \ddot{q}_1 \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \end{bmatrix}. \quad (7.21)$$

For example, a LoadCoordinate on coordinate 1 of the node would add a term in  $f_1$  on the RHS.

Position-based markers can measure position  $\mathbf{p}_{\text{config}}$ . The **position jacobian**

$$\mathbf{J}_{\text{pos}} = \partial \mathbf{p}_{\text{cur}} / \partial \mathbf{c}_{\text{cur}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad (7.22)$$

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

$$\mathbf{Q} = \mathbf{J}_{\text{pos}} {}^0\mathbf{f}_a. \quad (7.23)$$

### 7.2.3.4 MINI EXAMPLE for ObjectMassPoint2D

```

node = mbs.AddNode(NodePoint2D(referenceCoordinates = [1,1],
                                initialCoordinates=[0.5,0],
                                initialVelocities=[0.5,0]))
mbs.AddObject(MassPoint2D(nodeNumber = node, physicsMass=1))

#assemble and solve system for default parameters
mbs.Assemble()
exu.SolveDynamic(mbs)

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

```

---

For examples on ObjectMassPoint2D see Examples and TestModels:

- [myFirstExample.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- [ANCF\\_slidingJoint2Drigid.py](#) (Examples/)
- [geneticOptimizationSliderCrank.py](#) (Examples/)
- [pendulum2Dconstraint.py](#) (Examples/)
- [SliderCrank.py](#) (Examples/)
- [sliderCrank3DwithANCFbeltDrive2.py](#) (Examples/)
- [slidercrankWithMassSpring.py](#) (Examples/)
- [SpringDamperMassUserFunction.py](#) (Examples/)
- [switchingConstraintsPendulum.py](#) (Examples/)
- [modelUnitTests.py](#) (TestModels/)
- [coordinateVectorConstraint.py](#) (TestModels/)
- [sliderCrankFloatingTest.py](#) (TestModels/)
- [sparseMatrixSpringDamperTest.py](#) (TestModels/)

## 7.2.4 ObjectMass1D

A 1D (translational) mass which is attached to Node1D. Note, that the mass does not need to have the interpretation as a translational mass.

**Additional information for ObjectMass1D:**

- The Object has the following types = Body, SingleNoded
- Requested node type = GenericODE2
- **Short name for Python** = **Mass1D**
- **Short name for Python (visualization object)** = **VMass1D**

The item **ObjectMass1D** with type = 'Mass1D' has the following parameters:

Name	type	size	default value	description
name	String		"	objects's unique name
physicsMass	UReal		0.	mass [SI:kg] of mass
nodeNumber	NodeIndex		MAXINT	node number (type NodeIndex) for Node1D
referencePosition	Vector3D	3	[0.,0.,0.]	a reference position, used to transform the 1D coordinate to a position
referenceRotation	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	the constant body rotation matrix, which transforms body-fixed (b) to global (0) coordinates
visualization	VObjectMass1D			parameters for visualization of item

The item VObjectMass1D has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
graphicsData	BodyGraphicsData			Structure contains data for body visualization; data is defined in special list / dictionary structure

---

### 7.2.4.1 DESCRIPTION of ObjectMass1D:

**Information on input parameters:**

input parameter	symbol	description see tables above
physicsMass	$m$	
nodeNumber	$n_0$	
referencePosition	${}^0\mathbf{r}_0$	
referenceRotation	${}^0_b\mathbf{A}_0 \in \mathbb{R}^{3 \times 3}$	

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

output variable	symbol	description
Position	${}^0\mathbf{p}_{\text{config}}$	global position vector; for interpretation see intermediate variables
Displacement	${}^0\mathbf{u}_{\text{config}}$	global displacement vector; for interpretation see intermediate variables
Velocity	${}^0\mathbf{v}_{\text{config}}$	global velocity vector; for interpretation see intermediate variables
RotationMatrix	${}^{0b}\mathbf{A}$	vector with 9 components of the rotation matrix (row-major format)
Rotation		vector with 3 components of the Euler angles in xyz-sequence ( $R=R_x \cdot R_y \cdot R_z$ ), recomputed from rotation matrix ${}^{0b}\mathbf{A}$
AngularVelocity	${}^0\boldsymbol{\omega}_{\text{config}}$	angular velocity of body
AngularVelocityLocal	${}^b\boldsymbol{\omega}_{\text{config}}$	local (body-fixed) 3D velocity vector of node

#### 7.2.4.2 Definition of quantities

intermediate variables	symbol	description
position coordinate	$p_{0\text{config}} = c_{0\text{config}} + c_{0\text{ref}}$	position coordinate of node (nodal coordinate $c_0$ ) in any configuration
displacement coordinate	$u_{0\text{config}} = c_{0\text{config}}$	displacement coordinate of mass node in any configuration
velocity coordinate	$u_{0\text{config}}$	velocity coordinate of mass node in any configuration
Position	${}^0\mathbf{p}_{\text{config}} = {}^0\mathbf{r}_0 + {}^{0b}\mathbf{A}_0 \begin{bmatrix} p_0 \\ 0 \\ 0 \end{bmatrix}_{\text{config}}$	(translational) position of mass object in any configuration
Displacement	${}^0\mathbf{u}_{\text{config}} = {}^{0b}\mathbf{A}_0 \begin{bmatrix} q_0 \\ 0 \\ 0 \end{bmatrix}_{\text{config}}$	(translational) displacement of mass object in any configuration
Velocity	${}^0\mathbf{v}_{\text{config}} = {}^{0b}\mathbf{A}_0 \begin{bmatrix} \dot{q}_0 \\ 0 \\ 0 \end{bmatrix}_{\text{config}}$	(translational) velocity of mass object in any configuration
residual force	$f$	residual of all forces on mass object
applied force	${}^0\mathbf{f}_a = [f_0, f_1, f_2]^T$	3D applied force (loads, connectors, joint reaction forces, ...)
applied torque	${}^0\boldsymbol{\tau}_a = [\tau_0, \tau_1, \tau_2]^T$	3D applied torque (loads, connectors, joint reaction forces, ...)

A rigid body marker (e.g., `MarkerBodyRigid`) may be attached to this object and forces/torques can be applied. However, torques will have no effect and forces will only have effect in 'direction' of the coordinate.

#### 7.2.4.3 Equations of motion

$$m \cdot \ddot{q}_0 = f. \quad (7.24)$$

Note that  $f$  is computed from all connectors and loads upon the object. E.g., a 3D force vector  ${}^0\mathbf{f}_a$  is transformed to  $f$  as

$$f = {}^b[1, 0, 0] {}^{b0}\mathbf{A}_0 {}^0\mathbf{f}_a \quad (7.25)$$

Thus, the **position jacobian** reads

$$\mathbf{J}_{pos} = \partial \mathbf{p}_{cur} / \partial q_{0,cur} = {}^b[1, 0, 0] {}^{b0}\mathbf{A}_0 \quad (7.26)$$


---

#### 7.2.4.4 MINI EXAMPLE for ObjectMass1D

```
node = mbs.AddNode(Node1D(referenceCoordinates = [1],
                           initialCoordinates=[0.5],
                           initialVelocities=[0.5]))
mass = mbs.AddObject(Mass1D(nodeNumber = node, physicsMass=1))

#assemble and solve system for default parameters
mbs.Assemble()
exu.SolveDynamic(mbs)

#check result, get current mass position at local position [0,0,0]
exodynTestGlobals.testResult = mbs.GetObjectOutputBody(mass, exu.OutputVariableType.
Position, [0,0,0])[0]
#final x-coordinate of position shall be 2
```

---

For examples on ObjectMass1D see Examples and TestModels:

- [lugreFrictionTest.py](#) (Examples/)
- [multiprocessingTest.py](#) (Examples/)
- [nMassOscillatorInteractive.py](#) (Examples/)
- [driveTrainTest.py](#) (TestModels/)

## 7.2.5 ObjectRotationalMass1D

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

**Additional information for ObjectRotationalMass1D:**

- The Object has the following types = Body, SingleNoded
- Requested node type = GenericODE2
- **Short name for Python = Rotor1D**
- **Short name for Python (visualization object) = VRotor1D**

The item **ObjectRotationalMass1D** with type = 'RotationalMass1D' has the following parameters:

Name	type	size	default value	description
name	String		"	objects's unique name
physicsInertia	UReal		0.	inertia components [SI:kgm <sup>2</sup> ] of rotor / rotational mass
nodeNumber	NodeIndex		MAXINT	node number (type NodeIndex) of Node1D, providing rotation coordinate $\psi_0 = c_0$
referencePosition	Vector3D	3	[0.,0.,0.]	a constant reference position = reference point, used to assign joint constraints accordingly and for drawing
referenceRotation	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	an intermediate rotation matrix, which transforms the 1D coordinate into 3D, see description
visualization	VObjectRotationalMass1D			parameters for visualization of item

The item VObjectRotationalMass1D has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
graphicsData	BodyGraphicsData			Structure contains data for body visualization; data is defined in special list / dictionary structure

---

### 7.2.5.1 DESCRIPTION of ObjectRotationalMass1D:

**Information on input parameters:**

input parameter	symbol	description see tables above
physicsInertia	$J$	
nodeNumber	$n_0$	
referencePosition	${}^0\mathbf{r}_0$	
referenceRotation	${}^0\mathbf{A}_0 \in \mathbb{R}^{3 \times 3}$	

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

output variable	symbol	description
Position	${}^0\mathbf{p}_{\text{config}} = p_{\text{RefG}}$	global position vector; for interpretation see intermediate variables
Displacement	${}^0\mathbf{u}_{\text{config}}$	global displacement vector; for interpretation see intermediate variables
Velocity	${}^0\mathbf{v}_{\text{config}}$	global velocity vector; for interpretation see intermediate variables
RotationMatrix	${}^{0b}\mathbf{A}$	vector with 9 components of the rotation matrix (row-major format)
Rotation	$\theta$	scalar rotation angle obtained from underlying node
AngularVelocity	${}^0\boldsymbol{\omega}_{\text{config}}$	angular velocity of body
AngularVelocityLocal	${}^b\boldsymbol{\omega}_{\text{config}}$	local (body-fixed) 3D velocity vector of node

### 7.2.5.2 Definition of quantities

intermediate variables	symbol	description
position coordinate	$\theta_{0\text{config}} = c_{0\text{config}} + c_{0\text{ref}}$	total rotation coordinate of node (e.g., Node1D) in any configuration (nodal coordinate $c_0$ )
displacement coordinate	$\psi_{0\text{config}} = c_{0\text{config}}$	change of rotation coordinate of mass node (e.g., Node1D) in any configuration (nodal coordinate $c_0$ )
velocity coordinate	$\dot{\psi}_{0\text{config}}$	rotation velocity coordinate of mass node (e.g., Node1D) in any configuration
Position	${}^0\mathbf{p}_{\text{config}} = {}^0\mathbf{r}_0$	constant (translational) position of mass object in any configuration
Displacement	${}^0\mathbf{u}_{\text{config}} = [0, 0, 0]^T$	(translational) displacement of mass object in any configuration
Velocity	${}^0\mathbf{v}_{\text{config}} = [0, 0, 0]^T$	(translational) velocity of mass object in any configuration
AngularVelocity	${}^0\boldsymbol{\omega}_{\text{config}} = {}^{0i}\mathbf{A}_0 \begin{bmatrix} 0 \\ 0 \\ \dot{\psi}_0 \end{bmatrix}^T$	
AngularVelocityLocal	${}^b\boldsymbol{\omega}_{\text{config}} = \begin{bmatrix} 0 \\ 0 \\ \dot{\psi}_0 \end{bmatrix}^T$	
RotationMatrix	${}^{0b}\mathbf{A} = {}^{0i}\mathbf{A}_0 \begin{bmatrix} \cos(\theta_0) & -\sin(\theta_0) & 0 \\ \sin(\theta_0) & \cos(\theta_0) & 0 \\ 0 & 0 & 1 \end{bmatrix}$	transformation of local body ( $b$ ) coordinates to global (0) coordinates
residual force	$\tau$	residual of all forces on mass object
applied force	${}^0\mathbf{f}_a = [f_0, f_1, f_2]^T$	3D applied force (loads, connectors, joint reaction forces, ...)
applied torque	${}^0\boldsymbol{\tau}_a = [\tau_0, \tau_1, \tau_2]^T$	3D applied torque (loads, connectors, joint reaction forces, ...)

A rigid body marker (e.g., MarkerBodyRigid) may be attached to this object and forces/torques can be applied. However, forces will have no effect and torques will only have effect in 'direction' of the coordinate.

### 7.2.5.3 Equations of motion

$$J \cdot \ddot{\psi}_0 = \tau. \quad (7.27)$$

Note that  $\tau$  is computed from all connectors and loads upon the object. E.g., a 3D torque vector  ${}^0\boldsymbol{\tau}_a$  is transformed to  $\tau$  as

$$\tau = {}^b[0, 0, 1] {}^{b0}\mathbf{A}_0 {}^0\boldsymbol{\tau}_a \quad (7.28)$$

Thus, the **rotation jacobian** reads

$$\mathbf{J}_{rot} = \partial \boldsymbol{\omega}_{cur} / \partial \dot{q}_{0,cur} = {}^b[0, 0, 1] {}^{b0}\mathbf{A}_0 \quad (7.29)$$


---

### 7.2.5.4 MINI EXAMPLE for ObjectRotationalMass1D

```

node = mbs.AddNode(Node1D(referenceCoordinates = [1], #\psi_0ref
                           initialCoordinates=[0.5],   #\psi_0ini
                           initialVelocities=[0.5]))  #\psi_t0ini
rotor = mbs.AddObject(Rotor1D(nodeNumber = node, physicsInertia=1))

#assemble and solve system for default parameters
mbs.Assemble()
exu.SolveDynamic(mbs)

#check result, get current rotor z-rotation at local position [0,0,0]
exudynTestGlobals.testResult = mbs.GetObjectOutputBody(rotor, exu.OutputVariableType.
Rotation, [0,0,0])
#final z-angle of rotor shall be 2

```

---

For examples on ObjectRotationalMass1D see Examples and TestModels:

- [driveTrainTest.py](#) (TestModels/)

## 7.2.6 ObjectRigidBody

A 3D rigid body which is attached to a 3D rigid body node. The rotation parametrization of the rigid body follows the rotation parametrization of the node. Use Euler parameters in the general case (no singularities) in combination with implicit solvers (GeneralizedAlpha or TrapezoidalIndex2), Tait-Bryan angles for special cases, e.g., rotors where no singularities occur if you rotate about  $x$  or  $z$  axis, or use Lie-group formulation with rotation vector together with explicit solvers. REMARK: Use the class `RigidBodyInertia`, see [Section 5.13.1](#) and `AddRigidBody(...)`, see [Section 5.13](#), of `exudyn.rigidBodyUtilities` to handle inertia, `COM` and mass.

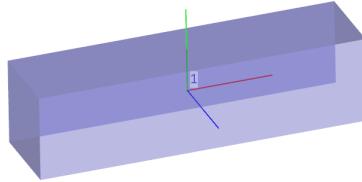


Figure 7.1: Example of ObjectRigidBody

### Additional information for ObjectRigidBody:

- The Object has the following types = `Body`, `SingleNoded`
- Requested node type = `Position + Orientation + RigidBody`
- **Short name for Python** = `RigidBody`
- **Short name for Python (visualization object)** = `VRigidBody`

The item `ObjectRigidBody` with type = 'RigidBody' has the following parameters:

Name	type	size	default value	description
name	String		"	objects's unique name
physicsMass	UReal		0.	mass [SI:kg] of rigid body
physicsInertia	Vector6D		[0.,0.,0.,0.,0.,0.]	inertia components [SI:kgm <sup>2</sup> ]: $[J_{xx}, J_{yy}, J_{zz}, J_{yz}, J_{xz}, J_{xy}]$ in body-fixed coordinate system and w.r.t. to the reference point of the body, NOT necessarily w.r.t. to <code>COM</code> ; use the class <code>RigidBodyInertia</code> and <code>AddRigidBody(...)</code> of <code>exudynRigidBodyUtilities.py</code> to handle inertia, <code>COM</code> and mass
physicsCenterOfMass	Vector3D	3	[0.,0.,0.]	local position of <code>COM</code> relative to the body's reference point; if the vector of the <code>COM</code> is [0,0,0], the computation will not consider additional terms for the <code>COM</code> and it is faster
nodeNumber	NodeIndex		MAXINT	node number (type <code>NodeIndex</code> ) for rigid body node
visualization	VObjectRigidBody			parameters for visualization of item

The item VObjectRigidBody has the following parameters:

<b>Name</b>	<b>type</b>	<b>size</b>	<b>default value</b>	<b>description</b>
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
graphicsDataUserFunction	PyFunctionGraphicsData	0		A Python function which returns a body-GraphicsData object, which is a list of graphics data in a dictionary computed by the user function; the graphics elements need to be defined in the local body coordinates and are transformed by mbs to global coordinates
graphicsData	BodyGraphicsData			Structure contains data for body visualization; data is defined in special list / dictionary structure

### 7.2.6.1 DESCRIPTION of ObjectRigidBody:

#### Information on input parameters:

<b>input parameter</b>	<b>symbol</b>	<b>description see tables above</b>
physicsMass	$m$	
physicsInertia	${}^b\mathbf{j}_6$	
physicsCenterOfMass	${}^b\mathbf{b}_{COM}$	
nodeNumber	$n0$	

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

<b>output variable</b>	<b>symbol</b>	<b>description</b>
Position	${}^0\mathbf{p}_{config}({}^b\mathbf{b}) = {}^0\mathbf{r}_{config} + {}^0\mathbf{r}_{ref} + {}^{0b}\mathbf{A} {}^b\mathbf{b}$	global position vector of body-fixed point given by local position vector ${}^b\mathbf{b}$
Displacement	${}^0\mathbf{u}_{config} + {}^{0b}\mathbf{A} {}^b\mathbf{b}$	global displacement vector of body-fixed point given by local position vector ${}^b\mathbf{b}$
Velocity	${}^0\mathbf{v}_{config}({}^b\mathbf{b}) = {}^0\dot{\mathbf{u}}_{config} + {}^{0b}\mathbf{A} ({}^b\boldsymbol{\omega} \times {}^b\mathbf{b}_{config})$	global velocity vector of body-fixed point given by local position vector ${}^b\mathbf{b}$
VelocityLocal	${}^b\mathbf{v}_{config}({}^b\mathbf{b}) = {}^{b0}\mathbf{A} {}^0\mathbf{v}_{config}({}^b\mathbf{b})$	local (body-fixed) velocity vector of body-fixed point given by local position vector ${}^b\mathbf{b}$
RotationMatrix	$\text{vec}({}^{0b}\mathbf{A}) = [A_{00}, A_{01}, A_{02}, A_{10}, \dots, A_{21}, A_{22}]_{config}^T$	vector with 9 components of the rotation matrix (row-major format)
Rotation		vector with 3 components of the Euler angles in xyz-sequence (R=Rx*Ry*Rz), recomputed from rotation matrix
AngularVelocity	${}^0\boldsymbol{\omega}_{config}$	angular velocity of body
AngularVelocityLocal	${}^b\boldsymbol{\omega}_{config}$	local (body-fixed) 3D velocity vector of node
Acceleration	${}^0\mathbf{a}_{config}({}^b\mathbf{b}) = {}^0\ddot{\mathbf{u}} + {}^0\boldsymbol{\alpha} \times ({}^{0b}\mathbf{A} {}^b\mathbf{b}) + {}^0\boldsymbol{\omega} \times ({}^0\boldsymbol{\omega} \times ({}^{0b}\mathbf{A} {}^b\mathbf{b}))$	global acceleration vector of body-fixed point given by local position vector ${}^b\mathbf{b}$

AccelerationLocal	${}^b\mathbf{a}_{\text{config}}({}^b\mathbf{b}) = ({}^{b0}\mathbf{A} \ {}^0\mathbf{a}_{\text{config}}({}^b\mathbf{b}))$	local (body-fixed) acceleration vector of body-fixed point given by local position vector ${}^b\mathbf{b}$
AngularAcceleration	${}^0\boldsymbol{\alpha}_{\text{config}}$	angular acceleration vector of body
AngularAccelerationLocal	${}^b\boldsymbol{\alpha}_{\text{config}} = ({}^{b0}\mathbf{A} \ {}^0\boldsymbol{\alpha}_{\text{config}})$	local angular acceleration vector of body

### 7.2.6.2 Definition of quantities

intermediate variables	symbol	description
inertia tensor	${}^b\mathbf{J} = \begin{bmatrix} J_{xx} & J_{xy} & J_{xz} \\ J_{yx} & J_{yy} & J_{yz} \\ J_{zx} & J_{zy} & J_{zz} \end{bmatrix}$	symmetric inertia tensor, based on components of ${}^b\mathbf{j}_6$ , in body-fixed (local) coordinates and w.r.t. body's reference point
reference coordinates	$\mathbf{q}_{\text{ref}} = [\mathbf{r}_{\text{ref}}^T, \boldsymbol{\psi}_{\text{ref}}^T]^T$	defines reference configuration, DIFFERENT meaning from body's reference point!
(relative) current coordinates	$\mathbf{q}_{\text{cur}} = [\mathbf{r}_{\text{cur}}^T, \boldsymbol{\psi}_{\text{cur}}^T]^T$	unknowns in solver; relative to the reference coordinates; current coordinates at initial configuration = initial coordinates $\mathbf{q}_{\text{ini}}$
current velocity coordinates	$\dot{\mathbf{q}}_{\text{cur}} = [\mathbf{v}_{\text{cur}}^T, \dot{\boldsymbol{\psi}}_{\text{cur}}^T]^T = [\dot{\mathbf{p}}_{\text{cur}}^T, \dot{\theta}_{\text{cur}}^T]^T$	current velocity coordinates
body's reference point	${}^0\mathbf{r}_{\text{config}} + {}^0\mathbf{r}_{\text{ref}} = {}^0\mathbf{p}(n_0)_{\text{config}}$	position of <b>body's reference point</b> provided by node $n_0$ in any configuration except for reference; if ${}^b\mathbf{b}_{\text{COM}} = [0, 0, 0]^T$ , this position becomes equal to the <b>COM</b> position
reference body's reference point	${}^0\mathbf{r}_{\text{ref}} = {}^0\mathbf{p}(n_0)_{\text{ref}}$	position of <b>body's reference point</b> in reference configuration
body's reference point displacement	${}^0\mathbf{u}_{\text{config}} = {}^0\mathbf{r}_{\text{config}} = [q_0, q_1, q_2]_{\text{config}}^T = {}^0\mathbf{u}(n_0)_{\text{config}}$	displacement of <b>body's reference point</b> which is provided by node $n_0$ in any configuration
body's reference point velocity	${}^0\mathbf{v}_{\text{config}} = {}^0\dot{\mathbf{r}}_{\text{config}} = [\dot{q}_0, \dot{q}_1, \dot{q}_2]_{\text{config}}^T = {}^0\mathbf{v}(n_0)_{\text{config}}$	velocity of <b>body's reference point</b> which is provided by node $n_0$ in any configuration
body's reference point acceleration	${}^0\mathbf{a}_{\text{config}} = [\ddot{q}_0, \ddot{q}_1, \ddot{q}_2]_{\text{config}}^T$	acceleration of <b>body's reference point</b> which is provided by node $n_0$ in any configuration
rotation coordinates	$\boldsymbol{\theta}_{\text{config}} = \boldsymbol{\psi}(n_0)_{\text{ref}} + \boldsymbol{\psi}(n_0)_{\text{config}}$	(total) rotation parameters of body as provided by node $n_0$ in any configuration
rotation parameters	$\boldsymbol{\theta}_{\text{config}} = \boldsymbol{\psi}(n_0)_{\text{ref}} + \boldsymbol{\psi}(n_0)_{\text{config}}$	(total) rotation parameters of body as provided by node $n_0$ in any configuration
body rotation matrix	${}^{0b}\mathbf{A}_{\text{config}} = {}^{0b}\mathbf{A}(n_0)_{\text{config}}$	rotation matrix which transforms local to global coordinates as given by node
local position	${}^b\mathbf{b} = [{}^b b_0, {}^b b_1, {}^b b_2]^T$	local position as used by markers or sensors
angular velocity	${}^0\boldsymbol{\omega}_{\text{config}} = {}^0[\omega_0(n_0), \omega_1(n_0), \omega_2(n_0)]_{\text{config}}^T$	global angular velocity of body as provided by node $n_0$ in any configuration
local angular velocity	${}^b\boldsymbol{\omega}_{\text{config}}$	local angular velocity of body as provided by node $n_0$ in any configuration
body angular acceleration	${}^0\boldsymbol{\alpha}_{\text{config}} = {}^0\dot{\boldsymbol{\omega}}_{\text{config}}$	angular acceleratoion of body as provided by node $n_0$ in any configuration
applied forces	${}^0\mathbf{f}_a = [f_0, f_1, f_2]^T$	calculated from loads, connectors, ...
applied torques	${}^0\boldsymbol{\tau}_a = [\tau_0, \tau_1, \tau_2]^T$	calculated from loads, connectors, ...

constraint reaction forces	${}^0\mathbf{f}_\lambda = [f_{\lambda 0}, f_{\lambda 1}, f_{\lambda 2}]^T$	calculated from joints or constraint)
constraint reaction torques	${}^0\boldsymbol{\tau}_\lambda = [\tau_{\lambda 0}, \tau_{\lambda 1}, \tau_{\lambda 2}]^T$	calculated from joints or constraints

### 7.2.6.3 Rotation parametrization

The equations of motion of the rigid body build upon a specific parameterization of the rigid body coordinates. Rigid body coordinates are defined by the underlying node given by `nodeNumber n0`. Appropriate nodes are

- `NodeRigidBodyEP` (Euler parameters)
- `NodeRigidBodyRxyz` (Euler angles / Tait Bryan angles)
- `NodeRigidBodyRotVecLG` (Rotation vector with Lie group integration option)

Note that all operations for rotation parameters, such as the computation of the rotation matrix, must be performed with the rotation parameters  $\theta$ , see table above, which are the sum of reference and current coordinates.

The angular velocity in body-fixed coordinates is related to the rotation parameters by means of a matrix  ${}^b\mathbf{G}_{rp}$ ,

$${}^b\boldsymbol{\omega} = {}^b\mathbf{G}_{rp} \dot{\boldsymbol{\theta}} = {}^b\mathbf{G}_{rp} \dot{\psi}, \quad (7.30)$$

and is specific for any rotation parametrization  $rp$ . The angular velocity in global coordinates is related to the rotation parameters by means of a matrix  ${}^0\mathbf{G}_{rp}$ ,

$${}^0\boldsymbol{\omega} = {}^0\mathbf{G}_{rp} \dot{\boldsymbol{\theta}}. \quad (7.31)$$

The local angular accelerations follow as

$${}^b\boldsymbol{\alpha} = {}^b\dot{\boldsymbol{\omega}} = {}^b\mathbf{G}_{rp} \ddot{\boldsymbol{\theta}} + {}^b\dot{\mathbf{G}}_{rp} \dot{\boldsymbol{\theta}}, \quad (7.32)$$

remember that derivatives for angular velocities can also be done in the local frame. In case of Euler parameters and the Lie-group rotation vector we find that  ${}^b\dot{\mathbf{G}}_{rp} \dot{\boldsymbol{\theta}} = \mathbf{0}$ .

### 7.2.6.4 Equations of motion for COM

The equations of motion for a rigid body, the so-called Newton-Euler equations, can be written for the special case of the reference point = `COM` and split for translations and rotations, using a coordinate-free notation,

$$\begin{bmatrix} m\mathbf{I}_{3 \times 3} & \mathbf{0} \\ \mathbf{0} & \mathbf{J} \end{bmatrix} \begin{bmatrix} \mathbf{a}_{COM} \\ \boldsymbol{\alpha} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ -\tilde{\omega}\mathbf{J}\boldsymbol{\omega} \end{bmatrix} + \begin{bmatrix} \mathbf{f}_a \\ \boldsymbol{\tau}_a \end{bmatrix} + \begin{bmatrix} \mathbf{f}_\lambda \\ \boldsymbol{\tau}_\lambda \end{bmatrix} \quad (7.33)$$

with the  $3 \times 3$  unit matrix  $\mathbf{I}_{3 \times 3}$  and forces  $\mathbf{f}$  resp. torques  $\boldsymbol{\tau}$  as described in the table above. A change of the reference point, using the vector  $\mathbf{b}_{COM}$  from the body's reference point  $\mathbf{p}$  to the `COM` position, is simple by replacing `COM` accelerations using the common relation known from Euler

$$\mathbf{a}_{COM} = \mathbf{a} + \tilde{\alpha}\mathbf{b}_{COM} + \tilde{\omega}\tilde{\omega}\mathbf{b}_{COM}, \quad (7.34)$$

which is inserted into the first line of Eq. (7.33). Additionally, the second line of Eq. (7.33) (second Euler equation related to rate of angular momentum) is rewritten for an arbitrary reference point,  $\mathbf{b}_{COM}$  denoting the vector from the body reference point to `COM`, using the well known relation

$$m\tilde{\mathbf{b}}_{COM}\boldsymbol{\alpha} + \mathbf{J}\boldsymbol{\alpha} + \tilde{\omega}\mathbf{J}\boldsymbol{\omega} = \boldsymbol{\tau}_a + \boldsymbol{\tau}_\lambda \quad (7.35)$$

### 7.2.6.5 Equations of motion for arbitrary reference point

This immediately leads to the equations of motion for the rigid body with respect to an arbitrary reference point ( $\neq \text{COM}$ ), see e.g. [40](page 258ff.), which have the general coordinate-free form

$$\begin{bmatrix} m\mathbf{I}_{3\times 3} & -m\tilde{\mathbf{b}}_{\text{COM}} \\ m\tilde{\mathbf{b}}_{\text{COM}} & \mathbf{J} \end{bmatrix} \begin{bmatrix} \mathbf{a} \\ \boldsymbol{\alpha} \end{bmatrix} = \begin{bmatrix} -m\tilde{\omega}\tilde{\omega}\mathbf{b}_{\text{COM}} \\ -\tilde{\omega}\mathbf{J}\boldsymbol{\omega} \end{bmatrix} + \begin{bmatrix} \mathbf{f}_a \\ \boldsymbol{\tau}_a \end{bmatrix} + \begin{bmatrix} \mathbf{f}_\lambda \\ \boldsymbol{\tau}_\lambda \end{bmatrix}, \quad (7.36)$$

in which  $\mathbf{J}$  is the inertia tensor w.r.t. the chosen reference point (which has local coordinates  ${}^b[0, 0, 0]^T$ ). Eq. (7.36) can be written in the global frame (0),

$$\begin{bmatrix} m\mathbf{I}_{3\times 3} & -m{}^0\tilde{\mathbf{b}}_{\text{COM}} \\ m{}^0\tilde{\mathbf{b}}_{\text{COM}} & {}^0\mathbf{J} \end{bmatrix} \begin{bmatrix} {}^0\mathbf{a} \\ {}^0\boldsymbol{\alpha} \end{bmatrix} = \begin{bmatrix} -m{}^0\tilde{\omega}{}^0\tilde{\omega}{}^0\mathbf{b}_{\text{COM}} \\ -{}^0\tilde{\omega}{}^0\mathbf{J}{}^0\boldsymbol{\omega} \end{bmatrix} + \begin{bmatrix} {}^0\mathbf{f}_a \\ {}^0\boldsymbol{\tau}_a \end{bmatrix} + \begin{bmatrix} {}^0\mathbf{f}_\lambda \\ {}^0\boldsymbol{\tau}_\lambda \end{bmatrix}. \quad (7.37)$$

Expressing the translational part (first line) of Eq. (7.37) in the global frame (0), using local coordinates (b) for quantities that are constant in the body-fixed frame,  ${}^b\mathbf{J}$  and  ${}^b\mathbf{b}_{\text{COM}}$ , thus expressing also the angular velocity  ${}^b\boldsymbol{\omega}$  in the body-fixed frame, applying Eq. (7.30) and Eq. (7.32), and using the relations

$${}^0\tilde{\omega}{}^0\tilde{\omega}{}^0\mathbf{b}_{\text{COM}} = {}^{0b}\mathbf{A}{}^b\tilde{\omega}{}^b\tilde{\omega}{}^b\mathbf{b}_{\text{COM}} = -{}^{0b}\mathbf{A}{}^b\tilde{\omega}{}^b\tilde{\mathbf{b}}_{\text{COM}}{}^b\boldsymbol{\omega} = -{}^{0b}\mathbf{A}{}^b\tilde{\omega}{}^b\tilde{\mathbf{b}}_{\text{COM}}{}^b\mathbf{G}_{rp}\dot{\theta}, \quad (7.38)$$

$$-m{}^0\tilde{\mathbf{b}}_{\text{COM}}{}^0\tilde{\boldsymbol{\alpha}} = -m{}^{0b}\mathbf{A}{}^b\tilde{\mathbf{b}}_{\text{COM}}{}^b\tilde{\boldsymbol{\alpha}} = -m{}^{0b}\mathbf{A}{}^b\tilde{\mathbf{b}}_{\text{COM}}\left({}^b\mathbf{G}_{rp}\ddot{\theta} + {}^b\dot{\mathbf{G}}_{rp}\dot{\theta}\right), \quad (7.39)$$

we obtain

$$\begin{aligned} & \begin{bmatrix} m\mathbf{I}_{3\times 3} & -m{}^{0b}\mathbf{A}{}^b\tilde{\mathbf{b}}_{\text{COM}}{}^b\mathbf{G}_{rp} \\ m{}^b\mathbf{G}_{rp}^T{}^b\tilde{\mathbf{b}}_{\text{COM}}{}^{0b}\mathbf{A}^T & {}^b\mathbf{G}_{rp}^T{}^b\mathbf{J}{}^b\mathbf{G}_{rp} \end{bmatrix} \begin{bmatrix} {}^0\mathbf{a} \\ \dot{\theta} \end{bmatrix} \\ &= \begin{bmatrix} m{}^{0b}\mathbf{A}{}^b\tilde{\omega}{}^b\tilde{\mathbf{b}}_{\text{COM}}{}^b\boldsymbol{\omega} + m{}^{0b}\mathbf{A}{}^b\tilde{\mathbf{b}}_{\text{COM}}{}^b\dot{\mathbf{G}}_{rp}\dot{\theta} \\ -{}^b\mathbf{G}_{rp}^T{}^b\tilde{\omega}{}^b\mathbf{J}{}^b\boldsymbol{\omega} - {}^b\mathbf{G}_{rp}^T{}^b\mathbf{J}{}^b\dot{\mathbf{G}}_{rp}\dot{\theta} \end{bmatrix} + \begin{bmatrix} {}^0\mathbf{f}_a \\ {}^0\mathbf{G}_{rp}^T{}^0\boldsymbol{\tau}_a \end{bmatrix} + \begin{bmatrix} {}^0\mathbf{f}_\lambda \\ \mathbf{f}_{\theta,\lambda} \end{bmatrix} \end{aligned} \quad (7.40)$$

with constraint reaction forces  $\mathbf{f}_{\theta,\lambda}$  for the rotation parameters. Note that the last line has been pre-multiplied with  ${}^b\mathbf{G}_{rp}^T$  (in order to make the mass matrix symmetric) and that  ${}^b\dot{\mathbf{G}}_{rp}\dot{\theta} = \mathbf{0}$  in case of Euler parameters and the Lie-group rotation vector .

### 7.2.6.6 Euler parameters

In case of Euler parameters, a constraint equation is automatically added, reading for the index 3 case

$$g_\theta(\boldsymbol{\theta}) = \theta_0^2 + \theta_1^2 + \theta_2^2 + \theta_3^2 - 1 = 0 \quad (7.41)$$

and for the index 2 case

$$\dot{g}_\theta(\boldsymbol{\theta}) = 2\theta_0\dot{\theta}_0 + 2\theta_1\dot{\theta}_1 + 2\theta_2\dot{\theta}_2 + 2\theta_3\dot{\theta}_3 = 0 \quad (7.42)$$

Given a Lagrange parameter (algebraic variable)  $\lambda_\theta$  related to the Euler parameter constraint (7.41), the constraint reaction forces in Eq. (7.40) then read

$$\mathbf{f}_{\theta,\lambda} = \frac{\partial g_\theta}{\partial \boldsymbol{\theta}^T} \lambda_\theta = [2\theta_0, 2\theta_1, 2\theta_2, 2\theta_3]^T \quad (7.43)$$

### Userfunction: `graphicsDataUserFunction(mbs, itemNumber)`

A user function, which is called by the visualization thread in order to draw user-defined objects. The function can be used to generate any BodyGraphicsData, see Section 9.3. Use `graphicsDataUtilities` functions, see Section 5.5, to create more complicated objects. Note that `graphicsDataUserFunction` needs to copy lots of data and is therefore inefficient and only designed to enable simpler tests, but not large scale problems.

For an example for `graphicsDataUserFunction` see ObjectGround, [Section 7.2.1](#).

arguments / return	type or size	description
mbs	MainSystem	provides reference to mbs, which can be used in user function to access all data of the object
itemNumber	Index	integer number of the object in mbs, allowing easy access
<b>return value</b>	BodyGraphicsData	list of GraphicsData dictionaries, see Section 9.3

For creating a ObjectRigidBody, there is a `rigidBodyUtilities` function `AddRigidBody`, see [Section 5.13](#), which simplifies the setup of a rigid body significantly!

---

For examples on ObjectRigidBody see Examples and TestModels:

- [`rigid3Dexample.py`](#) (Examples/)
- [`rigidBodyIMUtest.py`](#) (Examples/)
- [`addPrismaticJoint.py`](#) (Examples/)
- [`addRevoluteJoint.py`](#) (Examples/)
- [`bicycleIftommBenchmark.py`](#) (Examples/)
- [`fourBarMechanism3D.py`](#) (Examples/)
- [`gyroStability.py`](#) (Examples/)
- [`leggedRobot.py`](#) (Examples/)
- [`mouseInteractionExample.py`](#) (Examples/)
- [`multiMbsTest.py`](#) (Examples/)
- [`particleClusters.py`](#) (Examples/)
- [`plotSensorExamples.py`](#) (Examples/)
- ...
- [`explicitLieGroupIntegratorPythonTest.py`](#) (TestModels/)
- [`explicitLieGroupIntegratorTest.py`](#) (TestModels/)
- [`explicitLieGroupMBSTest.py`](#) (TestModels/)
- ...

## 7.2.7 ObjectRigidBody2D

A 2D rigid body which is attached to a rigid body 2D node. The body obtains coordinates, position, velocity, etc. from the underlying 2D node

**Additional information for ObjectRigidBody2D:**

- The Object has the following types = Body, SingleNoded
- Requested node type = Position2D + Orientation2D + Position + Orientation
- **Short name for Python** = **RigidBody2D**
- **Short name for Python (visualization object)** = **VRigidBody2D**

The item **ObjectRigidBody2D** with type = 'RigidBody2D' has the following parameters:

Name	type	size	default value	description
name	String		"	objects's unique name
physicsMass	UReal		0.	mass [SI:kg] of rigid body
physicsInertia	UReal		0.	inertia [SI:kgm <sup>2</sup> ] of rigid body w.r.t. center of mass
nodeNumber	NodeIndex		MAXINT	node number (type NodeIndex) for 2D rigid body node
visualization	VObjectRigidBody2D			parameters for visualization of item

The item VObjectRigidBody2D has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
graphicsDataUserFunction	PyFunctionGraphicsData		0	A Python function which returns a body-GraphicsData object, which is a list of graphics data in a dictionary computed by the user function; the graphics elements need to be defined in the local body coordinates and are transformed by mbs to global coordinates
graphicsData	BodyGraphicsData			Structure contains data for body visualization; data is defined in special list / dictionary structure

### 7.2.7.1 DESCRIPTION of ObjectRigidBody2D:

**Information on input parameters:**

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

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

output variable	symbol	description
Position	${}^0\mathbf{p}_{\text{config}}({}^b\mathbf{b}) = {}^0\mathbf{r}_{\text{config}} + {}^0\mathbf{r}_{\text{ref}} + {}^{0b}\mathbf{A} {}^b\mathbf{b}$	global position vector of body-fixed point given by local position vector ${}^b\mathbf{b}$
Displacement	${}^0\mathbf{u}_{\text{config}} + {}^{0b}\mathbf{A} {}^b\mathbf{b}$	global displacement vector of body-fixed point given by local position vector ${}^b\mathbf{b}$
Velocity	${}^0\mathbf{v}_{\text{config}}({}^b\mathbf{b}) = {}^0\dot{\mathbf{u}}_{\text{config}} + {}^{0b}\mathbf{A}({}^b\omega \times {}^b\mathbf{b}_{\text{config}})$	global velocity vector of body-fixed point given by local position vector ${}^b\mathbf{b}$
VelocityLocal	${}^b\mathbf{v}_{\text{config}}({}^b\mathbf{b}) = {}^{b0}\mathbf{A} {}^0\mathbf{v}_{\text{config}}({}^b\mathbf{b})$	local (body-fixed) velocity vector of body-fixed point given by local position vector ${}^b\mathbf{b}$
RotationMatrix	$\text{vec}({}^{0b}\mathbf{A}) = [A_{00}, A_{01}, A_{02}, A_{10}, \dots, A_{21}, A_{22}]_{\text{config}}^T$	vector with 9 components of the rotation matrix (row-major format)
Rotation	$\theta_{0\text{config}}$	scalar rotation angle of body
AngularVelocity	${}^0\boldsymbol{\omega}_{\text{config}}$	angular velocity of body
AngularVelocityLocal	${}^b\boldsymbol{\omega}_{\text{config}}$	local (body-fixed) 3D velocity vector of node
Acceleration	${}^0\mathbf{a}_{\text{config}}({}^b\mathbf{b}) = {}^0\ddot{\mathbf{u}} + {}^0\boldsymbol{\alpha} \times ({}^{0b}\mathbf{A} {}^b\mathbf{b}) + {}^0\boldsymbol{\omega} \times ({}^0\boldsymbol{\omega} \times ({}^{0b}\mathbf{A} {}^b\mathbf{b}))$	global acceleration vector of body-fixed point given by local position vector ${}^b\mathbf{b}$
AccelerationLocal	${}^b\mathbf{a}_{\text{config}}({}^b\mathbf{b}) = {}^{b0}\mathbf{A} {}^0\mathbf{a}_{\text{config}}({}^b\mathbf{b})$	local (body-fixed) acceleration vector of body-fixed point given by local position vector ${}^b\mathbf{b}$
AngularAcceleration	${}^0\boldsymbol{\alpha}_{\text{config}}$	angular acceleration vector of body
AngularAccelerationLocal	${}^b\boldsymbol{\alpha}_{\text{config}} = {}^{b0}\mathbf{A} {}^0\boldsymbol{\alpha}_{\text{config}}$	local angular acceleration vector of body

### 7.2.7.2 Definition of quantities

intermediate variables	symbol	description
COM position	${}^0\mathbf{r}_{\text{config}} + {}^0\mathbf{r}_{\text{ref}} = {}^0\mathbf{p}(n_0)_{\text{config}}$	reference point, equal to the position of COM; provided by node $n_0$ in any configuration (except reference)
COM displacement	${}^0\mathbf{u}_{\text{config}} = {}^0\mathbf{r}_{\text{config}} = [q_0, q_1, 0]_{\text{config}}^T = {}^0\mathbf{u}(n_0)_{\text{config}}$	displacement of center of mass which is provided by node $n_0$ in any configuration; NOTE that for configurations other than reference, it follows that ${}^0\mathbf{r}_{\text{ref}} - {}^0\mathbf{r}_{\text{config}}$
COM velocity	${}^0\mathbf{v}_{\text{config}} = [q_0, q_1, 0]_{\text{config}}^T = {}^0\mathbf{v}(n_0)_{\text{config}}$	velocity of center of mass which is provided by node $n_0$ in any configuration
body rotation	${}^0\theta_{0\text{config}} = \theta_{0(n_0)\text{config}} = \psi_0(n_0)_{\text{ref}} + \psi_0(n_0)_{\text{config}}$	rotation of body as provided by node $n_0$ in any configuration
body rotation matrix	${}^{0b}\mathbf{A}_{\text{config}} = {}^{0b}\mathbf{A}(n_0)_{\text{config}}$	rotation matrix which transforms local to global coordinates as given by node
local position	${}^b\mathbf{b} = [{}^b b_0, {}^b b_1, 0]^T$	local position as used by markers or sensors
body angular velocity	${}^0\boldsymbol{\omega}_{\text{config}} = {}^0[\omega_0(n_0), 0, 0]_{\text{config}}^T$	rotation of body as provided by node $n_0$ in any configuration
(generalized) coordinates	$\mathbf{c}_{\text{config}} = [q_0, q_1, \psi_0]^T$	generalized coordinates of body (= coordinates of node)
generalized forces	${}^0\mathbf{f} = [f_0, f_1, \tau_2]^T$	generalized forces applied to body
applied forces	${}^0\mathbf{f}_a = [f_0, f_1, 0]^T$	applied forces (loads, connectors, joint reaction forces, ...)
applied torques	${}^0\boldsymbol{\tau}_a = [0, 0, \tau_2]^T$	applied torques (loads, connectors, joint reaction forces, ...)

### 7.2.7.3 Equations of motion

$$\begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & J \end{bmatrix} \begin{bmatrix} \ddot{q}_0 \\ \ddot{q}_1 \\ \ddot{\psi}_0 \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ \tau_2 \end{bmatrix} = \mathbf{f}. \quad (7.44)$$

For example, a LoadCoordinate on coordinate 2 of the node would add a torque  $\tau_2$  on the RHS.

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

$${}^0\mathbf{J}_{\text{pos}} = \partial {}^0\mathbf{p}_{\text{config}}({}^b\mathbf{b})_{\text{cur}} / \partial \mathbf{c}_{\text{cur}} = \begin{bmatrix} 1 & 0 & -\sin(\theta) {}^b b_0 - \cos(\theta) {}^b b_1 \\ 0 & 1 & \cos(\theta) {}^b b_0 - \sin(\theta) {}^b b_1 \\ 0 & 0 & 0 \end{bmatrix} \quad (7.45)$$

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

$$\mathbf{Q} = {}^0\mathbf{J}_{\text{pos}}^T {}^0\mathbf{f}_a \quad (7.46)$$

The **rotation jacobian**, which is computed from angular velocity, reads

$${}^0\mathbf{J}_{\text{rot}} = \partial {}^0\boldsymbol{\omega}_{\text{cur}} / \partial \dot{\mathbf{c}}_{\text{cur}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (7.47)$$

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

$$\mathbf{Q} = {}^0\mathbf{J}_{\text{rot}}^T {}^0\boldsymbol{\tau}_a \quad (7.48)$$


---

### Userfunction: `graphicsDataUserFunction(mbs, itemNumber)`

A user function, which is called by the visualization thread in order to draw user-defined objects. The function can be used to generate any BodyGraphicsData, see Section 9.3. Use `graphicsDataUtilities` functions, see Section 5.5, to create more complicated objects. Note that `graphicsDataUserFunction` needs to copy lots of data and is therefore inefficient and only designed to enable simpler tests, but not large scale problems.

For an example for `graphicsDataUserFunction` see ObjectGround, [Section 7.2.1](#).

arguments / return	type or size	description
<code>mbs</code>	MainSystem	provides reference to <code>mbs</code> , which can be used in user function to access all data of the object
<code>itemNumber</code>	int	integer number of the object in <code>mbs</code> , allowing easy access
<code>return value</code>	BodyGraphicsData	list of GraphicsData dictionaries, see Section 9.3

---

### 7.2.7.4 MINI EXAMPLE for ObjectRigidBody2D

```

node = mbs.AddNode(NodeRigidBody2D(referenceCoordinates = [1,1,0.25*np.pi],
                                    initialCoordinates=[0.5,0,0],
                                    initialVelocities=[0.5,0,0.75*np.pi]))
mbs.AddObject(RigidBody2D(nodeNumber = node, physicsMass=1, physicsInertia=2))

#assemble and solve system for default parameters
mbs.Assemble()
exu.SolveDynamic(mbs)

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

```

For examples on ObjectRigidBody2D see Examples and TestModels:

- [beltDriveReevingSystem.py](#) (Examples/)
- [reevingSystem.py](#) (Examples/)
- [sliderCrank3DwithANCFbeltDrive2.py](#) (Examples/)
- [ANCF\\_moving\\_rigidbody.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- [ANCF\\_slidingJoint2Drigid.py](#) (Examples/)
- [ANCF\\_switchingSlidingJoint2D.py](#) (Examples/)
- [finiteSegmentMethod.py](#) (Examples/)
- [geneticOptimizationSliderCrank.py](#) (Examples/)
- [lavalRotor2Dtest.py](#) (Examples/)
- [rigid\\_pendulum.py](#) (Examples/)
- [SliderCrank.py](#) (Examples/)
- ...
- [ANCFbeltDrive.py](#) (TestModels/)
- [ANCFgeneralContactCircle.py](#) (TestModels/)
- [ANCFslidingAndALEjointTest.py](#) (TestModels/)
- ...

## 7.3 Objects (SuperElement)

A SuperElement is a special Object which acts on a set of nodes. Essentially, SuperElements can be linked with special SuperElement markers. SuperElements may represent complex flexible bodies, based on finite element formulations.

### 7.3.1 ObjectGenericODE2

A system of  $n$  second order ordinary differential equations ([ODE2](#)), having a mass matrix, damping/gyroscopic matrix, stiffness matrix and generalized forces. It can combine generic nodes, or node points. User functions can be used to compute mass matrix and generalized forces depending on given coordinates. NOTE that all matrices, vectors, etc. must have the same dimensions  $n$  or  $(n \times n)$ , or they must be empty ( $0 \times 0$ ), except for the mass matrix which always needs to have dimensions  $(n \times n)$ .

**Additional information for ObjectGenericODE2:**

- The Object has the following types = Body, MultiNoded, SuperElement
- Requested node type: read detailed information of item

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

Name	type	size	default value	description
name	String		"	objects's unique name
nodeNumbers	ArrayNodeIndex		[]	node numbers which provide the coordinates for the object (consecutively as provided in this list)
massMatrix	PyMatrixContainer		PyMatrixContainer[]	mass matrix of object as MatrixContainer (or numpy array / list of lists)
stiffnessMatrix	PyMatrixContainer		PyMatrixContainer[]	stiffness matrix of object as MatrixContainer (or numpy array / list of lists); NOTE that (dense/sparse triplets) format must agree with dampingMatrix and jacobianUserFunction
dampingMatrix	PyMatrixContainer		PyMatrixContainer[]	damping matrix of object as MatrixContainer (or numpy array / list of lists); NOTE that (dense/sparse triplets) format must agree with stiffnessMatrix and jacobianUserFunction
forceVector	NumpyVector		[]	generalized force vector added to RHS
forceUserFunction	PyFunctionVectorMbsScalarIndex2Vector		0	A Python user function which computes the generalized user force vector for the <a href="#">ODE2</a> equations; see description below
massMatrixUserFunction	PyFunctionMatrixContainerMbsScalarIndex2Vector		0	A Python user function which computes the mass matrix instead of the constant mass matrix given in <b>M</b> ; return numpy array or MatrixContainer; see description below

jacobianUserFunction	PyFunctionMatrixContainer	MbsScalarIndex2Vector2Scalar	0	A Python user function which computes the jacobian, i.e., the derivative of the left-hand-side object equation w.r.t. the coordinates (times $f_{ODE2}$ ) and w.r.t. the velocities (times $f_{ODE2_i}$ ). Terms on the RHS must be subtracted from the LHS equation; the respective terms for the stiffness matrix and damping matrix are automatically added; see description below
coordinateIndexPerNode	ArrayIndex	[]		this list contains the local coordinate index for every node, which is needed, e.g., for markers; the list is generated automatically every time parameters have been changed
tempCoordinates	NumpyVector	[]		temporary vector containing coordinates
tempCoordinates_t	NumpyVector	[]		temporary vector containing velocity coordinates
tempCoordinates_tt	NumpyVector	[]		temporary vector containing acceleration coordinates
visualization	VObjectGenericODE2			parameters for visualization of item

The item VObjectGenericODE2 has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
color	Float4	4	[-1,-1,-1,-1.]	RGBA color for object; 4th value is alpha-transparency; R=-1.f means, that default color is used
triangleMesh	NumpyMatrixI		MatrixI[]	a matrix, containing node number triples in every row, referring to the node numbers of the GenericODE2 object; the mesh uses the nodes to visualize the underlying object; contour plot colors are still computed in the local frame!
showNodes	Bool		False	set true, nodes are drawn uniquely via the mesh, eventually using the floating reference frame, even in the visualization of the node is show=False; node numbers are shown with indicator 'NF'
graphicsDataUserFunction	PyFunctionGraphicsData	0		A Python function which returns a body-GraphicsData object, which is a list of graphics data in a dictionary computed by the user function; the graphics data is drawn in global coordinates; it can be used to implement user element visualization, e.g., beam elements or simple mechanical systems; note that this user function may significantly slow down visualization

---

### 7.3.1.1 DESCRIPTION of ObjectGenericODE2:

**Information on input parameters:**

input parameter	symbol	description see tables above
nodeNumbers	$\mathbf{n}_n = [n_0, \dots, n_n]^T$	
massMatrix	$\mathbf{M} \in \mathbb{R}^{n \times n}$	
stiffnessMatrix	$\mathbf{K} \in \mathbb{R}^{n \times n}$	
dampingMatrix	$\mathbf{D} \in \mathbb{R}^{n \times n}$	
forceVector	$\mathbf{f} \in \mathbb{R}^n$	
forceUserFunction	$\mathbf{f}_{user} \in \mathbb{R}^n$	
massMatrixUserFunction	$\mathbf{M}_{user} \in \mathbb{R}^{n \times n}$	
jacobianUserFunction	$\mathbf{J}_{user} \in \mathbb{R}^{n \times n}$	
tempCoordinates	$\mathbf{c}_{temp} \in \mathbb{R}^n$	
tempCoordinates_t	$\dot{\mathbf{c}}_{temp} \in \mathbb{R}^n$	
tempCoordinates_tt	$\ddot{\mathbf{c}}_{temp} \in \mathbb{R}^n$	

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

output variable	symbol	description
Coordinates		all ODE2 coordinates
Coordinates_t		all ODE2 velocity coordinates
Coordinates_tt		all ODE2 acceleration coordinates
Force		generalized forces for all coordinates (residual of all forces except mass*acceleration; corresponds to ComputeODE2LHS)

### 7.3.1.2 Additional output variables for superelement node access

Functions like GetObjectOutputSuperElement(...), see [Section 4.4.2](#), or SensorSuperElement, see [Section 4.4.5](#), directly access special output variables (OutputVariableType) of the mesh nodes of the superelement. Additionally, the contour drawing of the object can make use the OutputVariableType of the meshnodes.

For this object, all nodes of ObjectGenericODE2 map their OutputVariableType to the meshnode → see at the according node for the list of OutputVariableType.

### 7.3.1.3 Equations of motion

An object with node numbers  $[n_0, \dots, n_n]$  and according numbers of nodal coordinates  $[n_{c_0}, \dots, n_{c_n}]$ , the total number of equations (=coordinates) of the object is

$$n = \sum_i n_{c_i}, \quad (7.49)$$

which is used throughout the description of this object.

### 7.3.1.4 Equations of motion

The equations of motion read,

$$\mathbf{M}\ddot{\mathbf{q}} + \mathbf{D}\dot{\mathbf{q}} + \mathbf{K}\mathbf{q} = \mathbf{f} + \mathbf{f}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}}) \quad (7.50)$$

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

In case that a user mass matrix is specified, Eq. (7.50) is replaced with

$$\mathbf{M}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}})\ddot{\mathbf{q}} + \mathbf{D}\dot{\mathbf{q}} + \mathbf{K}\mathbf{q} = \mathbf{f} + \mathbf{f}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}}) \quad (7.51)$$

The (internal) Jacobian  $\mathbf{J}$  of Eq. (7.50) (assuming  $\mathbf{f}$  to be constant!) reads

$$\mathbf{J} = f_{ODE2} \left( \mathbf{K} - \frac{\partial \mathbf{f}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}})}{\partial \mathbf{q}} \right) + f_{ODE2_t} \left( \mathbf{D} - \frac{\partial \mathbf{f}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}})}{\partial \dot{\mathbf{q}}} \right) + \dots \quad (7.52)$$

Choosing  $f_{ODE2} = 1$  and  $f_{ODE2_t} = 0$  would immediately give the jacobian of position quantities.

If no `jacobianUserFunction` is specified, the jacobian is – as with many objects in ExUDYN – computed by means of numerical differentiation. In case that a `jacobianUserFunction` is specified, it must represent the jacobian of the LHS of Eq. (7.50) without  $\mathbf{K}$  and  $\mathbf{D}$  (these matrices are added internally),

$$\mathbf{J}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}}, f_{ODE2}, f_{ODE2_t}) = -f_{ODE2} \left( \frac{\partial \mathbf{f}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}})}{\partial \mathbf{q}} \right) - f_{ODE2_t} \left( \frac{\partial \mathbf{f}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}})}{\partial \dot{\mathbf{q}}} \right) \quad (7.53)$$

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

---

#### Userfunction: `forceUserFunction(mbs, t, itemNumber, q, q_t)`

A user function, which computes a force vector depending on current time and states of object. Can be used to create any kind of mechanical system by using the object states. Note that itemNumber represents the index of the `ObjectGenericODE2` object in mbs, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

arguments / return	type or size	description
<code>mbs</code>	MainSystem	provides MainSystem mbs to which object belongs
<code>t</code>	Real	current time in mbs
<code>itemNumber</code>	Index	integer number $i_N$ of the object in mbs, allowing easy access to all object data via <code>mbs.GetObjectParameter(itemNumber, ...)</code>
<code>q</code>	Vector $\in \mathbb{R}^n$	object coordinates (e.g., nodal displacement coordinates) in current configuration, without reference values
<code>q_t</code>	Vector $\in \mathbb{R}^n$	object velocity coordinates (time derivative of $q$ ) in current configuration
<code>return value</code>	Vector $\in \mathbb{R}^n$	returns force vector for object

---

#### Userfunction: `massMatrixUserFunction(mbs, t, itemNumber, q, q_t)`

A user function, which computes a mass matrix depending on current time and states of object. Can be used to create any kind of mechanical system by using the object states.

arguments / return	type or size	description
mbs	MainSystem	provides MainSystem mbs to which object belongs to
t	Real	current time in mbs
itemNumber	Index	integer number $i_N$ of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
q	Vector $\in \mathbb{R}^n$	object coordinates (e.g., nodal displacement coordinates) in current configuration, without reference values
q_t	Vector $\in \mathbb{R}^n$	object velocity coordinates (time derivative of q) in current configuration
<b>return value</b>	MatrixContainer $\in \mathbb{R}^{n \times n}$	returns mass matrix for object, as exu.MatrixContainer, numpy array or list of lists; use MatrixContainer sparse format for larger matrices to speed up computations.

---

### Userfunction: jacobianUserFunction(mbs, t, itemNumber, q, q\_t, fODE2, fODE2\_t)

A user function, which computes the jacobian of the LHS of the equations of motion, depending on current time, states of object and two factors which are used to distinguish between position level and velocity level derivatives. Can be used to create any kind of mechanical system by using the object states.

arguments / return	type or size	description
mbs	MainSystem	provides MainSystem mbs to which object belongs to
t	Real	current time in mbs
itemNumber	Index	integer number $i_N$ of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
q	Vector $\in \mathbb{R}^n$	object coordinates (e.g., nodal displacement coordinates) in current configuration, without reference values
q_t	Vector $\in \mathbb{R}^n$	object velocity coordinates (time derivative of q) in current configuration
fODE2	Real	factor to be multiplied with the position level jacobian, see Eq. (7.53)
fODE2_t	Real	factor to be multiplied with the velocity level jacobian, see Eq. (7.53)
<b>return value</b>	MatrixContainer $\in \mathbb{R}^{n \times n}$	returns special jacobian for object, as exu.MatrixContainer, numpy array or list of lists; use MatrixContainer sparse format for larger matrices to speed up computations; NOTE that the format of returnValue must AGREE with (dense/sparse triplet) format of stiffnessMatrix and dampingMatrix; sparse triplets MAY NOT contain zero values!

---

### Userfunction: `graphicsDataUserFunction(mbs, itemNumber)`

A user function, which is called by the visualization thread in order to draw user-defined objects. The function can be used to generate any `BodyGraphicsData`, see Section 9.3. Use `graphicsDataUtilities` functions, see Section 5.5, to create more complicated objects. Note that `graphicsDataUserFunction` needs to copy lots of data and is therefore inefficient and only designed to enable simpler tests, but not large scale problems.

For an example for `graphicsDataUserFunction` see `ObjectGround`, [Section 7.2.1](#).

arguments / return	type or size	description
<code>mbs</code>	<code>MainSystem</code>	provides reference to <code>mbs</code> , which can be used in user function to access all data of the object
<code>itemNumber</code>	<code>Index</code>	integer number of the object in <code>mbs</code> , allowing easy access
<code>return value</code>	<code>BodyGraphicsData</code>	list of <code>GraphicsData</code> dictionaries, see Section 9.3

---

### User function example:

```
#user function, using variables M, K, ... from mini example, replacing ObjectGenericODE2
(...)

KD = numpy.diag([200,100])
#nonlinear force example
def UFforce(mbs, t, itemNumber, q, q_t):
    return np.dot(KD, q_t*q) #add nonlinear function for q_t and q, q_t*q gives vector

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

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

#now add object instead of object in mini-example:
oGenericODE2 = mbs.AddObject(ObjectGenericODE2(nodeNumbers=[nMass0,nMass1],
                                                massMatrix=M, stiffnessMatrix=K, dampingMatrix=D,
                                                forceUserFunction=UFforce, massMatrixUserFunction=UFmass,
```

```

    visualization=VObjectGenericODE2(graphicsDataUserFunction=UFgraphics)
)

```

---

### 7.3.1.5 MINI EXAMPLE for ObjectGenericODE2

```

#set up a mechanical system with two nodes; it has the structure: |~~M0~~M1
nMass0 = mbs.AddNode(NodePoint(referenceCoordinates=[0,0,0]))
nMass1 = mbs.AddNode(NodePoint(referenceCoordinates=[1,0,0]))

mass = 0.5 * np.eye(3)      #mass of nodes
stif = 5000 * np.eye(3)     #stiffness of nodes
damp = 50 * np.eye(3)       #damping of nodes
Z = 0. * np.eye(3)          #matrix with zeros
#build mass, stiffness and damping matrices (:
M = np.block([[mass,          0.*np.eye(3)],
              [0.*np.eye(3), mass           ] ] )
K = np.block([[2*stif, -stif],
              [-stif,  stiff] ])
D = np.block([[2*damp, -damp],
              [-damp, damp] ])

oGenericODE2 = mbs.AddObject(ObjectGenericODE2(nodeNumbers=[nMass0,nMass1],
                                                massMatrix=M,
                                                stiffnessMatrix=K,
                                                dampingMatrix=D))

mNode1 = mbs.AddMarker(MarkerNodePosition(nodeNumber=nMass1))
mbs.AddLoad(Force(markerNumber = mNode1, loadVector = [10, 0, 0])) #static solution
=10*(1/5000+1/5000)=0.0004

#assemble and solve system for default parameters
mbs.Assemble()

exu.SolveDynamic(mbs, solverType = exudyn.DynamicSolverType.TrapezoidalIndex2)

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

```

---

For examples on ObjectGenericODE2 see Examples and TestModels:

- [nMassOscillatorInteractive.py](#) (Examples/)
- [simulateInteractively.py](#) (Examples/)
- [coordinateVectorConstraintGenericODE2.py](#) (TestModels/)

- [genericODE2test.py](#) (TestModels/)
- [objectFFRFTest.py](#) (TestModels/)
- [objectGenericODE2Test.py](#) (TestModels/)
- [rigidBodyAsUserFunctionTest.py](#) (TestModels/)
- [solverExplicitODE1ODE2test.py](#) (TestModels/)

### 7.3.2 ObjectFFRF

This object is used to represent equations modelled by the [FFRF](#). It contains a RigidBodyNode (always node 0) and a list of other nodes representing the finite element nodes used in the [FFRF](#). Note that temporary matrices and vectors are subject of change in future.

Authors: Gerstmayr Johannes, Zwölfer Andreas

#### Additional information for ObjectFFRF:

- The Object has the following types = Body, MultiNoded, SuperElement
- Requested node type: read detailed information of item

The item **ObjectFFRF** with type = 'FFRF' has the following parameters:

Name	type	size	default value	description
name	String		"	objects's unique name
nodeNumbers	ArrayNodeIndex		[]	node numbers which provide the coordinates for the object (consecutively as provided in this list); the ( $n_{nf} + 1$ ) nodes represent the nodes of the FE mesh (except for node 0); the global nodal position needs to be reconstructed from the rigid-body motion of the reference frame
massMatrixFF	PyMatrixContainer		PyMatrixContainer[]	body-fixed and ONLY flexible coordinates part of mass matrix of object given in Python numpy format (sparse (CSR) or dense, converted to sparse matrix); internally data is stored in triplet format
stiffnessMatrixFF	PyMatrixContainer		PyMatrixContainer[]	body-fixed and ONLY flexible coordinates part of stiffness matrix of object in Python numpy format (sparse (CSR) or dense, converted to sparse matrix); internally data is stored in triplet format
dampingMatrixFF	PyMatrixContainer		PyMatrixContainer[]	body-fixed and ONLY flexible coordinates part of damping matrix of object in Python numpy format (sparse (CSR) or dense, converted to sparse matrix); internally data is stored in triplet format
forceVector	NumpyVector		[]	generalized, force vector added to RHS; the rigid body part $\mathbf{f}_r$ is directly applied to rigid body coordinates while the flexible part $\mathbf{f}_f$ is transformed from global to local coordinates; note that this force vector only allows to add gravity forces for bodies with <a href="#">COM</a> at the origin of the reference frame

forceUserFunction	PyFunctionVectorMbsScalarIndex2Vector	0	A Python user function which computes the generalized user force vector for the <a href="#">ODE2</a> equations; note the different coordinate systems for rigid body and flexible part; The function args are mbs, time, objectNumber, coordinates q (without reference values) and coordinate velocities q_t; see description below
massMatrixUserFunction	PyFunctionMatrixMbsScalarIndex2Vector	0	A Python user function which computes the TOTAL mass matrix (including reference node) and adds the local constant mass matrix; note the different coordinate systems as described in the <a href="#">FFRF</a> mass matrix; see description below
computeFFRFterms	Bool	True	flag decides whether the standard <a href="#">FFRF</a> terms are computed; use this flag for user-defined definition of <a href="#">FFRF</a> terms in mass matrix and quadratic velocity vector
coordinateIndexPerNode	ArrayIndex	[]	this list contains the local coordinate index for every node, which is needed, e.g., for markers; the list is generated automatically every time parameters have been changed
objectIsInitialized	Bool	False	ALWAYS set to False! flag used to correctly initialize all <a href="#">FFRF</a> matrices; as soon as this flag is False, internal (constant) <a href="#">FFRF</a> matrices are recomputed during Assemble()
physicsMass	UReal	0.	total mass [SI:kg] of <a href="#">FFRF</a> object, auto-computed from mass matrix ${}^b\mathbf{M}$
physicsInertia	Matrix3D	[[1,0,0], [0,1,0], [0,0,1]]	inertia tensor [SI:kgm <sup>2</sup> ] of rigid body w.r.t. to the reference point of the body, auto-computed from the mass matrix ${}^b\mathbf{M}$
physicsCenterOfMass	Vector3D	3 [0.,0.,0.]	local position of center of mass ( <a href="#">COM</a> ); auto-computed from mass matrix ${}^b\mathbf{M}$
PHItTM	NumpyMatrix	Matrix[]	projector matrix; may be removed in future
referencePositions	NumpyVector	[]	vector containing the reference positions of all flexible nodes
tempVector	NumpyVector	[]	temporary vector
tempCoordinates	NumpyVector	[]	temporary vector containing coordinates
tempCoordinates_t	NumpyVector	[]	temporary vector containing velocity coordinates
tempRefPosSkew	NumpyMatrix	Matrix[]	temporary matrix with skew symmetric local (deformed) node positions
tempVelSkew	NumpyMatrix	Matrix[]	temporary matrix with skew symmetric local node velocities
visualization	VObjectFFRF		parameters for visualization of item

The item VObjectFFRF has the following parameters:

<b>Name</b>	<b>type</b>	<b>size</b>	<b>default value</b>	<b>description</b>
show	Bool		True	set true, if item is shown in visualization and false if it is not shown; use visualizationSettings.bodies.deformationScaleFactor to draw scaled (local) deformations; the reference frame node is shown with additional letters RF
color	Float4	4	[-1.,-1.,-1.,-1.]	RGBA color for object; 4th value is alpha-transparency; R=-1.f means, that default color is used
triangleMesh	NumpyMatrixI		MatrixI[]	a matrix, containing node number triples in every row, referring to the node numbers of the GenericODE2 object; the mesh uses the nodes to visualize the underlying object; contour plot colors are still computed in the local frame!
showNodes	Bool		False	set true, nodes are drawn uniquely via the mesh, eventually using the floating reference frame, even in the visualization of the node is show=False; node numbers are shown with indicator 'NF'

### 7.3.2.1 DESCRIPTION of ObjectFFRF:

Information on input parameters:

<b>input parameter</b>	<b>symbol</b>	<b>description see tables above</b>
nodeNumbers	$\mathbf{n}_f = [n_0, \dots, n_{n_f}]^T$	
massMatrixFF	${}^b\mathbf{M} \in \mathbb{R}^{n_f \times n_f}$	
stiffnessMatrixFF	${}^b\mathbf{K} \in \mathbb{R}^{n_f \times n_f}$	
dampingMatrixFF	${}^b\mathbf{D} \in \mathbb{R}^{n_f \times n_f}$	
forceVector	${}^0\mathbf{f} = [{}^0\mathbf{f}_r, {}^0\mathbf{f}_f]^T \in \mathbb{R}^{n_c}$	
forceUserFunction	$\mathbf{f}_{user} = [{}^0\mathbf{f}_{r,user}, {}^b\mathbf{f}_{f,user}]^T \in \mathbb{R}^{n_c}$	
massMatrixUserFunction	$\mathbf{M}_{user} \in \mathbb{R}^{n_c \times n_c}$	
physicsMass	$m$	
physicsInertia	$J_r \in \mathbb{R}^{3 \times 3}$	
physicsCenterOfMass	${}^b\mathbf{b}_{COM}$	
PHITM	$\Phi_t^T \in \mathbb{R}^{n_f \times 3}$	
referencePositions	$\mathbf{x}_{ref} \in \mathbb{R}^{n_f}$	
tempVector	$\mathbf{v}_{temp} \in \mathbb{R}^{n_f}$	
tempCoordinates	$\mathbf{c}_{temp} \in \mathbb{R}^{n_f}$	
tempCoordinates_t	$\dot{\mathbf{c}}_{temp} \in \mathbb{R}^{n_f}$	

tempRefPosSkew	$\tilde{\mathbf{p}}_f \in \mathbb{R}^{n_f \times 3}$	
tempVelSkew	$\dot{\tilde{\mathbf{c}}}_f \in \mathbb{R}^{n_f \times 3}$	

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

output variable	symbol	description
Coordinates		all <code>ODE2</code> coordinates
Coordinates_t		all <code>ODE2</code> velocity coordinates
Coordinates_tt		all <code>ODE2</code> acceleration coordinates
Force		generalized forces for all coordinates (residual of all forces except mass*acceleration; corresponds to <code>ComputeODE2LHS</code> )

### 7.3.2.2 Additional output variables for superelement node access

Functions like `GetObjectOutputSuperElement(...)`, see [Section 4.4.2](#), or `SensorSuperElement`, see [Section 4.4.5](#), directly access special output variables (`OutputVariableType`) of the mesh nodes  $n_i$  of the superelement. Additionally, the contour drawing of the object can make use the `OutputVariableType` of the meshnodes.

### 7.3.2.3 Super element output variables

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

### 7.3.2.4 Definition of quantities

intermediate variables	symbol	description
object coordinates	$\mathbf{q} = [\mathbf{q}_t^T, \mathbf{q}_r^T, \mathbf{q}_f^T]^T$	object coordinates
rigid body coordinates	$\mathbf{q}_{\text{rigid}} = [\mathbf{q}_t^T, \mathbf{q}_r^T]^T = [q_0, q_1, q_2, \psi_0, \psi_1, \psi_2, \psi_3]^T$	rigid body coordinates in case of Euler parameters

reference frame (rigid body) position	${}^0\mathbf{r}_{\text{config}} = {}^0\mathbf{q}_{t,\text{config}} + {}^0\mathbf{q}_{t,\text{ref}}$	global position of underlying rigid body node $n_0$ which defines the reference frame origin
reference frame (rigid body) orientation	${}^{0b}\mathbf{A}(\theta)_{\text{config}}$	transformation matrix for transformation of local (reference frame) to global coordinates, given by underlying rigid body node $n_0$
local nodal position	${}^b\mathbf{p}^{(i)} = {}^b\mathbf{x}_{\text{ref}}^{(i)} + {}^b\mathbf{q}_f^{(i)}$	vector of body-fixed (local) position of node ( $i$ ), including flexible part
local nodal positions	${}^b\mathbf{p} = {}^b\mathbf{x}_{\text{ref}} + {}^b\mathbf{q}_f$	vector of all body-fixed (local) nodal positions including flexible part
rotation coordinates	$\theta_{\text{cur}} = [\psi_0, \psi_1, \psi_2, \psi_3]_{\text{ref}}^T + [\psi_0, \psi_1, \psi_2, \psi_3]_{\text{cur}}^T$	rigid body coordinates in case of Euler parameters
flexible coordinates	${}^b\mathbf{q}_f$	flexible, body-fixed coordinates
transformation of flexible coordinates	${}^{0b}\mathbf{A}_{bd} = \text{diag}([{}^{0b}\mathbf{A}, \dots, {}^{0b}\mathbf{A}])$	block diagonal transformation matrix, which transforms all flexible coordinates from local to global coordinates

The derivations follow Zwölfer and Gerstmayr [43] with only small modifications in the notation.

### 7.3.2.5 Nodal coordinates

Consider an object with  $n = 1 + n_{\text{nf}}$  nodes,  $n_{\text{nf}}$  being the number of 'flexible' nodes and one additional node is the rigid body node for the reference frame. The list of node numbers is  $[n_0, \dots, n_{n_{\text{nf}}}]$  and the according numbers of nodal coordinates are  $[n_{c_0}, \dots, n_{c_n}]$ , where  $n_0$  denotes the rigid body node. This gives  $n_c$  total nodal coordinates,

$$n_c = \sum_{i=0}^{n_{\text{nf}}} n_{c_i}, \quad (7.54)$$

whereof the number of flexible coordinates is

$$n_f = 3 \cdot n_{\text{nf}}. \quad (7.55)$$

The total number of equations (=coordinates) of the object is  $n_c$ . The first node  $n_0$  represents the rigid body motion of the underlying reference frame with  $n_{c_r} = n_{c_0}$  coordinates<sup>1</sup>.

### 7.3.2.6 Kinematics

We assume a finite element mesh with The kinematics of the FFRF is based on a splitting of translational ( $\mathbf{c}_t \in \mathbb{R}^{n_t}$ ), rotational ( $\mathbf{c}_r \in \mathbb{R}^{n_r}$ ) and flexible ( $\mathbf{c}_f \in \mathbb{R}^{n_f}$ ) nodal displacements,

$${}^0\mathbf{c} = {}^0\mathbf{c}_t + {}^0\mathbf{c}_r + {}^0\mathbf{c}_f. \quad (7.56)$$

which are written in global coordinates in Eq. (7.56) but will be transformed to other coordinates later on.

In the present formulation of ObjectFFRF, we use the following set of object coordinates (unknowns)

$$\mathbf{q} = \begin{bmatrix} {}^0\mathbf{q}_t^T & \boldsymbol{\theta}^T & {}^b\mathbf{q}_f^T \end{bmatrix}^T \in \mathbb{R}^{n_c} \quad (7.57)$$

<sup>1</sup>e.g.,  $n_{c_r} = 6$  coordinates for Euler angles and  $n_{c_r} = 7$  coordinates in case of Euler parameters; currently only the Euler parameter case is implemented.

with  ${}^0\mathbf{q}_t \in \mathbb{R}^3$ ,  $\boldsymbol{\theta} \in \mathbb{R}^4$  and  ${}^b\mathbf{q}_f \in \mathbb{R}^{n_f}$ . Note that parts of the coordinates  $\mathbf{q}$  can be already interpreted in specific coordinate systems, which is therefore added.

With the relations

$$\boldsymbol{\Phi}_t = [\mathbf{I}_{3 \times 3}, \dots, \mathbf{I}_{3 \times 3}]^T \in \mathbb{R}^{n_f \times 3}, \quad (7.58)$$

$${}^0\mathbf{c}_t = \boldsymbol{\Phi}_t {}^0\mathbf{q}_t, \quad (7.59)$$

$${}^0\mathbf{c}_r = ({}^{0b}\mathbf{A}_{bd} - \mathbf{I}_{bd}) {}^b\mathbf{x}_{ref}, \quad (7.60)$$

$${}^0\mathbf{c}_f = {}^{0b}\mathbf{A}_{bd} {}^b\mathbf{q}_f, \text{ and} \quad (7.61)$$

$$\mathbf{I}_{bd} = \text{diag}(\mathbf{I}_{3 \times 3}, \dots, \mathbf{I}_{3 \times 3}) \in \mathbb{R}^{n_f \times n_f}, \quad (7.62)$$

we obtain the total relation of (global) nodal displacements to the object coordinates

$${}^0\mathbf{c} = \boldsymbol{\Phi}_t {}^0\mathbf{q}_t + ({}^{0b}\mathbf{A}_{bd} - \mathbf{I}_{bd}) {}^b\mathbf{x}_{ref} + {}^{0b}\mathbf{A}_{bd} {}^b\mathbf{q}_f. \quad (7.63)$$

On velocity level, we have

$${}^0\dot{\mathbf{c}} = \mathbf{L}\dot{\mathbf{q}}, \quad (7.64)$$

with the matrix  $\mathbf{L} \in \mathbb{R}^{n_f \times n_c}$

$$\mathbf{L} = [\boldsymbol{\Phi}_t, -{}^{0b}\mathbf{A}_{bd} {}^b\tilde{\mathbf{p}} {}^b\mathbf{G}, {}^{0b}\mathbf{A}_{bd}] \quad (7.65)$$

with the rotation parameters specific matrix  ${}^b\mathbf{G}$ , implicitly defined in the rigid body node by the relation  ${}^b\boldsymbol{\omega} = {}^b\mathbf{G} \dot{\boldsymbol{\theta}}$  and the body-fixed nodal position vector (for node  $i$ )

$${}^b\mathbf{p} = {}^b\mathbf{x}_{ref} + {}^b\mathbf{q}_f, \quad {}^b\mathbf{p}^{(i)} = {}^b\mathbf{x}_{ref}^{(i)} + {}^b\mathbf{q}_{f,i}^{(i)} \quad (7.66)$$

and the special tilde matrix for vectors  $\mathbf{p} \in \mathbb{R}^{3n_f}$ ,

$${}^b\tilde{\mathbf{p}} = \begin{bmatrix} {}^b\tilde{\mathbf{p}}^{(i)} \\ \vdots \\ {}^b\tilde{\mathbf{p}}^{(i)} \end{bmatrix} \in \mathbb{R}^{3n_f \times 3}. \quad (7.67)$$

with the tilde operator for a  $\mathbf{p}^{(i)} \in \mathbb{R}^3$  defined in the common notations section.

### 7.3.2.7 Equations of motion

We use the Lagrange equations extended for constraint  $\mathbf{g}$ ,

$$\frac{d}{dt} \left( \frac{\partial T}{\partial \dot{\mathbf{q}}^T} \right) - \frac{\partial T}{\partial \mathbf{q}} + \frac{\partial V}{\partial \mathbf{q}} + \frac{\partial \lambda^T \mathbf{g}}{\partial \mathbf{q}} = \frac{\partial W}{\partial \mathbf{q}} \quad (7.68)$$

with the quantities

$$T({}^0\dot{\mathbf{c}}(\mathbf{q}, \dot{\mathbf{q}})) = \frac{1}{2} {}^0\dot{\mathbf{c}}^T {}^0\mathbf{M} {}^0\dot{\mathbf{c}} = \frac{1}{2} {}^0\dot{\mathbf{c}}^T {}^{0b}\mathbf{A}_{bd} {}^b\mathbf{M} {}^{0b}\mathbf{A}_{bd} {}^T {}^0\dot{\mathbf{c}} = \frac{1}{2} {}^0\dot{\mathbf{c}}^T {}^b\mathbf{M} {}^0\dot{\mathbf{c}} \quad (7.69)$$

$$V({}^0\mathbf{q}_f) = \frac{1}{2} {}^b\mathbf{q}_f^T {}^b\mathbf{K} {}^b\mathbf{q}_f \quad (7.70)$$

$$\delta W({}^0\mathbf{c}(\mathbf{q}), t) = {}^b\delta\mathbf{c}^T \mathbf{f} \quad (7.71)$$

$$\mathbf{g}(\mathbf{q}, t) = \mathbf{0} \quad (7.72)$$

$$(7.73)$$

Note that  ${}^b\mathbf{M}$  and  ${}^b\mathbf{K}$  are the conventional finite element mass and stiffness matrices defined in the body frame.

Elementary differentiation rules of the Lagrange equations lead to

$$\mathbf{L}^T \mathbf{M} \mathbf{L} \ddot{\mathbf{q}} + \mathbf{L}^T \mathbf{M} \dot{\mathbf{L}} \dot{\mathbf{q}} + \hat{\mathbf{K}} \mathbf{q} + \frac{\partial \mathbf{g}}{\partial \mathbf{q}^T} \lambda = \mathbf{L}^T \mathbf{f} \quad (7.74)$$

with  $\mathbf{M} = {}^b\mathbf{M}$  and  $\hat{\mathbf{K}}$  becoming obvious in Eq. (7.75). Note that Eq. (7.74) is given in global coordinates for the translational part, in terms of rotation parameters for the rotation part and in body-fixed coordinates for the flexible part of the equations.

In case that `computeFFRFterms = True`, the equations 7.74 can be transformed into the equations of motion,

$$\left( \mathbf{M}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}}) + \begin{bmatrix} \mathbf{M}_{tt} & \mathbf{M}_{tr} & \mathbf{M}_{tf} \\ & \mathbf{M}_{rr} & \mathbf{M}_{rf} \\ & \text{sym.} & {}^b\mathbf{M} \end{bmatrix} \right) \dot{\mathbf{q}} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & {}^b\mathbf{D} \end{bmatrix} \dot{\mathbf{q}} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & {}^b\mathbf{K} \end{bmatrix} \mathbf{q} = \mathbf{f}_v(\mathbf{q}, \dot{\mathbf{q}}) + \begin{bmatrix} \mathbf{f}_r \\ {}^{0b}\mathbf{A}_{bd}^T \mathbf{f}_f \end{bmatrix} + \mathbf{f}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}}) \quad (7.75)$$

in which `iN` represents the itemNumber (=objectNumber of ObjectFFRF in mbs) in the user function. The mass terms are given as

$$\mathbf{M}_{tt} = \Phi_t^T {}^b\mathbf{M} \Phi_t, \quad (7.76)$$

$$\mathbf{M}_{tr} = -{}^{0b}\mathbf{A} \Phi_t^T {}^b\mathbf{M} {}^b\tilde{\mathbf{p}} {}^b\mathbf{G}, \quad (7.77)$$

$$\mathbf{M}_{tf} = {}^{0b}\mathbf{A} \Phi_t^T {}^b\mathbf{M}, \quad (7.78)$$

$$\mathbf{M}_{rr} = {}^b\mathbf{G}^T {}^b\tilde{\mathbf{p}}^T {}^b\mathbf{M} {}^b\tilde{\mathbf{p}} {}^b\mathbf{G}, \quad (7.79)$$

$$\mathbf{M}_{rf} = -{}^b\mathbf{G}^T {}^b\tilde{\mathbf{p}}^T {}^b\mathbf{M}. \quad (7.80)$$

In case that `computeFFRFterms = False`, the mass terms  $\mathbf{M}_{tt}, \mathbf{M}_{tr}, \mathbf{M}_{tf}, \mathbf{M}_{rr}, \mathbf{M}_{rf}, {}^b\mathbf{M}$  in Eq. (7.75) are set to zero (and not computed) and the quadratic velocity vector  $\mathbf{f}_v = \mathbf{0}$ . Note that the user functions  $\mathbf{f}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}})$  and  $\mathbf{M}_{user}(mbs, t, i_N, \mathbf{q}, \dot{\mathbf{q}})$  may be empty ( $=0$ ). The detailed equations of motion for this element can be found in [42].

The quadratic velocity vector follows as

$$\mathbf{f}_v(\mathbf{q}, \dot{\mathbf{q}}) = \begin{bmatrix} -{}^{0b}\mathbf{A} \Phi_t^T {}^b\mathbf{M} \left( {}^b\tilde{\omega}_{bd} {}^b\tilde{\omega}_{bd} {}^b\mathbf{p} + 2 {}^b\tilde{\omega}_{bd} {}^b\dot{\mathbf{q}}_f - {}^b\tilde{\mathbf{p}} {}^b\dot{\mathbf{G}} \theta \right) \\ {}^b\mathbf{G}^T {}^b\tilde{\mathbf{p}}^T {}^b\mathbf{M} \left( {}^b\tilde{\omega}_{bd} {}^b\tilde{\omega}_{bd} {}^b\mathbf{p} + 2 {}^b\tilde{\omega}_{bd} {}^b\dot{\mathbf{q}}_f - {}^b\tilde{\mathbf{p}} {}^b\dot{\mathbf{G}} \theta \right) \\ -{}^b\mathbf{M} \left( {}^b\tilde{\omega}_{bd} {}^b\tilde{\omega}_{bd} {}^b\mathbf{p} + 2 {}^b\tilde{\omega}_{bd} {}^b\dot{\mathbf{q}}_f - {}^b\tilde{\mathbf{p}} {}^b\dot{\mathbf{G}} \theta \right) \end{bmatrix} \quad (7.81)$$

with the special matrix

$${}^b\tilde{\omega}_{bd} = \text{diag}({}^b\tilde{\omega}_{bd}, \dots, {}^b\tilde{\omega}_{bd}) \in \mathbb{R}^{n_f \times n_f} \quad (7.82)$$

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

If the rigid body node is using Euler parameters  $\theta = [\theta_0, \theta_1, \theta_2, \theta_3]^T$ , an **additional constraint** (constraint nr. 0) is added automatically for the Euler parameter norm, reading

$$1 - \sum_{i=0}^3 \theta_i^2 = 0. \quad (7.83)$$

In order to suppress the rigid body motion of the mesh nodes, you should apply a `ObjectConnectorCoordinateVector` object with the following constraint equations which impose constraints of a so-called Tisserand frame, giving 3 constraints for the position of the center of mass

$$\Phi_t^T {}^b\mathbf{M} \mathbf{q}_f = 0 \quad (7.84)$$

and 3 constraints for the rotation,

$$\tilde{\mathbf{x}}_f^T {}^b\mathbf{M} \mathbf{q}_f = 0 \quad (7.85)$$

### Userfunction: `forceUserFunction(mbs, t, itemNumber, q, q_t)`

A user function, which computes a force vector depending on current time and states of object. Can be used to create any kind of mechanical system by using the object states.

arguments / return	type or size	description
mbs	MainSystem	provides MainSystem mbs to which object belongs
t	Real	current time in mbs
itemNumber	Index	integer number of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
q	Vector $\in \mathbb{R}_c^n$	object coordinates (nodal displacement coordinates of rigid body and mesh nodes) in current configuration, without reference values
q_t	Vector $\in \mathbb{R}_c^n$	object velocity coordinates (time derivative of q) in current configuration
<b>return value</b>	Vector $\in \mathbb{R}^{n_c}$	returns force vector for object

---

### Userfunction: `massMatrixUserFunction(mbs, t, itemNumber, q, q_t)`

A user function, which computes a mass matrix depending on current time and states of object. Can be used to create any kind of mechanical system by using the object states.

arguments / return	type or size	description
mbs	MainSystem	provides MainSystem mbs to which object belongs
t	Real	current time in mbs
itemNumber	Index	integer number of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
q	Vector $\in \mathbb{R}_c^n$	object coordinates (nodal displacement coordinates of rigid body and mesh nodes) in current configuration, without reference values
q_t	Vector $\in \mathbb{R}_c^n$	object velocity coordinates (time derivative of q) in current configuration
<b>return value</b>	NumpyMatrix $\in \mathbb{R}^{n_c \times n_c}$	returns mass matrix for object

---

For examples on ObjectFFRF see Examples and TestModels:

- [objectFFRFTTest.py](#) (TestModels/)
- [objectFFRFTTest2.py](#) (TestModels/)

### 7.3.3 ObjectFFRFreducedOrder

This object is used to represent modally reduced flexible bodies using the [FFRF](#) and the [CMS](#). It can be used to model real-life mechanical systems imported from finite element codes or Python tools such as NETGEN/NGsolve, see the FEMinterface in [Section 5.4.8](#). It contains a RigidBodyNode (always node 0) and a NodeGenericODE2 representing the modal coordinates. Currently, equations must be defined within user functions, which are available in the FEM module, see class `ObjectFFRFreducedOrderInterface`, especially the user functions `UFmassFFRFreducedOrder` and `UFforceFFRFreducedOrder`, [Section 5.4.6](#).

Authors: Gerstmayr Johannes, Zwölfer Andreas

#### Additional information for ObjectFFRFreducedOrder:

- The Object has the following types = Body, MultiNoded, SuperElement
- Requested node type: read detailed information of item
- **Short name for Python** = **CMSobject**
- **Short name for Python (visualization object)** = **VCMSSobject**

The item **ObjectFFRFreducedOrder** with type = 'FFRFreducedOrder' has the following parameters:

Name	type	size	default value	description
name	String		"	objects's unique name
nodeNumbers	ArrayNodeIndex		[]	node numbers of rigid body node and NodeGenericODE2 for modal coordinates; the global nodal position needs to be reconstructed from the rigid-body motion of the reference frame, the modal coordinates and the mode basis
massMatrixReduced	PyMatrixContainer		PyMatrixContainer[]	body-fixed and ONLY flexible coordinates part of reduced mass matrix; provided as MatrixContainer(sparse/dense matrix)
stiffnessMatrixReduced	PyMatrixContainer		PyMatrixContainer[]	body-fixed and ONLY flexible coordinates part of reduced stiffness matrix; provided as MatrixContainer(sparse/dense matrix)
dampingMatrixReduced	PyMatrixContainer		PyMatrixContainer[]	body-fixed and ONLY flexible coordinates part of reduced damping matrix; provided as MatrixContainer(sparse/dense matrix)
forceUserFunction	PyFunctionVectorMbsScalarIndex2Vector		0	A Python user function which computes the generalized user force vector for the <a href="#">ODE2</a> equations; see description below
massMatrixUserFunction	PyFunctionMatrixMbsScalarIndex2Vector		0	A Python user function which computes the TOTAL mass matrix (including reference node) and adds the local constant mass matrix; see description below

computeFFRFterms	Bool	True	flag decides whether the standard <b>FFRF/CMS</b> terms are computed; use this flag for user-defined definition of <b>FFRF</b> terms in mass matrix and quadratic velocity vector
modeBasis	NumpyMatrix	Matrix[]	mode basis, which transforms reduced coordinates to (full) nodal coordinates, written as a single vector $[u_{x,n_0}, u_{y,n_0}, u_{z,n_0}, \dots, u_{x,n_n}, u_{y,n_n}, u_{z,n_n}]^T$
outputVariableModeBasis	NumpyMatrix	Matrix[]	mode basis, which transforms reduced coordinates to output variables per mode and per node; $s_{OV}$ is the size of the output variable, e.g., 6 for stress modes ( $S_{xx}, \dots, S_{xy}$ )
outputVariableTypeModeBasis	OutputVariableType	OutputVariableType::None	this must be the output variable type of the outputVariableModeBasis, e.g. exu.OutputVariableType.Stress
referencePositions	NumpyVector	[]	vector containing the reference positions of all flexible nodes, needed for graphics
objectIsInitialized	Bool	False	ALWAYS set to False! flag used to correctly initialize all <b>FFRF</b> matrices; as soon as this flag is False, some internal (constant) <b>FFRF</b> matrices are recomputed during Assemble()
physicsMass	UReal	0.	total mass [SI:kg] of FFRReducedOrder object
physicsInertia	Matrix3D	$[[1,0,0], [0,1,0], [0,0,1]]$	inertia tensor [SI:kgm <sup>2</sup> ] of rigid body w.r.t. to the reference point of the body
physicsCenterOfMass	Vector3D	3 [0.,0.,0.]	local position of center of mass ( <b>COM</b> )
mPsiTildePsi	NumpyMatrix	Matrix[]	special FFRReducedOrder matrix, computed in ObjectFFRRReducedOrderInterface
mPsiTildePsiTilde	NumpyMatrix	Matrix[]	special FFRReducedOrder matrix, computed in ObjectFFRRReducedOrderInterface
mPhitTPsi	NumpyMatrix	Matrix[]	special FFRReducedOrder matrix, computed in ObjectFFRRReducedOrderInterface
mPhitTPsiTilde	NumpyMatrix	Matrix[]	special FFRReducedOrder matrix, computed in ObjectFFRRReducedOrderInterface
mXRefTildePsi	NumpyMatrix	Matrix[]	special FFRReducedOrder matrix, computed in ObjectFFRRReducedOrderInterface
mXRefTildePsiTilde	NumpyMatrix	Matrix[]	special FFRReducedOrder matrix, computed in ObjectFFRRReducedOrderInterface

physicsCenterOfMassTilde	Matrix3D		[[0,0,0], [0,0,0], [0,0,0]]	tilde matrix from local position of <a href="#">COM</a> ; autocomputed during initialization
tempUserFunctionForce	NumpyVector		[]	temporary vector for UF force
visualization	VObjectFFRFreducedOrder			parameters for visualization of item

The item VObjectFFRFreducedOrder has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown; use visualizationSettings.bodies.deformationScaleFactor to draw scaled (local) deformations; the reference frame node is shown with additional letters RF
color	Float4	4	[-1.,-1.,-1.,-1.]	RGBA color for object; 4th value is alpha-transparency; R=-1.f means, that default color is used
triangleMesh	NumpyMatrixI		MatrixI[]	a matrix, containing node number triples in every row, referring to the node numbers of the GenericODE2 object; the mesh uses the nodes to visualize the underlying object; contour plot colors are still computed in the local frame!
showNodes	Bool		False	set true, nodes are drawn uniquely via the mesh, eventually using the floating reference frame, even in the visualization of the node is show=False; node numbers are shown with indicator 'NF'

### 7.3.3.1 DESCRIPTION of ObjectFFRFreducedOrder:

#### Information on input parameters:

input parameter	symbol	description see tables above
nodeNumbers	$\mathbf{n} = [n_0, n_1]^T$	
massMatrixReduced	$\mathbf{M}_{\text{red}} \in \mathbb{R}^{n_m \times n_m}$	
stiffnessMatrixReduced	$\mathbf{K}_{\text{red}} \in \mathbb{R}^{n_m \times n_m}$	
dampingMatrixReduced	$\mathbf{D}_{\text{red}} \in \mathbb{R}^{n_m \times n_m}$	
forceUserFunction	$\mathbf{f}_{\text{user}} \in \mathbb{R}^{n_{\text{ODE2}}}$	
massMatrixUserFunction	$\mathbf{M}_{\text{user}} \in \mathbb{R}^{n_{\text{ODE2}} \times n_{\text{ODE2}}}$	
modeBasis	${}^b\Psi \in \mathbb{R}^{n_t \times n_m}$	
outputVariableModeBasis	${}^b\Psi_{OV} \in \mathbb{R}^{n_n \times (n_m \cdot s_{OV})}$	

referencePositions	${}^b\mathbf{x}_{ref} \in \mathbb{R}^{n_f}$	
physicsMass	$m$	
physicsInertia	$\mathbf{J}_r \in \mathbb{R}^{3 \times 3}$	
physicsCenterOfMass	${}^b\mathbf{b}_{COM}$	
physicsCenterOfMassTilde	${}^b\tilde{\mathbf{b}}_{COM}$	
tempUserFunctionForce	$\mathbf{f}_{temp} \in \mathbb{R}^{n_{ODE2}}$	

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

output variable	symbol	description
Coordinates		all ODE2 coordinates
Coordinates_t		all ODE2 velocity coordinates
Force		generalized forces for all coordinates (residual of all forces except mass*acceleration; corresponds to ComputeODE2LHS)

### 7.3.3.2 Super element output variables

Functions like GetObjectOutputSuperElement(...), see [Section 4.4.2](#), or SensorSuperElement, see [Section 4.4.5](#), directly access special output variables (OutputVariableType) of the mesh nodes of the superelement. Additionally, the contour drawing of the object can make use the OutputVariableType of the meshnodes.

super element output variables	symbol	description
DisplacementLocal (mesh node $i$ )	${}^b\mathbf{u}_f^{(i)} = ({}^b\Psi\zeta)_{3:i...3:i+2} = \begin{bmatrix} {}^b\mathbf{q}_{f,i:3} \\ {}^b\mathbf{q}_{f,i:3+1} \\ {}^b\mathbf{q}_{f,i:3+2} \end{bmatrix}$	local nodal mesh displacement in reference (body) frame, measuring only flexible part of displacement
VelocityLocal (mesh node ( $i$ ))	${}^b\dot{\mathbf{u}}_f^{(i)} = ({}^b\Psi\zeta)_{3:i...3:i+2}$	local nodal mesh velocity in reference (body) frame, only for flexible part of displacement
Displacement (mesh node ( $i$ ))	${}^0\mathbf{u}_{config}^{(i)} = {}^0\mathbf{q}_{t,config} + {}^{0b}\mathbf{A}_{config} {}^b\mathbf{P}_{f,config}^{(i)} - ({}^0\mathbf{q}_{t,ref} + {}^{0b}\mathbf{A}_{ref} {}^b\mathbf{x}_{ref}^{(i)})$	nodal mesh displacement in global coordinates
Position (mesh node ( $i$ ))	${}^0\mathbf{p}^{(i)} = {}^0\mathbf{r} + {}^{0b}\mathbf{A} {}^b\mathbf{p}_f^{(i)}$	nodal mesh position in global coordinates
Velocity (mesh node ( $i$ ))	${}^0\dot{\mathbf{u}}^{(i)} = {}^0\dot{\mathbf{q}}_t + {}^{0b}\mathbf{A} ({}^b\dot{\mathbf{u}}_f^{(i)} + {}^b\tilde{\omega} {}^b\mathbf{p}_f^{(i)})$	nodal mesh velocity in global coordinates
Acceleration (mesh node ( $i$ ))	${}^0\mathbf{a}^{(i)} = {}^0\ddot{\mathbf{q}}_t + {}^{0b}\mathbf{A} {}^b\ddot{\mathbf{u}}_f^{(i)} + 2 {}^0\omega \times {}^{0b}\mathbf{A} {}^b\dot{\mathbf{u}}_f^{(i)} + {}^0\alpha \times {}^0\mathbf{p}_f^{(i)} + {}^0\omega \times ({}^0\omega \times {}^0\mathbf{p}_f^{(i)})$	global acceleration of mesh node $n_i$ including rigid body motion and flexible deformation; note that ${}^0\mathbf{x}(n_i) = {}^{0b}\mathbf{A} {}^b\mathbf{x}(n_i)$
StressLocal (mesh node ( $i$ ))	${}^b\sigma^{(i)} = ({}^b\Psi_{OV}\zeta)_{3:i...3:i+5}$	linearized stress components of mesh node ( $i$ ) in reference frame; $\sigma = [\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{yz}, \sigma_{xz}, \sigma_{xy}]^T$ ; ONLY available, if ${}^b\Psi_{OV}$ is provided and outputVariableTypeModeBasis==exu.OutputVariableType.StressLocal

StrainLocal (mesh node ( $i$ ))	${}^b\boldsymbol{\varepsilon}^{(i)} = ({}^b\Psi_{OV}\zeta)_{3:i \dots 3:i+5}$	linearized strain components of mesh node ( $i$ ) in reference frame; $\boldsymbol{\varepsilon} = [\varepsilon_{xx}, \varepsilon_{yy}, \varepsilon_{zz}, \varepsilon_{yz}, \varepsilon_{xz}, \varepsilon_{xy}]^T$ ; ONLY available, if ${}^b\Psi_{OV}$ is provided and <code>outputVariableTypeModeBasis==exu.OutputVariableType.StrainLocal</code>
---------------------------------	---	---

intermediate variables	symbol	description
reference frame	$b$	the body-fixed / local frame is always denoted by $b$
number of rigid body coordinates	$n_{rigid}$	number of rigid body node coordinates: 6 in case of Euler angles (not fully available for ObjectFFRFreducedOrder) and 7 in case of Euler parameters
number of flexible / mesh coordinates	$n_f = 3 \cdot n_n$	with number of nodes $n_n$ ; relevant for visualization
number of modal coordinates	$n_m \ll n_f$	the number of reduced or modal coordinates, computed from number of columns given in <code>modeBasis</code>
total number object coordinates	$n_{ODE2} = n_m + n_{rigid}$	
reference frame origin	${}^0\mathbf{r} = {}^0\mathbf{q}_t + {}^0\mathbf{q}_{t,ref}$	reference frame position (origin)
reference frame rotation	$\theta_{config} = \theta_{config} + \theta_{ref}$	reference frame rotation parameters in any configuration except reference
reference frame orientation	${}^{0b}\mathbf{A}_{config} = {}^{0b}\mathbf{A}_{config}(\theta_{config})$	transformation matrix for transformation of local (reference frame) to global coordinates, given by underlying rigid body node $n_0$
local vector of flexible coordinates	${}^b\mathbf{q}_f = {}^b\Psi\zeta$	represents mesh displacements; vector of alternating $x, y$ , and $z$ coordinates of local (in body frame) mesh displacements reconstructed from modal coordinates $\zeta$ ; only evaluated for selected node points (e.g., sensors) during computation; corresponds to same vector in <code>ObjectFFRF</code>
local nodal positions	${}^b\mathbf{p}_f = {}^b\mathbf{q}_f + {}^b\mathbf{x}_{ref}$	vector of all body-fixed nodal positions including flexible part; only evaluated for selected node points during computation
local position of node ( $i$ )	${}^b\mathbf{p}_f^{(i)} = {}^b\mathbf{u}_f^{(i)} + {}^b\mathbf{x}_{ref}^{(i)} = \begin{bmatrix} {}^b\mathbf{q}_{f,i:3} \\ {}^b\mathbf{q}_{f,i:3+1} \\ {}^b\mathbf{q}_{f,i:3+2} \end{bmatrix} + \begin{bmatrix} {}^b\mathbf{x}_{ref,i:3} \\ {}^b\mathbf{x}_{ref,i:3+1} \\ {}^b\mathbf{x}_{ref,i:3+2} \end{bmatrix}$	body-fixed, deformed nodal mesh position (including flexible part)
vector of modal coordinates	$\zeta = [\zeta_0, \dots, \zeta_{n_m-1}]^T$	vector of modal or reduced coordinates; these coordinates can either represent amplitudes of eigenmodes, static modes or general modes, depending on your mode basis

coordinate vector	$\mathbf{q} = [{}^0\mathbf{q}_t, \psi, \zeta]$	vector of object coordinates; $\mathbf{q}_t$ and $\psi$ are the translation and rotation part of displacements of the reference frame, provided by the rigid body node (node number 0)
flexible coordinates transformation matrix	${}^{0b}\mathbf{A}_{bd} = \text{diag}([{}^{0b}\mathbf{A}, \dots, {}^{0b}\mathbf{A}])$	block diagonal transformation matrix, which transforms all flexible coordinates from local to global coordinates

### 7.3.3.3 Modal reduction and reduced inertia matrices

The formulation is based on the EOM of ObjectFFRF, also regarding parts of notation and some input parameters, [Section 7.3.2](#), and can be found in Zwölfer and Gerstmayr [43] with only small modifications in the notation. The notation of kinematics quantities follows the floating frame of reference idea with quantities given in the tables above and sketched in Fig. 7.2.

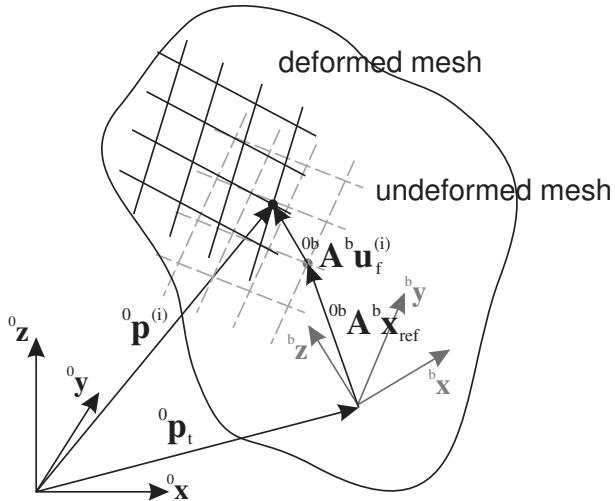


Figure 7.2: Floating frame of reference with exemplary position of a mesh node  $i$ .

The reduced order FFRF formulation is based on an approximation of flexible coordinates  ${}^b\mathbf{q}_f$  by means of a reduction or mode basis  ${}^b\Psi$  (modeBasis) and the the modal coordinates  $\zeta$ ,

$${}^b\mathbf{q}_f \approx {}^b\Psi \zeta \quad (7.86)$$

The mode basis  ${}^b\Psi$  contains so-called mode shape vectors in its columns, which may be computed from eigen analysis, static computation or more advanced techniques, see the helper functions in module exudyn.FEM, within the class FEMinterface. To compute eigen modes, use `FEMinterface.ComputeEigenmodes(...)` or `FEMinterface.ComputeHurtyCraigBamptonModes(...)`. For details on model order reduction and component mode synthesis, see [Section 6.2](#). In many applications,  $n_m$  typically ranges between 10 and 50, but also beyond – depending on the desired accuracy of the model.

The ObjectFFRF coordinates and Eqs. (7.75)<sup>2</sup> can be reduced by the matrix  $\mathbf{H} \in \mathbb{R}^{(n_t+n_{\text{rigid}}) \times n_{\text{ODE2}}}$ ,

$$\mathbf{q}_{FFRF} = \begin{bmatrix} \mathbf{q}_t \\ \boldsymbol{\theta} \\ {}^b\mathbf{q}_f \end{bmatrix} = \begin{bmatrix} \mathbf{I}_{3 \times 3} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_r & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & {}^b\Psi \end{bmatrix} \begin{bmatrix} \mathbf{q}_t \\ \boldsymbol{\theta} \\ \zeta \end{bmatrix} = \mathbf{H} \mathbf{q} \quad (7.87)$$

<sup>2</sup>this is not done for user functions and forceVector

with the  $4 \times 4$  identity matrix  $\mathbf{I}_r$  in case of Euler parameters and the reduced coordinates  $\mathbf{q}$ .

The reduced equations follow from the reduction of system matrices in Eqs. (7.75),

$$\mathbf{K}_{\text{red}} = {}^b\Psi^T {}^b\mathbf{K} {}^b\Psi, \quad (7.88)$$

$$\mathbf{M}_{\text{red}} = {}^b\Psi^T {}^b\mathbf{M} {}^b\Psi, \quad (7.89)$$

$$(7.90)$$

the computation of rigid body inertia

$${}^b\Theta_u = {}^b\tilde{x}_{\text{ref}}^T {}^b\mathbf{M} {}^b\tilde{x}_{\text{ref}} \quad (7.91)$$

$$(7.92)$$

the center of mass (and according tilde matrix), using  $\Phi_t$  from Eq. (7.58),

$${}^b\chi_u = \frac{1}{m} \Phi_t^T {}^b\mathbf{M} {}^b\mathbf{x}_{\text{ref}} \quad (7.93)$$

$${}^b\tilde{\chi}_u = \frac{1}{m} \Phi_t^T {}^b\mathbf{M} {}^b\tilde{\mathbf{x}}_{\text{ref}} \quad (7.94)$$

$$(7.95)$$

and seven inertia-like matrices [43],

$$\mathbf{M}_{AB} = \mathbf{A}^T {}^b\mathbf{M} \mathbf{B}, \quad \text{using } \mathbf{AB} \in [\Psi\Psi, \tilde{\Psi}\Psi, \tilde{\Psi}\tilde{\Psi}, \Phi_t\Psi, \Phi_t\tilde{\Psi}, \tilde{x}_{\text{ref}}\Psi, \tilde{x}_{\text{ref}}\tilde{\Psi}] \quad (7.96)$$

Note that the special tilde operator for vectors  $\mathbf{p} \in \mathbb{R}^{n_f}$  of Eq. (7.67) is frequently used.

### 7.3.3.4 Equations of motion

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

$$\left( \begin{array}{ccc} \mathbf{M}_{tt} & \mathbf{M}_{tr} & \mathbf{M}_{tf} \\ & \mathbf{M}_{rr} & \mathbf{M}_{rf} \\ \text{sym.} & & \mathbf{M}_{ff} \end{array} \right) \ddot{\mathbf{q}} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathbf{D}_{ff} \end{bmatrix} \dot{\mathbf{q}} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathbf{K}_{ff} \end{bmatrix} \mathbf{q} = \mathbf{f}_v(\mathbf{q}, \dot{\mathbf{q}}) + \mathbf{f}_{user}(mbs, t, \mathbf{q}, \dot{\mathbf{q}}) \quad (7.97)$$

<sup>3</sup> Note that in case of Euler parameters for the parameterization of rotations for the reference frame, the Euler parameter constraint equation is added automatically by this object. The single terms of the mass matrix are defined as [43]

$$\mathbf{M}_{tt} = m\mathbf{I}_{3 \times 3} \quad (7.99)$$

$$\mathbf{M}_{tr} = -{}^{0b}\mathbf{A} \left[ m {}^b\tilde{\chi}_u + \mathbf{M}_{\Phi_t\tilde{\Psi}} (\zeta \otimes \mathbf{I}) \right] {}^b\mathbf{G} \quad (7.100)$$

$$\mathbf{M}_{tf} = {}^{0b}\mathbf{A} \mathbf{M}_{\Phi_t\Psi} \quad (7.101)$$

$$\mathbf{M}_{rr} = {}^b\mathbf{G}^T \left[ {}^b\Theta_u + \mathbf{M}_{\tilde{x}_{\text{ref}}\tilde{\Psi}} (\zeta \otimes \mathbf{I}) + (\zeta \otimes \mathbf{I})^T \mathbf{M}_{\tilde{x}_{\text{ref}}\tilde{\Psi}}^T + (\zeta \otimes \mathbf{I})^T \mathbf{M}_{\tilde{\Psi}\tilde{\Psi}} (\zeta \otimes \mathbf{I}) \right] {}^b\mathbf{G} \quad (7.102)$$

$$\mathbf{M}_{rf} = -{}^b\mathbf{G}^T \left[ \mathbf{M}_{\tilde{x}_{\text{ref}}\Psi} + (\zeta \otimes \mathbf{I})^T \mathbf{M}_{\tilde{\Psi}\Psi} \right] \quad (7.103)$$

$$\mathbf{M}_{ff} = \mathbf{M}_{\Psi\Psi} \quad (7.104)$$

---

<sup>3</sup>NOTE that currently the internal (C++) computed terms are zero,

$$\left( \begin{array}{ccc} \mathbf{M}_{tt} & \mathbf{M}_{tr} & \mathbf{M}_{tf} \\ & \mathbf{M}_{rr} & \mathbf{M}_{rf} \\ \text{sym.} & & \mathbf{M}_{ff} \end{array} \right) = \mathbf{0} \quad \text{and} \quad \mathbf{f}_v(\mathbf{q}, \dot{\mathbf{q}}) = \mathbf{0}, \quad (7.98)$$

but they are implemented in predefined user functions, see `FEM.py`, [Section 5.4.6](#). In near future, these terms will be implemented in C++ and replace the user functions.

with the Kronecker product<sup>4</sup>,

$$\zeta \otimes \mathbf{I} = \begin{bmatrix} \zeta_0 \mathbf{I} \\ \vdots \\ \zeta_{m-1} \mathbf{I} \end{bmatrix} \quad (7.105)$$

The quadratic velocity vector  $\mathbf{f}_v(\mathbf{q}, \dot{\mathbf{q}}) = [\mathbf{f}_{vt}^T, \mathbf{f}_{vr}^T, \mathbf{f}_{vf}^T]^T$  reads

$$\begin{aligned} \mathbf{f}_{vt} &= {}^{0b}\mathbf{A} {}^b\tilde{\omega} [m {}^b\tilde{\chi}_u + \mathbf{M}_{\Phi\tilde{\Psi}}(\zeta \otimes \mathbf{I})] {}^b\omega + 2 {}^{0b}\mathbf{A} \mathbf{M}_{\Phi\tilde{\Psi}} (\zeta \otimes \mathbf{I}) {}^b\omega \\ &\quad + {}^{0b}\mathbf{A} [m {}^b\tilde{\chi}_u + \mathbf{M}_{\Phi\tilde{\Psi}}(\zeta \otimes \mathbf{I})] {}^b\dot{\mathbf{G}} \dot{\theta}, \end{aligned} \quad (7.106)$$

$$\begin{aligned} \mathbf{f}_{vr} &= -{}^b\mathbf{G}^T {}^b\tilde{\omega} \left[ {}^b\Theta_u + \mathbf{M}_{\tilde{x}_{ref}\tilde{\Psi}}(\zeta \otimes \mathbf{I}) + (\zeta \otimes \mathbf{I})^T \mathbf{M}_{\tilde{x}_{ref}\tilde{\Psi}}^T + (\zeta \otimes \mathbf{I})^T \mathbf{M}_{\tilde{\Psi}\tilde{\Psi}}(\zeta \otimes \mathbf{I}) \right] {}^b\omega \\ &\quad - 2 {}^b\mathbf{G}^T \left[ \mathbf{M}_{\tilde{x}_{ref}\tilde{\Psi}}(\zeta \otimes \mathbf{I}) + (\zeta \otimes \mathbf{I})^T \mathbf{M}_{\tilde{\Psi}\tilde{\Psi}}(\zeta \otimes \mathbf{I}) \right] {}^b\omega \\ &\quad - {}^b\mathbf{G}^T \left[ {}^b\Theta_u + \mathbf{M}_{\tilde{x}_{ref}\tilde{\Psi}}(\zeta \otimes \mathbf{I}) + (\zeta \otimes \mathbf{I})^T \mathbf{M}_{\tilde{x}_{ref}\tilde{\Psi}}^T + (\zeta \otimes \mathbf{I})^T \mathbf{M}_{\tilde{\Psi}\tilde{\Psi}}(\zeta \otimes \mathbf{I}) \right] {}^b\dot{\mathbf{G}} \dot{\theta}, \end{aligned} \quad (7.107)$$

$$\begin{aligned} \mathbf{f}_{vf} &= (\mathbf{I}_\zeta \otimes {}^b\omega)^T \left[ \mathbf{M}_{\tilde{x}_{ref}\tilde{\Psi}}^T + \mathbf{M}_{\tilde{\Psi}\tilde{\Psi}}(\zeta \otimes \mathbf{I}) \right] {}^b\omega + 2 \mathbf{M}_{\tilde{\Psi}\tilde{\Psi}}^T (\zeta \otimes \mathbf{I}) {}^b\omega \\ &\quad + \left[ \mathbf{M}_{\tilde{x}_{ref}\tilde{\Psi}}^T + \mathbf{M}_{\tilde{\Psi}\tilde{\Psi}}^T (\zeta \otimes \mathbf{I}) \right] {}^b\dot{\mathbf{G}} \dot{\theta}. \end{aligned} \quad (7.108)$$

Note that terms including  ${}^b\dot{\mathbf{G}} \dot{\theta}$  vanish in case of Euler parameters or in case that  ${}^b\dot{\mathbf{G}} = \mathbf{0}$ , and we use another Kronecker product with the unit matrix  $\mathbf{I}_\zeta \in \mathbb{R}^{n_m \times n_m}$ ,

$$\mathbf{I}_\zeta \otimes {}^b\omega = \begin{bmatrix} {}^b\omega & & \\ & \ddots & \\ & & {}^b\omega \end{bmatrix} \in \mathbb{R}^{3n_m \times n_m} \quad (7.109)$$

In case that `computeFFRFterms = False`, the mass terms  $\mathbf{M}_{tt} \dots \mathbf{M}_{ff}$  are zero (not computed) and the quadratic velocity vector  $\mathbf{f}_Q = \mathbf{0}$ . Note that the user functions  $\mathbf{f}_{user}(mbs, t, \mathbf{q}, \dot{\mathbf{q}})$  and  $\mathbf{M}_{user}(mbs, t, \mathbf{q}, \dot{\mathbf{q}})$  may be empty (=0). The detailed equations of motion for this element can be found in [43].

### 7.3.3.5 Position Jacobian

For joints and loads, the position jacobian of a node is needed in order to compute forces applied to averaged displacements and rotations at nodes. Recall that the modal coordinates  $\zeta$  are transformed to node coordinates by means of the mode basis  ${}^b\Psi$ ,

$${}^b\mathbf{q}_f = {}^b\Psi \zeta. \quad (7.110)$$

The local displacements  ${}^b\mathbf{u}_f^{(i)}$  of a specific node  $i$  can be reconstructed in this way by means of

$${}^b\mathbf{u}_f^{(i)} = \begin{bmatrix} {}^b\mathbf{q}_{f,i,3} \\ {}^b\mathbf{q}_{f,i,3+1} \\ {}^b\mathbf{q}_{f,i,3+2} \end{bmatrix}, \quad (7.111)$$

and the global position of a node, see tables above, reads

$${}^0\mathbf{p}^{(i)} = {}^0\mathbf{p}_t + {}^{0b}\mathbf{A} \left( {}^b\mathbf{u}_f^{(i)} + {}^b\mathbf{x}_{ref}^{(i)} \right) \quad (7.112)$$

Thus, the jacobian of the global position reads

$${}^0\mathbf{J}_{pos}^{(i)} = \frac{\partial {}^0\mathbf{p}^{(i)}}{\partial [\mathbf{q}_t, \theta, \zeta]} = \left[ \mathbf{I}_{3 \times 3}, -{}^{0b}\mathbf{A} \left( {}^b\tilde{\mathbf{u}}_f^{(i)} + {}^b\tilde{\mathbf{x}}_{ref}^{(i)} \right) {}^b\mathbf{G}, {}^{0b}\mathbf{A} \begin{bmatrix} {}^b\Psi_{r=3i}^T \\ {}^b\Psi_{r=3i+1}^T \\ {}^b\Psi_{r=3i+2}^T \end{bmatrix} \right], \quad (7.113)$$

<sup>4</sup>In Python numpy module this is computed by `numpy.kron(zeta, Im)`.

in which  ${}^b\Psi_{r=...}$  represents the row  $r$  of the mode basis (matrix)  ${}^b\Psi$ , and the matrix

$$\begin{bmatrix} {}^b\Psi_{r=3i}^T \\ {}^b\Psi_{r=3i+1}^T \\ {}^b\Psi_{r=3i+2}^T \end{bmatrix} \in \mathbb{R}^{3 \times n_m} \quad (7.114)$$

Furthermore, the jacobian of the local position reads

$${}^bJ_{\text{pos}}^{(i)} = \frac{\partial {}^bP_f^{(i)}}{\partial [\mathbf{q}_t, \boldsymbol{\theta}, \zeta]} = \left[ \mathbf{0}, \mathbf{0}, \begin{bmatrix} {}^b\Psi_{r=3i}^T \\ {}^b\Psi_{r=3i+1}^T \\ {}^b\Psi_{r=3i+2}^T \end{bmatrix} \right], \quad (7.115)$$

which is used in `MarkerSuperElementRigid`.

### 7.3.3.6 Joints and Loads

Use special `MarkerSuperElementPosition` to apply forces, SpringDampers or spherical joints. This marker can be attached to a single node of the underlying mesh or to a set of nodes, which is then averaged, see the according marker description.

Use special `MarkerSuperElementRigid` to apply torques or special joints (e.g., `JointGeneric`). This marker must be attached to a set of nodes which can represent rigid body motion. The rigid body motion is then averaged for all of these nodes, see the according marker description.

For application of mass proportional loads (gravity), you can use conventional `MarkerBodyMass`. However, **do not use** `MarkerBodyPosition` or `MarkerBodyRigid` for `ObjectFFRFreducedOrder`, unless wanted, because it only attaches to the floating frame. This means, that a force to a `MarkerBodyPosition` would only be applied to the (rigid) floating frame, but not onto the deformable body and results depend strongly on the choice of the reference frame (or the underlying mode shapes).

`CoordinateLoads` are added for each `ODE2` coordinate on the RHS of the equations of motion.

---

#### Userfunction: `forceUserFunction(mbs, t, itemNumber, q, q_t)`

A user function, which computes a force vector depending on current time and states of object. Can be used to create any kind of mechanical system by using the object states. Note that `itemNumber` represents the index of the `ObjectFFRFreducedOrder` object in `mbs`, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

arguments / return	type or size	description
<code>mbs</code>	MainSystem	provides MainSystem <code>mbs</code> to which object belongs
<code>t</code>	Real	current time in <code>mbs</code>
<code>itemNumber</code>	Index	integer number of the object in <code>mbs</code> , allowing easy access to all object data via <code>mbs.GetObjectParameter(itemNumber, ...)</code>
<code>q</code>	Vector $\in \mathbb{R}_{\text{ODE2}}^n$	<code>FFRF</code> object coordinates (rigid body coordinates and reduced coordinates in a list) in current configuration, without reference values
<code>q_t</code>	Vector $\in \mathbb{R}_{\text{ODE2}}^n$	object velocity coordinates (time derivatives of <code>q</code> ) in current configuration
<b>return value</b>	Vector $\in \mathbb{R}_{\text{ODE2}}^n$	returns force vector for object

---

**Userfunction: massMatrixUserFunction(mbs, t, itemNumber, q, q\_t)**

A user function, which computes a mass matrix depending on current time and states of object. Can be used to create any kind of mechanical system by using the object states.

arguments / return	type or size	description
mbs	MainSystem	provides MainSystem mbs to which object belongs
t	Real	current time in mbs
itemNumber	Index	integer number of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
q	Vector $\in \mathbb{R}_{ODE2}^n$	FFRF object coordinates (rigid body coordinates and reduced coordinates in a list) in current configuration, without reference values
q_t	Vector $\in \mathbb{R}_{ODE2}^n$	object velocity coordinates (time derivatives of q) in current configuration
<b>return value</b>	NumpyMatrix $\in \mathbb{R}^{n_{ODE2} \times n_{ODE2}}$	returns mass matrix for object

---

For examples on ObjectFFRFreducedOrder see Examples and TestModels:

- [NGsolvePistonEngine.py](#) (Examples/)
- [objectFFRFreducedOrderNetgen.py](#) (Examples/)
- [NGsolveCrankShaftTest.py](#) (TestModels/)
- [objectFFRFreducedOrderAccelerations.py](#) (TestModels/)
- [objectFFRFreducedOrderShowModes.py](#) (TestModels/)
- [objectFFRFreducedOrderStressModesTest.py](#) (TestModels/)
- [superElementRigidJointTest.py](#) (TestModels/)

## 7.4 Objects (FiniteElement)

A FiniteElement is a special Object and Body, which is used to define deformable bodies, such as beams or solid finite elements. FiniteElements are usually linked to two or more nodes.

### 7.4.1 ObjectANCF Cable2D

A 2D cable finite element using 2 nodes of type NodePoint2DSlope1. The localPosition of the beam with length  $L=\text{physicsLength}$  and height  $h$  ranges in X-direction in range  $[0, L]$  and in Y-direction in range  $[-h/2, h/2]$  (which is in fact not needed in the equations of motion ([EOM](#))).

**Additional information for ObjectANCF Cable2D:**

- Requested node type = Position2D + Orientation2D + Point2DSlope1 + Position + Orientation
- **Short name for Python = Cable2D**
- **Short name for Python (visualization object) = VCable2D**

The item **ObjectANCF Cable2D** with type = 'ANCF Cable2D' has the following parameters:

Name	type	size	default value	description
name	String		"	objects's unique name
physicsLength	UReal		0.	[SI:m] reference length of beam; such that the total volume (e.g. for volume load) gives $\rho AL$ ; must be positive
physicsMassPerLength	UReal		0.	[SI:kg/m] mass per length of beam
physicsBendingStiffness	UReal		0.	[SI:Nm <sup>2</sup> ] bending stiffness of beam; the bending moment is $m = EI(\kappa - \kappa_0)$ , in which $\kappa$ is the material measure of curvature
physicsAxialStiffness	UReal		0.	[SI:N] axial stiffness of beam; the axial force is $f_{ax} = EA(\varepsilon - \varepsilon_0)$ , in which $\varepsilon =  \mathbf{r}'  - 1$ is the axial strain
physicsBendingDamping	UReal		0.	[SI:Nm <sup>2</sup> /s] bending damping of beam ; the additional virtual work due to damping is $\delta W_\kappa = \int_0^L \dot{\kappa} \delta \kappa dx$
physicsAxialDamping	UReal		0.	[SI:N/s] axial stiffness of beam; the additional virtual work due to damping is $\delta W_\varepsilon = \int_0^L \dot{\varepsilon} \delta \varepsilon dx$
physicsReferenceAxialStrain	Real		0.	[SI:1] reference axial strain of beam (pre-deformation) of beam; without external loading the beam will statically keep the reference axial strain value
physicsReferenceCurvature	Real		0.	[SI:1/m] reference curvature of beam (pre-deformation) of beam; without external loading the beam will statically keep the reference curvature value
strainIsRelativeToReference	Real		0.	if set to 1., a pre-deformed reference configuration is considered as the stressless state; if set to 0., the straight configuration plus the values of $\varepsilon_0$ and $\kappa_0$ serve as a reference geometry; allows also values between 0. and 1.

nodeNumbers	NodeIndex2	[MAXINT, MAX-INT]	two node numbers ANCF cable element
useReducedOrderIntegration	Index	0	0/false: use Gauss order 9 integration for virtual work of axial forces, order 5 for virtual work of bending moments; 1/true: use Gauss order 7 integration for virtual work of axial forces, order 3 for virtual work of bending moments
visualization	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

#### 7.4.1.1 DESCRIPTION of ObjectANCFCable2D:

##### Information on input parameters:

input parameter	symbol	description see tables above
physicsLength	$L$	
physicsMassPerLength	$\rho A$	
physicsBendingStiffness	$EI$	
physicsAxialStiffness	$EA$	
physicsBendingDamping	$d_e$	
physicsAxialDamping	$d_K$	
physicsReferenceAxialStrain	$\varepsilon_0$	
physicsReferenceCurvature	$\kappa_0$	
strainIsRelativeToReference	$f_{ref}$	

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

output variable	symbol	description
Position	${}^0\mathbf{p}_{config}(x, y, 0) = \mathbf{r}_{config}(x) + y \cdot \mathbf{n}_{config}(x)$	global position vector of local position $[x, y, 0]$
Displacement	${}^0\mathbf{u}_{config}(x, y, 0) = {}^0\mathbf{p}_{config}(x, y, 0) - {}^0\mathbf{p}_{ref}(x, y, 0)$	global displacement vector of local position
Velocity	${}^0\mathbf{v}(x, y, 0) = {}^0\dot{\mathbf{r}}(x) - y \cdot \omega_2 \cdot {}^0\mathbf{t}(x)$	global velocity vector of local position
VelocityLocal	${}^b\mathbf{v}(x, y, 0) = {}^{b0}\mathbf{A} {}^0\mathbf{v}(x, y, 0)$	local velocity vector of local position

Rotation	$\varphi = \text{atan2}(r'_y, r'_x)$	(scalar) rotation angle of axial slope vector (relative to global x-axis)
Director1	$\mathbf{r}'(x)$	(axial) slope vector of local axis position (at $y=0$ )
StrainLocal	$\varepsilon$	axial strain (scalar) of local axis position (at $Y=0$ )
CurvatureLocal	$K$	axial strain (scalar)
ForceLocal	$N$	(local) section normal force (scalar, including reference strains) (at $y=0$ ); note that strains are highly inaccurate when coupled to bending, thus consider useReducedOrderIntegration=2 and evaluate axial strain at nodes or at midpoint
TorqueLocal	$M$	(local) bending moment (scalar) (at $y=0$ )
AngularVelocity	$\boldsymbol{\omega} = [0, , 0, \omega_2]$	angular velocity of local axis position (at $y=0$ )
Acceleration	${}^0\mathbf{a}(x, y, 0) = {}^0\dot{\mathbf{r}}(x) - \mathbf{y} \cdot \omega_2 \cdot {}^0\mathbf{t}(x) - \mathbf{y} \cdot \omega_2 \cdot {}^0\ddot{\mathbf{t}}(x)$	global acceleration vector of local position
AngularAcceleration	$\boldsymbol{\alpha} = [0, , 0, \dot{\omega}_2]$	angular acceleration of local axis position

#### 7.4.1.2 Definition of quantities

intermediate variables	symbol	description
beam height	$h$	beam height used in several definitions, but effectively undefined. The geometry of the cross section has no influence except for drawing or contact.
local beam position	${}^b\mathbf{b} = [x, y, 0]^T$	local position at axial coordinate $x \in [0, L]$ and cross section coordinate $y \in [-h/2, h/2]$ .
beam axis position	${}^0\mathbf{r}(x) = \mathbf{r}(x)$	
beam axis slope	${}^0\mathbf{r}'(x) = \mathbf{r}'(x)$	
beam axis tangent	${}^0\mathbf{t}(x) = \frac{{}^0\mathbf{r}'(x)}{\ {}^0\mathbf{r}'(x)\ }$	this (normalized) vector is normal to cross section
beam axis normal	${}^0\mathbf{n}(x) = [n_x, n_y]^T = [-t_y, t_x]^T$	this (normalized) vector lies within the cross section and defines positive $y$ -direction.
angular velocity	$\omega_2 = (-r'_y \cdot r'_x + r'_x \cdot r'_y) / \ {}^0\mathbf{r}'(x)\ ^2$	
rotation matrix	${}^{0b}\mathbf{A}$	

The Bernoulli-Euler beam is capable of large axial and bendig deformation as it employs the material measure of curvature for the bending.

#### 7.4.1.3 Kinematics and interpolation

<sup>5</sup> ANCF elements follow the original concept proposed by Shabana [33]. The present 2D element is based on the interpolation used by Berzeri and Shabana [4], but the formulation (especially of the elastic forces) is according

<sup>5</sup>Note that in this section, expressions are written in 2D, while output variables are in general 3D quantities, adding a zero for the z-coordinate.

to Gerstmayr and Irschik [17]. Slight improvements for the integration of elastic forces and additional terms for off-axis forces and constraints are mentioned here.

The current position of an arbitrary element at local axial position  $x \in [0, L]$ , where  $L$  is the beam length, reads

$$\mathbf{r} = \mathbf{r}(x, t), \quad (7.116)$$

The derivative of the position w.r.t. the axial reference coordinate is denoted as slope vector,

$$\mathbf{r}' = \frac{\partial \mathbf{r}(x, t)}{\partial x} \quad (7.117)$$

The interpolation is based on cubic (spline) interpolation of position, displacements and velocities. The generalized coordinates  $\mathbf{q} \in \mathbb{R}^8$  of the beam element is defined by

$$\mathbf{q} = [\mathbf{r}_0^T \ \mathbf{r}'_0^T \ \mathbf{r}_1^T \ \mathbf{r}'_1^T]^T. \quad (7.118)$$

in which  $\mathbf{r}_0$  is the position of node 0 and  $\mathbf{r}_1$  is the position of node 1,  $\mathbf{r}'_0$  the slope at node 0 and  $\mathbf{r}'_1$  the slope at node 1. Note that ANCF coordinates in the present notation are computed as sum of reference and current coordinates

$$\mathbf{q} = \mathbf{q}_{\text{cur}} + \mathbf{q}_{\text{ref}} \quad (7.119)$$

which is used throughout here. For time derivatives, it follows that  $\dot{\mathbf{q}} = \dot{\mathbf{q}}_{\text{cur}}$ .

Position and slope are interpolated with shape functions. The position and slope along the beam are interpolated by means of

$$\mathbf{r} = \mathbf{S}\mathbf{q} \quad \text{and} \quad \mathbf{r}' = \mathbf{S}'\mathbf{q}. \quad (7.120)$$

in which  $\mathbf{S}$  is the shape function matrix,

$$\mathbf{S}(x) = [S_1(x) \mathbf{I}_{2 \times 2} \ S_2(x) \mathbf{I}_{2 \times 2} \ S_3(x) \mathbf{I}_{2 \times 2} \ S_4(x) \mathbf{I}_{2 \times 2}]. \quad (7.121)$$

with identity matrix  $\mathbf{I}_{2 \times 2} \in \mathbb{R}^{2 \times 2}$  and the shape functions

$$\begin{aligned} S_1(x) &= 1 - 3\frac{x^2}{L^2} + 2\frac{x^3}{L^3}, & S_2(x) &= x - 2\frac{x^2}{L} + \frac{x^3}{L^2} \\ S_3(x) &= 3\frac{x^2}{L^2} - 2\frac{x^3}{L^3}, & S_4(x) &= -\frac{x^2}{L} + \frac{x^3}{L^2} \end{aligned} \quad (7.122)$$

Velocity simply follows as

$$\frac{\partial \mathbf{r}}{\partial t} = \dot{\mathbf{r}} = \mathbf{S}\dot{\mathbf{q}}. \quad (7.123)$$

#### 7.4.1.4 Mass matrix

The mass matrix is constant and therefore precomputed at the first time it is needed (e.g., during computation of initial accelerations). The analytical form of the mass matrix reads

$$\mathbf{M}_{\text{analytic}} = \int_0^L \rho A \mathbf{S}(x)^T \mathbf{S}(x) dx \quad (7.124)$$

which is approximated using

$$\mathbf{M} = \sum_{ip=0}^{n_{ip}-1} \frac{L}{2} \rho A \mathbf{S}(x_{ip})^T \mathbf{S}(x_{ip}) \quad (7.125)$$

with  $x_{ip}$  evaluated at the integration points,

$$x_{ip} = \frac{L}{2} \xi_{ip} + \frac{L}{2}. \quad (7.126)$$

Here, we use the Gauss integration rule with order 7, having  $n_{ip} = 4$  Gauss points, see [Section 6.1.8](#). Due to the third order polynomials, the integration is exact up to round-off errors.

#### 7.4.1.5 Elastic forces

The elastic forces  $\mathbf{Q}_e$  are implicitly defined by the relation to the virtual work of elastic forces,  $\delta W_e$ , of applied forces,  $\delta W_a$  and of viscous forces,  $\delta W_v$ ,

$$\mathbf{Q}_e^T \delta \mathbf{q} = \delta W_e + \delta W_a + \delta W_v. \quad (7.127)$$

The virtual work of elastic forces reads [17],

$$\delta W_e = \int_0^L (N\delta\varepsilon + M\delta K) dx, \quad (7.128)$$

in which the axial strain is defined as [17]

$$\varepsilon = \|\mathbf{r}'\| - 1. \quad (7.129)$$

and the material measure of curvature (bending strain) is given as

$$K = \mathbf{e}_3^T \frac{\mathbf{r}' \times \mathbf{r}''}{\|\mathbf{r}'\|^2}. \quad (7.130)$$

in which  $\mathbf{e}_3$  is the unit vector which is perpendicular to the plane of the planar beam element.

By derivation, we obtain the variation of axial strain

$$\delta\varepsilon = \frac{\partial\varepsilon}{\partial q_i} \delta q_i = \frac{1}{\|\mathbf{r}'\|} \mathbf{r}'^T \mathbf{S}'_i \delta q_i. \quad (7.131)$$

and the variation of  $K$

$$\begin{aligned} \delta K &= \frac{\partial}{\partial q_i} \left( \frac{(\mathbf{r}'^T \times \mathbf{r}'')^T \mathbf{e}_3}{\|\mathbf{r}'\|^2} \right) \delta q_i \\ &= \frac{1}{\|\mathbf{r}'\|^4} \left[ \|\mathbf{r}'\|^2 (\mathbf{S}'_i \times \mathbf{r}'' + \mathbf{r}' \times \mathbf{S}''_i) - 2(\mathbf{r}' \times \mathbf{r}'') (\mathbf{r}'^T \mathbf{S}'_i) \right]^T \mathbf{e}_3 \delta q_i \end{aligned} \quad (7.132)$$

The normal force (axial force)  $N$  in the beam is defined as function of the current strain  $\varepsilon$ ,

$$N = EA(\varepsilon - \varepsilon_0 - f_{\text{ref}} \cdot \varepsilon_{\text{ref}}). \quad (7.133)$$

in which  $\varepsilon_0$  includes the (pre-)stretch of the beam, e.g., due to temperature or plastic deformation and  $\varepsilon_{\text{ref}}$  includes the strain of the reference configuration. As can be seen, the reference strain is only considered, if  $f_{\text{ref}} = 1$ , which allows to consider the reference configuration to be completely stress-free (but the default value is  $f_{\text{ref}} = 0$ !). Note that – due to the inherent nonlinearity of  $\varepsilon$  – a combination of  $\varepsilon_0$  and  $f_{\text{ref}} = 1$  is physically only meaningful for small strains. A factor  $f_{\text{ref}} < 1$  allows to realize a smooth transition between deformed and straight reference configuration, e.g. for initial configurations.

The bending moment  $M$  in the beam is defined as function of the current material measure of curvature  $K$ ,

$$M = EI(K - K_0 - f_{\text{ref}} \cdot K_{\text{ref}}). \quad (7.134)$$

in which  $K_0$  includes the (pre-)curvature of the undeformed beam and  $K_{\text{ref}}$  includes the curvature of the reference configuration, multiplied with the factor  $f_{\text{ref}} = 1$ , see the axial strain above.

Using the latter definitions, the elastic forces follow from Eq. (7.127).

The virtual work of viscous damping forces, assuming viscous effects proportional to axial stretching and bending, is defined as

$$\delta W_v = \int_0^L (d_\varepsilon \dot{\varepsilon} \delta\varepsilon + d_K \dot{K} \delta K) dx. \quad (7.135)$$

with material coefficients  $d_\varepsilon$  and  $d_K$ . The time derivatives of axial strain  $\dot{\varepsilon}_p$  follows by elementary differentiation

$$\dot{\varepsilon} = \frac{\partial}{\partial t} (\|\mathbf{r}'\| - 1) = \frac{1}{\|\mathbf{r}'\|} \mathbf{r}'^T \mathbf{S}' \dot{\mathbf{q}} \quad (7.136)$$

as well as the derivative of the curvature,

$$\begin{aligned}\dot{K} &= \frac{\partial}{\partial t} \left( \mathbf{e}_3^T \frac{\mathbf{r}' \times \mathbf{r}''}{\|\mathbf{r}'\|^2} \right) \\ &= \frac{\mathbf{e}_3^T}{(\mathbf{r}'^T \mathbf{r}')^2} \left( (\mathbf{r}'^T \mathbf{r}') \frac{\partial (\mathbf{r}' \times \mathbf{r}'')^T}{\partial t} - (\mathbf{r}' \times \mathbf{r}'')^T \frac{\partial (\mathbf{r}'^T \mathbf{r}')}{\partial t} \right) \\ &= \frac{\mathbf{e}_3^T}{(\mathbf{r}'^T \mathbf{r}')^2} \left( (\mathbf{r}'^T \mathbf{r}') ((\mathbf{S}' \dot{\mathbf{q}}) \times \mathbf{r}'' + (\mathbf{S}'' \dot{\mathbf{q}}) \times \mathbf{r}') - (\mathbf{r}' \times \mathbf{r}'') (2\mathbf{r}'^T (\mathbf{S}' \dot{\mathbf{q}})) \right).\end{aligned}\quad (7.137)$$

The virtual work of applied forces reads

$$\delta W_a = \sum_i \mathbf{f}_i^T \delta \mathbf{r}_i(x_f) + \int_0^L \mathbf{b}^T \delta \mathbf{r}(x) dx,\quad (7.138)$$

in which  $\mathbf{f}_i$  are forces applied to a certain position  $x_f$  at the beam centerline. The second term contains a load per length  $\mathbf{b}$ , which in case of gravity vector  $\mathbf{g}$  reads

$$\mathbf{b} = \rho \mathbf{g}. \quad (7.139)$$

Note that the variation of  $\mathbf{r}$  simply follows as

$$\delta \mathbf{r} = \mathbf{S} \delta \mathbf{q} \quad (7.140)$$

#### 7.4.1.6 Numerical integration of Elastic Forces

The numerical integration of elastic forces  $\mathbf{Q}_e$  is split into terms due to  $\delta \varepsilon$  and  $\delta K$ ,

$$\mathbf{Q}_e = \int_0^L \left( \bullet(x) \frac{\partial \delta \varepsilon}{\partial \delta \mathbf{q}} + \bullet(x) \frac{\partial \delta K}{\partial \delta \mathbf{q}} \right) dx \quad (7.141)$$

using different integration rules

$$\mathbf{Q}_e \approx \sum_{ip=0}^{n_{ip}^e-1} \left( \frac{L}{2} \bullet(x_{ip}) \frac{\partial \delta \varepsilon}{\partial \delta \mathbf{q}} \right) + \sum_{ip=0}^{n_{ip}^K-1} \left( \frac{L}{2} \bullet(x_{ip}) \frac{\partial \delta K}{\partial \delta \mathbf{q}} \right) dx \quad (7.142)$$

with the integration points  $x_{ip}$  as defined in Eq. (7.126) and integration rules from [Section 6.1.8](#). There are 3 different options for integration rules depending on the flag `useReducedOrderIntegration`:

1. `useReducedOrderIntegration = 0`:  $n_{ip}^e = 5$  (Gauss order 9),  $n_{ip}^K = 3$  (Gauss order 5) – this is considered as full integration, leading to very small approximations; certainly, due to the high nonlinearity of expressions, this is only an approximation.
2. `useReducedOrderIntegration = 1`:  $n_{ip}^e = 4$  (Gauss order 7),  $n_{ip}^K = 2$  (Gauss order 3) – this is considered as reduced integration, which is usually sufficiently accurate but leads to slightly less computational efforts, especially for bending terms.
3. `useReducedOrderIntegration = 2`:  $n_{ip}^e = 3$  (Lobatto order 4),  $n_{ip}^K = 2$  (Gauss order 3) – this is a further reduced integration, with the exceptional property that axial strain and bending strain terms are computed at completely disjointed locations: axial strain terms are evaluated at 0,  $L/2$  and  $L$ , while bending terms are evaluated at  $\pm \frac{L}{2} \sqrt{1/3}$ . This allows axial strains to freely follow the bending terms at  $\pm \frac{L}{2} \sqrt{1/3}$ , while axial strains are almost independent from bending terms at 0,  $L/2$  and  $L$ . However, due to the highly reduced integration, spurious (hourglass) modes may occur in certain applications!

Note that the Jacobian of elastic forces is computed using automatic differentiation.

#### 7.4.1.7 Access functions

For application of forces and constraints at any local beam position  ${}^b\mathbf{b} = [x, y, 0]^T$ , the position / velocity Jacobian reads

$$\frac{\partial {}^0\mathbf{v}(x)}{\dot{\mathbf{q}}} = \mathbf{S}(x) + \left[ -y \cdot n_x S'_1(x) \frac{1}{\|\mathbf{r}'\|} {}^0\mathbf{t}, -y \cdot n_y S'_1(x) \frac{1}{\|\mathbf{r}'\|} {}^0\mathbf{t}, -y \cdot n_x S'_2(x) \frac{1}{\|\mathbf{r}'\|} {}^0\mathbf{t}, \dots \right] \quad (7.143)$$

with the normalized beam axis normal  ${}^0\mathbf{n} = [n_x, n_y]^T$ , see table above.

For application of torques at any axis point  $x$ , the rotation / angular velocity Jacobian  $\frac{\partial {}^0\omega(x)}{\dot{\mathbf{q}}} \in \mathbb{R}^{3 \times 8}$  reads

$$\frac{\partial {}^0\omega(x)}{\dot{\mathbf{q}}} = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ -r'_y \cdot S'_1(x) \frac{1}{r'^2} & r'_x \cdot S'_1(x) \frac{1}{r'^2} & -r'_y \cdot S'_2(x) \frac{1}{r'^2} & \dots & r'_x \cdot S'_4(x) \frac{1}{r'^2} \end{bmatrix} \quad (7.144)$$


---

#### 7.4.1.8 MINI EXAMPLE for ObjectANCFCable2D

```

rhoA = 78.
EA = 1000000.
EI = 833.333333333333
cable = Cable2D(physicsMassPerLength=rhoA,
                  physicsBendingStiffness=EI,
                  physicsAxialStiffness=EA,
                  )

ancf=GenerateStraightLineANCFCable2D(mbs=mbs,
                                         positionOfNode0=[0,0,0], positionOfNode1=[2,0,0],
                                         numberOfElements=32, #converged to 4 digits
                                         cableTemplate=cable, #this defines the beam element properties
                                         massProportionalLoad = [0,-9.81,0],
                                         fixedConstraintsNode0 = [1,1,0,1], #add constraints for pos and rot (r'
                                         -y)
                                         )
lastNode = ancf[0][-1]

#assemble and solve system for default parameters
mbs.Assemble()
exu.SolveStatic(mbs)

#check result
exudynTestGlobals.testResult = mbs.GetNodeOutput(lastNode, exu.OutputVariableType.
Displacement)[0]
#ux=-0.5013058140308901

```

---

For examples on ObjectANCFCable2D see Examples and TestModels:

- [sliderCrank3DwithANCFbeltDrive2.py](#) (Examples/)
- [ALEANCF\\_pipe.py](#) (Examples/)
- [ANCF\\_cantilever\\_test.py](#) (Examples/)
- [ANCF\\_cantilever\\_test\\_dyn.py](#) (Examples/)
- [ANCF\\_contact\\_circle.py](#) (Examples/)
- [ANCF\\_contact\\_circle2.py](#) (Examples/)
- [ANCF\\_moving\\_rigidbody.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- [ANCF\\_slidingJoint2Drigid.py](#) (Examples/)
- [ANCF\\_switchingSlidingJoint2D.py](#) (Examples/)
- [ANCF\\_tests2.py](#) (Examples/)
- [ANCF\\_test\\_halfcircle.py](#) (Examples/)
- ...
- [ANCFcontactCircleTest.py](#) (TestModels/)
- [ANCFcontactFrictionTest.py](#) (TestModels/)
- [computeODE2EigenvaluesTest.py](#) (TestModels/)
- ...

### 7.4.2 ObjectALEANCFCable2D

A 2D cable finite element using 2 nodes of type NodePoint2DSlope1 and a axially moving coordinate of type NodeGenericODE2, which adds additional (redundant) motion in axial direction of the beam. This allows modeling pipes but also axially moving beams. The localPosition of the beam with length  $L=\text{physicsLength}$  and height  $h$  ranges in X-direction in range  $[0, L]$  and in Y-direction in range  $[-h/2, h/2]$  (which is in fact not needed in the EOM).

#### Additional information for ObjectALEANCFCable2D:

- Requested node type: read detailed information of item
- **Short name for Python = ALECFable2D**
- **Short name for Python (visualization object) = VALECFable2D**

The item **ObjectALEANCFCable2D** with type = 'ALEANCFCable2D' has the following parameters:

Name	type	size	default value	description
name	String		"	objects's unique name
physicsLength	UReal		0.	[SI:m] reference length of beam; such that the total volume (e.g. for volume load) gives $\rho AL$ ; must be positive
physicsMassPerLength	UReal		0.	[SI:kg/m] total mass per length of beam (including axially moving parts / fluid)
physicsMovingMassFactor	UReal		1.	this factor denotes the amount of $\rho A$ 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 $\rho A$ is the mass of the pipe+fluid, while $\text{physicsMovingMassFactor} \cdot \rho A$ is the mass per unit length of the fluid
physicsBendingStiffness	UReal		0.	[SI:Nm <sup>2</sup> ] bending stiffness of beam; the bending moment is $m = EI(\kappa - \kappa_0)$ , in which $\kappa$ is the material measure of curvature
physicsAxialStiffness	UReal		0.	[SI:N] axial stiffness of beam; the axial force is $f_{ax} = EA(\varepsilon - \varepsilon_0)$ , in which $\varepsilon =  \dot{\mathbf{r}}'  - 1$ is the axial strain
physicsBendingDamping	UReal		0.	[SI:Nm <sup>2</sup> /s] bending damping of beam ; the additional virtual work due to damping is $\delta W_\kappa = \int_0^L \dot{\kappa} \delta \kappa dx$
physicsAxialDamping	UReal		0.	[SI:N/s] axial stiffness of beam; the additional virtual work due to damping is $\delta W_\varepsilon = \int_0^L \dot{\varepsilon} \delta \varepsilon dx$
physicsReferenceAxialStrain	Real		0.	[SI:1] reference axial strain of beam (pre-deformation) of beam; without external loading the beam will statically keep the reference axial strain value
physicsReferenceCurvature	Real		0.	[SI:1/m] reference curvature of beam (pre-deformation) of beam; without external loading the beam will statically keep the reference curvature value

physicsUseCouplingTerms	Bool	True	true: correct case, where all coupling terms due to moving mass are respected; false: only include constant mass for ALE node coordinate, but deactivate other coupling terms (behaves like ANCF Cable2D then)
nodeNumbers	NodeIndex3	[MAXINT, MAXINT, MAXINT]	two node numbers ANCF cable element, third node=ALE GenericODE2 node
useReducedOrderIntegration	Index	0	0/false: use Gauss order 9 integration for virtual work of axial forces, order 5 for virtual work of bending moments; 1/true: use Gauss order 7 integration for virtual work of axial forces, order 3 for virtual work of bending moments
strainIsRelativeToReference	Real	0.	if set to 1., a pre-deformed reference configuration is considered as the stressless state; if set to 0., the straight configuration plus the values of $\varepsilon_0$ and $\kappa_0$ serve as a reference geometry; allows also values between 0. and 1.
visualization	VObjectALEANCFCable2D		parameters for visualization of item

The item VObjectALEANCFCable2D has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawHeight	float		0.	if beam is drawn with rectangular shape, this is the drawing height
color	Float4		[-1.,-1.,-1.,-1.]	RGBA color of the object; if R== -1, use default color

#### 7.4.2.1 DESCRIPTION of ObjectALEANCFCable2D:

##### Information on input parameters:

input parameter	symbol	description see tables above
physicsLength	$L$	
physicsMassPerLength	$\rho A$	
physicsBendingStiffness	$EI$	
physicsAxialStiffness	$EA$	
physicsBendingDamping	$d_e$	
physicsAxialDamping	$d_K$	
physicsReferenceAxialStrain	$\varepsilon_0$	

physicsReferenceCurvature	$\kappa_0$	
strainIsRelativeToReference	$f_{\text{ref}}$	

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

output variable	symbol	description
Position		global position vector of local position (in X/Y beam coordinates)
Displacement		global displacement vector of local position
Velocity		global velocity vector of local position
VelocityLocal		local velocity vector of local position
Rotation		(scalar) rotation angle of axial slope vector (relative to global x-axis)
Director1		(axial) slope vector of local axis position (at Y=0)
StrainLocal	$\epsilon$	axial strain (scalar) of local axis position (at Y=0)
CurvatureLocal	$K$	axial strain (scalar)
ForceLocal	$N$	(local) section normal force (scalar, including reference strains) (at Y=0); note that strains are highly inaccurate when coupled to bending, thus consider <code>useReducedOrderIntegration=2</code> and evaluate axial strain at nodes or at midpoint
TorqueLocal	$M$	(local) bending moment (scalar) (at Y=0)

A 2D cable finite element using 2 nodes of type `NodePoint2DSlope1` and an axially moving coordinate of type `NodeGenericODE2`. The element has 8+1 coordinates and uses cubic polynomials for position interpolation. In addition to `ANCF` the element adds an Eulerian axial velocity by the `GenericODE2` coordinate. The parameter `physicsMovingMassFactor` allows to control the amount of mass, which moves with the Eulerian velocity (e.g., the fluid), and which is not moving (the pipe). A factor of `physicsMovingMassFactor=1` gives an axially moving beam.

The Bernoulli-Euler beam is capable of large deformation as it employs the material measure of curvature for the bending. Note that damping (`physicsBendingDamping`, `physicsAxialDamping`) only acts on the non-moving part of the beam, as it is the case for the pipe.

Note that most functions act on the underlying cable finite element, which is not co-moving axially. E.g., if you apply constraints to the nodal coordinates, the cable can be fixed, while still the axial component is freely moving. If you apply a `LoadForce` using a `MarkerPosition`, the force is acting on the beam finite element, but not on the axially moving coordinate. In contrast to the latter, the `ObjectJointALEMoving2D` and the `MarkerBodyMass` are acting on the moving coordinate as well.

A detailed paper on this element is yet under submission, but a similar formulation can be found in [31] and the underlying beam element is identical to `ObjectANCF`.

For examples on `ObjectALEANCF` see Examples and TestModels:

- [ALEANCF\\_pipe.py](#) (Examples/)
- [ANCFALTest.py](#) (Examples/)
- [ANCF\\_moving\\_rigidbody.py](#) (Examples/)
- [ANCFoutputTest.py](#) (TestModels/)

### 7.4.3 ObjectBeamGeometricallyExact2D

A 2D geometrically exact beam finite element, currently using 2 nodes of type NodeRigidBody2D. The local Position of the beam with length  $L=\text{physicsLength}$  and height  $h$  ranges in X-direction in range  $[-L/2, L/2]$  and in Y-direction in range  $[-h/2, h/2]$  (which is in fact not needed in the EOM).

**Additional information for ObjectBeamGeometricallyExact2D:**

- The Object has the following types = Body, MultiNoded
- Requested node type = Position2D + Orientation2D + Position + Orientation
- **Short name for Python = Beam2D**
- **Short name for Python (visualization object) = VBeam2D**

The item **ObjectBeamGeometricallyExact2D** with type = 'BeamGeometricallyExact2D' has the following parameters:

Name	type	size	default value	description
name	String		"	objects's unique name
nodeNumbers	NodeIndex2		[MAXINT, MAX-INT]	two node numbers for beam element
physicsLength	UReal		0.	[SI:m] reference length of beam; such that the total volume (e.g. for volume load) gives $\rho AL$ ; must be positive
physicsMassPerLength	UReal		0.	[SI:kg/m] mass per length of beam
physicsCrossSectionInertia	UReal		0.	[SI:kg m] cross section mass moment of inertia; inertia acting against rotation of cross section
physicsBendingStiffness	UReal		0.	[SI:Nm <sup>2</sup> ] bending stiffness of beam; the bending moment is $m = EI(\kappa - \kappa_0)$ , in which $\kappa$ is the material measure of curvature
physicsAxialStiffness	UReal		0.	[SI:N] axial stiffness of beam; the axial force is $f_{ax} = EA(\varepsilon - \varepsilon_0)$ , in which $\varepsilon$ is the axial strain
physicsShearStiffness	UReal		0.	[SI:N] effective shear stiffness of beam, including stiffness correction
visualization	VObjectBeamGeometricallyExact2D			parameters for visualization of item

The item VObjectBeamGeometricallyExact2D has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawHeight	float		0.	if beam is drawn with rectangular shape, this is the drawing height
color	Float4		[-1.,-1.,-1.,-1.]	RGB color of the object; if R==1, use default color

### 7.4.3.1 DESCRIPTION of ObjectBeamGeometricallyExact2D:

Information on input parameters:

input parameter	symbol	description see tables above
physicsLength	$L$	
physicsMassPerLength	$\rho A$	
physicsCrossSectionInertia	$\rho J$	
physicsBendingStiffness	$EI$	
physicsAxialStiffness	$EA$	
physicsShearStiffness	$GA$	

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

output variable	symbol	description
Position		global position vector of local axis (1) and cross section (2) position
Displacement		global displacement vector of local axis (1) and cross section (2) position
Velocity		global velocity vector of local axis (1) and cross section (2) position
Rotation		3D Tait-Bryan rotation components, containing rotation around z-axis only
StrainLocal		6 strain components, containing only axial ( $xx$ ) and shear strain ( $xy$ )
CurvatureLocal		3D vector of curvature, containing only curvature w.r.t. z-axis

See paper of Simo and Vu-Quoc (1986). Detailed description coming later.

For examples on ObjectBeamGeometricallyExact2D see Examples and TestModels:

- [geometricallyExactBeam2Dtest.py](#) (TestModels/)

## 7.5 Objects (Joint)

A Joint is a special Object, Connector and Constraint, which is attached to position or rigid body markers. The joint results in special algebraic equations and requires implicit time integration. Joints represent special constraints, as described in multibody system dynamics literature.

### 7.5.1 ObjectJointGeneric

A generic joint in 3D; constrains components of the absolute position and rotations of two points given by PointMarkers or RigidMarkers. An additional local rotation (rotationMarker) can be used to adjust the three rotation axes and/or sliding axes.

**Additional information for ObjectJointGeneric:**

- The Object has the following types = **Connector, Constraint**
- Requested marker type = **Position + Orientation**
- **Short name for Python = GenericJoint**
- **Short name for Python (visualization object) = VGenericJoint**

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

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex	2	[ MAXINT, MAX-INT ]	list of markers used in connector
constrainedAxes	ArrayIndex	6	[1,1,1,1,1]	flag, which determines which translation (0,1,2) and rotation (3,4,5) axes are constrained; for $j_i$ , two values are possible: 0=free axis, 1=constrained axis
rotationMarker0	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	local rotation matrix for marker $m_0$ ; translation and rotation axes for marker $m_0$ are defined in the local body coordinate system and additionally transformed by rotation-Marker0
rotationMarker1	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	local rotation matrix for marker $m_1$ ; translation and rotation axes for marker $m_1$ are defined in the local body coordinate system and additionally transformed by rotation-Marker1
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
offsetUserFunctionParameters	Vector6D		[0.,0.,0.,0.,0.,0.]	vector of 6 parameters for joint's offsetUser-Function

offsetUserFunction	PyFunctionVector6D	mbs	ScalarIndexVector6D	0	A Python function which defines the time-dependent (fixed) offset of translation (indices 0,1,2) and rotation (indices 3,4,5) joint coordinates with parameters (mbs, t, offsetUserFunctionParameters)
offsetUserFunction_t	PyFunctionVector6D	mbs	ScalarIndexVector6D	0	(NOT IMPLEMENTED YET)time derivative of offsetUserFunction using the same parameters
visualization	VObjectJointGeneric				parameters for visualization of item

The item VObjectJointGeneric has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
axesRadius	float		0.1	radius of joint axes to draw
axesLength	float		0.4	length of joint axes to draw
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R== -1, use default color

### 7.5.1.1 DESCRIPTION of ObjectJointGeneric:

#### Information on input parameters:

input parameter	symbol	description see tables above
markerNumbers	$[m_0, m_1]^T$	
constrainedAxes	$j = [j_0, \dots, j_5]$	
rotationMarker0	${}^{m_0,j_0} \mathbf{A}$	
rotationMarker1	${}^{m_1,j_1} \mathbf{A}$	
offsetUserFunctionParameters	$\mathbf{p}_{par}$	
offsetUserFunction	$UF \in \mathbb{R}^6$	
offsetUserFunction_t	$UF \in \mathbb{R}^6$	

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

output variable	symbol	description
Position	${}^0 \mathbf{p}_{m_0}$	current global position of position marker $m_0$
Velocity	${}^0 \mathbf{v}_{m_0}$	current global velocity of position marker $m_0$
DisplacementLocal	${}^{j_0} \Delta \mathbf{p}$	relative displacement in local joint0 coordinates; uses local J0 coordinates even for spherical joint configuration

VelocityLocal	${}^{J0}\Delta\mathbf{v}$	relative translational velocity in local joint0 coordinates
Rotation	${}^{J0}\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2]^T$	relative rotation parameters (Tait Bryan Rxyz); if all axes are fixed, this output represents the rotational drift; for a revolute joint, it contains the rotation of this axis
AngularVelocityLocal	${}^{J0}\Delta\boldsymbol{\omega}$	relative angular velocity in local joint0 coordinates; if all axes are fixed, this output represents the angular velocity constraint error; for a revolute joint, it contains the angular velocity of this axis
ForceLocal	${}^{J0}\mathbf{f}$	joint force in local $J0$ coordinates
TorqueLocal	${}^{J0}\mathbf{m}$	joint torque in local $J0$ coordinates; depending on joint configuration, the result may not be the according torque vector

### 7.5.1.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker m0
marker m0 orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker m0
joint J0 orientation	${}^{0,J0}\mathbf{A} = {}^{0,m0}\mathbf{A} {}^{m0,J0}\mathbf{A}$	joint $J0$ rotation matrix
joint J0 orientation vectors	${}^{0,J0}\mathbf{A} = [{}^0\mathbf{t}_{x0}, {}^0\mathbf{t}_{y0}, {}^0\mathbf{t}_{z0}]^T$	orientation vectors (represent local $x$ , $y$ , and $z$ axes) in global coordinates, used for definition of constraint equations
marker m1 position	${}^0\mathbf{p}_{m1}$	accordingly
marker m1 orientation	${}^{0,m1}\mathbf{A}$	current rotation matrix provided by marker m1
joint J1 orientation	${}^{0,J1}\mathbf{A} = {}^{0,m1}\mathbf{A} {}^{m1,J1}\mathbf{A}$	joint $J1$ rotation matrix
joint J1 orientation vectors	${}^{0,J1}\mathbf{A} = [{}^0\mathbf{t}_{x1}, {}^0\mathbf{t}_{y1}, {}^0\mathbf{t}_{z1}]^T$	orientation vectors (represent local $x$ , $y$ , and $z$ axes) in global coordinates, used for definition of constraint equations
marker m0 velocity	${}^0\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
marker m1 velocity	${}^0\mathbf{v}_{m1}$	accordingly
marker m0 velocity	${}^b\boldsymbol{\omega}_{m0}$	current local angular velocity vector provided by marker m0
marker m1 velocity	${}^b\boldsymbol{\omega}_{m1}$	current local angular velocity vector provided by marker m1
Displacement	${}^0\Delta\mathbf{p} = {}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0}$	used, if all translational axes are constrained
Velocity	${}^0\Delta\mathbf{v} = {}^0\mathbf{v}_{m1} - {}^0\mathbf{v}_{m0}$	used, if all translational axes are constrained (velocity level)
DisplacementLocal	${}^{J0}\Delta\mathbf{p}$	$({}^{0,m0}\mathbf{A} {}^{m0,J0}\mathbf{A})^T {}^0\Delta\mathbf{p}$
VelocityLocal	${}^{J0}\Delta\mathbf{v}$	$({}^{0,m0}\mathbf{A} {}^{m0,J0}\mathbf{A})^T {}^0\Delta\mathbf{v}$ ... note that this is the global relative velocity projected into the local $J0$ coordinate system

AngularVelocityLocal	${}^{J_0} \Delta \omega$	$({}^{0,m0} \mathbf{A} {}^{m0,J_0} \mathbf{A})^T ({}^{0,m1} \mathbf{A} {}^{m1} \omega - {}^{0,m0} \mathbf{A} {}^{m0} \omega)$
algebraic variables	$\mathbf{z} = [\lambda_0, \dots, \lambda_5]^T$	vector of algebraic variables (Lagrange multipliers) according to the algebraic equations

### 7.5.1.3 Connector constraint equations

#### Equations for translational part (`activeConnector = True`) :

If  $[j_0, \dots, j_2] = [1, 1, 1]^T$ , meaning that all translational coordinates are fixed, the translational index 3 constraints read ( $UF_{0,1,2}(mbs, t, \mathbf{p}_{par})$  is the translational part of the user function  $UF$ ),

$${}^0 \mathbf{p}_{m1} - {}^0 \mathbf{p}_{m0} - UF_{0,1,2}(mbs, t, i_N, \mathbf{p}_{par}) = \mathbf{0} \quad (7.145)$$

and the translational index 2 constraints read

$${}^0 \mathbf{v}_{m1} - {}^0 \mathbf{v}_{m0} - UF_{t,0,1,2}(mbs, t, i_N, \mathbf{p}_{par}) = \mathbf{0} \quad (7.146)$$

and  $i_N$  represents the itemNumber (=objectNumber). If  $[j_0, \dots, j_2] \neq [1, 1, 1]^T$ , meaning that at least one translational coordinate is free, the translational index 3 constraints read for every component  $k \in [0, 1, 2]$  of the vector  ${}^{J_0} \Delta \mathbf{p}$

$${}^{J_0} \Delta p_k - UF_k(mbs, t, i_N, \mathbf{p}_{par}) = 0 \quad \text{if } j_k = 1 \quad \text{and} \quad (7.147)$$

$$\lambda_k = 0 \quad \text{if } j_k = 0 \quad (7.148)$$

$$(7.149)$$

and the translational index 2 constraints read for every component  $k \in [0, 1, 2]$  of the vector  ${}^{J_0} \Delta \mathbf{v}$

$${}^{J_0} \Delta v_k - UF_{-t,k}(mbs, t, i_N, \mathbf{p}_{par}) = 0 \quad \text{if } j_k = 1 \quad \text{and} \quad (7.150)$$

$$\lambda_k = 0 \quad \text{if } j_k = 0 \quad (7.151)$$

$$(7.152)$$

#### Equations for rotational part (`activeConnector = True`) :

The following equations are exemplarily for certain constrained rotation axes configurations, which shall represent all other possibilities. Note that the axes are always given in global coordinates, compare the table in [Section 7.5.1.2](#).

Equations are only given for the index 3 case; the index 2 case can be derived from these equations easily (see C++ code...). In case of user functions, the additional rotation matrix  ${}^{J_0, J_0 U} \mathbf{A}$  ( $UF_{3,4,5}(mbs, t, \mathbf{p}_{par})$ ), in which the three components of  $UF_{3,4,5}$  are interpreted as Tait-Bryan angles that are added to the joint frame.

If 3 rotation axes are constrained (e.g., translational or planar joint),  $[j_3, \dots, j_5] = [1, 1, 1]^T$ , the index 3 constraint equations read

$${}^0 \mathbf{t}_{z0}^T {}^0 \mathbf{t}_{y1} = 0 \quad (7.153)$$

$${}^0 \mathbf{t}_{z0}^T {}^0 \mathbf{t}_{x1} = 0 \quad (7.154)$$

$${}^0 \mathbf{t}_{x0}^T {}^0 \mathbf{t}_{y1} = 0 \quad (7.155)$$

If 2 rotation axes are constrained (revolute joint), e.g.,  $[j_3, \dots, j_5] = [0, 1, 1]^T$ , the index 3 constraint equations read

$$\lambda_3 = 0 \quad (7.156)$$

$${}^0 \mathbf{t}_{x0}^T {}^0 \mathbf{t}_{y1} = 0 \quad (7.157)$$

$${}^0 \mathbf{t}_{x0}^T {}^0 \mathbf{t}_{z1} = 0 \quad (7.158)$$

If **1 rotation axis is constrained** (universal joint), e.g.,  $[j_3, \dots, j_5] = [1, 0, 0]^T$ , the index 3 constraint equations read

$${}^0\mathbf{t}_{y0}^T {}^0\mathbf{t}_{z1} = 0 \quad (7.159)$$

$$\lambda_4 = 0 \quad (7.160)$$

$$\lambda_5 = 0 \quad (7.161)$$

if `activeConnector = False,`

$$\mathbf{z} = \mathbf{0} \quad (7.162)$$


---

### Userfunction: `offsetUserFunction(mbs, t, itemNumber, offsetUserFunctionParameters)`

A user function, which computes scalar offset for relative joint translation and joint rotation for the GenericJoint, e.g., in order to move or rotate a body on a prescribed trajectory. It is NECESSARY to use sufficiently smooth functions, having **initial offsets** consistent with **initial configuration** of bodies, either zero or compatible initial offset-velocity, and no initial accelerations. The `offsetUserFunction` is **ONLY used** in case of static computation or index3 (generalizedAlpha) time integration. In order to be on the safe side, provide both `offsetUserFunction` and `offsetUserFunction_t`.

Note that `itemNumber` represents the index of the object in `mbs`, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

The user function gets time and the `offsetUserFunctionParameters` as an input and returns the computed offset vector for all relative translational and rotational joint coordinates:

arguments / return	type or size	description
<code>mbs</code>	MainSystem	provides MainSystem <code>mbs</code> in which underlying item is defined
<code>t</code>	Real	current time in <code>mbs</code>
<code>itemNumber</code>	Index	integer number of the object in <code>mbs</code> , allowing easy access to all object data via <code>mbs.GetObjectParameter(itemNumber, ...)</code>
<code>offsetUserFunctionParameters</code>	Real	$\mathbf{p}_{par}$ , set of parameters which can be freely used in user function
<b>return value</b>	Real	computed offset vector for given time

---

### Userfunction: `offsetUserFunction_t(mbs, t, itemNumber, offsetUserFunctionParameters)`

A user function, which computes an offset **velocity** vector for the GenericJoint. It is NECESSARY to use sufficiently smooth functions, having **initial offset velocities** consistent with **initial velocities** of bodies. The `offsetUserFunction_t` is used instead of `offsetUserFunction` in case of `velocityLevel = True`, or for index2 time integration and needed for computation of initial accelerations in second order implicit time integrators.

Note that `itemNumber` represents the index of the object in `mbs`, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

The user function gets time and the `offsetUserFunctionParameters` as an input and returns the computed offset velocity vector for all relative translational and rotational joint coordinates:

arguments / return	type or size	description
mbs	MainSystem	provides MainSystem mbs in which underlying item is defined
t	Real	current time in mbs
itemNumber	Index	integer number of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
offsetUserFunctionParameters	Real	$\mathbf{p}_{par}$ , set of parameters which can be freely used in user function
<b>return value</b>	Real	computed offset velocity vector for given time

---

### User function example:

```
#simple example, computing only the translational offset for x-coordinate
from math import sin, cos, pi
def UFoffset(mbs, t, itemNumber, offsetUserFunctionParameters):
    return [offsetUserFunctionParameters[0]*(1 - cos(t*10*2*pi)), 0,0,0,0,0]
```

---

For examples on ObjectJointGeneric see Examples and TestModels:

- [bicycleIftommBenchmark.py](#) (Examples/)
- [CMSEXampleCourse.py](#) (Examples/)
- [fourBarMechanism3D.py](#) (Examples/)
- [leggedRobot.py](#) (Examples/)
- [mouseInteractionExample.py](#) (Examples/)
- [multiMbsTest.py](#) (Examples/)
- [NGsolveCMSTutorial.py](#) (Examples/)
- [NGsolveCraigBampton.py](#) (Examples/)
- [ObjectFFRFconvergenceTestBeam.py](#) (Examples/)
- [ObjectFFRFconvergenceTestHinge.py](#) (Examples/)
- [pendulumVerify.py](#) (Examples/)
- [rigidBodyTutorial.py](#) (Examples/)
- ...
- [driveTrainTest.py](#) (TestModels/)
- [revoluteJointPrismaticJointTest.py](#) (TestModels/)
- [carRollingDiscTest.py](#) (TestModels/)
- ...

### 7.5.2 ObjectJointRevoluteZ

A revolute joint in 3D; constrains the position of two rigid body markers and the rotation about two axes, while the joint z-rotation axis (defined in local coordinates of marker 0 / joint J0 coordinates) can freely rotate. An additional local rotation (rotationMarker) can be used to transform the markers' coordinate systems into the joint coordinate system. For easier definition of the joint, use the exudyn.rigidbodyUtilities function AddRevoluteJoint(...), [Section 5.13](#), for two rigid bodies (or ground).

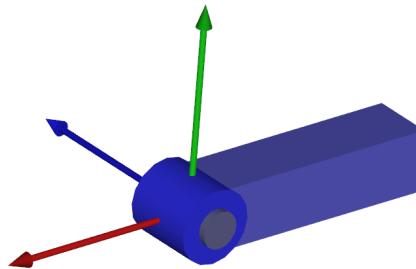


Figure 7.3: Example of RevoluteJointZ

#### Additional information for ObjectJointRevoluteZ:

- The Object has the following types = Connector, Constraint
- Requested marker type = Position + Orientation
- Short name for Python = **RevoluteJointZ**
- Short name for Python (visualization object) = **VRevoluteJointZ**

The item **ObjectJointRevoluteZ** with type = 'JointRevoluteZ' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex	2	[ MAXINT, MAX-INT ]	list of markers used in connector
rotationMarker0	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	local rotation matrix for marker $m_0$ ; translation and rotation axes for marker $m_0$ are defined in the local body coordinate system and additionally transformed by rotation-Marker0
rotationMarker1	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	local rotation matrix for marker $m_1$ ; translation and rotation axes for marker $m_1$ are defined in the local body coordinate system and additionally transformed by rotation-Marker1
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectJointRevoluteZ			parameters for visualization of item

The item VObjectJointRevoluteZ has the following parameters:

<b>Name</b>	<b>type</b>	<b>size</b>	<b>default value</b>	<b>description</b>
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
axisRadius	float		0.1	radius of joint axis to draw
axisLength	float		0.4	length of joint axis to draw
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==1, use default color

### 7.5.2.1 DESCRIPTION of ObjectJointRevoluteZ:

**Information on input parameters:**

<b>input parameter</b>	<b>symbol</b>	<b>description see tables above</b>
markerNumbers	$[m0, m1]^T$	
rotationMarker0	$^{m0, J0} \mathbf{A}$	
rotationMarker1	$^{m1, J1} \mathbf{A}$	

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

<b>output variable</b>	<b>symbol</b>	<b>description</b>
Position	${}^0 \mathbf{p}_{m0}$	current global position of position marker $m0$
Velocity	${}^0 \mathbf{v}_{m0}$	current global velocity of position marker $m0$
DisplacementLocal	${}^{J0} \Delta \mathbf{p}$	relative displacement in local joint0 coordinates; uses local J0 coordinates even for spherical joint configuration
VelocityLocal	${}^{J0} \Delta \mathbf{v}$	relative translational velocity in local joint0 coordinates
Rotation	${}^{J0} \boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2]^T$	relative rotation parameters (Tait Bryan Rxyz); if all axes are fixed, this output represents the rotational drift; for a revolute joint, it contains the rotation of this axis
AngularVelocityLocal	${}^{J0} \Delta \boldsymbol{\omega}$	relative angular velocity in local joint0 coordinates; if all axes are fixed, this output represents the angular velocity constraint error; for a revolute joint, it contains the angular velocity of this axis
ForceLocal	${}^{J0} \mathbf{f}$	joint force in local J0 coordinates
TorqueLocal	${}^{J0} \mathbf{m}$	joint torque in local J0 coordinates; depending on joint configuration, the result may not be the according torque vector

### 7.5.2.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker m0
marker m0 orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker m0
joint J0 orientation	${}^{0,J0}\mathbf{A} = {}^{0,m0}\mathbf{A} \cdot {}^{m0,J0}\mathbf{A}$	joint J0 rotation matrix
joint J0 orientation vectors	${}^{0,J0}\mathbf{A} = [{}^0\mathbf{t}_{x0}, {}^0\mathbf{t}_{y0}, {}^0\mathbf{t}_{z0}]^T$	orientation vectors (represent local x, y, and z axes) in global coordinates, used for definition of constraint equations
marker m1 position	${}^0\mathbf{p}_{m1}$	accordingly
marker m1 orientation	${}^{0,m1}\mathbf{A}$	current rotation matrix provided by marker m1
joint J1 orientation	${}^{0,J1}\mathbf{A} = {}^{0,m1}\mathbf{A} \cdot {}^{m1,J1}\mathbf{A}$	joint J1 rotation matrix
joint J1 orientation vectors	${}^{0,J1}\mathbf{A} = [{}^0\mathbf{t}_{x1}, {}^0\mathbf{t}_{y1}, {}^0\mathbf{t}_{z1}]^T$	orientation vectors (represent local x, y, and z axes) in global coordinates, used for definition of constraint equations
marker m0 velocity	${}^0\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
marker m1 velocity	${}^0\mathbf{v}_{m1}$	accordingly
marker m0 velocity	${}^b\boldsymbol{\omega}_{m0}$	current local angular velocity vector provided by marker m0
marker m1 velocity	${}^b\boldsymbol{\omega}_{m1}$	current local angular velocity vector provided by marker m1
Displacement	${}^0\Delta\mathbf{p} = {}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0}$	used, if all translational axes are constrained
Velocity	${}^0\Delta\mathbf{v} = {}^0\mathbf{v}_{m1} - {}^0\mathbf{v}_{m0}$	used, if all translational axes are constrained (velocity level)
DisplacementLocal	${}^{J0}\Delta\mathbf{p}$	$({}^{0,m0}\mathbf{A} \cdot {}^{m0,J0}\mathbf{A})^T {}^0\Delta\mathbf{p}$
VelocityLocal	${}^{J0}\Delta\mathbf{v}$	$({}^{0,m0}\mathbf{A} \cdot {}^{m0,J0}\mathbf{A})^T {}^0\Delta\mathbf{v}$ ... note that this is the global relative velocity projected into the local J0 coordinate system
AngularVelocityLocal	${}^{J0}\Delta\boldsymbol{\omega}$	$({}^{0,m0}\mathbf{A} \cdot {}^{m0,J0}\mathbf{A})^T ({}^{0,m1}\mathbf{A} \cdot {}^{m1}\boldsymbol{\omega} - {}^{0,m0}\mathbf{A} \cdot {}^{m0}\boldsymbol{\omega})$
algebraic variables	$\mathbf{z} = [\lambda_0, \dots, \lambda_5]^T$	vector of algebraic variables (Lagrange multipliers) according to the algebraic equations

### 7.5.2.3 Connector constraint equations

**Equations for translational part (activeConnector = True) :**

The translational index 3 constraints read,

$${}^0\Delta\mathbf{p} = \mathbf{0} \quad (7.163)$$

and the translational index 2 constraints read

$${}^0\Delta\mathbf{v} = \mathbf{0} \quad (7.164)$$

**Equations for rotational part (activeConnector = True) :**

Note that the axes are always given in global coordinates, compare the table in [Section 7.5.2.2](#). The index 3

constraint equations read

$$\lambda_3 = 0 \quad (7.165)$$

$${}^0\mathbf{t}_{x0}^T {}^0\mathbf{t}_{y1} = 0 \quad (7.166)$$

$${}^0\mathbf{t}_{x0}^T {}^0\mathbf{t}_{z1} = 0 \quad (7.167)$$

The index 2 constraints follow from the derivative of Eq. (7.165) w.r.t., and are given in the C++ code. if `activeConnector = False`,

$$\mathbf{z} = \mathbf{0} \quad (7.168)$$


---

#### 7.5.2.4 MINI EXAMPLE for ObjectJointRevoluteZ

```
#example with rigid body at [0,0,0], with torsional load
nBody = mbs.AddNode(RigidRxyz())
oBody = mbs.AddObject(RigidBody(physicsMass=1, physicsInertia=[1,1,1,0,0,0],
                                nodeNumber=nBody))

mBody = mbs.AddMarker(MarkerNodeRigid(nodeNumber=nBody))
mGround = mbs.AddMarker(MarkerBodyRigid(bodyNumber=oGround,
                                         localPosition = [0,0,0]))
mbs.AddObject(RevoluteJointZ(markerNumbers = [mGround, mBody])) #rotation around ground
Z-axis

#torque around z-axis;
mbs.AddLoad(Torque(markerNumber = mBody, loadVector=[0,0,1]))

#assemble and solve system for default parameters
mbs.Assemble()
exu.SolveDynamic(mbs, exu.SimulationSettings())

#check result at default integration time
exudynTestGlobals.testResult = mbs.GetNodeOutput(nBody, exu.OutputVariableType.Rotation)
[2]
```

---

For examples on ObjectJointRevoluteZ see Examples and TestModels:

- [addRevoluteJoint.py](#) (Examples/)
- [rigidBodyTutorial3.py](#) (Examples/)
- [fourBarMechanism3D.py](#) (Examples/)
- [solutionViewerTest.py](#) (Examples/)
- [CMSEexampleCourse.py](#) (Examples/)
- [plotSensorTest.py](#) (TestModels/)
- [revoluteJointPrismaticJointTest.py](#) (TestModels/)
- [perf3DRigidBodies.py](#) (TestModels/)

### 7.5.3 ObjectJointPrismaticX

A prismatic joint in 3D; constrains the relative rotation of two rigid body markers and relative motion w.r.t. the joint  $y$  and  $z$  axes, allowing a relative motion along the joint  $x$  axis (defined in local coordinates of marker 0/joint J0 coordinates). An additional local rotation (rotationMarker) can be used to transform the markers' coordinate systems into the joint coordinate system. For easier definition of the joint, use the exudyn.rigidbodyUtilities function AddPrismaticJoint(...), [Section 5.13](#), for two rigid bodies (or ground).

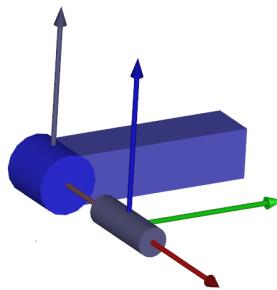


Figure 7.4: Example of PrismaticJointX

#### Additional information for ObjectJointPrismaticX:

- The Object has the following types = Connector, Constraint
- Requested marker type = Position + Orientation
- Short name for Python = **PrismaticJointX**
- Short name for Python (visualization object) = **VPrismaticJointX**

The item **ObjectJointPrismaticX** with type = 'JointPrismaticX' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex	2	[ MAXINT, MAX-INT ]	list of markers used in connector
rotationMarker0	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	local rotation matrix for marker $m_0$ ; translation and rotation axes for marker $m_0$ are defined in the local body coordinate system and additionally transformed by rotation-Marker0
rotationMarker1	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	local rotation matrix for marker $m_1$ ; translation and rotation axes for marker $m_1$ are defined in the local body coordinate system and additionally transformed by rotation-Marker1
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectJointPrismaticX			parameters for visualization of item

The item VObjectJointPrismaticX has the following parameters:

<b>Name</b>	<b>type</b>	<b>size</b>	<b>default value</b>	<b>description</b>
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
axisRadius	float		0.1	radius of joint axis to draw
axisLength	float		0.4	length of joint axis to draw
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==1, use default color

### 7.5.3.1 DESCRIPTION of ObjectJointPrismaticX:

#### Information on input parameters:

<b>input parameter</b>	<b>symbol</b>	<b>description see tables above</b>
markerNumbers	$[m0, m1]^T$	
rotationMarker0	$^{m0, J0} \mathbf{A}$	
rotationMarker1	$^{m1, J1} \mathbf{A}$	

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

<b>output variable</b>	<b>symbol</b>	<b>description</b>
Position	${}^0 \mathbf{p}_{m0}$	current global position of position marker $m0$
Velocity	${}^0 \mathbf{v}_{m0}$	current global velocity of position marker $m0$
DisplacementLocal	${}^{J0} \Delta \mathbf{p}$	relative displacement in local joint0 coordinates; uses local J0 coordinates even for spherical joint configuration
VelocityLocal	${}^{J0} \Delta \mathbf{v}$	relative translational velocity in local joint0 coordinates
Rotation	${}^{J0} \boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2]^T$	relative rotation parameters (Tait Bryan Rxyz); if all axes are fixed, this output represents the rotational drift; for a revolute joint, it contains the rotation of this axis
AngularVelocityLocal	${}^{J0} \Delta \boldsymbol{\omega}$	relative angular velocity in local joint0 coordinates; if all axes are fixed, this output represents the angular velocity constraint error; for a revolute joint, it contains the angular velocity of this axis
ForceLocal	${}^{J0} \mathbf{f}$	joint force in local J0 coordinates
TorqueLocal	${}^{J0} \mathbf{m}$	joint torque in local J0 coordinates; depending on joint configuration, the result may not be the according torque vector

### 7.5.3.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker m0
marker m0 orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker m0
joint J0 orientation	${}^{0,J0}\mathbf{A} = {}^{0,m0}\mathbf{A} {}^{m0,J0}\mathbf{A}$	joint J0 rotation matrix
joint J0 orientation vectors	${}^{0,J0}\mathbf{A} = [{}^0\mathbf{t}_{x0}, {}^0\mathbf{t}_{y0}, {}^0\mathbf{t}_{z0}]^T$	orientation vectors (represent local x, y, and z axes) in global coordinates, used for definition of constraint equations
marker m1 position	${}^0\mathbf{p}_{m1}$	accordingly
marker m1 orientation	${}^{0,m1}\mathbf{A}$	current rotation matrix provided by marker m1
joint J1 orientation	${}^{0,J1}\mathbf{A} = {}^{0,m1}\mathbf{A} {}^{m1,J1}\mathbf{A}$	joint J1 rotation matrix
joint J1 orientation vectors	${}^{0,J1}\mathbf{A} = [{}^0\mathbf{t}_{x1}, {}^0\mathbf{t}_{y1}, {}^0\mathbf{t}_{z1}]^T$	orientation vectors (represent local x, y, and z axes) in global coordinates, used for definition of constraint equations
marker m0 velocity	${}^0\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
marker m1 velocity	${}^0\mathbf{v}_{m1}$	accordingly
marker m0 velocity	${}^b\boldsymbol{\omega}_{m0}$	current local angular velocity vector provided by marker m0
marker m1 velocity	${}^b\boldsymbol{\omega}_{m1}$	current local angular velocity vector provided by marker m1
Displacement	${}^0\Delta\mathbf{p} = {}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0}$	used, if all translational axes are constrained
DisplacementLocal	${}^{J0}\Delta\mathbf{p}$	$({}^{0,m0}\mathbf{A} {}^{m0,J0}\mathbf{A})^T {}^0\Delta\mathbf{p}$
VelocityLocal	${}^{J0}\Delta\mathbf{v}$	$({}^{0,m0}\mathbf{A} {}^{m0,J0}\mathbf{A})^T {}^0\Delta\mathbf{v}$ ... note that this is the global relative velocity projected into the local J0 coordinate system
AngularVelocityLocal	${}^{J0}\Delta\boldsymbol{\omega}$	$({}^{0,m0}\mathbf{A} {}^{m0,J0}\mathbf{A})^T ({}^{0,m1}\mathbf{A} {}^{m1}\boldsymbol{\omega} - {}^{0,m0}\mathbf{A} {}^{m0}\boldsymbol{\omega})$
algebraic variables	$\mathbf{z} = [\lambda_0, \dots, \lambda_5]^T$	vector of algebraic variables (Lagrange multipliers) according to the algebraic equations

### 7.5.3.3 Connector constraint equations

Equations for translational part (**activeConnector = True**) :

The two translational index 3 constraints for a free motion along the local x-axis read (in the coordinate system J0),

$$\begin{aligned} {}^{J0}\mathbf{p}_{y,m1} - {}^{J0}\mathbf{p}_{y,m0} &= \mathbf{0} \\ {}^{J0}\mathbf{p}_{z,m1} - {}^{J0}\mathbf{p}_{z,m0} &= \mathbf{0} \end{aligned} \quad (7.169)$$

and the translational index 2 constraints read

$$\begin{aligned} {}^{J0}\mathbf{v}_{y,m1} - {}^{J0}\mathbf{v}_{y,m0} &= \mathbf{0} \\ {}^{J0}\mathbf{v}_{z,m1} - {}^{J0}\mathbf{v}_{z,m0} &= \mathbf{0} \end{aligned} \quad (7.170)$$

**Equations for rotational part (`activeConnector = True`) :**

Note that the axes are always given in global coordinates, compare the table in [Section 7.5.3.2](#). The index 3 constraint equations read

$${}^0\mathbf{t}_{z0}^T {}^0\mathbf{t}_{y1} = 0 \quad (7.171)$$

$${}^0\mathbf{t}_{z0}^T {}^0\mathbf{t}_{x1} = 0 \quad (7.172)$$

$${}^0\mathbf{t}_{x0}^T {}^0\mathbf{t}_{y1} = 0 \quad (7.173)$$

The index 2 constraints follow from the derivative of Eq. (7.171) w.r.t., and are given in the C++ code. if `activeConnector = False`,

$$\mathbf{z} = \mathbf{0} \quad (7.174)$$

---

For examples on ObjectJointPrismaticX see Examples and TestModels:

- [`revoluteJointPrismaticJointTest.py`](#) (TestModels/)

## 7.5.4 ObjectJointSpherical

A spherical joint, which constrains the relative translation between two position based markers.

**Additional information for ObjectJointSpherical:**

- The Object has the following types = Connector, Constraint
- Requested marker type = Position
- Short name for Python = **SphericalJoint**
- Short name for Python (visualization object) = **VsphericalJoint**

The item **ObjectJointSpherical** with type = 'JointSpherical' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex	2	[ MAXINT, MAX-INT ]	list of markers used in connector; $m_1$ is the moving coin rigid body and $m_0$ is the marker for the ground body, which use the localPosition=[0,0,0] for this marker!
constrainedAxes	ArrayIndex	3	[1,1,1]	flag, which determines which translation (0,1,2) and rotation (3,4,5) axes are constrained; for $j_i$ , two values are possible: 0=free axis, 1=constrained axis
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectJointSpherical			parameters for visualization of item

The item **VObjectJointSpherical** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
jointRadius	float		0.1	radius of joint to draw
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R== -1, use default color

---

### 7.5.4.1 DESCRIPTION of ObjectJointSpherical:

**Information on input parameters:**

input parameter	symbol	description see tables above
markerNumbers	$[m_0, m_1]^T$	
constrainedAxes	$j = [j_0, \dots, j_2]$	

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

output variable	symbol	description
Position	${}^0\mathbf{p}_{m0}$	current global position of position marker $m0$
Velocity	${}^0\mathbf{v}_{m0}$	current global velocity of position marker $m0$
Displacement	${}^0\Delta\mathbf{p} = {}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0}$	constraint drift or relative motion, if not all axes fixed
Force	${}^0\mathbf{f}$	joint force in global coordinates

#### 7.5.4.2 Definition of quantities

intermediate variables	symbol	description
marker $m0$ position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker $m0$
marker $m1$ position	${}^0\mathbf{p}_{m1}$	current global position which is provided by marker $m1$
marker $m0$ velocity	${}^0\mathbf{v}_{m0}$	current global velocity which is provided by marker $m0$
marker $m1$ velocity	${}^0\mathbf{v}_{m1}$	current global velocity which is provided by marker $m1$
relative velocity	${}^0\Delta\mathbf{v} = {}^0\mathbf{v}_{m1} - {}^0\mathbf{v}_{m0}$	constraint velocity error, or relative velocity if not all axes fixed
algebraic variables	$\mathbf{z} = [\lambda_0, \dots, \lambda_2]^T$	vector of algebraic variables (Lagrange multipliers) according to the algebraic equations

#### 7.5.4.3 Connector constraint equations

**activeConnector = True:** If  $[j_0, \dots, j_2] = [1, 1, 1]^T$ , meaning that all translational coordinates are fixed, the translational index 3 constraints read

$${}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0} = \mathbf{0} \quad (7.175)$$

and the translational index 2 constraints read

$${}^0\mathbf{v}_{m1} - {}^0\mathbf{v}_{m0} = \mathbf{0} \quad (7.176)$$

If  $[j_0, \dots, j_2] \neq [1, 1, 1]^T$ , meaning that at least one translational coordinate is free, the translational index 3 constraints read for every component  $k \in [0, 1, 2]$  of the vector  ${}^0\Delta\mathbf{p}$

$${}^0\Delta p_k = 0 \quad \text{if } j_k = 1 \quad \text{and} \quad (7.177)$$

$$\lambda_k = 0 \quad \text{if } j_k = 0 \quad (7.178)$$

$$(7.179)$$

and the translational index 2 constraints read for every component  $k \in [0, 1, 2]$  of the vector  ${}^0\Delta\mathbf{v}$

$${}^0\Delta v_k = 0 \quad \text{if } j_k = 1 \quad \text{and} \quad (7.180)$$

$$\lambda_k = 0 \quad \text{if } j_k = 0 \quad (7.181)$$

$$(7.182)$$

```
activeConnector = False:
```

**z = 0**

(7.183)

---

For examples on ObjectJointSpherical see Examples and TestModels:

- [NGsolvePostProcessingStresses.py](#) (Examples/)
- [objectFFRFreducedOrderNetgen.py](#) (Examples/)
- [sliderCrank3DwithANCFbeltDrive.py](#) (Examples/)
- [driveTrainTest.py](#) (TestModels/)
- [genericJointUserFunctionTest.py](#) (TestModels/)
- [objectFFRFreducedOrderAccelerations.py](#) (TestModels/)
- [objectFFRFreducedOrderStressModesTest.py](#) (TestModels/)
- [objectFFRFreducedOrderTest.py](#) (TestModels/)
- [objectFFRFTTest.py](#) (TestModels/)
- [objectFFRFTTest2.py](#) (TestModels/)
- [perfObjectFFRFreducedOrder.py](#) (TestModels/)
- [sphericalJointTest.py](#) (TestModels/)
- [superElementRigidJointTest.py](#) (TestModels/)

### 7.5.5 ObjectJointRollingDisc

A joint representing a rolling rigid disc (marker 1) on a flat surface (marker 0, ground body) in global  $x$ - $y$  plane. The constraint is based on an idealized rolling formulation with no slip. The constraints works for discs as long as the disc axis and the plane normal vector are not parallel. It must be assured that the disc has contact to ground in the initial configuration (adjust z-position of body accordingly). The ground body can be a rigid body which is moving. In this case, the flat surface is assumed to be in the  $x$ - $y$ -plane at  $z = 0$ . Note that the rolling body must have the reference point at the center of the disc. NOTE: the case of a moving ground body needs to be tested further, check your results!

**Additional information for ObjectJointRollingDisc:**

- The Object has the following types = Connector, Constraint
- Requested marker type = Position + Orientation
- **Short name for Python** = **RollingDiscJoint**
- **Short name for Python (visualization object)** = **VRollingDiscJoint**

The item **ObjectJointRollingDisc** with type = 'JointRollingDisc' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex	2	[ MAXINT, MAX-INT ]	list of markers used in connector; $m_0$ represents the ground and $m_1$ represents the rolling body, which has its reference point (=local position [0,0,0]) at the disc center point
constrainedAxes	ArrayIndex	3	[1,1,1]	flag, which determines which constraints are active, in which $j_0, j_1$ represent the tangential motion and $j_2$ represents the normal (contact) direction
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
discRadius	PReal		0	defines the disc radius
planeNormal	Vector3D		[0,0,1]	normal to the contact / rolling plane; cannot be changed at the moment
visualization	VObjectJointRollingDisc			parameters for visualization of item

The item **VObjectJointRollingDisc** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
discWidth	float		0.1	width of disc for drawing
color	Float4		[-1,-1,-1,-1.]	RGBA connector color; if R==1, use default color

---

### 7.5.5.1 DESCRIPTION of ObjectJointRollingDisc:

**Information on input parameters:**

input parameter	symbol	description see tables above
markerNumbers	$[m0, m1]^T$	
constrainedAxes	$j = [j_0, \dots, j_2]$	

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

output variable	symbol	description
Position	${}^0\mathbf{p}_G$	current global position of contact point between rolling disc and ground
Velocity	${}^0\mathbf{v}_{trail}$	current velocity of the trail (according to motion of the contact point along the trail!) in global coordinates; this is not the velocity of the contact point!
ForceLocal	${}^{J1}\mathbf{f} = {}^0[f_0, f_1, f_2]^T = [-\mathbf{z}^T {}^0\mathbf{w}_{lat}, -\mathbf{z}^T {}^0\mathbf{w}_2, -\mathbf{z}^T {}^0\mathbf{v}_{PN}]^T$	contact forces acting on disc, in special $J1$ joint coordinates, $f_0$ being the lateral force (parallel to ground plane), $f_1$ being the longitudinal force and $f_2$ being the normal force
RotationMatrix	${}^{0,J1}\mathbf{A} = [{}^0\mathbf{w}_{lat}, {}^0\mathbf{w}_2, {}^0\mathbf{v}_{PN}]^T$	transformation matrix of special joint $J1$ coordinates to global coordinates

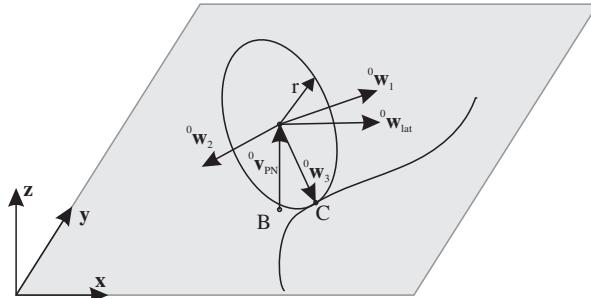
### 7.5.5.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^0\mathbf{p}_{m0}$	current global position of marker $m0$ ; needed only if body $m0$ is not a ground body
marker m0 orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker $m0$ (assumed to be rigid body)
marker m0 velocity	${}^0\mathbf{v}_{m0}$	current global velocity which is provided by marker $m0$ (assumed to be rigid body)
marker m0 angular velocity	${}^0\boldsymbol{\omega}_{m0}$	current angular velocity vector provided by marker $m0$ (assumed to be rigid body)
marker m1 position	${}^0\mathbf{p}_{m1}$	center of disc
marker m1 orientation	${}^{0,m1}\mathbf{A}$	current rotation matrix provided by marker $m1$
marker m1 velocity	${}^0\mathbf{v}_{m1}$	accordingly
marker m1 angular velocity	${}^0\boldsymbol{\omega}_{m1}$	current angular velocity vector provided by marker $m1$
ground normal vector	${}^0\mathbf{v}_{PN}$	normalized normal vector to the ground plane, currently $[0,0,1]$
ground position B	${}^0\mathbf{p}_B$	disc center point projected on ground in plane normal ( $z$ -direction, $z = 0$ )

ground position C	${}^0\mathbf{p}_C$	contact point of disc with ground in global coordinates
ground velocity C	${}^0\mathbf{v}_{Cm1}$	velocity of disc (marker 1) at ground contact point (must be zero if ground does not move)
ground velocity C	${}^0\mathbf{v}_{Cm2}$	velocity of ground (marker 0) at ground contact point (is always zero if ground does not move)
wheel axis vector	${}^0\mathbf{w}_1 = {}^{0,m1}\mathbf{A} \cdot [1, 0, 0]^T$	normalized disc axis vector, currently $[1, 0, 0]^T$ in local coordinates
longitudinal vector	${}^0\mathbf{w}_2$	vector in longitudinal (motion) direction
lateral vector	${}^0\mathbf{w}_{lat} = {}^0\mathbf{v}_{PN} \times {}^0\mathbf{w}_2$	vector in lateral direction, parallel to ground plane
contact point vector	${}^0\mathbf{w}_3$	normalized vector from disc center point in direction of contact point C
D1 transformation matrix	${}^{0,D1}\mathbf{A} = [{}^0\mathbf{w}_1, {}^0\mathbf{w}_2, {}^0\mathbf{w}_3]^T$	transformation of special disc coordinates D1 to global coordinates
algebraic variables	$\mathbf{z} = [\lambda_0, \lambda_1, \lambda_2]^T$	vector of algebraic variables (Lagrange multipliers) according to the algebraic equations

### 7.5.5.3 Geometric relations

The main geometrical setup is shown in the following figure:



First, the contact point  ${}^0\mathbf{p}_C$  must be computed. With the helper vector,

$${}^0\mathbf{x} = {}^0\mathbf{w}_1 \times {}^0\mathbf{v}_{PN} \quad (7.184)$$

we obtain a disc coordinate system, representing the longitudinal direction,

$${}^0\mathbf{w}_2 = \frac{1}{|{}^0\mathbf{x}|} {}^0\mathbf{x} \quad (7.185)$$

and the vector to the contact point,

$${}^0\mathbf{w}_3 = {}^0\mathbf{w}_1 \times {}^0\mathbf{w}_2 \quad (7.186)$$

The contact point C can be computed from

$${}^0\mathbf{p}_C = {}^0\mathbf{p}_{m1} + r \cdot {}^0\mathbf{w}_3 \quad (7.187)$$

The velocity of the contact point at the disc is computed from,

$${}^0\mathbf{v}_{Cm1} = {}^0\mathbf{v}_{m1} + {}^0\omega_{m1} \times (r \cdot {}^0\mathbf{w}_3) \quad (7.188)$$

If marker 0 body is (moving) rigid body instead of a ground body, the contact point  $C$  is reconstructed in body of marker 0,

$${}^{m0}\mathbf{p}_C = {}^{m0,0}\mathbf{A}({}^0\mathbf{p}_C - {}^0\mathbf{p}_{m0}) \quad (7.189)$$

The velocity of the contact point at the marker 0 body reads

$${}^0\mathbf{v}_{Cm0} = {}^0\mathbf{v}_{m0} + {}^0\boldsymbol{\omega}_{m0} \times ({}^{0,m0}\mathbf{A} {}^{m0}\mathbf{p}_C) \quad (7.190)$$

#### 7.5.5.4 Connector constraint equations

**activeConnector = True:**

The non-holonomic, index 2 constraints for the tangential and normal contact follow from (an index 3 formulation would be possible, but is not implemented yet because of mixing different jacobians)

$$\begin{bmatrix} {}^0\mathbf{v}_{Cm1,x} \\ {}^0\mathbf{v}_{Cm1,y} \\ {}^0\mathbf{v}_{Cm1,z} \end{bmatrix} - \begin{bmatrix} {}^0\mathbf{v}_{Cm0,x} \\ {}^0\mathbf{v}_{Cm0,y} \\ {}^0\mathbf{v}_{Cm0,z} \end{bmatrix} = \mathbf{0} \quad (7.191)$$

**activeConnector = False:**

$$\mathbf{z} = \mathbf{0} \quad (7.192)$$

For examples on ObjectJointRollingDisc see Examples and TestModels:

- [bicycleIftommBenchmark.py](#) (Examples/)
- [rollingCoinTest.py](#) (TestModels/)

## 7.5.6 ObjectJointRevolute2D

A revolute joint in 2D; constrains the absolute 2D position of two points given by PointMarkers or RigidMarkers

**Additional information for ObjectJointRevolute2D:**

- The Object has the following types = Connector, Constraint
- Requested marker type = Position
- **Short name for Python = RevoluteJoint2D**
- **Short name for Python (visualization object) = VRevoluteJoint2D**

The item **ObjectJointRevolute2D** with type = 'JointRevolute2D' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex		[ MAXINT, MAX-INT ]	list of markers used in connector
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectJointRevolute2D			parameters for visualization of item

The item **VObjectJointRevolute2D** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = radius of revolute joint; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R== -1, use default color

---

### 7.5.6.1 DESCRIPTION of ObjectJointRevolute2D:

---

For examples on ObjectJointRevolute2D see Examples and TestModels:

- [sliderCrank3DwithANCFbeltDrive2.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- [ANCF\\_slidingJoint2Drigid.py](#) (Examples/)
- [beltDriveReevingSystem.py](#) (Examples/)
- [finiteSegmentMethod.py](#) (Examples/)
- [geneticOptimizationSliderCrank.py](#) (Examples/)
- [reevingSystem.py](#) (Examples/)

- [rigid\\_pendulum.py](#) (Examples/)
- [SliderCrank.py](#) (Examples/)
- [sliderCrank3DwithANCFbeltDrive.py](#) (Examples/)
- [slidercrankWithMassSpring.py](#) (Examples/)
- [switchingConstraintsPendulum.py](#) (Examples/)
- ...
- [ANCFbeltDrive.py](#) (TestModels/)
- [ANCFgeneralContactCircle.py](#) (TestModels/)
- [ANCFoutputTest.py](#) (TestModels/)
- ...

### 7.5.7 ObjectJointPrismatic2D

A prismatic joint in 2D; allows the relative motion of two bodies, using two RigidMarkers; the vector  $\mathbf{t}_0 = \text{axisMarker0}$  is given in local coordinates of the first marker's (body) frame and defines the prismatic axis; the vector  $\mathbf{n}_1 = \text{normalMarker1}$  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

$$(\mathbf{p}_1 - \mathbf{p}_0)^T \cdot \mathbf{A}_1 \cdot \mathbf{n}_1 = 0 \quad (7.193)$$

$$(\mathbf{A}_0 \cdot \mathbf{t}_0)^T \cdot \mathbf{A}_1 \cdot \mathbf{n}_1 = 0 \quad (7.194)$$

The lagrange multipliers follow for these two equations  $[\lambda_0, \lambda_1]$ , in which  $\lambda_0$  is the transverse force and  $\lambda_1$  is the torque in the joint.

#### Additional information for ObjectJointPrismatic2D:

- The Object has the following types = Connector, Constraint
- Requested marker type = Position + Orientation
- Short name for Python = PrismaticJoint2D
- Short name for Python (visualization object) = VPrismaticJoint2D

The item **ObjectJointPrismatic2D** with type = 'JointPrismatic2D' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex		[ MAXINT, MAX-INT ]	list of markers used in connector
axisMarker0	Vector3D		[1.,0.,0.]	direction of prismatic axis, given as a 3D vector in Marker0 frame
normalMarker1	Vector3D		[0.,1.,0.]	direction of normal to prismatic axis, given as a 3D vector in Marker1 frame
constrainRotation	Bool		True	flag, which determines, if the connector also constrains the relative rotation of the two objects; if set to false, the constraint will keep an algebraic equation set equal zero
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectJointPrismatic2D			parameters for visualization of item

The item **VObjectJointPrismatic2D** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = radius of revolute joint; size == -1.f means that default connector size is used

color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==1, use default color
-------	--------	--	-------------------	--

---

### 7.5.7.1 DESCRIPTION of ObjectJointPrismatic2D:

For examples on ObjectJointPrismatic2D see Examples and TestModels:

- [sliderCrank3DwithANCFbeltDrive2.py](#) (Examples/)
- [geneticOptimizationSliderCrank.py](#) (Examples/)
- [PARTS\\_ATEs\\_moving.py](#) (TestModels/)
- [scissorPrismaticRevolute2D.py](#) (TestModels/)
- [sliderCrankFloatingTest.py](#) (TestModels/)

### 7.5.8 ObjectJointSliding2D

A specialized sliding joint (without rotation) in 2D between a Cable2D (marker1) and a position-based marker (marker0); the data coordinate x[0] provides the current index in slidingMarkerNumbers, and x[1] the local position in the cable element at the beginning of the timestep.

**Additional information for ObjectJointSliding2D:**

- The Object has the following types = Connector, Constraint
- Requested marker type = \_None
- Requested node type = GenericData
- **Short name for Python** = SlidingJoint2D
- **Short name for Python (visualization object)** = VSlidingJoint2D

The item **ObjectJointSliding2D** with type = 'JointSliding2D' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex		[ MAXINT, MAXINT ]	marker m0: position or rigid body marker of mass point or rigid body; marker m1: updated marker to Cable2D element, where the sliding joint currently is attached to; must be initialized with an appropriate (global) marker number according to the starting position of the sliding object; this marker changes with time (PostNewtonStep)
slidingMarkerNumbers	ArrayMarkerIndex		[]	these markers are used to update marker m1, if the sliding position exceeds the current cable's range; the markers must be sorted such that marker $m_{si}$ at $x=cable(i).length$ is equal to marker(i+1) at $x=0$ of cable(i+1)
slidingMarkerOffsets	Vector		[]	this list contains the offsets of every sliding object (given by slidingMarkerNumbers) w.r.t. to the initial position (0): marker m0: offset=0, marker m1: offset=Length(cable0), marker m2: offset=Length(cable0)+Length(cable1), ...
nodeNumber	NodeIndex		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	True: uses a formulation with 3 (+1) equations, including the force in sliding direction to be zero; forces in global coordinates, only index 3; False: use local formulation, which only needs 2 (+1) equations and can be used with index 2 formulation

constrainRotation	Bool	False	True: add constraint on rotation of marker m0 relative to slope (if True, marker m0 must be a rigid body marker); False: marker m0 body can rotate freely
axialForce	Real	0	ONLY APPLIES if classicalFormulation==True; axialForce represents an additional sliding force acting between beam and marker m0 body in axial (beam) direction; this force can be used to drive a body on a beam, but can only be changed with user functions.
activeConnector	Bool	True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectJointSliding2D		parameters for visualization of item

The item VObjectJointSliding2D has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = radius of revolute joint; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R== -1, use default color

### 7.5.8.1 DESCRIPTION of ObjectJointSliding2D:

Information on input parameters:

input parameter	symbol	description see tables above
markerNumbers	$[m_0, m_1]^T$	
slidingMarkerNumbers	$[m_{s0}, \dots, m_{sn}]^T$	
slidingMarkerOffsets	$[d_{s0}, \dots, d_{sn}]$	
nodeNumber	$n_{GD}$	
axialForce	$f_{ax}$	

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

output variable	symbol	description
Position		position vector of joint given by marker0
Velocity		velocity vector of joint given by marker0

SlidingCoordinate		global sliding coordinate along all elements; the maximum sliding coordinate is equivalent to the reference lengths of all sliding elements
Force		joint force vector (3D)

### 7.5.8.2 Definition of quantities

intermediate variables	symbol	description
data node	$\mathbf{x} = [x_{data0}, x_{data1}]^T$	coordinates of node with node number $n_{GD}$
data coordinate 0	$x_{data0}$	the current index in slidingMarkerNumbers
data coordinate 1	$x_{data1}$	the global sliding coordinate (ranging from 0 to the total length of all sliding elements) at <b>start-of-step</b> - beginning of the timestep
marker m0 position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker m0
marker m0 velocity	${}^0\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
marker m0 orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker m0 (assumed to be rigid body)
marker m0 angular velocity	${}^0\boldsymbol{\omega}_{m0}$	current angular velocity vector provided by marker m0 (assumed to be rigid body)
cable coordinates	$\mathbf{q}_{ANCF,m1}$	current coordinates of the ANCF cable element with the current marker $m1$ is referring to
sliding position	${}^0\mathbf{r}_{ANCF} = \mathbf{S}(s_{el})\mathbf{q}_{ANCF,m1}$	current global position at the ANCF cable element, evaluated at local sliding position $s_{el}$
sliding position slope	${}^0\mathbf{r}'_{ANCF} = \mathbf{S}'(s_{el})\mathbf{q}_{ANCF,m1} = [r'_0, r'_1]^T$	current global slope vector of the ANCF cable element, evaluated at local sliding position $s_{el}$
sliding velocity	${}^0\mathbf{v}_{ANCF} = \mathbf{S}(s_{el})\dot{\mathbf{q}}_{ANCF,m1}$	current global velocity at the ANCF cable element, evaluated at local sliding position $s_{el}$ ( $s_{el}$ not differentiated!!!)
sliding velocity slope	${}^0\mathbf{v}'_{ANCF} = \mathbf{S}'(s_{el})\dot{\mathbf{q}}_{ANCF,m1}$	current global slope velocity vector of the ANCF cable element, evaluated at local sliding position $s_{el}$
sliding normal vector	${}^0\mathbf{n} = [-r'_1, r'_0]$	2D normal vector computed from slope $\mathbf{r}' = {}^0\mathbf{r}'_{ANCF}$
sliding normal velocity vector	${}^0\dot{\mathbf{n}} = [-\dot{r}'_1, \dot{r}'_0]$	time derivative of 2D normal vector computed from slope velocity $\dot{\mathbf{r}'} = {}^0\dot{\mathbf{r}'}_{ANCF}$
algebraic coordinates	$\mathbf{z} = [\lambda_0, \lambda_1, s]^T$	algebraic coordinates composed of Lagrange multipliers $\lambda_0$ and $\lambda_1$ (in local cable coordinates: $\lambda_0$ 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 <b>relative to the start-of-step value</b> . 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

### 7.5.8.3 Geometric relations

Assume we have given the sliding coordinate  $s$  (e.g., as a guess of the Newton method or beginning of the time step). The element sliding coordinate (in the local coordinates of the current sliding element) is computed as

$$s_{el} = s + x_{data1} - d_{m1} = s_g - d_{m1}. \quad (7.195)$$

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

$${}^0\Delta\mathbf{p} = {}^0\mathbf{r}_{ANCF} - {}^0\mathbf{p}_{m0} \quad (7.196)$$

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

$${}^0\Delta\mathbf{v} = {}^0\dot{\mathbf{r}}_{ANCF} - {}^0\mathbf{v}_{m0} \quad (7.197)$$

### 7.5.8.4 Connector constraint equations (classicalFormulation=True)

The 2D sliding joint is implemented having 3 equations (4 if constrainRotation==True, see below), using the special algebraic coordinates  $\mathbf{z}$ . The algebraic equations read

$${}^0\Delta\mathbf{p} = \mathbf{0}, \quad \dots \text{two index 3 eqs, ensure sliding body stays at cable} \quad (7.198)$$

$$[\lambda_0, \lambda_1] \cdot {}^0\mathbf{r}'_{ANCF} - |{}^0\mathbf{r}'_{ANCF}| \cdot f_{ax} = 0, \quad \dots \text{one index 1 equ., ensure force in sliding dir.} = 0 \quad (7.199)$$

$$(7.200)$$

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

$$\mathbf{J}_{AE} = \begin{bmatrix} 0 & 0 & r'_0 \\ 0 & 0 & r'_1 \\ r'_0 & r'_1 & r''_0\lambda_0 + r''_1\lambda_1 \end{bmatrix} \quad (7.201)$$

$$\mathbf{J}_{ODE2} = \begin{bmatrix} -J_{pos,m0} & \mathbf{S}(s_{el}) \\ \mathbf{0}^T & [\lambda_0, \lambda_1] \cdot \mathbf{S}'(s_{el}) \end{bmatrix} \quad (7.202)$$

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

$$\lambda_0 = 0, \quad (7.203)$$

$$\lambda_1 = 0, \quad (7.204)$$

$$s = 0 \quad (7.205)$$

### 7.5.8.5 Connector constraint equations (`classicalFormulation=False`)

The 2D sliding joint is implemented having 3 equations (first equation is dummy and could be eliminated; 4 equations if `constrainRotation==True`, see below), using the special algebraic coordinates  $\mathbf{z}$ . The algebraic equations read

$$\lambda_0 = 0, \quad \dots \text{equation not necessary, but can be used for switching to other modes} \quad (7.206)$$

$${}^0\Delta\mathbf{p}^T {}^0\mathbf{n} = 0, \quad \dots \text{equation ensures that sliding body stays at cable centerline; index3} \quad (7.207)$$

$${}^0\Delta\mathbf{p}^T {}^0\mathbf{r}'_{ANCF} = 0. \quad \dots \text{resolves the sliding coordinate } s; \text{ index1 equation!} \quad (7.208)$$

In the index 2 case, the second equation reads

$${}^0\Delta\mathbf{v}^T {}^0\mathbf{n} + {}^0\Delta\mathbf{p}^T {}^0\dot{\mathbf{n}} = 0 \quad (7.209)$$

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

$$\lambda_0 = 0, \quad (7.210)$$

$$\lambda_1 = 0, \quad (7.211)$$

$$s = 0 \quad (7.212)$$

In case that `constrainRotation = True`, an additional constraint is added for the relative rotation between the slope of the cable and the orientation of marker m0 body. Assuming that the orientation of marker m0 is a 2D matrix (taking only  $x$  and  $y$  coordinates), the constraint reads

$${}^0\mathbf{r}'^T_{ANCF} {}^{0,m0}\mathbf{A} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 0 \quad (7.213)$$

The index 2 case follows straightforward to

$${}^0\mathbf{r}'^T_{ANCF} {}^{0,m0}\mathbf{A} \begin{bmatrix} 0 \\ 1 \end{bmatrix} + {}^0\mathbf{r}'^T_{ANCF} {}^{0,m0}\mathbf{A} {}^0\tilde{\omega}_{m0} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 0 \quad (7.214)$$

again assuming, that  ${}^0\tilde{\omega}_{m0}$  is only a  $2 \times 2$  matrix.

### 7.5.8.6 Post Newton Step

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

$$\begin{aligned} s_{el} < 0 &\rightarrow x_{data0} -= 1 \\ s_{el} > L &\rightarrow x_{data0} += 1 \end{aligned} \quad (7.215)$$

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,

$$x_{data1} += s. \quad (7.216)$$

For examples on ObjectJointSliding2D see Examples and TestModels:

- [`ANCF\_moving\_rigidbody.py`](#) (Examples/)
- [`ANCF\_slidingJoint2D.py`](#) (Examples/)
- [`ANCF\_slidingJoint2Drigid.py`](#) (Examples/)
- [`ANCF\_switchingSlidingJoint2D.py`](#) (Examples/)
- [`modelUnitTests.py`](#) (TestModels/)

### 7.5.9 ObjectJointALEMoving2D

A specialized axially moving joint (without rotation) in 2D between a ALE Cable2D (marker1) and a position-based marker (marker0); ALE=Arbitrary Lagrangian Eulerian; the data coordinate  $x[0]$  provides the current index in slidingMarkerNumbers, and the [ODE2](#) coordinate  $q[0]$  provides the (given) moving coordinate in the cable element.

**Additional information for ObjectJointALEMoving2D:**

- The Object has the following types = Connector, Constraint
- Requested marker type = \_None
- Requested node type: read detailed information of item
- **Short name for Python** = **ALEMovingJoint2D**
- **Short name for Python (visualization object)** = **VALEMovingJoint2D**

The item **ObjectJointALEMoving2D** with type = 'JointALEMoving2D' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex		[ MAXINT, MAX-INT ]	marker m0: position-marker of mass point or rigid body; marker m1: updated marker to ANCF Cable2D element, where the sliding joint currently is attached to; must be initialized with an appropriate (global) marker number according to the starting position of the sliding object; this marker changes with time (PostNewtonStep)
slidingMarkerNumbers	ArrayMarkerIndex		[]	a list of sn (global) marker numbers which are used to update marker1
slidingMarkerOffsets	Vector		[]	this list contains the offsets of every sliding object (given by slidingMarkerNumbers) w.r.t. to the initial position (0): marker0: offset=0, marker1: offset=Length(cable0), marker2: offset=Length(cable0)+Length(cable1), ...
slidingOffset	Real		0.	sliding offset list [SI:m]: a list of sn scalar offsets, which represent the (reference arc) length of all previous sliding cable elements
nodeNumbers	ArrayNodeIndex		[ MAXINT, MAX-INT ]	node number of NodeGenericData (GD) with one data coordinate and of Node-GenericODE2 (ALE) with one <a href="#">ODE2</a> coordinate
usePenaltyFormulation	Bool		False	flag, which determines, if the connector is formulated with penalty, but still using algebraic equations (IsPenaltyConnector() still false)
penaltyStiffness	Real		0.	penalty stiffness [SI:N/m] used if usePenaltyFormulation=True

activeConnector	Bool	True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectJointALEMoving2D		parameters for visualization of item

The item VObjectJointALEMoving2D has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = radius of revolute joint; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==1, use default color

### 7.5.9.1 DESCRIPTION of ObjectJointALEMoving2D:

Information on input parameters:

input parameter	symbol	description see tables above
markerNumbers	$[m_0, m_1]^T$	
slidingMarkerNumbers	$[m_{s0}, \dots, m_{sn}]^T$	
slidingMarkerOffsets	$[d_{s0}, \dots, d_{sn}]$	
slidingOffset	$s_{off}$	
nodeNumbers	$[n_{GD}, n_{ALE}]$	
penaltyStiffness	$k$	

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

output variable	symbol	description
Position	${}^0\mathbf{p}_{m0}$	current global position of position marker $m_0$
Velocity	${}^0\mathbf{v}_{m0}$	current global velocity of position marker $m_0$
SlidingCoordinate	$s_g = q_{ALE} + s_{off}$	current value of the global sliding ALE coordinate, including offset; note that reference coordinate of $q_{ALE}$ is ignored!
Coordinates	$[x_{data0}, q_{ALE}]^T$	provides two values: [0] = current sliding marker index, [1] = ALE sliding coordinate
Coordinates_t	$[\dot{q}_{ALE}]^T$	provides ALE sliding velocity
Force	$\mathbf{f}$	joint force vector (3D)

### 7.5.9.2 Definition of quantities

intermediate variables	symbol	description
generic data node	$\mathbf{x} = [x_{data0}]^T$	coordinates of node with node number $n_{GD}$
generic ODE2 node	$\mathbf{q} = [q_0]^T$	coordinates of node with node number $n_{ALE}$ , which is shared with all ALE-ANCF and ALE sliding joint objects
data coordinate	$x_{data0}$	the current index in slidingMarkerNumbers
ALE coordinate	$q_{ALE} = q_0$	current ALE coordinate (in fact this is the Eulerian coordinate in the ALE formulation); note that reference coordinate of $q_{ALE}$ is ignored!
marker m0 position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker m0
marker m0 velocity	${}^0\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
cable coordinates	$\mathbf{q}_{ANCF,m1}$	current coordinates of the ANCF cable element with the current marker $m1$ is referring to
sliding position	${}^0\mathbf{r}_{ANCF} = \mathbf{S}(s_{el})\mathbf{q}_{ANCF,m1}$	current global position at the ANCF cable element, evaluated at local sliding position $s_{el}$
sliding position slope	${}^0\mathbf{r}'_{ANCF} = \mathbf{S}'(s_{el})\mathbf{q}_{ANCF,m1}$	current global slope vector of the ANCF cable element, evaluated at local sliding position $s_{el}$
sliding velocity	${}^0\mathbf{v}_{ANCF} = \mathbf{S}(s_{el})\dot{\mathbf{q}}_{ANCF,m1} + \dot{q}_{ALE} {}^0\mathbf{r}'_{ANCF}$	current global velocity at the ANCF cable element, evaluated at local sliding position $s_{el}$ , including convective term
sliding normal vector	${}^0\mathbf{n} = [-r'_1, r'_0]$	2D normal vector computed from slope $\mathbf{r}' = {}^0\mathbf{r}'_{ANCF}$
algebraic variables	$\mathbf{z} = [\lambda_0, \lambda_1]^T$	algebraic variables (Lagrange multipliers) according to the algebraic equations

### 7.5.9.3 Geometric relations

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

$$s_{el} = q_{ALE} + s_{off} - d_{m1} = s_g - d_{m1}. \quad (7.217)$$

For the description of the according quantities, see the description above. The distance  $d_{m1}$  is obtained from the `slidingMarkerOffsets` list, using the current (local) index  $x_{data0}$ . The vector (=difference; error) between the marker  $m0$  and the marker  $m1$  ( $=\mathbf{r}_{ANCF}$ ) positions reads

$${}^0\Delta\mathbf{p} = {}^0\mathbf{r}_{ANCF} - {}^0\mathbf{p}_{m0} \quad (7.218)$$

Note that  ${}^0\mathbf{p}_{m0}$  represents the current position of the marker  $m0$ , which could represent the midpoint of a mass sliding along the beam. The position  ${}^0\mathbf{r}_{ANCF}$  is computed from the beam represented by marker  $m1$ , using the local beam coordinate  $x = s_{el}$ . The marker and the according beam finite element changes during movement using the list `slidingMarkerNumbers` and the index is updated in the PostNewtonStep. The vector (=difference; error) between the marker  $m0$  and the marker  $m1$  velocities reads

$${}^0\Delta\mathbf{v} = {}^0\mathbf{v}_{ANCF} - {}^0\mathbf{v}_{m0} \quad (7.219)$$

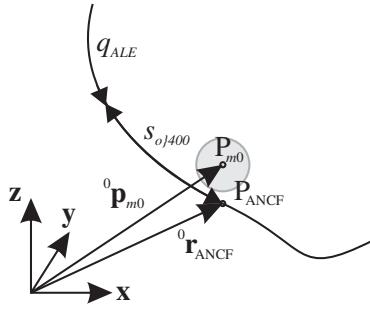


Figure 7.5: Geometrical relations for ALE sliding joint.

#### 7.5.9.4 Connector constraint equations

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

$${}^0\Delta\mathbf{p} = 0 \quad (7.220)$$

Note that the Lagrange multipliers  $[\lambda_0, \lambda_1]^T$  are the global forces in the joint. In the index 2 case the algebraic equations read

$${}^0\Delta\mathbf{v} = 0 \quad (7.221)$$

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

$${}^0\Delta\mathbf{p} - \frac{1}{k}\mathbf{z} = 0. \quad (7.222)$$

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

$$\lambda_0 = 0, \quad (7.223)$$

$$\lambda_1 = 0. \quad (7.224)$$

#### 7.5.9.5 Post Newton Step

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

$$\begin{aligned} s_{el} < 0 &\rightarrow x_{data0} -= 1 \\ s_{el} > L &\rightarrow x_{data0} += 1 \end{aligned} \quad (7.225)$$

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,

$$x_{data1} += s. \quad (7.226)$$

For examples on `ObjectJointALEMoving2D` see Examples and TestModels:

- [`ANCF\_moving\_rigidbody.py`](#) (Examples/)
- [`ANCFmovingRigidBodyTest.py`](#) (TestModels/)

## 7.6 Objects (Connector)

A Connector is a special Object, which links two or more markers. A Connector which is not a Constraint, is a force element (e.g., spring-damper) or a penalty based joint.

### 7.6.1 ObjectConnectorSpringDamper

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

**Additional information for ObjectConnectorSpringDamper:**

- The Object has the following types = Connector
- Requested marker type = Position
- **Short name for Python = SpringDamper**
- **Short name for Python (visualization object) = VSpringDamper**

The item **ObjectConnectorSpringDamper** with type = 'ConnectorSpringDamper' has the following parameters:

Name	type	size	default value	description
name	String		"	connector's unique name
markerNumbers	ArrayMarkerIndex		[ MAXINT, MAX-INT ]	list of markers used in connector
referenceLength	PReal		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	Real		0.	added constant force [SI:N] of spring; scalar force; f=1 is equivalent to reducing initialLength by 1/stiffness; f > 0: tension; f < 0: compression; can be used to model actuator force
velocityOffset	Real		0.	velocity offset [SI:m/s] of damper, being equivalent to time change of reference length
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
springForceUserFunction	PyFunctionMbsScalarIndexScalar5		0	A Python function which defines the spring force with parameters; the Python function will only be evaluated, if activeConnector is true, otherwise the SpringDamper is inactive; see description below
visualization	VObjectConnectorSpringDamper			parameters for visualization of item

The item VObjectConnectorSpringDamper has the following parameters:

Name	type	size	default value	description
------	------	------	---------------	-------------

show	Bool	True	set true, if item is shown in visualization and false if it is not shown
drawSize	float	-1.	drawing size = diameter of spring; size == -1.f means that default connector size is used
color	Float4	[-1.,-1.,-1.,-1.]	RGBA connector color; if R===-1, use default color

---

### 7.6.1.1 DESCRIPTION of ObjectConnectorSpringDamper:

Information on input parameters:

input parameter	symbol	description see tables above
markerNumbers	$[m0, m1]^T$	
referenceLength	$L_0$	
stiffness	$k$	
damping	$d$	
force	$f_a$	
velocityOffset	$\dot{L}_0$	
springForceUserFunction	$UF \in \mathbb{R}$	

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

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

### 7.6.1.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker m0
marker m1 position	${}^0\mathbf{p}_{m1}$	
marker m0 velocity	${}^0\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
marker m1 velocity	${}^0\mathbf{v}_{m1}$	
time derivative of distance	$\dot{L}$	$\Delta {}^0\mathbf{v}^T \mathbf{v}_f$

output variables	symbol	formula
Displacement	$\Delta {}^0\mathbf{p}$	${}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0}$
Velocity	$\Delta {}^0\mathbf{v}$	${}^0\mathbf{v}_{m1} - {}^0\mathbf{v}_{m0}$
Distance	$L$	$ \Delta {}^0\mathbf{p} $
Force	$f$	see below

### 7.6.1.3 Connector forces

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

$$\mathbf{v}_f = \frac{1}{L} \Delta^0 \mathbf{p} \quad (7.227)$$

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

$$f_{SD} = k \cdot (L - L_0) + d \cdot (\dot{L} - \dot{L}_0) + f_a \quad (7.228)$$

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

$$f_{SD} = \text{UF}(mbs, t, i_N, L - L_0, \dot{L} - \dot{L}_0, k, d, f_a) \quad (7.229)$$

and `iN` represents the itemNumber (=objectNumber). Note that, if `activeConnector = False`,  $f_{SD}$  is set to zero.

The vector of the spring-damper force applied at both markers finally reads

$$\mathbf{f} = f_{SD} \mathbf{v}_f \quad (7.230)$$

The virtual work of the connector force is computed from the virtual displacement

$$\delta \Delta^0 \mathbf{p} = \delta^0 \mathbf{p}_{m1} - \delta^0 \mathbf{p}_{m0}, \quad (7.231)$$

and the virtual work (not the transposed version here, because the resulting generalized forces shall be a column vector,

$$\delta W_{SD} = \mathbf{f} \delta \Delta^0 \mathbf{p} = (k \cdot (L - L_0) + d \cdot (\dot{L} - \dot{L}_0) + f_a) (\delta^0 \mathbf{p}_{m1} - \delta^0 \mathbf{p}_{m0})^T \mathbf{v}_f. \quad (7.232)$$

The generalized (elastic) forces thus result from

$$\mathbf{Q}_{SD} = \frac{\partial^0 \mathbf{p}}{\partial \mathbf{q}_{SD}^T} \mathbf{f}, \quad (7.233)$$

and read for the markers  $m0$  and  $m1$ ,

$$\mathbf{Q}_{SD,m0} = - (k \cdot (L - L_0) + d \cdot (\dot{L} - \dot{L}_0) + f_a) \mathbf{J}_{pos,m0}^T \mathbf{v}_f, \quad \mathbf{Q}_{SD,m1} = (k \cdot (L - L_0) + d \cdot (\dot{L} - \dot{L}_0) + f_a) \mathbf{J}_{pos,m1}^T \mathbf{v}_f, \quad (7.234)$$

where  $\mathbf{J}_{pos,m1}$  represents the derivative of marker  $m1$  w.r.t. its associated coordinates  $\mathbf{q}_{m1}$ , analogously  $\mathbf{J}_{pos,m0}$ .

### 7.6.1.4 Connector Jacobian

The position-level jacobian for the connector, involving all coordinates associated with markers  $m0$  and  $m1$ , follows from

$$\mathbf{J}_{SD} = \begin{bmatrix} \frac{\partial \mathbf{Q}_{SD,m0}}{\partial \mathbf{q}_{m0}} & \frac{\partial \mathbf{Q}_{SD,m0}}{\partial \mathbf{q}_{m1}} \\ \frac{\partial \mathbf{Q}_{SD,m0}}{\partial \mathbf{q}_{m1}} & \frac{\partial \mathbf{Q}_{SD,m1}}{\partial \mathbf{q}_{m1}} \end{bmatrix} \quad (7.235)$$

and the velocity level jacobian reads

$$\mathbf{J}_{SD,t} = \begin{bmatrix} \frac{\partial \mathbf{Q}_{SD,m0}}{\partial \dot{\mathbf{q}}_{m0}} & \frac{\partial \mathbf{Q}_{SD,m0}}{\partial \dot{\mathbf{q}}_{m1}} \\ \frac{\partial \mathbf{Q}_{SD,m0}}{\partial \dot{\mathbf{q}}_{m1}} & \frac{\partial \mathbf{Q}_{SD,m1}}{\partial \dot{\mathbf{q}}_{m1}} \end{bmatrix} \quad (7.236)$$

The sub-Jacobians follow from

$$\frac{\partial \mathbf{Q}_{SD,m0}}{\partial \mathbf{q}_{m0}} = - \frac{\partial \mathbf{J}_{pos,m0}^T}{\partial \mathbf{q}_{m0}} \mathbf{v}_f (k \cdot (L - L_0) + d \cdot (\dot{L} - \dot{L}_0) + f_a) - \mathbf{J}_{pos,m0}^T \frac{\partial \mathbf{v}_f (k \cdot (L - L_0) + d \cdot (\dot{L} - \dot{L}_0) + f_a)}{\partial \mathbf{q}_{m0}} \quad (7.237)$$

in which the term  $\frac{\partial \mathbf{J}_{pos,m0}^T}{\partial \mathbf{q}_{m0}}$  is computed from a special function provided by markers, that compute the derivative of the marker jacobian times a constant vector, in this case the spring force  $\mathbf{f}$ ; this jacobian term is usually less dominant, but is included in the numerical as well as the analytical derivatives, see the general jacobian computation information.

The other term, which is the dominant term, is computed as (dependence of velocity term on position coordinates and  $\dot{L}_0$  term neglected),

$$\begin{aligned}\frac{\partial \mathbf{Q}_{SD,m0}}{\partial \mathbf{q}_{m0}} &= -\mathbf{J}_{pos,m0}^T \frac{\partial \mathbf{v}_f (k \cdot (L - L_0) + d \cdot (\dot{L} - \dot{L}_0) + f_a)}{\partial \mathbf{q}_{m0}} \\ &= -\mathbf{J}_{pos,m0}^T \frac{\partial (k \cdot (\Delta^0 \mathbf{p} - L_0 \mathbf{v}_f) + \mathbf{v}_f (d \cdot \mathbf{v}_f^T \Delta^0 \mathbf{v} + f_a))}{\partial \mathbf{q}_{m0}} \\ &\approx \mathbf{J}_{pos,m0}^T \left( k \cdot \mathbf{I} - k \frac{L_0}{L} (\mathbf{I} - {}^0 \mathbf{v}_f \otimes {}^0 \mathbf{v}_f) + \frac{1}{L} (\mathbf{I} - {}^0 \mathbf{v}_f \otimes {}^0 \mathbf{v}_f) (d \cdot \mathbf{v}_f^T \Delta^0 \mathbf{v} + f_a) \right. \\ &\quad \left. + d {}^0 \mathbf{v}_f \otimes \left( \frac{1}{L} (\mathbf{I} - {}^0 \mathbf{v}_f \otimes {}^0 \mathbf{v}_f) {}^0 \mathbf{v}_f \right) \right) {}^0 \mathbf{J}_{pos,m0} \end{aligned} \quad (7.238)$$

Alternatively (again  $\dot{L}_0$  term neglected):

$$\begin{aligned}\frac{\partial \mathbf{Q}_{SD,m0}}{\partial \mathbf{q}_{m0}} &= -\mathbf{J}_{pos,m0}^T \frac{\partial \mathbf{v}_f (k \cdot (L - L_0) + d \cdot (\dot{L} - \dot{L}_0) + f_a)}{\partial \mathbf{q}_{m0}} \\ &= \mathbf{J}_{pos,m0}^T \frac{1}{L} (\mathbf{I} - {}^0 \mathbf{v}_f \otimes {}^0 \mathbf{v}_f) (k \cdot (L - L_0) + d \cdot (\dot{L} - \dot{L}_0) + f_a) \mathbf{J}_{pos,m0} \\ &\quad + \mathbf{J}_{pos,m0}^T {}^0 \mathbf{v}_f \otimes \left( k \cdot {}^0 \mathbf{v}_f + d \cdot \Delta^0 \mathbf{v} \frac{1}{L} (\mathbf{I} - {}^0 \mathbf{v}_f \otimes {}^0 \mathbf{v}_f) \right) \mathbf{J}_{pos,m0} - d \mathbf{J}_{pos,m0}^T {}^0 \mathbf{v}_f \otimes {}^0 \mathbf{v}_f \frac{\partial \Delta^0 \mathbf{v}}{\partial \mathbf{q}_{m0}} \\ &= \mathbf{J}_{pos,m0}^T \left( \frac{f_{SD}}{L} (\mathbf{I} - {}^0 \mathbf{v}_f \otimes {}^0 \mathbf{v}_f) + k {}^0 \mathbf{v}_f \otimes {}^0 \mathbf{v}_f + \frac{d}{L} ({}^0 \mathbf{v}_f \otimes \Delta^0 \mathbf{v}) \cdot (\mathbf{I} - {}^0 \mathbf{v}_f \otimes {}^0 \mathbf{v}_f) + \dots! \right) \mathbf{J}_{pos,m0} \end{aligned} \quad (7.239)$$

Noting that  $\frac{\partial \mathbf{v}_f}{\partial \mathbf{q}_{m0}} = -\frac{1}{L} (\mathbf{I} - {}^0 \mathbf{v}_f \otimes {}^0 \mathbf{v}_f) {}^0 \mathbf{J}_{pos,m0}$  and  $\frac{\partial \mathbf{v}_f}{\partial \mathbf{q}_{m1}} = \frac{1}{L} (\mathbf{I} - {}^0 \mathbf{v}_f \otimes {}^0 \mathbf{v}_f) {}^0 \mathbf{J}_{pos,m1}$ . The Jacobian w.r.t. velocity coordinates follows as

$$\begin{aligned}\frac{\partial \mathbf{Q}_{SD,m0}}{\partial \dot{\mathbf{q}}_{m0}} &= -\mathbf{J}_{pos,m0}^T \frac{\partial \mathbf{v}_f (k \cdot (L - L_0) + d \cdot (\dot{L} - \dot{L}_0) + f_a)}{\partial \dot{\mathbf{q}}_{m0}} \\ &= \mathbf{J}_{pos,m0}^T (d \mathbf{v}_f \otimes \mathbf{v}_f) {}^0 \mathbf{J}_{pos,m0} \end{aligned} \quad (7.240)$$

The term  $\frac{\partial \Delta^0 \mathbf{v}}{\partial \mathbf{q}_{m0}}$ , which is important for large damping, yields

$$\frac{\partial \Delta^0 \mathbf{v}}{\partial \mathbf{q}_{m0}} = \frac{\partial \mathbf{J}_{pos,m0} \dot{\mathbf{q}}_{m0}}{\partial \mathbf{q}_{m0}} = \frac{\partial \mathbf{J}_{pos,m0}}{\partial \mathbf{q}_{m0}} \dot{\mathbf{q}}_{m0} \quad (7.241)$$

The latter term is currently neglected.

Jacobians for markers  $m1$  and mixed  $m0/m1$  terms follow analogously.

---

**Userfunction:** `springForceUserFunction(mbs, t, itemNumber, deltaL, deltaL_t, stiffness, damping, force)`

A user function, which computes the spring force depending on time, object variables ( $\text{deltaL}$ ,  $\text{deltaL\_t}$ ) and object parameters ( $\text{stiffness}$ ,  $\text{damping}$ ,  $\text{force}$ ). The object variables are provided to the function using the current values of the SpringDamper object. Note that `itemNumber` represents the index of the object in `mbs`, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

arguments / return	type or size	description
mbs	MainSystem	provides MainSystem mbs to which object belongs
t	Real	current time in mbs
itemNumber	Index	integer number $i_N$ of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
deltaL	Real	$L - L_0$ , spring elongation
deltaL_t	Real	$(\dot{L} - \dot{L}_0)$ , spring velocity, including offset
stiffness	Real	copied from object
damping	Real	copied from object
force	Real	copied from object; constant force
<b>return value</b>	Real	scalar value of computed spring force

### User function example:

```
#define nonlinear force
def UFforce(mbs, t, itemNumber, u, v, k, d, F0):
    return k*u + d*v + F0
#markerNumbers taken from mini example
mbs.AddObject(ObjectConnectorSpringDamper(markerNumbers=[m0,m1],
                                             referenceLength = 1,
                                             stiffness = 100, damping = 1,
                                             springForceUserFunction = UFforce))
```

#### 7.6.1.5 MINI EXAMPLE for ObjectConnectorSpringDamper

```
node = mbs.AddNode(NodePoint(referenceCoordinates = [1.05,0,0]))
oMassPoint = mbs.AddObject(MassPoint(nodeNumber = node, physicsMass=1))

m0 = mbs.AddMarker(MarkerBodyPosition(bodyNumber=oGround, localPosition=[0,0,0]))
m1 = mbs.AddMarker(MarkerBodyPosition(bodyNumber=oMassPoint, localPosition=[0,0,0]))

mbs.AddObject(ObjectConnectorSpringDamper(markerNumbers=[m0,m1],
                                             referenceLength = 1, #shorter than initial
                                             distance
                                             stiffness = 100,
                                             damping = 1))

#assemble and solve system for default parameters
mbs.Assemble()
exu.SolveDynamic(mbs)
```

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

---

For examples on ObjectConnectorSpringDamper see Examples and TestModels:

- [SpringDamperMassUserFunction.py](#) (Examples/)
- [ANCF\\_contact\\_circle.py](#) (Examples/)
- [ANCF\\_contact\\_circle2.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- [Spring\\_with\\_constraints.py](#) (Examples/)
- [stiffFlyballGovernor2.py](#) (Examples/)
- [ANCFcontactCircleTest.py](#) (TestModels/)
- [modelUnitTests.py](#) (TestModels/)
- [PARTS\\_ATEs\\_moving.py](#) (TestModels/)
- [stiffFlyballGovernor.py](#) (TestModels/)

## 7.6.2 ObjectConnectorCartesianSpringDamper

An 3D spring-damper element, providing springs and dampers in three (global) directions (x,y,z); the connector can be attached to position-based markers.

**Additional information for ObjectConnectorCartesianSpringDamper:**

- The Object has the following types = Connector
- Requested marker type = Position
- Short name for Python = **CartesianSpringDamper**
- Short name for Python (visualization object) = **VCartesianSpringDamper**

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

Name	type	size	default value	description
name	String		"	connector's unique name
markerNumbers	ArrayMarkerIndex		[ MAXINT, MAX-INT ]	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	PyFunctionVector3DmbScalarIndexScalar4Vector3D		0	A Python function which computes the 3D force vector between the two marker points, if activeConnector=True; see description below
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectConnectorCartesianSpringDamper			parameters for visualization of item

The item **VObjectConnectorCartesianSpringDamper** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = diameter of spring; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==1, use default color

---

### 7.6.2.1 DESCRIPTION of ObjectConnectorCartesianSpringDamper:

#### Information on input parameters:

input parameter	symbol	description see tables above
markerNumbers	$[m_0, m_1]^T$	
stiffness	$k$	
damping	$d$	
offset	$v_{off}$	
springForceUserFunction	$UF \in \mathbb{R}^3$	

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

output variable	symbol	description
Displacement	$\Delta^0\mathbf{p} = {}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0}$	relative displacement in global coordinates
Distance	$L =  \Delta^0\mathbf{p} $	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

### 7.6.2.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker m0
marker m1 position	${}^0\mathbf{p}_{m1}$	
marker m0 velocity	${}^0\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
marker m1 velocity	${}^0\mathbf{v}_{m1}$	

### 7.6.2.3 Connector forces

Connector forces are based on relative displacements and relative velocities in global coordinates. Relative displacement between marker m0 to marker m1 positions is given by

$$\Delta^0\mathbf{p} = {}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0}, \quad (7.242)$$

and relative velocity reads

$$\Delta^0\mathbf{v} = {}^0\mathbf{v}_{m1} - {}^0\mathbf{v}_{m0}. \quad (7.243)$$

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

$${}^0\mathbf{f}_{SD} = \text{diag}(\mathbf{k}) \cdot (\Delta^0\mathbf{p} - {}^0\mathbf{v}_{off}) + \text{diag}(\mathbf{d}) \cdot \Delta^0\mathbf{v}. \quad (7.244)$$

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

$${}^0\mathbf{f}_{SD} = UF(mbs, t, i_N, \Delta^0\mathbf{p}, \Delta^0\mathbf{v}, \mathbf{k}, \mathbf{d}, \mathbf{v}_{off}), \quad (7.245)$$

and iN represents the itemNumber (=objectNumber). If activeConnector = False,  $\mathbf{f}_{SD}$  is set to zero.

The force  $\mathbf{f}_{SD}$  acts via the markers' position jacobians  $\mathbf{J}_{pos,m0}$  and  $\mathbf{J}_{pos,m1}$ . The generalized forces added to the LHS equations read for marker  $m0$ ,

$$\mathbf{f}_{LHS,m0} = -{}^0\mathbf{J}_{pos,m0}^T {}^0\mathbf{f}_{SD}, \quad (7.246)$$

and for marker  $m1$ ,

$$\mathbf{f}_{LHS,m1} = {}^0\mathbf{J}_{pos,m1}^T {}^0\mathbf{f}_{SD}. \quad (7.247)$$

The LHS equation parts are added accordingly using the LTG mapping. Note that the different signs result from the signs in Eq. (7.242).

The connector also provides an analytic jacobian, which is used if `newton.numericalDifferentiation.forODE2 = False` and if there is no `springForceUserFunction` (otherwise numerical differentiation is used).

The analytic jacobian for the coupled equation parts  $\mathbf{f}_{LHS,m0}$  and  $\mathbf{f}_{LHS,m1}$  is based on the local jacobians

$$\begin{aligned} \mathbf{J}_{loc0} &= f_{ODE2} \frac{\partial {}^0\mathbf{f}_{SD}}{\partial {}^0\mathbf{p}_{m0}} + f_{ODE2_t} \frac{\partial {}^0\mathbf{f}_{SD}}{\partial {}^0\mathbf{v}_{m0}} = -f_{ODE2} \cdot \text{diag}(\mathbf{k}) - f_{ODE2_t} \cdot \text{diag}(\mathbf{d}), \\ \mathbf{J}_{loc1} &= f_{ODE2} \frac{\partial {}^0\mathbf{f}_{SD}}{\partial {}^0\mathbf{p}_{m1}} + f_{ODE2_t} \frac{\partial {}^0\mathbf{f}_{SD}}{\partial {}^0\mathbf{v}_{m1}} = f_{ODE2} \cdot \text{diag}(\mathbf{k}) + f_{ODE2_t} \cdot \text{diag}(\mathbf{d}). \end{aligned} \quad (7.248)$$

Here,  $f_{ODE2}$  is the factor for the position derivative and  $f_{ODE2_t}$  is the factor for the velocity derivative, which allows a computation of the computation for both the position as well as the velocity part at the same time. The complete jacobian for the LHS equations then reads,

$$\begin{aligned} \mathbf{J}_{CSD} &= \begin{bmatrix} \frac{\partial \mathbf{f}_{LHS,m0}}{\partial \mathbf{q}_{m0}} & \frac{\partial \mathbf{f}_{LHS,m0}}{\partial \mathbf{q}_{m1}} \\ \frac{\partial \mathbf{f}_{LHS,m1}}{\partial \mathbf{q}_{m0}} & \frac{\partial \mathbf{f}_{LHS,m1}}{\partial \mathbf{q}_{m1}} \end{bmatrix} + \mathbf{J}_{CSD'} \\ &= \begin{bmatrix} -{}^0\mathbf{J}_{pos,m0}^T \mathbf{J}_{loc0} \mathbf{J}_{pos,m0} & -{}^0\mathbf{J}_{pos,m0}^T \mathbf{J}_{loc1} \mathbf{J}_{pos,m1} \\ {}^0\mathbf{J}_{pos,m1}^T \mathbf{J}_{loc0} \mathbf{J}_{pos,m0} & {}^0\mathbf{J}_{pos,m1}^T \mathbf{J}_{loc1} \mathbf{J}_{pos,m1} \end{bmatrix} + \mathbf{J}_{CSD'} \\ &= \begin{bmatrix} {}^0\mathbf{J}_{pos,m0}^T \mathbf{J}_{loc1} \mathbf{J}_{pos,m0} & -{}^0\mathbf{J}_{pos,m0}^T \mathbf{J}_{loc1} \mathbf{J}_{pos,m1} \\ -{}^0\mathbf{J}_{pos,m1}^T \mathbf{J}_{loc1} \mathbf{J}_{pos,m0} & {}^0\mathbf{J}_{pos,m1}^T \mathbf{J}_{loc1} \mathbf{J}_{pos,m1} \end{bmatrix} + \mathbf{J}_{CSD'} \end{aligned} \quad (7.249)$$

Here,  $\mathbf{q}_{m0}$  are the coordinates associated with marker  $m0$  and  $\mathbf{q}_{m1}$  of marker  $m1$ .

The second term  $\mathbf{J}_{CSD'}$  is only non-zero if  $\frac{\partial {}^0\mathbf{J}_{pos,i}^T}{\partial \mathbf{q}_i}$  is non-zero, using  $i \in \{m0, m1\}$ . As the latter terms would require to compute a 3-dimensional array, the second jacobian term is computed as

$$\mathbf{J}_{CSD'} = \begin{bmatrix} -f_{ODE2} \frac{\partial ({}^0\mathbf{J}_{pos,m0}^T \mathbf{f}')}{\partial \mathbf{q}_{m0}} & \mathbf{0} \\ \mathbf{0} & f_{ODE2} \frac{\partial ({}^0\mathbf{J}_{pos,m1}^T \mathbf{f}')}{\partial \mathbf{q}_{m1}} \end{bmatrix} \quad (7.250)$$

in which we set  $\mathbf{f}' = {}^0\mathbf{f}_{SD}$ , but the derivatives in Eq. (7.250) are evaluated by setting  $\mathbf{f}' = const.$

**Userfunction:** `springForceUserFunction(mbs, t, itemNumber, displacement, velocity, stiffness, damping, offset)`

A user function, which computes the 3D spring force vector depending on time, object variables (`deltaL`, `deltaL_t`) and object parameters (`stiffness`, `damping`, `force`). The object variables are provided to the function using the current values of the `SpringDamper` object. Note that `itemNumber` represents the index of the object in `mbs`, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

arguments / return	type or size	description
mbs	MainSystem	provides MainSystem mbs in which underlying item is defined
t	Real	current time in mbs
itemNumber	Index	integer number $i_N$ of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
displacement	Vector3D	$\Delta^0 \mathbf{p}$
velocity	Vector3D	$\Delta^0 \mathbf{v}$
stiffness	Vector3D	copied from object
damping	Vector3D	copied from object
offset	Vector3D	copied from object
<b>return value</b>	Vector3D	list or numpy array of computed spring force

### User function example:

```
#define simple force for spring-damper:
def UFforce(mbs, t, itemNumber, u, v, k, d, offset):
    return [u[0]*k[0],u[1]*k[1],u[2]*k[2]]

#markerNumbers and parameters taken from mini example
mbs.AddObject(CartesianSpringDamper(markerNumbers = [mGround, mMass],
                                      stiffness = [k,k,k],
                                      damping = [0,k*0.05,0], offset = [0,0,0],
                                      springForceUserFunction = UFforce))
```

#### 7.6.2.4 MINI EXAMPLE for ObjectConnectorCartesianSpringDamper

```
#example with mass at [1,1,0], 5kg under load 5N in -y direction
k=5000
nMass = mbs.AddNode(NodePoint(referenceCoordinates=[1,1,0]))
oMass = mbs.AddObject(MassPoint(physicsMass = 5, nodeNumber = nMass))

mMass = mbs.AddMarker(MarkerNodePosition(nodeNumber=nMass))
mGround = mbs.AddMarker(MarkerBodyPosition(bodyNumber=oGround, localPosition = [1,1,0]))
mbs.AddObject(CartesianSpringDamper(markerNumbers = [mGround, mMass],
                                      stiffness = [k,k,k],
                                      damping = [0,k*0.05,0], offset = [0,0,0]))
mbs.AddLoad(Force(markerNumber = mMass, loadVector = [0, -5, 0])) #static solution
=-5/5000=-0.001m

#assemble and solve system for default parameters
mbs.Assemble()
```

```

exu.SolveDynamic(mbs)

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

```

---

For examples on ObjectConnectorCartesianSpringDamper see Examples and TestModels:

- [mouseInteractionExample.py](#) (Examples/)
- [rigid3Dexample.py](#) (Examples/)
- [sliderCrank3DwithANCFbeltDrive2.py](#) (Examples/)
- [ANCF\\_contact\\_circle.py](#) (Examples/)
- [ANCF\\_contact\\_circle2.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- [cartesianSpringDamper.py](#) (Examples/)
- [flexibleRotor3Dtest.py](#) (Examples/)
- [lavalRotor2Dtest.py](#) (Examples/)
- [NGsolvePistonEngine.py](#) (Examples/)
- [NGsolvePostProcessingStresses.py](#) (Examples/)
- [objectFFRFreducedOrderNetgen.py](#) (Examples/)
- ...
- [scissorPrismaticRevolute2D.py](#) (TestModels/)
- [sphericalJointTest.py](#) (TestModels/)
- [ANCFcontactCircleTest.py](#) (TestModels/)
- ...

### 7.6.3 ObjectConnectorRigidBodySpringDamper

An 3D spring-damper element acting on relative displacements and relative rotations of two rigid body (position+orientation) markers; connects to (position+orientation)-based markers and represents a penalty-based rigid joint (or prismatic, revolute, etc.)

**Additional information for ObjectConnectorRigidBodySpringDamper:**

- The Object has the following types = Connector
- Requested marker type = Position + Orientation
- Requested node type = GenericData
- **Short name for Python = RigidBodySpringDamper**
- **Short name for Python (visualization object) = VRigidBodySpringDamper**

The item **ObjectConnectorRigidBodySpringDamper** with type = 'ConnectorRigidBodySpringDamper' has the following parameters:

Name	type	size	default value	description
name	String		"	connector's unique name
markerNumbers	ArrayMarkerIndex		[ MAXINT, MAX-INT ]	list of markers used in connector
nodeNumber	NodeIndex		MAXINT	node number of a NodeGenericData (size depends on application) for dataCoordinates for user functions (e.g., implementing contact/friction user function)
stiffness	Matrix6D		np.zeros([6,6])	stiffness [SI:N/m or Nm/rad] of translational, torsional and coupled springs; act against relative displacements in x, y, and z-direction as well as the relative angles (calculated as Euler angles); in the simplest case, the first 3 diagonal values correspond to the local stiffness in x,y,z direction and the last 3 diagonal values correspond to the rotational stiffness around x,y and z axis
damping	Matrix6D		np.zeros([6,6])	damping [SI:N/(m/s) or Nm/(rad/s)] of translational, torsional and coupled dampers; very similar to stiffness, however, the rotational velocity is computed from the angular velocity vector
rotationMarker0	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	local rotation matrix for marker 0; stiffness, damping, etc. components are measured in local coordinates relative to rotationMarker0
rotationMarker1	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	local rotation matrix for marker 1; stiffness, damping, etc. components are measured in local coordinates relative to rotationMarker1

offset	Vector6D	[0.,0.,0.,0.,0.,0.]	translational and rotational offset considered in the spring force calculation
activeConnector	Bool	True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
springForceTorqueUserFunction	PyFunctionVector6DmbsScalarIndex4Vector3D	0	2Matrix6D2Matrix3DVector6D A Python function which computes the 6D force-torque vector (3D force + 3D torque) between the two rigid body markers, if activeConnector=True; see description below
postNewtonStepUserFunction	PyFunctionVectorMbsScalarIndex4VectorVector3D	0	2Matrix6D2Matrix3DVector6D A Python function which computes the error of the PostNewtonStep; see description below
visualization	VObjectConnectorRigidBodySpringDamper		parameters for visualization of item

The item VObjectConnectorRigidBodySpringDamper has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = diameter of spring; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==1, use default color

### 7.6.3.1 DESCRIPTION of ObjectConnectorRigidBodySpringDamper:

**Information on input parameters:**

input parameter	symbol	description see tables above
nodeNumber	$n_d$	
springForceTorqueUserFunction	$UF \in \mathbb{R}^6$	
postNewtonStepUserFunction	$UF_{PN} \in \mathbb{R}$	

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

output variable	symbol	description
DisplacementLocal	${}^{j0}\Delta p$	relative displacement in local joint0 coordinates
VelocityLocal	${}^{j0}\Delta v$	relative translational velocity in local joint0 coordinates

Rotation	${}^{J0}\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2]^T$	relative rotation parameters (Tait Bryan Rxyz); these are the angles used for calculation of joint torques (e.g. if $cX$ is the diagonal rotational stiffness, the moment for axis X reads $mX=cX*\phi_iX$ , etc.)
AngularVelocityLocal	${}^{J0}\Delta\boldsymbol{\omega}$	relative angular velocity in local joint0 coordinates
ForceLocal	${}^{J0}\mathbf{f}$	joint force in local joint0 coordinates
TorqueLocal	${}^{J0}\mathbf{m}$	joint torque in local joint0 coordinates

### 7.6.3.2 Definition of quantities

input parameter	symbol	description
stiffness	$\mathbf{k} \in \mathbb{R}^{6 \times 6}$	stiffness in $J0$ coordinates
damping	$\mathbf{d} \in \mathbb{R}^{6 \times 6}$	damping in $J0$ coordinates
offset	${}^{J0}\mathbf{v}_{off} \in \mathbb{R}^6$	offset in $J0$ coordinates
rotationMarker0	${}^{m0,J0}\mathbf{A}$	rotation matrix which transforms from joint 0 into marker 0 coordinates
rotationMarker1	${}^{m1,J1}\mathbf{A}$	rotation matrix which transforms from joint 1 into marker 1 coordinates
markerNumbers[0]	$m0$	global marker number $m0$
markerNumbers[1]	$m1$	global marker number $m1$

intermediate variables	symbol	description
marker $m0$ position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker $m0$
marker $m0$ orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker $m0$
marker $m1$ position	${}^0\mathbf{p}_{m1}$	accordingly
marker $m1$ orientation	${}^{0,m1}\mathbf{A}$	current rotation matrix provided by marker $m1$
marker $m0$ velocity	${}^0\mathbf{v}_{m0}$	current global velocity which is provided by marker $m0$
marker $m1$ velocity	${}^0\mathbf{v}_{m1}$	accordingly
marker $m0$ velocity	${}^{m0}\boldsymbol{\omega}_{m0}$	current local angular velocity vector provided by marker $m0$
marker $m1$ velocity	${}^{m1}\boldsymbol{\omega}_{m1}$	current local angular velocity vector provided by marker $m1$
Displacement	${}^0\Delta\mathbf{p}$	${}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0}$
Velocity	${}^0\Delta\mathbf{v}$	${}^0\mathbf{v}_{m1} - {}^0\mathbf{v}_{m0}$
DisplacementLocal	${}^{J0}\Delta\mathbf{p}$	$({}^{0,m0}\mathbf{A} {}^{m0,J0}\mathbf{A})^T {}^0\Delta\mathbf{p}$
VelocityLocal	${}^{J0}\Delta\mathbf{v}$	$({}^{0,m0}\mathbf{A} {}^{m0,J0}\mathbf{A})^T {}^0\Delta\mathbf{v}$
AngularVelocityLocal	${}^{J0}\Delta\boldsymbol{\omega}$	$({}^{0,m0}\mathbf{A} {}^{m0,J0}\mathbf{A})^T ({}^{0,m1}\boldsymbol{\omega} - {}^{0,m0}\boldsymbol{\omega})$

### 7.6.3.3 Connector forces

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

$$\begin{bmatrix} {}^{J0}\mathbf{f}_{SD} \\ {}^{J0}\mathbf{m}_{SD} \end{bmatrix} = \mathbf{k} \left( \begin{bmatrix} {}^{J0}\Delta\mathbf{p} \\ {}^{J0}\boldsymbol{\theta} \end{bmatrix} - {}^{J0}\mathbf{v}_{off} \right) + \mathbf{d} \begin{bmatrix} {}^{J0}\Delta\mathbf{v} \\ {}^{J0}\Delta\boldsymbol{\omega} \end{bmatrix} \quad (7.251)$$

For the application of joint forces to markers,  $[{}^{J0}\mathbf{f}_{SD}, {}^{J0}\mathbf{m}_{SD}]^T$  is transformed into global coordinates. if `activeConnector = False`,  ${}^{J0}\mathbf{f}_{SD}$  and  ${}^{J0}\mathbf{m}_{SD}$  are set to zero.:

If the `springForceTorqueUserFunction` UF is defined and `activeConnector = True`,  $\mathbf{f}_{SD}$  instead becomes ( $t$  is current time)

$$\mathbf{f}_{SD} = \text{UF}(mbs, t, i_N, {}^{J0}\Delta\mathbf{p}, {}^{J0}\boldsymbol{\theta}, {}^{J0}\Delta\mathbf{v}, {}^{J0}\Delta\boldsymbol{\omega}, \text{stiffness}, \text{damping}, \text{rotationMarker0}, \text{rotationMarker1}, \text{offset}) \quad (7.252)$$

and `iN` represents the itemNumber (=objectNumber).

---

**Userfunction:** `springForceTorqueUserFunction(mbs, t, itemNumber, displacement, rotation, velocity, angularVelocity, stiffness, damping, rotJ0, rotJ1, offset)`

A user function, which computes the 6D spring-damper force-torque vector depending on mbs, time, local quantities (displacement, rotation, velocity, angularVelocity, stiffness), which are evaluated at current time, which are relative quantities between both markers and which are defined in joint J0 coordinates. As relative rotations are defined by Tait-Bryan rotation parameters, it is recommended to use this connector for small relative rotations only (except for rotations about one axis). Furthermore, the user function contains object parameters (stiffness, damping, rotationMarker0/1, offset). Note that itemNumber represents the index of the object in mbs, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

Detailed description of the arguments and local quantities:

arguments / return	type or size	description
<code>mbs</code>	MainSystem	provides MainSystem mbs in which underlying item is defined
<code>t</code>	Real	current time in mbs
<code>itemNumber</code>	Index	integer number $i_N$ of the object in mbs, allowing easy access to all object data via <code>mbs.GetObjectParameter(itemNumber, ...)</code>
<code>displacement</code>	Vector3D	${}^{J0}\Delta\mathbf{p}$
<code>rotation</code>	Vector3D	${}^{J0}\boldsymbol{\theta}$
<code>velocity</code>	Vector3D	${}^{J0}\Delta\mathbf{v}$
<code>angularVelocity</code>	Vector3D	${}^{J0}\Delta\boldsymbol{\omega}$
<code>stiffness</code>	Vector6D	copied from object
<code>damping</code>	Vector6D	copied from object
<code>rotJ0</code>	Matrix3D	rotationMarker0 copied from object
<code>rotJ1</code>	Matrix3D	rotationMarker1 copied from object
<code>offset</code>	Vector6D	copied from object
<code>return value</code>	Vector6D	list or numpy array of computed spring force-torque

---

## Userfunction: postNewtonStepUserFunction(mbs, t, Index itemIndex, dataCoordinates, displacement, rotation, velocity, angularVelocity, stiffness, damping, rotJ0, rotJ1, offset)

A user function which computes the error of the PostNewtonStep  $\varepsilon_{PN}$ , a recommended for stepsize reduction  $t_{recom}$  (use values  $> 0$  to recommend step size or values  $< 0$  else; 0 gives minimum step size) and the updated dataCoordinates  $\mathbf{d}^k$  of NodeGenericData  $n_d$ . Except from dataCoordinates, the arguments are the same as in `springForceTorqueUserFunction`. The `postNewtonStepUserFunction` should be used together with the dataCoordinates in order to implement a active set or switching strategy for discontinuous events, such as in contact, friction, plasticity, fracture or similar.

Detailed description of the arguments and local quantities:

arguments / return	type or size	description
mbs	MainSystem	provides MainSystem mbs in which underlying item is defined
t	Real	current time in mbs
itemNumber	Index	integer number of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
dataCoordinates	Vector	$\mathbf{d}^{k-1} = [d_0^{k-1}, d_1^{k-1}, \dots]$ for previous post Newton step $k - 1$
...	...	other arguments see <code>springForceTorqueUserFunction</code>
<b>return value</b>	Vector	$[\varepsilon_{PN}, t_{recom}, d_0^k, d_1^k, \dots]$ where $k$ indicates the current step

### User function example:

```
#define simple force for spring-damper:
def UFforce(mbs, t, itemNumber, displacement, rotation, velocity, angularVelocity,
            stiffness, damping, rotJ0, rotJ1, offset):
    k = stiffness #passed as list
    u = displacement
    return [u[0]*k[0][0],u[1]*k[1][1],u[2]*k[2][2], 0,0,0]

#markerNumbers and parameters taken from mini example
mbs.AddObject(RigidBodySpringDamper(markerNumbers = [mGround, mBody],
                                      stiffness = np.diag([k,k,k, 0,0,0]),
                                      damping = np.diag([0,k*0.01,0, 0,0,0]),
                                      offset = [0,0,0, 0,0,0],
                                      springForceTorqueUserFunction = UFforce))
```

#### 7.6.3.4 MINI EXAMPLE for ObjectConnectorRigidBodySpringDamper

```
#example with rigid body at [0,0,0], 1kg under initial velocity
k=500
```

```

nBody = mbs.AddNode(RigidRxyz(initialVelocities=[0,1e3,0, 0,0,0]))
oBody = mbs.AddObject(RigidBody(physicsMass=1, physicsInertia=[1,1,1,0,0,0],
                               nodeNumber=nBody))

mBody = mbs.AddMarker(MarkerNodeRigid(nodeNumber=nBody))
mGround = mbs.AddMarker(MarkerBodyRigid(bodyNumber=oGround,
                                         localPosition = [0,0,0]))
mbs.AddObject(RigidBodySpringDamper(markerNumbers = [mGround, mBody],
                                     stiffness = np.diag([k,k,k, 0,0,0]),
                                     damping = np.diag([0,k*0.01,0, 0,0,0]),
                                     offset = [0,0,0, 0,0,0]))


#assemble and solve system for default parameters
mbs.Assemble()
exu.SolveDynamic(mbs, exu.SimulationSettings())


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

```

For examples on ObjectConnectorRigidBodySpringDamper see Examples and TestModels:

- [stiffFlyballGovernor2.py](#) (Examples/)
- [connectorRigidBodySpringDamperTest.py](#) (TestModels/)
- [stiffFlyballGovernor.py](#) (TestModels/)
- [superElementRigidJointTest.py](#) (TestModels/)

#### 7.6.4 ObjectConnectorTorsionalSpringDamper

An torsional spring-damper element acting on relative rotations around Z-axis of local joint0 coordinate system; connects to orientation-based markers; if other rotation axis than the local joint0 Z axis shall be used, the joint rotationMarker0 / rotationMarker1 may be used. The joint perfectly extends a RevoluteJoint with a spring-damper, which can also be used to represent feedback control in an elegant and efficient way, by choosing appropriate user functions. It also allows to measure continuous / infinite rotations by making use of a NodeGeneric which compensates  $\pm\pi$  jumps in the measured rotation (OutputVariableType.Rotation).

**Additional information for ObjectConnectorTorsionalSpringDamper:**

- The Object has the following types = Connector
- Requested marker type = Orientation
- Requested node type = GenericData
- **Short name for Python = TorsionalSpringDamper**
- **Short name for Python (visualization object) = VTorsionalSpringDamper**

The item **ObjectConnectorTorsionalSpringDamper** with type = 'ConnectorTorsionalSpringDamper' has the following parameters:

Name	type	size	default value	description
name	String		"	connector's unique name
markerNumbers	ArrayMarkerIndex		[ MAXINT, MAX-INT ]	list of markers used in connector
nodeNumber	NodeIndex		MAXINT	node number of a NodeGenericData with 1 dataCoordinate for continuous rotation reconstruction; if this node is left to invalid index, it will not be used
stiffness	Real		0.	torsional stiffness [SI:Nm/rad] against relative rotation
damping	Real		0.	torsional damping [SI:Nm/(rad/s)]
rotationMarker0	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	local rotation matrix for marker 0; transforms joint into marker coordinates
rotationMarker1	Matrix3D		[[1,0,0], [0,1,0], [0,0,1]]	local rotation matrix for marker 1; transforms joint into marker coordinates
offset	Real		0.	rotational offset considered in the spring torque calculation
velocityOffset	Real		0.	angular velocity offset considered in the damper torque calculation
torque	Real		0.	additional constant torque [SI:Nm] added to spring-damper; this can be used to prescribe a torque between the two attached bodies (e.g., for actuation and control)
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint

springTorqueUserFunction	PyFunctionMbsScalarIndexScalar5	0	A Python function which computes the scalar torque between the two rigid body markers in local joint0 coordinates, if activeConnector=True; see description below
visualization	VObjectConnectorTorsionalSpringDamper		parameters for visualization of item

The item VObjectConnectorTorsionalSpringDamper has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = diameter of spring; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==1, use default color

#### 7.6.4.1 DESCRIPTION of ObjectConnectorTorsionalSpringDamper:

Information on input parameters:

input parameter	symbol	description see tables above
nodeNumber	$n_d$	
stiffness	$k$	
damping	$d$	
offset	$v_{off}$	
velocityOffset	$\dot{v}_{off}$	
torque	$\tau_c$	
springTorqueUserFunction	$UF \in \mathbb{R}$	

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

output variable	symbol	description
Rotation	$\Delta\theta$	relative rotation around the joint Z-coordinate, enhanced to a continuous rotation (infinite rotations $> +\pi$ and $< -\pi$ ) if a NodeGeneric with 1 coordinate as added
AngularVelocityLocal	$\Delta\omega$	scalar relative angular velocity around joint0 Z-axis
TorqueLocal	$\tau$	scalar joint torque around the local joint0 Z-axis

#### 7.6.4.2 Definition of quantities

input parameter	symbol	description
rotationMarker0	${}^{m0,J0}\mathbf{A}$	rotation matrix which transforms from joint 0 into marker 0 coordinates
rotationMarker1	${}^{m1,J1}\mathbf{A}$	rotation matrix which transforms from joint 1 into marker 1 coordinates
markerNumbers[0]	$m0$	global marker number m0
markerNumbers[1]	$m1$	global marker number m1
nodeNumber	$n0$	optional node number of a generic node (otherwise exu.InvalidIndex())

intermediate variables	symbol	description
marker m0 orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker m0
marker m1 orientation	${}^{0,m1}\mathbf{A}$	current rotation matrix provided by marker m1
marker m0 ang. velocity	${}^{m0}\boldsymbol{\omega}_{m0}$	current local angular velocity vector provided by marker m0
marker m1 ang. velocity	${}^{m1}\boldsymbol{\omega}_{m1}$	current local angular velocity vector provided by marker m1
AngularVelocityLocal	$\Delta\boldsymbol{\omega} = \left( {}^{J0,m1}\mathbf{A} \, {}^{m1}\boldsymbol{\omega} - {}^{J0,m0}\mathbf{A} \, {}^{m0}\boldsymbol{\omega} \right)_Z$	angular velocity around joint0 Z-axis

#### 7.6.4.3 Connector forces

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

$$\tau_{SD} = k(\Delta\theta - v_{off}) + d(\Delta\omega - \dot{v}_{off}) + \tau_c \quad (7.253)$$

if `activeConnector = False`,  $\tau_{SD}$  is set zero.

If the `springTorqueUserFunction` UF is defined and `activeConnector = True`,  $\tau_{SD}$  instead becomes ( $t$  is current time)

$$\tau_{SD} = \text{UF}(mbs, t, i_N, \Delta\theta, \Delta\omega, \text{stiffness}, \text{damping}, \text{rotationMarker0}, \text{rotationMarker1}, \text{offset}) \quad (7.254)$$

and `iN` represents the `itemNumber` (=objectNumber).

**Userfunction:** `springTorqueUserFunction(mbs, t, itemNumber, rotation, angularVelocity, stiffness, damping, offset)`

A user function, which computes the scalar torque depending on `mbs`, time, local quantities (relative rotation, relative angularVelocity), which are evaluated at current time. Furthermore, the user function contains object parameters (stiffness, damping, offset). Note that `itemNumber` represents the index of the object in `mbs`, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

Detailed description of the arguments and local quantities:

arguments / return	type or size	description
<code>mbs</code>	MainSystem	provides MainSystem <code>mbs</code> in which underlying item is defined

<b>t</b>	Real	current time in mbs
<b>itemNumber</b>	Index	integer number $i_N$ of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
<b>rotation</b>	Real	$\Delta\theta$
<b>angularVelocity</b>	Real	$\Delta\omega$
<b>stiffness</b>	Real	copied from object
<b>damping</b>	Real	copied from object
<b>offset</b>	Real	copied from object
<b>return value</b>	Real	computed torque

---

### User function example:

```
#define simple cubic force for spring-damper:
def UFforce(mbs, t, itemNumber, rotation, angularVelocity, stiffness, damping, offset):
    k = stiffness #passed as list
    u = rotation
    return k*u + 0.1*k*u**3

#markerNumbers and parameters taken from mini example
mbs.AddObject(TorsionalSpringDamper(markerNumbers = [mGround, mBody],
                                      stiffness = k,
                                      damping = k*0.01,
                                      offset = 0,
                                      springTorqueUserFunction = UFforce))
```

---

#### 7.6.4.4 MINI EXAMPLE for ObjectConnectorTorsionalSpringDamper

```
#example with rigid body at [0,0,0], with torsional load
k=2e3
nBody = mbs.AddNode(RigidRxyz())
oBody = mbs.AddObject(RigidBody(physicsMass=1, physicsInertia=[1,1,1,0,0,0],
                                nodeNumber=nBody))

mBody = mbs.AddMarker(MarkerNodeRigid(nodeNumber=nBody))
mGround = mbs.AddMarker(MarkerBodyRigid(bodyNumber=oGround,
                                         localPosition = [0,0,0]))
mbs.AddObject(RevoluteJointZ(markerNumbers = [mGround, mBody])) #rotation around ground
Z-axis
mbs.AddObject(TorsionalSpringDamper(markerNumbers = [mGround, mBody],
                                      stiffness = k, damping = k*0.01, offset = 0))

#torque around z-axis; expect approx. phiZ = 1/k=0.0005
```

```

mbs.AddLoad(Torque(markerNumber = mBody, loadVector=[0,0,1]))

#assemble and solve system for default parameters
mbs.Assemble()
exu.SolveDynamic(mbs, exu.SimulationSettings())

#check result at default integration time
exdynTestGlobals.testResult = mbs.GetNodeOutput(nBody, exu.OutputVariableType.Rotation)
[2]

```

For examples on ObjectConnectorTorsionalSpringDamper see Examples and TestModels:

- [serialRobotTSD.py](#) (Examples/)

## 7.6.5 ObjectConnectorCoordinateSpringDamper

A 1D (scalar) spring-damper element acting on single [ODE2](#) coordinates; connects to coordinate-based markers; NOTE that the coordinate markers only measure the coordinate (=displacement), but the reference position is not included as compared to position-based markers!; the spring-damper can also act on rotational coordinates.

**Additional information for ObjectConnectorCoordinateSpringDamper:**

- The Object has the following types = Connector
- Requested marker type = Coordinate
- **Short name for Python** = **CoordinateSpringDamper**
- **Short name for Python (visualization object)** = **VCoordinateSpringDamper**

The item **ObjectConnectorCoordinateSpringDamper** with type = 'ConnectorCoordinateSpringDamper' has the following parameters:

Name	type	size	default value	description
name	String		"	connector's unique name
markerNumbers	ArrayMarkerIndex		[ MAXINT, MAX-INT ]	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 (temporarily) a connector or constraint
springForceUserFunction	PyFunctionMbsScalarIndexScalar7	0		A Python function which defines the spring force with 8 parameters, see equations section / see description below
visualization	VObjectConnectorCoordinateSpringDamper			parameters for visualization of item

The item **VObjectConnectorCoordinateSpringDamper** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = diameter of spring; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==1, use default color

---

### 7.6.5.1 DESCRIPTION of ObjectConnectorCoordinateSpringDamper:

**Information on input parameters:**

input parameter	symbol	description see tables above
stiffness	$k$	
damping	$d$	
offset	$l_{\text{off}}$	
dryFriction	$f_\mu$	
dryFrictionProportionalZone	$v_\mu$	
springForceUserFunction	$UF \in \mathbb{R}$	

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

output variable	symbol	description
Displacement	$\Delta q$	relative scalar displacement of marker coordinates
Velocity	$\Delta v$	difference of scalar marker velocity coordinates
Force	$f_{SD}$	scalar spring force

### 7.6.5.2 Definition of quantities

intermediate variables	symbol	description
marker m0 coordinate	$q_{m0}$	current displacement coordinate which is provided by marker m0; does NOT include reference coordinate!
marker m1 coordinate	$q_{m1}$	
marker m0 velocity coordinate	$v_{m0}$	current velocity coordinate which is provided by marker m0
marker m1 velocity coordinate	$v_{m1}$	

### 7.6.5.3 Connector forces

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

$$\Delta q = q_{m1} - q_{m0} \quad (7.255)$$

and relative velocity,

$$\Delta v = v_{m1} - v_{m0} \quad (7.256)$$

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

$$f_{\text{friction}} = \begin{cases} \text{Sgn}(\Delta v) \cdot f_\mu & \text{if } |\Delta v| \geq v_\mu \\ \frac{\Delta v}{v_\mu} f_\mu & \text{if } |\Delta v| < v_\mu \end{cases} \quad (7.257)$$

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

$$f_{SD} = k(\Delta q - l_{\text{off}}) + d \cdot \Delta v + f_{\text{friction}} \quad (7.258)$$

If the `springForceUserFunction` UF is defined,  $f_{SD}$  instead becomes ( $t$  is current time)

$$f_{SD} = \text{UF}(mbs, t, i_N, \Delta q, \Delta v, k, d, l_{\text{off}}, f_u, v_\mu) \quad (7.259)$$

and `iN` represents the `itemNumber` (=objectNumber).

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

---

#### **Userfunction: `springForceUserFunction(mbs, t, itemNumber, displacement, velocity, stiffness, damping, offset, dryFriction, dryFrictionProportionalZone)`**

A user function, which computes the scalar spring force depending on time, object variables (displacement, velocity) and object parameters . The object variables are passed to the function using the current values of the `CoordinateSpringDamper` object. Note that `itemNumber` represents the index of the object in `mbs`, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

arguments / return	type or size	description
<code>mbs</code>	MainSystem	provides MainSystem <code>mbs</code> in which underlying item is defined
<code>t</code>	Real	current time in <code>mbs</code>
<code>itemNumber</code>	Index	integer number $i_N$ of the object in <code>mbs</code> , allowing easy access to all object data via <code>mbs.GetObjectParameter(itemNumber, ...)</code>
<code>displacement</code>	Real	$\Delta q$
<code>velocity</code>	Real	$\Delta v$
<code>stiffness</code>	Real	copied from object
<code>damping</code>	Real	copied from object
<code>offset</code>	Real	copied from object
<code>dryFriction</code>	Real	copied from object
<code>dryFrictionProportionalZone</code>	Real	copied from object
<b>return value</b>	Real	scalar value of computed force

---

#### **User function example:**

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

---

#### **7.6.5.4 MINI EXAMPLE for ObjectConnectorCoordinateSpringDamper**

```

def springForce(mbs, t, itemNumber, u, v, k, d, offset,
               dryFriction, dryFrictionProportionalZone):
    return 0.1*k*u+k*u**3+v*d

nMass=mbs.AddNode(Point(referenceCoordinates = [2,0,0]))
massPoint = mbs.AddObject(MassPoint(physicsMass = 5, nodeNumber = nMass))

groundMarker=mbs.AddMarker(MarkerNodeCoordinate(nodeNumber= nGround, coordinate = 0))
nodeMarker =mbs.AddMarker(MarkerNodeCoordinate(nodeNumber= nMass, coordinate = 0))

#Spring-Damper between two marker coordinates
mbs.AddObject(CoordinateSpringDamper(markerNumbers = [groundMarker, nodeMarker],
                                      stiffness = 5000, damping = 80,
                                      springForceUserFunction = springForce))
loadCoord = mbs.AddLoad(LoadCoordinate(markerNumber = nodeMarker, load = 1)) #static
linear solution:0.002

#assemble and solve system for default parameters
mbs.Assemble()
exu.SolveDynamic(mbs)

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

```

For examples on ObjectConnectorCoordinateSpringDamper see Examples and TestModels:

- [slidercrankWithMassSpring.py](#) (Examples/)
- [ANCFALEtest.py](#) (Examples/)
- [ANCF\\_switchingSlidingJoint2D.py](#) (Examples/)
- [ComputeSensitivitiesExample.py](#) (Examples/)
- [coordinateSpringDamper.py](#) (Examples/)
- [finiteSegmentMethod.py](#) (Examples/)
- [geneticOptimizationSliderCrank.py](#) (Examples/)
- [lugreFrictionTest.py](#) (Examples/)
- [massSpringFrictionInteractive.py](#) (Examples/)
- [minimizeExample.py](#) (Examples/)
- [mouseInteractionExample.py](#) (Examples/)
- [nMassOscillatorInteractive.py](#) (Examples/)
- ...
- [scissorPrismaticRevolute2D.py](#) (TestModels/)
- [ANCFslidingAndALEjointTest.py](#) (TestModels/)
- [ANCFbeltDrive.py](#) (TestModels/)
- ...

## 7.6.6 ObjectConnectorGravity

A connector for adding forces due to gravitational fields between two bodies, which can be used for aerospace and small-scale astronomical problems; DO NOT USE this connector for adding gravitational forces (loads), which should be using LoadMassProportional, which is acting global and always in the same direction.

**Additional information for ObjectConnectorGravity:**

- The Object has the following types = Connector
- Requested marker type = Position
- **Short name for Python = ConnectorGravity**
- **Short name for Python (visualization object) = VConnectorGravity**

The item **ObjectConnectorGravity** with type = 'ConnectorGravity' has the following parameters:

Name	type	size	default value	description
name	String		"	connector's unique name
markerNumbers	ArrayMarkerIndex		[ MAXINT, MAX-INT ]	list of markers used in connector
gravitationalConstant	Real		6.67430e-11	gravitational constant [SI:m <sup>3</sup> kg <sup>-1</sup> s <sup>-2</sup> ]}; while not recommended, a negative constant can represent a repulsive force
mass0	UReal		0.	mass [SI:kg] of object attached to marker m0
mass1	UReal		0.	mass [SI:kg] of object attached to marker m1
minDistanceRegularization	UReal		0.	distance [SI:m] at which a regularization is added in order to avoid singularities, if objects come close
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectConnectorGravity			parameters for visualization of item

The item **VObjectConnectorGravity** has the following parameters:

Name	type	size	default value	description
show	Bool		False	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = diameter of spring; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==1, use default color

### 7.6.6.1 DESCRIPTION of ObjectConnectorGravity:

**Information on input parameters:**

input parameter	symbol	description see tables above
markerNumbers	$[m0, m1]^T$	
gravitationalConstant	$G$	
mass0	$mass_0$	
mass1	$mass_1$	
minDistanceRegularization	$d_{min}$	

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

output variable	symbol	description
Distance	$L$	distance between both points
Displacement	$\Delta^0\mathbf{p}$	relative displacement between both points
Force	$\mathbf{f}$	gravity force vector, pointing from marker $m0$ to marker $m1$

### 7.6.6.2 Definition of quantities

intermediate variables	symbol	description
marker $m0$ position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker $m0$
marker $m1$ position	${}^0\mathbf{p}_{m1}$	
marker $m0$ velocity	${}^0\mathbf{v}_{m0}$	current global velocity which is provided by marker $m0$
marker $m1$ velocity	${}^0\mathbf{v}_{m1}$	

output variables	symbol	formula
Displacement	$\Delta^0\mathbf{p}$	${}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0}$
Velocity	$\Delta^0\mathbf{v}$	${}^0\mathbf{v}_{m1} - {}^0\mathbf{v}_{m0}$
Distance	$L$	$ \Delta^0\mathbf{p} $
Force	$\mathbf{f}$	see below

### 7.6.6.3 Connector forces

The unit vector in force direction reads (if  $L = 0$ , singularity can be avoided using regularization),

$$\mathbf{v}_f = \frac{1}{L} \Delta^0\mathbf{p} \quad (7.260)$$

If `activeConnector` = True, and  $L >= d_{min}$  the gravitational force is computed as

$$f_G = -G \frac{mass_0 \cdot mass_1}{L^2} \quad (7.261)$$

If `activeConnector` = True, and  $L < d_{min}$  the gravitational force is computed as

$$f_G = -G \frac{mass_0 \cdot mass_1}{L^2 + (L - d_{min})^2} \quad (7.262)$$

which results in a regularization for small distances, which is helpful if there are no restrictions in objects to keep apart. If  $d_{min} = 0$  and  $L = 0$ , there a system error is raised.

The vector of the gravitational force applied at both markers, pointing from marker  $m0$  to marker  $m1$ , finally reads

$$\mathbf{f} = f_G \mathbf{v}_f \quad (7.263)$$

The virtual work of the connector force is computed from the virtual displacement

$$\delta\Delta^0\mathbf{p} = \delta^0\mathbf{p}_{m1} - \delta^0\mathbf{p}_{m0}, \quad (7.264)$$

and the virtual work (not the transposed version here, because the resulting generalized forces shall be a column vector,

$$\delta W_G = \mathbf{f}\delta\Delta^0\mathbf{p} = -\left(-G\frac{\text{mass}_0 \cdot \text{mass}_1}{L^2}\right)\left(\delta^0\mathbf{p}_{m1} - \delta^0\mathbf{p}_{m0}\right)^T \mathbf{v}_f. \quad (7.265)$$

The generalized (elastic) forces thus result from

$$\mathbf{Q}_G = \frac{\partial^0\mathbf{p}}{\partial\mathbf{q}_G^T}\mathbf{f}, \quad (7.266)$$

and read for the markers  $m0$  and  $m1$ ,

$$\mathbf{Q}_{G,m0} = -\left(-G\frac{\text{mass}_0 \cdot \text{mass}_1}{L^2}\right)\mathbf{J}_{pos,m0}^T \mathbf{v}_f, \quad \mathbf{Q}_{G,m1} = \left(-G\frac{\text{mass}_0 \cdot \text{mass}_1}{L^2}\right)\mathbf{J}_{pos,m1}^T \mathbf{v}_f, \quad (7.267)$$

where  $\mathbf{J}_{pos,m1}$  represents the derivative of marker  $m1$  w.r.t. its associated coordinates  $\mathbf{q}_{m1}$ , analogously  $\mathbf{J}_{pos,m0}$ .

---

#### 7.6.6.4 MINI EXAMPLE for ObjectConnectorGravity

```

mass0 = 1e25
mass1 = 1e3
r = 1e5
G = 6.6743e-11
vInit = np.sqrt(G*mass0/r)
tEnd = (r*0.5*np.pi)/vInit #quarter period
node0 = mbs.AddNode(NodePoint(referenceCoordinates = [0,0,0])) #star
node1 = mbs.AddNode(NodePoint(referenceCoordinates = [r,0,0],
                               initialVelocities=[0,vInit,0])) #satellite
oMassPoint0 = mbs.AddObject(MassPoint(nodeNumber = node0, physicsMass=mass0))
oMassPoint1 = mbs.AddObject(MassPoint(nodeNumber = node1, physicsMass=mass1))

m0 = mbs.AddMarker(MarkerNodePosition(nodeNumber=node0))
m1 = mbs.AddMarker(MarkerNodePosition(nodeNumber=node1))

mbs.AddObject(ObjectConnectorGravity(markerNumbers=[m0,m1],
                                     mass0 = mass0, mass1=mass1))

#assemble and solve system for default parameters
mbs.Assemble()
sims = exu.SimulationSettings()
sims.timeIntegration.endTime = tEnd
exu.SolveDynamic(mbs, sims, solverType=exu.DynamicSolverType.RK67)

#check result at default integration time
#expect y=x after one period of orbiting (got: 100000.00000000485)
exodynTestGlobals.testResult = mbs.GetNodeOutput(node1, exu.OutputVariableType.Position)
[1]

```

---

For examples on ObjectConnectorGravity see Examples and TestModels:

- [connectorGravityTest.py](#) (TestModels/)

## 7.6.7 ObjectConnectorRollingDiscPenalty

A (flexible) connector representing a rolling rigid disc (marker 1) on a flat surface (marker 0, ground body, not moving) in global  $x$ - $y$  plane. The connector is based on a penalty formulation and adds friction and slipping. The constraints works for discs as long as the disc axis and the plane normal vector are not parallel. Parameters may need to be adjusted for better convergence (e.g., dryFrictionProportionalZone). The formulation is still under development and needs further testing. Note that the rolling body must have the reference point at the center of the disc.

**Additional information for ObjectConnectorRollingDiscPenalty:**

- The Object has the following types = Connector
- Requested marker type = Position + Orientation
- Requested node type = GenericData
- **Short name for Python = RollingDiscPenalty**
- **Short name for Python (visualization object) = VRollingDiscPenalty**

The item **ObjectConnectorRollingDiscPenalty** with type = 'ConnectorRollingDiscPenalty' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex	2	[ MAXINT, MAXINT ]	list of markers used in connector; $m_0$ represents the ground, which can undergo translations but not rotations, and $m_1$ represents the rolling body, which has its reference point (=local position [0,0,0]) at the disc center point
nodeNumber	NodeIndex		MAXINT	node number of a NodeGenericData (size=3) for 3 dataCoordinates, needed for discontinuous iteration (friction and contact)
dryFrictionAngle	Real		0.	angle [SI:1 (rad)] which defines a rotation of the local tangential coordinates dry friction; this allows to model Mecanum wheels with specified roll angle
contactStiffness	Real		0.	normal contact stiffness [SI:N/m]
contactDamping	Real		0.	normal contact damping [SI:N/(m s)]
dryFriction	Vector2D		[0,0]	dry friction coefficients [SI:1] in local marker 1 joint J1 coordinates; if $\alpha_t == 0$ , lateral direction $l = x$ and forward direction $f = y$ ; assuming a normal force $f_n$ , the local friction force can be computed as $\begin{bmatrix} f_{t,x} \\ f_{t,y} \end{bmatrix} = \begin{bmatrix} \mu_x f_n \\ \mu_y f_n \end{bmatrix}$
dryFrictionProportionalZone	Real		0.	limit velocity [m/s] up to which the friction is proportional to velocity (for regularization / avoid numerical oscillations)

rollingFrictionViscous	Real	0.	rolling friction [SI:1], which acts against the velocity of the trail on ground and leads to a force proportional to the contact normal force; currently, only implemented for disc axis parallel to ground!
activeConnector	Bool	True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
discRadius	Real	0	defines the disc radius
planeNormal	Vector3D	[0,0,1]	normal to the contact / rolling plane (ground); Currently, this is not co-rotating with the ground body, but will do so in the future
visualization	VObjectConnectorRollingDiscPenalty		parameters for visualization of item

The item VObjectConnectorRollingDiscPenalty has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
discWidth	float		0.1	width of disc for drawing
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R== -1, use default color

### 7.6.7.1 DESCRIPTION of ObjectConnectorRollingDiscPenalty:

**Information on input parameters:**

input parameter	symbol	description see tables above
markerNumbers	$[m_0, m_1]^T$	
nodeNumber	$n_d$	
dryFrictionAngle	$\alpha_t$	
contactStiffness	$k_c$	
contactDamping	$d_c$	
dryFriction	$[\mu_x, \mu_y]^T$	
dryFrictionProportionalZone	$v_\mu$	
rollingFrictionViscous	$\mu_r$	
planeNormal	${}^0\mathbf{v}_{PN}$ , $ {}^0\mathbf{v}_{PN}  = 1$	

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

output variable	symbol	description
Position	${}^0\mathbf{p}_G$	current global position of contact point between rolling disc and ground

Velocity	${}^0\mathbf{v}_{trail}$	current velocity of the trail (according to motion of the contact point along the trail!) in global coordinates; this is not the velocity of the contact point!
VelocityLocal	${}^{J1}\mathbf{v}$	relative slip velocity at contact point in special $J1$ joint coordinates
ForceLocal	${}^{J1}\mathbf{f} = {}^0[\mathbf{f}_{t,x}, \mathbf{f}_{t,y}, \mathbf{f}_n]^T$	contact forces acting on disc, in special $J1$ joint coordinates, see section Connector Forces, $\mathbf{f}_{t,x}$ being the lateral force (parallel to ground plane), $\mathbf{f}_{t,y}$ being the longitudinal force and $\mathbf{f}_n$ being the contact normal force
RotationMatrix	${}^{0,J1}\mathbf{A} = [{}^0\mathbf{w}_{lat}, {}^0\mathbf{w}_2, {}^0\mathbf{v}_{PN}]^T$	transformation matrix of special joint $J1$ coordinates to global coordinates

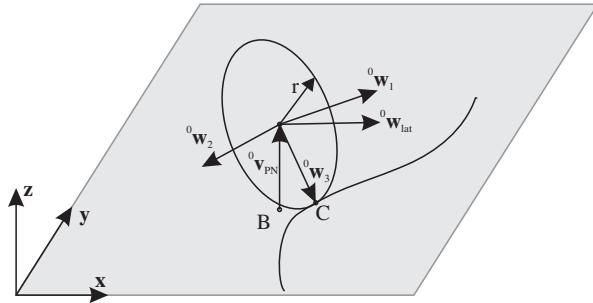
### 7.6.7.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker m0, any ground reference point; currently unused
marker m0 orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker m0; currently unused
marker m1 position	${}^0\mathbf{p}_{m1}$	center of disc
marker m1 orientation	${}^{0,m1}\mathbf{A}$	current rotation matrix provided by marker m1
data coordinates	$\mathbf{x} = [x_0, x_1, x_2]^T$	data coordinates for $[x_0, x_1]$ : hold the sliding velocity in lateral and longitudinal direction of last discontinuous iteration; $x_2$ : represents gap of last discontinuous iteration (in contact normal direction)
marker m1 velocity	${}^0\mathbf{v}_{m1}$	accordingly
marker m1 angular velocity	${}^0\boldsymbol{\omega}_{m1}$	current angular velocity vector provided by marker m1
ground normal vector	${}^0\mathbf{v}_{PN}$	normalized normal vector to the (moving, but not rotating) ground, by default [0,0,1]
ground position B	${}^0\mathbf{p}_B$	disc center point projected on ground (normal projection)
ground position C	${}^0\mathbf{p}_C$	contact point of disc with ground
ground velocity C	${}^0\mathbf{v}_C$	velocity of disc at ground contact point (must be zero at end of iteration)
wheel axis vector	${}^0\mathbf{w}_1 = {}^{0,m1}\mathbf{A} \cdot [1, 0, 0]^T$	normalized disc axis vector, currently $[1, 0, 0]^T$ in local coordinates
longitudinal vector	${}^0\mathbf{w}_2$	vector in longitudinal (motion) direction
contact point vector	${}^0\mathbf{w}_3$	normalized vector from disc center point in direction of contact point C
lateral vector	${}^0\mathbf{w}_{lat} = {}^0\mathbf{v}_{PN} \times {}^0\mathbf{w}_2$	vector in lateral direction, parallel to ground plane
$D1$ transformation matrix	${}^{0,D1}\mathbf{A} = [{}^0\mathbf{w}_1, {}^0\mathbf{w}_2, {}^0\mathbf{w}_3]^T$	transformation of special disc coordinates $D1$ to global coordinates

connector forces	${}^1\mathbf{f} = [f_{t,x}, f_{t,y}, f_n]^T$	joint force vector at contact point in joint 1 coordinates: x=lateral direction, y=longitudinal direction, z=plane normal (contact normal)
------------------	--	--

### 7.6.7.3 Geometric relations

The main geometrical setup is shown in the following figure:



First, the contact point  ${}^0\mathbf{p}_C$  must be computed. With the helper vector,

$${}^0\mathbf{x} = {}^0\mathbf{w}_1 \times {}^0\mathbf{v}_{PN} \quad (7.268)$$

we create a disc coordinate system  $D1$  ( ${}^0\mathbf{w}_1$ ,  ${}^0\mathbf{w}_2$ ,  ${}^0\mathbf{w}_3$ ), representing the longitudinal direction,

$${}^0\mathbf{w}_2 = \frac{1}{|{}^0\mathbf{x}|} {}^0\mathbf{x} \quad (7.269)$$

and the vector to the contact point,

$${}^0\mathbf{w}_3 = {}^0\mathbf{w}_1 \times {}^0\mathbf{w}_2 \quad (7.270)$$

The contact point can be computed from

$${}^0\mathbf{p}_C = {}^0\mathbf{p}_{m1} + r \cdot {}^0\mathbf{w}_3 \quad (7.271)$$

The velocity of the contact point at the disc is computed from,

$${}^0\mathbf{v}_C = {}^0\mathbf{v}_{m1} + {}^0\omega_{m1} \times (r \cdot {}^0\mathbf{w}_3) \quad (7.272)$$

A second coordinate system is defined by ( ${}^0\mathbf{w}_{lat}$ ,  ${}^0\mathbf{w}_2$ ,  ${}^0\mathbf{v}_{PN}$ ), using

$${}^0\mathbf{w}_{lat} = {}^0\mathbf{v}_{PN} \times {}^0\mathbf{w}_2 \quad (7.273)$$

Note that **in the case that** the rolling axis  ${}^0\mathbf{w}_1$  lies in the rolling plane, we obtain the special case  ${}^0\mathbf{w}_{lat} = {}^0\mathbf{w}_1$  and  ${}^0\mathbf{w}_1 = -{}^0\mathbf{v}_{PN}$ .

### 7.6.7.4 Computation of normal and tangential forces

The connector forces at the contact point C are computed as follows. The normal contact force reads

$$f_n = (k_c \cdot {}^0\mathbf{p}_C + d_c \cdot {}^0\mathbf{v}_C)^T {}^0\mathbf{v}_{PN} \quad (7.274)$$

The inplane velocity in joint coordinates,

$${}^1\mathbf{v}_t = [{}^0\mathbf{v}_C^T {}^0\mathbf{w}_{lat}, {}^0\mathbf{v}_C^T {}^0\mathbf{w}_2]^T, \quad (7.275)$$

is used for the computation of tangential forces,

$${}^J \mathbf{f}_t = [f_{t,x}, f_{t,y}]^T = {}^J \boldsymbol{\mu} \cdot (\phi(|\mathbf{v}_t|, v_\mu) \cdot f_n \cdot {}^J \mathbf{e}_t), \quad (7.276)$$

with the regularization function, see Geradin and Cardona [12] (Sec. 7.9.3):

$$\phi(v, v_\mu) = \begin{cases} \left(2 - \frac{v}{v_\mu}\right) \frac{v}{v_\mu} & \text{if } v \leq v_\mu \\ 1 & \text{if } v > v_\mu \end{cases} \quad (7.277)$$

The direction of tangential slip is given as

$${}^J \mathbf{e}_t = \begin{cases} \frac{{}^J \mathbf{v}_t}{|\mathbf{v}_t|} & \text{if } |\mathbf{v}_t| > 0 \\ \begin{bmatrix} 0 \\ 0 \end{bmatrix} & \text{else} \end{cases} \quad (7.278)$$

The friction coefficient matrix  ${}^J \boldsymbol{\mu}$  is given in joint coordinates and computed from

$${}^J \boldsymbol{\mu} = \begin{bmatrix} \mu_x & 0 \\ 0 & \mu_y \end{bmatrix} \quad (7.279)$$

where for isotropic behaviour of surface and wheel, it will give a diagonal matrix with the friction coefficient in the diagonal. In case that the dry friction angle  $\alpha_t$  is not zero, the  $\boldsymbol{\mu}$  changes to

$${}^J \boldsymbol{\mu} = \begin{bmatrix} \cos(\alpha_t) & \sin(\alpha_t) \\ -\sin(\alpha_t) & \cos(\alpha_t) \end{bmatrix} \begin{bmatrix} \mu_x & 0 \\ 0 & \mu_y \end{bmatrix} \begin{bmatrix} \cos(\alpha_t) & -\sin(\alpha_t) \\ \sin(\alpha_t) & \cos(\alpha_t) \end{bmatrix} \quad (7.280)$$

#### 7.6.7.5 Connector forces

Finally, the connector forces read in joint coordinates

$${}^J \mathbf{f} = \begin{bmatrix} f_{t,x} \\ f_{t,y} \\ f_n \end{bmatrix} \quad (7.281)$$

and in global coordinates, they are computed from

$${}^0 \mathbf{f} = f_{t,x} {}^0 \mathbf{w}_{lat} + f_{t,y} {}^0 \mathbf{w}_2 + f_n {}^0 \mathbf{v}_{PN} \quad (7.282)$$

The moment caused by the contact forces are given as

$${}^0 \mathbf{f} = (r \cdot {}^0 \mathbf{w}_3) \times {}^0 \mathbf{f} \quad (7.283)$$

Note that if `activeConnector = False`, we replace Eq. (7.281) with

$${}^J \mathbf{f} = \mathbf{0} \quad (7.284)$$

For examples on ObjectConnectorRollingDiscPenalty see Examples and TestModels:

- [`bicycleIftommBenchmark.py`](#) (Examples/)
- [`leggedRobot.py`](#) (Examples/)
- [`carRollingDiscTest.py`](#) (TestModels/)
- [`mecanumWheelRollingDiscTest.py`](#) (TestModels/)
- [`rollingCoinPenaltyTest.py`](#) (TestModels/)

## 7.6.8 ObjectContactConvexRoll

A contact connector representing a convex roll (marker 1) on a flat surface (marker 0, ground body, not moving) in global  $x$ - $y$  plane. The connector is similar to ObjectConnectorRollingDiscPenalty, but includes a (strictly) convex shape of the roll defined by a polynomial. It is based on a penalty formulation and adds friction and slipping. The formulation is still under development and needs further testing. Note that the rolling body must have the reference point at the center of the disc.

Author: Manzl Peter

### Additional information for ObjectContactConvexRoll:

- The Object has the following types = Connector
- Requested marker type = Position + Orientation
- Requested node type = GenericData

The item **ObjectContactConvexRoll** with type = 'ContactConvexRoll' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex	2	[ MAXINT, MAX-INT ]	list of markers used in connector; $m_0$ represents the ground, which can undergo translations but not rotations, and $m_1$ represents the rolling body, which has its reference point (=local position [0,0,0]) at the roll's center point
nodeNumber	NodeIndex		MAXINT	node number of a NodeGenericData (size=3) for 3 dataCoordinates, needed for discontinuous iteration (friction and contact)
contactStiffness	Real		0.	normal contact stiffness [SI:N/m]
contactDamping	Real		0.	normal contact damping [SI:N/(m s)]
dynamicFriction	UReal		0.	dynamic friction coefficient for friction model, see StribeckFunction in exudyn.physics, <a href="#">Section 5.10</a>
staticFrictionOffset	UReal		0.	static friction offset for friction model (static friction = dynamic friction + static offset), see StribeckFunction in exudyn.physics, <a href="#">Section 5.10</a>
viscousFriction	UReal		0.	viscous friction coefficient (velocity dependent part) for friction model, see StribeckFunction in exudyn.physics, <a href="#">Section 5.10</a>
exponentialDecayStatic	PReal		1e-3	exponential decay of static friction offset (must not be zero!), see StribeckFunction in exudyn.physics (named expVel there!), <a href="#">Section 5.10</a>

frictionProportionalZone	UReal	1e-3	limit velocity [m/s] up to which the friction is proportional to velocity (for regularization / avoid numerical oscillations), see StribeckFunction in exudyn.physics (named regVel there!), <a href="#">Section 5.10</a>
rollLength	UReal	0.	roll length [m], symmetric w.r.t. centerpoint
coefficientsHull	NumpyVector	[]	a vector of polynomial coefficients, which provides the polynomial of the CONVEX hull of the roll; $\text{hull}(x) = k_0 x^{n_p-1} + k_1 x^{n_p-2} + \dots + k_{n_p-2} x + k_{n_p-1}$
coefficientsHullDerivative	NumpyVector	[]	polynomial coefficients of the polynomial $\text{hull}'(x)$
coefficientsHullDDerivative	NumpyVector	[]	second derivative of the hull polynomial.
rBoundingSphere	UReal	0	The radius of the bounding sphere for the contact pre-check, calculated from the polynomial coefficients of the hull
pContact	Vector3D	[0,0,0]	The current potential contact point. Contact occurs if $\text{pContact}[2] < 0$ .
activeConnector	Bool	True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectContactConvexRoll		parameters for visualization of item

The item VObjectContactConvexRoll has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==1, use default color

### 7.6.8.1 DESCRIPTION of ObjectContactConvexRoll:

Information on input parameters:

input parameter	symbol	description see tables above
markerNumbers	$[m_0, m_1]^T$	
nodeNumber	$n_d$	
contactStiffness	$k_c$	
contactDamping	$d_c$	
dynamicFriction	$\mu_d$	
staticFrictionOffset	$\mu_{soff}$	
viscousFriction	$\mu_v$	
exponentialDecayStatic	$v_{exp}$	

frictionProportionalZone	$v_{reg}$	
rollLength	$L$	
coefficientsHull	$\mathbf{k} \in \mathbb{R}^{np}$	
coefficientsHullDerivative	$\mathbf{k}' \in \mathbb{R}^{np}$	

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

output variable	symbol	description
Position	${}^0\mathbf{p}_C$	current global position of contact point between roller and ground
Velocity	${}^0\mathbf{v}_C$	current velocity of the trail (contact) point in global coordinates; this is the velocity with which the contact moves over the ground plane
Force	${}^0\mathbf{f}$	Roll-ground force in ground coordinates
Torque	${}^0\mathbf{m}$	Roll-ground torque in ground coordinates

### 7.6.8.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker m0, any ground reference point; currently unused
marker m0 orientation	${}^{0,m0}\mathbf{A}$	current rotation matrix provided by marker m0; currently unused
marker m1 position	${}^0\mathbf{p}_{m1}$	center of roll
Contact position	${}^0\mathbf{p}_C$	Position of the Contact point C in the global frame 0
Position marker m1 to contact	${}^0\mathbf{p}_{m1,C}$	Position of the contact point C relative to the marker m1 in global frame
marker m1 orientation	${}^{0,m1}\mathbf{A}$	current rotation matrix provided by marker m1
data coordinates	$\mathbf{x} = [x_0, x_1, x_2]^T$	data coordinates for $[x_0, x_1]$ : hold the sliding velocity in lateral and longitudinal direction of last discontinuous iteration; $x_2$ : represents gap of last discontinuous iteration (in contact normal direction)
marker m1 velocity	${}^0\mathbf{v}_{m1}$	current global velocity which is provided by marker m1
marker m1 angular velocity	${}^0\boldsymbol{\omega}_{m1}$	current angular velocity vector provided by marker m1
ground normal vector	${}^0\mathbf{n}$	normalized normal vector to the (moving, but not rotating) ground, by default [0,0,1]

### 7.6.8.3 Geometric relations

The geometrical setup is shown in Fig. 7.6. To calculate the contact point of the convex body of revolution the contact (ground) plane is rotated into the local frame of the body. In this local frame in which the generatrix of

the body of revolution is described by the polynomial function

$$r(b^x) = \sum_{i=0}^n k_i x^{n-i} \quad (7.285)$$

with the coefficients of the hull  $a_i$ . As a pre-Check for the contact two spheres are put into both ends of the object with the maximum radius and only if one of these is in contact. The contact point  ${}^b p_{m1,C}$  is calculated relative to the bodies marker  $m1$  in the bodies local frame and transformed accordingly. The contact point C can for be calculated convex bodies by matching the derivative of the polynomial  $r(b^x)$  with the gradient of the contact plane, shown in Fig. 7.6, explained in detail in [28]. At the contact point a normal force  $f_N = [0 \ 0 \ f_N]^T$  with

$$f_N = \begin{cases} -(k_c z_{\text{pen}} + d_c \dot{z}_{\text{pen}}) & z_{\text{pen}} > 0 \\ 0 & \text{else} \end{cases} \quad (7.286)$$

acts against the penetration of the ground. The penetration depth  $z_{\text{pen}}$  is the z-component of the position vector of the contact point relative to the ground frame  ${}^0 p_C$ .

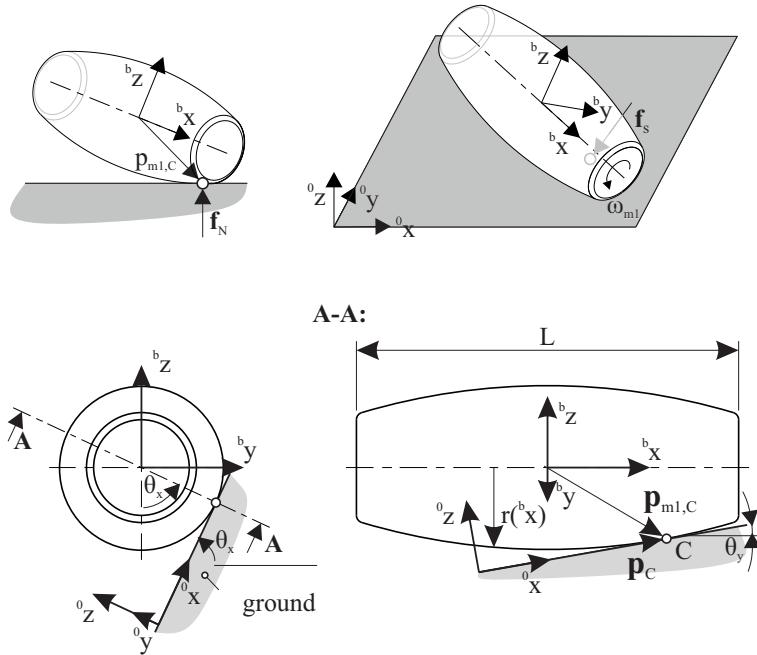


Figure 7.6: Sketch of the roller Dimensions. The rollers radius  $r(b^x)$  is described by the polynomial **coefficientsHull**.

The revolution results in a velocity of

$${}^0 v_C = {}^0 \omega_{m1} \times {}^0 p_{m1,C} \quad (7.287)$$

in the contact point, while the tangential component of the velocity of the body itself with the normal Vector to the contact plane  $\mathbf{n}$  follows to

$${}^0 v_{m1,t} = {}^0 v_{m1} - {}^0 \mathbf{n} \left( {}^0 \mathbf{n}^T {}^0 v_{m1} \right). \quad (7.288)$$

Therefore the slip velocity of the body can be calculated with

$${}^0 v_s = {}^0 v_C - {}^0 v_{m1,t} \quad (7.289)$$

and points in the direction

$${}^0\mathbf{r}_s = \frac{1}{\|{}^0\mathbf{v}_s\|} {}^0\mathbf{v}_s. \quad (7.290)$$

The slip force is then calculated

$${}^0\mathbf{f}_s = \mu(\|{}^0\mathbf{v}_s\|) f_N {}^0\mathbf{r}_s \quad (7.291)$$

and uses for the friction coefficient  $\mu$  the regularized friction approach from the StribeckFunction, see [Section 5.10](#).

The torque

$${}^0\boldsymbol{\tau} = {}^0\mathbf{p}_{m1,C} \times ({}^0\mathbf{f}_N + {}^0\mathbf{f}_s) \quad (7.292)$$

acts onto the body, resulting from the slip force acting not in the bodies center.

For examples on ObjectContactConvexRoll see Examples and TestModels:

- [ConvexContactTest.py](#) (TestModels/)

### 7.6.9 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 activated only, if  $dataCoordinate[0] \leq 0$ ;  $dataCoordinate$  is set equal to  $gap$  in nonlinear iterations, but not modified in Newton iterations.

**Additional information for ObjectContactCoordinate:**

- The Object has the following types = Connector
- Requested marker type = Coordinate
- Requested node type = GenericData

The item **ObjectContactCoordinate** with type = 'ContactCoordinate' has the following parameters:

Name	type	size	default value	description
name	String		"	connector's unique name
markerNumbers	ArrayMarkerIndex		[ MAXINT, MAX-INT ]	markers define contact gap
nodeNumber	NodeIndex		MAXINT	node number of a NodeGenericData for 1 dataCoordinate (used for active set strategy ==> holds the gap of the last discontinuous iteration)
contactStiffness	UReal		0.	contact (penalty) stiffness [SI:N/m]; acts only upon penetration
contactDamping	UReal		0.	contact damping [SI:N/(m s)]; acts only upon penetration
offset	Real		0.	offset [SI:m] of contact
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectContactCoordinate			parameters for visualization of item

The item **VObjectContactCoordinate** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = diameter of spring; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R== -1, use default color

### **7.6.9.1 DESCRIPTION of ObjectContactCoordinate:**

---

For examples on ObjectContactCoordinate see Examples and TestModels:

- [ANCF\\_contact\\_circle.py](#) (Examples/)
- [ANCF\\_contact\\_circle2.py](#) (Examples/)
- [ANCFcontactCircleTest.py](#) (TestModels/)
- [contactCoordinateTest.py](#) (TestModels/)

### 7.6.10 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 coordinates 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 activated only, if  $dataCoordinate[0] \leq 0$ ; dataCoordinate is set equal to gap in nonlinear iterations, but not modified in Newton iterations.

**Additional information for ObjectContactCircleCable2D:**

- The Object has the following types = Connector
- Requested marker type = \_None
- Requested node type = GenericData

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

Name	type	size	default value	description
name	String		"	connector's unique name
markerNumbers	ArrayMarkerIndex		[ MAXINT, MAX-INT ]	markers define contact gap
nodeNumber	NodeIndex		MAXINT	node number of a NodeGenericData for nSegments dataCoordinates (used for active set strategy ==> hold the gap of the last discontinuous iteration and the friction state)
numberOfContactSegments	Index		3	number of linear contact segments to determine contact; each segment is a line and is associated to a data (history) variable; must be same as in according marker
contactStiffness	UReal		0.	contact (penalty) stiffness [SI:N/m/(contact segment)]; the stiffness is per contact segment; specific contact forces (per length) $f_N$ act in contact normal direction only upon penetration
contactDamping	UReal		0.	contact damping [SI:N/(m s)/(contact segment)]; the damping is per contact segment; acts in contact normal direction only upon penetration
circleRadius	UReal		0.	radius [SI:m] of contact circle
offset	Real		0.	offset [SI:m] of contact, e.g. to include thickness of cable element
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectContactCircleCable2D			parameters for visualization of item

The item VObjectContactCircleCable2D has the following parameters:

<b>Name</b>	<b>type</b>	<b>size</b>	<b>default value</b>	<b>description</b>
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
showContactCircle	Bool		True	if True and show=True, the underlying contact circle is shown; uses circleTiling*4 for tiling (from VisualizationSettings.general)
drawSize	float		-1.	drawing size = diameter of spring; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R===-1, use default color

---

#### 7.6.10.1 DESCRIPTION of ObjectContactCircleCable2D:

#### 7.6.10.2 Connector equations

Geometry and equations are very similar to ObjectContactFrictionCircleCable2D, while friction is not used and no torque is transferred to the circle object.

---

For examples on ObjectContactCircleCable2D see Examples and TestModels:

- [ANCF\\_contact\\_circle.py](#) (Examples/)
- [ANCF\\_contact\\_circle2.py](#) (Examples/)
- [ANCF\\_moving\\_rigidbody.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- [ACNFslidingAndALEjointTest.py](#) (TestModels/)
- [ANCFcontactCircleTest.py](#) (TestModels/)
- [ANCFmovingRigidBodyTest.py](#) (TestModels/)

### 7.6.11 ObjectContactFrictionCircleCable2D

A very specialized penalty-based contact/friction condition between a 2D circle in the local x/y plane (=marker0, a Rigid-Body Marker) on a body and an ANCFCircle2DShape (=marker1, Marker: BodyCircle2DShape), in xy-plane; a node NodeGenericData is required with  $3 \times (\text{number of contact segments})$  – containing per segment: [contact gap, stick/slip (stick=0, slip=+1, undefined=-2), last friction position].

**Additional information for ObjectContactFrictionCircleCable2D:**

- The Object has the following types = Connector
- Requested marker type = \_None
- Requested node type = GenericData

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

Name	type	size	default value	description
name	String		"	connector's unique name
markerNumbers	ArrayMarkerIndex		[ MAXINT, MAX-INT ]	a marker $m_0$ with position and orientation and a marker $m_1$ of type BodyCable2DShape; together defining the contact geometry
nodeNumber	NodeIndex		MAXINT	node number of a NodeGenericData with $3 \times n_{cs}$ dataCoordinates (used for active set strategy → hold the gap of the last discontinuous iteration, friction state (+1=slip, 0=stick, -2=undefined) and the last sticking position; initialize coordinates with list $[0.1]*n_{cs} + [-2]*n_{cs} + [0.]*n_{cs}$ , meaning that there is no initial contact with undefined slip/stick)
numberOfContactSegments	PInt		3	number of linear contact segments to determine contact; each segment is a line and is associated to a data (history) variable; must be same as in according marker
contactStiffness	UReal		0.	contact (penalty) stiffness [SI:N/m/(contact segment)]; the stiffness is per contact segment; specific contact forces (per length) $f_n$ act in contact normal direction only upon penetration
contactDamping	UReal		0.	contact damping [SI:N/(m s)/(contact segment)]; the damping is per contact segment; acts in contact normal direction only upon penetration
frictionVelocityPenalty	UReal		0.	tangential velocity dependent penalty coefficient for friction [SI:N/(m s)/(contact segment)]; the coefficient causes tangential (contact) forces against relative tangential velocities in the contact area

frictionStiffness	UReal	0.	tangential displacement dependent penalty/stiffness coefficient for friction [SI:N/m/(contact segment)]; the coefficient causes tangential (contact) forces against relative tangential displacements in the contact area
frictionCoefficient	UReal	0.	friction coefficient [SI: 1]; tangential specific friction forces (per length) $f_t$ must fulfill the condition $f_t \leq \mu f_n$
circleRadius	UReal	0.	radius [SI:m] of contact circle
useSegmentNormals	Bool	True	True: use normal and tangent according to linear segment; this is appropriate for very long (compared to circle) segments; False: use normals at segment points according to vector to circle center; this is more consistent for short segments, as forces are only applied in beam tangent and normal direction
activeConnector	Bool	True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectContactFrictionCircleCable2D		parameters for visualization of item

The item VObjectContactFrictionCircleCable2D has the following parameters:

Name	type	size	default value	description
show	Bool		True	set True, if item is shown in visualization and false if it is not shown; note that only normal contact forces can be drawn, which are approximated by $k_c \cdot g$ (neglecting damping term)
showContactCircle	Bool		True	if True and show=True, the underlying contact circle is shown; uses circleTiling*4 for tiling (from VisualizationSettings.general)
drawSize	float		-1.	drawing size = diameter of spring; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==1, use default color

#### 7.6.11.1 DESCRIPTION of ObjectContactFrictionCircleCable2D:

Information on input parameters:

input parameter	symbol	description see tables above
-----------------	--------	------------------------------

markerNumbers	$[m0, m1]^T$	
nodeNumber	$n_g$	
numberOfContactSegments	$n_{cs}$	
contactStiffness	$k_c$	
contactDamping	$d_c$	
frictionVelocityPenalty	$\mu_v$	
frictionStiffness	$\mu_k$	
frictionCoefficient	$\mu$	
circleRadius	$r$	

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

output variable	symbol	description
Coordinates	$[u_{t,0}, g_0, u_{t,1}, g_1, \dots, u_{t,n_{cs}}, g_{n_{cs}}]^T$	local (relative) displacement in tangential ( $t$ ) and normal ( $n$ ) direction per segment ( $n_{cs}$ ); values are only provided in case of contact, otherwise zero; tangential displacement is only non-zero in case of sticking!
Coordinates_t	$[v_{t,0}, v_{n,0}, v_{t,1}, v_{n,1}, \dots, v_{t,n_{cs}}, v_{n,n_{cs}}]^T$	local (relative) velocity in tangential ( $t$ ) and normal ( $n$ ) direction per segment ( $n_{cs}$ ); values are only provided in case of contact, otherwise zero
ForceLocal	$[f_{t,0}, f_{n,0}, f_{t,1}, f_{n,1}, \dots, f_{t,n_{cs}}, f_{n,n_{cs}}]^T$	local contact forces in tangential ( $t$ ) and normal ( $n$ ) direction per segment ( $n_{cs}$ )

### 7.6.11.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^0\mathbf{p}_{m0}$	represents current global position of the circle's centerpoint
marker m0 velocity	${}^0\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
marker m1		represents the 2D ANCF cable
data node	$\mathbf{x} = [x_i, \dots, x_{3n_{cs}-1}]^T$	coordinates of node with node number $n_{GD}$
data coordinates for segment $i$	$[x_i, x_{n_{cs}+i}, x_{2n_{cs}+i}]^T$ $[x_{gap}, x_{isSlipStick}, x_{lastStick}]^T$ , $i \in [0, n_{cs} - 1]$	= with The data coordinates include the gap $x_{gap}$ , the stick-slip state $x_{isSlipStick}$ and the previous sticking position $x_{lastStick}$ as computed in the PostNewtonStep, see description below.
shortest distance to segment $s_i$	$\mathbf{d}_{g,i}$	shortest distance of center of circle to contact segment, considering the endpoint of the segment

### 7.6.11.3 Connector forces: contact geometry

The connector represents a force element between a 'circle' (or cylinder) represented by a marker  $m0$ , which has position and orientation, and an ANCFBeam2D beam element (denoted as 'cable') represented by a MarkerBodyCable2DShape  $m1$ . The cable with reference length  $L$  is discretized by splitting into  $n_{cs}$  straight

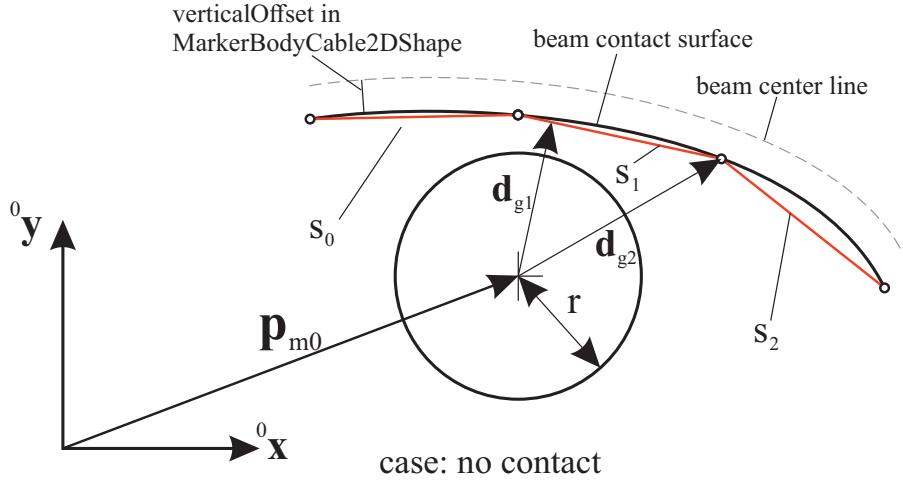


Figure 7.7: Sketch of cable, contact segments and circle; case A shows contact with  $|\mathbf{d}_{g1}| > r$ , while case B shows contact with  $|\mathbf{d}_{g1}| \leq r$ ; the shortest distance vector  $\mathbf{d}_{g1}$  is related to segment  $s_1$  (which is perpendicular to the segment line) and  $\mathbf{d}_{g2}$  is the shortest distance to the end point of segment  $s_2$ , not being perpendicular.

segments  $s_i$ , located between points  $\mathbf{p}_i$  and  $\mathbf{p}_{i+1}$ . Note that these points can be placed with an offset from the cable centerline, see `verticalOffset` defined in `MarkerBodyCable2DShape`. In order to compute the gap function for a line segment, the shortest distance of one line segment with points  $\mathbf{p}_i, \mathbf{p}_{i+1}$ , the circle centerpoint given by the marker  $\mathbf{p}_{m0}$  are computed, all in the global coordinates system (0), including edge points of every segment.

With the intermediate quantities (all of them related to segment  $s_i$ )<sup>6</sup>,

$$\mathbf{v}_s = \mathbf{p}_{i+1} - \mathbf{p}_i, \quad \mathbf{v}_p = \mathbf{p}_{m0} - \mathbf{p}_i, \quad n = \mathbf{v}_s^T \mathbf{v}_p, \quad d = \mathbf{v}_s^T \mathbf{v}_s \quad (7.293)$$

and assuming that  $d \neq 0$  (otherwise the two segment points would be identical and the shortest distance would be  $d_g = |\mathbf{v}_p|$ ), we find the relative position  $\rho$  of the shortest (projected) point on the segment, which runs from 0 to 1 if lying on the segment, as

$$\rho = \frac{n}{d} \quad (7.294)$$

We distinguish 3 cases (see also Fig. 7.7 for cases 1 and 2):

1. If  $\rho \leq 0$ , the shortest distance would be the distance to point  $\mathbf{p}_p = \mathbf{p}_i$ , reading

$$d_g = |\mathbf{p}_{m0} - \mathbf{p}_i| \quad (\rho \leq 0) \quad (7.295)$$

2. If  $\rho \geq 1$ , the shortest distance would be the distance to point  $\mathbf{p}_p = \mathbf{p}_{i+1}$ , reading

$$d_g = |\mathbf{p}_{m0} - \mathbf{p}_{i+1}| \quad (\rho \geq 1) \quad (7.296)$$

3. Finally, if  $0 < \rho < 1$ , then the shortest distance has a projected point somewhere on the segment with the point (projected on the segment)

$$\mathbf{p}_p = \mathbf{p}_i + \rho \cdot \mathbf{v}_s \quad (7.297)$$

and the distance

$$d_g = |\mathbf{d}_g| = \sqrt{\mathbf{v}_p^T \mathbf{v}_p - (n^2)/d} \quad (7.298)$$

<sup>6</sup>we omit  $s_i$  in some terms for brevity!

Here, the shortest distance vector for every segment results from the projected point  $\mathbf{p}_p$  of the above mentioned cases, see also Fig. 7.7, with the relation

$$\mathbf{d}_g = \mathbf{d}_{g,s_i} = \mathbf{p}_{m0} - \mathbf{p}_p . \quad (7.299)$$

The contact gap for a specific point for segment  $s_i$  is in general defined as

$$g = g_{s_i} = d_g - r . \quad (7.300)$$

using  $d_g = |\mathbf{d}_g|$ .

#### 7.6.11.4 Contact frame and relative motion

Irrespective of the choice of `useSegmentNormals`, the contact normal vector  $\mathbf{n}_{s_i}$  and tangential vector  $\mathbf{t}_{s_i}$  are defined per segment as

$$\mathbf{n}_{s_i} = \mathbf{n} = [n_0, n_1]^T = \frac{1}{|\mathbf{d}_{g,s_i}|} \mathbf{d}_{g,s_i}, \quad \mathbf{t}_{s_i} = \mathbf{t} = [-n_1, n_0]^T \quad (7.301)$$

The vectors  $\mathbf{t}_{s_i}$  and  $\mathbf{n}_{s_i}$  define the local (contact) frame for further computations.

The velocity at the closest point of the segment  $s_i$  is interpolated using  $\rho$  and computed as

$$\dot{\mathbf{p}}_p = (1 - \rho) \cdot \mathbf{v}_i + \rho \cdot \mathbf{v}_{i+1} \quad (7.302)$$

Alternatively,  $\dot{\mathbf{p}}_p$  could be computed from the cable element by evaluating the velocity at the contact points, but we feel that this choice is more consistent with the computations at position level.

The gap velocity  $v_n$  ( $\neq \dot{g}$ ) thus reads

$$v_n = (\dot{\mathbf{p}}_p - \dot{\mathbf{p}}_{m0}) \mathbf{n} \quad (7.303)$$

In a similar, the tangential velocity reads

$$v_t = (\dot{\mathbf{p}}_p - \dot{\mathbf{p}}_{m0}) \mathbf{t} \quad (7.304)$$

In case of `frictionStiffness != 0`, we continuously track the sticking position at which the cable element (or segment) and the circle previously stucked together, similar as proposed by Ladrás et al. [27]. The difference here to the latter reference, is that we explicitly exclude switching from Newton's method and that Ladrás et al. used contact points, while we use linear segments. For a simple 1D example using this position based approach for friction, see `Examples/lugreFrictionText.py`, which compares the traditional LuGre friction model [9] with the position based model with tangential stiffness.

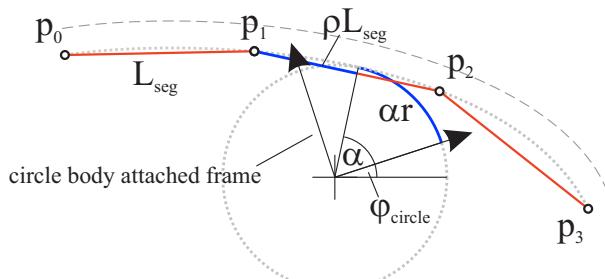


Figure 7.8: Calculation of last sticking position; blue parts mark the sticking position calculated as  $x_{curStick}^*$ .

Because there is the chance to wind/unwind relative to the (last) sticking position without slipping, the following strategy is used. In case of sliding (which could be the last time sliding before sticking), we compute the **current sticking position**, see Fig. 7.8, as the sum of the relative position at the segment  $s$

$$x_{s,curStick} = \rho \cdot L_{seg} \quad (7.305)$$

in which  $\rho \in [0, 1]$  denotes the relative position of contact at the segment with reference length  $L_{seg} = \frac{L}{n_{cs}}$ . The relative position at the circle  $c$  is

$$x_{c,curStick} = \alpha \cdot r \quad (7.306)$$

We immediately see, that under pure rolling<sup>7</sup>,

$$x_{s,curStick} + x_{c,curStick} = \text{const.} \quad (7.307)$$

Note that the `verticalOffset` from the cable center line, as defined in the related `MarkerBodyCable2DShape`, influences the behavior significantly, which is why we recommend to use `verticalOffset=0` whenever this is an appropriate assumption. Thus, the current sticking position  $x_{curStick}$  is computed per segment as

$$x_{curStick}^* = x_{s,curStick} + x_{c,curStick}, \quad (7.308)$$

Due to the possibility of switching of  $\alpha + \phi$  between  $-pi$  and  $\pi$ , the result is normalized to

$$x_{curStick} = x_{curStick}^* - \text{floor}\left(\frac{x_{curStick}^*}{2\pi \cdot r} + \frac{1}{2}\right) \cdot 2\pi \cdot r, \quad (7.309)$$

which gives  $\bar{x}_{curStick} \in [-\pi \cdot r, \pi \cdot r]$  is stored in the 3rd data variable (per segment). The function `floor()` is a standardized version of rounding, available in C and Python programming languages. In the `PostNewtonStep`, the last sticking position is computed,  $x_{lastStick} = x_{curStick}$ , and it is also available in the `startOfStep` state.

### 7.6.11.5 Contact forces: definition

The contact force  $f_n$  is zero for  $g > 0$  and otherwise computed from

$$f_n = k_c \cdot g + d_c \cdot v_n \quad (7.310)$$

NOTE that currently, there is only a linear spring-damper model available, assuming that the impact dynamics is not dominating (such as in belt drives or reeving systems).

Friction forces are primarily based on relative (tangential) velocity at each segment. The 'linear' friction force, based on the velocity penalty parameter  $\mu_v$  reads

$$f_t^{(lin)} = \mu_v \cdot v_t, \quad (7.311)$$

### 7.6.11.6 PostNewtonStep

In general, see the solver flow chart for the `DiscontinuousIteration`, see Fig. 11.5, should be considered when reading this description. Every step is started with values `startOfStep`, while current values are iterated and updated in the Newton or `DiscontinuousIteration`.

The `PostNewtonStep` computes 3 values per segment, which are used for computation of contact forces, irrespectively of the current geometry of the contact. The `PostNewtonStep` is called after every full Newton method and evaluates the current state w.r.t. the assumed data variables. If the assumptions do not fit, new data variables are computed. This is necessary in order to avoid discontinuities in the equations, while otherwise the Newton iterations would not (or only slowly) converge.

The data variables per segment are

$$[x_{gap}, x_{isSlipStick}, x_{lastStick}] \quad (7.312)$$

Here,  $x_{gap}$  contains the gap of the segment ( $\leq 0$  means contact),  $x_{lastStick}$  is described in Eq. (7.309), and  $x_{isSlipStick}$  defines the stick or slip case,

---

<sup>7</sup>neglecting the effects of small penetration, usually much smaller than shown for visibility in Fig. 7.8.

- $x_{isSlipStick} = -2$ : undefined, used for initialization
- $x_{isSlipStick} = 0$ : sticking
- $x_{isSlipStick} = \pm 1$ : slipping, sign defines slipping direction

The basic algorithm in the `PostNewtonStep`, with all operations given for any segment  $s_i$ , can be summarized as follows:

- Evaluated gap per segment  $g$  using Eq. (7.300) and store in data variable:  $x_{gap} = g$
- If  $x_{gap} < 0$  and ( $\mu_v \neq 0$  or  $\mu_k \neq 0$ ):

1. Compute contact force  $f_n$  according to Eq. (7.310)
2. Compute current sticking position  $x_{curStick}$  according to Eq. (7.308)<sup>8</sup>
3. Retrieve `startOfStep` sticking position<sup>9</sup> in  $x_{lastStick}^{startOfStep}$  and compute and normalize difference in sticking position<sup>10</sup>:

$$\Delta x_{stick}^* = x_{curStick} - x_{lastStick}^{startOfStep}, \quad \Delta x_{stick} = x_{stick}^* - \text{floor}\left(\frac{\Delta x_{stick}^*}{2\pi \cdot r} + \frac{1}{2}\right) \cdot 2\pi \cdot r \quad (7.313)$$

4. Compute linear tangential force for friction stiffness and velocity penalty:

$$f_{t,lin} = \mu_v \cdot v_t + \mu_k \Delta x_{stick} \quad (7.314)$$

5. Compute tangential force according to Coulomb friction model<sup>11</sup>:

$$f_t = \begin{cases} f_t^{(lin)}, & \text{if } |f_t^{(lin)}| \leq \mu \cdot |f_n| \\ \mu \cdot |f_n| \cdot \text{Sign}(\Delta x_{stick}), & \text{else} \end{cases} \quad (7.315)$$

6. In the case of slipping, given by  $|f_t^{(lin)}| > \mu \cdot |f_n|$ , we update the last sticking position in the data variable, such that the spring is pre-tensioned already,

$$x_{lastStick} = x_{curStick} - \text{Sign}(\Delta x_{stick}) \frac{\mu \cdot |f_n|}{\mu_k}, \quad x_{isSlipStick} = \text{Sign}(\Delta x_{stick}) \quad (7.316)$$

7. In the case of sticking, given by  $|f_t^{(lin)}| \leq \mu \cdot |f_n|$ : Set  $x_{isSlipStick} = 0$  and, if  $x_{isSlipStick}^{startOfStep} = -2$  (undefined), we update  $x_{lastStick} = x_{curStick}$ , while otherwise,  $x_{lastStick}$  is unchanged.

- If  $x_{gap} > 0$  or ( $\mu_v == 0$  and  $\mu_k == 0$ ), we set  $x_{isSlipStick} = -2$  (undefined); this means that in the next step (if this step is accepted), there is no stored sticking position.

- Compute an error  $\varepsilon_{PNS} = \varepsilon_{PNS}^n + \varepsilon_{PNS}^t$ , with physical units forces (per segment point), for `PostNewtonStep`:

1. if gap  $x_{gap, lastPNS}$  of previous `PostNewtonStep` had different sign to current gap, set

$$\varepsilon_{PNS}^n = k_c \cdot \|x_{gap} - x_{gap, lastPNS}\| \quad (7.317)$$

while otherwise  $\varepsilon_{PNS}^n = 0$ .

2. if stick-slip-state  $x_{isSlipStick, lastPNS}$  of previous `PostNewtonStep` is different from current  $x_{isSlipStick}$ , set

$$\varepsilon_{PNS}^t = \|\left(\|f_t^{(lin)}\| - \mu \cdot |f_n|\right)\| \quad (7.318)$$

while otherwise  $\varepsilon_{PNS}^t = 0$ .

---

<sup>8</sup>terms are only evaluated if  $\mu_k \neq 0$

<sup>9</sup>Importantly, the `PostNewtonStep` always refers to the `startOfStep` state in the sticking position, because in the discontinuous iterations, the algorithm could switch to slipping in between and override the last sticking position in the current step

<sup>10</sup>in case that  $x_{isSlipStick} = -2$ , meaning that there is no stored sticking position, we set  $\Delta x_{stick} = 0$

<sup>11</sup>note that the sign of  $\Delta x_{stick}$  is used here, but alternatively we may also use the sign of  $f_{t,lin}$

Note that the PostNewtonStep is iterated and the data variables are updated continuously until convergence, or until a max. number of iterations is reached. If ignoreMaxIterations == 0, computation will continue even if no convergence is reached after the given number of iterations. This will lead to larger errors in such steps, but may have less influence on the overall solution if such cases are rare.

### 7.6.11.7 Computation of connector forces in Newton

The computation of LHS terms, the action of forces produced by the contact-friction element, is done during Newton iterations and may not have discontinuous behavior, thus relating computations to data variables computed in the PostNewtonStep. For efficiency, the LHS computation is only performed, if the PostNewtonStep determined contact in any segment.

The algorithm reads is similar to the previous subsection. The following operations are performed for each segment  $s_i$ , if  $x_{gap,s_i} \leq 0$ :

I. Compute contact force  $f_n$ , Eq. (7.310).

II. In case of sticking:

II.1 the current sticking position  $x_{curStick}$  is computed from Eq. (7.308), and the difference of current and last sticking position reads<sup>12</sup>:

$$\Delta x_{stick}^* = x_{curStick} - x_{lastStick} \quad \Delta x_{stick} = x_{stick}^* - \text{floor}\left(\frac{\Delta x_{stick}^*}{2\pi \cdot r} + \frac{1}{2}\right) \cdot 2\pi \cdot r \quad (7.319)$$

II.2 however, if the friction stiffness is  $\mu_k == 0$  or if  $x_{isSlipStick} == -2$ , we also set  $\Delta x_{stick} = 0$

II.3 using the tangential velocity from Eq. (7.304), the linear tangent force follows as

$$f_{t,lin} = \mu_v \cdot v_t + \mu_k \Delta x_{stick} \quad (7.320)$$

II.4 the tangential friction force then results in<sup>13</sup>,

$$f_t = \begin{cases} f_t^{(lin)}, & \text{if } \|x_{isSlipStick}\| \neq 1 \\ \mu \cdot |f_n| \cdot x_{isSlipStick}, & \text{else} \end{cases} \quad (7.321)$$

### 7.6.11.8 Computation of LHS terms for circle and circle

If activeConnector = True, contact forces  $f_i$  with  $i \in [0, n_{cs}]$  – these are  $(n_{cs} + 1)$  forces – are applied at the points  $p_i$ , and they are computed for every contact segments (i.e., two segments may contribute to contact forces of one point). For every contact computation, first all contact forces at segment points are set to zero. We distinguish two cases SN and PWN. If useSegmentNormals==True, we use the SN case, while otherwise the PWN case is used, compare Fig. 7.9.

Segment normals (=SN) lead to always good approximations for normal directions, irrespectively of short or extremely long segments as compared to the circle. However, in case of segments that are short as compared to the circle radius, normals computed from the center of the circle to the segment points (=PWN) are more consistent and produce tangents only in circumferential direction, which may improve behavior in some applications. The equations for the two cases read:

**CASE SN: use Segment Normals**

If there is contact in a segment  $s_i$ , i.e., gap state  $x_{gap} \leq 0$ , see Fig. 7.7(right), contact forces  $f_{s_i}$  are computed per segment,

$$f_{s_i} = f_n \cdot \mathbf{n}_{s_i} + f_t \mathbf{t}_{s_i} \quad (7.322)$$

<sup>12</sup>see the difference to the PostNewtonStep: we use  $x_{lastStick}$  here, not the startOfStep variant.

<sup>13</sup>see again difference to PostNewtonStep!

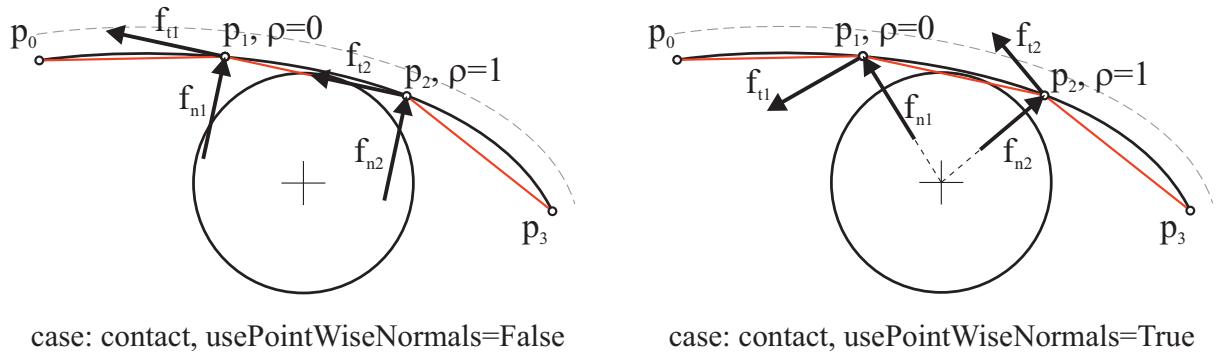


Figure 7.9: Choice of normals and tangent vectors for calculation of normal contact forces and tangential (friction) forces; note that the `useSegmentNormals=False` is not appropriate for this setup and would produce highly erroneous forces.

and added to every force at segment points according to

$$\begin{aligned} \mathbf{f}_i &+= (1 - \rho) \cdot \mathbf{f}_{s_i} \\ \mathbf{f}_{i+1} &+= \rho \cdot \mathbf{f}_{s_i} \end{aligned} \quad (7.323)$$

while in case  $x_{gap} > 0$  nothing is added.

**CASE PWN:** use Point Wise Normals (at segment points)

If there is contact in a segment  $s_i$ , i.e., gap  $x_{gap} \leq 0$ , see Fig. 7.7(right), intermediate contact forces  $\mathbf{f}_i^{l,r}$  are computed per segment point,

$$\mathbf{f}^l = f_n \cdot \mathbf{n}_{l,s_i} + f_t \mathbf{t}_{l,s_i}, \quad \mathbf{f}^r = f_n \cdot \mathbf{n}_{r,s_i} + f_t \mathbf{t}_{r,s_i} \quad (7.324)$$

in which  $\mathbf{n}_{l,s_i}$  is the vector from circle center to the left point ( $i$ ) of the segment  $s_i$ , and  $\mathbf{n}_{r,s_i}$  to the right point ( $i + 1$ ). The tangent vectors are perpendicular to the normals. The forces are then applied to the contact forces  $\mathbf{f}_i$  using the parameter  $\rho$ , which takes into account the distance of contact to the left or right side of the segment,

$$\begin{aligned} \mathbf{f}_i &+= (1 - \rho) \cdot \mathbf{f}^l \\ \mathbf{f}_{i+1} &+= \rho \cdot \mathbf{f}^r \end{aligned} \quad (7.325)$$

while in case  $x_{gap} > 0$  nothing is added.

The forces  $\mathbf{f}_i$  are then applied through the marker to the `ObjectANFCable2D` element as point loads via a position jacobian (using the according access function), for details see the C++ implementation.

The forces on the circle marker  $m0$  are computed as the total sum of all segment contact forces,

$$\mathbf{f}_{m0} = - \sum_{s_i} \mathbf{f}_{s_i} \quad (7.326)$$

and additional torques on the circle's rotation simply follow from

$$\tau_{m0} = - \sum_{s_i} r \cdot f_{t_{s_i}}. \quad (7.327)$$

During Newton iterations, the contact forces for segment  $s_i$  are considered only, if  $x_i \leq 0$ . The `dataCoordinate`  $x_i$  is not modified during Newton iterations, but computed during the `DiscontinuousIteration`, see Fig. 11.5 in

the solver description.

If **activeConnector = False**, all contact and friction forces on the cable and the force and torque on the circle's marker are set to zero.

---

For examples on ObjectContactFrictionCircleCable2D see Examples and TestModels:

- [beltDriveReevingSystem.py](#) (Examples/)
- [sliderCrank3DwithANCFbeltDrive.py](#) (Examples/)
- [sliderCrank3DwithANCFbeltDrive2.py](#) (Examples/)
- [ANCFslidingAndALEjointTest.py](#) (TestModels/)
- [ANCFcontactFrictionTest.py](#) (TestModels/)
- [ANCFmovingRigidBodyTest.py](#) (TestModels/)

## 7.7 Objects (Constraint)

A Constraint is a special Object and Connector, which links two or more markers. A Constraint leads to algebraic equations, which exactly fulfill special constraints on the kinematic behavior of the multibody system, such as a constraint on a coordinate or a distance constraint.

### 7.7.1 ObjectConnectorDistance

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

**Additional information for ObjectConnectorDistance:**

- The Object has the following types = Connector, Constraint
- Requested marker type = Position
- **Short name for Python = DistanceConstraint**
- **Short name for Python (visualization object) = VDistanceConstraint**

The item **ObjectConnectorDistance** with type = 'ConnectorDistance' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex		[ MAXINT, MAX-INT ]	list of markers used in connector
distance	UReal		0.	prescribed distance [SI:m] of the used markers
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectConnectorDistance			parameters for visualization of item

The item **VObjectConnectorDistance** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = link size; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==1, use default color

#### 7.7.1.1 DESCRIPTION of ObjectConnectorDistance:

**Information on input parameters:**

input parameter	symbol	description see tables above
-----------------	--------	------------------------------

markerNumbers	$[m0, m1]^T$	
distance	$d_0$	

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

output variable	symbol	description
Displacement	${}^0\Delta\mathbf{p}$	relative displacement in global coordinates
Velocity	${}^0\Delta\mathbf{v}$	relative translational velocity in global coordinates
Distance	$ {}^0\Delta\mathbf{p} $	distance between markers (should stay constant; shows constraint deviation)
Force	$\lambda_0$	joint force (=scalar Lagrange multiplier)

### 7.7.1.2 Definition of quantities

intermediate variables	symbol	description
marker m0 position	${}^0\mathbf{p}_{m0}$	current global position which is provided by marker m0
marker m1 position	${}^0\mathbf{p}_{m1}$	accordingly
marker m0 velocity	${}^0\mathbf{v}_{m0}$	current global velocity which is provided by marker m0
marker m1 velocity	${}^0\mathbf{v}_{m1}$	accordingly
relative displacement	${}^0\Delta\mathbf{p}$	${}^0\mathbf{p}_{m1} - {}^0\mathbf{p}_{m0}$
relative velocity	${}^0\Delta\mathbf{v}$	${}^0\mathbf{v}_{m1} - {}^0\mathbf{v}_{m0}$
algebraicVariable	$\lambda_0$	Lagrange multiplier = force in constraint

### 7.7.1.3 Connector forces constraint equations

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

$$|{}^0\Delta\mathbf{p}| - d_0 = 0 \quad (7.328)$$

The index 2 (velocity level) algebraic equation reads

$$\left( \frac{{}^0\Delta\mathbf{p}}{|{}^0\Delta\mathbf{p}|} \right)^T \Delta\mathbf{v} = 0 \quad (7.329)$$

if `activeConnector` = `False`, the algebraic equation reads

$$\lambda_0 = 0 \quad (7.330)$$

### 7.7.1.4 MINI EXAMPLE for ObjectConnectorDistance

```
#example with 1m pendulum, 50kg under gravity
nMass = mbs.AddNode(NodePoint2D(referenceCoordinates=[1,0]))
oMass = mbs.AddObject(MassPoint2D(physicsMass = 50, nodeNumber = nMass))
```

```

mMass = mbs.AddMarker(MarkerNodePosition(nodeNumber=nMass))
mGround = mbs.AddMarker(MarkerBodyPosition(bodyNumber=oGround, localPosition = [0,0,0]))
oDistance = mbs.AddObject(DistanceConstraint(markerNumbers = [mGround, mMass], distance
= 1))

mbs.AddLoad(Force(markerNumber = mMass, loadVector = [0, -50*9.81, 0]))

#assemble and solve system for default parameters
mbs.Assemble()

sims=exu.SimulationSettings()
sims.timeIntegration.generalizedAlpha.spectralRadius=0.7
exu.SolveDynamic(mbs, sims)

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

```

For examples on ObjectConnectorDistance see Examples and TestModels:

- [pendulum2Dconstraint.py](#) (Examples/)
- [fourBarMechanismTest.py](#) (TestModels/)
- [coordinateVectorConstraint.py](#) (TestModels/)
- [coordinateVectorConstraintGenericODE2.py](#) (TestModels/)
- [modelUnitTests.py](#) (TestModels/)
- [PARTS\\_ATEs\\_moving.py](#) (TestModels/)

### 7.7.2 ObjectConnectorCoordinate

A coordinate constraint which constrains two (scalar) coordinates of Marker[Node|Body]Coordinates attached to nodes or bodies. The constraint acts directly on coordinates, but does not include reference values, e.g., of nodal values. This constraint is computationally efficient and should be used to constrain nodal coordinates.

**Additional information for ObjectConnectorCoordinate:**

- The Object has the following types = Connector, Constraint
- Requested marker type = Coordinate
- **Short name for Python** = **CoordinateConstraint**
- **Short name for Python (visualization object)** = **VCoordinateConstraint**

The item **ObjectConnectorCoordinate** with type = 'ConnectorCoordinate' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex		[ MAXINT, MAX-INT ]	list of markers used in connector
offset	Real		0.	An offset between the two values
factorValue1	Real		1.	An additional factor multiplied with value1 used in algebraic equation
velocityLevel	Bool		False	If true: connector constrains velocities (only works for <a href="#">ODE2</a> coordinates!); offset is used between velocities; in this case, the offsetUserFunction_t is considered and offsetUserFunction is ignored
offsetUserFunction	PyFunctionMbsScalarIndexScalar		0	A Python function which defines the time-dependent offset; see description below
offsetUserFunction_t	PyFunctionMbsScalarIndexScalar		0	time derivative of offsetUserFunction; needed for velocity level constraints; see description below
activeConnector	Bool		True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectConnectorCoordinate			parameters for visualization of item

The item **VObjectConnectorCoordinate** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
drawSize	float		-1.	drawing size = link size; size == -1.f means that default connector size is used
color	Float4		[-1.,-1.,-1.,-1.]	RGBA connector color; if R==1, use default color

### 7.7.2.1 DESCRIPTION of ObjectConnectorCoordinate:

**Information on input parameters:**

input parameter	symbol	description see tables above
markerNumbers	$[m0, m1]^T$	
offset	$l_{\text{off}}$	
factorValue1	$k_{m1}$	
offsetUserFunction	$\text{UF} \in \mathbb{R}$	
offsetUserFunction_t	$\text{UF}_t \in \mathbb{R}$	

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

output variable	symbol	description
Displacement	$\Delta q$	relative scalar displacement of marker coordinates, not including factorValue1
Velocity	$\Delta v$	difference of scalar marker velocity coordinates, not including factorValue1
ConstraintEquation	$\mathbf{c}$	(residuum of) constraint equation
Force	$\lambda_0$	scalar constraint force (Lagrange multiplier)

### 7.7.2.2 Definition of quantities

intermediate variables	symbol	description
marker m0 coordinate	$q_{m0}$	current displacement coordinate which is provided by marker m0; does NOT include reference coordinate!
marker m1 coordinate	$q_{m1}$	
marker m0 velocity coordinate	$v_{m0}$	current velocity coordinate which is provided by marker m0
marker m1 velocity coordinate	$v_{m1}$	
difference of coordinates	$\Delta q = q_{m1} - q_{m0}$	Displacement between marker m0 to marker m1 coordinates (does NOT include reference coordinates)
difference of velocity coordinates	$\Delta v = v_{m1} - v_{m0}$	

### 7.7.2.3 Connector constraint equations

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

$$\mathbf{c}(q_{m0}, q_{m1}) = k_{m1} \cdot q_{m1} - q_{m0} - l_{\text{off}} = 0 \quad (7.331)$$

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

$$\mathbf{c}(q_{m0}, q_{m1}) = k_{m1} \cdot q_{m1} - q_{m0} - \text{UF}(mbs, t, i_N, l_{\text{off}}) = 0 \quad (7.332)$$

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

$$\dot{\mathbf{c}}(\dot{q}_{m0}, \dot{q}_{m1}) = k_{m1} \cdot \dot{q}_{m1} - \dot{q}_{m0} - d = 0 \quad (7.333)$$

The factor  $d$  in velocity level equations is zero, except if parameters.velocityLevel = True, then  $d = l_{\text{off}}$ . If velocity level constraints are active and the velocity level offsetUserFunction\_t  $\text{UF}_t$  is defined,  $\mathbf{c}$  instead becomes ( $t$  is current time)

$$\mathbf{c}(\dot{q}_{m0}, \dot{q}_{m1}) = k_{m1} \cdot \dot{q}_{m1} - \dot{q}_{m0} - \text{UF}_t(mbs, t, i_N, l_{\text{off}}) = 0 \quad (7.334)$$

and  $i_N$  represents the itemNumber (=objectNumber). Note that the index 2 equations are used, if the solver uses index 2 formulation OR if the flag parameters.velocityLevel = True (or both). The user functions include dependency on time  $t$ , but this time dependency is not respected in the computation of initial accelerations. Therefore, it is recommended that UF and  $\text{UF}_t$  does not include initial accelerations.

If `activeConnector = False`, the (index 1) algebraic equation reads for ALL cases:

$$\mathbf{c}(\lambda_0) = \lambda_0 = 0 \quad (7.335)$$


---

### Userfunction: `offsetUserFunction(mbs, t, itemNumber, l0ffset)`

A user function, which computes scalar offset for the coordinate constraint, e.g., in order to move a node on a prescribed trajectory. It is NECESSARY to use sufficiently smooth functions, having **initial offsets** consistent with **initial configuration** of bodies, either zero or compatible initial offset-velocity, and no initial accelerations. The `offsetUserFunction` is **ONLY used** in case of static computation or index3 (generalizedAlpha) time integration. In order to be on the safe side, provide both `offsetUserFunction` and `offsetUserFunction_t`.

Note that itemNumber represents the index of the object in mbs, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

The user function gets time and the offset parameter as an input and returns the computed offset:

arguments / return	type or size	description
<code>mbs</code>	MainSystem	provides MainSystem mbs in which underlying item is defined
<code>t</code>	Real	current time in mbs
<code>itemNumber</code>	Index	integer number $i_N$ of the object in mbs, allowing easy access to all object data via <code>mbs.GetObjectParameter(itemNumber, ...)</code>
<code>l0ffset</code>	Real	$l_{\text{off}}$
<b>return value</b>	Real	computed offset for given time

---

### Userfunction: `offsetUserFunction_t(mbs, t, itemNumber, l0ffset)`

A user function, which computes scalar offset **velocity** for the coordinate constraint. It is NECESSARY to use sufficiently smooth functions, having **initial offset velocities** consistent with **initial velocities** of bodies. The `offsetUserFunction_t` is used instead of `offsetUserFunction` in case of `velocityLevel = True`, or for index2 time integration and needed for computation of initial accelerations in second order implicit time integrators.

Note that itemNumber represents the index of the object in mbs, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

The user function gets time and the offset parameter as an input and returns the computed offset velocity:

arguments / return	type or size	description
mbs	MainSystem	provides MainSystem mbs in which underlying item is defined
t	Real	current time in mbs
itemNumber	Index	integer number of the object in mbs, allowing easy access to all object data via mbs.GetObjectParameter(itemNumber, ...)
lOffset	Real	$l_{\text{off}}$
<b>return value</b>	Real	computed offset velocity for given time

---

### User function example:

```
#see also mini example!
from math import sin, cos, pi
def UFoffset(mbs, t, itemNumber, lOffset):
    return 0.5*lOffset*(1-cos(0.5*pi*t))

def UFoffset_t(mbs, t, itemNumber, lOffset): #time derivative of UFoffset
    return 0.5*lOffset*0.5*pi*sin(0.5*pi*t)

nMass=mbs.AddNode(Point(referenceCoordinates = [2,0,0]))
massPoint = mbs.AddObject(MassPoint(physicsMass = 5, nodeNumber = nMass))

groundMarker=mbs.AddMarker(MarkerNodeCoordinate(nodeNumber= nGround, coordinate = 0))
nodeMarker =mbs.AddMarker(MarkerNodeCoordinate(nodeNumber= nMass, coordinate = 0))

#Spring-Damper between two marker coordinates
mbs.AddObject(CoordinateConstraint(markerNumbers = [groundMarker, nodeMarker],
                                    offset = 0.1,
                                    offsetUserFunction = UFoffset,
                                    offsetUserFunction_t = UFoffset_t))
```

---

#### 7.7.2.4 MINI EXAMPLE for ObjectConnectorCoordinate

```
def OffsetUF(mbs, t, itemNumber, lOffset): #gives 0.05 at t=1
    return 0.5*(1-np.cos(2*3.141592653589793*0.25*t))*lOffset

nMass=mbs.AddNode(Point(referenceCoordinates = [2,0,0]))
massPoint = mbs.AddObject(MassPoint(physicsMass = 5, nodeNumber = nMass))

groundMarker=mbs.AddMarker(MarkerNodeCoordinate(nodeNumber= nGround, coordinate = 0))
nodeMarker =mbs.AddMarker(MarkerNodeCoordinate(nodeNumber= nMass, coordinate = 0))
```

```

#Spring-Damper between two marker coordinates
mbs.AddObject(CoordinateConstraint(markerNumbers = [groundMarker, nodeMarker],
                                    offset = 0.1, offsetUserFunction = OffsetUF))

#assemble and solve system for default parameters
mbs.Assemble()
exu.SolveDynamic(mbs)

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

```

---

For examples on ObjectConnectorCoordinate see Examples and TestModels:

- [sliderCrank3DwithANCFbeltDrive2.py](#) (Examples/)
- [ALEANCF\\_pipe.py](#) (Examples/)
- [ANCFALEtest.py](#) (Examples/)
- [ANCF\\_cantilever\\_test\\_dyn.py](#) (Examples/)
- [ANCF\\_contact\\_circle.py](#) (Examples/)
- [ANCF\\_contact\\_circle2.py](#) (Examples/)
- [ANCF\\_moving\\_rigidbody.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- [ANCF\\_slidingJoint2Drigid.py](#) (Examples/)
- [ANCF\\_switchingSlidingJoint2D.py](#) (Examples/)
- [ANCF\\_tests2.py](#) (Examples/)
- [ANCF\\_test\\_halfcircle.py](#) (Examples/)
- ...
- [ACNFslidingAndALEjointTest.py](#) (TestModels/)
- [ANCFbeltDrive.py](#) (TestModels/)
- [ANCFcontactCircleTest.py](#) (TestModels/)
- ...

### 7.7.3 ObjectConnectorCoordinateVector

A constraint which constrains the coordinate vectors of two markers Marker[Node|Object|Body]Coordinates attached to nodes or bodies. The marker uses the objects [LTG](#)-lists to build the according coordinate mappings.

**Additional information for ObjectConnectorCoordinateVector:**

- The Object has the following types = **Connector**, **Constraint**
- Requested marker type = **Coordinate**
- **Short name for Python** = **CoordinateVectorConstraint**
- **Short name for Python (visualization object)** = **VCoordinateVectorConstraint**

The item **ObjectConnectorCoordinateVector** with type = 'ConnectorCoordinateVector' has the following parameters:

Name	type	size	default value	description
name	String		"	constraints's unique name
markerNumbers	ArrayMarkerIndex		[ MAXINT, MAX-INT ]	list of markers used in connector
scalingMarker0	NumpyMatrix		Matrix[]	linear scaling matrix for coordinate vector of marker 0; matrix provided in Python numpy format
scalingMarker1	NumpyMatrix		Matrix[]	linear scaling matrix for coordinate vector of marker 1; matrix provided in Python numpy format
quadraticTermMarker0	NumpyMatrix		Matrix[]	quadratic scaling matrix for coordinate vector of marker 0; matrix provided in Python numpy format
quadraticTermMarker1	NumpyMatrix		Matrix[]	quadratic scaling matrix for coordinate vector of marker 1; matrix provided in Python numpy format
offset	NumpyVector		[]	offset added to constraint equation; only active, if no userFunction is defined
velocityLevel	Bool		False	If true: connector constrains velocities (only works for <a href="#">ODE2</a> coordinates!); offset is used between velocities; in this case, the offsetUserFunction_t is considered and offsetUserFunction is ignored
constraintUserFunction	PyFunctionVectorMbsScalarIndex2VectorBool	0		A Python user function which computes the constraint equations; to define the number of algebraic equations, set scalingMarker0 as a numpy.zeros((nAE,1)) array with nAE being the number algebraic equations; see description below

jacobianUserFunction	PyFunctionMatrixContainerMbsScalarIndex2VectorBool	0	A Python user function which computes the jacobian, i.e., the derivative of the left-hand-side object equation w.r.t. the coordinates (times $f_{ODE2}$ ) and w.r.t. the velocities (times $f_{ODE2_i}$ ). Terms on the RHS must be subtracted from the LHS equation; the respective terms for the stiffness matrix and damping matrix are automatically added; see description below
activeConnector	Bool	True	flag, which determines, if the connector is active; used to deactivate (temporarily) a connector or constraint
visualization	VObjectConnectorCoordinateVector		parameters for visualization of item

The item VObjectConnectorCoordinateVector has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
color	Float4		[-1,-1,-1,-1]	RGBA connector color; if R== -1, use default color

### 7.7.3.1 DESCRIPTION of ObjectConnectorCoordinateVector:

**Information on input parameters:**

input parameter	symbol	description see tables above
markerNumbers	$[m_0, m_1]^T$	
scalingMarker0	$\mathbf{X}_{m0} \in \mathbb{R}^{n_{ae} \times n_{qm0}}$	
scalingMarker1	$\mathbf{X}_{m1} \in \mathbb{R}^{n_{ae} \times n_{qm1}}$	
quadraticTermMarker0	$\mathbf{Y}_{m0} \in \mathbb{R}^{n_{ae} \times n_{qm0}}$	
quadraticTermMarker1	$\mathbf{Y}_{m1} \in \mathbb{R}^{n_{ae} \times n_{qm1}}$	
offset	$\mathbf{v}_{off} \in \mathbb{R}^{n_{ae}}$	
constraintUserFunction	$\mathbf{c}_{user} \in \mathbb{R}^{n_{ae}}$	
jacobianUserFunction	$\mathbf{J}_{user} \in \mathbb{R}^{(n_{qm0} + n_{qm1}) \times n_{ae}}$	

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

output variable	symbol	description
Displacement	$\Delta \mathbf{q}$	relative scalar displacement of marker coordinates, not including scaling matrices
Velocity	$\Delta \mathbf{v}$	difference of scalar marker velocity coordinates, not including scaling matrices

ConstraintEquation	$\mathbf{c}$	(residuum of) constraint equations
Force	$\lambda$	constraint force vector (vector of Lagrange multipliers), resulting from action of constraint equations

### 7.7.3.2 Definition of quantities

intermediate variables	symbol	description
marker m0 coordinate vector	$\mathbf{q}_{m0} \in \mathbb{R}^{n_{q_{m0}}}$	coordinate vector provided by marker $m0$ ; depending on the marker, the coordinates may or may not include reference coordinates
marker m1 coordinate vector	$\mathbf{q}_{m1} \in \mathbb{R}^{n_{q_{m1}}}$	coordinate vector provided by marker $m1$ ; depending on the marker, the coordinates may or may not include reference coordinates
marker m0 velocity coordinate vector	$\dot{\mathbf{q}}_{m0} \in \mathbb{R}^{n_{q_{m0}}}$	velocity coordinate vector provided by marker $m0$
marker m1 velocity coordinate vector	$\dot{\mathbf{q}}_{m1} \in \mathbb{R}^{n_{q_{m1}}}$	velocity coordinate vector provided by marker $m1$
number of algebraic equations	$n_{ae}$	number of algebraic equations must be same as number of rows in $\mathbf{X}_{m0}$ and $\mathbf{X}_{m1}$
difference of coordinates	$\Delta\mathbf{q} = \mathbf{q}_{m1} - \mathbf{q}_{m0}$	Displacement between marker $m0$ to marker $m1$ coordinates
difference of velocity coordinates	$\Delta\mathbf{v} = \dot{\mathbf{q}}_{m1} - \dot{\mathbf{q}}_{m0}$	

### 7.7.3.3 Remarks

The number of algebraic equations depends on the maximum number of rows in  $\mathbf{X}_{m0}$ ,  $\mathbf{Y}_{m0}$ ,  $\mathbf{X}_{m1}$  and  $\mathbf{Y}_{m1}$ . The number of rows of the latter matrices must either be zero or the maximum of these rows.

The number of columns in  $\mathbf{X}_{m0}$  (or  $\mathbf{Y}_{m0}$ ) must agree with the length of the coordinate vector  $\mathbf{q}_{m0}$  and the number of columns in  $\mathbf{X}_{m1}$  (or  $\mathbf{Y}_{m1}$ ) must agree with the length of the coordinate vector  $\mathbf{q}_{m1}$ , if these matrices are not empty matrices. If one marker  $k$  is a ground marker (node/object), the length of  $\mathbf{q}_{m,k}$  is zero and also the according matrices  $\mathbf{X}_{m,k}$ ,  $\mathbf{Y}_{m,k}$  have zero size and will not be considered in the computation of the constraint equations.

### 7.7.3.4 Connector constraint equations

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

$$\mathbf{c}(\mathbf{q}_{m0}, \mathbf{q}_{m1}) = \mathbf{X}_{m1} \cdot \mathbf{q}_{m1} + \mathbf{Y}_{m1} \cdot \mathbf{q}_{m1}^2 - \mathbf{X}_{m0} \cdot \mathbf{q}_{m0} - \mathbf{Y}_{m0} \cdot \mathbf{q}_{m0}^2 - \mathbf{v}_{off} = 0 \quad (7.336)$$

Note that the squared coordinates are understood as  $\mathbf{q}_{m0}^2 = [q_{0,m0}^2, q_{1,m0}^2, \dots]^T$ , same for  $\mathbf{q}_{m1}^2$ .

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

$$\mathbf{c}(\mathbf{q}_{m0}, \mathbf{q}_{m1}) = \mathbf{X}_{m1} \cdot \mathbf{q}_{m1} + \mathbf{Y}_{m1} \cdot \mathbf{q}_{m1}^2 - \mathbf{X}_{m0} \cdot \mathbf{q}_{m0} - \mathbf{Y}_{m0} \cdot \mathbf{q}_{m0}^2 - UF(mbs, t, \mathbf{v}_{off}) = 0 \quad (7.337)$$

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

$$\dot{\mathbf{c}}(\dot{\mathbf{q}}_{m0}, \dot{\mathbf{q}}_{m1}) = \mathbf{X}_{m1} \cdot \dot{\mathbf{q}}_{m1} + \mathbf{Y}_{m1} \cdot \dot{\mathbf{q}}_{m1}^2 - \mathbf{X}_{m0} \cdot \dot{\mathbf{q}}_{m0} - \mathbf{Y}_{m0} \cdot \dot{\mathbf{q}}_{m0}^2 - \mathbf{d}_{off} = 0 \quad (7.338)$$

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

If velocity level constraints are active and the velocity level `offsetUserFunction_t` UF<sub>*t*</sub> is defined,  $\dot{\mathbf{c}}$  instead becomes (*t* is current time)

$$\dot{\mathbf{c}}(\dot{\mathbf{q}}_{m0}, \dot{\mathbf{q}}_{m1}) = \mathbf{X}_{m1} \cdot \dot{\mathbf{q}}_{m1} + \mathbf{Y}_{m1} \cdot \dot{\mathbf{q}}_{m1}^2 - \mathbf{X}_{m0} \cdot \dot{\mathbf{q}}_{m0} - \mathbf{Y}_{m0} \cdot \dot{\mathbf{q}}_{m0}^2 - \text{UF}_t(mbs, t, \mathbf{v}_{\text{off}}) = 0 \quad (7.339)$$

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<sub>*t*</sub> does not include initial accelerations.

If `activeConnector = False`, the (index 1) algebraic equation reads for ALL cases:

$$\mathbf{c}(\lambda) = \lambda = 0 \quad (7.340)$$


---

### Userfunction: `constraintUserFunction(mbs, t, itemNumber, q, q_t, velocityLevel)`

A user function, which computes algebraic equations for the connector based on the marker coordinates stored in `q` and `q_t`. Depending on `velocityLevel`, the user function needs to compute either the position-level (`velocityLevel=False`) or the velocity level (`velocityLevel=True`) constraint equations. Note that for Index 2 solvers, the `constraintUserFunction` may be called with `velocityLevel=True` but `jacobianUserFunction` is called with `velocityLevel=False`. To define the number of algebraic equations, set `scalingMarker0` as a `numpy.zeros((nAE, 1))` array with `nAE` being the number algebraic equations. The returned vector of `constraintUserFunction` must have size `nAE`.

Note that `itemNumber` represents the index of the `ObjectGenericODE2` object in `mbs`, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

arguments / return	type or size	description
<code>mbs</code>	MainSystem	provides MainSystem <code>mbs</code> to which object belongs to
<code>t</code>	Real	current time in <code>mbs</code>
<code>itemNumber</code>	Index	integer number $i_N$ of the object in <code>mbs</code> , allowing easy access to all object data via <code>mbs.GetObjectParameter(itemNumber, ...)</code>
<code>q</code>	Vector $\in \mathbb{R}^{(n_{q_{m0}} + n_{q_{m1}})}$	connector coordinates, subsequently for marker $m0$ and marker $m1$ , in current configuration
<code>q_t</code>	Vector $\in \mathbb{R}^{(n_{q_{m0}} + n_{q_{m1}})}$	connector velocity coordinates in current configuration
<code>velocityLevel</code>	Bool	velocityLevel as currently stored in connector
<code>return value</code>	Vector $\in \mathbb{R}^{n_{ae}}$	returns vector (numpy array or list) of evaluated constraint equations for connector

---

### Userfunction: `jacobianUserFunction(mbs, t, itemNumber, q, q_t, velocityLevel)`

A user function, which computes the jacobian of the algebraic equations w.r.t. the ODE2 coordinates

(ODE2\_t velocity coordinates if `velocityLevel=True`). The jacobian needs to exactly represent the derivative of the constraintUserFunction. The returned matrix of `jacobianUserFunction` must have `nAE` rows and `len(q)` columns.

arguments / return	type or size	description
<code>mbs</code>	MainSystem	provides MainSystem mbs to which object belongs to
<code>t</code>	Real	current time in mbs
<code>itemNumber</code>	Index	integer number $i_N$ of the object in mbs, allowing easy access to all object data via <code>mbs.GetObjectParameter(itemNumber, ...)</code>
<code>q</code>	$\text{Vector} \in \mathbb{R}^{(n_{qm0} + n_{qm1})}$	connector coordinates, subsequently for marker $m0$ and marker $m1$ , in current configuration
<code>q_t</code>	$\text{Vector} \in \mathbb{R}^{(n_{qm0} + n_{qm1})}$	connector velocity coordinates in current configuration
<code>velocityLevel</code>	Bool	velocityLevel as currently stored in connector
<code>return value</code>	<code>MatrixContainer</code> $\in \mathbb{R}^{(n_{qm0} + n_{qm1}) \times n_{ae}}$	returns special jacobian for connector, as <code>exu.MatrixContainer</code> , numpy array or list of lists; use <code>MatrixContainer</code> sparse format for larger matrices to speed up computations; sparse triplets MAY NOT contain zero values!

---

For examples on ObjectConnectorCoordinateVector see Examples and TestModels:

- [`coordinateVectorConstraint.py`](#) (TestModels/)
- [`coordinateVectorConstraintGenericODE2.py`](#) (TestModels/)
- [`rigidBodyAsUserFunctionTest.py`](#) (TestModels/)

## 7.8 Objects (Object)

A Object provides equations, using coordinates from Nodes. General objects lead to system equations, that do not represent physical Bodies or Connectors.

### 7.8.1 ObjectGenericODE1

A system of  $n$  first order ordinary differential equations ([ODE1](#)), having a system matrix, a rhs vector, but mostly it will use a user function to describe special [ODE1](#) systems. It is based on NodeGenericODE1 nodes. NOTE that all matrices, vectors, etc. must have the same dimensions  $n$  or  $(n \times n)$ , or they must be empty ( $0 \times 0$ ), using [] in Python.

**Additional information for ObjectGenericODE1:**

- The Object has the following types = `MultiNoded`
- Requested node type: read detailed information of item

The item **ObjectGenericODE1** with type = 'GenericODE1' has the following parameters:

Name	type	size	default value	description
name	String		"	objects's unique name
nodeNumbers	ArrayNodeIndex		[]	node numbers which provide the coordinates for the object (consecutively as provided in this list)
systemMatrix	NumpyMatrix		Matrix[]	system matrix (state space matrix) of first order ODE
rhsVector	NumpyVector		[]	a constant rhs vector (e.g., for constant input)
rhsUserFunction	PyFunctionVectorMbsScalarIndexVector		0	A Python user function which computes the right-hand-side (rhs) of the first order ODE; see description below
coordinateIndexPerNode	ArrayIndex		[]	this list contains the local coordinate index for every node, which is needed, e.g., for markers; the list is generated automatically every time parameters have been changed
tempCoordinates	NumpyVector		[]	temporary vector containing coordinates
tempCoordinates_t	NumpyVector		[]	temporary vector containing velocity coordinates
visualization	VObjectGenericODE1			parameters for visualization of item

The item **VObjectGenericODE1** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

### 7.8.1.1 DESCRIPTION of ObjectGenericODE1:

#### Information on input parameters:

input parameter	symbol	description see tables above
nodeNumbers	$\mathbf{n}_n = [n_0, \dots, n_n]^T$	
systemMatrix	$\mathbf{A} \in \mathbb{R}^{n \times n}$	
rhsVector	$\mathbf{f} \in \mathbb{R}^n$	
rhsUserFunction	$\mathbf{f}_{user} \in \mathbb{R}^n$	
tempCoordinates	$\mathbf{c}_{temp} \in \mathbb{R}^n$	
tempCoordinates_t	$\dot{\mathbf{c}}_{temp} \in \mathbb{R}^n$	

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

output variable	symbol	description
Coordinates		all <code>ODE1</code> coordinates
Coordinates_t		all <code>ODE1</code> velocity coordinates

### 7.8.1.2 Equations of motion

An object with node numbers  $[n_0, \dots, n_n]$  and according numbers of nodal coordinates  $[n_{c_0}, \dots, n_{c_n}]$ , the total number of equations (=coordinates) of the object is

$$n = \sum_i n_{c_i}, \quad (7.341)$$

which is used throughout the description of this object.

### 7.8.1.3 Equations of motion

$$\dot{\mathbf{q}} = \mathbf{f} + \mathbf{f}_{user}(mbs, t, i_N, \mathbf{q}) \quad (7.342)$$

Note that the user function  $\mathbf{f}_{user}(mbs, t, i_N, \mathbf{q})$  may be empty ( $=0$ ), and that `iN` represents the itemNumber (=objectNumber).

CoordinateLoads are added for the respective `ODE1` coordinate on the RHS of the latter equation.

#### Userfunction: `rhsUserFunction(mbs, t, itemNumber, q)`

A user function, which computes a RHS vector depending on current time and states of the object. Can be used to create any kind of first order system, especially state space equations (inputs are added via CoordinateLoads to every node). Note that itemNumber represents the index of the `ObjectGenericODE1` object in `mbs`, which can be used to retrieve additional data from the object through `mbs.GetObjectParameter(itemNumber, ...)`, see the according description of `GetObjectParameter`.

arguments / return	type or size	description
<code>mbs</code>	MainSystem	provides MainSystem <code>mbs</code> to which object belongs
<code>t</code>	Real	current time in <code>mbs</code>
<code>itemNumber</code>	Index	integer number $i_N$ of the object in <code>mbs</code> , allowing easy access to all object data via <code>mbs.GetObjectParameter(itemNumber, ...)</code>

<code>q</code>	Vector $\in \mathbb{R}^n$	object coordinates (composed from <code>ODE1</code> nodal coordinates) in current configuration, without reference values
<code>return value</code>	Vector $\in \mathbb{R}^n$	returns force vector for object

---

### User function example:

```
A = numpy.diag([200,100])
#simple linear user function returning A*q
def UFrhs(mbs, t, itemNumber, q):
    return np.dot(A, q) + np.array([0,2])

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

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

---

#### 7.8.1.4 MINI EXAMPLE for ObjectGenericODE1

```
#set up a 2-DOF system
nODE1 = mbs.AddNode(NodeGenericODE1(referenceCoordinates=[0,0],
                                      initialCoordinates=[1,0],
                                      numberOfODE1Coordinates=2))

#build system matrix and force vector
#undamped mechanical system with m=1, K=100, f=1
A = np.array([[0,1],
              [-100,0]])
b = np.array([0,1])

oGenericODE1 = mbs.AddObject(ObjectGenericODE1(nodeNumbers=[nODE1],
                                                systemMatrix=A,
                                                rhsVector=b))

#assemble and solve system for default parameters
mbs.Assemble()

sims=exu.SimulationSettings()
solverType = exu.DynamicSolverType.RK44
exu.SolveDynamic(mbs, solverType=solverType, simulationSettings=sims)
```

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

---

For examples on ObjectGenericODE1 see Examples and TestModels:

- [lugreFrictionODE1.py](#) (Examples/)
- [lugreFrictionTest.py](#) (Examples/)
- [solverExplicitODE1ODE2test.py](#) (TestModels/)

## 7.9 Markers

A Marker provides an interface between a large variety of Nodes / Objects and Connectors or Loads.

### 7.9.1 MarkerBodyMass

A marker attached to the body mass; use this marker to apply a body-load (e.g. gravitational force).

**Additional information for MarkerBodyMass:**

- The Marker has the following types = Object, Body, BodyMass

The item **MarkerBodyMass** with type = 'BodyMass' has the following parameters:

Name	type	size	default value	description
name	String		"	marker's unique name
bodyNumber	ObjectIndex		MAXINT	body number to which marker is attached to
visualization	VMarkerBodyMass			parameters for visualization of item

The item VMarkerBodyMass has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

#### 7.9.1.1 DESCRIPTION of MarkerBodyMass:

---

For examples on MarkerBodyMass see Examples and TestModels:

- [ALEANCF\\_pipe.py](#) (Examples/)
- [ANCF\\_moving\\_rigidbody.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- [ANCF\\_slidingJoint2Drigid.py](#) (Examples/)
- [ANCF\\_switchingSlidingJoint2D.py](#) (Examples/)
- [CMSExampleCourse.py](#) (Examples/)
- [finiteSegmentMethod.py](#) (Examples/)
- [NGsolveCMSTutorial.py](#) (Examples/)
- [NGsolveCraigBampton.py](#) (Examples/)
- [NGsolvePostProcessingStresses.py](#) (Examples/)
- [ObjectFFRFconvergenceTestBeam.py](#) (Examples/)
- [ObjectFFRFconvergenceTestHinge.py](#) (Examples/)
- ...

- [fourBarMechanismRedundant.py](#) (TestModels/)
- [genericJointUserFunctionTest.py](#) (TestModels/)
- [modelUnitTests.py](#) (TestModels/)
- ...

## 7.9.2 MarkerBodyPosition

A position body-marker attached to a local (body-fixed) position  ${}^b\mathbf{b} = [b_0, b_1, b_2]$  ( $x, y$ , and  $z$  coordinates) of the body.

**Additional information for MarkerBodyPosition:**

- The Marker has the following types = Object, Body, Position

The item **MarkerBodyPosition** with type = 'BodyPosition' has the following parameters:

Name	type	size	default value	description
name	String		"	marker's unique name
bodyNumber	ObjectIndex		MAXINT	body number to which marker is attached to
localPosition	Vector3D	3	[0.,0.,0.]	local body position of marker; e.g. local (body-fixed) position where force is applied to
visualization	VMarkerBodyPosition			parameters for visualization of item

The item **VMarkerBodyPosition** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

### 7.9.2.1 DESCRIPTION of MarkerBodyPosition:

**Information on input parameters:**

input parameter	symbol	description see tables above
localPosition	${}^b\mathbf{b}$	

For examples on MarkerBodyPosition see Examples and TestModels:

- [ANCF\\_contact\\_circle.py](#) (Examples/)
- [ANCF\\_contact\\_circle2.py](#) (Examples/)
- [ANCF\\_moving\\_rigidbody.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- [ANCF\\_slidingJoint2Drigid.py](#) (Examples/)
- [ANCF\\_switchingSlidingJoint2D.py](#) (Examples/)
- [cartesianSpringDamper.py](#) (Examples/)
- [coordinateSpringDamper.py](#) (Examples/)
- [finiteSegmentMethod.py](#) (Examples/)

- [flexibleRotor3Dtest.py](#) (Examples/)
- [geneticOptimizationSliderCrank.py](#) (Examples/)
- [lavalRotor2Dtest.py](#) (Examples/)
- ...
- [ACNFslidingAndALEjointTest.py](#) (TestModels/)
- [ANCFcontactCircleTest.py](#) (TestModels/)
- [ANCFcontactFrictionTest.py](#) (TestModels/)
- ...

### 7.9.3 MarkerBodyRigid

A rigid-body (position+orientation) body-marker attached to a local (body-fixed) position  ${}^b\mathbf{b} = [b_0, b_1, b_2]$  ( $x$ ,  $y$ , and  $z$  coordinates) of the body.

**Additional information for MarkerBodyRigid:**

- The Marker has the following types = Object, Body, Position, Orientation

The item **MarkerBodyRigid** with type = 'BodyRigid' has the following parameters:

Name	type	size	default value	description
name	String		"	marker's unique name
bodyNumber	ObjectIndex		MAXINT	body number to which marker is attached to
localPosition	Vector3D	3	[0.,0.,0.]	local body position of marker; e.g. local (body-fixed) position where force is applied to
visualization	VMarkerBodyRigid			parameters for visualization of item

The item VMarkerBodyRigid has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

#### 7.9.3.1 DESCRIPTION of MarkerBodyRigid:

**Information on input parameters:**

input parameter	symbol	description see tables above
localPosition	${}^b\mathbf{b}$	

For examples on MarkerBodyRigid see Examples and TestModels:

- [addPrismaticJoint.py](#) (Examples/)
- [addRevoluteJoint.py](#) (Examples/)
- [ANCF\\_contact\\_circle.py](#) (Examples/)
- [ANCF\\_contact\\_circle2.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- [ANCF\\_tests2.py](#) (Examples/)
- [ANCF\\_test\\_halfcircle.py](#) (Examples/)
- [beltDriveReevingSystem.py](#) (Examples/)
- [bicycleIftommBenchmark.py](#) (Examples/)

- [CMSexampleCourse.py](#) (Examples/)
- [flexibleRotor3Dtest.py](#) (Examples/)
- [fourBarMechanism3D.py](#) (Examples/)
- ...
- [ACNFslidingAndALEjointTest.py](#) (TestModels/)
- [ANCFbeltDrive.py](#) (TestModels/)
- [ANCFcontactCircleTest.py](#) (TestModels/)
- ...

#### 7.9.4 MarkerNodePosition

A node-Marker attached to a position-based node.

##### Additional information for MarkerNodePosition:

- The Marker has the following types = Node, Position

The item **MarkerNodePosition** with type = 'NodePosition' has the following parameters:

Name	type	size	default value	description
name	String		''	marker's unique name
nodeNumber	NodeIndex		MAXINT	node number to which marker is attached to
visualization	VMarkerNodePosition			parameters for visualization of item

The item **VMarkerNodePosition** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

##### 7.9.4.1 DESCRIPTION of MarkerNodePosition:

---

For examples on MarkerNodePosition see Examples and TestModels:

- [ALEANCF\\_pipe.py](#) (Examples/)
- [ANCF\\_cantilever\\_test.py](#) (Examples/)
- [ANCF\\_cantilever\\_test\\_dyn.py](#) (Examples/)
- [ANCF\\_contact\\_circle.py](#) (Examples/)
- [ANCF\\_contact\\_circle2.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- [ANCF\\_slidingJoint2Drigid.py](#) (Examples/)
- [ANCF\\_tests2.py](#) (Examples/)
- [ANCF\\_test\\_halfcircle.py](#) (Examples/)
- [flexibleRotor3Dtest.py](#) (Examples/)
- [geneticOptimizationSliderCrank.py](#) (Examples/)
- [interactiveTutorial.py](#) (Examples/)
- ...
- [ANCFcontactCircleTest.py](#) (TestModels/)
- [ANCFcontactFrictionTest.py](#) (TestModels/)
- [connectorGravityTest.py](#) (TestModels/)
- ...

### 7.9.5 MarkerNodeRigid

A rigid-body (position+orientation) node-marker attached to a rigid-body node.

#### Additional information for MarkerNodeRigid:

- The Marker has the following types = Node, Position, Orientation

The item **MarkerNodeRigid** with type = 'NodeRigid' has the following parameters:

Name	type	size	default value	description
name	String		''	marker's unique name
nodeNumber	NodeIndex		MAXINT	node number to which marker is attached to
visualization	VMarkerNodeRigid			parameters for visualization of item

The item **VMarkerNodeRigid** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

#### 7.9.5.1 DESCRIPTION of MarkerNodeRigid:

---

For examples on MarkerNodeRigid see Examples and TestModels:

- [ANCF\\_contact\\_circle.py](#) (Examples/)
- [ANCF\\_contact\\_circle2.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- [ANCF\\_tests2.py](#) (Examples/)
- [ANCF\\_test\\_halfcircle.py](#) (Examples/)
- [beltDriveReevingSystem.py](#) (Examples/)
- [CMSEexampleCourse.py](#) (Examples/)
- [NGsolveCMSTutorial.py](#) (Examples/)
- [NGsolveCraigBampton.py](#) (Examples/)
- [NGsolvePostProcessingStresses.py](#) (Examples/)
- [ObjectFFRFconvergenceTestHinge.py](#) (Examples/)
- [objectFFRFreducedOrderNetgen.py](#) (Examples/)
- ...
- [ANCFbeltDrive.py](#) (TestModels/)
- [ANCFcontactCircleTest.py](#) (TestModels/)
- [ANCFgeneralContactCircle.py](#) (TestModels/)
- ...

## 7.9.6 MarkerNodeCoordinate

A node-Marker attached to a [ODE2](#) coordinate of a node; this marker allows to connect a coordinate-based constraint or connector to a nodal coordinate (also NodeGround); for [ODE1](#) coordinates use MarkerNodeODE1Coordinate.

**Additional information for MarkerNodeCoordinate:**

- The Marker has the following types = Node, Coordinate

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

Name	type	size	default value	description
name	String		''	marker's unique name
nodeNumber	NodeIndex		MAXINT	node number to which marker is attached to
coordinate	UInt		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

---

### 7.9.6.1 DESCRIPTION of MarkerNodeCoordinate:

---

For examples on MarkerNodeCoordinate see Examples and TestModels:

- [ALEANCF\\_pipe.py](#) (Examples/)
- [ANCFALTest.py](#) (Examples/)
- [ANCF\\_cantilever\\_test.py](#) (Examples/)
- [ANCF\\_cantilever\\_test\\_dyn.py](#) (Examples/)
- [ANCF\\_contact\\_circle.py](#) (Examples/)
- [ANCF\\_contact\\_circle2.py](#) (Examples/)
- [ANCF\\_moving\\_rigidbody.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- [ANCF\\_slidingJoint2Drigid.py](#) (Examples/)
- [ANCF\\_switchingSlidingJoint2D.py](#) (Examples/)
- [ANCF\\_tests2.py](#) (Examples/)
- [ANCF\\_test\\_halfcircle.py](#) (Examples/)

- ...
- [ACNFslidingAndALEjointTest.py](#) (TestModels/)
- [ANCFbeltDrive.py](#) (TestModels/)
- [ANCFcontactCircleTest.py](#) (TestModels/)
- ...

### 7.9.7 MarkerNodeCoordinates

A node-Marker attached to all [ODE2](#) coordinates of a node; IN CONTRAST to MarkerNodeCoordinate, the marker coordinates INCLUDE the reference values! for [ODE1](#) coordinates use MarkerNodeODE1Coordinates (under development).

**Additional information for MarkerNodeCoordinates:**

- The Marker has the following types = Node, Coordinate

The item **MarkerNodeCoordinates** with type = 'NodeCoordinates' has the following parameters:

Name	type	size	default value	description
name	String		''	marker's unique name
nodeNumber	NodeIndex		MAXINT	node number to which marker is attached to
visualization	VMarkerNodeCoordinates			parameters for visualization of item

The item VMarkerNodeCoordinates has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

#### 7.9.7.1 DESCRIPTION of MarkerNodeCoordinates:

---

For examples on MarkerNodeCoordinates see Examples and TestModels:

- [coordinateVectorConstraint.py](#) (TestModels/)
- [coordinateVectorConstraintGenericODE2.py](#) (TestModels/)
- [rigidBodyAsUserFunctionTest.py](#) (TestModels/)

### 7.9.8 MarkerNodeODE1Coordinate

A node-Marker attached to a [ODE1](#) coordinate of a node.

#### Additional information for MarkerNodeODE1Coordinate:

- The Marker has the following types = Node, Coordinate

The item **MarkerNodeODE1Coordinate** with type = 'NodeODE1Coordinate' has the following parameters:

Name	type	size	default value	description
name	String		"	marker's unique name
nodeNumber	NodeIndex		MAXINT	node number to which marker is attached to
coordinate	UInt		MAXINT	coordinate of node to which marker is attached to
visualization	VMarkerNodeODE1Coordinate			parameters for visualization of item

The item **VMarkerNodeODE1Coordinate** has the following parameters:

Name	type	size	default value	description
show	Bool		False	currently not available; set true, if item is shown in visualization and false if it is not shown

### 7.9.9 MarkerNodeRotationCoordinate

A node-Marker attached to a node containing rotation; the Marker measures a rotation coordinate (Tait-Bryan angles) or angular velocities on the velocity level.

**Additional information for MarkerNodeRotationCoordinate:**

- The Marker has the following types = Node, Orientation, Coordinate

The item **MarkerNodeRotationCoordinate** with type = 'NodeRotationCoordinate' has the following parameters:

Name	type	size	default value	description
name	String		''	marker's unique name
nodeNumber	NodeIndex		MAXINT	node number to which marker is attached to
rotationCoordinate	UInt		MAXINT	rotation coordinate: 0=x, 1=y, 2=z
visualization	VMarkerNodeRotationCoordinate			parameters for visualization of item

The item VMarkerNodeRotationCoordinate has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

#### 7.9.9.1 DESCRIPTION of MarkerNodeRotationCoordinate:

---

For examples on MarkerNodeRotationCoordinate see Examples and TestModels:

- [rigidRotor3DbasicBehaviour.py](#) (Examples/)
- [sliderCrank3DwithANCFbeltDrive2.py](#) (Examples/)
- [driveTrainTest.py](#) (TestModels/)

### 7.9.10 MarkerSuperElementPosition

A position marker attached to a SuperElement, such as ObjectFFRF, ObjectGenericODE2 and ObjectFFRReducedOrder (for which it is in its current implementation inefficient for large number of meshNodeNumbers). The marker acts on the mesh (interface) nodes, not on the underlying nodes of the object.

#### Additional information for MarkerSuperElementPosition:

- The Marker has the following types = Object, Body, Position

The item **MarkerSuperElementPosition** with type = 'SuperElementPosition' has the following parameters:

Name	type	size	default value	description
name	String		''	marker's unique name
bodyNumber	ObjectIndex		MAXINT	body number to which marker is attached to
meshNodeNumbers	ArrayIndex		[]	a list of $n_m$ mesh node numbers of superelement (=interface nodes) which are used to compute the body-fixed marker position; the related nodes must provide 3D position information, such as NodePoint, NodePoint2D, NodeRigidBody[.]; in order to retrieve the global node number, the generic body needs to convert local into global node numbers
weightingFactors	Vector		[]	a list of $n_m$ weighting factors per node to compute the final local position; the sum of these weights shall be 1, such that a summation of all nodal positions times weights gives the average position of the marker
visualization	VMarkerSuperElementPosition			parameters for visualization of item

The item VMarkerSuperElementPosition has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
showMarkerNodes	Bool		True	set true, if all nodes are shown (similar to marker, but with less intensity)

---

#### 7.9.10.1 DESCRIPTION of MarkerSuperElementPosition:

##### Information on input parameters:

input parameter	symbol	description see tables above
bodyNumber	$n_b$	

meshNodeNumbers	$[k_0, \dots, k_{n_m-1}]^T$	
weightingFactors	$[w_0, \dots, w_{n_m-1}]^T$	

#### Definition of marker quantities:

intermediate variables	symbol	description
number of mesh nodes	$n_m$	size of meshNodeNumbers and weightingFactors which are marked; this must not be the number of mesh nodes in the marked object
mesh node number	$i = k_i$	abbreviation
mesh node points	${}^0\mathbf{p}_i$	position of mesh node $k_i$ in object $n_b$
mesh node velocities	${}^0\mathbf{v}_i$	velocity of mesh node $i$ in object $n_b$
marker position	${}^0\mathbf{p}_m = \sum_i w_i \cdot {}^0\mathbf{p}_i$	current global position which is provided by marker
marker velocity	${}^0\mathbf{v}_m = \sum_i w_i \cdot {}^0\mathbf{v}_i$	current global velocity which is provided by marker

#### 7.9.10.2 Marker quantities

The marker provides a 'position' jacobian, which is the derivative of the marker velocity w.r.t. the object velocity coordinates  $\dot{\mathbf{q}}_{n_b}$ ,

$$\mathbf{J}_{m, pos} = \frac{\partial {}^0\mathbf{v}_m}{\partial \dot{\mathbf{q}}_{n_b}} = \sum_i w_i \cdot \mathbf{J}_{i, pos} \quad (7.343)$$

in which  $\mathbf{J}_{i, pos}$  denotes the position jacobian of mesh node  $i$ ,

$$\mathbf{J}_{i, pos} = \frac{\partial {}^0\mathbf{v}_i}{\partial \dot{\mathbf{q}}_{n_b}} \quad (7.344)$$

The jacobian  $\mathbf{J}_{i, pos}$  usually contains mostly zeros for ObjectGenericODE2, because the jacobian only affects one single node. In ObjectFFRFreducedOrder, the jacobian may affect all reduced coordinates.

Note that  $\mathbf{J}_{m, pos}$  is actually computed by the ObjectSuperElement within the function GetAccessFunctionSuperElement.

#### 7.9.10.3 MINI EXAMPLE for MarkerSuperElementPosition

```
#set up a mechanical system with two nodes; it has the structure: |~~M0~~M1
#==>further examples see objectGenericODE2Test.py, objectFFRFTest2.py, etc.
nMass0 = mbs.AddNode(NodePoint(referenceCoordinates=[0,0,0]))
nMass1 = mbs.AddNode(NodePoint(referenceCoordinates=[1,0,0]))
mGround = mbs.AddMarker(MarkerBodyPosition(bodyNumber=oGround, localPosition = [1,0,0]))

mass = 0.5 * np.eye(3)      #mass of nodes
stif = 5000 * np.eye(3)    #stiffness of nodes
damp = 50 * np.eye(3)      #damping of nodes
Z = 0. * np.eye(3)         #matrix with zeros
#build mass, stiffness and damping matrices (:
```

```

M = np.block([[mass,          0.*np.eye(3)],
              [0.*np.eye(3), mass       ] ])
K = np.block([[2*stif, -stif],
              [-stif,  stiff] ])
D = np.block([[2*damp, -damp],
              [-damp,  damp] ])

oGenericODE2 = mbs.AddObject(ObjectGenericODE2(nodeNumbers=[nMass0,nMass1],
                                                massMatrix=M,
                                                stiffnessMatrix=K,
                                                dampingMatrix=D))

#EXAMPLE for single node marker on super element body, mesh node 1; compare results to
ObjectGenericODE2 example!!!
mSuperElement = mbs.AddMarker(MarkerSuperElementPosition(bodyNumber=oGenericODE2,
                                                          meshNodeNumbers=[1], weightingFactors=[1]))
mbs.AddLoad(Force(markerNumber = mSuperElement, loadVector = [10, 0, 0]))

#assemble and solve system for default parameters
mbs.Assemble()

exu.SolveDynamic(mbs, solverType = exdyn.DynamicSolverType.TrapezoidalIndex2)

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

```

For examples on MarkerSuperElementPosition see Examples and TestModels:

- [NGsolvePistonEngine.py](#) (Examples/)
- [NGsolvePostProcessingStresses.py](#) (Examples/)
- [objectFFRFreducedOrderNetgen.py](#) (Examples/)
- [objectFFRFreducedOrderAccelerations.py](#) (TestModels/)
- [objectFFRFreducedOrderStressModesTest.py](#) (TestModels/)
- [objectFFRFreducedOrderTest.py](#) (TestModels/)
- [objectFFRFTTest.py](#) (TestModels/)
- [objectFFRFTTest2.py](#) (TestModels/)
- [objectGenericODE2Test.py](#) (TestModels/)
- [perfObjectFFRFreducedOrder.py](#) (TestModels/)
- [superElementRigidJointTest.py](#) (TestModels/)

### 7.9.11 MarkerSuperElementRigid

A position and orientation (rigid-body) marker attached to a SuperElement, such as ObjectFFRF, ObjectGenericODE2 and ObjectFFRFreducedOrder (for which it may be inefficient). The marker acts on the mesh nodes, not on the underlying nodes of the object. Note that in contrast to the MarkerSuperElementPosition, this marker needs a set of interface nodes which are not aligned at one line, such that these node points can represent a rigid body motion. Note that definitions of marker positions are slightly different from MarkerSuperElementPosition.

#### Additional information for MarkerSuperElementRigid:

- The Marker has the following types = Object, Body, Position, Orientation

The item **MarkerSuperElementRigid** with type = 'SuperElementRigid' has the following parameters:

Name	type	size	default value	description
name	String		''	marker's unique name
bodyNumber	ObjectIndex		MAXINT	body number to which marker is attached to
offset	Vector3D	3	[0,0,0.]	local marker SuperElement reference position offset used to correct the center point of the marker, which is computed from the weighted average of reference node positions (which may have some offset to the desired joint position). Note that this offset shall be small and larger offsets can cause instability in simulation models (better to have symmetric meshes at joints).
meshNodeNumbers	ArrayIndex		[]	a list of $n_m$ mesh node numbers of superelement (=interface nodes) which are used to compute the body-fixed marker position and orientation; the related nodes must provide 3D position information, such as NodePoint, NodePoint2D, NodeRigidBody[...]; in order to retrieve the global node number, the generic body needs to convert local into global node numbers
weightingFactors	Vector		[]	a list of $n_m$ weighting factors per node to compute the final local position and orientation; these factors could be based on surface integrals of the constrained mesh faces
useAlternativeApproach	Bool		True	this flag switches between two versions for the computation of the rotation and angular velocity of the marker; alternative approach uses skew symmetric matrix of reference position; follows the inertia concept
visualization	VMarkerSuperElementRigid			parameters for visualization of item

The item VMarkerSuperElementRigid has the following parameters:

<b>Name</b>	<b>type</b>	<b>size</b>	<b>default value</b>	<b>description</b>
show	Bool		True	set true, if item is shown in visualization and false if it is not shown
showMarkerNodes	Bool		True	set true, if all nodes are shown (similar to marker, but with less intensity)

---

### 7.9.11.1 DESCRIPTION of MarkerSuperElementRigid:

#### Information on input parameters:

<b>input parameter</b>	<b>symbol</b>	<b>description see tables above</b>
bodyNumber	$n_b$	
offset	${}^r \mathbf{o}_{ref}$	
meshNodeNumbers	$[k_0, \dots, k_{n_m-1}]^T$	
weightingFactors	$[w_0, \dots, w_{n_m-1}]^T$	

#### Definition of marker quantities:

<b>intermediate variables</b>	<b>symbol</b>	<b>description</b>
number of mesh nodes	$n_m$	size of meshNodeNumbers and weightingFactors which are marked; this must not be the number of mesh nodes in the marked object
mesh node number	$i = k_i$	abbreviation, runs over all marker mesh nodes
mesh node local displacement	${}^r \mathbf{u}^{(i)}$	current local (within reference frame $r$ ) displacement of mesh node $k_i$ in object $n_b$
mesh node local position	${}^r \mathbf{p}^{(i)} = {}^r \mathbf{x}_{ref}^{(i)} + {}^r \mathbf{u}^{(i)}$	current local (within reference frame $r$ , which is the body frame $b$ , e.g., in ObjectFFRFreducedOrder) position of mesh node $k_i$ in object $n_b$
mesh node local reference position	${}^r \mathbf{x}_{ref}^{(i)}$	local (within reference frame $r$ ) reference position of mesh node $k_i$ in object $n_b$ , see e.g. ObjectFFRFreducedOrder
averaged local reference position	${}^r \mathbf{x}_{ref}^{avg} = \sum_i w_i {}^r \mathbf{x}_{ref}^{(i)}$	midpoint reference position of marker; averaged local reference positions of all mesh nodes $k_i$ , using weighting for averaging; may not coincide with center point of your idealized joint surface (e.g., midpoint of cylinder), see Fig. 7.10
marker centered mesh node local reference position	${}^r \mathbf{p}_{ref}^{(i)} = {}^r \mathbf{x}_{ref}^{(i)} - {}^r \mathbf{x}_{ref}^{avg}$	local reference position of mesh node $k_i$ relative to the center position of marker
mesh node local velocity	${}^r \mathbf{v}^{(i)}$	current local (within reference frame $r$ ) velocity of mesh node $k_i$ in object $n_b$
super element reference point	${}^0 \mathbf{p}_r$ ( $= {}^0 \mathbf{p}_t$ in ObjectFFRFreducedOrder)	current position (origin) of super element's floating frame ( $r$ ), which is zero, if the object does not provide a reference frame (such as GenericODE2)

super element rotation matrix	${}^0r\mathbf{A}$	current rigid body transformation matrix of super element's floating frame (r), which is the identity matrix, if the object does not provide a reference frame (such as GenericODE2)
super element angular velocity	${}^r\boldsymbol{\omega}_r$	current local angular velocity of super element's floating frame (r), which is zero, if the object does not provide a reference frame (such as GenericODE2)
marker position	${}^0\mathbf{p}_m = {}^0\mathbf{p}_r + {}^0r\mathbf{A} \left( {}^r\mathbf{o}_{ref} + \sum_i w_i \cdot {}^r\mathbf{p}^{(i)} \right)$	current global position which is provided by marker; note offset ${}^r\mathbf{o}_{ref}$ added, if used as a correction of marker mesh nodes
marker velocity	$\begin{aligned} {}^0\mathbf{v}_m &= {}^0\dot{\mathbf{p}}_r + \\ {}^0r\mathbf{A} \left( {}^r\tilde{\boldsymbol{\omega}}_r \sum_i (w_i \cdot {}^r\mathbf{p}^{(i)}) + \sum_i (w_i \cdot {}^r\dot{\mathbf{u}}^{(i)}) \right) \end{aligned}$	current global velocity which is provided by marker
marker rotation matrix	${}^0r\mathbf{A}_m = {}^0r\mathbf{A} \cdot \exp({}^r\boldsymbol{\theta}_m)$	current rotation matrix, which transforms the local marker coordinates and adds the rigid body transformation of floating frames ${}^0r\mathbf{A}$ ; uses exponential map for SO3, assumes that $\boldsymbol{\theta}$ represents a rotation vector
marker local rotation	${}^r\boldsymbol{\theta}_m$	current local linearized rotations (rotation vector); for the computation, see below for the standard and alternative approach
marker local angular velocity	${}^r\boldsymbol{\omega}_m$	local angular velocity due to mesh node velocity only; for the computation, see below for the standard and alternative approach
marker global angular velocity	${}^0\boldsymbol{\omega}_m = {}^0\boldsymbol{\omega}_r + {}^0r\mathbf{A} {}^r\boldsymbol{\omega}_m$	current global angular velocity

### 7.9.11.2 Marker background

The marker allows to realize a multi-point constraint (assuming that the marker is used in a joint constraint), connecting to averaged nodal displacements and rotations (also known as RBE3 in NASTRAN), see e.g. [21]. However, using Craig-Bampton RBE2 modes, will create RBE2 multi-point constraints for `ObjectFFRFreducedOrder` objects.

For more information on the various quantities and their coordinate systems, see table above and Fig. 7.10.

### 7.9.11.3 Marker quantities

The marker provides a 'position' jacobian, which is the derivative of the global marker velocity w.r.t. the object velocity coordinates  $\dot{\mathbf{q}}_{n_b}$ ,

$${}^0\mathbf{J}_{m, pos} = \frac{\partial {}^0\mathbf{v}_m}{\partial \dot{\mathbf{q}}_{n_b}} . \quad (7.345)$$

In case of `ObjectGenericODE2`, assuming pure displacement based nodes, the jacobian will consist of zeros and unit matrices  $\mathbf{I}$ ,

$${}^0\mathbf{J}_{m, pos}^{\text{GenericODE2}} = \frac{\partial {}^0\mathbf{v}_m}{\partial \dot{\mathbf{q}}_{n_b}} = [\mathbf{0}, \dots, \mathbf{0}, \mathbf{I}, \mathbf{0}, \dots, \mathbf{0}, \mathbf{I}, \mathbf{0}, \dots, \mathbf{0}] , \quad (7.346)$$

in which the  $\mathbf{I}$  matrices are placed at the according indices of marker nodes.

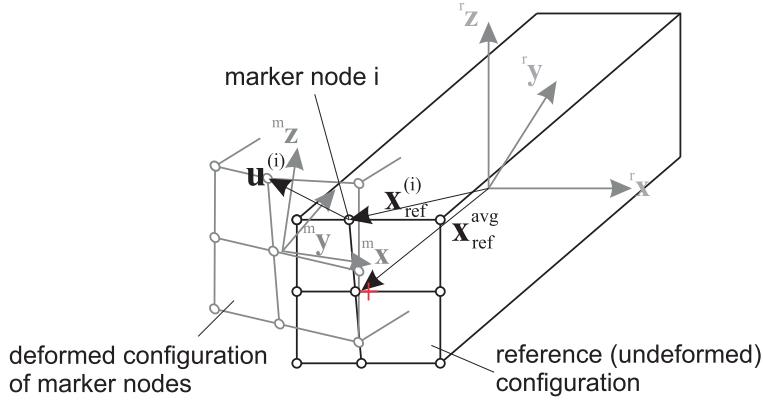


Figure 7.10: Sketch of marker nodes, exemplary node  $i$ , reference coordinates and marker coordinate system; note the difference of the center of the marker ‘surface’ (rectangle) marked with the red cross, and the averaged of the averaged local reference position.

In case of `ObjectFFRFreducedOrder`, this jacobian is computed as weighted sum of the position jacobians, see `ObjectFFRFreducedOrder`,

$${}^0\mathbf{J}_{m, pos}^{FFRFreduced} = \frac{\partial {}^0\mathbf{v}_m}{\partial \dot{\mathbf{q}}_{nb}} = \sum_i w_i {}^0\mathbf{J}_{pos}^{(i)} = \left[ \mathbf{I}, -{}^{0r}\mathbf{A} \left( \sum_i ({}^r\tilde{\mathbf{u}}_f^{(i)} + {}^r\tilde{\mathbf{x}}_{ref}^{(i)}) \right) {}^r\mathbf{G}, \sum_i w_i {}^{0r}\mathbf{A} \begin{bmatrix} {}^r\Psi_{r=3i}^T \\ {}^r\Psi_{r=3i+1}^T \\ {}^r\Psi_{r=3i+2}^T \end{bmatrix} \right]. \quad (7.347)$$

In `ObjectFFRFreducedOrder`, the jacobian usually affects all reduced coordinates.

#### 7.9.11.4 Standard approach for computation of rotation (`useAlternativeApproach = False`)

As compared to `MarkerSuperElementPosition`, `MarkerSuperElementRigid` also links the marker to the orientation of the set of nodes provided. For this reason, the check performed in `mbs.assemble()` will take care that the nodes are capable to describe rotations. The first approach, here called as a standard, follows the idea that displacements contribute to rotation are weighted by their quadratic distance, cf. [21], and gives the (small rotation) rotation vector

$${}^r\boldsymbol{\theta}_m = \frac{\sum_i w_i {}^r\mathbf{p}_{ref}^{(i)} \times {}^r\mathbf{u}^{(i)}}{\sum_i w_i |{}^r\mathbf{p}_{ref}^{(i)}|^2} \quad (7.348)$$

Note that  $\mathbf{p}_{ref}^{(i)}$  is not the reference position in the `ObjectFFRFreducedOrder` object, but it is relative to the midpoint reference position all marker nodes, given in  ${}^r\mathbf{x}_{ref}^{avg}$ . Accordingly, the marker local angular velocity can be calculated as

$${}^r\boldsymbol{\omega}_m = {}^r\dot{\boldsymbol{\theta}}_m = \frac{\sum_i w_i {}^r\tilde{\mathbf{p}}_{ref}^{(i)} {}^r\mathbf{v}_i}{\sum_i w_i |{}^r\mathbf{p}_{ref}^{(i)}|^2} \quad (7.349)$$

The marker also provides a ‘rotation’ jacobian, which is the derivative of the marker angular velocity  ${}^0\boldsymbol{\omega}_m$  w.r.t. the object velocity coordinates  $\dot{\mathbf{q}}_{nb}$ ,

$${}^0\mathbf{J}_{m, rot} = \frac{\partial {}^0\boldsymbol{\omega}_m}{\partial \dot{\mathbf{q}}_{nb}} = \frac{\partial {}^{0r}\mathbf{A} ({}^r\boldsymbol{\omega}_r + {}^r\boldsymbol{\omega}_m)}{\partial \dot{\mathbf{q}}_{nb}} = {}^{0r}\mathbf{A} \left( \frac{\partial {}^r\boldsymbol{\omega}_r}{\partial \dot{\mathbf{q}}_{nb}} + \frac{\sum_i w_i {}^r\tilde{\mathbf{p}}_{ref}^{(i)} {}^r\mathbf{J}_{pos}^{(i)}}{\sum_i w_i |{}^r\mathbf{p}_{ref}^{(i)}|^2} \right) \quad (7.350)$$

In case of `ObjectFFRFreducedOrder`, this jacobian is computed as

$${}^0\mathbf{J}_{m,rot}^{FFRFreduced} = \left[ \mathbf{0}, {}^{0r}\mathbf{A} {}^r\mathbf{G}_{local}, {}^{0r}\mathbf{A} \frac{\sum_i w_i {}^r\tilde{\mathbf{p}}_{ref}^{(i)} {}^r\mathbf{J}_{pos,f}^{(i)}}{\sum_i w_i |{}^r\tilde{\mathbf{p}}_{ref}^{(i)}|^2} \right] \quad (7.351)$$

in which you should know that

- we used  $\frac{\partial {}^r\boldsymbol{\omega}_r}{\partial \theta_r} = {}^r\mathbf{G}_{local}$ ,
- $\theta_r$  represent the rotation parameters for the rigid body node of `ObjectFFRFreducedOrder`,
- ${}^r\mathbf{J}_{pos,f}^{(i)}$  is the **local** jacobian, which only includes the flexible part of the local jacobian for a single mesh node,  ${}^r\mathbf{J}_{pos}^{(i)}$  (note the small  $r$  on the upper left), as defined in `ObjectFFRFreducedOrder`.

For further quantities also consult the according description in `ObjectFFRFreducedOrder`.

### 7.9.11.5 Alternative computation of rotation (`useAlternativeApproach = True`)

Note that this approach is **still under development** and needs further validation. However, tests show that this model is superior to the standard approach, as it improves the averaging of motion w.r.t. rotations at the marker nodes.

In the alternative approach, the weighting matrix  $\mathbf{W}$  has the interpretation of an inertia tensor built from nodes using weights equal to node masses. In such an interpretation, the 'local angular momentum' w.r.t. the marker (averaged) position can be computed as

$$\mathbf{W} {}^r\boldsymbol{\omega}_m = \sum_i w_i {}^r\tilde{\mathbf{p}}_{ref}^{(i)} ({}^r\mathbf{v}^{(i)} - {}^r\mathbf{v}^{\text{avg}}) = - \sum_i (w_i {}^r\tilde{\mathbf{p}}_{ref}^{(i)} {}^r\tilde{\mathbf{p}}_{ref}^{(i)}) {}^r\boldsymbol{\omega}_m \quad (7.352)$$

which implicitly defines the weighting matrix  $\mathbf{W}$ , which must be invertable (but it is only a  $3 \times 3$  matrix!),

$$\mathbf{W} = - \sum_i w_i {}^r\tilde{\mathbf{p}}_{ref}^{(i)} {}^r\tilde{\mathbf{p}}_{ref}^{(i)} \quad (7.353)$$

Furthermore, we need to introduce the averaged velocity of the marker averaged reference position, using  ${}^r\dot{\mathbf{u}}^{(i)} = {}^r\mathbf{v}^{(i)}$ , which is defined as

$${}^r\mathbf{v}^{\text{avg}} = \sum_i w_i {}^r\mathbf{v}^{(i)}, \quad (7.354)$$

similar to the averaged local reference position  ${}^r\mathbf{x}_{ref}^{\text{avg}}$  given in the table above, see also Fig. 7.10.

In the alternative approach, thus the marker local rotations read

$${}^r\boldsymbol{\theta}_{m,alt} = \mathbf{W}^{-1} \sum_i w_i {}^r\tilde{\mathbf{p}}_{ref}^{(i)} ({}^r\mathbf{u}^{(i)} - {}^r\mathbf{x}_{ref}^{\text{avg}}), \quad (7.355)$$

and the marker local angular velocity is defined as

$${}^r\boldsymbol{\omega}_{m,alt} = \mathbf{W}^{-1} \sum_i w_i {}^r\tilde{\mathbf{p}}_{ref}^{(i)} ({}^r\mathbf{v}^{(i)} - {}^r\mathbf{v}^{\text{avg}}). \quad (7.356)$$

Note that, the average velocity  ${}^r\mathbf{v}^{\text{avg}}$  would cancel out in a symmetric mesh, but would cause spurious angular velocities in unsymmetric (w.r.t. the axis of rotation) distribution of mesh nodes. This could even lead to spurious rotations or angular velocities in pure translatory motion.

In the alternative mode, the Jacobian for the rotation / angular velocity is defined as

$${}^0\mathbf{J}_{m,rot,alt} = \frac{\partial {}^0\boldsymbol{\omega}_m}{\partial \dot{\mathbf{q}}_{nb}} = \frac{\partial {}^{0r}\mathbf{A} ({}^r\boldsymbol{\omega}_r + {}^r\boldsymbol{\omega}_m)}{\partial \dot{\mathbf{q}}_{nb}} = {}^{0r}\mathbf{A} \left( \frac{\partial {}^r\boldsymbol{\omega}_r}{\partial \dot{\mathbf{q}}_{nb}} + \mathbf{W}^{-1} \sum_i w_i {}^r\tilde{\mathbf{p}}_{ref}^{(i)} {}^r\mathbf{J}_{pos}^{(i)} \right) \quad (7.357)$$

In case of `ObjectFFRFreducedOrder`, this jacobian is computed as

$${}^0\mathbf{J}_{m,rot,alt}^{FFRFreduced} = \left[ \mathbf{0}, {}^{0r}\mathbf{A} {}^r\mathbf{G}_{local}, {}^{0r}\mathbf{A} \mathbf{W}^{-1} \sum_i w_i {}^r\tilde{\mathbf{p}}_{ref}^{(i)} {}^r\mathbf{J}_{pos,f}^{(i)} \right] \quad (7.358)$$

see also the descriptions given after Eq. (7.351) in the ‘standard’ approach.

#### EXAMPLE for marker on body 4, mesh nodes 10,11,12,13:

```
MarkerSuperElementRigid(bodyNumber = 4, meshNodeNumber = [10, 11, 12, 13], weightingFactors = [0.25, 0.25, 0.25, 0.25], referencePosition=[0,0,0])
```

For detailed examples, see `TestModels`.

For examples on `MarkerSuperElementRigid` see Examples and `TestModels`:

- [`CMSexampleCourse.py`](#) (Examples/)
- [`NGsolveCMSTutorial.py`](#) (Examples/)
- [`NGsolveCraigBampton.py`](#) (Examples/)
- [`ObjectFFRFconvergenceTestBeam.py`](#) (Examples/)
- [`ObjectFFRFconvergenceTestHinge.py`](#) (Examples/)
- [`pendulumVerify.py`](#) (Examples/)
- [`superElementRigidJointTest.py`](#) (TestModels/)

### 7.9.12 MarkerObjectODE2Coordinates

A Marker attached to all coordinates of an object (currently only body is possible), e.g. to apply special constraints or loads on all coordinates. The measured coordinates INCLUDE reference + current coordinates.

**Additional information for MarkerObjectODE2Coordinates:**

- The Marker has the following types = Object, Body, Coordinate

The item **MarkerObjectODE2Coordinates** with type = 'ObjectODE2Coordinates' has the following parameters:

Name	type	size	default value	description
name	String		''	marker's unique name
objectNumber	ObjectIndex		MAXINT	body number to which marker is attached to
visualization	VMarkerObjectODE2Coordinates			parameters for visualization of item

The item VMarkerObjectODE2Coordinates has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

#### 7.9.12.1 DESCRIPTION of MarkerObjectODE2Coordinates:

---

For examples on MarkerObjectODE2Coordinates see Examples and TestModels:

- [coordinateVectorConstraintGenericODE2.py](#) (TestModels/)

### 7.9.13 MarkerBodyCable2DShape

A special Marker attached to a 2D ANCF beam finite element with cubic interpolation and 8 coordinates.

#### Additional information for MarkerBodyCable2DShape:

- The Marker has the following types = Object, Body, Coordinate

The item **MarkerBodyCable2DShape** with type = 'BodyCable2DShape' has the following parameters:

Name	type	size	default value	description
name	String		"	marker's unique name
bodyNumber	ObjectIndex		MAXINT	body number to which marker is attached to
numberOfSegments	PInt		3	number of number of segments; each segment is a line and is associated to a data (history) variable; must be same as in according contact element
verticalOffset	Real		0.	vertical offset from beam axis in positive (local) Y-direction; this offset accounts for consistent computation of positions and velocities at the surface of the beam
visualization	VMarkerBodyCable2DShape			parameters for visualization of item

The item **VMarkerBodyCable2DShape** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

#### 7.9.13.1 DESCRIPTION of MarkerBodyCable2DShape:

---

For examples on MarkerBodyCable2DShape see Examples and TestModels:

- [ANCF\\_contact\\_circle.py](#) (Examples/)
- [ANCF\\_contact\\_circle2.py](#) (Examples/)
- [ANCF\\_moving\\_rigidbody.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- [beltDriveReevingSystem.py](#) (Examples/)
- [sliderCrank3DwithANCFbeltDrive.py](#) (Examples/)
- [sliderCrank3DwithANCFbeltDrive2.py](#) (Examples/)
- [ANCFslidingAndALEjointTest.py](#) (TestModels/)
- [ANCFcontactCircleTest.py](#) (TestModels/)
- [ANCFcontactFrictionTest.py](#) (TestModels/)

- [ANCFmovingRigidBodyTest.py](#) (TestModels/)

### 7.9.14 MarkerBodyCable2DCoordinates

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

**Additional information for MarkerBodyCable2DCoordinates:**

- The Marker has the following types = Object, Body, Coordinate

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

Name	type	size	default value	description
name	String		"	marker's unique name
bodyNumber	ObjectIndex		MAXINT	body number to which marker is attached to
visualization	VMarkerBodyCable2DCoordinates			parameters for visualization of item

The item VMarkerBodyCable2DCoordinates has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

#### 7.9.14.1 DESCRIPTION of MarkerBodyCable2DCoordinates:

---

For examples on MarkerBodyCable2DCoordinates see Examples and TestModels:

- [ANCF\\_moving\\_rigidbody.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- [ANCF\\_slidingJoint2Drigid.py](#) (Examples/)
- [ANCF\\_switchingSlidingJoint2D.py](#) (Examples/)
- [ANCFmovingRigidBodyTest.py](#) (TestModels/)
- [modelUnitTests.py](#) (TestModels/)

## 7.10 Loads

A Load applies a (usually constant) force, torque, mass-proportional or generalized load onto Nodes or Objects via Markers

### 7.10.1 LoadForceVector

Load with (3D) force vector; attached to position-based marker.

**Additional information for LoadForceVector:**

- Requested marker type = **Position**
- **Short name for Python** = **Force**
- **Short name for Python** (visualization object) = **VForce**

The item **LoadForceVector** with type = 'ForceVector' has the following parameters:

Name	type	size	default value	description
name	String		"	load's unique name
markerNumber	MarkerIndex		MAXINT	marker's number to which load is applied
loadVector	Vector3D		[0.,0.,0.]	vector-valued load [SI:N]
bodyFixed	Bool		False	if bodyFixed is true, the load is defined in body-fixed (local) coordinates, leading to a follower force; if false: global coordinates are used
loadVectorUserFunction	PyFunctionVector3DmbsScalarVector3D	0		A Python function which defines the time-dependent load; see description below; NOTE that in static computations, the loadFactor is always 1 for forces computed by user functions (this means for the static computation, that a user function returning [t*5,t*1,0] corresponds to loadVector=[5,1,0] without a user function); NOTE that forces are drawn using the value of loadVector; thus the current values according to the user function are NOT shown in the render window; however, a sensor (SensorLoad) returns the user function force which is applied to the object; to draw forces with current user function values, use a graphics-DataUserFunction of a ground object
visualization	VLoadForceVector			parameters for visualization of item

The item **VLoadForceVector** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

### 7.10.1.1 DESCRIPTION of LoadForceVector:

Information on input parameters:

input parameter	symbol	description see tables above
loadVector	f	
loadVectorUserFunction	UF $\in \mathbb{R}^3$	

### 7.10.1.2 Details

The load vector acts on a body or node via the local (`bodyFixed = True`) or global coordinates of a body or at a node. The marker transforms the (translational) force via the according jacobian matrix of the object (or node) to object (or node) coordinates.

---

#### Userfunction: `loadVectorUserFunction(mbs, t, loadVector)`

A user function, which computes the force vector depending on time and object parameters, which is hereafter applied to object or node.

arguments / return	type or size	description
mbs	MainSystem	provides MainSystem mbs to which load belongs
t	Real	current time in mbs
loadVector	Vector3D	f copied from object; WARNING: this parameter does not work in combination with static computation, as it is changed by the solver over step time
return value	Vector3D	computed force vector

---

#### User function example:

```
from math import sin, cos, pi
def UForce(mbs, t, loadVector):
    return [loadVector[0]*sin(t*10*2*pi), 0, 0]
```

---

For examples on LoadForceVector see Examples and TestModels:

- [addPrismaticJoint.py](#) (Examples/)
- [addRevoluteJoint.py](#) (Examples/)
- [interactiveTutorial.py](#) (Examples/)
- [pendulumVerify.py](#) (Examples/)
- [solutionViewerTest.py](#) (Examples/)
- [SpringDamperMassUserFunction.py](#) (Examples/)

- [ANCF\\_cantilever\\_test.py](#) (Examples/)
- [ANCF\\_cantilever\\_test\\_dyn.py](#) (Examples/)
- [ANCF\\_contact\\_circle.py](#) (Examples/)
- [ANCF\\_contact\\_circle2.py](#) (Examples/)
- [ANCF\\_moving\\_rigidbody.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- ...
- [perf3DRigidBodies.py](#) (TestModels/)
- [plotSensorTest.py](#) (TestModels/)
- [revoluteJointPrismaticJointTest.py](#) (TestModels/)
- ...

## 7.10.2 LoadTorqueVector

Load with (3D) torque vector; attached to rigidbody-based marker.

**Additional information for LoadTorqueVector:**

- Requested marker type = Orientation
- Short name for Python = **Torque**
- Short name for Python (visualization object) = **VTorque**

The item **LoadTorqueVector** with type = 'TorqueVector' has the following parameters:

Name	type	size	default value	description
name	String		"	load's unique name
markerNumber	MarkerIndex		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	PyFunctionVector3Dmbs	ScalarVector3D 0		A Python function which defines the time-dependent load; see description below; see also notes on loadFactor and drawing in LoadForceVector! Example for Python function: def f(mbs, t, loadVector): return [loadVector[0]*np.sin(t*10**2*3.1415),0,0]
visualization	VLoadTorqueVector			parameters for visualization of item

The item **VLoadTorqueVector** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

### 7.10.2.1 DESCRIPTION of LoadTorqueVector:

**Information on input parameters:**

input parameter	symbol	description see tables above
loadVector	$\tau$	
loadVectorUserFunction	$UF \in \mathbb{R}^3$	

### 7.10.2.2 Details

The torque vector acts on a body or node via the local (`bodyFixed = True`) or global coordinates of a body or at a node. The marker transforms the torque via the according jacobian matrix of the object (or node) to object (or node) coordinates.

---

#### Userfunction: `loadVectorUserFunction(mbs, t, loadVector)`

A user function, which computes the torque vector depending on time and object parameters, which is hereafter applied to object or node.

arguments / return	type or size	description
<code>mbs</code>	MainSystem	provides MainSystem mbs to which load belongs
<code>t</code>	Real	current time in mbs
<code>loadVector</code>	Vector3D	$\tau$ copied from object; WARNING: this parameter does not work in combination with static computation, as it is changed by the solver over step time
<code>return value</code>	Vector3D	computed torque vector

---

#### User function example:

```
from math import sin, cos, pi
def UForce(mbs, t, loadVector):
    return [loadVector[0]*sin(t*10*2*pi), 0, 0]
```

---

For examples on LoadTorqueVector see Examples and TestModels:

- [leggedRobot.py](#) (Examples/)
- [reevingSystem.py](#) (Examples/)
- [sliderCrank3DwithANCFbeltDrive2.py](#) (Examples/)
- [ANCF\\_contact\\_circle.py](#) (Examples/)
- [ANCF\\_contact\\_circle2.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)
- [ANCF\\_tests2.py](#) (Examples/)
- [ANCF\\_test\\_halfcircle.py](#) (Examples/)
- [flexibleRotor3Dtest.py](#) (Examples/)
- [rigid3Dexample.py](#) (Examples/)
- [rigidBodyIMUtest.py](#) (Examples/)
- [rigidRotor3DbasicBehaviour.py](#) (Examples/)
- ...
- [ANCFbeltDrive.py](#) (TestModels/)
- [ANCFgeneralContactCircle.py](#) (TestModels/)
- [ANCFcontactCircleTest.py](#) (TestModels/)
- ...

### 7.10.3 LoadMassProportional

Load attached to MarkerBodyMass marker, applying a 3D vector load (e.g. the vector [0,-g,0] is used to apply gravitational loading of size g in negative y-direction).

**Additional information for LoadMassProportional:**

- Requested marker type = Body + BodyMass
- Short name for Python = **Gravity**
- Short name for Python (visualization object) = **VGravity**

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

Name	type	size	default value	description
name	String		"	load's unique name
markerNumber	MarkerIndex		MAXINT	marker's number to which load is applied
loadVector	Vector3D		[0.,0.,0.]	vector-valued load [SI:N/kg = m/s <sup>2</sup> ]; typically, this will be the gravity vector in global coordinates
loadVectorUserFunction	PyFunctionVector3DmbsScalarVector3D		0	A Python function which defines the time-dependent load; see description below; see also notes on loadFactor and drawing in LoadForceVector!
visualization	VLoadMassProportional			parameters for visualization of item

The item **VLoadMassProportional** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

#### 7.10.3.1 DESCRIPTION of LoadMassProportional:

**Information on input parameters:**

input parameter	symbol	description see tables above
loadVector	b	
loadVectorUserFunction	UF ∈ ℝ <sup>3</sup>	

#### 7.10.3.2 Details

The load applies a (translational) and distributed load proportional to the distributed body's density. The marker of type MarkerBodyMass transforms the loadVector via an according jacobian matrix to object coordinates.

### Userfunction: `loadVectorUserFunction(mbs, t, loadVector)`

A user function, which computes the mass proportional load vector depending on time and object parameters, which is hereafter applied to object or node.

arguments / return	type or size	description
mbs	MainSystem	provides MainSystem mbs to which load belongs
t	Real	current time in mbs
loadVector	Vector3D	<b>b</b> copied from object; WARNING: this parameter does not work in combination with static computation, as it is changed by the solver over step time
<b>return value</b>	Vector3D	computed load vector

Example of user function: functionality same as in `LoadForceVector`

---

#### 7.10.3.3 MINI EXAMPLE for LoadMassProportional

```
node = mbs.AddNode(NodePoint(referenceCoordinates = [1,0,0]))
body = mbs.AddObject(MassPoint(nodeNumber = node, physicsMass=2))
mMass = mbs.AddMarker(MarkerBodyMass(bodyNumber=body))
mbs.AddLoad(LoadMassProportional(markerNumber=mMass, loadVector=[0,0,-9.81]))

#assemble and solve system for default parameters
mbs.Assemble()
exu.SolveDynamic(mbs)

#check result
exudynTestGlobals.testResult = mbs.GetNodeOutput(node, exu.OutputVariableType.Position)
[2]
#final z-coordinate of position shall be -g/2 due to constant acceleration with g=-9.81
#result independent of mass
```

---

For examples on LoadMassProportional see Examples and TestModels:

- [CMexampleCourse.py](#) (Examples/)
- [finiteSegmentMethod.py](#) (Examples/)
- [NGsolveCMSTutorial.py](#) (Examples/)
- [NGsolveCraigBampton.py](#) (Examples/)
- [NGsolvePostProcessingStresses.py](#) (Examples/)
- [ObjectFFRFconvergenceTestBeam.py](#) (Examples/)
- [ObjectFFRFconvergenceTestHinge.py](#) (Examples/)
- [pendulumVerify.py](#) (Examples/)
- [ALEANCF\\_pipe.py](#) (Examples/)
- [ANCF\\_moving\\_rigidbody.py](#) (Examples/)
- [ANCF\\_slidingJoint2D.py](#) (Examples/)

- [ANCF\\_slidingJoint2Drigid.py](#) (Examples/)
- ...
- [fourBarMechanismRedundant.py](#) (TestModels/)
- [genericJointUserFunctionTest.py](#) (TestModels/)
- [modelUnitTests.py](#) (TestModels/)
- ...

#### 7.10.4 LoadCoordinate

Load with scalar value, which is attached to a coordinate-based marker; the load can be used e.g. to apply a force to a single axis of a body, a nodal coordinate of a finite element or a torque to the rotatory DOF of a rigid body.

**Additional information for LoadCoordinate:**

- Requested marker type = Coordinate

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

Name	type	size	default value	description
name	String		"	load's unique name
markerNumber	MarkerIndex		MAXINT	marker's number to which load is applied
load	Real		0.	scalar load [SI:N]
loadUserFunction	PyFunctionMbsScalar2		0	A Python function which defines the time-dependent load; see description below; see also notes on loadFactor and drawing in LoadForceVector!
visualization	VLoadCoordinate			parameters for visualization of item

The item **VLoadCoordinate** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

##### 7.10.4.1 DESCRIPTION of LoadCoordinate:

**Information on input parameters:**

input parameter	symbol	description see tables above
load	$f$	
loadUserFunction	$UF \in \mathbb{R}$	

##### 7.10.4.2 Details

The scalar load is applied on a coordinate defined by a Marker of type 'Coordinate', e.g., **MarkerNodeCoordinate**. This can be used to create simple 1D problems, or to simply apply a translational force on a Node or even a torque on a rotation coordinate (but take care for its meaning).

##### Userfunction: **loadUserFunction(mbs, t, load)**

A user function, which computes the scalar load depending on time and the object's load parameter.

arguments / return	type or size	description
mbs	MainSystem	provides MainSystem mbs to which load belongs
t	Real	current time in mbs
load	Real	<b>b</b> copied from object; WARNING: this parameter does not work in combination with static computation, as it is changed by the solver over step time
<b>return value</b>	Real	computed load

---

### User function example:

```
from math import sin, cos, pi
#this example uses the object's stored parameter load to compute a time-dependent load
def UFload(mbs, t, load):
    return load*sin(10*(2*pi)*t)

n0=mbs.AddNode(Point())
nodeMarker = mbs.AddMarker(MarkerNodeCoordinate(nodeNumber=n0, coordinate=0))
mbs.AddLoad(LoadCoordinate(markerNumber = markerCoordinate,
                           load = 10,
                           loadUserFunction = UFload))
```

---

For examples on LoadCoordinate see Examples and TestModels:

- [ComputeSensitivitiesExample.py](#) (Examples/)
- [coordinateSpringDamper.py](#) (Examples/)
- [geneticOptimizationSliderCrank.py](#) (Examples/)
- [lavalRotor2Dtest.py](#) (Examples/)
- [massSpringFrictionInteractive.py](#) (Examples/)
- [minimizeExample.py](#) (Examples/)
- [nMassOscillatorInteractive.py](#) (Examples/)
- [parameterVariationExample.py](#) (Examples/)
- [plotSensorExamples.py](#) (Examples/)
- [simulateInteractively.py](#) (Examples/)
- [slidercrankWithMassSpring.py](#) (Examples/)
- [springDamperTutorial.py](#) (Examples/)
- ...
- [ACNFslidingAndALEjointTest.py](#) (TestModels/)
- [contactCoordinateTest.py](#) (TestModels/)
- [driveTrainTest.py](#) (TestModels/)
- ...

## 7.11 Sensors

A Sensor is used to measure quantities during simulation. Sensors may be attached to Nodes, Objects, Markers or Loads. Sensor values may be directly read via mbs or can be continuously written to files or SensorRecorder during simulation. The exudyn.plot Python utility function PlotSensor(...) can be conveniently used to show Sensor values over time.

### 7.11.1 SensorNode

A sensor attached to a [ODE2](#) or [ODE1](#) node. The sensor measures OutputVariables and outputs values into a file, showing per line [time, sensorValue[0], sensorValue[1], ...]. Use SensorUserFunction to modify sensor results (e.g., transforming to other coordinates) and writing to file.

The item **SensorNode** with type = 'Node' has the following parameters:

Name	type	size	default value	description
name	String		"	sensor's unique name
nodeNumber	NodeIndex		MAXINT	node number to which sensor is attached to
writeToFile	Bool		True	True: write sensor output to file; flag is ignored (interpreted as False), if fileName=""
fileName	String		"	directory and file name for sensor file output; default: empty string generates sensor + sensorNumber + outputVariableType; directory will be created if it does not exist
outputVariableType	OutputVariableType		OutputVariableType::None OutputVariableType for sensor	
storeInternal	Bool		False	true: store sensor data in memory (faster, but may consume large amounts of memory); false: internal storage not available
visualization	VSensorNode			parameters for visualization of item

The item VSensorNode has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

#### 7.11.1.1 DESCRIPTION of SensorNode:

---

For examples on SensorNode see Examples and TestModels:

- [ANCFALEtest.py](#) (Examples/)
- [beltDriveReevingSystem.py](#) (Examples/)

- [geneticOptimizationSliderCrank.py](#) (Examples/)
- [gyroStability.py](#) (Examples/)
- [lugreFrictionODE1.py](#) (Examples/)
- [lugreFrictionTest.py](#) (Examples/)
- [massSpringFrictionInteractive.py](#) (Examples/)
- [mouseInteractionExample.py](#) (Examples/)
- [nMassOscillatorInteractive.py](#) (Examples/)
- [particleClusters.py](#) (Examples/)
- [plotSensorExamples.py](#) (Examples/)
- [reevingSystem.py](#) (Examples/)
- ...
- [ANCFbeltDrive.py](#) (TestModels/)
- [ANCFgeneralContactCircle.py](#) (TestModels/)
- [ANCFoutputTest.py](#) (TestModels/)
- ...

### 7.11.2 SensorObject

A sensor attached to any object except bodies (connectors, constraint, spring-damper, etc). As a difference to other SensorBody, the connector sensor measures quantities without a local position. The sensor measures OutputVariable and outputs values into a file, showing per line [time, sensorValue[0], sensorValue[1], ...]. Use SensorUserFunction to modify sensor results (e.g., transforming to other coordinates) and writing to file.

The item **SensorObject** with type = 'Object' has the following parameters:

Name	type	size	default value	description
name	String		''	sensor's unique name
objectNumber	ObjectIndex		MAXINT	object (e.g. connector) number to which sensor is attached to
writeToFile	Bool		True	True: write sensor output to file; flag is ignored (interpreted as False), if fileName = ''
fileName	String		''	directory and file name for sensor file output; default: empty string generates sensor + sensorNumber + outputVariableType; directory will be created if it does not exist
outputVariableType	OutputVariableType		OutputVariableType::None	OutputVariableType for sensor
storeInternal	Bool		False	true: store sensor data in memory (faster, but may consume large amounts of memory); false: internal storage not available
visualization	VSensorObject			parameters for visualization of item

The item VSensorObject has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown; sensors can be shown at the position associated with the object - note that in some cases, there might be no such position (e.g. data object)!

---

#### 7.11.2.1 DESCRIPTION of SensorObject:

---

For examples on SensorObject see Examples and TestModels:

- [bicycleIftommBenchmark.py](#) (Examples/)
- [ComputeSensitivitiesExample.py](#) (Examples/)
- [leggedRobot.py](#) (Examples/)
- [lugreFrictionTest.py](#) (Examples/)

- [minimizeExample.py](#) (Examples/)
- [parameterVariationExample.py](#) (Examples/)
- [plotSensorExamples.py](#) (Examples/)
- [serialRobotOldTests.py](#) (Examples/)
- [serialRobotTestDH2.py](#) (Examples/)
- [serialRobotTSD.py](#) (Examples/)
- [sliderCrank3DwithANCFbeltDrive.py](#) (Examples/)
- [springDamperTutorial.py](#) (Examples/)
- ...
- [carRollingDiscTest.py](#) (TestModels/)
- [geneticOptimizationTest.py](#) (TestModels/)
- [mecanumWheelRollingDiscTest.py](#) (TestModels/)
- ...

### 7.11.3 SensorBody

A sensor attached to a body-object with local position  ${}^b\mathbf{b}$ . As a difference to ObjectSensors, the body sensor needs a local position at which the sensor is attached to. The sensor measures OutputVariableBody and outputs values into a file, showing per line [time, sensorValue[0], sensorValue[1], ...]. Use SensorUserFunction to modify sensor results (e.g., transforming to other coordinates) and writing to file.

The item **SensorBody** with type = 'Body' has the following parameters:

Name	type	size	default value	description
name	String		"	sensor's unique name
bodyNumber	ObjectIndex		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; flag is ignored (interpreted as False), if fileName=""
fileName	String		"	directory and file name for sensor file output; default: empty string generates sensor + sensorNumber + outputVariableType; directory will be created if it does not exist
outputVariableType	OutputVariableType		OutputVariableType::None OutputVariableType for sensor	
storeInternal	Bool		False	true: store sensor data in memory (faster, but may consume large amounts of memory); false: internal storage not available
visualization	VSensorBody			parameters for visualization of item

The item VSensorBody has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

#### 7.11.3.1 DESCRIPTION of SensorBody:

**Information on input parameters:**

input parameter	symbol	description see tables above
localPosition	${}^b\mathbf{b}$	

For examples on SensorBody see Examples and TestModels:

- [beltDriveReevingSystem.py](#) (Examples/)
- [bicycleIFTOMMBenchmark.py](#) (Examples/)

- [finiteSegmentMethod.py](#) (Examples/)
- [fourBarMechanism3D.py](#) (Examples/)
- [rigidBodyIMUtest.py](#) (Examples/)
- [rigidBodyTutorial2.py](#) (Examples/)
- [rigidBodyTutorial3.py](#) (Examples/)
- [rigidRotor3DbasicBehaviour.py](#) (Examples/)
- [rigidRotor3DFWBW.py](#) (Examples/)
- [rigidRotor3Drunup.py](#) (Examples/)
- [sliderCrank3DwithANCFbeltDrive.py](#) (Examples/)
- [sliderCrank3DwithANCFbeltDrive2.py](#) (Examples/)
- ...
- [ANCFoutputTest.py](#) (TestModels/)
- [carRollingDiscTest.py](#) (TestModels/)
- [ConvexContactTest.py](#) (TestModels/)
- ...

#### 7.11.4 SensorSuperElement

A sensor attached to a SuperElement-object with mesh node number. As a difference to other ObjectSensors, the SuperElement sensor has a mesh node number at which the sensor is attached to. The sensor measures OutputVariableSuperElement and outputs values into a file, showing per line [time, sensorValue[0], sensorValue[1], ...]. Use SensorUserFunction to modify sensor results (e.g., transforming to other coordinates) and writing to file.

The item **SensorSuperElement** with type = 'SuperElement' has the following parameters:

<b>Name</b>	<b>type</b>	<b>size</b>	<b>default value</b>	<b>description</b>
name	String		"	sensor's unique name
bodyNumber	ObjectIndex		MAXINT	body (=object) number to which sensor is attached to
meshNodeNumber	UInt		MAXINT	mesh node number, which is a local node number with in the object (starting with 0); the node number may represent a real Node in mbs, or may be virtual and reconstructed from the object coordinates such as in ObjectFFRFreducedOrder
writeToFile	Bool		True	True: write sensor output to file; flag is ignored (interpreted as False), if fileName=""
fileName	String		"	directory and file name for sensor file output; default: empty string generates sensor + sensorNumber + outputVariableType; directory will be created if it does not exist
outputVariableType	OutputVariableType		OutputVariableType::None	OutputVariableType for sensor, based on the output variables available for the mesh nodes (see special section for super element output variables, e.g, in ObjectFFRFreducedOrder, <a href="#">Section 7.3.3.2</a> )
storeInternal	Bool		False	true: store sensor data in memory (faster, but may consume large amounts of memory); false: internal storage not available
visualization	VSensorSuperElement			parameters for visualization of item

The item VSensorSuperElement has the following parameters:

<b>Name</b>	<b>type</b>	<b>size</b>	<b>default value</b>	<b>description</b>
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

#### 7.11.4.1 DESCRIPTION of SensorSuperElement:

---

For examples on SensorSuperElement see Examples and TestModels:

- [CMSexampleCourse.py](#) (Examples/)
- [NGsolveCraigBampton.py](#) (Examples/)
- [NGsolvePostProcessingStresses.py](#) (Examples/)
- [ObjectFFRFconvergenceTestBeam.py](#) (Examples/)
- [objectFFRFreducedOrderNetgen.py](#) (Examples/)
- [pendulumVerify.py](#) (Examples/)
- [objectFFRFreducedOrderAccelerations.py](#) (TestModels/)
- [objectFFRFreducedOrderStressModesTest.py](#) (TestModels/)
- [objectFFRFreducedOrderTest.py](#) (TestModels/)
- [objectFFRFTTest.py](#) (TestModels/)
- [objectFFRFTTest2.py](#) (TestModels/)
- [objectGenericODE2Test.py](#) (TestModels/)
- [perfObjectFFRFreducedOrder.py](#) (TestModels/)
- [superElementRigidJointTest.py](#) (TestModels/)

### 7.11.5 SensorMarker

A sensor attached to a marker. The sensor measures the selected marker values and outputs values into a file, showing per line [time, sensorValue[0], sensorValue[1], ...]. Depending on markers, it can measure Coordinates (MarkerNodeCoordinate), Position and Velocity (MarkerXXXPosition), Position, Velocity, Rotation and AngularVelocityLocal (MarkerXXXRigid). Note that marker values are only available for the current configuration. Use SensorUserFunction to modify sensor results (e.g., transforming to other coordinates) and writing to file

The item **SensorMarker** with type = 'Marker' has the following parameters:

<b>Name</b>	<b>type</b>	<b>size</b>	<b>default value</b>	<b>description</b>
name	String		"	sensor's unique name
markerNumber	MarkerIndex		MAXINT	marker number to which sensor is attached to
writeToFile	Bool		True	True: write sensor output to file; flag is ignored (interpreted as False), if fileName=""
fileName	String		"	directory and file name for sensor file output; default: empty string generates sensor + sensorNumber + outputVariableType; directory will be created if it does not exist
outputVariableType	OutputVariableType		OutputVariableType::_None	OutputVariableType for sensor; output variables are only possible according to markertype, see general description of SensorMarker
storeInternal	Bool		False	true: store sensor data in memory (faster, but may consume large amounts of memory); false: internal storage not available
visualization	VSensorMarker			parameters for visualization of item

The item VSensorMarker has the following parameters:

<b>Name</b>	<b>type</b>	<b>size</b>	<b>default value</b>	<b>description</b>
show	Bool		True	set true, if item is shown in visualization and false if it is not shown

---

#### 7.11.5.1 DESCRIPTION of SensorMarker:

---

For examples on SensorMarker see Examples and TestModels:

- [bicycleIftommBenchmark.py](#) (Examples/)
- [NGsolveCMSTutorial.py](#) (Examples/)
- [ObjectFFRFconvergenceTestHinge.py](#) (Examples/)

- [pendulumVerify.py](#) (Examples/)
- [pendulumFriction.py](#) (TestModels/)
- [plotSensorTest.py](#) (TestModels/)

### 7.11.6 SensorLoad

A sensor attached to a load. The sensor measures the load values and outputs values into a file, showing per line [time, sensorValue[0], sensorValue[1], ...]. Use SensorUserFunction to modify sensor results (e.g., transforming to other coordinates) and writing to file.

The item **SensorLoad** with type = 'Load' has the following parameters:

Name	type	size	default value	description
name	String		"	sensor's unique name
loadNumber	LoadIndex		MAXINT	load number to which sensor is attached to
writeToFile	Bool		True	True: write sensor output to file; flag is ignored (interpreted as False), if fileName=""
fileName	String		"	directory and file name for sensor file output; default: empty string generates sensor + sensorNumber + outputVariableType; directory will be created if it does not exist
storeInternal	Bool		False	true: store sensor data in memory (faster, but may consume large amounts of memory); false: internal storage not available
visualization	VSensorLoad			parameters for visualization of item

The item VSensorLoad has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown; sensor visualization CURRENTLY NOT IMPLEMENTED

#### 7.11.6.1 DESCRIPTION of SensorLoad:

For examples on SensorLoad see Examples and TestModels:

- [leggedRobot.py](#) (Examples/)
- [nMassOscillatorInteractive.py](#) (Examples/)
- [serialRobotOldTests.py](#) (Examples/)
- [serialRobotTestDH2.py](#) (Examples/)
- [serialRobotTSD.py](#) (Examples/)
- [simulateInteractively.py](#) (Examples/)
- [sliderCrank3DwithANCFbeltDrive2.py](#) (Examples/)
- [plotSensorTest.py](#) (TestModels/)
- [serialRobotTest.py](#) (TestModels/)
- [springDamperUserFunctionTest.py](#) (TestModels/)

### 7.11.7 SensorUserFunction

A sensor defined by a user function. The sensor is intended to collect sensor values of a list of given sensors and recombine the output into a new value for output or control purposes. It is also possible to use this sensor without any dependence on other sensors in order to generate output for, e.g., any quantities in mbs or solvers.

The item **SensorUserFunction** with type = 'UserFunction' has the following parameters:

Name	type	size	default value	description
name	String		"	sensor's unique name
sensorNumbers	ArraySensorIndex		[]	optional list of $n$ sensor numbers for use in user function
factors	Vector		[]	optional list of $m$ factors which can be used, e.g., for weighting sensor values
writeToFile	Bool		True	True: write sensor output to file; flag is ignored (interpreted as False), if fileName=""
fileName	String		"	directory and file name for sensor file output; default: empty string generates sensor + sensorNumber + outputVariableType; directory will be created if it does not exist
sensorUserFunction	PyFunctionVectorMbsScalarArrayIndexVector	0		Configuration A Python function which defines the time-dependent user function, which usually evaluates one or several sensors and computes a new sensor value, see example
storeInternal	Bool		False	true: store sensor data in memory (faster, but may consume large amounts of memory); false: internal storage not available
visualization	VSensorUserFunction			parameters for visualization of item

The item **VSensorUserFunction** has the following parameters:

Name	type	size	default value	description
show	Bool		True	set true, if item is shown in visualization and false if it is not shown; sensor visualization CURRENTLY NOT IMPLEMENTED

---

#### 7.11.7.1 DESCRIPTION of SensorUserFunction:

**Information on input parameters:**

input parameter	symbol	description see tables above
sensorNumbers	$\mathbf{n}_s = [s_0, \dots, s_n]^T$	
factors	$\mathbf{f}_s = [f_0, \dots, f_m]^T$	

---

## Userfunction: `sensorUserFunction(mbs, t, sensorNumbers, factors, configuration)`

A user function, which computes a sensor output from other sensor outputs (or from generic time dependent functions). The configuration in general will be the `exudyn.ConfigurationType.Current`, but others could be used as well except for `SensorMarker`. The user function arguments are as follows:

arguments / return	type or size	description
<code>mbs</code>	<code>MainSystem</code>	provides MainSystem <code>mbs</code> to which object belongs
<code>t</code>	<code>Real</code>	current time in <code>mbs</code>
<code>sensorNumbers</code>	$\text{Array } \in \mathbb{N}^n$	list of sensor numbers
<code>factors</code>	$\text{Vector } \in \mathbb{R}^n$	list of factors that can be freely used for the user function
<code>configuration</code>	<code>exudyn.ConfigurationType</code>	usually the <code>exudyn.ConfigurationType.Current</code> , but could also be different in user defined functions.
<code>return value</code>	$\text{Vector } \in \mathbb{R}^{n_r}$	returns list or numpy array of sensor output values; size $n_r$ , is implicitly defined by the returned list and may not be changed during simulation.

### User function example:

```

import exudyn as exu
from exudyn.itemInterface import *
from math import pi, atan2
SC = exu.SystemContainer()
mbs = SC.AddSystem()
node = mbs.AddNode(NodePoint(referenceCoordinates = [1,1,0],
                               initialCoordinates=[0,0,0],
                               initialVelocities=[0,-1,0]))
mbs.AddObject(MassPoint(nodeNumber = node, physicsMass=1))

sNode = mbs.AddSensor(SensorNode(nodeNumber=node, fileName='solution/sensorTest.txt',
                                  outputVariableType=exu.OutputVariableType.Position))

#user function for sensor, convert position into angle:
def UFsensor(mbs, t, sensorNumbers, factors, configuration):
    val = mbs.GetSensorValues(sensorNumbers[0]) #x,y,z
    phi = atan2(val[1],val[0]) #compute angle from x,y: atan2(y,x)
    return [factors[0]*phi] #return angle in degree

sUser = mbs.AddSensor(SensorUserFunction(sensorNumbers=[sNode], factors=[180/pi],
                                         fileName='solution/sensorTest2.txt',
                                         sensorUserFunction=UFsensor))

#assemble and solve system for default parameters
mbs.Assemble()

```

```
exu.SolveDynamic(mbs)

if False:
    from exudyn.plot import PlotSensor
    PlotSensor(mbs, [sNode, sNode, sUser], [0, 1, 0])
```

---

For examples on SensorUserFunction see Examples and TestModels:

- [bicycleIftommBenchmark.py](#) (Examples/)
- [lugreFrictionODE1.py](#) (Examples/)
- [lugreFrictionTest.py](#) (Examples/)
- [fourBarMechanismRedundant.py](#) (TestModels/)
- [sensorUserFunctionTest.py](#) (TestModels/)

# Chapter 8

## ExUDYN Settings and Solver Structures

This section includes the reference manual for settings which are available in the python interface, e.g. simulation settings, visualization settings, and structures for solvers. The data is auto-generated from the according interfaces in order to keep fully up-to-date with changes.

### 8.1 Simulation settings

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

#### 8.1.1 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
appendToFile	bool		False	flag (true/false); if true, solution and solver-Information is appended to existing file (otherwise created); in BINARY mode, files are always replaced and this parameter is ineffective!
binarySolutionFile	bool		False	if true, the solution file is written in binary format for improved speed and smaller file sizes; setting outputPrecision >= 8 uses double (8 bytes), otherwise float (4 bytes) is used; note that appendToFile is ineffective and files are always replaced without asking! If not provided, file ending will read .sol in case of binary files and .txt in case of text files

coordinatesSolutionFileName	FileName	'coordinatesSolution'	filename and (relative) path of solution file (coordinatesSolutionFile) containing all coordinates versus time; directory will be created if it does not exist; character encoding of string is up to your filesystem, but for compatibility, it is recommended to use letters, numbers and '_' only; filename ending will be added automatically if not provided: .txt in case of text mode and .sol in case of binary solution files (binarySolutionFile=True)
exportAccelerations	bool	True	add <a href="#">ODE2</a> accelerations to solution file (coordinatesSolutionFile)
exportAlgebraicCoordinates	bool	True	add algebraicCoordinates (=Lagrange multipliers) to solution file (coordinatesSolutionFile)
exportDataCoordinates	bool	True	add DataCoordinates to solution file (coordinatesSolutionFile)
exportODE1Velocities	bool	True	add coordinatesODE1_t to solution file (coordinatesSolutionFile)
exportVelocities	bool	True	add <a href="#">ODE2</a> velocities to solution file (coordinatesSolutionFile)
flushFilesDOF	PInt	10000	number of DOF, above which solution file (coordinatesSolutionFile) buffers are always flushed, irrespectively of whether flushFilesImmediately is set True or False (see also flushFilesImmediately); for larger files, writing takes so much time that flushing does not add considerable time
flushFilesImmediately	bool	False	flush file buffers after every solution period written (coordinatesSolutionFile and sensor files); if set False, the output is written through a buffer, which is highly efficient, but during simulation, files may be always in an incomplete state; if set True, this may add a large amount of CPU time as the process waits until files are really written to hard disc (especially for simulation of small scale systems, writing 10.000s of time steps; at least 5us per step/file, depending on hardware)
outputPrecision	UInt	10	precision for floating point numbers written to solution and sensor files

recordImagesInterval	Real	-1.	record frames (images) during solving; amount of time to wait until next image (frame) is recorded; set recordImages = -1. if no images shall be recorded; set, e.g., recordImages = 0.01 to record an image every 10 milliseconds (requires that the time steps / load steps are sufficiently small!); for file names, etc., see VisualizationSettings.exportImages
restartFileName	FileName	'restartFile.txt'	filename and (relative) path of text file for storing solution after every restartWritePeriod if writeRestartFile=True; backup file is created with ending .bck, which should be used if restart file is crashed; use Python utility function InitializeFromRestartFile(...) to consistently restart
restartWritePeriod	UReal	0.01	time span (period), determines how often the restart file is updated; this should be often enough to enable restart without too much loss of data; too low values may influence performance
sensorsAppendToFile	bool	False	flag (true/false); if true, sensor output is appended to existing file (otherwise created) or in case of internal storage, it is appended to existing currently stored data; this allows storing sensor values over different simulations
sensorsWriteFileHeader	bool	True	flag (true/false); if true, file header is written for sensor output (turn off, e.g. for multiple runs of time integration)
sensorsWritePeriod	UReal	0.01	time span (period), determines how often the sensor output is written to file or internal storage during a simulation
solutionInformation	String	"	special information added to header of solution file (e.g. parameters and settings, modes, ...); character encoding may be UTF-8, restricted to characters in <a href="#">Section 9.4</a> , but for compatibility, it is recommended to use ASCII characters only (95 characters, see wiki)
solutionWritePeriod	UReal	0.01	time span (period), determines how often the solution file (coordinatesSolutionFile) is written during a simulation

solverInformationFileName	FileName	'solverInformation.txt'	filename and (relative) path of text file showing detailed information during solving; detail level according to yourSolver.verboseModeFile; if solutionSettings.appendToFile is true, the information is appended in every solution step; directory will be created if it does not exist; character encoding of string is up to your filesystem, but for compatibility, it is recommended to use letters, numbers and '_' only
writeFileFooter	bool	True	flag (true/false); if true, information at end of simulation is written: convergence, total solution time, statistics
writeFileHeader	bool	True	flag (true/false); if true, file header is written (turn off, e.g. for multiple runs of time integration)
writeRestartFile	bool	False	flag (true/false), which determines if restart file is written regularly, see restartFileName for details
writeSolutionToFile	bool	True	flag (true/false), which determines if (global) solution vector is written to the solution file (coordinatesSolutionFile); standard quantities that are written are: solution is written as displacements and coordinatesODE1; for additional coordinates in the solution file, see the options below

### 8.1.2 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
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
doSystemWideDifferentiation	bool		False	True: system wide differentiation (e.g. all ODE2 equations w.r.t. all ODE2 coordinates); False: only local (object) differentiation

forAE	bool	False	flag (true/false); false = perform direct computation of jacobian for algebraic equations (AE), true = use numerical differentiation; as there must always exist an analytical implemented jacobian for AE, 'true' should only be used for verification
forODE2	bool	False	flag (true/false); false = perform direct computation (e.g., using autodiff) of jacobian for ODE2 equations, true = use numerical differentiation; numerical differentiation is less efficient and may lead to numerical problems, but may smoothen problems of analytical derivatives; sometimes the analytical derivative may neglect terms
forODE2connectors	bool	False	flag (true/false); false: if also forODE2==false, perform direct computation of jacobian for ODE2 terms for connectors; else: use numerical differentiation; NOTE: THIS FLAG IS FOR DEVELOPMENT AND WILL BE ERASED IN FUTURE
minimumCoordinateSize	UReal	1e-2	minimum size of coordinates in relative differentiation parameter
relativeEpsilon	UReal	1e-7	relative differentiation parameter epsilon; the numerical differentiation parameter $\varepsilon$ follows from the formula ( $\varepsilon = \varepsilon_{\text{relative}} * \max(q_{\min},  q_i + [q_i^{\text{Ref}}] )$ ), with $\varepsilon_{\text{relative}} = \text{relativeEpsilon}$ , $q_{\min} = \text{minimumCoordinateSize}$ , $q_i$ is the current coordinate which is differentiated, and $q_{\text{Ref}i}$ is the reference coordinate of the current coordinate

### 8.1.3 DiscontinuousSettings

Settings for discontinuous iterations, as in contact, friction, plasticity and general switching phenomena.

DiscontinuousSettings has the following items:

Name	type/function return type	size	default value / function args	description
ignoreMaxIterations	bool		True	continue solver if maximum number of discontinuous (post Newton) iterations is reached (ignore tolerance)
iterationTolerance	UReal		1	absolute tolerance for discontinuous (post Newton) iterations; the errors represent absolute residuals and can be quite high
maxIterations	UInt		5	maximum number of discontinuous (post Newton) iterations

### 8.1.4 NewtonSettings

Settings for Newton method used in static or dynamic simulation.

NewtonSettings has the following items:

Name	type/function return type	size	default value / function args	description
numericalDifferentiation	NumericalDifferentiationSettings			numerical differentiation parameters for numerical jacobian (e.g. Newton in static solver or implicit time integration)
absoluteTolerance	UReal		1e-10	absolute tolerance of residual for Newton (needed e.g. if residual is fulfilled right at beginning); condition: $\text{sqrt}(q^*q)/\text{numberOfCoordinates} \leq \text{absoluteTolerance}$
adaptInitialResidual	bool		True	flag (true/false); false = standard; True: if initialResidual is very small (or zero), it may increase dramatically in first step; to achieve relativeTolerance, the initialResidual will be updated by a higher residual within the first Newton iteration
maximumSolutionNorm	UReal		1e38	this is the maximum allowed value for solutionU.L2NormSquared() which is the square of the square norm ( $\text{value} = u_1^2 + u_2^2 + \dots$ ), 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)
maxIterations	UInt		25	maximum number of iterations (including modified + restart Newton steps); after that iterations, the static/dynamic solver stops with error
maxModifiedNewtonIterations	UInt		8	maximum number of iterations for modified Newton (without Jacobian update); after that number of iterations, the modified Newton method gets a Jacobian update and is further iterated
maxModifiedNewtonRestartIterations	UInt		7	maximum number of iterations for modified Newton after a Jacobian update; after that number of iterations, the full Newton method is started for this step
modifiedNewtonContractivity	PReal		0.5	maximum contractivity (=reduction of error in every Newton iteration) accepted by modified Newton; if contractivity is greater, a Jacobian update is computed
modifiedNewtonJacUpdatePerStep	bool		False	True: compute Jacobian at every time step, but not in every iteration (except for bad convergence ==> switch to full Newton)

newtonResidualMode	UInt	0	0 ... use residual for computation of error (standard); 1 ... use <a href="#">ODE2</a> and <a href="#">ODE1</a> newton increment for error (set relTol and absTol to same values!) ==> may be advantageous if residual is zero, e.g., in kinematic analysis; TAKE CARE with this flag
relativeTolerance	UReal	1e-8	relative tolerance of residual for Newton (general goal of Newton is to decrease the residual by this factor)
useModifiedNewton	bool	False	True: compute Jacobian only at first step; no Jacobian updates per step; False: Jacobian computed in every step
useNewtonSolver	bool	True	flag (true/false); false = linear computation, true = use Newton solver for nonlinear solution
weightTolerancePerCoordinate	bool	False	flag (true/false); false = compute error as L2-Norm of residual; true = compute error as (L2-Norm of residual) / (sqrt(number of coordinates)), which can help to use common tolerance independent of system size

### 8.1.5 GeneralizedAlphaSettings

Settings for generalized-alpha, implicit trapezoidal or Newmark time integration methods.

GeneralizedAlphaSettings has the following items:

Name	type/function re-turn type	size	default value / func-tion args	description
computeInitialAccelerations	bool		True	True: compute initial accelerations from system EOM in acceleration form; NOTE that initial accelerations that are following from user functions in constraints are not considered for now! False: use zero accelerations
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
spectralRadius	UReal		0.9	spectral radius for Generalized-alpha solver; set this value to 1 for no damping or to $0 < \text{spectralRadius} < 1$ for damping of high-frequency dynamics; for position-level constraints (index 3), spectralRadius must be $< 1$

useIndex2Constraints	bool	False	set useIndex2Constraints = true in order to use index2 (velocity level constraints) formulation
useNewmark	bool	False	if true, use Newmark method with beta and gamma instead of generalized-Alpha

### 8.1.6 ExplicitIntegrationSettings

Settings for generalized-alpha, implicit trapezoidal or Newmark time integration methods.

ExplicitIntegrationSettings has the following items:

Name	type/function re- turn type	size	default value / func- tion args	description
computeEndOfStepAccelerations	bool		True	accelerations are computed at stages of the explicit integration scheme; if the user needs accelerations at the end of a step, this flag needs to be activated; if True, this causes a second call to the RHS of the equations, which may DOUBLE COMPUTATIONAL COSTS for one-step-methods; if False, the accelerations are re-used from the last stage, being slightly different
dynamicSolverType	DynamicSolverType		DynamicSolverType::DOPRI5	selection of explicit solver type (DOPRI5, ExplicitEuler, ExplicitMidpoint, RK44, RK67, ...), for detailed description see DynamicSolverType, <a href="#">Section 4.9.5</a> , but only referring to explicit solvers.
eliminateConstraints	bool		True	True: make explicit solver work for simple CoordinateConstraints, which are eliminated for ground constraints (e.g. fixed nodes in finite element models). False: incompatible constraints are ignored (BE CAREFUL)!
useLieGroupIntegration	bool		True	True: use Lie group integration for rigid body nodes; must be turned on for Lie group nodes, but also improves integration of other rigid body nodes. Only available for RK44 integrator.

### 8.1.7 TimeIntegrationSettings

General parameters used in time integration; specific parameters are provided in the according solver settings, e.g. for generalizedAlpha.

TimeIntegrationSettings has the following items:

Name	type/function return type	size	default value / function args	description
discontinuous	DiscontinuousSettings			parameters for treatment of discontinuities
explicitIntegration	ExplicitIntegrationSettings			special parameters for explicit time integration
generalizedAlpha	GeneralizedAlphaSettings			parameters for generalized-alpha, implicit trapezoidal rule or Newmark (options only apply for these methods)
newton	NewtonSettings			parameters for Newton method; used for implicit time integration methods only
absoluteTolerance	UReal	1e-8		$a_{tol}$ : if automaticStepSize=True, absolute tolerance for the error control; must fulfill $a_{tol} > 0$ ; see <a href="#">Section 11.3</a>
adaptiveStep	bool	True		True: the step size may be reduced if step fails; no automatic stepsize control
adaptiveStepDecrease	UReal	0.5		Multiplicative factor (MUST BE: $0 < \text{factor} < 1$ ) for step size to decrease due to discontinuousIteration or Newton errors
adaptiveStepIncrease	UReal	2		Multiplicative factor (MUST BE $> 1$ ) for step size to increase after previous step reduction due to discontinuousIteration or Newton errors
adaptiveStepRecoveryIterations	UInt	7		Number of max. (Newton iterations + discontinuous iterations) at which a step increase is considered; in order to immediately increase steps after reduction, chose a high value
adaptiveStepRecoverySteps	UInt	10		Number of steps needed after which steps will be increased after previous step reduction due to discontinuousIteration or Newton errors
automaticStepSize	bool	True		True: for specific integrators with error control (e.g., DOPRI5), compute automatic step size based on error estimation; False: constant step size (step may be reduced if adaptiveStep=True); the maximum stepSize reads $h = h_{max} = \frac{t_{end}-t_{start}}{n_{steps}}$
endTime	UReal	1		$t_{end}$ : end time of time integration
initialStepSize	UReal	0		$h_{init}$ : if automaticStepSize=True, initial step size; if initialStepSize==0, max. stepSize, which is $(endTime-startTime)/\text{numberOfSteps}$ , is used as initial guess; a good choice of initialStepSize may help the solver to start up faster.
minimumStepSize	PReal	1e-8		$h_{min}$ : if automaticStepSize=True or adaptiveStep=True: lower limit of time step size, before integrator stops with adaptiveStep; lower limit of automaticStepSize control (continues but raises warning)

numberOfSteps	PInt	100	$n_{steps}$ : number of steps in time integration; (maximum) stepSize $h$ is computed from $h = \frac{t_{end} - t_{start}}{n_{steps}}$ ; for automatic stepsize control, this stepSize is the maximum steps size, $h_{max} = h$
realtimeFactor	PReal	1	if simulateInRealtime=True, this factor is used to make the simulation slower than realtime (factor < 1) or faster than realtime (factor > 1)
relativeTolerance	UReal	1e-8	$r_{tol}$ : if automaticStepSize=True, relative tolerance for the error control; must fulfill $r_{tol} \geq 0$ ; see <a href="#">Section 11.3</a>
reuseConstantMassMatrix	bool	True	True: does not recompute constant mass matrices (e.g. of some finite elements, mass points, etc.); if False, it always recomputes the mass matrix (e.g. needed, if user changes mass parameters via Python)
simulateInRealtime	bool	False	True: simulate in realtime; the solver waits for computation of the next step until the CPU time reached the simulation time; if the simulation is slower than realtime, it simply continues
startTime	UReal	0	$t_{start}$ : start time of time integration (usually set to zero)
stepInformation	UInt	67	add up the following binary flags: 0 ... show only step time, 1 ... show time to go, 2 ... show newton iterations (Nit) per step or period, 4 ... show Newton jacobians (jac) per step or period, 8 ... show discontinuous iterations (Dit) per step or period, 16 ... show step size (dt), 32 ... show CPU time spent; 64 ... show adaptive step reduction warnings; 128 ... show step increase information; 1024 ... show every time step; time is usually shown in fractions of seconds (s), hours (h), or days
stepSizeMaxIncrease	UReal	2	$f_{maxInc}$ : if automaticStepSize=True, maximum increase of step size per step, see <a href="#">Section 11.3</a> ; make this factor smaller (but > 1) if too many rejected steps
stepSizeSafety	UReal	0.90	$r_{sfty}$ : if automaticStepSize=True, a safety factor added to estimated optimal step size, in order to prevent from many rejected steps, see <a href="#">Section 11.3</a> . Make this factor smaller if many steps are rejected.

verboseMode	UInt	0	0 ... no output, 1 ... show short step information every 2 seconds (every 30 seconds after 1 hour CPU time), 2 ... show every step information, 3 ... show also solution vector, 4 ... show also mass matrix and jacobian (implicit methods), 5 ... show also Jacobian inverse (implicit methods)
verboseModeFile	UInt	0	same behaviour as verboseMode, but outputs all solver information to file

### 8.1.8 StaticSolverSettings

Settings for static solver linear or nonlinear (Newton).

StaticSolverSettings has the following items:

Name	type / function return type	size	default value / function args	description
discontinuous	DiscontinuousSettings			parameters for treatment of discontinuities
newton	NewtonSettings			parameters for Newton method (e.g. in static solver or time integration)
adaptiveStep	bool		True	True: use step reduction if step fails; False: fixed step size
adaptiveStepDecrease	UReal		0.25	Multiplicative factor (MUST BE: 0 < factor < 1) for step size to decrease due to discontinuousIteration or Newton errors
adaptiveStepIncrease	UReal		2	Multiplicative factor (MUST BE > 1) for step size to increase after previous step reduction due to discontinuousIteration or Newton errors
adaptiveStepRecoveryIterations	UInt		7	Number of max. (Newton iterations + discontinuous iterations) at which a step increase is considered; in order to immediately increase steps after reduction, chose a high value
adaptiveStepRecoverySteps	UInt		4	Number of steps needed after which steps will be increased after previous step reduction due to discontinuousIteration or Newton errors
loadStepDuration	PReal		1	quasi-time for all load steps (added to current time in load steps)

loadStepGeometric	bool	False	if loadStepGeometric=false, the load steps are incremental (arithmetic series, e.g. 0.1,0.2,0.3,...); if true, the load steps are increased in a geometric series, e.g. for $n = 8$ numberOfLoadSteps and $d = 1000$ loadStepGeometricRange, it follows: $1000^{1/8}/1000 = 0.00237$ , $1000^{2/8}/1000 = 0.00562$ , $1000^{3/8}/1000 = 0.0133$ , ..., $1000^{7/8}/1000 = 0.422$ , $1000^{8/8}/1000 = 1$
loadStepGeometricRange	PReal	1000	if loadStepGeometric=true, the load steps are increased in a geometric series, see loadStepGeometric
loadStepStart	UReal	0	a quasi time, which can be used for the output (first column) as well as for time-dependent forces; quasi-time is increased in every step i by loadStepDuration/numberOfLoadSteps; loadStepTime = loadStepStart + i*loadStepDuration/numberOfLoadSteps, but loadStepStart untouched ==> increment by user
minimumStepSize	PReal	1e-8	lower limit of step size, before nonlinear solver stops
numberOfLoadSteps	PInt	1	number of load steps; if numberOfLoadSteps=1, no load steps are used and full forces are applied at once
stabilizerODE2term	UReal	0	add mass-proportional stabilizer term in <a href="#">ODE2</a> part of jacobian for stabilization (scaled), e.g. of badly conditioned problems; the diagonal terms are scaled with $stabilizer = (1 - loadStepFactor^2)$ , and go to zero at the end of all load steps: $loadStepFactor = 1 \rightarrow stabilizer = 0$
stepInformation	UInt	67	add up the following binary flags: 0 ... show only step time, 1 ... show time to go, 2 ... show newton iterations (Nit) per step or period, 4 ... show Newton jacobians (jac) per step or period, 8 ... show discontinuous iterations (Dit) per step or period, 16 ... show step size (dt), 32 ... show CPU time spent; 64 ... show adaptive step reduction warnings; 128 ... show step increase information; 1024 ... show every time step; time is usually shown in fractions of seconds (s), hours (h), or days
useLoadFactor	bool	True	True: compute a load factor $\in [0, 1]$ from static step time; all loads are scaled by the load factor; False: loads are always scaled with 1 – use this option if time dependent loads use a userFunction

verboseMode	UInt	1	0 ... no output, 1 ... show errors and load steps, 2 ... show short Newton step information (error), 3 ... show also solution vector, 4 ... show also jacobian, 5 ... show also Jacobian inverse
verboseModeFile	UInt	0	same behaviour as verboseMode, but outputs all solver information to file

### 8.1.9 LinearSolverSettings

Settings for linear solver, both dense and sparse (Eigen).

LinearSolverSettings has the following items:

Name	type/function return type	size	default value / function args	description
ignoreRedundantConstraints	bool		False	[ONLY implemented for dense matrices] False: standard way, fails if redundant equations or singular matrices occur; True: if redundant constraints appear, the solver tries to resolve them by setting according Lagrange multipliers to zero; in case of redundant constraints, this may help, but it may lead to erroneous behaviour
ignoreSingularJacobian	bool		False	[ONLY implemented for dense matrices] False: standard way, fails if jacobian is singular; True: if singularities appear in jacobian (e.g. no equation attributed to a node, redundant equations, zero mass matrix, zero eigenvalue for static problem, etc.), the jacobian inverse is resolved such that according solution variables are set to zero; this may help, but it MAY LEAD TO ERRONEOUS BEHAVIOUR; for static problems, this may suppress static motion or resolve problems in case of instabilities, but should in general be considered with care!
pivotTreshold	PReal		0	threshold for dense linear solver, can be used to detect close to singular solutions, setting this to, e.g., 1e-12; solver then reports on equations that are causing close to singularity

showCausingItems	bool	True	False: no output, if solver fails; True: if redundant equations appear, they are resolved such that according solution variables are set to zero; in case of redundant constraints, this may help, but it may lead to erroneous behaviour; for static problems, this may suppress static motion or resolve problems in case of instabilities, but should in general be considered with care!
------------------	------	------	--

### 8.1.10 Parallel

Settings for linear solver, both dense and sparse (Eigen).

Parallel has the following items:

Name	type/function return type	size	default value / function args	description
numberOfThreads	PInt		1	number of threads used for parallel computation (1 == scalar processing); do not use more threads than available threads (in most cases it is good to restrict to the number of cores)
parallelizeJacobians	bool		True	(not available yet) compute jacobians multi-threaded (only makes sense for medium to large systems!)
parallelizeMassMatrix	bool		True	(not available yet) compute mass matrix multi-threaded (only makes sense for medium to large systems!)
parallelizeResiduals	bool		True	(not available yet) compute RHS vectors of objects and connectors and loads multi-threaded (only makes sense for medium to large systems!)
stopThreadsInSerialSections	bool		True	(not available yet) for large scale problems, if steps take longer than 5 ms, parallel threads are stopped in serial regions (etc. for solver) to improve overall performance
useMTforSolver	bool		True	(not available yet) use multi-threaded optimized vector, vector-matrix and matrix-matrix operations for solver and system operations (only makes sense for medium to large systems!)
useSIMDforSolver	bool		True	(not available yet) use AVX optimized vector, vector-matrix and matrix-matrix operations for solver and system operations (may already speedup for 16 coordinates)

### 8.1.11 SimulationSettings

General Settings for simulation; according settings for solution and solvers are given in subitems of this structure.

SimulationSettings has the following items:

Name	type/function re- turn type	size	default value / func- tion args	description
linearSolverSettings	LinearSolverSettings			linear solver parameters (used for dense and sparse solvers)
parallel	Parallel			parameters for vectorized and parallelized (multi-threaded) computations
solutionSettings	SolutionSettings			settings for solution files
staticSolver	StaticSolverSettings			static solver parameters
timeIntegration	TimeIntegrationSettings			time integration parameters
cleanUpMemory	bool		False	True: solvers will free memory at exit (recommended for large systems); False: keep allocated memory for repeated computations to increase performance
displayComputationTime	bool		False	display computation time statistics at end of solving
displayGlobalTimers	bool		True	display global timer statistics at end of solving (e.g., for contact, but also for internal timings during development)
displayStatistics	bool		False	display general computation information at end of time step (steps, iterations, function calls, step rejections, ...)
linearSolverType	LinearSolverType		LinearSolverType::EXUdense	selection of numerical linear solver: exu.LinearSolverType.EXUDense (dense matrix inverse), exu.LinearSolverType.EigenSparse (sparse matrix LU-factorization), ... (enumeration type)
outputPrecision	UInt		6	precision for floating point numbers written to console; e.g. values written by solver
pauseAfterEachStep	bool		False	pause after every time step or static load step(user press SPACE)

## 8.2 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.

### 8.2.1 VSettingsGeneral

General settings for visualization.

VSettingsGeneral has the following items:

Name	type/function return type	size	default value / function args	description
autoFitScene	bool		True	automatically fit scene within first second after StartRenderer()
axesTiling	PInt		12	global number of segments for drawing axes cylinders and cones (reduce this number, e.g. to 4, if many axes are drawn)
backgroundColor	Float4	4	[1.0,1.0,1.0,1.0]	red, green, blue and alpha values for background color of render window (white=[1,1,1,1]; black = [0,0,0,1])
backgroundColorBottom	Float4	4	[0.8,0.8,1.0,1.0]	red, green, blue and alpha values for bottom background color in case that useGradientBackground = True
circleTiling	PInt		16	global number of segments for circles; if smaller than 2, 2 segments are used (flat)
coordinateSystemSize	float		5.	size of coordinate system relative to font size
cylinderTiling	PInt		16	global number of segments for cylinders; if smaller than 2, 2 segments are used (flat)
drawCoordinateSystem	bool		True	false = no coordinate system shown
drawWorldBasis	bool		False	true = draw world basis coordinate system at (0,0,0)
graphicsUpdateInterval	float		0.1	interval of graphics update during simulation in seconds; 0.1 = 10 frames per second; low numbers might slow down computation speed
minSceneSize	float		0.1	minimum scene size for initial scene size and for autoFitScene, to avoid division by zero; SET GREATER THAN ZERO
pointSize	float		0.01	global point size (absolute)
rendererPrecision	PInt		4	precision of general floating point numbers shown in render window: total number of digits used (max. 16)
renderWindowString	String		"	string shown in render window (use this, e.g., for debugging, etc.; written below EXUDYN, similar to solutionInformation in SimulationSettings.solutionSettings)
showComputationInfo	bool		True	true = show (hide) all computation information including Exudyn and version
showHelpOnStartup	PInt		5	seconds to show help message on startup (0=deactivate)
showSolutionInformation	bool		True	true = show solution information (from simulationSettings.solution)
showSolverInformation	bool		True	true = solver name and further information shown in render window
showSolverTime	bool		True	true = solver current time shown in render window

sphereTiling	PInt	6	global number of segments for spheres; if smaller than 2, 2 segments are used (flat)
textColor	Float4	4 [0.,0.,0.,1.0]	general text color (default); used for system texts in render window
textSize	float	12.	general text size (font size) in pixels if not overwritten; if useWindowsMonitorScaleFactor=True, the the textSize is multiplied with the windows monitor scaling factor for larger texts on on high resolution monitors; for bitmap fonts, the maximum size of any font (standard/large/huge) is limited to 256 (which is not recommended, especially if you do not have a powerful graphics card)
threadSafeGraphicsUpdate	bool	True	true = updating of visualization is thread-safe, but slower for complicated models; deactivate this to speed up computation, but activate for generation of animations; may be improved in future by adding a safe visualizationUpdate state
useBitmapText	bool	True	if true, texts are displayed using pre-defined bitmaps for the text; may increase the complexity of your scene, e.g., if many (>10000) node numbers shown
useGradientBackground	bool	False	true = use vertical gradient for background;
useMultiThreadedRendering	bool	True	true = rendering is done in separate thread; false = no separate thread, which may be more stable but has lagging interaction for large models (do not interact with models during simulation); set this parameter before call to exudyn.StartRenderer(); MAC OS: uses always false, because MAC OS does not support multi threaded GLFW
useWindowsMonitorScaleFactor	bool	True	the windows monitor scaling is used for increased visibility of texts on high resolution monitors; based on GLFW glfwGetWindowContentScale
worldBasisSize	float	1.0	size of world basis coordinate system

## 8.2.2 VSettingsContour

Settings for contour plots; use these options to visualize field data, such as displacements, stresses, strains, etc. for bodies, nodes and finite elements.

VSettingsContour has the following items:

Name	type/function return type	size	default value / function args	description
------	---------------------------	------	-------------------------------	-------------

automaticRange	bool		True	if true, the contour plot value range is chosen automatically to the maximum range
colorBarPrecision	PInt		4	precision of floating point values shown in color bar; total number of digits used (max. 16)
colorBarTiling	PInt	1	12	number of tiles (segements) shown in the colorbar for the contour plot
maxValue	float	1	1	maximum value for contour plot; set manually, if automaticRange == False
minValue	float	1	0	minimum value for contour plot; set manually, if automaticRange == False
outputVariable	OutputVariableType		OutputVariableType:_None	selected contour plot output variable type; select OutputVariableType._None to deactivate contour plotting.
outputVariableComponent	Int	1	0	select the component of the chosen output variable; e.g., for displacements, 3 components are available: 0 == x, 1 == y, 2 == z component; for stresses, 6 components are available, see OutputVariableType description; to draw the norm of a outputVariable, set component to -1; if a certain component is not available by certain objects or nodes, no value is drawn (using default color)
reduceRange	bool		True	if true, the contour plot value range is also reduced; better for static computation; in dynamic computation set this option to false, it can reduce visualization artifacts; you should also set minVal to max(float) and maxVal to min(float)
showColorBar	bool		True	show the colour bar with minimum and maximum values for the contour plot

### 8.2.3 VSettingsNodes

Visualization settings for nodes.

VSettingsNodes has the following items:

Name	type/function return type	size	default value / function args	description
basisSize	float		0.2	size of basis for nodes
defaultColor	Float4	4	[0.2,0.2,1,1.]	default cRGB color for nodes; 4th value is alpha-transparency
defaultSize	float		-1.	global node size; if -1.f, node size is relative to openGL.initialMaxSceneSize

drawNodesAsPoint	bool	True	simplified/faster drawing of nodes; uses general->pointSize as drawing size; if drawNodesAsPoint==True, the basis of the node will be drawn with lines
show	bool	True	flag to decide, whether the nodes are shown
showBasis	bool	False	show basis (three axes) of coordinate system in 3D nodes
showNodalSlopes	UInt	False	draw nodal slope vectors, e.g. in ANCF beam finite elements
showNumbers	bool	False	flag to decide, whether the node number is shown
tiling	PInt	4	tiling for node if drawn as sphere; used to lower the amount of triangles to draw each node; if drawn as circle, this value is multiplied with 4

## 8.2.4 VSettingsBeams

Visualization settings for beam finite elements.

VSettingsBeams has the following items:

Name	type/function return type	size	default value / function args	description
axialTiling	PInt	8		number of segments to discretise the beams axis
crossSectionTiling	PInt	4		number of quads drawn over height of beam, if drawn as flat objects; leads to higher accuracy of components drawn over beam height or width, but also to larger CPU costs for drawing
drawVertical	bool		False	draw contour plot outputVariables 'vertical' along beam height; contour.outputVariable must be set accordingly
drawVerticalColor	Float4	4	[0.2,0.2,0.2,1.]	color for outputVariable to be drawn along cross section (vertically)
drawVerticalFactor	float		1.	factor for outputVariable to be drawn along cross section (vertically)
drawVerticalLines	bool		True	draw additional vertical lines for better visibility
drawVerticalOffset	float		0.	offset for vertical drawn lines; offset is added before multiplication with drawVerticalFactor
drawVerticalValues	bool		False	show values at vertical lines; note that these numbers are interpolated values and may be different from values evaluated directly at this point!

reducedAxialInterpolation	bool		True	if True, the interpolation along the beam axis may be lower than the beam element order; this may be, however, show more consistent values than a full interpolation, e.g. for strains or forces
---------------------------	------	--	------	--

## 8.2.5 VSettingsBodies

Visualization settings for bodies.

VSettingsBodies has the following items:

Name	type/function return type	size	default value / function args	description
beams	VSettingsBeams			visualization settings for beams (e.g. ANCFCable or other beam elements)
defaultColor	Float4	4	[0.3,0.3,1,1.]	default cRGB color for bodies; 4th value is
defaultSize	Float3	3	[1.,1.,1.]	global body size of xyz-cube
deformationScaleFactor	float		1	global deformation scale factor; also applies to nodes, if drawn; used for scaled drawing of (linear) finite elements, beams, etc.
show	bool		True	flag to decide, whether the bodies are shown
showNumbers	bool		False	flag to decide, whether the body(=object) number is shown

## 8.2.6 VSettingsConnectors

Visualization settings for connectors.

VSettingsConnectors has the following items:

Name	type/function return type	size	default value / function args	description
contactPointsDefaultSize	float		0.02	DEPRECATED: do not use! global contact points size; if -1.f, connector size is relative to maxSceneSize
defaultColor	Float4	4	[0.2,0.2,1,1.]	default cRGB color for connectors; 4th value is alpha-transparency
defaultSize	float		0.1	global connector size; if -1.f, connector size is relative to maxSceneSize
jointAxesLength	float		0.2	global joint axes length
jointAxesRadius	float		0.02	global joint axes radius

show	bool	True	flag to decide, whether the connectors are shown
showContact	bool	False	flag to decide, whether contact points, lines, etc. are shown
showJointAxes	bool	False	flag to decide, whether contact joint axes of 3D joints are shown
showNumbers	bool	False	flag to decide, whether the connector(=object) number is shown
springNumberOfWindings	PInt	8	number of windings for springs drawn as helical spring

## 8.2.7 VSettingsMarkers

Visualization settings for markers.

VSettingsMarkers has the following items:

Name	type/function re- turn type	size	default value / func- tion args	description
defaultColor	Float4	4	[0.1,0.5,0.1,1.]	default cRGB color for markers; 4th value is alpha-transparency
defaultSize	float		-1.	global marker size; if -1.f, marker size is relative to maxSceneSize
drawSimplified	bool		True	draw markers with simplified symbols
show	bool		True	flag to decide, whether the markers are shown
showNumbers	bool		False	flag to decide, whether the marker numbers are shown

## 8.2.8 VSettingsLoads

Visualization settings for loads.

VSettingsLoads has the following items:

Name	type/function re- turn type	size	default value / func- tion args	description
defaultColor	Float4	4	[0.7,0.1,0.1,1.]	default cRGB color for loads; 4th value is alpha-transparency
defaultRadius	float		0.005	global radius of load axis if drawn in 3D
defaultSize	float		0.2	global load size; if -1.f, load size is relative to maxSceneSize
drawSimplified	bool		True	draw markers with simplified symbols

fixedLoadSize	bool	True	if true, the load is drawn with a fixed vector length in direction of the load vector, independently of the load size
loadSizeFactor	float	0.1	if fixedLoadSize=false, then this scaling factor is used to draw the load vector
show	bool	True	flag to decide, whether the loads are shown
showNumbers	bool	False	flag to decide, whether the load numbers are shown

## 8.2.9 VSettingsSensors

Visualization settings for sensors.

VSettingsSensors has the following items:

Name	type/function re-turn type	size	default value / func-tion args	description
defaultColor	Float4	4	[0.6,0.6,0.1,1.]	default cRGB color for sensors; 4th value is alpha-transparency
defaultSize	float		-1.	global sensor size; if -1.f, sensor size is relative to maxSceneSize
drawSimplified	bool		True	draw sensors with simplified symbols
show	bool		True	flag to decide, whether the sensors are shown
showNumbers	bool		False	flag to decide, whether the sensor numbers are shown

## 8.2.10 VSettingsContact

Global visualization settings for GeneralContact. This allows to easily switch on/off during visualization.

VSettingsContact has the following items:

Name	type/function re-turn type	size	default value / func-tion args	description
colorBoundingBoxes	Float4	4	[0.9,0.1,0.1,1.]	cRGB color
colorSearchTree	Float4	4	[0.1,0.1,0.9,1.]	cRGB color
contactForcesFactor	float		0.001	factor used for scaling of contact forces is showContactForces=True
contactPointsDefaultSize	float		0.001	global contact points size; if -1.f, connector size is relative to maxSceneSize; used for some contacts, e.g., in ContactFrictionCircle
showBoundingBoxes	bool		False	show bounding boxes of all GeneralContacts

showContactForces	bool	False	if True, contact forces are drawn for certain contact models
showContactForcesValues	bool	False	if True and showContactForces=True, numerical values for contact forces are shown at certain points
showSearchTree	bool	False	show search tree of all GeneralContacts
showSearchTreeCells	bool	False	show cells inside search tree

## 8.2.11 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 re- turn type	size	default value / func- tion args	description
alwaysOnTop	bool		False	True: OpenGL render window will be always on top of all other windows
ignoreKeys	bool		False	True: ignore keyboard input except escape and 'F2' keys; used for interactive mode, e.g., to perform kinematic analysis; This flag can be switched with key 'F2'
keyPressUserFunction	KeyPressUserFunction	0		add a Python function f(key, action, mods) here, which is called every time a key is pressed; function shall return true, if key has been processed; Example: <pre>def f(key, action, mods):     print('key=',key);     use chr(key) to convert key codes [32 ...96]     to ascii; special key codes (&gt;256) are pro-     vided in the exudyn.KeyCode enumeration     type; key action needs to be checked (0=re-     leased, 1=pressed, 2=repeated); mods pro-     vide information (binary) for SHIFT (1),     CTRL (2), ALT (4), Super keys (8), CAP-     SLOCK (16)</pre>
maximize	bool		False	True: OpenGL render window will be maximized at startup
renderWindowSize	Index2	2	[1024,768]	initial size of OpenGL render window in pixel
ResetKeyPressUserFunction()	void			set keyPressUserFunction to zero (no function); because this cannot be assign to the variable itself
showMouseCoordinates	bool		False	True: show OpenGL coordinates and distance to last left mouse button pressed position; switched on/off with key 'F3'

showWindow	bool		True	True: OpenGL render window is shown on startup; False: window will be iconified at startup (e.g. if you are starting multiple computations automatically)
startupTimeout	PInt		2500	OpenGL render window startup timeout in ms (change might be necessary if CPU is very slow)

## 8.2.12 VSettingsOpenGL

OpenGL settings for 2D and 2D rendering. For further details, see the OpenGL functionality.

VSettingsOpenGL has the following items:

Name	type/function return type	size	default value / function args	description
drawFaceNormals	bool	1	False	draws triangle normals, e.g. at center of triangles; used for debugging of faces
drawNormalsLength	float	1	0.1	length of normals; used for debugging
drawVertexNormals	bool	1	False	draws vertex normals; used for debugging
enableLight0	bool	1	True	turn on/off light0
enableLight1	bool	1	True	turn on/off light1
enableLighting	bool	1	True	generally enable lighting (otherwise, colors of objects are used); OpenGL: glEnable(GL_LIGHTING)
facesTransparent	bool	1	False	True: show faces transparent independent of transparency (A)-value in color of objects; allow to show otherwise hidden node/marker/object numbers
initialCenterPoint	Float3	3	[0.,0.,0.]	centerpoint of scene (3D) at renderer startup; overwritten if autoFitScene = True
initialMaxSceneSize	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]]
initialZoom	float		1.	initial zoom of scene; overwritten/ignored if autoFitScene = True
light0ambient	float	1	0.3	ambient value of GL_LIGHT0

light0constantAttenuation	float	1	1.0	constant attenuation coefficient of GL_LIGHT0, this is a constant factor that attenuates the light source; attenuation factor = $1/(k_x + k_l \cdot d + k_q \cdot d^2)$ ; $(k_c, k_l, k_q) = (1, 0, 0)$ means no attenuation; only used for lights, where last component of light position is 1
light0diffuse	float	1	0.6	diffuse value of GL_LIGHT0
light0linearAttenuation	float	1	0.0	linear attenuation coefficient of GL_LIGHT0, this is a linear factor for attenuation of the light source with distance
light0position	Float4	4	[0.2,0.2,10.,0.]	4f position vector of GL_LIGHT0; 4th value should be 0 for lights like sun, but 1 for directional lights (and for attenuation factor being calculated); see opengl manuals
light0quadraticAttenuation	float	1	0.0	quadratic attenuation coefficient of GL_LIGHT0, this is a quadratic factor for attenuation of the light source with distance
light0specular	float	1	0.5	specular value of GL_LIGHT0
light1ambient	float	1	0.0	ambient value of GL_LIGHT1
light1constantAttenuation	float	1	1.0	constant attenuation coefficient of GL_LIGHT1, this is a constant factor that attenuates the light source; attenuation factor = $1/(k_x + k_l \cdot d + k_q \cdot d^2)$ ; only used for lights, where last component of light position is 1
light1diffuse	float	1	0.5	diffuse value of GL_LIGHT1
light1linearAttenuation	float	1	0.0	linear attenuation coefficient of GL_LIGHT1, this is a linear factor for attenuation of the light source with distance
light1position	Float4	4	[1.,1.,-10.,0.]	4f position vector of GL_LIGHT0; 4th value should be 0 for lights like sun, but 1 for directional lights (and for attenuation factor being calculated); see opengl manuals
light1quadraticAttenuation	float	1	0.0	quadratic attenuation coefficient of GL_LIGHT1, this is a quadratic factor for attenuation of the light source with distance
light1specular	float	1	0.6	specular value of GL_LIGHT1
lightModelAmbient	Float4	4	[0.,0.,0.,1.]	global ambient light; maps to OpenGL glLightModel(GL_LIGHT_MODEL_AMBIENT,[r,g,b,a])

lightModelLocalViewer	bool	1	False	select local viewer for light; maps to OpenGL glLightModel(GL_LIGHT_MODEL_LOCAL_VIEWER,...)
lightModelTwoSide	bool	1	True	enlighten also backside of object; maps to OpenGL glLightModel(GL_LIGHT_MODEL_TWO_SIDE,...)
lineSmooth	bool	1	True	draw lines smooth
lineWidth	float	1	1.	width of lines used for representation of lines, circles, points, etc.
materialAmbientAndDiffuse	Float4	4	[0.6,0.6,0.6,1.]	4f ambient color of material
materialShininess	float	1	32.	shininess of material
materialSpecular	Float4	4	[0.6,0.6,0.6,1.]	4f specular color of material
multiSampling	PInt	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!
shadeModelSmooth	bool	1	True	True: turn on smoothing for shaders, which uses vertex normals to smooth surfaces
showFaceEdges	bool	1	False	show edges of faces; using the options showFaces=false and showFaceEdges=true gives are wire frame representation
showFaces	bool	1	True	show faces of triangles, etc.; using the options showFaces=false and showFaceEdges=true gives are wire frame representation
textLineSmooth	bool	1	False	draw lines for representation of text smooth
textLineWidth	float	1	1.	width of lines used for representation of text

### 8.2.13 VSettingsExportImages

Functionality to export images to files (PNG or TGA format) which can be used to create animations; to activate image recording during the solution process, set SolutionSettings.recordImagesInterval accordingly.  
VSettingsExportImages has the following items:

Name	type/function return type	size	default value / function args	description
heightAlignment	PInt	2		alignment of exported image height; using a value of 2 helps to reduce problems with video conversion (additional horizontal lines are lost)

saveImageFileCounter	UInt	0	current value of the counter which is used to consecutively save frames (images) with consecutive numbers
saveImageFileName	FileName	'images/frame'	filename (without extension!) and (relative) path for image file(s) with consecutive numbering (e.g., frame0000.png, frame0001.png,...); ; directory will be created if it does not exist
saveImageFormat	String	'PNG'	format for exporting figures: currently only PNG and TGA available; PNG is not available for Ubuntu18.04 (check use TGA has highest compatibility with all platforms)
saveImageSingleFile	bool	False	True: only save single files with given filename, not adding numbering; False: add numbering to files, see saveImageFileName
saveImageTimeOut	PInt	5000	timeout in milliseconds for saving a frame as image to disk; this is the amount of time waited for redrawing; increase for very complex scenes
widthAlignment	PInt	4	alignment of exported image width; using a value of 4 helps to reduce problems with video conversion (additional vertical lines are lost)

## 8.2.14 VSettingsInteractive

Functionality to interact with render window; will include left and right mouse press actions and others in future.

VSettingsInteractive has the following items:

Name	type/function return type	size	default value / function args	description
highlightColor	Float4	4	[0.8,0.05,0.05,0.75]	cRGB color for highlighted item; 4th value is alpha-transparency
highlightItemIndex	Int		-1	index of item that shall be highlighted (e.g., need to find item due to errors); if set -1, no item is highlighted
highlightItemType	ItemType		ItemType::_None	item type (Node, Object, ...) that shall be highlighted (e.g., need to find item due to errors)
highlightMbsNumber	UInt		0	index of main system (mbs) for which the item shall be highlighted; number is related to the ID in SystemContainer (first mbs = 0, second = 1, ...)

highlightOtherColor	Float4	4	[0.5,0.5,0.5,0.4]	cRGB color for other items (which are not highlighted); 4th value is alpha-transparency
joystickScaleRotation	float	200.		rotation scaling factor for joystick input
joystickScaleTranslation	float	6.		translation scaling factor for joystick input
keypressRotationStep	float	5.		rotation increment per keypress in degree (full rotation = 360 degree)
keypressTranslationStep	float	0.1		translation increment per keypress relative to window size
mouseMoveRotationFactor	float	1.		rotation increment per 1 pixel mouse movement in degree
selectionLeftMouse	bool	True		True: left mouse click on items and show basic information
selectionRightMouse	bool	True		True: right mouse click on items and show dictionary (read only!)
useJoystickInput	bool	True		True: read joystick input (use 6-axis joystick with lowest ID found when starting renderer window) and interpret as (x,y,z) position and (rotx, roty, rotz) rotation: as available from 3Dconnexion space mouse and maybe others as well; set to False, if external joystick makes problems ...
zoomStepFactor	float	1.15		change of zoom per keypress (keypad +/-) or mouse wheel increment

## 8.2.15 VisualizationSettings

Settings for visualization.

VisualizationSettings has the following items:

Name	type/function return type	size	default value / function args	description
bodies	VSettingsBodies			body visualization settings
connectors	VSettingsConnectors			connector visualization settings
contact	VSettingsContact			contact visualization settings
contour	VSettingsContour			contour plot visualization settings
exportImages	VSettingsExportImages			settings for exporting (saving) images to files in order to create animations
general	VSettingsGeneral			general visualization settings
interactive	VSettingsInteractive			Settings for interaction with renderer
loads	VSettingsLoads			load visualization settings
markers	VSettingsMarkers			marker visualization settings
nodes	VSettingsNodes			node visualization settings
openGL	VSettingsOpenGL			OpenGL rendering settings
sensors	VSettingsSensors			sensor visualization settings
window	VSettingsWindow			visualization window and interaction settings

## 8.3 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.

### 8.3.1 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
AERHS	Real	0.		time for residual evaluation of algebraic equations right-hand-side
errorEstimator	Real	0.		for explicit solvers, additional evaluation
factorization	Real	0.		solve or inverse
integrationFormula	Real	0.		time spent for evaluation of integration formulas
jacobianAE	Real	0.		jacobian of algebraic equations (not counted in sum)
jacobianODE1	Real	0.		jacobian w.r.t. coordinates of <code>ODE1</code> equations (not counted in sum)
jacobianODE2	Real	0.		jacobian w.r.t. coordinates of <code>ODE2</code> equations (not counted in sum)
jacobianODE2_t	Real	0.		jacobian w.r.t. coordinates_t of <code>ODE2</code> equations (not counted in sum)
massMatrix	Real	0.		mass matrix computation
newtonIncrement	Real	0.		$\text{Jac}^{-1} * \text{RHS}$ ; backsubstitution
ODE1RHS	Real	0.		time for residual evaluation of <code>ODE1</code> right-hand-side
ODE2RHS	Real	0.		time for residual evaluation of <code>ODE2</code> right-hand-side
overhead	Real	0.		overhead, such as initialization, copying and some matrix-vector multiplication
python	Real	0.		time spent for Python functions
reactionForces	Real	0.		$CqT * \lambda$
Reset(...)	void		useSolverTimer	reset solver timings to initial state by assigning default values; <code>useSolverTimer</code> sets the <code>useTimer</code> flag
StartTimer(...)	void		value	start timer function for a given variable; subtracts current CPU time from value
StopTimer(...)	void		value	stop timer function for a given variable; adds current CPU time to value
Sum()	Real			compute sum of all timers (except for those counted multiple, e.g., jacobians)

ToString()	String		converts the current timings to a string
total	Real	0.	total time measured between start and end of computation (static/dynamics)
totalJacobian	Real	0.	time for all jacobian computations
useTimer	bool	True	flag to decide, whether the timer is used (true) or not
visualization	Real	0.	time spent for visualization in computation thread
writeSolution	Real	0.	time for writing solution

### 8.3.2 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
aAlgorithmic	ResizableVector			additional term needed for generalized alpha (current state)
CleanUpMemory()	void			if desired, temporary data is cleaned up to safe memory
GetLinearSolverType()	LinearSolverType			return current linear solver type (dense/sparse)
nAE	Index	0		number of algebraic coordinates
nData	Index	0		number of data coordinates
newtonSolution	ResizableVector			Newton decrement (computed from residual and jacobian)
nODE1	Index	0		number of first order ordinary diff. eq. coordinates
nODE2	Index	0		number of second order ordinary diff. eq. coordinates
nSys	Index	0		number of system (unknown) coordinates = nODE2+nODE1+nAE
SetLinearSolverType(...)	void		linearSolverType	set linear solver type and matrix version: links system matrices to according dense/sparse versions
startAE	Index	0		start of algebraic coordinates, but set to zero if nAE==0
startOfStepStateAAlgorithmic	ResizableVector			additional term needed for generalized alpha (startOfStep state)
systemResidual	ResizableVector			system residual vector (vectors will be linked to this vector!)
temp2ODE2	ResizableVector			second temporary vector for ODE2 quantities; use in static computation
tempODE1F0	ResizableVector			temporary vector for ODE1 Jacobian
tempODE1F1	ResizableVector			temporary vector for ODE1 Jacobian
tempODE2	ResizableVector			temporary vector for ODE2 quantities; use in initial accelerations and during Newton

tempODE2F0	ResizableVector		temporary vector for <a href="#">ODE2</a> Jacobian
tempODE2F1	ResizableVector		temporary vector for <a href="#">ODE2</a> Jacobian

### 8.3.3 SolverIterationData

Solver internal structure for counters, steps, step size, time, etc.; solution vectors, residuals, etc. are Solver-LocalData. The given default values are overwritten by the simulationSettings when initializing the solver. SolverIterationData has the following items:

Name	type/function return type	size	default value / function args	description
adaptiveStep	bool		True	True: the step size may be reduced if step fails; no automatic stepsize control
automaticStepSize	bool		True	True: if timeIntegration.automaticStepSize == True AND chosen integrators supports automatic step size control (e.g., DOPRI5); False: constant step size used (step may be reduced if adaptiveStep=True)
automaticStepSizeError	Real		0	estimated error (relative to atol + rtol*solution) of last step; must be $\leq 1$ for a step to be accepted
currentStepIndex	Index		0	current step index; $i$
currentStepSize	Real		0.	stepSize of current step
currentTime	Real		0.	holds the current simulation time, copy of state.current.time; interval is [start-Time,tEnd]; in static solver, duration is loadStepDuration
discontinuousIteration	Index		0	number of current discontinuous iteration
discontinuousIterationsCount	Index		0	count total number of discontinuous iterations (min. 1 per step)
endTime	Real		0.	end time of static/dynamic solver
initialStepSize	Real		1e-6	initial stepSize for dynamic solver; only used, if automaticStepSize is activated
lastStepSize	Real		0.	stepSize suggested from last step or by initial step size; only used, if automaticStepSize is activated
maxStepSize	Real		0.	constant or maximum stepSize
minStepSize	Real		0.	minimum stepSize for static/dynamic solver; only used, if automaticStepSize is activated
newtonJacobiCount	Index		0	count total Newton jacobian computations
newtonSteps	Index		0	number of current newton steps
newtonStepsCount	Index		0	count total Newton steps
numberOfSteps	Index		0	number of time steps (if fixed size); $n$

recommendedStepSize	Real	-1.	recommended step size $h_{recom}$ after Post-Newton(...): $h_{recom} < 0$ : no recommendation, $h_{recom} == 0$ : use minimum step size, $h_{recom} > 0$ : use specific step size, if no smaller size requested by other reason
rejectedAutomaticStepSizeSteps	Index	0	count the number of rejected steps in case of automatic step size control (rejected steps are repeated with smaller step size)
rejectedModifiedNewtonSteps	Index	0	count the number of rejected modified Newton steps (switch to full Newton)
startTime	Real	0.	time at beginning of time integration
ToString()	String		convert iteration statistics to string; used for displayStatistics option

### 8.3.4 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
contractivity	Real	0.		Newton contractivity = geometric decay of error in every step
discontinuousIterationError	Real	0.		error of discontinuous iterations (contact, friction, ...) outside of Newton iteration
discontinuousIterationSuccessful	bool	True		true, if last discontinuous iteration had success (failure may be recovered by adaptive step)
errorCoordinateFactor	Real	1.		factor may include the number of system coordinates to reduce the residual
InitializeData()	void			initialize SolverConvergenceData by assigning default values
jacobianUpdateRequested	bool	True		true, if a jacobian update is requested in modified Newton (determined in previous step)
lastResidual	Real	0.		last Newton residual to determine contractivity
linearSolverCausingRow	Index	-1		-1 if successful, 0 ... n-1, the system equation (=coordinate) index which may have caused the problem, at which the linear solver failed
linearSolverFailed	bool	False		true, if linear solver failed to factorize
massMatrixNotInvertible	bool	True		true, if mass matrix is not invertable during initialization or solution (explicit solver)
newtonConverged	bool	False		true, if Newton has (finally) converged
newtonSolutionDiverged	bool	False		true, if Newton diverged (may be recovered)

residual	Real	0.	current Newton residual
stepReductionFailed	bool	False	true, if iterations over time/static steps failed (finally, cannot be recovered)
stopNewton	bool	False	set true by Newton, if Newton was stopped, e.g., because of exceeding iterations or linear solver failed

### 8.3.5 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
cpuLastTimePrinted	Real	0.		CPU time when output has been printed last time
cpuStartTime	Real	0.		CPU start time of computation (starts counting at computation of initial conditions)
finishedSuccessfully	bool	False		flag is false until solver finished successfully (can be used as external trigger)
InitializeData()	void			initialize SolverOutputData by assigning default values
lastDiscontinuousIterationsCount	Index	0		discontinuous iterations count when written to console (or file) last time
lastImageRecorded	Real	0.		simulation time when last image has been recorded
lastNewtonJacobiCount	Index	0		jacobian update count when written to console (or file) last time
lastNewtonStepsCount	Index	0		newton steps count when written to console (or file) last time
lastSensorsWritten	Real	0.		simulation time when last sensors have been written
lastSolutionWritten	Real	0.		simulation time when last solution has been written
lastVerboseStepIndex	Index	0		step index when last time written to console (or file)
sensorValuesTemp	ResizableVector			temporary vector for per sensor values (overwritten for every sensor; usually contains last sensor values)
sensorValuesTemp2	ResizableVector			additional temporary vector for per sensor values (overwritten for every sensor; usually contains time+last sensor values)
stepInformation	Index	0		this is a copy of the solvers stepInformation used for console output
verboseMode	Index	0		this is a copy of the solvers verboseMode used for console output

verboseModeFile	Index	0	this is a copy of the solvers verboseModeFile used for file
writeToSolutionFile	bool	False	if false, no solution file is generated and no file is written
writeToSolverFile	bool	False	if false, no solver output file is generated and no file is written

### 8.3.6 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
conv	SolverConvergenceData			all information about tolerances, errors and residuals
it	SolverIterationData			all information about iterations (steps, discontinuous iteration, newton,...)
newton	NewtonSettings			copy of newton settings from timeint or staticSolver
output	SolverOutputData			output modes and timers for exporting solver information and solution
timer	CSolverTimer			timer which measures the CPU time of solver sub functions
CheckInitialized(...)	bool		mainSystem	check if MainSolver and MainSystem are correctly initialized ==> otherwise raise SysError
ComputeAlgebraicEquations(...)	void		mainSystem, velocityLevel=false	compute the algebraic equations in systemResidual in range(nODE2+nODE1, nODE2+nODE1+nAE)
ComputeJacobianAE(...)	void		mainSystem, scalarFactor_ODE2=1., scalarFactor_ODE2_t=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
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
ComputeLoadFactor(...)	Real	simulationSettings	for static solver, this is a factor in interval [0,1]; MUST be overwritten
ComputeMassMatrix(...)	void	mainSystem, scalarFactor=1.	compute systemMassMatrix (multiplied with factor) in cSolver and return mass matrix
ComputeNewtonJacobian(...)	void	mainSystem, simulationSettings	compute jacobian for newton method of given solver method; store result in systemJacobian
ComputeNewtonResidual(...)	Real	mainSystem, simulationSettings	compute residual for Newton method (e.g. static or time step); store residual vector in systemResidual and return scalar residual (specific computation may depend on solver types)
ComputeNewtonUpdate(...)	void	mainSystem, simulationSettings, initial=true	compute update for currentState from newtonSolution (decrement from residual and jacobian); if initial, this is for the initial update with newtonSolution=0
ComputeODE2RHS(...)	void	mainSystem	compute the RHS of <a href="#">ODE2</a> equations in systemResidual in range(0,nODE2)
DiscontinuousIteration(...)	bool	mainSystem, simulationSettings	perform discontinuousIteration for static step / time step; CALLS ComputeNewtonResidual
FinalizeSolver(...)	void	mainSystem, simulationSettings	write concluding information (timer statistics, messages) and close files
FinishStep(...)	void	mainSystem, simulationSettings	finish static step / time step; write output of results to file
GetAEmax()	Index		number of algebraic equations in solver
GetDataSize()	Index		number of data (history) variables in solver
GetNewtonSolution()	NumpyVector		get locally stored / last computed solution (=increment) of Newton
GetODE1size()	Index		number of <a href="#">ODE1</a> equations in solver (not yet implemented)
GetODE2size()	Index		number of <a href="#">ODE2</a> equations in solver
GetSimulationEndTime(...)	Real	simulationSettings	compute simulation end time (depends on static or time integration solver)
GetSolverName()	std::string		get solver name - needed for output file header and visualization window
GetSystemJacobian()	NumpyMatrix		get locally stored / last computed system jacobian of solver
GetSystemMassMatrix()	NumpyMatrix		get locally stored / last computed mass matrix of solver
GetSystemResidual()	NumpyVector		get locally stored / last computed system residual
HasAutomaticStepSizeControl(...)	bool	mainSystem, simulationSettings	return true, if solver supports automatic stepsize control, otherwise false
IncreaseStepSize(...)	void	mainSystem, simulationSettings	increase step size if convergence is good

InitializeSolver(...)	bool	mainSystem, simulationSettings	initialize solverSpecific,data,it,conv; set/compute initial conditions (solver-specific!); initialize output files
InitializeSolverData(...)	void	mainSystem, simulationSettings	initialize all data,it,conv; called from InitializeSolver()
InitializeSolverInitialConditions(...)	void	mainSystem, simulationSettings	set/compute initial conditions (solver-specific!); called from InitializeSolver()
InitializeSolverOutput(...)	void	mainSystem, simulationSettings	initialize output files; called from InitializeSolver()
InitializeSolverPreChecks(...)	bool	mainSystem, simulationSettings	check if system is solvable; initialize dense/sparse computation modes
InitializeStep(...)	void	mainSystem, simulationSettings	initialize static step / time step; Python-functions; do some outputs, checks, etc.
IsStaticSolver()	bool		return true, if static solver; needs to be overwritten in derived class
IsVerboseCheck(...)	bool	level	return true, if file or console output is at or above the given level
loadStepGeometricFactor	Real		multiplicative load step factor; this factor is computed from loadStepGeometric parameters in SolveSystem(...)
Newton(...)	bool	mainSystem, simulationSettings	perform Newton method for given solver method
PostInitializeSolverSpecific(...)	void	mainSystem, simulationSettings	post-initialize for solver specific tasks; called at the end of InitializeSolver
PreInitializeSolverSpecific(...)	void	mainSystem, simulationSettings	pre-initialize for solver specific tasks; called at beginning of InitializeSolver, right after Solver data reset
ReduceStepSize(...)	bool	mainSystem, simulationSettings, severity	reduce step size (1..normal, 2..severe problems); return true, if reduction was successful
SetSystemJacobian(...)	void	systemJacobian	set locally stored system jacobian of solver; must have size nODE2+nODE1+nAE
SetSystemMassMatrix(...)	void	systemMassMatrix	set locally stored mass matrix of solver; must have size nODE2+nODE1+nAE
SetSystemResidual(...)	void	systemResidual	set locally stored system residual; must have size nODE2+nODE1+nAE
SolveSteps(...)	bool	mainSystem, simulationSettings	main solver part: calls multiple InitializeStep(...)/DiscontinuousIteration(...)/FinishStep(...); do step reduction if necessary; return true if success, false else
SolveSystem(...)	bool	mainSystem, simulationSettings	solve System: InitializeSolver, SolveSteps, FinalizeSolver
UpdateCurrentTime(...)	void	mainSystem, simulationSettings	update currentTime (and load factor); MUST be overwritten in special solver class
VerboseWrite(...)	void	level, str	write to console and/or file in case of level
WriteCoordinatesToFile(...)	void	mainSystem, simulationSettings	write unique coordinates solution file
WriteSolutionFileHeader(...)	void	mainSystem, simulationSettings	write unique file header, depending on static/ dynamic simulation

### 8.3.7 MainSolverImplicitSecondOrder

PyBind interface (trampoline) class for dynamic implicit solver. Note that this solver includes the classical Newmark method (set useNewmark True; with option of index 2 reduction) as well as the generalized-alpha method. With the interface, the dynamic implicit solver and its substructures can be accessed via Python. NOTE that except from SolveSystem(...), these functions are only intended for experienced users and they need to be handled with care, as unexpected crashes may happen if used inappropriate. Furthermore, the functions have a lot of overhead (still fast, but performance much lower than internal solver) due to Python interfaces, and should thus be used for small systems. To access the solver in Python, write

```
solver = MainSolverImplicitSecondOrder()
```

and hereafter you can access all data and functions via 'solver'. In this solver, user functions are possible to extend the solver at certain parts, while keeping the overall C++ performance. User functions, which are added with SetUserFunction...(...), have the arguments (MainSolver, MainSystem, simulationSettings), except for ComputeNewtonUpdate which adds the initial flag as an additional argument and ComputeNewtonResidual, which returns the scalar residual.

MainSolverImplicitSecondOrder has the following items:

Name	type/function return type	size	default value / function args	description
conv	SolverConvergenceData			all information about tolerances, errors and residuals
it	SolverIterationData			all information about iterations (steps, discontinuous iteration, newton,...)
newton	NewtonSettings			copy of newton settings from timeint or staticSolver
output	SolverOutputData			output modes and timers for exporting solver information and solution
timer	CSolverTimer			timer which measures the CPU time of solver sub functions; note that solver structures can only be written indirectly, e.g., timer=dynamicSolver.timer; timer.useTimer = False; dynamicSolver.timer=timer; however, dynamicSolver.timer.useTimer cannot be written.
alphaF	Real			copy of parameter in timeIntegration.generalizedAlpha
alphaM	Real			copy of parameter in timeIntegration.generalizedAlpha
CheckInitialized(...)	bool		mainSystem	check if MainSolver and MainSystem are correctly initialized ==> otherwise raise SysError
ComputeAlgebraicEquations(...)	void		mainSystem, velocityLevel=false	compute the algebraic equations in systemResidual in range(nODE2+nODE1, nODE2+nODE1+nAE)

ComputeJacobianAE(...)	void	mainSystem, scalarFactor_ODE2=1., scalarFactor_ODE2_t=1., velocityLevel=false	add jacobian of algebraic equations (multiplied with factor) to systemJacobian in cSolver; the scalarFactors are scaling the derivatives w.r.t. <b>ODE2</b> coordinates and w.r.t. ODE2_t (velocity) coordinates; if velocityLevel == true, the constraints are evaluated at velocity level
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
ComputeLoadFactor(...)	Real	simulationSettings	for static solver, this is a factor in interval [0,1]; MUST be overwritten
ComputeMassMatrix(...)	void	mainSystem, scalarFactor=1.	compute systemMassMatrix (multiplied with factor) in cSolver and return mass matrix
ComputeNewtonJacobian(...)	void	mainSystem, simulationSettings	compute jacobian for newton method of given solver method; store result in systemJacobian
ComputeNewtonResidual(...)	Real	mainSystem, simulationSettings	compute residual for Newton method (e.g. static or time step); store residual vector in systemResidual and return scalar residual (specific computation may depend on solver types)
ComputeNewtonUpdate(...)	void	mainSystem, simulationSettings, initial=true	compute update for currentState from newtonSolution (decrement from residual and jacobian); if initial, this is for the initial update with newtonSolution=0
ComputeODE1RHS(...)	void	mainSystem	compute the RHS of <b>ODE1</b> equations in systemResidual in range(0,nODE1)
ComputeODE2RHS(...)	void	mainSystem	compute the RHS of <b>ODE2</b> equations in systemResidual in range(0,nODE2)
DiscontinuousIteration(...)	bool	mainSystem, simulationSettings	perform discontinuousIteration for static step / time step; CALLS ComputeNewtonResidual
factJacAlgorithmic	Real		locally computed parameter from generalizedAlpha parameters
FinalizeSolver(...)	void	mainSystem, simulationSettings	write concluding information (timer statistics, messages) and close files
FinishStep(...)	void	mainSystem, simulationSettings	finish static step / time step; write output of results to file
GetAAlgorithmic()	NumpyVector		get locally stored / last computed algorithmic accelerations
GetAEsize()	Index		number of algebraic equations in solver
GetDataSize()	Index		number of data (history) variables in solver
GetNewtonSolution()	NumpyVector		get locally stored / last computed solution (=increment) of Newton
GetODE1size()	Index		number of <b>ODE1</b> equations in solver (not yet implemented)

GetODE2size()	Index		number of <a href="#">ODE2</a> equations in solver
GetSimulationEndTime(...)	Real	simulationSettings	compute simulation end time (depends on static or time integration solver)
GetSolverName()	std::string		get solver name - needed for output file header and visualization window
GetStartOfStepStateAAlgorithmic()	NumpyVector		get locally stored / last computed algorithmic accelerations at start of step
GetSystemJacobian()	NumpyMatrix		get locally stored / last computed system jacobian of solver
GetSystemMassMatrix()	NumpyMatrix		get locally stored / last computed mass matrix of solver
GetSystemResidual()	NumpyVector		get locally stored / last computed system residual
HasAutomaticStepSizeControl(...)	bool	mainSystem, simulationSettings	return true, if solver supports automatic stepsize control, otherwise false
IncreaseStepSize(...)	void	mainSystem, simulationSettings	increase step size if convergence is good
InitializeSolver(...)	bool	mainSystem, simulationSettings	initialize solverSpecific,data,it,conv; set/compute initial conditions (solver-specific!); initialize output files
InitializeSolverData(...)	void	mainSystem, simulationSettings	initialize all data,it,conv; called from InitializeSolver()
InitializeSolverInitialConditions(...)	void	mainSystem, simulationSettings	set/compute initial conditions (solver-specific!); called from InitializeSolver()
InitializeSolverOutput(...)	void	mainSystem, simulationSettings	initialize output files; called from InitializeSolver()
InitializeSolverPreChecks(...)	bool	mainSystem, simulationSettings	check if system is solvable; initialize dense/sparse computation modes
InitializeStep(...)	void	mainSystem, simulationSettings	initialize static step / time step; Python-functions; do some outputs, checks, etc.
IsStaticSolver()	bool		return true, if static solver; needs to be overwritten in derived class
IsVerboseCheck(...)	bool	level	return true, if file or console output is at or above the given level
newmarkBeta	Real		copy of parameter in timeIntegration.generalizedAlpha
newmarkGamma	Real		copy of parameter in timeIntegration.generalizedAlpha
Newton(...)	bool	mainSystem, simulationSettings	perform Newton method for given solver method
PostInitializeSolverSpecific(...)	void	mainSystem, simulationSettings	post-initialize for solver specific tasks; called at the end of InitializeSolver
PostNewton(...)	Real	mainSystem, simulationSettings	call PostNewton for all relevant objects (contact, friction, ... iterations); returns error for discontinuous iteration
PreInitializeSolverSpecific(...)	void	mainSystem, simulationSettings	pre-initialize for solver specific tasks; called at beginning of InitializeSolver, right after Solver data reset

ReduceStepSize(...)	bool	mainSystem, simulationSettings, severity	reduce step size (1..normal, 2..severe problems); return true, if reduction was successful
SetSystemJacobian(...)	void	systemJacobian	set locally stored system jacobian of solver; must have size nODE2+nODE1+nAE
SetSystemMassMatrix(...)	void	systemMassMatrix	set locally stored mass matrix of solver; must have size nODE2+nODE1+nAE
SetSystemResidual(...)	void	systemResidual	set locally stored system residual; must have size nODE2+nODE1+nAE
SetUserFunctionComputeNewtonJacobian(...) void	mainSystem, userFunction		set user function
SetUserFunctionComputeNewtonResidual(...) void	mainSystem, userFunction		set user function
SetUserFunctionComputeNewtonUpdate(...) void	mainSystem, userFunction		set user function
SetUserFunctionDiscontinuousIteration(...) void	mainSystem, userFunction		set user function
SetUserFunctionFinishStep(...) void	mainSystem, userFunction		set user function
SetUserFunctionInitializeStep(...) void	mainSystem, userFunction		set user function
SetUserFunctionNewton(...)	void	mainSystem, userFunction	set user function
SetUserFunctionPostNewton(...) void	mainSystem, userFunction		set user function
SetUserFunctionUpdateTimeCurrentTime(...) void	mainSystem, userFunction		set user function
SolveSteps(...)	bool	mainSystem, simulationSettings	main solver part: calls multiple InitializeStep(...)/ DiscontinuousIteration(...)/ FinishStep(...); do step reduction if necessary; return true if success, false else
SolveSystem(...)	bool	mainSystem, simulationSettings	solve System: InitializeSolver, SolveSteps, FinalizeSolver
spectralRadius	Real		copy of parameter in timeIntegration.generalizedAlpha
UpdatecurrentTime(...)	void	mainSystem, simulationSettings	update currentTime (and load factor); MUST be overwritten in special solver class
VerboseWrite(...)	void	level, str	write to console and/or file in case of level
WriteCoordinatesToFile(...)	void	mainSystem, simulationSettings	write unique coordinates solution file
WriteSolutionFileHeader(...)	void	mainSystem, simulationSettings	write unique file header, depending on static/ dynamic simulation

### 8.3.8 MainSolverExplicit

PyBind interface (trampoline) class for dynamic explicit solver. Note that this solver includes the 1st order explicit Euler scheme and the 4th order Runge-Kutta scheme with 5th order error estimation (DOPRI5). With the interface, the solver and its substructures can be accessed via Python. NOTE that except from SolveSystem(...),

these functions are only intended for experienced users and they need to be handled with care, as unexpected crashes may happen if used inappropriate. Furthermore, the functions have a lot of overhead (still fast, but performance much lower than internal solver) due to Python interfaces, and should thus be used for small systems. To access the solver in Python, write

```
solver = MainSolverExplicit()
```

and hereafter you can access all data and functions via 'solver'. In this solver, no user functions are possible, but you can use SolverImplicitSecondOrder instead (turning off Newton gives explicit scheme ...).

MainSolverExplicit has the following items:

Name	type/function re- turn type	size	default value / func- tion args	description
conv	SolverConvergence	Data		all information about tolerances, errors and residuals
it	SolverIterationData			all information about iterations (steps, discontinuous iteration, newton,...)
output	SolverOutputData			output modes and timers for exporting solver information and solution
timer	CSolverTimer			timer which measures the CPU time of solver sub functions
ComputeLoadFactor(...)	Real		simulationSettings	for static solver, this is a factor in interval [0,1]; MUST be overwritten
ComputeMassMatrix(...)	void		mainSystem, scalarFactor=1.	compute systemMassMatrix (multiplied with factor) in cSolver and return mass matrix
ComputeNewtonJacobian(...)	void		mainSystem, simulationSettings	compute jacobian for newton method of given solver method; store result in systemJacobian
ComputeNewtonResidual(...)	Real		mainSystem, simulationSettings	compute residual for Newton method (e.g. static or time step); store residual vector in systemResidual and return scalar residual (specific computation may depend on solver types)
ComputeNewtonUpdate(...)	void		mainSystem, simulationSettings, initial=true	compute update for currentState from newtonSolution (decrement from residual and jacobian); if initial, this is for the initial update with newtonSolution=0
ComputeODE1RHS(...)	void		mainSystem	compute the RHS of <a href="#">ODE1</a> equations in systemResidual in range(0,nODE1)
ComputeODE2RHS(...)	void		mainSystem	compute the RHS of <a href="#">ODE2</a> equations in systemResidual in range(0,nODE2)
DiscontinuousIteration(...)	bool		mainSystem, simulationSettings	perform discontinuousIteration for static step / time step; CALLS ComputeNewtonResidual
FinalizeSolver(...)	void		mainSystem, simulationSettings	write concluding information (timer statistics, messages) and close files
FinishStep(...)	void		mainSystem, simulationSettings	finish static step / time step; write output of results to file
GetAEsize()	Index			number of algebraic equations in solver
GetDataSize()	Index			number of data (history) variables in solver

GetMethodOrder()	Index		return order of method (higher value in methods with automatic step size, e.g., DOPRI5=5)
GetNumberOfStages()	Index		return number of stages in current method
GetODE1size()	Index		number of <a href="#">ODE1</a> equations in solver (not yet implemented)
GetODE2size()	Index		number of <a href="#">ODE2</a> equations in solver
GetSimulationEndTime(...)	Real	simulationSettings	compute simulation end time (depends on static or time integration solver)
GetSolverName()	std::string		get solver name - needed for output file header and visualization window
GetSystemMassMatrix()	NumpyMatrix		get locally stored / last computed mass matrix of solver
GetSystemResidual()	NumpyVector		get locally stored / last computed system residual
HasAutomaticStepSizeControl(...)	bool	mainSystem, simulationSettings	return true, if solver supports automatic stepsize control, otherwise false
IncreaseStepSize(...)	void	mainSystem, simulationSettings	increase step size if convergence is good
InitializeSolver(...)	bool	mainSystem, simulationSettings	initialize solverSpecific,data,it,conv; set/compute initial conditions (solver-specific!); initialize output files
InitializeSolverData(...)	void	mainSystem, simulationSettings	initialize all data,it,conv; called from InitializeSolver()
InitializeSolverInitialConditions(...)	void	mainSystem, simulationSettings	set/compute initial conditions (solver-specific!); called from InitializeSolver()
InitializeSolverOutput(...)	void	mainSystem, simulationSettings	initialize output files; called from InitializeSolver()
InitializeSolverPreChecks(...)	bool	mainSystem, simulationSettings	check if system is solvable; initialize dense/sparse computation modes
InitializeStep(...)	void	mainSystem, simulationSettings	initialize static step / time step; Python-functions; do some outputs, checks, etc.
IsStaticSolver()	bool		return true, if static solver; needs to be overwritten in derived class
IsVerboseCheck(...)	bool	level	return true, if file or console output is at or above the given level
Newton(...)	bool	mainSystem, simulationSettings	perform Newton method for given solver method
PostInitializeSolverSpecific(...)	void	mainSystem, simulationSettings	post-initialize for solver specific tasks; called at the end of InitializeSolver
PreInitializeSolverSpecific(...)	void	mainSystem, simulationSettings	pre-initialize for solver specific tasks; called at beginning of InitializeSolver, right after Solver data reset
ReduceStepSize(...)	bool	mainSystem, simulationSettings, severity	reduce step size (1..normal, 2..severe problems); return true, if reduction was successful
SetSystemMassMatrix(...)	void	systemMassMatrix	set locally stored mass matrix of solver; must have size nODE2+nODE1+nAE

SetSystemResidual(...)	void	systemResidual	set locally stored system residual; must have size nODE2+nODE1+nAE
SolveSteps(...)	bool	mainSystem, simulationSettings	main solver part: calls multiple InitializeStep(...)/DiscontinuousIteration(...)/FinishStep(...); do step reduction if necessary; return true if success, false else
SolveSystem(...)	bool	mainSystem, simulationSettings	solve System: InitializeSolver, SolveSteps, FinalizeSolver
UpdateCurrentTime(...)	void	mainSystem, simulationSettings	update currentTime (and load factor); MUST be overwritten in special solver class
VerboseWrite(...)	void	level, str	write to console and/or file in case of level
WriteCoordinatesToFile(...)	void	mainSystem, simulationSettings	write unique coordinates solution file
WriteSolutionFileHeader(...)	void	mainSystem, simulationSettings	write unique file header, depending on static/ dynamic simulation



# Chapter 9

## 3D graphics visualization

The 3D graphics visualization window is kept simple, but useful to see the animated results of the multibody system. The graphics output is restricted to a 3D window (renderwindow) into which the renderer draws the visualization state of the `MainSystem.mbs`.

### 9.1 Mouse input

The following table includes the mouse functions.

Button	action	remarks
<b>left mouse button</b>	move model	keep left mouse button pressed to move the model in the current x/y plane
<b>left mouse button</b>	select item	mouse click on any node, object, etc. to see its basic information in status line
<b>right mouse button</b>	rotate model	keep right mouse button pressed to rotate model around current current $X_1/X_2$ axes
<b>right mouse button</b>	show item dictionary	[EXPERIMENTAL, must be activated in <code>visualizationSettings.interactive</code> ] (short) press and release right mouse button
<b>mouse wheel</b>	zoom	use mouse wheel to zoom (on touch screens 'pinch-to-zoom' might work as well)

Note that current mouse coordinates can be obtained via `SystemContainer.GetCurrentMouseCoordinates()`.

#### 9.1.1 6D mouse

Graphics engines, especially in CAD and finite elements allow input of special 3D or 6D mouse devices. There is a basic interface for so-called 3D mouse / 6D mouse or space mouse, allowing to map the 6D joystick to translation and rotation, see `visualizationSettings.interactive.useJoystickInput` and similar options. The interface only works, if the device maps 6 coordinates to the joystick input of GLFW (tested with 3DCONNEXION mouse).

## 9.2 Keyboard input

The following table includes the keyboard shortcuts available in the window.

Key(s)	action	remarks
<b>1,2,3,4 or 5</b>	visualization update speed	the entered digit controls the visualization update, ranging within 0.02, 0.1 (default), 0.5, 2, and 100 seconds
<b>CTRL+1</b> <b>or</b> <b>SHIFT+CTRL+1</b>	change view	set view in 1/2-plane (+SHIFT: view from opposite side)
<b>CTRL+2</b> <b>or</b> <b>SHIFT+CTRL+2</b>	change view	set view in 1/3-plane (+SHIFT: view from opposite side)
<b>CTRL+3</b> <b>or</b> <b>SHIFT+CTRL+3</b>	change view	set view in 2/3-plane (+SHIFT: view from opposite side)
<b>CTRL+4</b> <b>or</b> <b>SHIFT+CTRL+4</b>	change view	set view in 2/1-plane (+SHIFT: view from opposite side)
<b>CTRL+5</b> <b>or</b> <b>SHIFT+CTRL+5</b>	change view	set view in 3/1-plane (+SHIFT: view from opposite side)
<b>CTRL+6</b> <b>or</b> <b>SHIFT+CTRL+6</b>	change view	set view in 3/2-plane (+SHIFT: view from opposite side)
<b>A</b>	zoom all	set zoom such that the whole scene is visible
<b>CURSOR UP, DOWN, ...</b>	move scene	use cursor keys to move the scene up, down, left, and right (use CTRL for small movements)
<b>C</b>	show/hide connectors	pressing this key switches the visibility of connectors
<b>CTRL+C</b>	show/hide connector numbers	pressing this key switches the visibility of connector numbers
<b>B</b>	show/hide bodies	pressing this key switches the visibility of bodies
<b>CTRL+B</b>	show/hide body numbers	pressing this key switches the visibility of body numbers
<b>L</b>	show/hide loads	pressing this key switches the visibility of loads
<b>CTRL+L</b>	show/hide load numbers	pressing this key switches the visibility of load numbers
<b>M</b>	show/hide markers	pressing this key switches the visibility of markers
<b>CTRL+M</b>	show/hide marker numbers	pressing this key switches the visibility of marker numbers
<b>N</b>	show/hide nodes	pressing this key switches the visibility of nodes
<b>CTRL+N</b>	show/hide node numbers	pressing this key switches the visibility of node numbers
<b>S</b>	show/hide sensors	pressing this key switches the visibility of sensors
<b>CTRL+S</b>	show/hide sensor numbers	pressing this key switches the visibility of sensor numbers
<b>T</b>	faces / edges mode	switch between faces transparent/ faces transparent + edges / only face edges / full faces with edges / only faces visible
<b>Q</b>	stop solver	current solver is stopped (proceeds to next simulation or end of file)
<b>X</b>	execute command	open dialog to enter a python command (in global python scope); dialog may appear behind the visualization window! User errors may lead to crash – be careful! Examples: 'print(mbs)', 'x=5', 'mbs.GetObject(0)',etc.
<b>V</b>	visualization settings	open dialog to modify visualization settings; dialog may appear behind the visualization window!
<b>F3</b>	show mouse coordinates	shown in status line
<b>ESCAPE</b>	close simulation	stops the simulation (and further simulations) and closes the render window (same as close window)

<b>SPACE</b>	continue simulation	if simulation is paused, it can be continued by pressing space; use SHIFT+SPACE to continuously activate 'continue simulation'
--------------	---------------------	--

Special keys:

<b>F2</b>	ignore keys	ignore all keyboard input, except for KeyPress user function, F2 and escape
<b>KEYPAD 2/8,4/6,1/9</b>	rotate scene	about 1, 2 and 3-axis (use CTRL for small rotations)
<b>.' or KEYPAD +</b>	zoom in	zoom one step into scene (additionally press CTRL to perform small zoom step)
<b>,' or KEYPAD -</b>	zoom out	zoom one step out of scene (additionally press CTRL to perform small zoom step)

## 9.3 GraphicsData

All graphics objects are defined by a `GraphicsData` structure. Note that currently the visualization is based on a very simple and ancient OpenGL implementation, as there is currently no simple platform independent alternative. However, most of the heavy load triangle-based operations are implemented in C++ and are realized by very efficient OpenGL commands. However, note that the number of triangles to represent the object should be kept in a feasible range (< 1000000) in order to obtain a fast response of the renderer.

Many objects include a `GraphicsData` dictionary structure for definition of attached visualization of the object. Typically, you can use primitives (cube, sphere, ...) or `STL` data to define the objects appearance. `GraphicsData` dictionaries can be created with functions provided in the utilities module `exudyn.graphicsDataUtilities`. `py` see [Section 5.5](#).

`GraphicsData` can be transformed into points and triangles (mesh) and can be used for contact computation, as well. **NOTE** that for correct rendering and correct contact computations, all triangle nodes must follow a strict local order and triangle normals – if defined – must point outwards, see Fig. 9.1.

`BodyGraphicsData` contains a list of `GraphicsData` items, i.e. `bodyGraphicsData = [graphicsItem1, graphicsItem2, ...]`. Every single `graphicsItem` may be defined as one of the following structures using a specific 'type':

Name	type	default value	description
<b>type = 'Line':</b>			<i>draws a polygonal line between all specified points</i>
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: <code>data=[0,0,0, 1,0,0, 1,1,0, 0,1,0, 0,0,0] ...</code> draws a rectangle with side length 1
<b>type = 'Circle':</b>			<i>draws a circle with center point, normal (defines plane of circle) and radius</i>
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
<b>type = 'Text':</b>			<i>places the given text at position</i>
color	list	[0,0,0,1]	list of 4 floats to define RGB-color and transparency
text	string	mandatory	text to be displayed, using UTF-8 encoding (see <a href="#">Section 9.4</a> )
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]
<b>type = 'TriangleList':</b>			<i>draws a flat triangle mesh for given points and connectivity</i>
points	list	mandatory	list [x0,y0,z0, x1,y1,z1, ...] containing $n \times 3$ floats (grouped x0,y0,z0, x1,y1,z1, ...) to define x,y,z coordinates of points, $n$ being the number of points (=vertices)
colors	list	empty	list [R0,G0,B0,A0, R1,G2,B1,A1, ...] containing $n \times 4$ floats to define RGB-color and transparency A, where $n$ must be according to number of points; if field 'colors' does not exist, default colors will be used
normals	list	empty	list [n0x,n0y,n0z, ...] containing $n \times 3$ floats to define normal direction of triangles per point, where $n$ must be according to number of points; if field 'normals' does not exist, default normals [0,0,0] will be used
triangles	list	mandatory	list [T0point0, T0point1, T0point2, ...] containing $n_{trig} \times 3$ floats to define point indices of each vertex of the triangles (=connectivity); point indices start with index 0; the maximum index must be $\leq$ points.size()

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

## 9.4 Character encoding: UTF-8

Character encoding is a major issue in computer systems, as different languages need a huge amount of different characters, see the amusing blog of Joel Spolsky:

[The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode ...](#)

More about encoding can be found in [Wikipedia:UTF-8](#). UTF-8 encoding tables can be found within the wikipedia article and a comparison with the first 256 characters of unicode is provided at [UTF-8 char table](#).

For short, EXUDYN uses UTF-8 character encoding in texts / strings drawn in OpenGL renderer window. However, the set of available UTF-8 characters in EXUDYN is restricted to a very small set of characters (as compared to available characters in UTF-8), see the following table of available characters (using hex codes, e.g. `0x20` = 32):

unicode (hex code)	UTF-8 (hex code)	character

20	20	''
21	21	'!'
...	...	...
30	30	'0'
...	...	...
39	39	'9'
40	40	'@'
41	41	'A'
...	...	...
5A	5A	'Z'
...	...	...
7E	7E	'~'
7F	7F	control, not shown
...	...	...
A0	C2 A0	no-break space
A1	C2 A1	inverted exclamation mark: '¡'
...	...	...
BF	C2 BF	inverted 'ȝ'
C0	C3 80	A with grave
...	...	...
FF	C3 BF	y with diaeresis: 'ÿ'

special characters (selected):

	E2 89 88	≈
	E2 88 82	∂
	E2 88 AB	ƒ
	E2 88 9A	√
	CE B1	α
	CE B2	β
	...	(complete list of greek letters see below)
	F0 9F 99 82	smiley
	F0 9F 98 92	frowney
	E2 88 9e	infinity: ∞

Greek characters include:  $\alpha, \beta, \gamma, \delta, \varepsilon, \zeta, \eta, \theta, \kappa, \lambda, \nu, \xi, \pi, \rho, \sigma, \varphi, \Delta, \Pi, \Sigma, \Omega$ . Note, that unicode character codes are shown only for completeness, but they **cannot be encoded by Exodyn!**

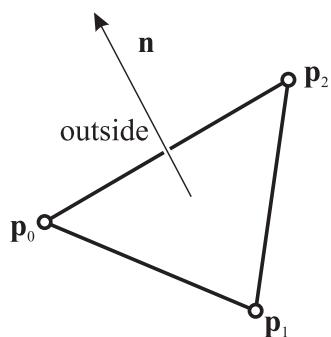


Figure 9.1: Definition of triangle normals and outside/inside regions in Exudyn: the normal to a triangle with vertex positions  $\mathbf{p}_0$ ,  $\mathbf{p}_1$ ,  $\mathbf{p}_2$  is computed from cross product as  $\mathbf{n} = \frac{(\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)}{|(\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)|}$ ; the normal  $\mathbf{n}$  then points to the outside region of the mesh or body; the direction of  $\mathbf{n}$  just depends on the ordering of the vertex points (interchange of two points changes the normal direction); correct normals are needed for contact computations as well as for correct shading effects in visualization.

# Chapter 10

## Notation and System of equations of motion

The general idea of the code is to have objects, which provide equations ([ODE2](#), [ODE1](#), [AE](#)). The solver then assembles these equations and solves the static or dynamic problem. The system structure and solver are similar but much more advanced and modular as earlier solvers by the main developer [[18](#), [14](#), [15](#)].

### 10.1 LHS–RHS naming conventions in EXUDYN

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

The objects have the following [LHS–RHS](#) conventions:

- the acceleration term, e.g.,  $m \cdot \ddot{q}$  is always positive on the [LHS](#)
- objects, connectors, etc., use [LHS](#) conventions for most terms: mass, stiffness matrix, elastic forces, damping, etc., are computed at [LHS](#) of the object equation
- object forces are written at the [RHS](#) of the object equation
- in case of constraint or connector equations, there is no [LHS](#) or [RHS](#), as there is no acceleration term. Therefore, the computation function evaluates the term as given in the description of the object, adding it to the [LHS](#).

Object equations may read, e.g., for one coordinate  $q$ , mass  $m$ , damping coefficient  $d$ , stiffness  $k$  and applied force  $f$ ,

$$\underbrace{m \cdot \ddot{q} + d \cdot \dot{q} + k \cdot q}_{\text{LHS}} = \underbrace{f}_{\text{RHS}} \quad (10.1)$$

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

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

$$\underbrace{M_{sys} \cdot \ddot{q}_{sys}}_{\text{LHS}} = \underbrace{f_{sys}}_{\text{RHS}} . \quad (10.2)$$

<sup>1</sup>except for the acceleration  $\times$  mass matrix and constraint reaction forces, see Eq. (10.13)

In the case of the object equation

$$m \cdot \ddot{q} + d \cdot \dot{q} + k \cdot q = f, \quad (10.3)$$

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

## 10.2 System assembly

Assembling equations of motion is done within the C++ class `CSystem`, see the file `CSystem.cpp`. The general idea is to assemble, i.e. to sum up, (parts of) residuals attributed by different objects. The summation process is based on coordinate indices to which the single equations belong to. Let's assume that we have two simple `ObjectMass1D` objects, with object indices  $o_0$  and  $o_1$  and having mass  $m_0$  and  $m_1$ . They are connected to nodes of type `Node1D`  $n_0$  and  $n_1$ , with global coordinate indices  $c_0$  and  $c_1$ . The partial object residuals, which are fully independent equations, read

$$m_0 \cdot \ddot{q}_{c0} = RHS_{c0}, \quad (10.4)$$

$$m_1 \cdot \ddot{q}_{c1} = RHS_{c1}, \quad (10.5)$$

where  $RHS_{c0}$  and  $RHS_{c1}$  the right-hand-side of the respective equations/coordinates. They represent forces, e.g., from `LoadCoordinate` items (which directly are applied to coordinates of nodes), say  $f_{c0}$  and  $f_{c1}$ , that are in case also summed up on the right hand side. Let us for now assume that

$$RHS_{c0} = f_{c0} \quad \text{and} \quad RHS_{c1} = f_{c1}. \quad (10.6)$$

Now we add another `ObjectMass1D` object with object index  $o_2$ , having mass  $m_2$ , but letting the object *again* use node  $n_0$  with coordinate  $c_0$ . In this case, the total object residuals read

$$(m_0 + m_2) \cdot \ddot{q}_{c0} = RHS_{c0}, \quad (10.7)$$

$$m_1 \cdot \ddot{q}_{c1} = RHS_{c1}. \quad (10.8)$$

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

Finally, if we add a `CoordinateSpringDamper`, assuming a spring  $k$  between coordinates  $c_0$  and  $c_1$ , the [RHS](#) of equations related to  $c_0$  and  $c_1$  is now augmented to

$$RHS_{c0} = f_{c0} + k \cdot (q_{c1} - q_{c0}), \quad (10.9)$$

$$RHS_{c1} = f_{c1} + k \cdot (q_{c0} - q_{c1}). \quad (10.10)$$

The system of equation would therefore read

$$(m_0 + m_2) \cdot \ddot{q}_{c0} = f_{c0} + k \cdot (q_{c1} - q_{c0}), \quad (10.11)$$

$$m_1 \cdot \ddot{q}_{c1} = f_{c1} + k \cdot (q_{c0} - q_{c1}). \quad (10.12)$$

It should be noted, that all (components of) residuals ('equations') are summed up for the according coordinates, and also all contributions to the mass matrix. Only constraint equations, which are related to Lagrange parameters always get their 'own' Lagrange multipliers, which are automatically assigned by the system and therefore independent for every constraint. Therefore, it is not possible right now to attach two `ObjectRigidBody` to the same node, as it would create two constraint equations. This will be resolved within issue #0564.

## 10.3 Nomenclature for system equations of motion and solvers

Using the basic notation for coordinates in [Section 6.1](#), we use the following quantities and symbols for equations of motion and solvers:

quantity	symbol	description
number of <b>ODE2</b> coordinates	$n$	second order ordinary differential equations ( <b>ODE2</b> )
number of <b>ODE1</b> coordinates	$n_y$	first order ordinary differential equations ( <b>ODE1</b> )
number of <b>AE</b> coordinates	$m$	algebraic equations ( <b>AE</b> )
number of system coordinates	$n_s$	system equations
<b>ODE2</b> coordinates	$\mathbf{q} = [q_0, \dots, q_{n_q}]^T$	<b>ODE2</b> , displacement-based coordinates (could also be rotation or deformation coordinates)
<b>ODE2</b> velocities	$\mathbf{v} = \dot{\mathbf{q}} = [\dot{q}_0, \dots, \dot{q}_{n_q}]^T$	<b>ODE2</b> velocity coordinates
<b>ODE2</b> accelerations	$\ddot{\mathbf{q}} = [\ddot{q}_0, \dots, \ddot{q}_{n_q}]^T$	<b>ODE2</b> acceleration coordinates
<b>ODE1</b> coordinates	$\mathbf{y} = [y_0, \dots, y_{n_y}]^T$	vector of $n_y$ coordinates for first order ordinary differential equations ( <b>ODE1</b> )
<b>ODE1</b> velocities	$\dot{\mathbf{y}} = [\dot{y}_0, \dots, \dot{y}_{n_y}]^T$	vector of $n$ velocities for first order ordinary differential equations ( <b>ODE1</b> )
<b>ODE2</b> Lagrange multipliers	$\boldsymbol{\lambda} = [\lambda_0, \dots, \lambda_m]^T$	vector of $m$ Lagrange multipliers (=algebraic coordinates), representing the linear factors (often forces or torques) to fulfill the algebraic equations; for <b>ODE1</b> and <b>ODE2</b> coordinates
data coordinates	$\mathbf{x} = [x_0, \dots, x_l]^T$	vector of $l$ data coordinates in any configuration
RHS <b>ODE2</b>	$\mathbf{f}_q \in \mathbb{R}^{n_q}$	right-hand-side of <b>ODE2</b> equations; (all terms except mass matrix $\times$ acceleration and joint reaction forces)
RHS <b>ODE1</b>	$\mathbf{f}_q \in \mathbb{R}^{n_y}$	right-hand-side of <b>ODE1</b> equations
<b>AE</b>	$\mathbf{g} \in \mathbb{R}^m$	algebraic equations
mass matrix	$\mathbf{M} \in \mathbb{R}^{n_q \times n_q}$	mass matrix, only for <b>ODE2</b> equations
(tangent) stiffness matrix	$\mathbf{K} \in \mathbb{R}^{n_q \times n_q}$	includes all derivatives of $\mathbf{f}_q$ w.r.t. $\mathbf{q}$
damping/gyroscopic matrix	$\mathbf{D} \in \mathbb{R}^{n_q \times n_q}$	includes all derivatives of $\mathbf{f}_q$ w.r.t. $\mathbf{v}$
step size	$h$	current step size in time integration method
residual	$\mathbf{r}_q \in \mathbb{R}^{n_q}, \mathbf{r}_y \in \mathbb{R}^{n_y}, \mathbf{r}_\lambda \in \mathbb{R}^m$	residuals for each type of coordinates within static/time integration – depends on method
system residual	$\mathbf{r} \in \mathbb{R}^{n_s}$	system residual – depends on method
system coordinates	$\xi$	system coordinates and unknowns for solver; definition depends on solver
Jacobian	$\mathbf{J} \in \mathbb{R}^{n_s \times n_s}$	system Jacobian – depends on method

### 10.3.1 System equations of motion

The system equations of motion in ExUDYN are represented as

$$\mathbf{M}\ddot{\mathbf{q}} + \frac{\partial \mathbf{g}}{\partial \mathbf{q}^T} \lambda = \mathbf{f}_q(\mathbf{q}, \dot{\mathbf{q}}, t) \quad (10.13)$$

$$\dot{\mathbf{y}} + \frac{\partial \mathbf{g}}{\partial \mathbf{y}^T} \lambda = \mathbf{f}_y(\mathbf{y}, t) \quad (10.14)$$

$$\mathbf{g}(\mathbf{q}, \dot{\mathbf{q}}, \mathbf{y}, \lambda, t) = 0 \quad (10.15)$$

Note that the term  $\frac{\partial \mathbf{g}}{\partial \mathbf{y}} \lambda_y$  is not yet implemented, such that algebraic equations may not yet depend on 1<sup>st</sup> order differential equations coordinates.

It is important to note, that for linear mechanical the term  $\mathbf{f}_q$  becomes

$$\mathbf{f}_q^{lin} = \mathbf{f}_a - \mathbf{K}\mathbf{q} - \mathbf{D}\dot{\mathbf{q}} \quad (10.16)$$

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

# Chapter 11

## Solvers

Note that the information on solvers is not as complete as the reference pages for items.

### 11.1 Solvers in EXUDYN

The user has a couple of basic solvers available in EXUDYN :

- `exudyn.SolveStatic(...)`: compute static solution for given problem (may also be used to compute kinematic behaviour by prescribing joint motion)
- `exudyn.SolveDynamic(...)`: time integration of equations of motion
- `exudyn.ComputeLinearizedSystem(...)`: computes the linearized system of equations and returns mass, stiffness, damping matrices
- `exudyn.ComputeODE2Eigenvalues(...)`: computes the eigenvalues of the linearized system of equations; only possible if no algebraic constraints in system; uses `scipy` to compute eigenvalues

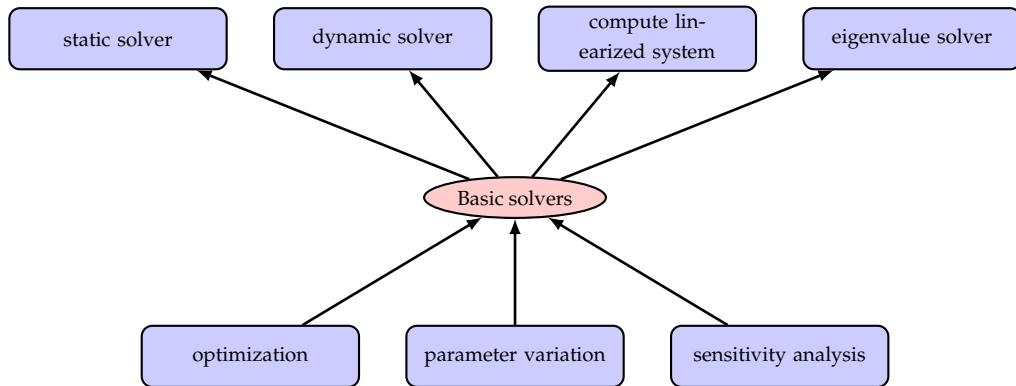


Figure 11.1: Basic and advanced solvers in EXUDYN ; advanced solvers build upon any basic solver to perform more sophisticated operations

There are advanced solvers, like in `exudyn.processing`:

- **Optimization:**

`GeneticOptimization(...)`: find optimum for given set of parameter ranges using genetic optimization; works in parallel  
`Minimize(...)`: find optimum with `scipy.optimize.minimize(...)`

- **ParameterVariation(...)**: compute a series of simulations for given set(s) of parameters; works in parallel

- `ComputeSensitivities(...)`: compute sensitivities for certain parameters; works in parallel

The advanced methods are build upon the basic solvers and essentially run single simulations in the background, see the according examples.

The basic solvers need a `MainSystem`, usually denoted as `mbs`, to be solved. Furthermore, a couple of options are usually to be given, which are explained shortly:

- `simulationSettings`: This is a big structure, containing all solver options; note that only the according options for `staticSolver` or `timeIntegration` are used. Look at the detailed description of these options in [Section 8](#). These settings influence the output rate and output quantity of the solution, solver reporting, accuracy, solver type, etc. Specifically, the `verboseMode` may be increased (2-4) to see the behavior of the solver and intermediate quantities.
- `solverType`: Only for `exudyn.SolveDynamic(...)`: This is a simpler access to the `solverType` given in the internal structure of

```
timeIntegration.generalizedAlpha and
simulationSettings.timeIntegration.explicitIntegration.dynamicSolverType.
```

The function `exudyn.SolveDynamic(...)` sets the according variables internally. For available solver types, see the description of `exudyn.DynamicSolverType` in [Section 4.9.5](#).

- `storeSolver`: if `True`, the solver is stored in `mbs.sys['staticSolver']` or `mbs.sys['dynamicSolver']` and also solver settings are stored in `mbs.sys['simulationSettings']`. After the solver has finished, `mbs.sys['staticSolver']` can be used to retrieve additional information on convergence, system matrices, etc. (see the solver structure).
- `showHints`: This shows a lot of possible solutions in case of no convergence
- `showCausingItems`: This shows a potential causing item if the linear solver failed; the item number is computed from the coordinate number that caused problems (e.g., a row that became zero during factorization); note that this item may not be the real cause in your problem

## 11.2 General solver structure

The description of solvers in this section follows the nomenclature given in [Chapter 10](#). Both in the static as well as in the dynamic case, the solvers run in a loop to solve a nonlinear system of (differential and/or algebraic) equations over a given time or load interval. Explicit solvers only perform a factorization of the mass matrix, but the Newton loop, see Fig. 11.6, is replaced by an explicit computation of the time step according to a given Runge-Kutta tableau.

In case of an implicit time integration, Fig. 11.2 shows the basic loops for the solution process. The inner loops are shown in Fig. 11.4 and Fig. 11.5. The static solver behaves very similar, while no velocities or accelerations need to be solved and time is replaced by load steps.

Settings for the solver substructures, like timer, output, iterations, etc. are described in Sections 8.3.1 – 8.3.5. The description of interfaces for solvers starts in [Section 8.3.6](#).

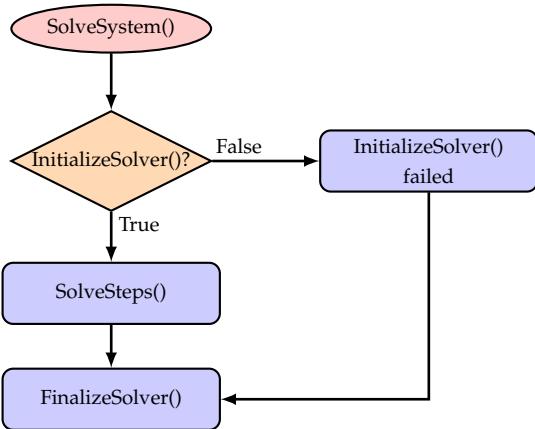


Figure 11.2: Basic solver flow chart for `SolveSystem()`. This flow chart is the same for static solver and for time integration.

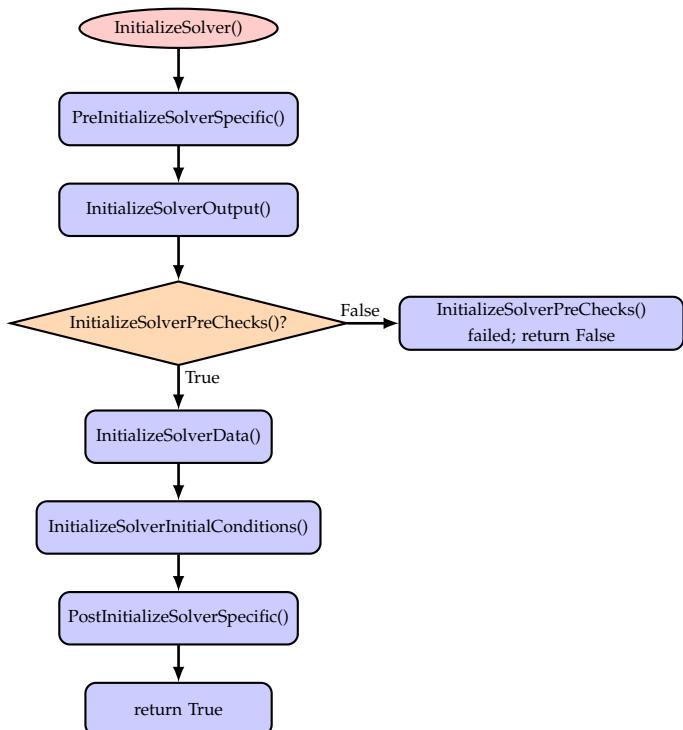


Figure 11.3: Basic solver flow chart for function `InitializeSolver()`.

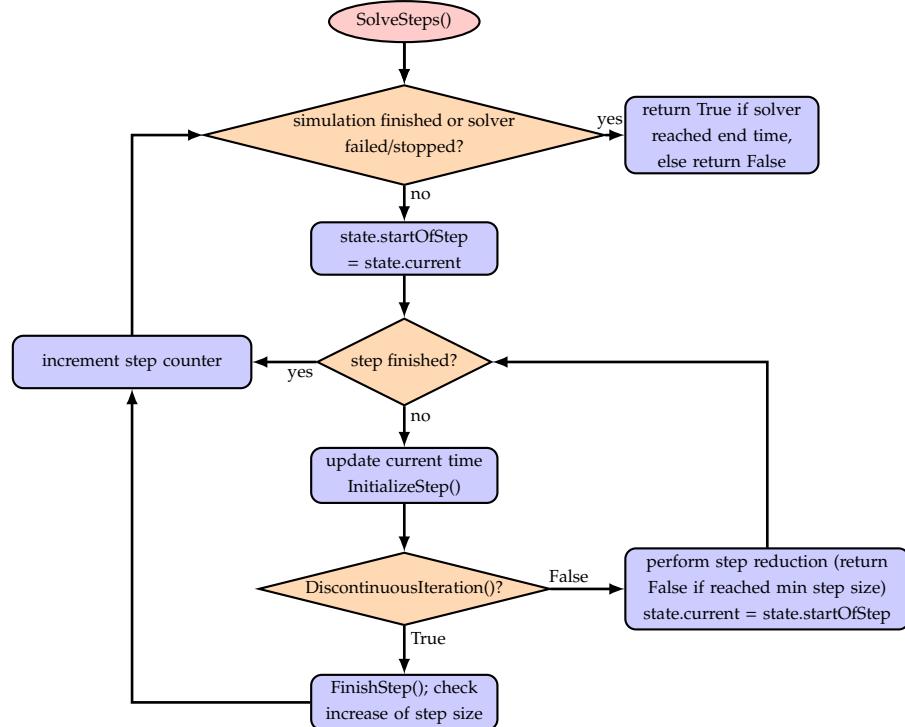


Figure 11.4: Solver flow chart for `SolveSteps()`, which is the inner loop of the solver.

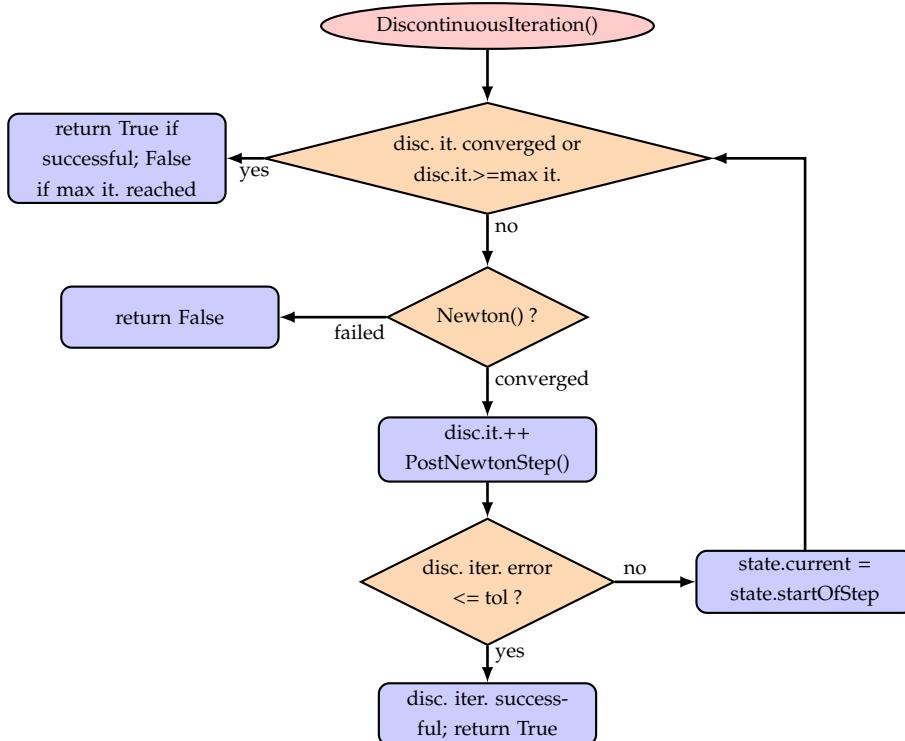


Figure 11.5: Solver flow chart for `DiscontinuousIteration()`, which is run for every solved step inside the static/dynamic solvers. If the `DiscontinuousIteration()` returns `False`, `SolveSteps()` will try to reduce the step size.

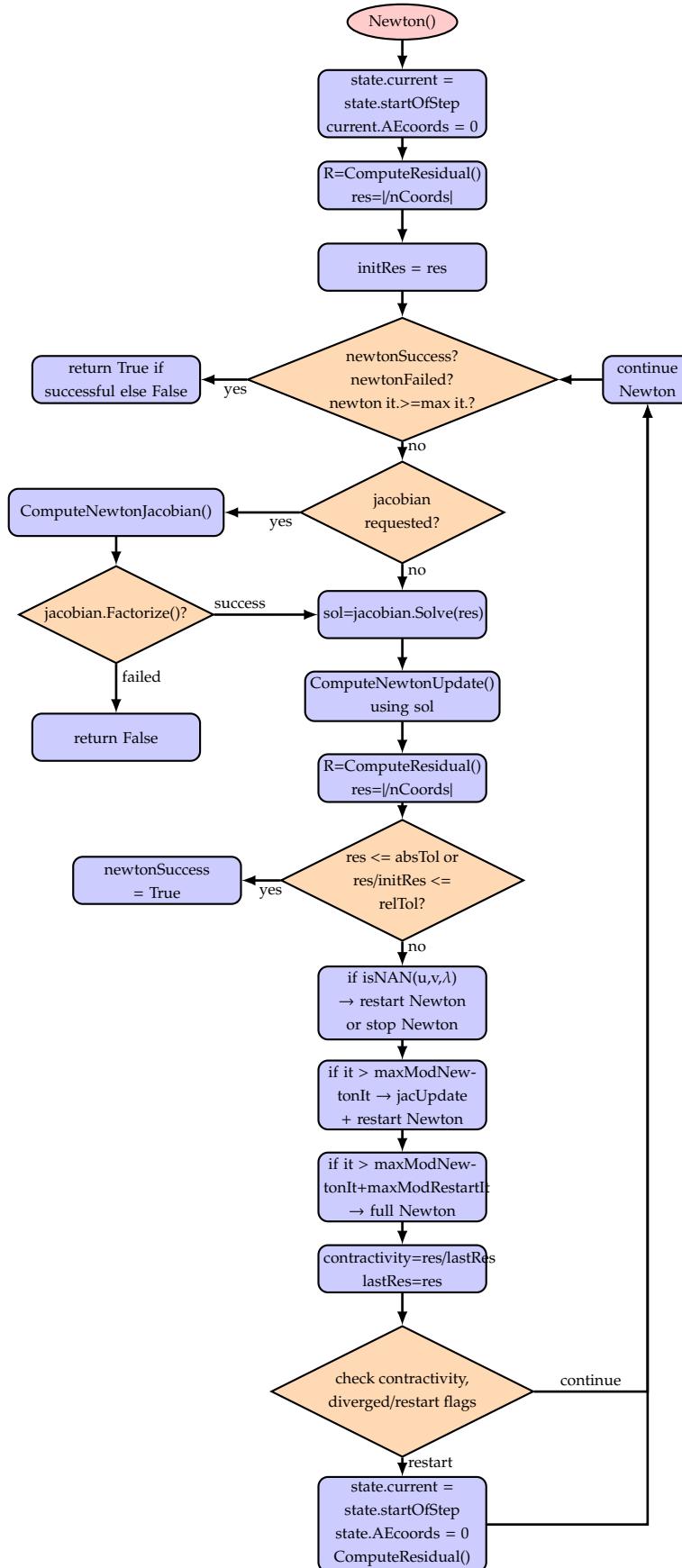


Figure 11.6: Solver flow chart for Newton(), which is run inside the DiscontinuousIteration(). The shown case is valid for newtonResidualMode = 0.

## 11.3 Explicit solvers

Explicit solvers are in general only applicable for systems without constraints (i.e., no joints!). However, some solvers accept simple `CoordinateConstraint`, e.g., fixing coordinates to the ground. Nevertheless, for constraint-free systems, e.g., with penalty constraints, can be solved for very high order and with great efficiency. A list of explicit solvers is available, see [Section 4.9.5](#), for an overview of all implicit and explicit solvers.

The solution vector  $\xi$  (denoted as  $y$  in the literature [20]), which is defined as

$$\xi = [\mathbf{q}^T \ \dot{\mathbf{q}}^T \ \mathbf{y}^T]^T \quad (11.1)$$

and which includes `ODE2` coordinates and velocities and `ODE1` coordinates. All coordinates are computed without reference values.

The `ODE1` and `ODE2` equations of Eq. (11.2), with  $\lambda = 0$ , are written in explicit form and converted to first order equations,

$$\begin{aligned} \dot{\mathbf{q}} &= \mathbf{v} \\ \ddot{\mathbf{v}} &= \mathbf{M}^{-1} \mathbf{f}_q(\mathbf{q}, \mathbf{v}, t) \\ \dot{\mathbf{y}} &= \mathbf{f}_y(\mathbf{y}, t) \end{aligned} \quad (11.2)$$

(11.3)

The system first order differential equations for explicit solvers thus read

$$\dot{\xi} = \mathbf{f}_e(\xi, t) \quad (11.4)$$

### 11.3.1 Explicit Runge-Kutta method

Explicit time integration methods seek the solution  $\xi_{t+h}$  at time  $t + h$  for given initial value  $\xi_t$  (at the beginning of one step  $t$  or at the beginning of the simulation,  $t = 0$ ),

$$\xi_{t+h} = \xi_t + \Delta\xi. \quad (11.5)$$

For any given Runge-Kutta method, the integration of one step with step size  $h$  is performed by an approximation

$$\Delta\xi = \int_t^{t+h} \mathbf{f}_e(\tau, \xi(\tau)) d\tau \approx h [b_1 \mathbf{f}_e(t, \xi(t)) + b_2 \mathbf{f}_e(t + c_2 h, \xi(t + c_2 h)) + \dots + b_s \mathbf{f}_e(t + \xi_s h, u(t + \xi_s h))] \quad (11.6)$$

in which  $t + c_i h$  is the time for stage  $i$  and  $b_i$  the according weight given in the integration formula. Stages are within one step (therefore called one-step-methods), where  $c_i = 0$  represents the beginning of the step and  $c_i = 1$  the end. Note that  $c_1 = 0$  for explicit integration formulas.

The unknown solution vectors  $\xi$  at the stages are abbreviated by

$$\mathbf{g}_i \approx \xi(t + c_i h) \quad (11.7)$$

and computed by explicit integration (quadrature) formulas of lower order ( $g_i$  not to be mixed up with algebraic equations!),

$$\begin{aligned} \mathbf{g}_1 &= \xi_t \\ \mathbf{g}_2 &= \xi_t + h a_{21} \mathbf{f}_e(t, \mathbf{g}_1) \\ \mathbf{g}_3 &= \xi_t + h [a_{31} \mathbf{f}_e(t, \mathbf{g}_1) + a_{32} \mathbf{f}_e(t + c_2 h, \mathbf{g}_2)] \\ &\vdots \\ \mathbf{g}_s &= \xi_t + h [a_{s1} \mathbf{f}_e(t, \mathbf{g}_1) + a_{s2} \mathbf{f}_e(t + c_2 h, \mathbf{g}_2) + \dots + a_{s,s-1} \mathbf{f}_e(t + c_{s-1} h, \mathbf{g}_{s-1})] \end{aligned} \quad (11.8)$$

Explicit Euler method, number of stages  $s = 1$ , order  $p = 1$ :

0	
1	

Explicit midpoint rule, number of stages  $s = 2$ , order  $p = 2$ :

0	
1/2	
0    1	

Classical explicit Runge-Kutta method (RK44), number of stages  $s = 4$ , order  $p = 4$ :

0	
1/2	
1/2	
1	
1/6    1/3    1/3    1/6	

Table 11.1: Several examples of tableaus for the implemented Runge-Kutta methods.

After all vectors  $\mathbf{g}_i$  have been consecutively evaluated, the step is updated by Eq. (11.6).

Conventional explicit Runge-Kutta solvers, such as `ExplicitMidPoint`, `RK44` or `RK67` are based on fixed step size and users must control the error by choosing an appropriate global step size. The tableaus for some lower order methods are given in Table 11.1, using the structure

$\mathbf{c}$	
$\mathbf{b}^T$	

with

0	
$c_2$	
⋮	
$c_s$	
0	
$a_{21}$	
⋮	
$a_{s1}$	
$b_1$	
$\cdots$	
$a_{s,s-1}$	
$b_{s-1}$	
0	
$b_s$	

with  $\mathbf{c} = [c_0 = 0, c_1, \dots, c_s]$ ,  $\mathbf{b} = [b_0, b_1, \dots, b_s]$ , and  $\mathbf{A}$  having only entries in the lower left triangle. For number of stages  $s > 4$ , the maximum order of explicit methods is lower than the number of stages, such as for `RK67`, which is order 6 but 7 stages.

### 11.3.2 Automatic step size control

Advanced solvers, such as `ODE23` and `DOPRI5`, include automatic step size control<sup>1</sup>.

<sup>1</sup>activated with `timeIntegration.automaticStepSize = True` in `simulationSettings`

We estimate the error of a time step with current step size  $h$  by using an embedded Runge-Kutta formula, which includes two approximations 11.6 of order  $p$  and  $\hat{p} = p - 1$ , which is obtained by using two different integration formulas with common coefficients  $c_i$ , but two sets of weights  $b_i$  and  $\hat{b}_i$ , leading to two approximations  $\xi$  and  $\hat{\xi}$ . These so-called embedded Runge-Kutta formulas are widely used, for details see Hairer et al. [20].

The according apporximations  $\xi$  and  $\hat{\xi}$  are used to estimate an error

$$e_j = |\xi_j - \hat{\xi}_j| \quad (11.9)$$

for every component  $j$  of the solution vector  $\xi$ . A scaling is used for every component of the solution vector, evaluating at the beginning (0) and end (1) of the time step:

$$s_j = a_{tol} + r_{tol} \cdot \max(|\xi_{0j}|, |\xi_{1j}|) \quad (11.10)$$

Then the relative, scaled, scalar error for the step, which needs to fulfill  $err \leq 1$ , is computed as

$$err = \sqrt{\frac{1}{n} \sum_{j=1}^n \left( \frac{\xi_{1j} - \hat{\xi}_{1j}}{s_j} \right)^2} \quad (11.11)$$

The optimal step size then reads

$$h_{opt} = h \cdot \left( \frac{1}{err} \right)^{(1/(q+1))} \quad (11.12)$$

Currently we use the suggested step size as

$$h_{new} = \min(h_{max}, \min(h \cdot f_{maxInc}, \max(h_{min}, f_{sfty} \cdot h_{opt}))) \quad (11.13)$$

With the maximum step size  $h_{max} = \frac{t_{start} - t_{end}}{n_{steps}}$  and the minimum step size  $h_{min}$ , given in the `timeIntegration simulationSettings`. The factor  $f_{maxInc}$  limits the increase of the current step size  $h$ , the factor  $f_{sfty}$  is a safety factor for limiting the chosen step size relative to the optimal one in order to avoid frequent step rejections. If  $h_{new} \leq h$ , the current step is accepted, otherwise the step is recomputed with  $h_{new}$ . For more details, see Hairer et al. [20].

### 11.3.3 Stability limit

Note that there are hard limitations for every explicit integration method regarding the step size. Especially for stiff systems (basically with high stiffness parameters and small masses, but also with restrictions to damping), the **step size  $h$  has an upper limit**:  $h < h_{lim}$ . Above that limit the method is inherently unstable, which needs to be considered both for constant and automatic step size selection.

### 11.3.4 Explicit Lie group integrators

All explicit solvers including the automatic step size solvers (DOPRI5, ODE23) have been equiped with Lie group integration functionality – details will be published soon and some tests are made, but handle this with care.

Basically, the integration formulas, see [Section 11.3.1](#) are extended for special rotation parameters. Lie group integration is currently only available for `NodeRigidBodyRotVecLG` used in `ObjectRigidBody` (3D rigid body). `FFRFreducedOrder` will be extended to such nodes in the near future. To get Lie group integrators running with rigid body models, all 3D node types need to be set to `NodeRigidBodyRotVecLG` and set `explicitIntegration.useLieGroupIntegration == True`.

### 11.3.5 Constraints with explicit solvers

Explicit solvers generally do not solve for algebraic constraints, except for very simple CoordinateConstraint. All connectors having the additional type=Constraint, see the according object in [Section 7.6.1ff.](#), are in general not solvable by explicit solvers. Currently, only CoordinateConstraint with one coordinate fixed to ground can be accounted for, if `explicitIntegration.eliminateConstraints == True`. However, this offers the great flexibility to compute finite elements (imported meshes or ANCF beams) to be (partially) fixed to ground. A CoordinateConstraint that fixes a coordinate with index  $j$  to ground leads to the simple algebraic `ODE2` equation

$$g_j(\mathbf{q}) = 0 \Leftrightarrow q_j = 0 \quad (11.14)$$

which can be solved by the implemented explicit solvers by just setting  $q_j = 0$  previously to every computation and  $\dot{q}_j = 0$  after every [RHS](#) evaluation.

NOTE that, if `explicitIntegration.eliminateConstraints == False`, constraints are ignored by the explicit solver (and all algebraic variables are set to zero). This may be wanted (e.g. to investigate the free motion of bodies), but in general leads to wrong and meaningless solution.

## 11.4 Implicit (trapezoidal rule-based, Newmark, generalized- $\alpha$ ) solver

This solver represents a class of solvers, which are – in the undamped case – based on the implicit trapezoidal rule (in the view of Runge-Kutta methods). The interpolation of the quantities for one step includes the start and the end value of the time step, thus being called trapezoidal integration rule. In some special cases in Newmark's method [30], the interpolation might only depend on the start value or the end value.

For now, all implemented solvers can be viewed as a generalization of Newmark's method, but there are called differently in the solver interfaces

- **Implicit trapezoidal rule** (= Newmark with  $\beta = \frac{1}{4}$  and  $\gamma = \frac{1}{2}$ )
- **Newmark's method** [30]
- **Generalized- $\alpha$  method** (= generalized Newmark method with additional parameters), see Chung and Hulbert [7] for the original method and Arnold and Brüls [1] for the application to multibody system dynamics.

### 11.4.1 Newmark and generalized- $\alpha$ method

Newmark's method has two parameters  $\beta$  and  $\gamma$ . The main ideas are

- Interpolate the displacements and the velocities linearly using the accelerations  $\ddot{\mathbf{q}}$  of the beginning of the time step (subindex '0') and the end of the time step (subindex 'T'). The 2<sup>nd</sup> order differential equations displacements and velocities and for 1<sup>st</sup> order differential equations coordinates are given by (definition of  $\mathbf{a}$  will become clear later):

$$\begin{aligned} \mathbf{q}_T &= \mathbf{q}_0 + h\dot{\mathbf{q}}_0 + h^2(\frac{1}{2} - \beta)\mathbf{a}_0 + h^2\beta\mathbf{a}_T \\ \dot{\mathbf{q}}_T &= \dot{\mathbf{q}}_0 + h(1 - \gamma)\mathbf{a}_0 + h\gamma\mathbf{a}_T \\ \mathbf{y}_T &= \mathbf{y}_0 + h(1 - \gamma_y)\mathbf{v}_y^0 + h\gamma_y\mathbf{v}_y^T \end{aligned} \quad (11.15)$$

- Solve the system equations at the end of the time step ( $T$ ) for the unknown accelerations as well as for 1<sup>st</sup> order differential equations and algebraic equations coordinates.

Remarks:

- The system of equations may be solved for accelerations  $\ddot{\mathbf{q}}$ , but also for displacements  $\mathbf{q}$  or even velocities as unknowns while the remaining quantities are reconstructed from Eq. (11.15). In case of displacements as unknowns, a scaling of the Jacobian is necessary, see later.
- For consistency reasons, one may set  $\gamma_y = \gamma$ , but currently we use  $\gamma_T = \frac{1}{2}$ , leading to no numerical damping for ODE1 variables  $\mathbf{y}$ .
- In the extension to the so-called generalized- $\alpha$  method [7], algorithmic accelerations  $\mathbf{a}$  are used in Eq. (11.15). Algorithmic accelerations are no longer equivalent to the time derivatives of displacements,  $\mathbf{a} \neq \ddot{\mathbf{q}}$ ; thus, both sets of variables are used independently. In case of Newmark or the implicit trapezoidal rule just use  $\mathbf{a} = \ddot{\mathbf{q}}$ .
- For generalized- $\alpha$ , the algorithmic accelerations  $\mathbf{a}$  are computed from the recurrence relation

$$(1 - \alpha_m)\mathbf{a}_T + \alpha_m\mathbf{a}_0 = (1 - \alpha_f)\ddot{\mathbf{u}}_T + \alpha_f\ddot{\mathbf{u}}_0 \quad (11.16)$$

which can be resolved for the unknown  $\mathbf{a}_T$ ,

$$\mathbf{a}_T = \frac{(1 - \alpha_f)\ddot{\mathbf{u}}_T + \alpha_f\ddot{\mathbf{u}}_0 - \alpha_m\mathbf{a}_0}{(1 - \alpha_m)} \quad (11.17)$$

For the first step, one can simply use  $\mathbf{a}_0 = \ddot{\mathbf{q}}_0$ .

### 11.4.2 Parameter selection for generalized- $\alpha$

Compared to alternative implicit integration methods (including the Newmark method), the generalized- $\alpha$  integrator's parameters break down to one single parameter  $\rho_\infty$ , which allows to chose numerical damping in a practical way.

Based on a simple single DOF mass-spring-damper model [3], having the eigen frequency  $\omega = 2\pi f$  with frequency  $f$  and period  $T = 1/f$ , the spectral radius  $\rho$  for the integrator defines the amount of damping for a given step size  $h$  related to  $T$ , thus using the dimensionless step size  $\bar{h} = h/T$ .

In Fig. 11.7 the spectral radius is shown versus  $\bar{h}$  for various spectral radii at infinity  $\rho_\infty$ . Here,  $\rho_\infty$  specifies the numerical damping of very time step for large step sizes (or very high frequencies). An amount of  $\rho_\infty = 0.9$  means that high frequency parts of the system ( $(\bar{h} \gg 1)$ ; high compared to the step rate) are damped to 90% in every step, reducing an initial value 1 to  $2.66e-5$  after 100 steps, which is already much larger than usual physical damping in many cases.

Furthermore, low frequency parts of the system ( $\bar{h} \ll 1$ ) receive almost no numerical damping, see again Fig. 11.7. Exemplarily, consider  $\rho$  a low frequency situation with different  $\rho_\infty$ :

- $\rho(\bar{h} = 0.01, \rho_\infty = 0.9) = 1 - 1.13 \cdot 10^{-9}$
- $\rho(\bar{h} = 0.01, \rho_\infty = 0.6) = 1 - 1.22 \cdot 10^{-7}$

which shows that numerical damping is very low for moderately small step sizes (100 steps for one oscillation).

Obviously,  $\rho_\infty$  does not have a large influence for very high or low frequencies in the system as long as it is  $\neq 1$  and we could even use  $\rho_\infty = 0$ . Regarding differential algebraic equations (DAEs),  $\rho_\infty < 1$  allows to integrate index 3 DAEs. Typically a value of  $\rho_\infty = 0.7$  leads to a stable integration, but values depend on the structure of the multibody system.

Once having chosen  $\rho_\infty$ , all other parameters follow automatically [7], regarding the  $\alpha$ s

$$\alpha_m = \frac{2\rho_\infty - 1}{\rho_\infty + 1}, \quad \alpha_f = \frac{\rho_\infty}{\rho_\infty + 1} \quad (11.18)$$

and Newmarks's parameters,

$$\gamma = \frac{1}{2} - \alpha_m + \alpha_f, \quad \beta = \frac{1}{4}(1 - \alpha_m + \alpha_f)^2 \quad (11.19)$$

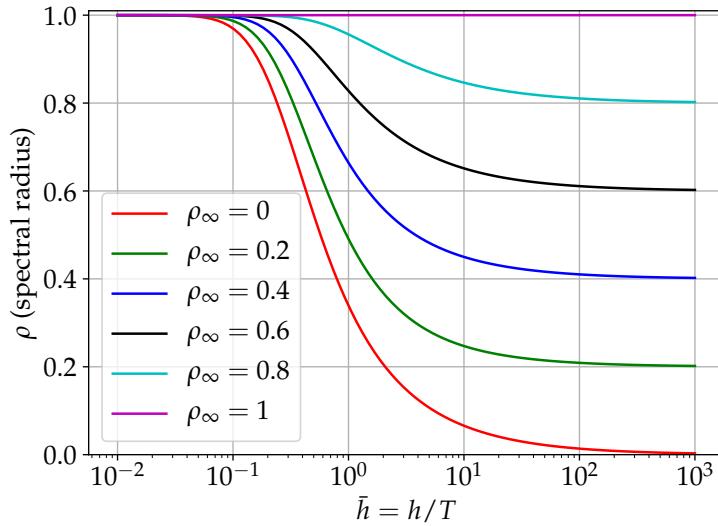


Figure 11.7: Spectral radius for generalized- $\alpha$  method depending on dimensionless step size  $\bar{h} = h/T$ , in which  $T$  is the period of an equivalent single DOF mass-spring-damper system.

### 11.4.3 Newton iteration

Thus, the residuals at the end of the time step ( $T$ ) read (put all terms to LHS):

$$\mathbf{r}_q^{G\alpha} = \mathbf{M}\ddot{\mathbf{q}}_T + \frac{\partial \mathbf{g}}{\partial \mathbf{q}^T} \lambda_T - \mathbf{f}_q(\mathbf{q}_T, \dot{\mathbf{q}}_T, t) = 0 \quad (11.20)$$

$$\mathbf{r}_y^{G\alpha} = \dot{\mathbf{y}}_T + \frac{\partial \mathbf{g}}{\partial \mathbf{y}^T} \lambda_T - \mathbf{f}_y(\mathbf{y}_T, t) = 0 \quad (11.21)$$

$$\mathbf{r}_\lambda^{G\alpha} = \mathbf{g}(\mathbf{q}_T, \dot{\mathbf{q}}_T, \mathbf{y}_T, \lambda_T, t) = 0 \quad (11.22)$$

We consider two options for 2<sup>nd</sup> order differential equations: (A) solve for unknown accelerations  $\ddot{\mathbf{q}}_T$ , or (B) for unknown displacements  $\mathbf{q}_T$ .

#### 11.4.3.1 (A) Solve for unknown accelerations

The unknowns for the Newton method then are

$$\xi_{k+1}^{G\alpha} = \begin{bmatrix} \ddot{\mathbf{q}}_T \\ \mathbf{y}_T \\ \lambda_T \end{bmatrix} \quad (11.23)$$

and at the beginning of the step, we have

$$\xi_k^{G\alpha} = \begin{bmatrix} \ddot{\mathbf{q}}_0 \\ \mathbf{y}_0 \\ \lambda_0 \end{bmatrix} \quad (11.24)$$

For the Newton method, we need to compute an update for the unknowns of Eq. (11.23), using the previous residual  $\mathbf{r}_k$  and the inverse of the Jacobian  $\mathbf{J}_k$  of Newton iteration  $k$ ,

$$\xi_{k+1}^{G\alpha} = \xi_k^{G\alpha} - \mathbf{J}^{-1}(\xi_k^{G\alpha}) \cdot \mathbf{r}^{G\alpha}(\xi_k^{G\alpha}) \quad (11.25)$$

The Jacobian has the following  $3 \times 3$  structure,

$$\mathbf{J} = \begin{bmatrix} \mathbf{J}_{qq} & \mathbf{J}_{qy} & \mathbf{J}_{q\lambda} \\ \mathbf{J}_{yq} & \mathbf{J}_{yy} & \mathbf{J}_{y\lambda} \\ \mathbf{J}_{\lambda q} & \mathbf{J}_{\lambda y} & \mathbf{J}_{\lambda\lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{J}_{qq} & \mathbf{0} & \mathbf{J}_{q\lambda} \\ \mathbf{0} & \mathbf{J}_{yy} & \mathbf{J}_{y\lambda} \\ \mathbf{J}_{\lambda q} & \mathbf{J}_{\lambda y} & \mathbf{J}_{\lambda\lambda} \end{bmatrix} \quad (11.26)$$

in which we consider  $\mathbf{J}_{yq}$  and  $\mathbf{J}_{qy}$  to vanish in the current implementations, which means that coupling of [ODE1](#) and [ODE2](#) coordinates is only possible due to algebraic equations.

The remaining terms in the Jacobian read

$$\begin{aligned} \mathbf{J}_{qq} &= \frac{\partial \mathbf{r}_q^{G\alpha}}{\partial \ddot{\mathbf{q}}} = \frac{\partial \mathbf{r}_q^{G\alpha}}{\partial \mathbf{q}} \frac{\partial \mathbf{q}}{\partial \ddot{\mathbf{q}}} + \frac{\partial \mathbf{r}_q^{G\alpha}}{\partial \dot{\mathbf{q}}} \frac{\partial \dot{\mathbf{q}}}{\partial \ddot{\mathbf{q}}} = h^2 \beta \mathbf{K} + h\gamma \mathbf{D} \\ \mathbf{J}_{q\lambda} &= \frac{\partial \mathbf{r}_q^{G\alpha}}{\partial \lambda} = \frac{\partial \mathbf{g}}{\partial \mathbf{q}} \quad (\text{or } \frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}} \text{ for constraints at velocity level}) \\ \mathbf{J}_{yy} &= \frac{\partial \mathbf{r}_y^{G\alpha}}{\partial \mathbf{y}} \\ \mathbf{J}_{\lambda q} &= \frac{\partial \mathbf{r}_\lambda^{G\alpha}}{\partial \ddot{\mathbf{q}}} = \frac{\partial \mathbf{g}}{\partial \ddot{\mathbf{q}}} = \frac{\partial \mathbf{g}}{\partial \mathbf{q}} \frac{\partial \mathbf{q}}{\partial \ddot{\mathbf{q}}} + \frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}} \frac{\partial \dot{\mathbf{q}}}{\partial \ddot{\mathbf{q}}} = h^2 \beta \frac{\partial \mathbf{g}}{\partial \mathbf{q}} + h\gamma \frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}} \\ \mathbf{J}_{\lambda y} &= \frac{\partial \mathbf{r}_\lambda^{G\alpha}}{\partial \mathbf{y}} \\ \mathbf{J}_{\lambda\lambda} &= \frac{\partial \mathbf{r}_\lambda^{G\alpha}}{\partial \lambda} = \frac{\partial \mathbf{g}}{\partial \lambda} \end{aligned} \quad (11.27)$$

Once an update  $\mathbf{q}_{k+1}^{\text{Newton}}$  has been computed, the interpolation formulas (11.15) need to be evaluated before the next residual and Jacobian can be computed.

#### 11.4.3.2 (B) Solve for unknown displacements

This approach is similar to the previous approach and follows exactly the algorithm given by Arnold and Brüls [1], however, extended for [ODE1](#) variables, which are integrated by the (undamped) trapezoidal rule. Documentation will be added lateron.

#### 11.4.4 Initial accelerations

For the solvers based on the implicit trapezoidal rule, initial accelerations are necessary in order to significantly increase the accuracy of the first time step. For this reason, the constraints  $\mathbf{g}(\mathbf{q}_0, \dot{\mathbf{q}}_0, \mathbf{y}_0, \lambda_0, t) = 0$  in Eq. (10.13) are differentiated w.r.t. time,

$$\dot{\mathbf{g}}(\mathbf{q}_0, \dot{\mathbf{q}}_0, \mathbf{y}_0, \lambda_0, t) = \frac{\partial \mathbf{g}}{\partial \mathbf{q}} \dot{\mathbf{q}}_0 + \frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}} \ddot{\mathbf{q}}_0 + \frac{\partial \mathbf{g}}{\partial \mathbf{y}} \dot{\mathbf{y}}_0 + \frac{\partial \mathbf{g}}{\partial \lambda} \dot{\lambda} + \frac{\partial \mathbf{g}}{\partial t} = 0. \quad (11.28)$$

Currently, we assume  $\frac{\partial \mathbf{g}}{\partial \lambda} = 0$  for all further derivations on initial accelerations. For velocity level constraints, Eq. (11.28) is used to extract initial accelerations  $\ddot{\mathbf{q}}_0$ ,

$$\frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}} \dot{\mathbf{q}}_0 = - \frac{\partial \mathbf{g}}{\partial \mathbf{q}} \dot{\mathbf{q}}_0 - \frac{\partial \mathbf{g}}{\partial \mathbf{y}} \dot{\mathbf{y}}_0 - \frac{\partial \mathbf{g}}{\partial t}. \quad (11.29)$$

Finally, the equations for the computation of the initial accelerations read for velocity level constraints, note that  $\mathbf{y}_{init}$  are the nodal initial values for  $\mathbf{y}$ ,

$$\begin{bmatrix} \mathbf{M} & \mathbf{0} & \frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \ddot{\mathbf{q}}_0 \\ \mathbf{y}_0 \\ \lambda_0 \end{bmatrix} = \begin{bmatrix} \mathbf{f}_q(\mathbf{q}_T, \dot{\mathbf{q}}_T, t) \\ \mathbf{y}_{init} \\ -\frac{\partial \mathbf{g}}{\partial \mathbf{q}} \dot{\mathbf{q}}_0 - \frac{\partial \mathbf{g}}{\partial \mathbf{y}} \dot{\mathbf{y}}_0 - \frac{\partial \mathbf{g}}{\partial t} \end{bmatrix}, \quad (11.30)$$

The term  $\frac{\partial \mathbf{g}}{\partial t}$  can only occur in case of user functions and therefore currently not implemented, and the ODE1 term  $\frac{\partial \mathbf{g}}{\partial y} = 0$  is not used yet in constraints.

For position level constraints, we assume  $\frac{\partial \mathbf{g}}{\partial \dot{\mathbf{q}}} = 0$  and  $\frac{\partial \mathbf{g}}{\partial y} = 0$  in Eq. (11.28) and perform a second derivation w.r.t. time,

$$\ddot{\mathbf{g}}(\mathbf{q}_0, \dot{\mathbf{q}}_0, \mathbf{y}_0, \lambda_0, t) = \frac{\partial^2 \mathbf{g}}{\partial \mathbf{q}^2} \dot{\mathbf{q}}_0^2 + 2 \frac{\partial^2 \mathbf{g}}{\partial \mathbf{q} \partial t} \dot{\mathbf{q}}_0 + \frac{\partial \mathbf{g}}{\partial \mathbf{q}} \ddot{\mathbf{q}}_0 + \frac{\partial^2 \mathbf{g}}{\partial t^2} = 0. \quad (11.31)$$

For position level constraints, Eq. (11.31) is used to extract initial accelerations  $\ddot{\mathbf{q}}_0$ ,

$$\frac{\partial \mathbf{g}}{\partial \mathbf{q}} \ddot{\mathbf{q}}_0 = -2 \frac{\partial^2 \mathbf{g}}{\partial \mathbf{q} \partial t} \dot{\mathbf{q}}_0 - \frac{\partial^2 \mathbf{g}}{\partial \mathbf{q}^2} \dot{\mathbf{q}}_0^2 - \frac{\partial^2 \mathbf{g}}{\partial t^2}. \quad (11.32)$$

Finally, the equations for the computation of the initial accelerations for position level constraints read

$$\begin{bmatrix} \mathbf{M} & \mathbf{0} & \frac{\partial \mathbf{g}}{\partial \mathbf{q}^T} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \frac{\partial \mathbf{g}}{\partial \mathbf{q}} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \ddot{\mathbf{q}}_0 \\ \mathbf{y}_0 \\ \lambda_0 \end{bmatrix} = \begin{bmatrix} \mathbf{f}_q(\mathbf{q}_T, \dot{\mathbf{q}}_T, t) \\ \mathbf{y}_{init} \\ -2 \frac{\partial^2 \mathbf{g}}{\partial \mathbf{q} \partial t} \dot{\mathbf{q}}_0 - \frac{\partial^2 \mathbf{g}}{\partial \mathbf{q}^2} \dot{\mathbf{q}}_0^2 - \frac{\partial^2 \mathbf{g}}{\partial t^2} \end{bmatrix}, \quad (11.33)$$

The linear system of equations 11.30 or 11.33 is solved prior to an implicit time integration if

```
simulationSettings.timeIntegration.generalizedAlpha.computeInitialAccelerations = True
```

which is the default value.

## 11.5 Optimization and parameter variation

The real benefit of powerful multi-body simulation emerges only if combined with modern but also simple analysis and evaluation methods. Therefore, ExUDYN has been integrated into the Python language, which offers a virtually unlimited number of methods of post-processing, evaluation and optimization. In this section, two methods that are directly integrated into ExUDYN are revisited.

### 11.5.1 Parameter Variation

Parameter variation is one of the simplest tools to evaluate the dependency of the solution of a problem on certain parameters. This usually requires the computation of an objective (goal, result) value for a single computation (e.g., some error norm, maximum vibration amplitude, maximum stress, maximum deflection, etc.) for every computation. Furthermore, it needs to be run for a set of parameters, e.g., using a `for` loop. While this could be done manually in ExUDYN, it is recommended to use built-in functions, which simplify evaluation and postprocessing and directly enable parallelization. The according function `ParameterVariation(...)`, see [Section 5.12](#), performs a set of multi-dimensional parameter variations using a dictionary that describes the variation of parameters. See also `parameterVariationExample.py` in the `Examples` folder for a simple example showing a 2D parameter variation. The function `ParameterVariation(...)` requires the `multiprocessing` Python module which enables simple multi-threaded parallelism and has been tested for up to 80 cores on the LEO4 supercomputer at the University of Innsbruck, achieving a speedup of 50 as compared to a serial computation.

### 11.5.2 Genetic Optimization

In engineering, we often need to find a set of unknown, independent parameters  $\mathbf{x} \in \mathbb{R}^n$ ,  $\mathbf{x}$  being denoted as design variables and  $\mathbb{R}^n$  as design space. Sometimes, the design space is further subjected to constraints  $\mathbf{g}(\mathbf{x}) = 0$  as well as inequalities  $\mathbf{h}(\mathbf{x}) \leq 0$ , which are not considered here. For simple solutions for constrained optimization problems using penalty methods, see the introductory literature [26].

Optimization problems are written in general in the form

$$\min_{\mathbf{x}} f(\mathbf{x}), \quad \mathbf{x} \in \mathbb{R}^n, \quad (11.34)$$

where  $f(\mathbf{x})$  denotes the *objective function* (=fitness function). If we would like to maximize a function  $\tilde{f}(\mathbf{x})$ , simply set  $f(\mathbf{x}) = -\tilde{f}(\mathbf{x})$ .

In engineering, the optimization problem could seek model parameters, e.g., the geometric dimensions and inertia parameters of a slider crank mechanism, in order to achieve smallest possible forces at the supports. Another example is the identification of unknown physical parameters, such as stiffness, damping or friction. This can be achieved by comparing measurement and simulation data (e.g., accelerations measured at relevant parts of a machine). Lets assume that  $\epsilon(t)$  is an error computed in every time step of a computation, then we can set the objective (=fitness) function, e.g., as

$$f(\mathbf{x}) = \frac{1}{T} \sqrt{\int_{t=0}^T \epsilon(t)^2 dt} \quad (11.35)$$

as the integral over the error  $\epsilon$  between measurement and simulation data. In general, a parameter variation would be sufficient to compute sufficient computations for all combinations within the design space, however, a 3D design space with 100 variations into every direction (e.g., varying the unknown damping coefficient between 1 and 100, etc.) would already require 1000.000 computations, which in an ideal case of 1 second/computation leads to almost 2 weeks of computation time.

As an alternative stochastic methods can be used to compute only the objective function for a smaller set of randomly generated design variables, which usually show regions with better parameters (lower  $f$ ) in scatter plots.

**Genetic algorithms**[19, 39] can significantly reduce the necessary amount of objective function evaluations in order to perform the optimization. Genetic identification algorithms have been already successfully applied to multibody system dynamics[10].

The general structure of a (canonical) genetic algorithm is depicted in Fig. 11.8. For details, see the cited

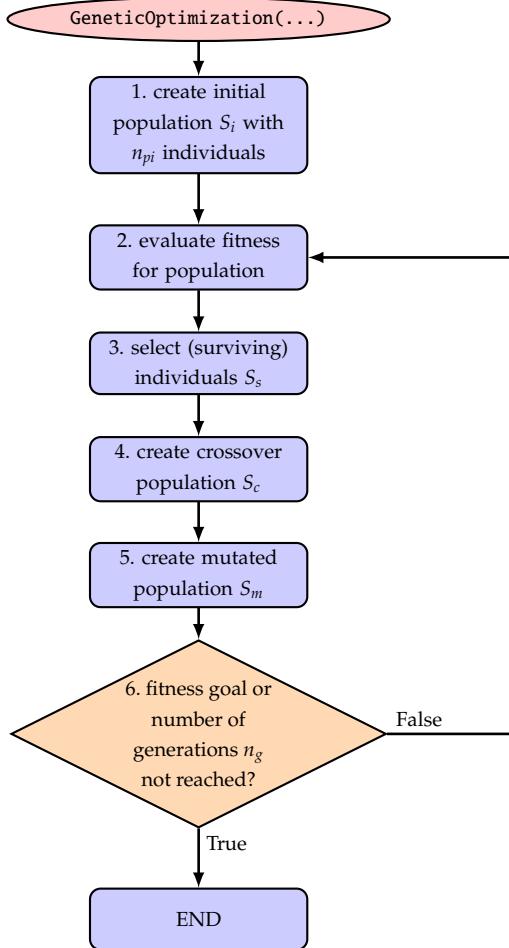


Figure 11.8: Basic solver flow chart genetic algorithm / optimization.

literature. Here, we focus on the implementation of the function `GeneticOptimization(...)`, see [Section 5.12](#). The initial population (step 1) is created with `initialPopulationSize` individuals with uniformly distributed random design variables  $[x_0, \dots, x_{n_{pi}-1}]^2$  in the search space, which is given in the dictionary `parameters`. Herafter (steps 2-6), we iteratively process a population for a certain `numberOfGenerations` generations.

In step 3, the surviving individuals  $S_s$  with best fitness (smallest value from evaluation of `objectiveFunction`) are selected and considered further in the optimization. If the `distanceFactor` is used, the surviving individuals must be located within a certain distance (measured relative to the range of the search space) to all other surviving individuals. This option guarantees the search within several local minima, while a conventional search often converges to one single minimum. Crossover (step 4) is performed using a crossover of all available parameters of two randomly selected parents when generating children from the surviving individuals.

<sup>2</sup> $\mathbf{x}_i = [x_{i0}, x_{i1}, \dots]$  being a set of genes, with single genes  $x_{i0}, x_{i1}, \dots$

The crossover of genes is performed only for a part of the new population, defined by `crossoverAmount`.

Finally, in step 5, we apply mutation to all genes, which extends the search to the surrounding design space of the individuals created by crossover. The mutation could be performed by means of certain distribution functions in order to focus on the currently best search regions. However, in the current implementation of `GeneticOptimization(...)` we simply use a uniform random variable to distribute the genes over a certain percentage of the design space, which is reduced in every generation defined by the `rangeReductionFactor`  $r_r$ . This allows us to restrict further search to a smaller subregion of the design space and in general allows a reduction of search space by means of  $r_r^{n_g}$ . In the ideal case, using sufficiently large population sizes and being lucky with the found random values, a range reduction factor  $r_r = 0.7$  reduces the search space by a factor of 100 after every 13 generations, allowing to obtain 4 digits of accuracy for design variables after 26 generations for suitable optimization problems.

It should be noted that still this optimization method is based on random values and thus may fail occasionally for any problem case. In order to get reproducible results, set `randomizerInitialization` to any integer value (simply: 0) in order to get identical results for repeated runs. Setting the latter variable guarantees that the Python (numpy) randomizer creates the same series of random values for initial population, mutation, etc.

# Chapter 12

## Issues and bugs

This section contains resolved issues per release and known bugs. Use this information to understand changes compared to previous versions. The mentioning of author is omitted if it was Johannes Gerstmayr (JG). The extension .dev1 is not added in the issues list (e.g., 1.2.2.dev1==1.2.2), as it only marks versions that will not be available in pypi with standard pip install, but only with the -pre option or by specifying the exact version name, see versions on <https://pypi.org/project/exudyn/>. BUG numbers refer to the according issue numbers. For details, see the `trackerlog.html` file.

General information on current version:

- Exudyn version = 1.2.28,
- last change = 2022-04-04,
- Number of issues = 1030,
- Number of resolved issues = 870 (28 in current version),

### 12.1 Resolved issues and resolved bugs

The following list contains the issues which have been **RESOLVED** in the according version:

- **Version 1.2.28:** **resolved BUG 1029: ContactFrictionCircleCable2D**
  - description: computes wrong segment length internally, leading to wrong sticking position
  - date resolved: **2022-04-04 12:14**, date raised: 2022-04-04
- **Version 1.2.27:** **resolved BUG 1028: cnt in Render window**
  - description: debug information cnt=.. shown in Render window
  - date resolved: **2022-04-04 11:50**, date raised: 2022-04-04
- **Version 1.2.26:** resolved Issue 1027: **RotationMatrix2EulerParameters** (change)
  - description: RotationMatrix2EulerParameters computes Euler parameters with large deviations from unit norm in case of inaccurate rotation matrices; this may lead to failure of CheckPreAssembleConsistencies; resolved by adding normalization before returning Euler parameters
  - date resolved: **2022-04-03 00:24**, date raised: 2022-04-03
- **Version 1.2.25:** resolved Issue 1025: **build venv** (extension)
  - description: switch to building windows purely on virtual conda environments, allowing to build all windows and linux versions in parallel
  - date resolved: **2022-04-02 00:28**, date raised: 2022-04-02

- **Version 1.2.24:** resolved Issue 1022: **virtual environments** (change)
  - description: switch to virtual environments in anaconda for compilation on different platforms
  - date resolved: **2022-04-02 00:28**, date raised: 2022-03-31
- **Version 1.2.23:** resolved Issue 1023: **remove /Zi in MSVC** (change)
  - description: remove compilation flag /Zi in setup.py which prevents from parallel runs of MSVC cl.exe
  - date resolved: **2022-03-31 11:28**, date raised: 2022-03-31
- **Version 1.2.22:** resolved Issue 1020: **robotics links** (docu)
  - description: links to github are not resolved correctly for robotics module
  - date resolved: **2022-03-30 10:47**, date raised: 2022-03-30
- **Version 1.2.21:** resolved Issue 1019: **SpaceMouse** (extension)
  - description: add functionality for 3D mouse / spacemouse by reading joystick inputs and interpret as 3D position and rotation data; add visualization flag interactive.useJoystickInput (default=True); deactivate this flag if your external device makes problems
  - date resolved: **2022-03-29 14:37**, date raised: 2022-03-29
- **Version 1.2.20:** resolved Issue 1017: **ContactFrictionCircleCable2D** (testing)
  - description: check and test computation of sticking position segment length: undeformed versus deformed
  - **notes:** switched to reference length in computation of relative sticking position as given in theDoc; leads to improved results
  - date resolved: **2022-03-28 20:33**, date raised: 2022-03-28
- **Version 1.2.19:** resolved Issue 1014: **SaveImage** (change)
  - description: make window height divisible by 2 (skip one line if necessary)
  - **notes:** added alignment option for width and height; default options work well for ffmpeg conversion
  - date resolved: **2022-03-28 14:25**, date raised: 2022-03-28
- **Version 1.2.18:** resolved Issue 1013: **TGA output** (change)
  - description: switch from TGA to .png output using stb\_image\_write.h in GLFW
  - **notes:** added mode exportImages.saveImageFormat to chose between PNG and TGA - PNG with much smaller files!
  - date resolved: **2022-03-28 14:24**, date raised: 2022-03-28
- **Version 1.2.17:** resolved Issue 1012: **GLFW** (change)
  - description: switch to GLFW3.3.6 includes in order to be in line with AppleM1 version
  - date resolved: **2022-03-28 12:03**, date raised: 2022-03-28
- **Version 1.2.16:** resolved Issue 1011: **Apple M1** (change)
  - description: use universal glfw libs for both Apple x86 and Apple arm M1
  - date resolved: **2022-03-28 12:03**, date raised: 2022-03-28
- **Version 1.2.15:** resolved Issue 1010: **autodiff** (change)
  - description: extract autodiff as separate module
  - **notes:** now using AutomaticDifferentiation.h in Utilities
  - date resolved: **2022-03-28 11:56**, date raised: 2022-03-28

- **Version 1.2.14:** **resolved BUG 1009: solutionViewer**
  - description: does not load automatically due to change of coordinatesSolution filename ending
  - **notes: changed loading of default file in SolutionViewer**
  - date resolved: **2022-03-28 09:11**, date raised: 2022-03-28
- **Version 1.2.13:** **resolved BUG 1007: sensor double values**
  - description: sensor outputs two times for a single time step
  - **notes: error occurred due to automaticStepSize activated and call to ReduceStepSize even in case adaptiveStep=0; automaticStepSize now deactivated for solvers without step size control**
  - date resolved: **2022-03-27 19:31**, date raised: 2022-03-24
- **Version 1.2.12:** resolved Issue 1006: **ContactFrictionCircleCable2D** (change)
  - description: LHS computation: exclude undefined state from sticking position computation
  - date resolved: **2022-03-22 12:43**, date raised: 2022-03-22
- **Version 1.2.11:** resolved Issue 1005: **PostNewton timer** (change)
  - description: remove timer from timer structures and activate as special timer
  - date resolved: **2022-03-22 12:30**, date raised: 2022-03-22
- **Version 1.2.10:** resolved Issue 1004: **ContactFrictionCircleCable2D** (extension)
  - description: add option usePointWiseNormals flag as an additional option to control the way forces are applied to cable
  - date resolved: **2022-03-21 10:12**, date raised: 2022-03-21
- **Version 1.2.9:** resolved Issue 1003: **setup.py** (extension)
  - description: include manifest.in and readme.md in main
  - date resolved: **2022-03-20 11:47**, date raised: 2022-03-20
- **Version 1.2.8:** resolved Issue 1002: **pypi problems** (change)
  - description: previous version marked beta
  - date resolved: **2022-03-19 10:54**, date raised: 2022-03-19
- **Version 1.2.7:** resolved Issue 0986: **ANCFCable2D** (docu)
  - description: extend/finalize description - specifically for integration points and OutputVariable functions
  - date resolved: **2022-03-18 23:09**, date raised: 2022-03-15
- **Version 1.2.6:** resolved Issue 1001: **pypi** (change)
  - description: finalized conversion of markup file for description at pypi.org: use only links to github, as pypi does not recognize .rst file in github format
  - date resolved: **2022-03-18 20:54**, date raised: 2022-03-18
- **Version 1.2.5:** resolved Issue 1000: **adjust version in docu** (docu)
  - description: recompile
  - date resolved: **2022-03-18 19:07**, date raised: 2022-03-18
- **Version 1.2.4:** resolved Issue 0999: **pypi** (extension)
  - description: improved versioning
  - date resolved: **2022-03-18 18:28**, date raised: 2022-03-18

- **Version 1.2.3:** resolved Issue 0998: **pypi** (extension)
  - description: add description and tags
  - date resolved: **2022-03-18 18:09**, date raised: 2022-03-18
- **Version 1.2.2:** resolved Issue 0997: **pre-release version** (extension)
  - description: add ".dev" tag in version and wheel name for pre-releases, allowing to distinguish on pypi for different versions
  - date resolved: **2022-03-18 17:02**, date raised: 2022-03-18
- **Version 1.2.1:** resolved Issue 0996: **pre-release** (extension)
  - description: allow pre-releases to be uploaded on pypi, fetched with "pip install exudyn -pre"
  - date resolved: **2022-03-18 15:52**, date raised: 2022-03-18
- **Version 1.2.0:** resolved Issue 0995: **add to pypi index** (extension)
  - description: add exudyn to pypi index; allows to use "pip install exudyn"
  - date resolved: **2022-03-18 10:44**, date raised: 2022-03-18
- **Version 1.1.177:** **resolved BUG 0994: ObjectContactFrictionCircleCable2D**
  - description: forces on circle added up twice, because weighting factor not considered
  - **notes: tested force on circle with static computation in belt drive**
  - date resolved: **2022-03-18 08:30**, date raised: 2022-03-18
- **Version 1.1.176:** resolved Issue 0880: **User function for ConnectorCoordinateVector** (extension)
  - description: add user function for constraint and for jacobian; test with double pendulum made of masses
  - date resolved: **2022-03-17 18:14**, date raised: 2022-01-24
- **Version 1.1.175:** **resolved BUG 0993: ObjectConnectorDistance**
  - description: activeConnector=False produces wrong jacobian
  - date resolved: **2022-03-17 17:52**, date raised: 2022-03-17
- **Version 1.1.174:** resolved Issue 0992: **PlotSensor** (extension)
  - description: add PlotSensorDefaults function which allows to set default values for all subsequent PlotSensor calls
  - **notes: use e.g. PlotSensorDefaults().fontSize=16 to change fontSize for all subsequent calls**
  - date resolved: **2022-03-17 17:28**, date raised: 2022-03-17
- **Version 1.1.173:** resolved Issue 0991: **velocityOffset** (extension)
  - description: add velocity offset for SpringDamper and TorsionalSpringDamper, allowing simple controllers using offset and velocityOffset in preStepUser functions
  - date resolved: **2022-03-16 23:51**, date raised: 2022-03-16
- **Version 1.1.172:** resolved Issue 0989: **convergence problems** (docu)
  - description: add specific section in theDoc - Exudyn Basics related to ways for resolving convergence problems
  - date resolved: **2022-03-16 18:59**, date raised: 2022-03-16
- **Version 1.1.171:** resolved Issue 0938: **ANCFCable2D** (extension)
  - description: add drawing function for forces in normal direction with factor
  - date resolved: **2022-03-15 20:33**, date raised: 2022-02-10

- **Version 1.1.170:** resolved Issue 0933: **RigidBody** (extension)
  - description: add accelerationLocal and angularAccelerationLocal output
  - **notes:** added to **ObjectRigidBody** and **ObjectRigidBody2D**
  - date resolved: **2022-03-15 20:14**, date raised: 2022-02-07
- **Version 1.1.169:** resolved Issue 0967: **ANCF Cable2D** (extension)
  - description: add OutputVariables RotationMatrix, Rotation, AngularVelocity(Local) and AngularAcceleration
  - **notes:** Acceleration, AngularVelocity and AngularAcceleration currently only implemented for **ANCF Cable2D** but not for **ALEANCF Cable2D**
  - date resolved: **2022-03-15 20:01**, date raised: 2022-03-03
- **Version 1.1.168:** resolved Issue 0966: **ANCF Cable2D** (extension)
  - description: add missing terms for OutputVariables and AccessFunctions to allow constraints that are at local position  $y \neq 0$
  - date resolved: **2022-03-15 19:54**, date raised: 2022-03-03
- **Version 1.1.167:** resolved Issue 0983: **plot** (extension)
  - description: add function to create plot-ready data from mbs.GetObjectOutputBody for a list of consecutive beams, e.g., axial force, displacement or curvature along axial reference coordinate
  - **notes:** added function **DataArrayFromSensorList** which allows to create data from a list of sensors
  - date resolved: **2022-03-14 20:41**, date raised: 2022-03-11
- **Version 1.1.166:** resolved Issue 0982: **PlotSensor** (extension)
  - description: add option to plot 2D arrays
  - **notes:** numpy arrays are used instead of sensorNumbers; these arrays must have the same format as data stored in sensor files; this data format does not create any labels
  - date resolved: **2022-03-14 16:24**, date raised: 2022-03-11
- **Version 1.1.165:** resolved Issue 0985: **startOfStepState** (extension)
  - description: initialize startOfStepState together with currentState in InitializeSolverInitialConditions(...) in order to be valid when sensors are written in initialization
  - date resolved: **2022-03-14 13:07**, date raised: 2022-03-14
- **Version 1.1.164:** resolved Issue 0981: **ObjectContactFrictionCircleCable2D** (extension)
  - description: add OutputVariable functions for Coordinates (gap, slip), Coordinates\_t (gap\_t, slip\_t), and contact and friction forces per segment (ForceLocal)
  - date resolved: **2022-03-14 12:58**, date raised: 2022-03-11
- **Version 1.1.163:** resolved Issue 0980: **renderer precision** (extension)
  - description: add options for general precision in renderer: general.rendererPrecision as well as precision for colorbars in contour: contour.colorBarPrecision
  - date resolved: **2022-03-11 14:05**, date raised: 2022-03-11
- **Version 1.1.162:** resolved Issue 0979: **VisualizationSettings** (extension)
  - description: VisualizationSettings.showContactForcesValues is added to show numerical values for contact forces
  - date resolved: **2022-03-11 10:19**, date raised: 2022-03-11
- **Version 1.1.161:** resolved Issue 0978: **ObjectANCF Cable2D** (extension)

- description: add improved axial strain computation for reduced order integration
- **notes: works for axial strain and axial force if reducedAxialInterpolation=True**
- date resolved: **2022-03-10 20:36**, date raised: 2022-03-10
- **Version 1.1.160:** resolved Issue 0977: **ObjectANCF Cable2D** (extension)
  - description: add new mode useReducedOrderIntegration=2 with good performance/accuracy and exceptional representation of axial strains
  - date resolved: **2022-03-10 20:08**, date raised: 2022-03-10
- **Version 1.1.159:** resolved Issue 0976: **ContactCircleCable2D** (extension)
  - description: add visualization flag showContactCircle; uses circleTiling\*4 for tiling (from VisualizationSettings.general)!
  - date resolved: **2022-03-10 14:15**, date raised: 2022-03-10
- **Version 1.1.158:** resolved Issue 0975: **VisualizationSettings** (extension)
  - description: added showContactForces and contactForcesFactor in contact; this flag is currently only available in ContactCircleCable2D
  - date resolved: **2022-03-10 13:52**, date raised: 2022-03-10
- **Version 1.1.157:** resolved Issue 0974: **VisualizationSettings** (change)
  - description: moved contactPointsDefaultSize from connectors to contact; connectors.contactPointsDefaultSize is inactive from now!
  - date resolved: **2022-03-10 13:50**, date raised: 2022-03-10
- **Version 1.1.156:** resolved Issue 0953: **ContactFrictionCircleCable2D** (extension)
  - description: add improved (static) friction model
  - **notes: also added improved PostNewton switching strategies and changed initial values for NodeGenericData**
  - date resolved: **2022-03-09 21:42**, date raised: 2022-02-25
- **Version 1.1.155:** resolved Issue 0932: **ANCF Cable2D** (extension)
  - description: add velocityLocal + accelerationLocal output in axial/normal direction
  - **notes: only velocityLocal added**
  - date resolved: **2022-03-06 18:24**, date raised: 2022-02-07
- **Version 1.1.154:** resolved Issue 0931: **ANCF Cable2D** (extension)
  - description: add acceleration as output variable
  - **notes: only added for ANCF, but not for ALEANCF due to coupling terms with vALE**
  - date resolved: **2022-03-06 18:24**, date raised: 2022-02-07
- **Version 1.1.153:** resolved Issue 0970: **PlotSensor** (extension)
  - description: PlotSensor allows to set fileCommentChar and fileDelimiterChar
  - date resolved: **2022-03-04 13:27**, date raised: 2022-03-04
- **Version 1.1.152:** resolved Issue 0969: **exudyn.plot** (extension)
  - description: added method to convert output files from other codes; in particular plot.FileStripSpaces(...) can be used to strip leading / trailing spaces and remove double spaces
  - date resolved: **2022-03-04 13:27**, date raised: 2022-03-04

- **Version 1.1.151:** resolved Issue 0965: **ANCFCable2D** (extension)
  - description: add option for strainIsRelativeToReference, which if set to 1. accounts for the reference geometry as the stress-free configuration; also works for ALE Cable2D
  - date resolved: **2022-03-03 08:13**, date raised: 2022-03-03
- **Version 1.1.150:** resolved Issue 0964: **CreateReevingCurve** (change)
  - description: corrected sign of returned curvatures, now can be directly used for beam elements
  - date resolved: **2022-03-02 17:20**, date raised: 2022-03-02
- **Version 1.1.149:** resolved **BUG 0963: CreateReevingCurve**
  - description: removeFirstLine=True not working: wrong case i==0 needs to be changed to i==1
  - date resolved: **2022-03-02 15:25**, date raised: 2022-03-02
- **Version 1.1.148:** resolved Issue 0962: **CreateReevingCurve** (change)
  - description: adjust number of nodes in case of closed Curve (numberOfANCFnodes=20 gives 20 elements in this case)
  - date resolved: **2022-03-02 15:25**, date raised: 2022-03-02
- **Version 1.1.147:** resolved Issue 0961: **stepInformation** (change)
  - description: changed flags in timeIntegration and staticSolver stepInformation: value of 1024 now causes output at every step; all values > 16 have been divided by two; 255=detailed overall output, 2047=detailed output every step; see SimulationSettings in theDoc
  - date resolved: **2022-03-02 10:33**, date raised: 2022-03-02
- **Version 1.1.146:** resolved Issue 0959: **coordinatesSolution** (change)
  - description: change ending of coordinates solution files from .txt to .sol in case of binary files
  - date resolved: **2022-03-02 10:14**, date raised: 2022-03-02
- **Version 1.1.145:** resolved Issue 0960: **ObjectContactFrictionCircleCable2D** (change)
  - description: move to ObjectContactFrictionCircleCable2DOld and improve new version
  - date resolved: **2022-03-02 09:50**, date raised: 2022-03-02
- **Version 1.1.144:** resolved Issue 0958: **LoadForceVector** (docu)
  - description: add note to description in forces that user function values are available in sensors, but they are not updated during drawing
  - date resolved: **2022-03-01 19:40**, date raised: 2022-02-28
- **Version 1.1.143:** resolved Issue 0952: **rolling joints** (extension)
  - description: ConnectorRollingDiscPenalty and JointRollingDisc now add a OutputVariable RotationMatrix, containing the J1 to global transformation
  - date resolved: **2022-03-01 19:35**, date raised: 2022-02-25
- **Version 1.1.142:** resolved Issue 0951: **Friction** (test)
  - description: test LuGre friction model as ODE1 model versus position/history based model
  - notes: added **lugreFrictionODE1.py** as a demo showing the LuGre model based on a ODE1 user function
  - date resolved: **2022-03-01 19:34**, date raised: 2022-02-24
- **Version 1.1.141:** resolved Issue 0956: **ProfileLinearAccelerationsList** (extension)

- description: add linear acceleration profile to robotics.motion, currently only allowing to create profile directly from accelerations
- date resolved: **2022-02-28 19:05**, date raised: 2022-02-28
- **Version 1.1.140:** resolved Issue 0955: **robotics.motion** (change)
  - description: change PTPprofile to BasicProfile
  - date resolved: **2022-02-28 09:59**, date raised: 2022-02-28
- **Version 1.1.139:** **resolved BUG 0950: JointRollingDisc**
  - description: OutputVariable VelocityLocal is returned in global coordinates; description in theDoc is wrong; will be changed to outputVariable Velocity; local velocity will represent the slippage in special local joint J1 coordinates; see theDoc for updated functionality and outputs
  - date resolved: **2022-02-25 14:32**, date raised: 2022-02-22
- **Version 1.1.138:** **resolved BUG 0949: ConnectorRollingDiscPenalty**
  - description: OutputVariable VelocityLocal local is returned in global coordinates; description in theDoc is wrong; will be changed to outputVariable Velocity; see theDoc for updated functionality and outputs
  - date resolved: **2022-02-25 14:32**, date raised: 2022-02-22
- **Version 1.1.137:** resolved Issue 0946: **sensors storeInternal** (extension)
  - description: adapt most TestModels to store sensordata internally
  - date resolved: **2022-02-21 00:43**, date raised: 2022-02-20
- **Version 1.1.136:** resolved Issue 0947: **TestModels** (change)
  - description: change most test models to use Sensors storeInternal mode; this avoids creating many files during TestSuite runs
  - date resolved: **2022-02-20 20:40**, date raised: 2022-02-20
- **Version 1.1.135:** resolved Issue 0921: **PlotSensor** (extension)
  - description: add option to add subplots
  - notes: **allows to create subplots, adjusting also the plot size using sizeInches; for examples see Examples/plot-SensorExamples.py**
  - date resolved: **2022-02-19 22:45**, date raised: 2022-02-02
- **Version 1.1.134:** resolved Issue 0922: **PlotSensor** (extension)
  - description: add option to add linewidth, markersize, markerStyles=["o",...], markerSizes=[], lineStyles=["-",...], colors=[..], markerDensity=...; lineWidths=[] allowing to plot a reduced number of markers on top of a line
  - notes: **added many options for line and marker styles, check: colors, lineStyles, lineWidths, markerStyles, markerSizes, markerDensity**
  - date resolved: **2022-02-19 22:41**, date raised: 2022-02-02
- **Version 1.1.133:** resolved Issue 0920: **PlotSensor** (extension)
  - description: add x-range and y-range for zoom
  - notes: **added rangeX and rangeY options to specify range**
  - date resolved: **2022-02-19 16:59**, date raised: 2022-02-02
- **Version 1.1.132:** resolved Issue 0945: **PlotSensor** (extension)
  - description: extend option for offsets, allowing sensor data to use as offset (e.g., loaded from file or from internal sensor data)

- notes: see `plotSensorExamples` for some particular usage
- date resolved: **2022-02-19 16:45**, date raised: 2022-02-19
- **Version 1.1.131:** resolved Issue 0944: `PlotSensor` (extension)
  - description: add argument labels, which can be string (for one sensor) or list of strings (according to number of sensors resp. components) representing the labels used in legend; if not provided, automatically generated legend is used
  - date resolved: **2022-02-19 15:47**, date raised: 2022-02-19
- **Version 1.1.130:** resolved Issue 0918: `Sensors store values internally` (extension)
  - description: add option to store sensor values internally; using ResizableMatrix internally; rows added and matrix is automatically resized
  - notes: **storeInternal boost the speed of writing sensor values, however, most of time is spent on computing sensor values, which typically about 2 seconds for 1e6 time steps per sensor; file write adds usually 3 seconds extra on that bill**
  - date resolved: **2022-02-19 00:13**, date raised: 2022-02-02
- **Version 1.1.129:** resolved Issue 0943: `mbs.GetSensorStoredData()` (extension)
  - description: add MainSystem functionality to retrieve internally stored data in sensor; used, e.g., for PlotSensor to plot data without storing in files
  - date resolved: **2022-02-19 00:10**, date raised: 2022-02-18
- **Version 1.1.128: resolved BUG 0942: `sensorsAppendToFile`**
  - description: appendToFile is used instead of sensorsAppendToFile for switching between append and replace operations for files
  - date resolved: **2022-02-18 20:54**, date raised: 2022-02-18
- **Version 1.1.127:** resolved Issue 0901: `SimulationSettings` (extension)
  - description: improve type completion by adding `py::init<...>` functions for all subclasses
  - notes: **not solvable in this simple way as type completion is not improved when adding this information**
  - date resolved: **2022-02-18 20:30**, date raised: 2022-01-31
- **Version 1.1.126:** resolved Issue 0941: `MotionInterpolator` (extension)
  - description: mark as deprecated; instead, created motion submodule in robotics with class Trajectory; uses classes ProfileConstantAcceleration and ProfilePTP to construct piecewise profiles for trajectory; precomputes acceleration profiles in first step and thus is several factors faster in Python implementation; Trajectory converts to dict and can be printed in order to obtain key values of computed trajectories
  - date resolved: **2022-02-18 16:31**, date raised: 2022-02-15
- **Version 1.1.125:** resolved Issue 0940: `MotionInterpolator` (extension)
  - description: add option to add trajectories defined by duration as well as by maxVelocity and maxAcceleration using synchronous PTP trajectory generation
  - date resolved: **2022-02-15 12:38**, date raised: 2022-02-15
- **Version 1.1.124: resolved BUG 0939: `NodeRigidBody2D`**
  - description: does not correctly measure rotations (returns x-coordinate instead of angle)
  - date resolved: **2022-02-15 09:58**, date raised: 2022-02-15
- **Version 1.1.123:** resolved Issue 0745: `SensitivityAnalysis()` (extension)

- description: add functionality to processing, evaluating the sensitivities of certain sensor values w.r.t. parameters; same interface as ParameterVariation
- **notes:** see `processing.ComputeSensitivities(...)`
- date resolved: **2022-02-14 09:50**, date raised: 2021-09-03 (resolved by: Manzl Peter)
- **Version 1.1.122:** resolved **BUG 0934:** `exudyn.signal`
  - description: conflicts with Python 3.8.8 and Python 3.9.7 (and possibly other) with internal Python signal package; change `exudyn.signal` to `exudyn.signalProcessing`
  - date resolved: **2022-02-10 09:23**, date raised: 2022-02-10
- **Version 1.1.121:** resolved Issue 0870: **binary output** (extension)
  - description: add option to create binary solution files
  - **notes:** notes: use `outputPrecision` to switch between float and double - see there; speeds up file writing considerably and reduces file sizes; Integrated into `LoadSolutionFile`
  - date resolved: **2022-02-09 08:45**, date raised: 2022-01-18
- **Version 1.1.120:** resolved Issue 0930: **LoadSolutionFile** (extension)
  - description: extend function to check if binary file; if yes, switches to binary mode
  - date resolved: **2022-02-09 08:44**, date raised: 2022-02-07
- **Version 1.1.119:** resolved Issue 0929: **coordinatesSolution, sensors** (change)
  - description: add version to coordinatesSolutionFile and sensor output files; should not affect current parsing of output files
  - date resolved: **2022-02-07 00:02**, date raised: 2022-02-07
- **Version 1.1.118:** resolved Issue 0878: **sort settings options** (docu)
  - description: sort settings representation in Python by sorting dictionaries prior to writing interface files; keep current sorting (grouping) in latex documentation
  - **notes:** type completion may change sorting afterwards
  - date resolved: **2022-02-06 21:57**, date raised: 2022-01-24
- **Version 1.1.117:** resolved Issue 0928: **CPP UNIT TESTS** (testing)
  - description: fail because of change of `ConstSizeVector` to use move assignment and move constructor
  - **notes:** adapted test to avoid move assignment
  - date resolved: **2022-02-06 00:04**, date raised: 2022-02-06
- **Version 1.1.116:** resolved Issue 0919: **Minimize** (extension)
  - description: add optimization with same interface as `GenticOptimization` but based on `scipy.optimize.minimize`
  - **notes:** added Minimize to processing; usage is nearly same as `GeneticOptimization(...)`; example under Examples/minimizeExample.py
  - date resolved: **2022-02-04 15:45**, date raised: 2022-02-02 (resolved by: Stefan Holzinger)
- **Version 1.1.115:** resolved **BUG 0924:** **GeneralContact**
  - description: WARNING message raised in ANCF contact
  - date resolved: **2022-02-03 12:01**, date raised: 2022-02-03
- **Version 1.1.114:** resolved Issue 0923: **CreateReevingCurve** (extension)

- description: add function CreateReevingCurve(...) in exudyn.beams to create reeving system along circles, allows to create curve and nodes for ANCF Cable2D elements created with PointsAndSlopes2ANCF Cable2D(...); see Examples/reevingSystem.py
- date resolved: **2022-02-03 12:00**, date raised: 2022-02-02
- **Version 1.1.113:** resolved Issue 0803: **PostNewton** (check)
  - description: Check discontinuous iterations in combination with adaptive step (immediately reduces step size even if ignoreMaxSteps=True)
  - **notes: did not further show up; possibly due to non-convergence**
  - date resolved: **2022-02-02 08:20**, date raised: 2021-11-25
- **Version 1.1.112:** resolved Issue 0906: **PlotSensor** (extension)
  - description: add option componentsX to add x-components for figures, e.g., to plot y over x position, instead over time
  - date resolved: **2022-02-01 18:34**, date raised: 2022-02-01
- **Version 1.1.111:** resolved Issue 0864: **automatic example referencing** (fix)
  - description: fix searching for examples, e.g., NodePoint as NodePoint() in order no to find NodePoint2D examples
  - date resolved: **2022-01-31 23:01**, date raised: 2022-01-14
- **Version 1.1.110:** **resolved BUG 0903: GeneralContact**
  - description: autocomputed searchTree gives very large values and visualization not working
  - **notes: ANCF Cable2D bounding box computation had 1 wrong else case**
  - date resolved: **2022-01-31 21:34**, date raised: 2022-01-31
- **Version 1.1.109:** resolved Issue 0899: **GenerateCircularArcANCF Cable2D** (extension)
  - description: add function to create beams along circular arc
  - date resolved: **2022-01-31 14:20**, date raised: 2022-01-30
- **Version 1.1.108:** **resolved BUG 0902: GenerateStraightLineANCF Cable2D**
  - description: cableNodePositionList does not returns 3D vectors for nodes except first node
  - date resolved: **2022-01-31 14:19**, date raised: 2022-01-31
- **Version 1.1.107:** resolved Issue 0898: **GenerateStraightLineANCF Cable2D** (extension)
  - description: add option to use existing nodes in generation of beams
  - date resolved: **2022-01-31 10:11**, date raised: 2022-01-30
- **Version 1.1.106:** resolved Issue 0897: **add Python utility beams** (extension)
  - description: add exudyn.beams utility module, containing helper functions for creating beams, etc.; move existing functions to this module
  - date resolved: **2022-01-31 10:11**, date raised: 2022-01-30
- **Version 1.1.105:** **resolved BUG 0900: GenerateStraightLineANCF Cable2D**
  - description: arguments vALE and ConstrainAleCoordinate are not implemented and need to be removed
  - date resolved: **2022-01-30 23:32**, date raised: 2022-01-30
- **Version 1.1.104:** resolved Issue 0896: **theDoc objects** (docu)
  - description: sort objects into bodies, basic connectors, constraints and joints

- date resolved: **2022-01-30 23:12**, date raised: 2022-01-30
- **Version 1.1.103:** resolved Issue 0895: **ConnectorGravity** (extension)
  - description: add connector representing gravitational forces between heavy masses (planet, satellite, etc.)
  - date resolved: **2022-01-30 19:05**, date raised: 2022-01-30
- **Version 1.1.102:** resolved Issue 0894: **colors** (extension)
  - description: added several colors in graphicsDataUtilities, added color4black, extended color4list to 16 colors
  - date resolved: **2022-01-30 18:10**, date raised: 2022-01-30
- **Version 1.1.101:** resolved Issue 0893: **GeneralContact** (change)
  - description: changed flag introSpheresContact to sphereSphereContact; added flag for special mode to recycle last Post Newton step friction force
  - date resolved: **2022-01-28 18:55**, date raised: 2022-01-28
- **Version 1.1.100:** resolved Issue 0891: **adapt to Python3.9** (extension)
  - description: make wheels and installers for Python3.9, running on Anaconda3-2021-11 Windows-x86\_64
  - date resolved: **2022-01-25 23:31**, date raised: 2022-01-25
- **Version 1.1.99:** resolved Issue 0889: **PlotSensor** (testing)
  - description: add test for PlotSensor, returning 1 if all plotting runs without crashing
  - date resolved: **2022-01-25 19:03**, date raised: 2022-01-25
- **Version 1.1.98:** resolved Issue 0890: **PlotSensor** (extension)
  - description: add possibility to use filenames instead of sensor numbers, automatically loading these files
  - date resolved: **2022-01-25 18:00**, date raised: 2022-01-25
- **Version 1.1.97:** **resolved BUG 0887: iPython output stops**
  - description: after a solver error, output of iPython stops
  - **notes:** **changed to correct catching/throwing of Python and C++ exception types; output continues after solver error**
  - date resolved: **2022-01-25 14:25**, date raised: 2022-01-25
- **Version 1.1.96:** resolved Issue 0885: **GetInterpolatedSignalValue** (extension)
  - description: add check in case that time values are distributed non-uniform; add tolerance as option
  - date resolved: **2022-01-25 12:10**, date raised: 2022-01-25
- **Version 1.1.95:** resolved Issue 0884: **PlotSensor** (extension)
  - description: add optional title to plot
  - date resolved: **2022-01-25 11:52**, date raised: 2022-01-25
- **Version 1.1.94:** resolved Issue 0883: **PlotSensor** (extension)
  - description: if sensorNumbers is scalar, components is a list, sensorNumbers is automatically adjusted; accepts now e.g. PlotSensor(mbs, 0, components=[0,1,2])
  - date resolved: **2022-01-25 11:36**, date raised: 2022-01-25
- **Version 1.1.93:** resolved Issue 0881: **PlotSensor** (extension)
  - description: add option to apply factors and offsets to plotted signals; add option to add labels which appears at legend

- date resolved: **2022-01-25 11:36**, date raised: 2022-01-25
- **Version 1.1.92:** resolved Issue 0882: **PlotSensor** (change)
  - description: add option to use X, Y and Z components instead of 0, 1, 2 for Position, Displacement etc.; this option is enabled by default and changes the appearance -> set False, to preserve the old mode; component now also shown if only one curve plotted
  - date resolved: **2022-01-25 11:19**, date raised: 2022-01-25
- **Version 1.1.91:** resolved Issue 0879: **LoadSolutionFile** (extension)
  - description: changed safeMode to loading single lines, which saves memory enormously, and added new options for loading huge files
  - date resolved: **2022-01-24 14:53**, date raised: 2022-01-24
- **Version 1.1.90:** resolved **BUG 0877: AddSensorRecorder**
  - description: fails for scalar Sensors OutputVariableTypes (e.g. when measuring Rotation of TorsionalSpring-Damper)
  - notes: added special scalar case
  - date resolved: **2022-01-21 09:17**, date raised: 2022-01-21
- **Version 1.1.89:** resolved Issue 0876: **flush files** (extension)
  - description: add option to flush solution and sensor files immediately after writing, simplifying the readout process; add option for large scale simulations, which are always flushed - helping for continuation of computations on supercomputers
  - date resolved: **2022-01-20 14:01**, date raised: 2022-01-20
- **Version 1.1.88:** resolved Issue 0875: **PlotSensor** (extension)
  - description: fixed fontSize option and add options for minor/major ticks and SAVE figure to PlotSensor(...) using fileName=...
  - date resolved: **2022-01-19 11:08**, date raised: 2022-01-19
- **Version 1.1.87:** resolved Issue 0800: **GeneralContact ANCF Cable** (extension)
  - description: add ANCF Cable 2D to GeneralContact, enabling contact with planar spheres (cylinders)
  - date resolved: **2022-01-18 18:54**, date raised: 2021-11-19
- **Version 1.1.86:** resolved Issue 0866: **numberOfThreads** (extension)
  - description: move simulationSettings.numberOfThreads into new section parallel in simulationSettings; remove comment [not implemented]
  - date resolved: **2022-01-18 17:43**, date raised: 2022-01-15
- **Version 1.1.85:** resolved Issue 0872: **preStepPyExecute** (change)
  - description: remove preStepPyExecute from docu, time integration / static solver interface and from CSolverBase
  - date resolved: **2022-01-18 16:57**, date raised: 2022-01-18
- **Version 1.1.84:** resolved Issue 0874: **improved Newton restart** (change)
  - description: added an additional Newton iteration after restarting modified Newton or when switching to full Newton; this reduces effects in generalized alpha and may improve behaviour with severe nonlinearities
  - date resolved: **2022-01-18 13:44**, date raised: 2022-01-18
- **Version 1.1.83:** resolved Issue 0869: **adaptiveStepRecoveryIterations** (change)

- description: add option to static and dynamic solvers to adjust max. Newton+disc. iterations prior to increase of step size; changed (previous internal) default value from 5 to 7
- date resolved: **2022-01-17 19:54**, date raised: 2022-01-17
- **Version 1.1.82:** resolved Issue 0868: **solverSettings.stepInformation** (extension)
  - description: change modes to add up binary flags; ADD several new options to show Newton iterations, jacobians, discontinuous iterations, per step or period; also add option to show output at every step
  - date resolved: **2022-01-17 12:11**, date raised: 2022-01-17
- **Version 1.1.81:** resolved Issue 0857: **GetInterpolatedSignalValue** (extension)
  - description: new function to interpolate a numeric signal with time/data vectors at a certain time point
  - date resolved: **2022-01-11 15:41**, date raised: 2022-01-11
- **Version 1.1.80:** resolved Issue 0856: **IndexFromValue** (extension)
  - description: function got faster mode in case of constant sampling rate
  - date resolved: **2022-01-11 14:39**, date raised: 2022-01-11
- **Version 1.1.79:** resolved Issue 0854: **Add LTG description** (docu)
  - description: add section on local-to-global mapping of coordinates [Section 2.3](#)
  - date resolved: **2022-01-08 12:09**, date raised: 2022-01-08
- **Version 1.1.78:** resolved Issue 0849: **publications directory** (docu)
  - description: create separate directory Examples/publications/ for publication data, Python files of numerical examples, etc.
  - date resolved: **2022-01-06 16:32**, date raised: 2022-01-06
- **Version 1.1.77:** resolved Issue 0846: **analytic jacobians** (extension)
  - description: add analytic jacobians general functionality, realized for CartesianSpringDamper and Coordinate-SpringDamper; deactivate with newton.numericalDifferentiation.forODE2connectors = False
  - date resolved: **2021-12-23 11:07**, date raised: 2021-12-23
- **Version 1.1.76:** resolved Issue 0770: **Marker jacobian derivative2** (extension)
  - description: add jacobian derivative to most important connector markers
  - notes: implemented for markers except MarkerSuperElement; only MarkerPosition and MarkerRigidBody affected mostly; first tests show that jacobianDerivative agrees with numerical differentiation, BUT even increases iteration numbers
  - date resolved: **2021-12-22 21:21**, date raised: 2021-09-28
- **Version 1.1.75:** resolved Issue 0842: **implement AccessFunctionType::JacobianTtimesVector\_q** (extension)
  - description: implement function needed for MarkerBody for objects ANCF Cable, ObjectFFRF and ObjectFFRFReducedOrder; currently raising exception if used in this setup!
  - date resolved: **2021-12-22 21:18**, date raised: 2021-12-22
- **Version 1.1.74:** resolved Issue 0831: **Explicit solvers** (change)
  - description: remove second ComputeODE2Acceleration() call and copy solutionODE2\_tt from beginning of time step rk.stageDerivODE2\_t[0], same for ODE1 variables; add flag but change that by default as it speeds up 2x; could also use information from previous step ...?
  - date resolved: **2021-12-20 13:09**, date raised: 2021-12-15

- **Version 1.1.73:** resolved Issue 0832: **explicit integrator** (extension)
  - description: add flag timeintegration.explicitIntegration.computeEndOfStepAccelerations to compute end-of-step accelerations; this computation doubles the effort of explicit one-step-methods, particularly relevant in particles or contact simulations
  - date resolved: **2021-12-17 12:38**, date raised: 2021-12-17
- **Version 1.1.72:** resolved Issue 0450: **MT integration** (extension)
  - description: fully integrate multithreading into system.cpp, vector.cpp and dense solver
  - **notes:** integrated into system, missing mass matrix and jacobian
  - date resolved: **2021-12-09 18:17**, date raised: 2020-09-16
- **Version 1.1.71:** resolved Issue 0799: **GeneralContact description** (docu)
  - description: add general section in theDoc for description of GeneralContact
  - date resolved: **2021-12-09 12:36**, date raised: 2021-11-19
- **Version 1.1.70:** resolved Issue 0793: **performance section** (docu)
  - description: add performance and speedup section to theDoc, explaining most useful settings like modifiedNewton, EigenSparse, writeToFile, step size, numberOfThreads, constant mass matrix, etc.
  - date resolved: **2021-12-09 12:36**, date raised: 2021-11-02
- **Version 1.1.69:** resolved Issue 0823: **add trig-sphere friction tests** (extension)
  - description: add simple test cases to check friction implementation
  - date resolved: **2021-12-09 08:28**, date raised: 2021-12-06
- **Version 1.1.68:** resolved Issue 0822: **shrink mesh** (extension)
  - description: add method to shrink meshes using max distance to surface with normals; used for contact trig-sphere implementation
  - **notes:** currently slows down for larger meshes due to elimination of duplicate points!
  - date resolved: **2021-12-09 08:28**, date raised: 2021-12-06
- **Version 1.1.67:** resolved **BUG 0827:** **GraphicsData cube**
  - description: cubes has wrong numbering of nodes
  - **notes:** FIXED numbering in order to allow for correct normals needed in contact computation
  - date resolved: **2021-12-07 17:14**, date raised: 2021-12-07
- **Version 1.1.66:** resolved Issue 0826: **GraphicsData normals** (extension)
  - description: add normals to some GraphicsData cube objects
  - date resolved: **2021-12-07 16:27**, date raised: 2021-12-07
- **Version 1.1.65:** resolved Issue 0825: **GraphicsData add defaults** (extension)
  - description: add some default values, especially to (center)point of GraphicsDataSphere, GraphicsDataCylinder, GraphicsDataOrthoCube, etc. in order to reduce interface sizes for objects added in the centerpoint [0,0,0]; thus some defaults were also necessary for radius or sizes!
  - date resolved: **2021-12-07 14:41**, date raised: 2021-12-07
- **Version 1.1.64:** resolved Issue 0824: **PlotSensor** (change)
  - description: added serval NEW options, including: newFigure, colorCodeOffset, figureName and closeAll; NOTE that now PlotSensor by default opens a new figure!

- date resolved: **2021-12-07 12:11**, date raised: 2021-12-07
- **Version 1.1.63: resolved BUG 0820: RigidBody visualization**
  - description: normals in rigid body visualization transformed wrongly
  - **notes: before, rotating bodies may have shown shading, now resolved**
  - date resolved: **2021-12-05 17:32**, date raised: 2021-12-05
- **Version 1.1.62: resolved Issue 0816: GraphicsData convert (extension)**
  - description: add function GraphicsData2TrigsAndPoints(...) to convert graphicsData into triangles and points
  - date resolved: **2021-12-05 15:43**, date raised: 2021-12-02
- **Version 1.1.61: resolved Issue 0814: robotics.future, robotics.utilities, robotics.mobile (extension)**
  - description: add special submodules for robotics functions; future contains currently developed submodules that will be available in future at a different location in robotics
  - date resolved: **2021-12-05 15:43**, date raised: 2021-12-02
- **Version 1.1.60: resolved Issue 0819: ObjectRigidBody (change)**
  - description: correct HasConstantMassMatrix for Lie group nodes and COM=0
  - date resolved: **2021-12-05 11:19**, date raised: 2021-12-05
- **Version 1.1.59: resolved Issue 0804: GeneralContact TriangleMesh (extension)**
  - description: Add rigid-body-marker based triangle mesh to GeneralContact
  - date resolved: **2021-12-04 10:08**, date raised: 2021-11-25
- **Version 1.1.58: resolved Issue 0818: MergeGraphicsDataTriangleList (extension)**
  - description: now works if either both lists contain normals or both do not
  - date resolved: **2021-12-03 20:15**, date raised: 2021-12-03
- **Version 1.1.57: resolved Issue 0817: NodePointGround (extension)**
  - description: add Node::Orientation to NodeType, such that it can also be used as a rigidBody node
  - date resolved: **2021-12-03 14:34**, date raised: 2021-12-03
- **Version 1.1.56: resolved Issue 0815: tCPU showing wrong time (change)**
  - description: minor bug; time shown is since starting of iPython
  - date resolved: **2021-12-02 17:48**, date raised: 2021-12-02
- **Version 1.1.55: resolved Issue 0813: exudyn.robotics.special (change)**
  - description: extend and move roboticsSpecial to robotics.special; add special robotics functionality like manipulability
  - date resolved: **2021-12-02 11:03**, date raised: 2021-12-02 (resolved by: Martin Sereinig)
- **Version 1.1.54: resolved Issue 0618: robotics submodule (extension)**
  - description: Create robotics.special and robotics.mecanum or robotics.ros submodules with subdirectories
  - date resolved: **2021-12-02 11:02**, date raised: 2021-03-23
- **Version 1.1.53: resolved Issue 0812: Suppress warnings (extension)**
  - description: add flag to globally suppress warnings
  - **notes: use exudyn.SuppressWarnings(True) to turn off warnings**

- date resolved: **2021-12-01 08:50**, date raised: 2021-12-01
- **Version 1.1.52:** resolved Issue 0811: **add default constructors** (change)
  - description: add rule of five default constructors to SlimVector, SlimArray, ConstSizeVector and ConstSizeMatrix; remove mutable from data
  - date resolved: **2021-11-28 21:56**, date raised: 2021-11-28
- **Version 1.1.51:** resolved Issue 0810: **GeneralContact visualization** (extension)
  - description: add visualization for searchtree (box) and bounding boxes
  - **notes:** use SC.visualizationSettings.contact to adjust the various options to visualize the contact search tree
  - date resolved: **2021-11-26 23:20**, date raised: 2021-11-26
- **Version 1.1.50:** resolved Issue 0809: **ParameterVariation** (extension)
  - description: Added a additional argument parameterFunctionData= to function ParameterVariation(). The argument parameterFunctionData can be used to make global data available inside the parameterFunction.
  - date resolved: **2021-11-26 12:57**, date raised: 2021-11-26 (resolved by: Stefan Holzinger)
- **Version 1.1.49:** resolved Issue 0808: **Performance test for GeneralContact** (testint)
  - description: added test with sphere contact
  - date resolved: **2021-11-26 10:49**, date raised: 2021-11-26
- **Version 1.1.48:** resolved Issue 0802: **GeneralContact** (testing)
  - description: add TestModel for Sphere-Sphere GeneralContact
  - date resolved: **2021-11-25 22:58**, date raised: 2021-11-25
- **Version 1.1.47:** resolved Issue 0807: **GeneralContact** (change)
  - description: added new functions to initialize searchTree, searchTreeBox and frictionPairings which were previously in FinalizeContact(...)
  - date resolved: **2021-11-25 22:16**, date raised: 2021-11-25
- **Version 1.1.46:** resolved Issue 0806: **removed FinalizeContact** (change)
  - description: removed this function, which is now automatically called in mbs.Assemble()
  - date resolved: **2021-11-25 22:15**, date raised: 2021-11-25
- **Version 1.1.45:** resolved Issue 0805: **AssembleSystemInitialize** (extension)
  - description: add additional function inside mbs.Assemble() to initialize GeneralContact
  - date resolved: **2021-11-25 22:14**, date raised: 2021-11-25
- **Version 1.1.44:** resolved Issue 0779: **sensor recorder** (extension)
  - description: add utilities function for recording signals internally in mbs; avoids writing to sensor files, which helps reducing overhead in ParameterVariation and GeneticOptimization
  - **notes:** available in Python utilities function AddSensorRecorder(...)
  - date resolved: **2021-11-25 11:10**, date raised: 2021-10-17
- **Version 1.1.43:** resolved Issue 0801: **Box3D** (change)
  - description: check Ubuntu20 warnings; replace Vector3D pmin with Real pmin[3] to avoid warnings and gain speedup
  - date resolved: **2021-11-25 11:08**, date raised: 2021-11-23

- **Version 1.1.42:** resolved Issue 0798: **Implicit GeneralContact** (extension)
  - description: add ODE2RHS jacobian and PostNewton to GeneralContact
  - date resolved: **2021-11-19 16:07**, date raised: 2021-11-19
- **Version 1.1.41:** resolved Issue 0787: **GeneralContact** (extension)
  - description: add contact object, directly in mbs, which allows different types of contact with efficient computation and search trees; start with simple spherical contact
  - date resolved: **2021-11-19 16:06**, date raised: 2021-11-01
- **Version 1.1.40:** resolved **BUG 0784: verboseMode**
  - description: output of every step in verboseMode=1 after long time or if there are some very long lasting steps
  - notes: **not fully clarified, but modified time when data is output; may occur in case of very long running iPython?**
  - date resolved: **2021-11-14 23:23**, date raised: 2021-10-31
- **Version 1.1.39:** resolved **BUG 0790: multithreading fails for user functions**
  - description: build separate lists for objects and loads with user functions, excluded in MT evaluation
  - date resolved: **2021-11-14 23:21**, date raised: 2021-11-02
- **Version 1.1.38:** resolved **BUG 0794: error when switching from n to 1 threads**
  - description: RuntimeError: TemporaryComputationdataArray::operator[]: index out of range caused when switching from numberOfThreads>1 to 1 thread
  - date resolved: **2021-11-13 23:59**, date raised: 2021-11-04
- **Version 1.1.37:** resolved Issue 0788: **multithreaded ODE2RHS and compute loads** (extension)
  - description: use multithreaded computation for ODE2 RHS and for loads computation; use simulationSettings.numberOfThreads > 1 for multithreaded computation
  - date resolved: **2021-11-13 23:59**, date raised: 2021-11-01
- **Version 1.1.36:** resolved Issue 0796: **GL list GeneralContact** (extension)
  - description: speed up visualization with GL list for spheres in GeneralContact
  - date resolved: **2021-11-13 23:03**, date raised: 2021-11-10
- **Version 1.1.35:** resolved Issue 0797: **itemInterface** (extension)
  - description: add representation for item interface classes, using `__repr__()` = `dict(self)`
  - date resolved: **2021-11-10 08:40**, date raised: 2021-11-10
- **Version 1.1.34:** resolved Issue 0789: **constant mass matrix** (extension)
  - description: do not recompute mass matrix if it is totally constant in implicit solver; add flag to solver options to force recompute
  - date resolved: **2021-11-08 10:30**, date raised: 2021-11-02
- **Version 1.1.33:** resolved Issue 0795: **add TCP/IP interface** (extension)
  - description: add CreateTCPIPconnection and other functions to utilities for interconnection with other programs via TCP/IP
  - date resolved: **2021-11-08 10:29**, date raised: 2021-11-08
- **Version 1.1.32:** resolved Issue 0786: **TemporaryComputationdataArray** (extension)

- description: add array of TemporaryCompData for multithreaded computation
- date resolved: **2021-11-01 23:01**, date raised: 2021-11-01
- **Version 1.1.31:** resolved Issue 0785: **ComputeSystemODE1RHS** (extension)
  - description: add list of loads with ODE1 relevancy, otherwise all loads are computed even if there are no ODE1 coordinates
  - **notes: added simple flag to avoid computation if no ODE1 coordinates available**
  - date resolved: **2021-11-01 21:28**, date raised: 2021-11-01
- **Version 1.1.30:** resolved Issue 0781: **add n-mass-oscillator** (example)
  - description: add interactive example based on simulateInteractively for n-mass-oscillator with step and frequency excitation
  - date resolved: **2021-10-28 16:44**, date raised: 2021-10-25
- **Version 1.1.29:** resolved Issue 0747: **ComputeLinearizedSystem** (extension)
  - description: add exodyn.ComputeLinearizedSystem similar to what is done in ComputeODEEigenvalues, returning M, K, D, ...
  - date resolved: **2021-10-27 18:40**, date raised: 2021-09-03
- **Version 1.1.28:** resolved Issue 0780: **enable close window button** (extension)
  - description: close window button is now enabled, which stops current and following simulations until render window is restarted or SetRenderEngineStopFlag(False)
  - **notes: if a simulation with renderer is quit (also ESCAPE button), then a further call to solver will be ignored until the simulation is reset, or SetRenderEngineStopFlag(False)**
  - date resolved: **2021-10-21 09:31**, date raised: 2021-10-21
- **Version 1.1.27:** resolved Issue 0778: **GenericODE2 FEM** (extension)
  - description: add FEMinterface function for creation of ObjectGenericODE2 with linear FEM model and nonlinear FEM model (using NGsolve)
  - date resolved: **2021-10-16 23:23**, date raised: 2021-10-16
- **Version 1.1.26:** **resolved BUG 0776: MatrixContainer::SetWithSparseMatrixCSR**
  - description: does not set number of columns and rows due to error in MatrixContainer::SetAllZero()
  - date resolved: **2021-10-09 16:53**, date raised: 2021-10-08
- **Version 1.1.25:** resolved Issue 0767: **sparse ObjectJacobianODE2** (extension)
  - description: add dense and sparse interface to ObjectJacobianODE2
  - date resolved: **2021-10-09 16:53**, date raised: 2021-09-27
- **Version 1.1.24:** **resolved BUG 0775: ObjectGenericODE2, ObjectFFRFreducedOrder**
  - description: visualization fails if outputVariable = None
  - date resolved: **2021-10-08 17:26**, date raised: 2021-10-08
- **Version 1.1.23:** resolved Issue 0773: **ImportMeshFromNGsolve** (extension)
  - description: added option meshOrder which allows to use second order elements with meshOrder=2, leading to much higher accuracy of displacements and stresses
  - date resolved: **2021-10-01 14:04**, date raised: 2021-10-01
- **Version 1.1.22:** resolved Issue 0774: **ComputePostProcessingModesNGsolve** (extension)

- description: added improved functionality for ComputePostProcessingModes using NGsolve, speeding up computations by factor of 10
- date resolved: **2021-10-01 14:03**, date raised: 2021-10-01
- **Version 1.1.21:** resolved Issue 0772: **compute HCB modes with NGsolve** (extension)
  - description: add much faster computation function ComputeHurtyCraigBamptonModesNGsolve for computation of eigenmodes
  - date resolved: **2021-10-01 14:03**, date raised: 2021-10-01
- **Version 1.1.20:** resolved Issue 0771: **GeneticOptimization** (extension)
  - description: add normal distribution and distanceFactorGenerations
  - date resolved: **2021-09-29 17:38**, date raised: 2021-09-29
- **Version 1.1.19:** resolved Issue 0769: **Marker jacobian derivative** (extension)
  - description: add jacobian derivative to markers to allow analytical differentiation of connectors
  - date resolved: **2021-09-28 18:57**, date raised: 2021-09-28
- **Version 1.1.18:** resolved Issue 0768: **Newton.useNumericalDifferentiation** (change)
  - description: change to Newton.numericalDifferentiation.forAE and Newton.numericalDifferentiation.forODE2; previous Newton.useNumericalDifferentiation only affected AE (algebraic equations) and should be changed now to Newton.numericalDifferentiation.forAE; default is False
  - date resolved: **2021-09-27 15:11**, date raised: 2021-09-27
- **Version 1.1.17:** resolved Issue 0612: **sparse object matrices** (extension)
  - description: add sparse matrix computation mode for ComputeMassMatrix and for ObjectJacobianODE2; consider Lie algebra derivatives
  - **notes:** **sparse ObjectJacobianODE2 computation not yet implemented and moved to issue767**
  - date resolved: **2021-09-27 14:35**, date raised: 2021-03-21
- **Version 1.1.16:** resolved Issue 0766: **ObjectGenericODE2** (change)
  - description: change mass matrix, stiffnessmatrix, etc. types to PyMatrixContainer in order to accept dense and sparse matrices
  - date resolved: **2021-09-27 14:32**, date raised: 2021-09-26
- **Version 1.1.15:** resolved Issue 0765: **describe Python types** (docu)
  - description: describe Python types such as NumpyMatrix or PyMatrixContainer in intro to objects, nodes, ...
  - date resolved: **2021-09-27 14:32**, date raised: 2021-09-26
- **Version 1.1.14:** resolved Issue 0764: **PyMatrixContainer** (extension)
  - description: extend PyMatrixContainer to accept numpy.array or list of lists as input
  - date resolved: **2021-09-26 23:35**, date raised: 2021-09-26
- **Version 1.1.13:** resolved Issue 0763: **ComputeMassMatrix** (extension)
  - description: add sparse mode with MatrixContainer
  - date resolved: **2021-09-26 18:01**, date raised: 2021-09-26
- **Version 1.1.12:** resolved Issue 0762: **MatrixBase** (performance)
  - description: remove virtual from begin/end operators
  - date resolved: **2021-09-26 17:36**, date raised: 2021-09-26

- **Version 1.1.11: resolved BUG 0750: ANCF/ALE contour plot**
  - description: contour plot not showing displacements or forces
  - **notes: bug due to issue 760, which has been resolved now!**
  - date resolved: **2021-09-24 08:48**, date raised: 2021-09-03
- **Version 1.1.10: resolved Issue 0761: LinkedDataVectorBase (extension)**
  - description: allowing SetNumberOfItems to make LinkedDataVectors smaller after linking
  - date resolved: **2021-09-24 08:47**, date raised: 2021-09-24
- **Version 1.1.9: resolved BUG 0760: contour plot**
  - description: nodes in contour plot leading to exudyn crash due to ConstSizeVector decoupled from Vector
  - date resolved: **2021-09-24 08:47**, date raised: 2021-09-24
- **Version 1.1.8: resolved Issue 0758: FEM CMSObjectComputeNorm (extension)**
  - description: add function into FEM to compute maximum stress / strain / etc for CMSObject (ObjectFFRFreduce-dOrder), using only the objectNumber as an input; options are outputVariableType=StressLocal, norm="" (Mises, L2norm, none), nodeNumbers=[] ... providing optional list of nodes to restrict the computation
  - date resolved: **2021-09-23 19:25**, date raised: 2021-09-22
- **Version 1.1.7: resolved Issue 0757: FEM GetNodePositionsMean (extension)**
  - description: add function into FEMinterface to compute mean (average) position based on nodeNumbers (as list)
  - date resolved: **2021-09-23 17:38**, date raised: 2021-09-22
- **Version 1.1.6: resolved BUG 0759: contour plot**
  - description: equivalent stress showing negative color bar values in contour plot
  - **notes: contour plot with norm (component -1) showing only positive min and max values when tested**
  - date resolved: **2021-09-23 17:19**, date raised: 2021-09-23
- **Version 1.1.5: resolved Issue 0756: FEM MisesStress (extension)**
  - description: add function that computes Mises stress from 6 stress components as obtained in stress sensor
  - **notes: put into exudyn.physics module**
  - date resolved: **2021-09-23 16:02**, date raised: 2021-09-22
- **Version 1.1.4: resolved Issue 0755: GetKinematicTree66 in robotics (extension)**
  - description: add function to export KinematicTree66 from Robotic class
  - date resolved: **2021-09-22 18:06**, date raised: 2021-09-22
- **Version 1.1.3: resolved Issue 0754: performance tests (test)**
  - description: add automated performance tests for solver speed to determine significant drop of performance
  - date resolved: **2021-09-22 10:10**, date raised: 2021-09-22
- **Version 1.1.2: resolved Issue 0620: TorsionalSpringDamper (extension)**
  - description: add torsional spring damper similar to SpringDamper, fixed on a single local axis of marker0, allowing to realize controllers and torques
  - date resolved: **2021-09-16 10:45**, date raised: 2021-04-06
- **Version 1.1.1: resolved Issue 0679: Renderer tkinter (extension)**

- description: add flag to disable calls to tkinter from Renderer, which is not possible if tkinter is already used for interactive dialogs. This allows to open visualizationSettings in AnimateModes and SolutionViewer
- date resolved: **2021-09-14 10:21**, date raised: 2021-05-14
- **Version 1.1.0: resolved BUG 0751: SetMarkerParameter**
  - description: causes internal error, because of wrong index check; workaround: use int(..) to cast marker index
  - date resolved: **2021-09-10 16:22**, date raised: 2021-09-10
- **Version 1.0.295: resolved BUG 0749: ObjectALEANCF Cable2D**
  - description: precomputed mass terms not computed accordingly; leads to crash if not static solution computed in advance
  - date resolved: **2021-09-03 15:12**, date raised: 2021-09-03
- **Version 1.0.294: resolved Issue 0748: Extend solver description (docu)**
  - description: add some description for SolveStatic / SolveDynamic in solver chapter
  - date resolved: **2021-09-03 13:54**, date raised: 2021-09-03
- **Version 1.0.293: resolved Issue 0504: serialrobot (extension)**
  - description: build completely from homogenous transformations, COMs, inertia tensors, masses, axes, axesTypes
  - **notes: done earlier**
  - date resolved: **2021-08-22 11:12**, date raised: 2020-12-16
- **Version 1.0.292: resolved Issue 0540: github (extension)**
  - description: update README.rst file and make it similar to other packages (e.g. pydy)
  - **notes: done earlier**
  - date resolved: **2021-08-22 11:11**, date raised: 2021-01-09
- **Version 1.0.291: resolved Issue 0729: README.rst (docu)**
  - description: add .rst readme file containing gettingStarted, introduction, tutorial and other information
  - date resolved: **2021-08-22 10:58**, date raised: 2021-08-05
- **Version 1.0.290: resolved Issue 0740: robotics (extension)**
  - description: extend Robot class for Modified DH parameters and general transformations; add transformations before and after joint axis
  - date resolved: **2021-08-19 00:22**, date raised: 2021-08-18
- **Version 1.0.289: resolved BUG 0739: robotics**
  - description: CreateRedundantCoordinateMBS draws wrong axes
  - date resolved: **2021-08-19 00:22**, date raised: 2021-08-18
- **Version 1.0.288: resolved BUG 0742: robotics**
  - description: Robot.JointHT computes LinkHT
  - date resolved: **2021-08-18 23:31**, date raised: 2021-08-18
- **Version 1.0.287: resolved Issue 0741: robotics (change)**
  - description: add base and tool class to robotics to have more flexibility for future developments; replace toolHT to tool.HT, baseHT to base.HT; add tool.visualization and base.visualization

- date resolved: **2021-08-18 15:07**, date raised: 2021-08-18
- **Version 1.0.286:** resolved Issue 0733: **ContactCoordinate** (extension)
  - description: add recommendedStepSize to ContactCoordinate and find optimal solution with data variable from StartOfStep configuration; check if step size is permanently reduced with recommendedStepSize; check a way of an overall recommendedStepSize (with filter) or allow a single event not to change global step size
  - date resolved: **2021-08-13 13:23**, date raised: 2021-08-12
- **Version 1.0.285:** resolved Issue 0313: **Add user node ODE2** (extension)
  - description: add user node with getposition, rotation, access functions
  - **notes: not needed: GenericNodes can be used for that**
  - date resolved: **2021-08-10 12:57**, date raised: 2020-01-10
- **Version 1.0.284:** resolved Issue 0730: **RigidBody tutorial** (docu)
  - description: add tutorial for rigid body with AddRigidBody(...), AddRevoluteJoint(...) functionalities
  - date resolved: **2021-08-06 20:05**, date raised: 2021-08-05
- **Version 1.0.283:** resolved Issue 0732: **DrawSystemGraph** (extension)
  - description: improve visualization and return graph and other information
  - date resolved: **2021-08-06 17:58**, date raised: 2021-08-06
- **Version 1.0.282:** **resolved BUG 0731: AddRevoluteJoint**
  - description: AddRevoluteJoint shows error in axis definition
  - date resolved: **2021-08-05 13:40**, date raised: 2021-08-05
- **Version 1.0.281:** resolved Issue 0704: **Optimization2** (optimize)
  - description: add direct function to NodeRigidBody to retrieve essential data for rigid body EOM and Marker-RigidBody; add flag, if rotation matrix and other quantities needed
  - **notes: still no optimization for Lie group nodes, which however have simpler matrices**
  - date resolved: **2021-07-31 22:33**, date raised: 2021-07-04
- **Version 1.0.280:** resolved Issue 0514: **general wheel** (extension)
  - description: add general wheel model (with general rotation body); for mecanum wheel rolls
  - **notes: implemented ObjectContactConvexRoll for general usage in Mecanum wheels and other applications**
  - date resolved: **2021-07-31 21:20**, date raised: 2020-12-19 (resolved by: Peter Manzl)
- **Version 1.0.279:** resolved Issue 0727: **NodeRigidBody2D** (docu)
  - description: Outputvariable Rotation gives 3D vector, but wrong description in DOCU
  - **notes: additionally: rotation is now directly copied from rotation coordinate and is not recomputed from Tait-Bryan angles of rotation matrix**
  - date resolved: **2021-07-31 21:18**, date raised: 2021-07-14
- **Version 1.0.278:** resolved Issue 0726: **description of nodes** (docu)
  - description: finish detailed description of 3D nodes and add rotation parameter description for Tait-Bryan in theory part
  - date resolved: **2021-07-13 21:17**, date raised: 2021-07-13
- **Version 1.0.277:** resolved Issue 0725: **unify description** (docu)

- description: unify notation for special vectors, e.g., reference point or local position; add unified abbreviations for ODE2, etc.
- date resolved: **2021-07-13 21:17**, date raised: 2021-07-13
- **Version 1.0.276: resolved BUG 0724: Linux version**
  - description: exudyn fails after import exudyn on Ubuntu18.04 and 20.04, showing error with RenderStateMachine selectionString
  - **notes: version 276 tested with Ubuntu20.04, working again**
  - date resolved: **2021-07-12 22:47**, date raised: 2021-07-12
- **Version 1.0.275: resolved Issue 0723: SolutionViewer (change)**
  - description: remove SolutionViewer from exudyn init file, as it causes problems if no tkinter or matplotlib installed
  - **notes: use exudyn.interactive.SolutionViewer(..) instead**
  - date resolved: **2021-07-12 20:23**, date raised: 2021-07-12
- **Version 1.0.274: resolved BUG 0722: glfwGetWindowContentScale**
  - description: function causes immediate crash on linux (UBUNTU) when importing exudyn
  - **notes: added flag for linux compilation, excluding font scaling**
  - date resolved: **2021-07-12 19:31**, date raised: 2021-07-12
- **Version 1.0.273: resolved Issue 0719: Pybind11 2.6 (change)**
  - description: switch to Pybind11 2.6 in included C++ files
  - date resolved: **2021-07-12 16:57**, date raised: 2021-07-12
- **Version 1.0.272: resolved Issue 0718: Python3.8 FASTLINALG (change)**
  - description: using now \_\_FAST\_EXUDYN\_LINALG option, which excludes all range checks and other checks in arrays, matrices, etc.; leads usually to 30percent higher performance
  - date resolved: **2021-07-12 16:57**, date raised: 2021-07-12
- **Version 1.0.271: resolved BUG 0721: FEM.GetNodesOnLine(..)**
  - description: fails because self. missing in call to GetNodesOnCylinder
  - date resolved: **2021-07-12 16:35**, date raised: 2021-07-12
- **Version 1.0.270: resolved Issue 0717: SC.StaticSolve, SC.TimeIntegrationSolve (change)**
  - description: remove these deprecated functions from interface
  - date resolved: **2021-07-12 15:33**, date raised: 2021-07-11
- **Version 1.0.269: resolved Issue 0669: remove old solvers (change)**
  - description: remove old static and dynamic solvers as they are not any more up to date with graphics interface
  - date resolved: **2021-07-12 15:32**, date raised: 2021-05-10
- **Version 1.0.268: resolved Issue 0716: SystemIsConsistent (change)**
  - description: add checks for functions that may not be called if not SystemIsConsistent
  - date resolved: **2021-07-11 17:30**, date raised: 2021-07-11
- **Version 1.0.267: resolved BUG 0714: mbs.GetSensorValues**
  - description: raises error for ObjectFFRFreducedOrder: ERROR: LinkedDataVectorBase(const VectorBase<T>&, Index), startPosition < 0

- notes: caused when called before Assemble(); checks added in future
  - date resolved: 2021-07-11 17:08, date raised: 2021-07-10
- Version 1.0.266: resolved **BUG 0715**: **ObjectFFRFreducedOrder**
  - description: OutputVariable Displacement includes localPosition, but should not
    - notes: GetMeshNodeLocalPosition included reference position twice
  - date resolved: 2021-07-11 16:33, date raised: 2021-07-10
- Version 1.0.265: resolved Issue 0706: **ConstSizeVector** (optimize)
  - description: decouple ConstSizeVector and ConstSizeMatrix from Vector / Matrix and avoid virtual calls, erase all rule of 5 member functions, optimize algebra
    - notes: improved speed up to factor 2 for some items!
  - date resolved: 2021-07-11 15:06, date raised: 2021-07-06
- Version 1.0.264: resolved **BUG 0713**: **ObjectFFRFreducedOrder**
  - description: GetOutputVariableSuperElement does not agree with types described in theDoc; object does not provide Displacement or Position, sensors return wrong values
    - notes: corrected C++ implementation and theDoc.pdf for **OutputVariableTypesSuperElement**
  - date resolved: 2021-07-09 20:53, date raised: 2021-07-09
- Version 1.0.263: resolved Issue 0712: **serialRobot** (change)
  - description: improve speed of serial robot by transferring controllers from load userfunctions to mbs.SetPreStepUserFunction
    - date resolved: 2021-07-09 13:28, date raised: 2021-07-09
- Version 1.0.262: resolved Issue 0711: **generator files** (change)
  - description: changed backslash to slash in generator files such that they can also be executed on Linux and MacOS
    - date resolved: 2021-07-09 12:18, date raised: 2021-07-09
- Version 1.0.261: resolved Issue 0699: **CMarkerBodyRigid::ComputeMarkerData** (optimize)
  - description: implement optimized version for Rigid node and ObjectRigidBody and avoid repeated computation of rotation matrix, etc.
    - date resolved: 2021-07-08 00:46, date raised: 2021-07-01
- Version 1.0.260: resolved **BUG 0709**: **Linux/MacOS compile error**
  - description: compiler error caused by EXU::Square
    - date resolved: 2021-07-07 19:11, date raised: 2021-07-07
- Version 1.0.259: resolved Issue 0708: **preprocessor flags** (change)
  - description: move EXUDYN\_RELEASE to preprocessor flags in setup.py
    - date resolved: 2021-07-07 08:53, date raised: 2021-07-07
- Version 1.0.258: resolved Issue 0705: **Optimization3** (optimize)
  - description: optimize ObjectRigidBody EOM, take Glocal columns instead numberOfRotationCoordinates, move rot\_t into loop, etc.
    - date resolved: 2021-07-06 23:04, date raised: 2021-07-04
- Version 1.0.257: resolved Issue 0703: **ComputeOrthonormalBasis** (change)
  - description: changed rigidBodyUtilities function, which returns a list of basis vectors, into ComputeOrthonormalBasisVectors, while ComputeOrthonormalBasis now returns a rotation matrix

- date resolved: **2021-07-02 08:49**, date raised: 2021-07-02
- **Version 1.0.256:** resolved Issue 0702: **AddPrismaticJoint** (extension)
  - description: add convenient utility function to add prismatic joint based on 2 bodies, point and axis, doing all necessary work in background
  - date resolved: **2021-07-02 08:49**, date raised: 2021-07-02
- **Version 1.0.255:** resolved Issue 0701: **AddRevoluteJoint** (extension)
  - description: add convenient utility function to add revolute joint based on 2 bodies, point and axis, doing all necessary work in background
  - date resolved: **2021-07-02 08:49**, date raised: 2021-07-02
- **Version 1.0.254:** resolved Issue 0700: **add links for utility functions** (docu)
  - description: ADDED LINKS to Examples/ and TestModels/ example files at end of each python utility function and class, see [Section 5](#)
  - date resolved: **2021-07-01 21:46**, date raised: 2021-07-01
- **Version 1.0.253:** **resolved BUG 0697: GenericJoint**
  - description: index2 equations not properly implemented for prismatic joints
  - **notes:** added second term for index2 case if joint position not constrained; TrapezoidalIndex2 solver now works if translational joint axes not constrained
  - date resolved: **2021-07-01 15:50**, date raised: 2021-07-01
- **Version 1.0.252:** resolved Issue 0696: **add PrismaticJoint** (extension)
  - description: add 3D prismatic joint with rotationMarker0/1 to adjust local coordinate systems and joint local x axis as the free axis of the joint
  - date resolved: **2021-07-01 13:43**, date raised: 2021-07-01
- **Version 1.0.251:** resolved Issue 0369: **add RevoluteJoint** (extension)
  - description: add 3D revolute joint with rotationMarker0/1 to adjust joint coordinates and joint local z axis as rotation axis
  - date resolved: **2021-07-01 13:43**, date raised: 2020-04-10
- **Version 1.0.250:** resolved Issue 0695: **Solution functions** (change)
  - description: remove exu and SC arguments from exudyn.utilities functions SetSolutionState(...), AnimateSolution(...); remove functoin SetVisualizationState(...): use SetSolutionState instead!
  - date resolved: **2021-06-29 17:47**, date raised: 2021-06-29
- **Version 1.0.249:** resolved Issue 0694: **SolutionViewer** (extension)
  - description: add interactive dialog to view solution based on coordinateSolution.txt
  - date resolved: **2021-06-29 17:31**, date raised: 2021-06-29
- **Version 1.0.248:** resolved Issue 0693: **RigidBody user function** (extension)
  - description: add test model for GenericODE2 user function based rigid body with Euler parameter and constraint
  - date resolved: **2021-06-28 19:34**, date raised: 2021-06-28
- **Version 1.0.247:** resolved Issue 0564: **NodeRigidBodyEP** (change)
  - description: transfer EP constraint from object to node, for future application to 3D beams
  - date resolved: **2021-06-28 19:34**, date raised: 2021-01-28

- **Version 1.0.246:** resolved Issue 0413: **ConnectorCoordinateVectorUF** (extension)
  - description: implement coordinate vector constraint user function; can be used as generic joint
  - date resolved: **2021-06-28 16:19**, date raised: 2020-05-25
- **Version 1.0.245:** resolved Issue 0691: **extend ConnectorCoordinateVector** (extension)
  - description: extend ConnectorCoordinateVector for quadratic terms to be used as Euler Parameters constraint
  - date resolved: **2021-06-27 23:29**, date raised: 2021-06-27
- **Version 1.0.244:** resolved Issue 0690: **add MarkerNodeCoordinates** (extension)
  - description: used for CoordinateVector constraint
  - date resolved: **2021-06-27 23:29**, date raised: 2021-06-27
- **Version 1.0.243:** resolved Issue 0689: **add itemIndex to user functions** (change)
  - description: CHANGE OF userFunctions interface with additional itemIndex for ConnectorSpringDamper, ConnectorCartesianSpringDamper, ConnectorRigidBodySpringDamper, ConnectorCoordinate, ConnectorCoordinateVector, ConnectorJointGeneric; see theDoc for changes in the interface of these user functions and adapt your models!
  - **notes:** **WARNING: Interface of user functions for ConnectorSpringDamper, ConnectorCartesianSpring-Damper, ConnectorRigidBodySpringDamper, ConnectorCoordinate, ConnectorCoordinateVector, Connector-JointGeneric changed!!!**
  - date resolved: **2021-06-27 20:45**, date raised: 2021-06-27
- **Version 1.0.242:** resolved Issue 0688: **ObjectGenericODE2** (change)
  - description: extend ObjectGenericODE2 and ObjectGenericODE1 user functions for the item index to have access to nodes and other informaiton: WARNING: you need to adapt your existing user functions!
  - **notes:** **WARNING: Interface of user functions for ObjectGenericODE2, ObjectFFRE... changed!!!**
  - date resolved: **2021-06-27 20:44**, date raised: 2021-06-24
- **Version 1.0.241:** resolved Issue 0687: **ObjectRigidBody** (extension)
  - description: add output variable VelocityLocal to 2D and 3D rigid body objects
  - date resolved: **2021-06-22 16:55**, date raised: 2021-06-22
- **Version 1.0.240:** resolved Issue 0686: **eigenvalue solver** (extension)
  - description: use ngsolve solver for eigenvalue computation speedup
  - date resolved: **2021-06-14 15:21**, date raised: 2021-06-14
- **Version 1.0.239: resolved BUG 0684: openGL.multisampling**
  - description: Mac OS multisampling option in visualization settings crashes; ==> do not change this option under Mac OS
  - **notes:** **excluded multisampling option for MacOS compilation**
  - date resolved: **2021-05-30 12:02**, date raised: 2021-05-25
- **Version 1.0.238: resolved BUG 0681: ObjectALEANCF Cable2D**
  - description: ObjectALEANCF Cable2D position jacobian does not provide axially moving part for ObjectContact-FrictionCircleCable2D, NEED TO BE ADDED
  - **notes:** **had been already included in MarkerBodyCable2Dshape and works for roll contact**
  - date resolved: **2021-05-30 12:01**, date raised: 2021-05-17

- **Version 1.0.237:** resolved Issue 0685: **NodeRigidBody2D** (change)
  - description: add same drawing as 3D nodes (with reference frame)
  - date resolved: **2021-05-30 11:37**, date raised: 2021-05-30
- **Version 1.0.236:** resolved Issue 0683: **SlidingJointRigid** (extension)
  - description: Add functionality for rigid sliding joint or add a flag for sliding joint to do both options
  - date resolved: **2021-05-20 09:30**, date raised: 2021-05-19
- **Version 1.0.235:** resolved Issue 0682: **copy paste code from theDoc.pdf** (extension)
  - description: changed ' characters to enable copy/paste of code with quotes
  - date resolved: **2021-05-19 08:54**, date raised: 2021-05-19
- **Version 1.0.234:** **resolved BUG 0680: SetSystemState**
  - description: mbs.systemData.SetSystemState does not set data coordinates
  - date resolved: **2021-05-15 11:07**, date raised: 2021-05-15
- **Version 1.0.233:** resolved Issue 0676: **single threaded renderer** (change)
  - description: improve exu.DoRendererIdleTasks() and use it in all python function - AnimateModes, Interactive, etc.
  - **notes: can now be also called in multithreaded renderer**
  - date resolved: **2021-05-14 21:42**, date raised: 2021-05-12
- **Version 1.0.232:** **resolved BUG 0678: mouseInteractiveExample**
  - description: not running any more, check new Renderer functions
  - date resolved: **2021-05-14 21:41**, date raised: 2021-05-14
- **Version 1.0.231:** resolved Issue 0630: **HurtyCraigBampton** (extension)
  - description: extend computation to work with 0 eigenmodes
  - date resolved: **2021-05-12 23:51**, date raised: 2021-04-23
- **Version 1.0.230:** resolved Issue 0642: **ComputeHurtyCraigBamptonModes** (extension)
  - description: add possibility to add position only interfaces
  - **notes: abandoned, because makes no sense with RBE2 modes**
  - date resolved: **2021-05-12 23:35**, date raised: 2021-04-30
- **Version 1.0.229:** **resolved BUG 0674: OutputVariable.StressLocal**
  - description: norm of OutputVariable stresses does not work
  - date resolved: **2021-05-12 23:21**, date raised: 2021-05-12
- **Version 1.0.228:** **resolved BUG 0456: ObjectFFRF bug with GenericJoint**
  - description: raises error: CSolverBase::SolveSteps CObjectSuperElement::GetAccessFunctionSuperElement: AngularVelocity\_qt not implemented; cannot compute jacobian for orientation
  - date resolved: **2021-05-12 22:27**, date raised: 2020-10-13
- **Version 1.0.226:** resolved Issue 0100: **UPDATE Lest tests** (new feature)
  - description: update lest tests (select C++ vs. python tests)
  - date resolved: **2021-05-12 22:21**, date raised: 2019-04-01

- **Version 1.0.225:** resolved Issue 0372: **add manual solver example** (extension)
  - description: add manual for solver and example; also add new prestep user function
  - **notes: resolved earlier**
  - date resolved: **2021-05-12 22:20**, date raised: 2020-04-10
- **Version 1.0.224:** resolved Issue 0384: **Solver interface** (extension)
  - description: change solver interface such that it stores MainSystem/mbs for user functions; MainSolverXYZ and CSolverXYZ take MainSystem as argument in constructor
  - date resolved: **2021-05-12 22:18**, date raised: 2020-05-06
- **Version 1.0.223:** resolved Issue 0675: **PostProcessingModes** (extension)
  - description: compute PostProcessingModes with multiprocessing
  - date resolved: **2021-05-12 22:10**, date raised: 2021-05-12
- **Version 1.0.222:** resolved Issue 0672: **right-mouse-dialog** (extension)
  - description: add item indices to right mouse dialog
  - date resolved: **2021-05-12 22:05**, date raised: 2021-05-11
- **Version 1.0.221:** resolved Issue 0673: **FEM.PostProcessingModes** (extension)
  - description: compute PostProcessingModes in FEMinterface with multiprocessing option
  - date resolved: **2021-05-12 15:37**, date raised: 2021-05-12
- **Version 1.0.220:** resolved Issue 0430: **stress modes FEMinterface** (extension)
  - description: add into FEMinterface and allow storing that data
  - **notes: resolved already earlier, see issue 623**
  - date resolved: **2021-05-12 15:36**, date raised: 2020-07-01
- **Version 1.0.219:** **resolved BUG 0631: ObjectFFRFreducedOrder**
  - description: freefree eigenmodes and Hurty-Craig-Bampton modes do not converge to same results
  - **notes: convergence for eigenmodes and HCB modes given, but differences due to HCB boundary sets, inconsistent initial conditions for MarkerSuperElementRigid; pure beam bending converges well**
  - date resolved: **2021-05-12 14:05**, date raised: 2021-04-23
- **Version 1.0.218:** **resolved BUG 0671: mbs.Reset() and SC.Reset()**
  - description: reset MainSystem mbs and SystemContainer SC hangs; current SC is erroneously stolen from renderer when another SC is deleted
  - date resolved: **2021-05-11 10:55**, date raised: 2021-05-11
- **Version 1.0.217:** resolved Issue 0670: **MacOS graphics support** (extension)
  - description: add compatibility to MacOS in single-threaded graphics mode (tested with OS X 10.7)
  - date resolved: **2021-05-10 22:34**, date raised: 2021-05-10
- **Version 1.0.216:** resolved Issue 0647: **Single Thread Renderer** (extension)
  - description: Implement single thread renderer version for MAC OS compatibility test
  - date resolved: **2021-05-10 22:32**, date raised: 2021-04-30
- **Version 1.0.215:** resolved Issue 0634: **Set visualization state** (extension)
  - description: add thread-safe variant for updating the visualization state

- date resolved: **2021-05-10 18:32**, date raised: 2021-04-25
- **Version 1.0.214:** resolved Issue 0668: **multiple mbs and SystemContainer support** (extension)
  - description: adapt renderer and MainSystemContainer to work with multiple MainSystems (mbs) and SC instances at same time; add SC.AttachToRenderEngine, SC.DetachFromRenderEngine
  - date resolved: **2021-05-10 18:20**, date raised: 2021-05-10
- **Version 1.0.213:** **resolved BUG 0665: StartRenderer()**
  - description: without being in a render loop (e.g., SC.WaitForRenderEngineStopFlag()), the pure StartRenderer() crashes upon left mouse click
  - date resolved: **2021-05-10 14:51**, date raised: 2021-05-05
- **Version 1.0.212:** resolved Issue 0664: **right mouse** (change)
  - description: add function to retrieve py::dict from items safely in python thread into temporary storage; check also other python calls to operate fully in main thread
  - date resolved: **2021-05-10 14:51**, date raised: 2021-05-05
- **Version 1.0.211:** **resolved BUG 0662: Render window**
  - description: during open tkinter dialogs, the render window responds on keyboard or mouse input, which calls again python functions that hang up the system
  - date resolved: **2021-05-10 14:51**, date raised: 2021-05-04
- **Version 1.0.210:** resolved Issue 0633: **WaitAndLockSemaphoreIgnore** (check)
  - description: check which atomic\_flags are needed in C++ to make code threadsafe
  - date resolved: **2021-05-10 14:51**, date raised: 2021-04-25
- **Version 1.0.209:** resolved Issue 0667: **tkinter dialogs focus and on top** (extension)
  - description: when opening tkinter dialogs - visualizationSettings, edit dialogs, help, ... - they immediately get focus and are on top
  - date resolved: **2021-05-10 14:49**, date raised: 2021-05-10
- **Version 1.0.208:** resolved Issue 0666: **make renderer Python and thread safe** (change)
  - description: add strict separation between renderer (thread) and Python (thread); add rendererPythonInterface between both threads; left and right mouse clicks now safe to press; render window does not accept any input as long as tkinter window is open, but does not produce crashes any more
  - date resolved: **2021-05-10 14:49**, date raised: 2021-05-10
- **Version 1.0.207:** resolved Issue 0663: **help button** (docu)
  - description: show "press h for help" as startup message for 10 seconds and sync help message with theDoc.pdf
  - date resolved: **2021-05-05 10:02**, date raised: 2021-05-05
- **Version 1.0.206:** resolved Issue 0653: **add right mouse edit dialog** (extension)
  - description: open Edit dialog for item on right-mouse-press
  - date resolved: **2021-05-04 20:58**, date raised: 2021-05-01
- **Version 1.0.205:** resolved Issue 0652: **identify itemID under mouse cursor** (extension)
  - description: identify object/node/... under mouse using unique color for itemID (left mouse button press)
  - date resolved: **2021-05-04 12:49**, date raised: 2021-05-01
- **Version 1.0.204:** resolved Issue 0637: **Python3.8 windows wheels** (extension)

- description: create Python3.8 windows wheels automatically
  - date resolved: **2021-05-03 18:51**, date raised: 2021-04-26
- **Version 1.0.203:** resolved Issue 0661: **add C++ unit tests** (extension)
  - description: add C++ unit tests to Python3.6 64bits version and to testSuite. Changed initialization of all vector types to avoid errors of Vector(5), now allowing only Vector(4.) in constructors
  - date resolved: **2021-05-03 18:50**, date raised: 2021-05-03
- **Version 1.0.202:** resolved Issue 0660: **initializerList** (check)
  - description: check if Vector is used with initializer list with one item - Vector(10), converting to std::vector or Vector(10)
  - date resolved: **2021-05-03 18:50**, date raised: 2021-05-03
- **Version 1.0.201:** resolved Issue 0659: **Troubleshooting** (docu)
  - description: Add Trouble shooting section, treating common Python and solver errors to theDoc.pdf
  - date resolved: **2021-05-03 10:18**, date raised: 2021-05-03
- **Version 1.0.200:** resolved Issue 0650: **Highlight item#** (extension)
  - description: Highlight item# for object/node/etc.; add to visualization options (itemType, item#, colorHighlightItem, colorOtherItems), draw all other items in gray
  - date resolved: **2021-05-03 01:18**, date raised: 2021-05-01
- **Version 1.0.199:** resolved Issue 0649: **add ItemType** (extension)
  - description: Add enum ItemType: Node, Object, ...
  - date resolved: **2021-05-03 01:18**, date raised: 2021-05-01
- **Version 1.0.198:** resolved Issue 0651: **add itemID to graphics objects** (extension)
  - description: Add itemID (nodes, objects, markers, loads, sensors, in that order) to graphics objects (for right-mouse-press)
  - date resolved: **2021-05-02 21:26**, date raised: 2021-05-01
- **Version 1.0.197:** resolved Issue 0658: **add VisualizationSettings() interactive** (change)
  - description: move visualization settings window functions keypressRotationStep, mouseMoveRotationFactor, keypressTranslationStep, zoomStepFactor to new substructure "interactive"
  - **notes: adapt your models if you used these options!**
  - date resolved: **2021-05-01 23:36**, date raised: 2021-05-01
- **Version 1.0.196:** resolved Issue 0654: **coordinates sizes** (extension)
  - description: add function ODE2Size(...), ODE1Size(...), SystemSize(...) to mbs.systemData to retrieve number of ODE2,ODE1,AE and Data coordinates for certain configurationType; only works after mbs.Assemble()
  - date resolved: **2021-05-01 23:23**, date raised: 2021-05-01
- **Version 1.0.195:** resolved Issue 0656: **mbs.systemData** (change)
  - description: removed GetcurrentTime() and SetVisualizationTime(...) which have been marked as deprecated already
  - **notes: Use GetTime(...) and SetTime(...) in mbs.systemData instead**
  - date resolved: **2021-05-01 23:08**, date raised: 2021-05-01
- **Version 1.0.194:** resolved Issue 0644: **solver messages** (extension)

- description: add solver message if not converged with helpful hints (especially if invert fails or newton fails)
- date resolved: **2021-05-01 02:31**, date raised: 2021-04-30
- **Version 1.0.193:** resolved Issue 0349: **add causing row** (extension)
  - description: output causing row/column (=coordinate) which leads to singular matrix; do this for Matrix.Invert as well as for SparseLU .info code; matrix class creates string with error message!
  - date resolved: **2021-05-01 02:30**, date raised: 2020-03-02
- **Version 1.0.192:** resolved Issue 0646: **jacobian singular** (extension)
  - description: resolve singularities in general jacobian: resolves coordinates which are still free for static problems, but is marked as unsafe
  - date resolved: **2021-05-01 02:29**, date raised: 2021-04-30
- **Version 1.0.191:** resolved Issue 0645: **redundant constraints** (extension)
  - description: resolve redundant constraints: add flag linearSolverSettings.ignoreSingularJacobian in SC.SimulationSettings() to ignore singular constraint jacobians
  - date resolved: **2021-05-01 02:29**, date raised: 2021-04-30
- **Version 1.0.190:** resolved Issue 0333: **node numbers with type** (extension)
  - description: extend node/object/... numbers as python class with type information to check if item numbers are mixed illegally
  - **notes: done already earlier, but still marked as unresolved**
  - date resolved: **2021-04-30 17:04**, date raised: 2020-02-06
- **Version 1.0.189:** resolved Issue 0641: **ObjectContactFrictionCircleCable2D** (docu)
  - description: add description and figure for theory and computation
  - date resolved: **2021-04-29 08:40**, date raised: 2021-04-29
- **Version 1.0.188: resolved BUG 0423:** fix **MarkerSuperElementRigidBody**
  - description: fix velocity level for MarkerSuperElementRigidBody (check constraint equations)
  - **notes: fixed several errors and test examples work now on velocity level, but further checks are necessary**
  - date resolved: **2021-04-27 18:24**, date raised: 2020-06-09
- **Version 1.0.187:** resolved Issue 0635: **AnimateModes** (check)
  - description: check, why animate modes has threading-conflicts; use std::cout to find issues
  - **notes: resolved threading conflicts, but visualization state set inbetween graphics update, which needs to resolve #634**
  - date resolved: **2021-04-27 18:20**, date raised: 2021-04-25
- **Version 1.0.186:** resolved Issue 0640: **MarkerSuperElementRigid** (extension)
  - description: remove referencePosition and add offset instead (to correct errors of midpoint due to small mesh-unsymmetries)
  - **notes: CHANGED interface: MarkerSuperElementRigid does not have a referencePosition anymore, but adds a parameter offset**
  - date resolved: **2021-04-27 18:18**, date raised: 2021-04-26
- **Version 1.0.185: resolved BUG 0639:** **ObjectFFRFreducedOrder**
  - description: incorrect AccessFunction AccessFunctionType::AngularVelocity\_qt, missing correct reference point = midpoint for computation of rotation

- date resolved: **2021-04-26 21:47**, date raised: 2021-04-26
- **Version 1.0.184:** resolved Issue 0638: **MarkerSuperElementRigid** (extension)
  - description: use consistent reference point = midpoint for computation of rotation and use exponential Map for rotation matrix
  - date resolved: **2021-04-26 21:47**, date raised: 2021-04-26
- **Version 1.0.183:** resolved Issue 0636: **Python3.8** (extension)
  - description: add python 3.8 compilation tests; resolve issues with `__index__` method needed fore NodeIndex, MarkerIndex, etc.
  - date resolved: **2021-04-26 16:25**, date raised: 2021-04-26
- **Version 1.0.182:** **resolved BUG 0629:** **mesh visualization**
  - description: visualization artifacts in larger FE meshes due to multithreading
  - notes: **added flag `threadSafeGraphicsUpdate` to avoid thread conflicts between graphics and computation, which is by default set True and MAY SLOW DOWN your computation speed if True**
  - date resolved: **2021-04-25 22:28**, date raised: 2021-04-22
- **Version 1.0.181:** resolved Issue 0632: **CMS theory** (docu)
  - description: add theory section for Hurty-Craig-Bampton modes and eigenmode computation
  - date resolved: **2021-04-23 19:03**, date raised: 2021-04-23
- **Version 1.0.180:** **resolved BUG 0628:** **FEMinterface.GetNodesOnCylinder**
  - description: returns erroneous indices
  - notes: **corrected indexing and add warnings for illegal node types**
  - date resolved: **2021-04-22 22:59**, date raised: 2021-04-22
- **Version 1.0.179:** resolved Issue 0616: **Craig-Bampton** (extension)
  - description: add static modes (Hurty-Craig-Bampton) to computation of modes in CMSinterface
  - notes: **implemented in FEMinterface.ComputeHurtyCraigBamptonModes(...)**
  - date resolved: **2021-04-21 11:09**, date raised: 2021-03-21
- **Version 1.0.178:** resolved Issue 0624: **norm in contour plots** (extension)
  - description: show norm (of vectors or stresses) in contour plot, using special outputVariable component=-1
  - date resolved: **2021-04-09 18:18**, date raised: 2021-04-09
- **Version 1.0.177:** resolved Issue 0623: **postProcessingModes** (extension)
  - description: add function to compute stress or strain modes for postprocessing, working for linear tetrahedrons (Tet4); see Examples/NGsolvePostProcessingStresses.py
  - date resolved: **2021-04-09 15:46**, date raised: 2021-04-09
- **Version 1.0.176:** resolved Issue 0424: **show modes** (extension)
  - description: add feature to visualize eigenmodes, e.g. using ObjectFFRFreducedOrder and set one initialCoordinate nonzero
  - date resolved: **2021-04-07 13:48**, date raised: 2020-06-12
- **Version 1.0.175:** resolved Issue 0619: **Eigenmode visualizer** (extension)
  - description: visualize eigenmodes with interactive tools with new function AnimateModes(...) to show eigenmodes of system or ObjectFFRFreducedOrder (see [Section 5.7](#))

- date resolved: **2021-04-07 13:47**, date raised: 2021-03-30
- **Version 1.0.174:** resolved Issue 0622: **InteractiveDialog** (extension)
  - description: improved functionality of InteractiveDialog in interactive.py, specially for animating modes
  - date resolved: **2021-04-07 12:20**, date raised: 2021-04-07
- **Version 1.0.173:** **resolved BUG 0621: mbs.GetNodeODE2Index**
  - description: Fails for NodeRigidBodyEP, because mix of AE and ODE2 variables
  - **notes: added correct type check in MainSystem:PyGetNodeODE2Index**
  - date resolved: **2021-04-07 09:12**, date raised: 2021-04-07
- **Version 1.0.172:** resolved Issue 0403: **CMS C++** (extension)
  - description: add ObjectFFRFreducedOrder (CMS) equations in C++ and clean up code
  - date resolved: **2021-03-30 16:57**, date raised: 2020-05-21
- **Version 1.0.171:** resolved Issue 0470: **geometrically exact beam2D** (extension)
  - description: add to CPP
  - date resolved: **2021-03-25 18:07**, date raised: 2020-11-21
- **Version 1.0.170:** resolved Issue 0563: **ODE1Coordinate** (extension)
  - description: add MarkerODE1Coordinate and extend LoadCoordinate for ODE1
  - date resolved: **2021-03-25 07:43**, date raised: 2021-01-27
- **Version 1.0.169:** resolved Issue 0615: **MarkerNodeCoordinate** (extension)
  - description: add check in CSystem for valid coordinate numbers in MarkerNodeCoordinate
  - date resolved: **2021-03-22 14:18**, date raised: 2021-03-21
- **Version 1.0.168:** resolved Issue 0611: **adaptiveStep** (extension)
  - description: add adaptiveStepIncrease, Decrease and RecoverySteps options to control behavior in case of discontinuous problems
  - date resolved: **2021-03-21 00:21**, date raised: 2021-03-21
- **Version 1.0.167:** resolved Issue 0603: **loadFactor** (change)
  - description: exclude load factor for loads with user functions in static computations
  - date resolved: **2021-03-20 23:24**, date raised: 2021-03-18
- **Version 1.0.166:** resolved Issue 0610: **startOfStep** (extension)
  - description: add access function for nodal coordinates at startOfStep configuration, used in mbs.GetNodeOutput(configuration = exu.ConfigurationType.startOfStep)
  - date resolved: **2021-03-20 23:23**, date raised: 2021-03-20
- **Version 1.0.165:** resolved Issue 0609: **SolveDynamic, SolveStatic** (change)
  - description: store dynamicSolver and staticSolver in mbs.sys dictionary immediately after creation, which allows to use these structures in user functions during static or dynamic solution
  - date resolved: **2021-03-20 23:23**, date raised: 2021-03-20
- **Version 1.0.164:** resolved Issue 0607: **test recommendedStepSize** (test)
  - description: test recommendedStepSize and PostNewtonUserFunction with simple elastic contact example
  - date resolved: **2021-03-20 23:23**, date raised: 2021-03-20

- **Version 1.0.163:** resolved Issue 0605: **UIIndex, UReal** (extension)
  - description: change all relevant unsigned quantities to UIIndex and UReal, as well as Vectors of UReal and Arrays of UIIndex
  - date resolved: **2021-03-20 23:23**, date raised: 2021-03-19
- **Version 1.0.162:** resolved Issue 0604: **UIIndex check** (extension)
  - description: add automatic check in item interface to check for correctness of UIIndex and UReal quantities
  - date resolved: **2021-03-20 23:23**, date raised: 2021-03-19
- **Version 1.0.161:** resolved Issue 0337: **local quantities beam** (change)
  - description: change beam output of Force, Torque, Curvature, Stress and Strain to ForceLocal, TorqueLocal, CurvatureLocal, StressLocal, and StrainLocal
  - notes: **WARNING: you need to adapt force, torque and stress, strain and curvature output variables accordingly as they may have changed specifically for ANCF beams; adapt all your model files regarding Force, Torque, etc.**
  - date resolved: **2021-03-20 23:22**, date raised: 2020-02-18
- **Version 1.0.160:** resolved Issue 0139: **Index** (change)
  - description: change Index to (signed) int and use UIIndex in python interface for unsigned parameters
  - notes: **ATTENTION: this change affects many routines. All TestSuite examples passed the change but there may still be open problems due to this major change.**
  - date resolved: **2021-03-20 23:21**, date raised: 2019-05-20
- **Version 1.0.159:** resolved Issue 0606: **resolve errors 32bit testsuite** (test)
  - description: add extra tolerances for 32bit
  - date resolved: **2021-03-20 23:19**, date raised: 2021-03-20
- **Version 1.0.158: resolved BUG 0575: new genAlpha solver**
  - description: new generalized alpha/implicit trapezoidal solver does not call solver user functions; Solution: derive CSolverImplicitSecondOrderTimeIntNew from CSolverBase and add user functions on top
  - notes: **solved by removing old solver structure; new solver fully supports user functions now**
  - date resolved: **2021-03-18 21:34**, date raised: 2021-02-08
- **Version 1.0.157:** resolved Issue 0602: **PostNewton step size recommendation** (extension)
  - description: add step recommendation as outcome of PostNewton function to improve contact and friction accuracy
  - date resolved: **2021-03-18 21:33**, date raised: 2021-03-18
- **Version 1.0.156:** resolved Issue 0601: **mbs.postNewtonUserFunction** (extension)
  - description: add function PostNewton(...) to be called after step update (Newton or explicit step)
  - date resolved: **2021-03-18 21:33**, date raised: 2021-03-18
- **Version 1.0.155:** resolved Issue 0600: **ImplicitSecondOrderSolver** (cleanup)
  - description: remove old solver
  - date resolved: **2021-03-18 21:33**, date raised: 2021-03-18
- **Version 1.0.154:** resolved Issue 0598: **rigidBodyUtilities** (extension)
  - description: add G matrices for Rxyz (Tait-Bryan angles) and also time derivatives of G

- date resolved: **2021-03-18 17:05**, date raised: 2021-03-18
- **Version 1.0.153:** resolved Issue 0594: **CMS rotations** (extension)
  - description: test and extend CMS / ObjectFFRFreducedOrder object for other rotation parameterizations (Tait-Bryan and rotation vector/Lie group) such that they work with explicit codes
  - date resolved: **2021-03-18 17:04**, date raised: 2021-02-24
- **Version 1.0.152:** resolved Issue 0597: **ObjectRigidBody** (description)
  - description: fix description of equations of motion (missing m) and add steps in derivation
  - date resolved: **2021-03-18 08:22**, date raised: 2021-03-18
- **Version 1.0.151:** resolved Issue 0283: **cylinder with hole** (new feature)
  - description: add TriangleList for cylinder with hole
  - notes: **not implemented, because it can be easily created with GraphicsDataSolidOfRevolution**
  - date resolved: **2021-03-16 16:59**, date raised: 2019-12-07
- **Version 1.0.150:** resolved Issue 0396: **description** (description)
  - description: add latex description for ObjectFFRF
  - date resolved: **2021-03-16 16:57**, date raised: 2020-05-16
- **Version 1.0.149:** resolved Issue 0394: **description** (description)
  - description: add latex description for ObjectSuperElement
  - date resolved: **2021-03-16 16:57**, date raised: 2020-05-16
- **Version 1.0.148:** resolved Issue 0595: **ObjectFFRFreducedOrder** (extension)
  - description: add general nodeType to AddObjectFFRFreducedOrderWithUserFunctions
  - date resolved: **2021-03-16 16:56**, date raised: 2021-03-14
- **Version 1.0.147:** resolved Issue 0461: **ObjectRigidBody** (check)
  - description: check discription of output variables
  - date resolved: **2021-03-16 16:55**, date raised: 2020-11-12
- **Version 1.0.146:** resolved Issue 0596: **GetRigidBodyNode** (extension)
  - description: add function GetRigidBodyNode into rigidBodyUtilities, which returns a node item for an according node type, e.g., Euler parameters or rotation vector
  - date resolved: **2021-03-14 14:43**, date raised: 2021-03-14
- **Version 1.0.145:** resolved Issue 0583: **FFRF docu** (docu)
  - description: add documentation for FFRF and FFRFfreducedOrder (CMS) to documentation
  - date resolved: **2021-03-01 12:15**, date raised: 2021-02-16
- **Version 1.0.144:** resolved Issue 0455: **FEM help** (docu)
  - description: add comments to ObjectFFRF (Tisserand frame!) and reduced that there are convenient helper functions in FEM, etc. for creating objects
  - date resolved: **2021-02-24 21:21**, date raised: 2020-10-13
- **Version 1.0.143:** resolved Issue 0593: **Add MacOS support** (change)
  - description: make minor adjustments for MacOS to run setup.py
  - date resolved: **2021-02-22 13:10**, date raised: 2021-02-22

- **Version 1.0.142:** **resolved BUG 0592:** **StartRenderer**
  - description: flag verbose=True not working
  - date resolved: **2021-02-22 10:08**, date raised: 2021-02-22
- **Version 1.0.141:** resolved Issue 0590: **SensorMarker visualization** (extension)
  - description: add visualization for SensorMarker according to marker position
  - date resolved: **2021-02-19 10:21**, date raised: 2021-02-19
- **Version 1.0.140:** resolved Issue 0400: **add SensorMarker** (extension)
  - description: add sensor for markers, restricting to position/velocity and rotation/angular velocity
  - date resolved: **2021-02-18 19:15**, date raised: 2020-05-20
- **Version 1.0.139:** resolved Issue 0585: **UserSensor** (extension)
  - description: add user sensor, which enables the user to add any kind of sensor, specifically to combine several sensor outputs into one single sensor
  - date resolved: **2021-02-18 19:14**, date raised: 2021-02-17
- **Version 1.0.138:** resolved Issue 0589: **solver updateInitialValues** (change)
  - description: update also initial coordinates in order to avoid jumps in accelerations when continuing simulation
  - date resolved: **2021-02-18 18:15**, date raised: 2021-02-18
- **Version 1.0.137:** resolved Issue 0588: **Solver file header** (change)
  - description: move writing of solution file and sensor files headers from InitializeSolverPreChecks(...) to InitializeSolverInitialConditions(...) to avoid sensor evaluation for initial configuration
  - date resolved: **2021-02-18 16:23**, date raised: 2021-02-18
- **Version 1.0.136:** resolved Issue 0587: **ALEANCF Cable2D** (fix)
  - description: change mass proportional load to include force in direction of ALE coordinate
  - date resolved: **2021-02-17 18:19**, date raised: 2021-02-17
- **Version 1.0.135:** resolved Issue 0586: **GeneticOptimization** (fix)
  - description: add special warnings and adaptations in order to catch cases where elitistRatio\*populationSize < 1 and if distanceFactor >= 1
  - date resolved: **2021-02-17 18:17**, date raised: 2021-02-17
- **Version 1.0.134:** resolved Issue 0584: **parameter variation** (extension)
  - description: add possibility to prescribe set of parameters using list, e.g., 'mass':[1,2,4,8] instead of tuple which describes the range: 'mass':(0,6,4)
  - date resolved: **2021-02-16 09:43**, date raised: 2021-02-16
- **Version 1.0.133:** resolved Issue 0582: **optimization** (docu)
  - description: add parameter variation and genetic optimization to documentation of solvers
  - date resolved: **2021-02-16 08:21**, date raised: 2021-02-16
- **Version 1.0.132:** **resolved BUG 0581:** **SetODE2Coordinates\_tt**
  - description: SetODE2Coordinates\_tt writes to velocities instead of accelerations
  - date resolved: **2021-02-14 11:12**, date raised: 2021-02-14
- **Version 1.0.131:** resolved Issue 0580: **ParameterVariation** (extension)

- description: processing.ParameterVariation(...): write to resultsFile to show progress in resultsMonitor.py
- date resolved: **2021-02-11 17:49**, date raised: 2021-02-11
- **Version 1.0.130:** resolved Issue 0576: **add ClearWorkspace** (extension)
  - description: add ClearWorkspace() to basicUtilities which allows simple and save cleanup of globals() in python environment; recommended to be called at beginning of complex models
  - date resolved: **2021-02-10 12:35**, date raised: 2021-02-08
- **Version 1.0.129:** resolved Issue 0569: **contour text** (fix)
  - description: add space to computation info text before contour plot text
  - date resolved: **2021-02-10 12:35**, date raised: 2021-02-05
- **Version 1.0.128:** resolved Issue 0568: **Renderer axes** (fix)
  - description: use X(0), Y(1) and Z(2) for axes description to be compliant with Python indexing starting with 0 as well as contour components
  - date resolved: **2021-02-10 12:35**, date raised: 2021-02-05
- **Version 1.0.127:** resolved Issue 0579: **ClearWorkspace** (extension)
  - description: add ClearWorkspacefunction to exudyn.basicUtilities, which allows to reset global variables in ipython; see example in function description
  - date resolved: **2021-02-10 12:13**, date raised: 2021-02-10
- **Version 1.0.126:** resolved Issue 0578: **SmoothStep** (extension)
  - description: add SmoothStep function to exudyn.utilities, which produces a smooth step function using cosine
  - date resolved: **2021-02-10 12:13**, date raised: 2021-02-10
- **Version 1.0.125:** resolved Issue 0572: **void** (fix)
  - description: redundant with issue 568
  - date resolved: **2021-02-10 12:05**, date raised: 2021-02-05
- **Version 1.0.124:** resolved Issue 0571: **void** (fix)
  - description: redundant with issue 568
  - date resolved: **2021-02-10 12:05**, date raised: 2021-02-05
- **Version 1.0.123:** resolved Issue 0570: **void** (fix)
  - description: redundant with issue 568
  - date resolved: **2021-02-10 12:05**, date raised: 2021-02-05
- **Version 1.0.122:** **resolved BUG 0577: PostNewtonStep**
  - description: perform PostNewtonStep and PostDiscontinuousIterationStep only for active objects
  - date resolved: **2021-02-09 14:00**, date raised: 2021-02-09
- **Version 1.0.121:** resolved Issue 0355: **generalized alpha** (extension)
  - description: implement version of Brüls and Arnold for generalized alpha solver
  - **notes:** **WARNING: switched to new solver based on displacement increments (Arnold/Brüls,2007), which leads to DIFFERENT (but improved) RESULTS than previous dynamic implicit integrator; new implicit solver now works with ODE1 variables**
  - date resolved: **2021-02-08 01:56**, date raised: 2020-03-08

- **Version 1.0.120:** resolved Issue 0573: **merge solver documentation** (docu)
  - description: merge docu on solver in EXUDYN overview and in solver section
  - date resolved: **2021-02-07 17:33**, date raised: 2021-02-07
- **Version 1.0.119:** resolved Issue 0567: **solvers description** (docu)
  - description: extend description for equations of motion, explicit and implicit solvers
  - date resolved: **2021-02-04 01:14**, date raised: 2021-02-03
- **Version 1.0.118:** resolved Issue 0560: **Impl integrator ODE1** (extension)
  - description: add ODE1 coordinates to implicit integrator
  - date resolved: **2021-02-02 15:16**, date raised: 2021-01-26
- **Version 1.0.117:** resolved Issue 0508: **implicit Lie group integrator** (extension)
  - description: implement implicit index2/index3 Lie group integrator as python function
  - **notes: cancelled, because will be directly done in C++**
  - date resolved: **2021-02-02 15:15**, date raised: 2020-12-17
- **Version 1.0.116:** resolved Issue 0566: **memory alloc cnt** (check)
  - description: add control to check whether large amount of memory allocations happen during time integration+test suite
  - date resolved: **2021-02-01 01:38**, date raised: 2021-01-29
- **Version 1.0.115:** resolved Issue 0541: **objectODE1/2, constraint lists** (extension)
  - description: add lists of ODE1 and ODE2 objects, constraints, etc. in cSystemData in order to speed up processing
  - date resolved: **2021-02-01 01:38**, date raised: 2021-01-13
- **Version 1.0.114:** resolved Issue 0557: **RK with constraints** (extension)
  - description: add CoordinateConstraints to explicit Runge-Kutta solvers
  - **notes: only ground constraints included for now**
  - date resolved: **2021-01-27 17:38**, date raised: 2021-01-26
- **Version 1.0.113:** resolved Issue 0558: **Lie group tests** (test)
  - description: add Lie group integrator simple tests
  - date resolved: **2021-01-27 12:00**, date raised: 2021-01-26
- **Version 1.0.112:** resolved Issue 0550: **GraphicsDataArrow** (extension)
  - description: add arrow to graphicsDataUtilities
  - **notes: also added GraphicsDataBasis(...) for drawing 3 orthogonal basis vectors, GraphicsDataCheckerBoard(...) for simple drawing of checker board background and MergeGraphicsDataTriangleList(...) for merging graphicsData triangle lists**
  - date resolved: **2021-01-27 00:10**, date raised: 2021-01-17
- **Version 1.0.111:** resolved Issue 0495: **add ODE1 coordinates** (extension)
  - description: extend system (Jacobian, etc.) for ODE1 coordinates
  - date resolved: **2021-01-26 13:21**, date raised: 2020-12-09
- **Version 1.0.110:** resolved Issue 0556: **explicit RK tests** (test)
  - description: add tests for explicit Runge-Kutta integrators to TestModels

- date resolved: **2021-01-26 13:17**, date raised: 2021-01-25
- **Version 1.0.109:** resolved Issue 0555: **explicit Lie group integrator** (extension)
  - description: add existing Lie group integrator in C++
  - date resolved: **2021-01-26 13:17**, date raised: 2021-01-25
- **Version 1.0.108:** resolved Issue 0554: **explicit integrator** (extension)
  - description: add explicit integrator with automatic step size control (DOPRI5, ODE23); checkout [Section 11.3](#) for description of explicit solvers and [Section 4.9.5](#) for available solver types
  - date resolved: **2021-01-25 00:54**, date raised: 2021-01-24
- **Version 1.0.107:** resolved Issue 0513: **add RK4 integrator** (extension)
  - description: put existing python RK4 integrator into CPP
  - date resolved: **2021-01-25 00:54**, date raised: 2020-12-19
- **Version 1.0.106:** resolved Issue 0533: **ObjectGenericODE1** (extension)
  - description: add object ObjectGenericODE1 for generic first order ODEs
  - date resolved: **2021-01-21 17:27**, date raised: 2021-01-04
- **Version 1.0.105:** resolved Issue 0553: **create physics submodule** (extension)
  - description: create exudyn.physics and add friction functions
  - date resolved: **2021-01-20 10:25**, date raised: 2021-01-20
- **Version 1.0.104:** **resolved BUG 0552: DrawSystemGraph**
  - description: does not work with RigidBodySpringDamper due to invalid GenericNodeData number
  - date resolved: **2021-01-19 14:43**, date raised: 2021-01-19
- **Version 1.0.103:** resolved Issue 0551: **InteractiveDialog** (extension)
  - description: add interactive tkinter dialog and new submodule exudyn.interactive to interact with models
  - date resolved: **2021-01-19 00:26**, date raised: 2021-01-19
- **Version 1.0.102:** resolved Issue 0549: **show solver name and time** (extension)
  - description: add options to show/hide solver name and current time in render window
  - date resolved: **2021-01-17 17:42**, date raised: 2021-01-17
- **Version 1.0.101:** **resolved BUG 0545: mbs.WaitForUserToContinue()**
  - description: call to WaitForUserToContinue() does not always wait for keypress. Check StartRender() function and flag settings
  - date resolved: **2021-01-17 16:55**, date raised: 2021-01-15
- **Version 1.0.100:** **resolved BUG 0548: SolveDynamic/SolveStatic**
  - description: option updateInitialValues not working
  - **notes:** corrected SetSystemState call
  - date resolved: **2021-01-17 00:08**, date raised: 2021-01-17
- **Version 1.0.99:** resolved Issue 0542: **GeneticOptimization** (extension)
  - description: store values continuously to file, add automatic loader and animate optimized values
  - **notes:** added resultsMonitor.py to exudyn module

- date resolved: **2021-01-15 15:18**, date raised: 2021-01-13
- **Version 1.0.98:** resolved Issue 0547: **realtimeSimulation** (extension)
  - description: add factor for timeIntegration.simulateInRealtime
  - date resolved: **2021-01-15 15:16**, date raised: 2021-01-15
- **Version 1.0.97:** resolved Issue 0546: **add \_\_version\_\_ version to module** (extension)
  - description: enable exudyn.\_\_version\_\_ as commonly used in other modules
  - date resolved: **2021-01-15 15:05**, date raised: 2021-01-15
- **Version 1.0.96:** resolved Issue 0544: **geneticOptimization** (extension)
  - description: add optional argument resultsFile to specify a file for output of results data
  - date resolved: **2021-01-14 23:17**, date raised: 2021-01-14
- **Version 1.0.95:** resolved Issue 0543: **add results monitor** (extension)
  - description: add resultsMonitor.py to be called from command line for doing continuous visualization of sensors and geneticOptimization output
  - date resolved: **2021-01-14 23:08**, date raised: 2021-01-14
- **Version 1.0.94:** resolved Issue 0532: **NodeGenericODE1** (extension)
  - description: add node NodeGenericODE1 for arbitrary number of ODE1 coordinates
  - date resolved: **2021-01-13 20:12**, date raised: 2021-01-04
- **Version 1.0.93:** resolved Issue 0531: **solidExtrusion** (extension)
  - description: add graphicsData for solid extrusion (prismatic) body; based on 2D point and segment list for flat boundaries
  - date resolved: **2021-01-10 20:43**, date raised: 2021-01-04
- **Version 1.0.92:** resolved Issue 0539: **RigidBodySpringDamper** (extension)
  - description: add postNewtonStepUserFunction and dataCoordinates
  - date resolved: **2021-01-08 14:34**, date raised: 2021-01-08
- **Version 1.0.91:** **resolved BUG 0537: Render window**
  - description: double calling of Render(...) function could happen from RunLoop/Render and glfwSetWindowRefreshCallback (set in InitCreateWindow(...)); check if semaphore would remove visualization problems
  - **notes: added semaphore but FEM visualization anomalies are still there**
  - date resolved: **2021-01-07 11:23**, date raised: 2021-01-06
- **Version 1.0.90:** resolved Issue 0385: **add solver eigenvalues example** (extension)
  - description: add Examples/solverFunctionsTestEigenvalues to test suite
  - **notes: added ComputeODE2EigenvaluesTest.py using new functionality exudyn.solver.ComputeODE2Eigenvalues(...)**
  - date resolved: **2021-01-07 11:08**, date raised: 2020-05-06
- **Version 1.0.89:** resolved Issue 0494: **add all tests** (extension)
  - description: add all TestModel/\*.py to testsuite and also examples before making changes to solver
  - date resolved: **2021-01-07 11:04**, date raised: 2020-12-09
- **Version 1.0.88:** resolved Issue 0515: **user function connector** (extension)
  - description: add forceUserFunction for ObjectConnectorRigidBodySpringDamper to enable User connector

- date resolved: **2021-01-07 11:03**, date raised: 2020-12-19
- **Version 1.0.87:** resolved Issue 0506: **utilities docu** (extension)
  - description: complete documentation for all exodyn python utilities and add unique headers for documentation
  - date resolved: **2021-01-06 22:57**, date raised: 2020-12-16
- **Version 1.0.86:** resolved Issue 0530: **solidOfRevolution** (extension)
  - description: add graphicsData for solid of revolution
  - date resolved: **2021-01-06 00:31**, date raised: 2021-01-04
- **Version 1.0.85:** resolved Issue 0536: **GraphicsDataPlane** (extension)
  - description: add graphicsData for simple rectangular plane with option for checkerboard pattern
  - date resolved: **2021-01-05 22:57**, date raised: 2021-01-05
- **Version 1.0.84:** resolved Issue 0535: **alternating color for cylinder** (extension)
  - description: add alternatingColor argument in GraphicsDataCylinder for visualization of rotation of cylindric bodies
  - date resolved: **2021-01-05 21:46**, date raised: 2021-01-05
- **Version 1.0.83:** **resolved Issue 0529: add MainSystem to userFunctions** (change)
  - description: add MainSystem "mbs" to all user functions as first argument (WARNING: this changes ALL user functions!!!)
  - date resolved: **2021-01-05 14:31**, date raised: 2021-01-04
- **Version 1.0.82:** resolved Issue 0447: **test examples** (check)
  - description: test all examples with new index types
  - date resolved: **2021-01-05 14:31**, date raised: 2020-09-09
- **Version 1.0.81:** **resolved BUG 0534: PlotSensor**
  - description: PlotSensor crashes for Load sensors because no outputVariableType exists
  - **notes: added special treatment for load sensors**
  - date resolved: **2021-01-04 20:11**, date raised: 2021-01-04
- **Version 1.0.80:** resolved Issue 0527: **faces transparent** (extension)
  - description: add general transparency flag for faces in visualizationSettings.opengl, switchable with button "T"; allows to make node/marker/object numbers visible
  - date resolved: **2021-01-03 21:53**, date raised: 2021-01-03
- **Version 1.0.79:** resolved Issue 0509: **ComputeODE2Eigenvalues** (test)
  - description: add example in TestModels
  - date resolved: **2021-01-03 10:44**, date raised: 2020-12-18
- **Version 1.0.78:** resolved Issue 0528: **textured fonts** (extension)
  - description: use TEXTURED based bitmap fonts based stored in glLists, allowing better interpolation, scalability (currently up to font size 64 without quality drop) and much higher performance
  - date resolved: **2021-01-03 10:29**, date raised: 2021-01-03
- **Version 1.0.77:** **resolved BUG 0526: solver.ComputeODE2Eigenvalues**
  - description: dense mode returned unsorted eigenvalues==>add sorting

- date resolved: **2021-01-03 10:21**, date raised: 2021-01-03
- **Version 1.0.76:** resolved Issue 0524: **interpret UTF8** (change)
  - description: add conversion from UTF8 to unicode to interpret most central European characters + some important characters correctly (see [Section 9.3](#))
  - date resolved: **2021-01-02 20:13**, date raised: 2020-12-29
- **Version 1.0.75:** resolved Issue 0525: **opengl write UTF8** (extension)
  - description: use UTF8 encoding in opengl text output
  - date resolved: **2020-12-29 21:00**, date raised: 2020-12-29
- **Version 1.0.74:** resolved Issue 0523: **show version** (extension)
  - description: show current version info in openGL window; can be switched off with showComputationInfo=False
  - date resolved: **2020-12-27 01:33**, date raised: 2020-12-27
- **Version 1.0.73:** resolved Issue 0522: **openGl issues** (fix)
  - description: fix positioning problems of coordinate system and contour colorbar
  - **notes:** now using pixel coordinates for info texts and different font sizes
  - date resolved: **2020-12-27 01:31**, date raised: 2020-12-27
- **Version 1.0.72:** resolved Issue 0521: **useWindowsMonitorScaleFactor** (extension)
  - description: add new option useWindowsMonitorScaleFactor in visualizationSettings.general to include monitor scaling factor for font sizes
  - date resolved: **2020-12-27 01:25**, date raised: 2020-12-27
- **Version 1.0.71:** resolved Issue 0520: **useBitmapText** (extension)
  - description: add new option useBitmapText in visualizationSettings.general to activate bitmap fonts (deprecated; now using textured fonts)
  - date resolved: **2020-12-27 01:25**, date raised: 2020-12-27
- **Version 1.0.70:** resolved Issue 0516: **add bitmap font** (extension)
  - description: add font using OpenGL bitmaps to improve visibility of texts (deprecated, now using textured fonts)
  - date resolved: **2020-12-27 01:23**, date raised: 2020-12-21
- **Version 1.0.69:** resolved Issue 0519: **correct coordinateSystemSize** (change)
  - description: set visualizationSettings.general.coordinateSystemSize relative to fontSize which scales better with larger screens
  - date resolved: **2020-12-24 01:25**, date raised: 2020-12-24
- **Version 1.0.68:** resolved Issue 0518: **windows monitor scaling** (extension)
  - description: include windows monitor (screen) scaling into drawing of texts to increase visibility on high dpi screens
  - **notes:** added flag in visualizationSettings: general.useWindowsMonitorScaleFactor
  - date resolved: **2020-12-24 00:22**, date raised: 2020-12-24
- **Version 1.0.67:** resolved Issue 0511: **GeneticOptimization** (test)
  - description: add example in TestModels
  - date resolved: **2020-12-19 23:31**, date raised: 2020-12-19

- **Version 1.0.66:** resolved Issue 0510: **ParameterVariation** (test)
  - description: add example in TestModels
  - date resolved: **2020-12-19 23:31**, date raised: 2020-12-19
- **Version 1.0.65:** resolved Issue 0502: **rigidbodyinertia** (docu)
  - description: add description for rigidBodyUtilities class RigidBodyInertia
  - date resolved: **2020-12-19 23:28**, date raised: 2020-12-14
- **Version 1.0.64:** **resolved BUG 0512: testsuite**
  - description: EXUDYN build date referred shows wrong path
  - **notes: refer now to installed module**
  - date resolved: **2020-12-19 00:44**, date raised: 2020-12-19
- **Version 1.0.63:** resolved Issue 0507: **changes** (extension)
  - description: incorporate resolved issues and bugs into theDoc.pdf
  - date resolved: **2020-12-17**, date raised: 2020-12-16
- **Version 1.0.62:** resolved Issue 0505: **rigidBodyUtilities** (extension)
  - description: add description for RigidBodyInertia class
  - date resolved: **2020-12-17**, date raised: 2020-12-16
- **Version 1.0.61:** resolved Issue 0501: **geneticOptimization add crossover** (extension)
  - description: added crossover and improved parameters for GeneticOptimization
  - date resolved: **2020-12-14**, date raised: 2020-12-14
- **Version 1.0.60:** resolved Issue 0497: **genetic algorithm** (check)
  - description: check if stochsearch or genetic algorithm has simpler interface
  - date resolved: **2020-12-14**, date raised: 2020-12-10
- **Version 1.0.59:** resolved Issue 0500: **FilterSignal** (extension)
  - description: put in signal module, make it working for 1D signals as well
  - date resolved: **2020-12-11**, date raised: 2020-12-10
- **Version 1.0.58:** resolved Issue 0492: **FEMinterface GetNodesInOrthoCube** (extension)
  - description: add function which returns all nodes lying in cube aligned with global coordinate system, using [pMin, pMax], with tolerance
  - date resolved: **2020-12-11**, date raised: 2020-12-08
- **Version 1.0.57:** resolved Issue 0491: **FEMinterface GetNodesOnCylinder** (extension)
  - description: add function which returns all nodes lying on specific cylinder, with tolerance
  - date resolved: **2020-12-11**, date raised: 2020-12-08
- **Version 1.0.56:** **resolved BUG 0499: key V gives error**
  - description: keypress v for visualization dialog gives error
  - date resolved: **2020-12-10**, date raised: 2020-12-10
- **Version 1.0.55:** **resolved BUG 0498: SensorObject position**
  - description: wrong position shown in sensor

- date resolved: **2020-12-10**, date raised: 2020-12-10
- **Version 1.0.54:** resolved Issue 0484: **test DEAP** (test)
  - description: test genetic optimization with DEAP
  - **notes: too many parameters and too involved to simply include**
  - date resolved: **2020-12-10**, date raised: 2020-12-04
- **Version 1.0.53: resolved BUG 0493: CheckForValidFunction**
  - description: modify / add this check to setParameters; additional if for setting this to 0
  - date resolved: **2020-12-09**, date raised: 2020-12-09
- **Version 1.0.52: resolved BUG 0490: keypress crash**
  - description: find out causes for crash in keyPress user function; find way to deactivate the user function (set it to 0)
  - date resolved: **2020-12-09**, date raised: 2020-12-07
- **Version 1.0.51:** resolved Issue 0389: **MainSystem includes** (cleanup)
  - description: put SystemIntegrity item checks into separate file, to reduce includig MainSystem into every .cpp item file
  - date resolved: **2020-12-09**, date raised: 2020-05-13
- **Version 1.0.50:** resolved Issue 0357: **solver flag prolong solution** (extension)
  - description: add flag for solvers that current state at end of computation is set as initial state for next solving
  - **notes: added into new python interface of solver**
  - date resolved: **2020-12-09**, date raised: 2020-03-11
- **Version 1.0.49:** resolved Issue 0489: **add gradient background** (extension)
  - description: add according visualization.general option
  - date resolved: **2020-12-06**, date raised: 2020-12-06
- **Version 1.0.48: resolved BUG 0488: problem with coordinate sys**
  - description: fix problems with drawing of coordinate system: text moves strangely and axes dissappear after rotation
  - date resolved: **2020-12-06**, date raised: 2020-12-06
- **Version 1.0.47:** resolved Issue 0487: **draw world basis** (extension)
  - description: add option to draw coordinate system at origin (world basis)
  - date resolved: **2020-12-06**, date raised: 2020-12-06
- **Version 1.0.46:** resolved Issue 0486: **realtime** (extension)
  - description: add flag to time integration to simulate in realtime
  - date resolved: **2020-12-05**, date raised: 2020-12-05
- **Version 1.0.45:** resolved Issue 0485: **mouse coordinates** (extension)
  - description: store mouse coordinates in renderState
  - date resolved: **2020-12-05**, date raised: 2020-12-05
- **Version 1.0.44:** resolved Issue 0467: **mouse coordinates** (extension)
  - description: show mouse coordinates in render window (without transformation)

- date resolved: **2020-12-05**, date raised: 2020-11-19
- **Version 1.0.43:** resolved Issue 0325: **key callback** (extension)
  - description: add key callback function into graphics module to enable interactive settings, etc.; transfer latin letters, SHIFT, CTRL, ALT, 0-9,A-Z,,SPACE as ASCII code
  - date resolved: **2020-12-05**, date raised: 2020-01-26
- **Version 1.0.42:** resolved Issue 0460: **test accelerations** (test)
  - description: test GetODE2Coordinates\_tt, nodal accelerations and rigidbody2D/3D accelerations
  - date resolved: **2020-12-04**, date raised: 2020-11-12
- **Version 1.0.41:** resolved Issue 0482: **store model view** (extension)
  - description: store renderState in exudyn.sys dictionary after exu.StopRenderer() for subsequent simulations
  - date resolved: **2020-12-03**, date raised: 2020-12-03
- **Version 1.0.40:** resolved Issue 0478: **link examples** (docu)
  - description: automatically add links to examples in thedoc
  - date resolved: **2020-12-03**, date raised: 2020-12-02
- **Version 1.0.39:** resolved Issue 0477: **links in theDoc** (extension)
  - description: add links between user functions, add labels to item sections
  - date resolved: **2020-12-03**, date raised: 2020-12-02
- **Version 1.0.38:** resolved Issue 0463: **accelerations** (extension)
  - description: add accelerations Outputvariable to Super elements
  - date resolved: **2020-12-03**, date raised: 2020-11-18
- **Version 1.0.37:** resolved Issue 0481: **eigenvalue solver** (extension)
  - description: add eigenvalue computation interface for mbs in python
  - date resolved: **2020-12-02**, date raised: 2020-12-02
- **Version 1.0.36:** resolved Issue 0480: **python solver** (extension)
  - description: add solver interfaces in python for MainSolverStatic and MainSolverImplicitSecondOrder, helping to retrieve solver data and to make solvers accessible for users
  - date resolved: **2020-12-02**, date raised: 2020-12-02
- **Version 1.0.35:** resolved Issue 0479: **solver return** (extension)
  - description: add return value to solvers and copy solver structures to mbs.sys variables after finishing
  - **notes:** added python interfaces and kept old cpp solvers
  - date resolved: **2020-12-02**, date raised: 2020-12-02
- **Version 1.0.34:** resolved Issue 0469: **userfunctions** (extension)
  - description: put user function generation in objectdefinition, with seperate U userfunction flag - this will automatically document the user function parameters (AND return values); this improves documentation and adds a unique interface in C++ using exception handling as well as GIL handling
  - date resolved: **2020-12-02**, date raised: 2020-11-20
- **Version 1.0.33:** resolved Issue 0458: **graphicsDataUserFunction** (docu)

- description: add example to docu in ObjectGround and GenericODE2 and add more accurate docu to ALL python user functions
- date resolved: **2020-12-02**, date raised: 2020-11-10
- **Version 1.0.32:** resolved Issue 0428: **queue user functions** (extension)
  - description: implement drawing user functions as global function similar to PyProcessQueue, in order to avoid messing up the CSystem and visualization modules
  - date resolved: **2020-12-02**, date raised: 2020-06-26
- **Version 1.0.31:** resolved Issue 0476: **add RequireVersion** (extension)
  - description: functionality to allow to add a simple check to see if the installed version meets the requirements
  - date resolved: **2020-11-30**, date raised: 2020-11-30
- **Version 1.0.30:** resolved Issue 0475: **rolling disc ext** (extension)
  - description: add force on ground and moving ground for ObjectJointRollingDisc
  - **notes: needs to be tested further**
  - date resolved: **2020-11-29**, date raised: 2020-11-26
- **Version 1.0.29:** resolved Issue 0474: **auto compilation** (check)
  - description: check automatic compilation; check version in wheels; check linux wheels
  - **notes: linux wheels can not be built with admin rights**
  - date resolved: **2020-11-29**, date raised: 2020-11-25
- **Version 1.0.28:** resolved Issue 0473: **no glfw option** (extension)
  - description: add simple option in setup.py to deactivate glfw both in setup.py as well as in C++ part
  - date resolved: **2020-11-29**, date raised: 2020-11-25
- **Version 1.0.27:** resolved Issue 0457: **GetVersionString** (extension)
  - description: put into docu with pybindings
  - date resolved: **2020-11-29**, date raised: 2020-11-07
- **Version 1.0.26:** resolved Issue 0472: **examples in utilities** (extension)
  - description: activate lstlisting for examples
  - date resolved: **2020-11-25**, date raised: 2020-11-25
- **Version 1.0.25:** resolved Issue 0468: **test WSL2** (test)
  - description: test compilation on WSL2 - Windows subsystem for Linux
  - **notes: WSL2 now used to automatically create linux wheels**
  - date resolved: **2020-11-21**, date raised: 2020-11-19
- **Version 1.0.24:** **resolved BUG 0465: SC.GetSystem(..)**
  - description: raises RuntimeError: should return reference instead of copy
  - date resolved: **2020-11-21**, date raised: 2020-11-18
- **Version 1.0.23:** resolved Issue 0446: **NodeIndex in arrays** (check)
  - description: use additional functionality to enable index type checks also in arrays, e.g., ArrayIndex of node numbers
  - **notes: not needed for now**

- date resolved: **2020-11-21**, date raised: 2020-09-09
- **Version 1.0.22:** resolved Issue 0383: **pybind11 submodule** (extension)
  - description: used for advanced functions, not necessarily included in exudyn or make other module
  - **notes: not needed for now**
  - date resolved: **2020-11-21**, date raised: 2020-05-06
- **Version 1.0.21:** resolved Issue 0191: **Newton lambda** (check)
  - description: Check whether Newton can be implemented as lambda-function
  - **notes: not needed for now**
  - date resolved: **2020-11-21**, date raised: 2019-06-17
- **Version 1.0.20:** resolved Issue 0466: **main/bin** (change)
  - description: remove main/bin from github and from Tools folder
  - date resolved: **2020-11-19**, date raised: 2020-11-19
- **Version 1.0.19:** resolved Issue 0464: **processing module** (extension)
  - description: create processing module for parameter variation and optimization using multiprocessing library
  - date resolved: **2020-11-18**, date raised: 2020-11-18
- **Version 1.0.18:** resolved Issue 0462: **AVX Celeron problems** (docu)
  - description: add info to documentation - FAQ AND common problems and installation instructions that CPUs without AVX support only work with 32bit version
  - date resolved: **2020-11-18**, date raised: 2020-11-16
- **Version 1.0.17:** resolved Issue 0459: **lie group utilities** (extension)
  - description: add documented lie group utilities to exudyn (python) lib
  - date resolved: **2020-11-11**, date raised: 2020-11-11 (resolved by: Stefan Holzinger)
- **Version 1.0.16:** resolved Issue 0454: **add item graph** (extension)
  - description: add graph containing nodes, objects, etc.
  - date resolved: **2020-10-08**, date raised: 2020-10-08
- **Version 1.0.15:** resolved Issue 0453: **systemdata.NumberOfSensors** (extension)
  - description: add access function for systemdata.NumberOfSensors()
  - date resolved: **2020-10-08**, date raised: 2020-10-08
- **Version 1.0.14:** resolved Issue 0449: **MT ngsolve** (extension)
  - description: add NGsolve multithreading library (task manager)
  - **notes: first tests made**
  - date resolved: **2020-09-16**, date raised: 2020-09-15
- **Version 1.0.13:** resolved Issue 0330: **correct ODE2RHS** (change)
  - description: correct ODE2RHS to ODE2Terms in objects because it is left-hand-side
  - **notes: changed object computation function from RHS to LHS, as it always computed the LHS (the system.cpp function ComputeODE2RHS then puts it to RHS)**
  - date resolved: **2020-09-10**, date raised: 2020-02-03
- **Version 1.0.12:** resolved Issue 0435: **check runtimeError** (check)

- description: check if exception runtimeerror works for all catch cases (test in windows?)
  - date resolved: **2020-09-09**, date raised: 2020-07-21
- **Version 1.0.11:** resolved Issue 0445: **remove GetItemByName()** (change)
  - description: remove GetNodeByName, GetObjectByName, etc. from C++ interface; already disabled in python interface before
  - date resolved: **2020-09-08**, date raised: 2020-09-08
- **Version 1.0.10:** resolved Issue 0288: **Item::CallFunction** (change)
  - description: Disable Item::CallFunction functionality from EXUDYN; either outputvariables can be used, or some functions are automatically created including the documentation
  - **notes: already removed from python interface earlier**
  - date resolved: **2020-09-08**, date raised: 2019-12-10
- **Version 1.0.9:** resolved Issue 0443: **SensorObject** (warning)
  - description: add error message, if sensorobject is used for a body (and check if SensorBody excepts object other than body)
  - **notes: added test for SensorObject if attached to body**
  - date resolved: **2020-09-04**, date raised: 2020-09-03
- **Version 1.0.8:** **resolved BUG 0442: difference MSC and setuptools**
  - description: compilation with MSC and setuptools gives different results
  - **notes: problem with VS2019 compilation of Eigen; resolved by removing VS2019 installation**
  - date resolved: **2020-08-25**, date raised: 2020-08-24
- **Version 1.0.7:** resolved Issue 0431: **auto create dirs** (extension)
  - description: automatically create dictionaries if they do not exist
  - date resolved: **2020-08-25**, date raised: 2020-07-01
- **Version 1.0.6:** resolved Issue 0439: **setuptools** (extension)
  - description: use setuptools for installation
  - date resolved: **2020-08-17**, date raised: 2020-08-13
- **Version 1.0.5:** resolved Issue 0381: **test pybind11\_2020** (test)
  - description: downloaded in Download folder
  - date resolved: **2020-08-17**, date raised: 2020-05-06
- **Version 1.0.4:** resolved Issue 0378: **setup tools** (extension)
  - description: use setup tools to install EXUDYN on local user accounts; use installed python version to decide which version to install
  - date resolved: **2020-08-17**, date raised: 2020-05-04
- **Version 1.0.3:** resolved Issue 0441: **remove WorkingRelease path** (change)
  - description: do not include WorkingRelease to sys.path any more, but require installation of modules
  - date resolved: **2020-08-14**, date raised: 2020-08-14
- **Version 1.0.2:** resolved Issue 0440: **exudyn package** (extension)
  - description: make a package with sub .py files in exudyn package - requires renaming of C++ module

- date resolved: **2020-08-14**, date raised: 2020-08-13
- **Version 1.0.1:** resolved Issue 0438: **UBUNTU** (extension)
  - description: adapt setup.py and implementation for gcc and UBUNTU
  - date resolved: **2020-08-13**, date raised: 2020-08-13
- **Version 1.0.0:** **resolved BUG 0434: CheckSystemIntegrity**
  - description: gives wrong node, marker, etc. numbers for some checks
  - date resolved: **2020-07-20**, date raised: 2020-07-20

## 12.2 Known open bugs

- open **BUG 0830: PostNewton**
  - description: PostNewton missing in explicit solvers; add warning or add after single steps (but exclude in contact computation!)
  - date raised: 2021-12-15
- open **BUG 0752: tkinter MacOS**
  - description: tkinter fails when loaded inside interactive.py; early call to Tk() inside an example works and seems to help; check options to correctly load tkinter in MacOS (Rosetta 2, on M1)
  - date raised: 2021-09-20
- open **BUG 0738: ObjectContactCoordinate**
  - description: modified Newton does not work, no Jacobian update computed when switching
  - date raised: 2021-08-13
- open **BUG 0677: single threaded renderer**
  - description: correct crash with visualization dialog (MacOS)
  - date raised: 2021-05-12
- open **BUG 0448: ObjectGenericODE2 bug**
  - description: ObjectGenericODE2 crashes without message when initialized with invalid node numbers
  - date raised: 2020-09-09

# References

- [1] M. Arnold and O. Brüls. Convergence of the generalized- $\alpha$  scheme for constrained mechanical systems. *Multibody System Dynamics*, 85:187–202, 2007.
- [2] M. C. C. Bampton and R. R. Craig Jr. Coupling of substructures for dynamic analyses. *American Institute of Aeronautics and Astronautics Journal*, 6(7):1313–1319, 1968.
- [3] O. A. Bauchau. *Flexible Multibody Dynamics*. Springer New York, Philadelphia, 2011.
- [4] M. Berzeri and A. Shabana. Development of simple models for the elastic forces in the absolute nodal co-ordinate formulation. *Journal of Sound and Vibration*, 235(4):539–565, 2000.
- [5] O. Brüls, M. Arnold, and A. Cardona. Two Lie group formulations for dynamic multibody systems with large rotations. In *Proceedings of IDETC/MSNDC 2011, ASME 2011 International Design Engineering Technical Conferences*, Washington, USA, 2011.
- [6] P. Chiacchio and M. Concilio. The dynamic manipulability ellipsoid for redundant manipulators. *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No.98CH36146)*, 1:95–100 vol.1, 1998.
- [7] J. Chung and G. M. Hulbert. A Time Integration Algorithm for Structural Dynamics With Improved Numerical Dissipation: The Generalized- $\alpha$  Method. *Journal of Applied Mechanics*, 60(2):371, 1993.
- [8] P. Corke. *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*. Springer Publishing Company, Incorporated, 1st edition, 2013.
- [9] C. C. de Wit, H. Olsson, K. Aström, and P. Lischinsky. Dynamic friction models and control design. In *1993 American Control Conference*, pages 1920–1926, 1993.
- [10] R. Eder and J. Gerstmayr. Special genetic identification algorithm with smoothing in the frequency domain. *Advances in Engineering Software*, 70:113–122, 2014.
- [11] P. Flores, J. Ambrósio, J. Pimenta Claro, and H. Lankarani. *Kinematics and Dynamics of Multibody Systems with Imperfect Joints*. Springer Berlin, 2008.
- [12] M. Geradin and A. Cardona. *Flexible Multibody Dynamics*. John Wiley & Sons, 2001.
- [13] J. Gerstmayr. *EXUDYN github repository*.
- [14] J. Gerstmayr. HOTINT – A C++ Environment for the simulation of multibody dynamics systems and finite elements. In K. Arczewski, J. Fraczek, and M. Wojtyra, editors, *Proceedings of the Multibody Dynamics 2009 Ecomas Thematic Conference*, 2009.
- [15] J. Gerstmayr, A. Dorninger, R. Eder, P. Gruber, D. Reischl, M. Saxinger, M. Schiessl, A. Humer, K. Nachbagauer, A. Pechstein, and Y. Vetyukov. HOTINT: A Script Language Based Framework for the Simulation of Multibody Dynamics Systems. In *9th International Conference on Multibody Systems, Nonlinear Dynamics, and Control, International Design Engineering Technical Conferences and Computers and Information in Engineering Conference (ASME IDETC 2013)*, 2013.

- [16] J. Gerstmayr and S. Holzinger. Explicit time integration of multibody systems modelled with three rotation parameters. In *International Conference on Multibody Systems, Nonlinear Dynamics, and Control, International Design Engineering Technical Conferences and Computers and Information in Engineering Conference (ASME IDETC 2020)*, 2020.
- [17] J. Gerstmayr and H. Irschik. On the correct representation of bending and axial deformation in the absolute nodal coordinate formulation with an elastic line approach. *Journal of Sound and Vibration*, 318(3):461–487, 2008.
- [18] J. Gerstmayr and M. Stangl. High-Order Implicit Runge-Kutta Methods for Discontinuous Multibody Systems. In D. A. Indeitsev, editor, *Proceedings of the {XXXII} Summer School on Advanced Problems in Mechanics ({APM} 2004)*, 2004.
- [19] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA., 1989.
- [20] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving ordinary differential equations I, nonstiff problems*. Springer Berlin Heidelberg, 1987.
- [21] G. H. Heirman and W. Desmet. Interface reduction of flexible bodies for efficient modeling of body flexibility in multibody dynamics. *Multibody system dynamics*, 24:219–234, 2010.
- [22] D. Henderson. Euler angles, quaternions, and transformation matrices for space shuttle analysis. Technical report, NASA, 1977.
- [23] S. Holzinger and J. Gerstmayr. Time integration of rigid bodies modelled with three rotation parameters. *Multibody System Dynamics*, page (online), 2021.
- [24] W. C. Hurty. Dynamic analysis of structural systems using component modes. *American Institute of Aeronautics and Astronautics Journal*, 4(3):678–685, 1965.
- [25] W. Jakob, J. Rhinelander, and D. Moldovan. pybind11 – seamless operability between c++11 and python, 2016. <https://github.com/pybind/pybind11>.
- [26] J. Kiusalaas. *Numerical methods in engineering with Python 3*. Cambridge University Press, 2013.
- [27] U. Ligrís, J. Escalona, D. Dopico, and J. Cuadrado. Efficient and accurate simulation of the rope-sheave interaction in weight-lifting machines. *Proceedings of the Institution of Mechanical Engineers, Part K: Journal of Multi-body Dynamics*, 225(4):331–343, 2011.
- [28] P. Manzl and J. Gerstmayr. An improved dynamic model of the mecanum wheel for multibody simulations. In *ASME 2021 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. American Society of Mechanical Engineers Digital Collection, 2021.
- [29] A. Müller. Coordinate Mappings for Rigid Body Motions. *Journal of Computational and Nonlinear Dynamics*, 12(2):10, 2017.
- [30] N. M. Newmark. A Method of Computation for Structural Dynamics. *ASCE Journal of the Engineering Mechanics Division*, 85(3):67–94, 1959.
- [31] A. Pechstein and J. Gerstmayr. A Lagrange-Eulerian formulation of an axially moving beam based on the absolute nodal coordinate formulation. *Multibody System Dynamics*, 30(3):343–358, 2013.
- [32] Z. Qian, D. Zhang, and C. Jin. A regularized approach for frictional impact dynamics of flexible multi-link manipulator arms considering the dynamic stiffening effect. *Multibody System Dynamics*, 43:229–255, 2018.
- [33] A. A. Shabana. Definition of the slopes and the finite element absolute nodal coordinate formulation. *Multibody system dynamics*, 1(3):339–348, 1997.

- [34] B. Siciliano and O. Khatib, editors. *Springer Handbook of Robotics*. Springer, 2016.
- [35] J. C. Simo and L. Vu-Quoc. On the dynamics in space of rods undergoing large motions - A geometrically exact approach. *Computer Methods in Applied Mechanics and Engineering*, 66(2):125–161, 1988.
- [36] V. Sonnevile, O. Brüls, and O. A. Bauchau. Interpolation schemes for geometrically exact beams: A motion approach. *International Journal for Numerical Methods in Engineering*, 112:1129–1153, 2017.
- [37] V. Sonnevile, A. Cardona, and O. Brüls. Geometrically exact beam finite element formulated on the special Euclidean group SE(3). *Computer Methods in Applied Mechanics and Engineering*, 268:451–474, 2014.
- [38] Z. Terze, A. Müller, and D. Zlatar. Singularity-free time integration of rotational quaternions using non-redundant ordinary differential equations. *Multibody System Dynamics*, 38(3):201–225, 2016.
- [39] D. Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4:65–85, 1994.
- [40] C. Woernle. *Mehrkörpersysteme: Eine Einführung in die Kinematik und Dynamik von Systemen starrer Körper*. Lecture Notes in Applied and Computational Mechanics, Springer Berlin Heidelberg, 2016.
- [41] T. Yoshikawa. Manipulability of robotic mechanisms. *The International Journal of Robotics Research*, 4(2):3–9, 1985.
- [42] A. Zwölfer and J. Gerstmayr. A concise nodal-based derivation of the floating frame of reference formulation for displacement-based solid finite elements. *Multibody System Dynamics*, 49(3):291–313, 2020.
- [43] A. Zwölfer and J. Gerstmayr. The nodal-based floating frame of reference formulation with modal reduction. *Acta Mechanica*, 2021.



# Chapter 13

## License

---

### EXUDYN General License (Version 1.0)

---

Copyright (c) 2018-2021 Johannes Gerstmayr, Institute of Mechatronics, University of Innsbruck, Austria.

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

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

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

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

---

### HotInt General License (Version 1.0)

---

Copyright (c) 1997 - 2018 Johannes Gerstmayr, Linz Center of Mechatronics GmbH, Austrian Center of Competence in Mechatronics GmbH, Institute of Technical Mechanics at the Johannes Kepler Universitaet Linz, Austria. All rights reserved.

Copyright (c) 2018 Institute of Mechatronics, University of Innsbruck, Austria.

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

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

provided with the distribution.

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

The copyright holders provide no reassurances that the source code provided does not infringe any patent, copyright, or any other intellectual property rights of third parties. The copyright holders disclaim any liability to any recipient for claims brought against recipient by any third party for infringement of that parties intellectual property rights.

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

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

---

-----  
NGsolve / NETGEN:

GNU LESSER GENERAL PUBLIC LICENSE  
Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc.  
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA  
Everyone is permitted to copy and distribute verbatim copies  
of this license document, but changing it is not allowed.

(This is the first released version of the Lesser GPL. It also counts  
as the successor of the GNU Library Public License, version 2, hence  
the version number 2.1.)

#### Preamble

The licenses for most software are designed to take away your  
freedom to share and change it. By contrast, the GNU General Public  
Licenses are intended to guarantee your freedom to share and change  
free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some  
specially designated software packages--typically libraries--of the  
Free Software Foundation and other authors who decide to use it. You  
can use it too, but we suggest you first think carefully about whether  
this license or the ordinary General Public License is the better  
strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use,  
not price. Our General Public Licenses are designed to make sure that  
you have the freedom to distribute copies of free software (and charge  
for this service if you wish); that you receive source code or can get  
it if you want it; that you can change the software and use pieces of  
it in new free programs; and that you are informed that you can do  
these things.

To protect your rights, we need to make restrictions that forbid  
distributors to deny you these rights or to ask you to surrender these  
rights. These restrictions translate to certain responsibilities for  
you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis  
or for a fee, you must give the recipients all the rights that we gave  
you. You must make sure that they, too, receive or can get the source  
code. If you link other code with the library, you must provide  
complete object files to the recipients, so that they can relink them  
with the library after making changes to the library and recompiling

it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

#### GNU LESSER GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a) The modified work must itself be a software library.
- b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.
- c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.
- d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest

your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the

Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

- a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)
- b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.
- c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.
- d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.
- e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

- a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.
- b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

#### NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

#### END OF TERMS AND CONDITIONS

#### How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
{description}
Copyright (C) {year} {fullname}
```

```
This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.
```

```
This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
Lesser General Public License for more details.
```

```
You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301
USA
```

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the
library 'Frob' (a library for tweaking knobs) written by James Random
Hacker.
```

```
{signature of Ty Coon}, 1 April 1990
Ty Coon, President of Vice
```

That's all there is to it!

=====
GLFW 3.3 - [www.glfw.org](http://www.glfw.org)

Copyright (c) 2002-2006 Marcus Geelnard
Copyright (c) 2006-2016 Camilla Löwy <elmindreda@glfw.org>

This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any damages

arising from the use of this software.

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

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

=====

PYBIND11

Copyright (c) 2016 Wenzel Jakob <wenzel.jakob@epfl.ch>, All rights reserved.

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

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

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

You are under no obligation whatsoever to provide any bug fixes, patches, or upgrades to the features, functionality or performance of the source code ("Enhancements") to anyone; however, if you choose to make your Enhancements available either publicly, or directly to the author of this software, without imposing a separate written license agreement for such Enhancements, then you hereby grant the following license: a non-exclusive, royalty-free perpetual license to install, use, modify, prepare derivative works, incorporate into other computer software, distribute, and sublicense such enhancements or derivative works thereof, in binary and source code form.

=====

LEST - Copyright 2013-2018 by Martin Moene  
Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the "Software") to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT

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

=====  
Eigen3 is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.

For more information go to <http://eigen.tuxfamily.org/>.  
Eigen3 is used under the Mozilla Public License Version 2.0 (MPL2) license:

Mozilla Public License Version 2.0  
=====

## 1. Definitions

---

### 1.1. "Contributor"

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

### 1.2. "Contributor Version"

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

### 1.3. "Contribution"

means Covered Software of a particular Contributor.

### 1.4. "Covered Software"

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

### 1.5. "Incompatible With Secondary Licenses"

means

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

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

### 1.6. "Executable Form"

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

### 1.7. "Larger Work"

means a work that combines Covered Software with other material, in a separate file or files, that is not Covered Software.

### 1.8. "License"

means this document.

### 1.9. "Licensable"

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

### 1.10. "Modifications"

means any of the following:

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

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

### 1.11. "Patent Claims" of a Contributor

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

### 1.12. "Secondary License"

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

1.13. "Source Code Form"

means the form of the work preferred for making modifications.

1.14. "You" (or "Your")

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

## 2. License Grants and Conditions

---

### 2.1. Grants

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

- (a) under intellectual property rights (other than patent or trademark) Licensable by such Contributor to use, reproduce, make available, modify, display, perform, distribute, and otherwise exploit its Contributions, either on an unmodified basis, with Modifications, or as part of a Larger Work; and
- (b) under Patent Claims of such Contributor to make, use, sell, offer for sale, have made, import, and otherwise transfer either its Contributions or its Contributor Version.

### 2.2. Effective Date

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

### 2.3. Limitations on Grant Scope

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

- (a) for any code that a Contributor has removed from Covered Software; or
- (b) for infringements caused by: (i) Your and any other third party's modifications of Covered Software, or (ii) the combination of its Contributions with other software (except as part of its Contributor Version); or
- (c) under Patent Claims infringed by Covered Software in the absence of its Contributions.

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

### 2.4. Subsequent Licenses

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

### 2.5. Representation

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

## 2.6. Fair Use

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

## 2.7. Conditions

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

## 3. Responsibilities

---

### 3.1. Distribution of Source Form

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

### 3.2. Distribution of Executable Form

If You distribute Covered Software in Executable Form then:

- (a) such Covered Software must also be made available in Source Code Form, as described in Section 3.1, and You must inform recipients of the Executable Form how they can obtain a copy of such Source Code Form by reasonable means in a timely manner, at a charge no more than the cost of distribution to the recipient; and
- (b) You may distribute such Executable Form under the terms of this License, or sublicense it under different terms, provided that the license for the Executable Form does not attempt to limit or alter the recipients rights in the Source Code Form under this License.

### 3.3. Distribution of a Larger Work

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

### 3.4. Notices

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

### 3.5. Application of Additional Terms

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

## 4. Inability to Comply Due to Statute or Regulation

---

If it is impossible for You to comply with any of the terms of this License with respect to some or all of the Covered Software due to statute, judicial order, or regulation then You must: (a) comply with

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

## 5. Termination

---

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

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

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

\*\*\*\*\*  
\* 6. Disclaimer of Warranty \*  
\* ----- \*  
\* \*  
\* Covered Software is provided under this License on an "as is" \*  
\* basis, without warranty of any kind, either expressed, implied, or \*  
\* statutory, including, without limitation, warranties that the \*  
\* Covered Software is free of defects, merchantable, fit for a \*  
\* particular purpose or non-infringing. The entire risk as to the \*  
\* quality and performance of the Covered Software is with You. \*  
\* Should any Covered Software prove defective in any respect, You \*  
\* (not any Contributor) assume the cost of any necessary servicing, \*  
\* repair, or correction. This disclaimer of warranty constitutes an \*  
\* essential part of this License. No use of any Covered Software is \*  
\* authorized under this License except under this disclaimer. \*  
\* \*  
\*\*\*\*\*

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

## 8. Litigation

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

## 9. Miscellaneous

-----

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

## 10. Versions of the License

-----

### 10.1. New Versions

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

### 10.2. Effect of New Versions

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

### 10.3. Modified Versions

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

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

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

#### Exhibit A - Source Code Form License Notice

-----

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

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

You may add additional accurate notices of copyright ownership.

#### Exhibit B - "Incompatible With Secondary Licenses" Notice

-----

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