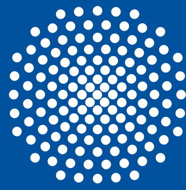
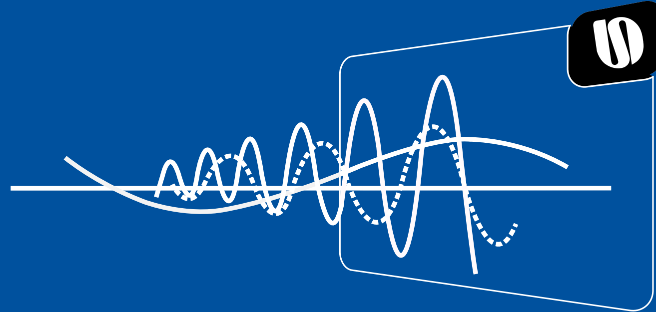


Universität Stuttgart



University of Stuttgart
Institute of Mechanics,
Structural Analysis, and Dynamics
Prof. Dr.-Ing. T. Ricken

# Computational Simulation of Fluid-Structure Interaction of Soft Kites

von Niklas Johannes Adam

| Report | No. 26 (2018) |
|---|---|

Report of the Institute of Mechanics, Structural Analysis, and Dynamics

Editor: Prof. Dr.-Ing. Tim Ricken
Supervisor: Dipl.-Ing. Karsten Keller

Pfaffenwaldring 27, 70569 Stuttgart

Master's Thesis
14. November 2018

# Computational Simulation of Fluid-Structure Interaction of Soft Kites

Niklas Johannes Adam

**Supervisor:**
Dipl.-Ing. Karsten Keller

-

Institut für Statik und Dynamik
der Luft- und Raumfahrtkonstruktionen

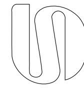
Universität Stuttgart

Pfaffenwaldring 27
70569 Stuttgart
Deutschland

*Master's Thesis*

14. November 2018

UNIVERSITÄT STUTTGART
Pfaffenwaldring 27, D-70569 Stuttgart
Tel. 0711/685 63612, Fax 0711/685 63706



INSTITUT FÜR STATIK UND DYNAMIK DER
LUFT- UND RAUMFAHRTKONSTRUKTIONEN
Institusleiter: Prof. Dr.-Ing. T. Ricken

# Master's Thesis

*for cand. aer. Niklas Johannes Adam*

## Computational Simulation of Fluid-Structure Interaction of Soft Kites

Airborne wind energy is a new renewable energy technology to harness wind energy from high altitudes where the winds are generally stronger and steadier. One of the main AWE concepts are traction kites which fly fast crosswind maneuvers to maximize the lift force. The lift force produces a high tension on the tether that goes to the ground. The tether is reeled from a drum connected to a generator which produces electricity. An accurate aerodynamic model of the inflatable membrane wing is essential for designing an airborne wind energy system.

In this project we consider soft kites which consist of a tubular inflatable frame and a membrane canopy. Connected to the tether by a system of bridle lines, such kites are highly flexible and strongly deform during flight, under the action of a distributed aerodynamic loading. Consequently, the structural deformation of the kite has to be taken into account for an accurate physical model. The objective is to couple the open-source CFD solver OpenFOAM with the open-source FE solver MBDyn.

The proposed approach consists of:

1. Qualitatively assess modeling features of MBDyn for modeling of bridled soft kites.
2. Quantitatively evaluate relevant modeling features using simple test cases.
3. Implement simplified wing and bridle system as FE model in MBDyn.
4. Validate the implemented FE model.
5. Implement an adapter to preCICE coupling environment for MBDyn.
6. Validate the coupled simulations with MBDyn and OpenFOAM.
7. Investigate the aerodynamics of the latest kite designs at TU Delft.

| | |
|---|---|
| Supervisors: | Dipl.-Ing. Karsten Keller (University of Stuttgart) |
| | Dr.-Ing. Roland Schmehl (Delft University of Technology) |
| Duration: | 6 months |
| Submission date: | 15.11.2018 |

Dipl.-Ing. Karsten Keller

# Statement of Authorship

Hiermit erkläre ich, Niklas J. Adam, dass ich die vorliegende Arbeit nur mit den angegebenen Hilfsmitteln und ohne unerlaubte fremde Hilfe selbstständig angefertigt und verfasst habe.

I hereby declare that I, Niklas J. Adam, have independently prepared and written the present work only with the specified aids and without unauthorized assistance.

Stuttgart, 14. November 2018

.....................................................

*(Unterschrift)*

## Abstract

In order to aid the development and automation of airborne wind energy (AWE) systems, the foundation for fluid-structure interaction (FSI) simulations considering soft kites is developed. FSI simulations are used as a way to predict the deformation of highly flexible structures exposed to a fluid flow and the resulting interaction of solid and fluid. This is especially important for kites since the aeroelastic effects can not be neglected if a realistic approach is regarded. Therefore, the open-source structural multibody dynamics solver MBDyn is coupled to an extension of the open-source computational fluid dynamics (CFD) solver OpenFOAM, namely FOAM-FSI, via the coupling environment preCICE. Relevant modeling features of MBDyn for soft kites such as membrane elements and appropriate boundary conditions are evaluated by means of simple test cases. Furthermore, an adapter for the communication between preCICE and MBDyn is developed and assessed as well. Since an adapter for FOAM-FSI and preCICE already exists, no efforts considering this aspect had to be made. Using this approach, a simple FSI simulation on a ram-air kite section is performed. Due to convincing results regarding the test cases, MBDyn is considered to be a suitable solver for the simulation of soft kites. Moreover, the correct implementation of the adapter is verified by the coupled FSI simulation of a modified benchmark with respect to the aforementioned participating solvers. An approach to FSI simulations on soft kites is successfully developed and verified. However, no reliable final evaluation for the kite section can be made due to the lack of reference solutions.

## Kurzfassung

Zur Unterstützung der Entwicklung und Automatisierung von Airborne Wind Energy (AWE) Systemen wird die Grundlage für Simulationen an hochgradig elastischen Kites auf Basis der Fluid-Struktur-Kopplung (engl.: fluid-structure interaction (FSI)) entwickelt. FSI Simulationen werden verwendet, um die Verformung von flexiblen, einer Strömung ausgesetzten Strukturen und die daraus resultierende Wechselwirkung von Festkörper und Fluid vorherzusagen. Da die aeroelastischen Effekte bei einem realistischen Ansatz nicht vernachlässigt werden können, ist dies bei Kites besonders wichtig. Dazu wird das Open-Source-Programm MBDyn, eine Software zum Lösen von Mehrkörpersystemen, mit der Erweiterung FOAM-FSI des Open-Source-Programms OpenFOAM zur Lösung von Strömungsproblemen (engl.: Computational Fluid Dynamics (CFD)) über die Schnittstellensoftware preCICE gekoppelt. Die entsprechenden Bausteine zur Modellierung von elastischen Kites, wie Membran-Elemente und geeignete Randbedingungen, werden anhand einfacher Testfälle beurteilt. Zudem wird ein Adapter zwischen preCICE und MBDyn entwickelt und bewertet. Da ein solcher Adapter für FOAM-FSI und preCICE schon existiert, mussten dafür keine Anstrengungen unternommen werden. Unter Verwendung dieser Grundlage wird eine FSI Simulation eines schlichten Ram-Air Kites durchgeführt. Aufgrund überzeugender Testergebnisse wird MBDyn als geeignetes Programm zur Simulation elastischer Kites erachtet. Darüber hinaus wird die korrekte Umsetzung des Adapters durch eine gekoppelte FSI Simulation eines modifizierten Benchmark-Tests mit den vorher genannten Programmen bestätigt. Ein Ansatz für FSI Simulationen an elastischen Kites wurde erfolgreich entwickelt und verifiziert. Jedoch kann eine abschließende zuverlässige Bewertung des Kites nicht stattfinden, da es keine veröffentlichten Vergleichsfälle gibt.

# Contents


**List of Figures** xiii

**List of Tables** xv

**List of Listings** xvii

**Nomenclature** xix

**1. Introduction** 1

**2. Research** 5
- 2.1. Airborne Wind Energy 5
- 2.2. MBDyn 7
- 2.3. OpenFOAM 9
- 2.4. Fluid-Structure Interaction and preCICE 12

**3. MBDyn** 17
- 3.1. Theory 17
  - 3.1.1. Fundamental Equations 17
  - 3.1.2. Structural Nodes 20
  - 3.1.3. Elements 22
- 3.2. Simulation 34
  - 3.2.1. Preprocessing 34
  - 3.2.2. Postprocessing 38
  - 3.2.3. Python Interface 39
  - 3.2.4. Implementation 41
- 3.3. Validation Tests 45
  - 3.3.1. Cantilever Beam 45
  - 3.3.2. Cable 46
  - 3.3.3. Circular Membrane Under Uniform Pressure 47
  - 3.3.4. Square Airbag 48

**4. FSI** 51
- 4.1. preCICE 51
- 4.2. MBDyn - preCICE Adapter 54
- 4.3. OpenFOAM - preCICE Adapter 58
- 4.4. Validation Tests 58
  - 4.4.1. 2D Lid-Driven Cavity with Flexible Bottom Membrane 59
  - 4.4.2. Ram-Air Kite Section 61

**5. Results** 65
5.1. MBDyn Validation Tests . . . 65
5.1.1. Cantilever Beam . . . 65
5.1.2. Cable . . . 68
5.1.3. Circular Membrane Under Uniform Pressure . . . 68
5.1.4. Square Airbag . . . 70
5.2. FSI Validation Tests . . . 74
5.2.1. 2D Lid-Driven Cavity with Flexible Bottom Membrane . . . 74
5.2.2. Ram-Air Kite Section . . . 77

**6. Conclusion** 81

**Bibliography** 83

**A. Appendix A** I
A.1. Newton's Method . . . I
A.2. Hencky's Theory . . . II

**B. Appendix B** V
B.1. Input File for Hencky's Problem . . . V
B.2. Mesh to Input File Conversion: Hencky's Problem . . . IX
B.3. VTK File Format: Hencky's Problem . . . XIII
B.4. Input File to VTK Conversion: Hencky's Problem . . . XIV
B.5. Python Code for Hencky's Problem . . . XVII

# List of Figures


2.1. AWE system consisting of a leading edge inflatable kite, bridle lines and a kite control unit [33] . . . 5
2.2. TU Delft pumping cycle [18] . . . 6
2.3. Input and Output of MBDyn presented as a black box . . . 8
2.4. Input file for MBDyn . . . 9
2.5. Directory structure for OpenFOAM (cf. [13]) . . . 12
2.6. Coupling schemes: serial (a) and parallel (b) explicit scheme and serial (c) and parallel (d) implicit scheme (cf. [24]) . . . 14

3.1. Configuration of a three-node beam in MBDyn (cf. [27]) . . . 26
3.2. FEM Discretization of a four-node shell or membrane element in MBDyn (cf. [40]) . . . 28
3.3. Node surrounded by four membrane elements; the small numbers represent the node labels and the large numbers represent the element labels . . . 33
3.4. The calculation of the area of an arbitrary quadrilateral . . . 37
3.5. UML class diagram for MBDyn (cf. [54]) . . . 43
3.6. UML activity diagram for MBDyn's solution process . . . 44
3.7. Cantilever beam (top) as described by [59] and the employed mesh (bottom) 45
3.8. Schematic drawing of the cable . . . 46
3.9. Schematic drawing of Hencky's problem and the employed mesh . . . 49
3.10. Schematic drawing of the square airbag as described by [66] and the employed mesh . . . 49

4.1. Schematic drawing of the coupling mechanism for FSI simulations with the coupling library preCICE for MBDyn and OpenFOAM including adapters (colored) . . . 51
4.2. Schematic drawing of the original lid-driven cavity benchmark (a) and a modified version (b) in order to include fluid-structure interaction . . . 60
4.3. Airfoil of the ram-air kite section (cf. [3]) . . . 61
4.4. Schematic drawing of the fluid and structural domain of the ram-air kite section (cf. [23]) . . . 61
4.5. Structural discretization of the ram-air kite section . . . 62

5.1. The initial (bottom) and deformed (top) mesh of the cantilever beam (cf. [59]) 65
5.2. Cantilever tip deflections with eight shell elements . . . 66
5.3. Cantilever tip deflections with 16 shell elements . . . 66
5.4. The deformation of the cable due to dead load . . . 67
5.5. The deflection at the apex of the cable . . . 67
5.6. The mid-point deflection of Hencky's problem . . . 69

5.7. The mid-point deflection of Hencky's problem for absolute and follower forces 69
5.8. Final shape of the airbag with 16 elements (cf. [66]) . . . . . . . . . . . . . 71
5.9. Final shape of the airbag with 25 elements (cf. [66]) . . . . . . . . . . . . . 71
5.10. Final shape of the airbag with 100 elements (cf. [66]) . . . . . . . . . . . . . 72
5.11. Final shape of the airbag with 400 elements (cf. [66]) . . . . . . . . . . . . . 72
5.12. Mid-point deflection of MBDyn and mem4py for the oscillating lid-driven cavity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
5.13. Mid-point deflection of the present work (MBDyn) and results from Mok [46] and Garcia [62] for the oscillating lid-driven cavity . . . . . . . . . . . . 75
5.14. The pressure field of the fluid domain and the deforming membrane within one period . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
5.15. Pressurized shape of the bisected kite section . . . . . . . . . . . . . . . . . 77
5.16. Original and pressurized contour at the mid-plane of the kite . . . . . . . . 77
5.17. Steady-state pressure distribution of the kite section . . . . . . . . . . . . . 78
5.18. Shape of the bisected kite section after the interaction with the fluid flow . 79
5.19. Contour of the kite at the mid-plane after the inflation and after the interaction with the fluid flow . . . . . . . . . . . . . . . . . . . . . . . . . . . 79

# List of Tables

3.1. MBDyn output . . . . . 39
3.2. Accessible properties of the nodes . . . . . 41
3.3. Numerical data for the cantilever beam . . . . . 45
3.4. Numerical data for the cable . . . . . 46
3.5. Numerical data for Hencky's problem . . . . . 48
3.6. Numerical data for the square airbag . . . . . 48

4.1. Numerical data for the lid-driven cavity . . . . . 59
4.2. Numerical data for the ram-air kite . . . . . 63

5.1. Deflection values for the midpoint of Hencky's problem . . . . . 70
5.2. Deflection values for the midpoint A of the square airbag . . . . . 70

A.1. Newton iteration . . . . . I
A.2. Value for $b_0$ . . . . . III
A.3. Exemplary displacements for Hencky's problem for a radius $a = 0.1425$ m, a membrane modulus $E_t = 311\,488$ N/m, a Poisson's number $\nu = 0.34$ and a uniform lateral loading $p = 100$ kPa . . . . . III

# List of Listings


3.1. Set up of the environment . . . . . 39
3.2. Set up of the socket . . . . . 40
3.3. Set up of the iteration . . . . . 41

4.1. preCICE data and mesh . . . . . 52
4.2. preCICE participants . . . . . 52
4.3. preCICE coupling scheme . . . . . 53
4.4. Set up of preCICE . . . . . 54
4.5. Set up of MBDyn . . . . . 55
4.6. Set up of the data exchange . . . . . 55
4.7. Coupling loop 1 . . . . . 57
4.8. Coupling loop 2 . . . . . 58

B.1. hencky.mbd . . . . . V
B.2. hencky_msh_to_mbd.py . . . . . IX
B.3. hencky1.vtk . . . . . XIII
B.4. hencky_mbd_to_vtk.py . . . . . XIV
B.5. hencky.py . . . . . XVII

# Nomenclature

Coordinate Systems

| Symbol | Description |
|---|---|
| $x, y, z$ | Global coordinate system |
| $U, V, W$ | Global coordinate system for displacements |
| $\xi, \eta, \zeta$ | Normalized coordinate system |

Mathematical Notation

| Symbol | Description |
|---|---|
| $\nabla$ | Nabla operator |
| $(...)_{,x} = \frac{\partial(...)}{\partial x}$ | Derivative with respect to $x$ |
| $(...)^T$ | Transpose |
| $\dot{(...)} = \frac{\mathrm{d}(...)}{\mathrm{d}t}$ | Time derivative |

Calligraphic Letters

| Symbol | Unit | Description |
|---|---|---|
| $\mathcal{F}$ | – | Fluid solver |
| $\mathcal{S}$ | – | Structural solver |

Matrices and Vectors

| Symbol | Unit | Description |
|---|---|---|
| $\boldsymbol{C}$ | – | Differentiation matrix |
| $\boldsymbol{E}$ | – | Green-Lagrange strain tensor |
| $\boldsymbol{f}$ | N | Force vector |

Matrices and Vectors cont.

| Symbol | Unit | Description |
|---|---|---|
| $\boldsymbol{F}$ | – | Deformation gradient |
| $\boldsymbol{g}$ | – | Caley-Gibbs-Rodriguez parameters |
| $\boldsymbol{I}$ | – | Identity matrix |
| $\boldsymbol{J}$ | $\mathrm{kgm}^2$ | Inertia tensor |
| $\boldsymbol{J}_f$ | – | Jacobi matrix |
| $\boldsymbol{K}$ | – | Penalty matrix |
| $\boldsymbol{m}$ | Nm | Torque vector |
| $\boldsymbol{M}$ | kg | Mass matrix |
| $\boldsymbol{n}$ | – | Normal vector |
| $\boldsymbol{R}$ | – | Rotation matrix |
| $\boldsymbol{T}$ | – | Orientation triad |
| $\boldsymbol{u}$ | – | Deformation vector |

Latin Letters

| Symbol | Unit | Description |
|---|---|---|
| $a$ | m | Membrane radius |
| $b$ | m | Width |
| $c$ | m | Chord length |
| $d$ | – | Axis with respect to Caley-Gibbs-Rodriguez parameters |
| $\mathrm{d}S$ | – | Surface element |
| $\mathrm{d}V$ | – | Volume element |
| $e$ | J | Energy |
| $E$ | $\mathrm{N/m}^2$ | Young's modulus |
| $E_t$ | N/m | Membrane modulus |
| $F$ | N | Force |
| $F_g$ | N | Gravitational force |
| $g$ | $\mathrm{m/s}^2$ | Gravitational acceleration |
| $G$ | $\mathrm{N/m}^2$ | Shear modulus |
| $h$ | m | Thickness |

Latin Letters cont.

| Symbol | Unit | Description |
|---|---|---|
| $J$ | $\mathrm{m}^4$ | Second moment of area |
| $l$ | m | Length |
| $m$ | kg | Mass |
| $M$ | Nm | Moment of force |
| $n_{\mathrm{b}}$ | $-$ | Amount of rigid or flexible bodies |
| $n_{\mathrm{DOF}}$ | $-$ | Amount of degrees of freedom |
| $n_{\phi}$ | $-$ | Amount of constraints |
| $p$ | $\mathrm{N/m}^2$ | Pressure |
| $q$ | $\mathrm{kg \cdot m/s}$ | Linear momentum |
| $Q$ | J | Heat |
| $r$ | m | Radius |
| $s$ | m | Cable length |
| $S$ | $\mathrm{m}^3$ | First order inertia moment |
| $t$ | s | Time |
| $\Delta t$ | s | Time step |
| $u$ | m/s | Flow velocity |
| $v$ | m/s | Velocity |
| $V$ | $\mathrm{m}^3$ | Volume |
| $W$ | J | Work |
| $\delta W$ | J | Virtual internal work |
| $x$ | m | Position |

Greek Letters

| Symbol | Unit | Description |
|---|---|---|
| $\alpha$ | $^{\circ}$ | Angle of Attack |
| $\gamma$ | $\mathrm{kg \cdot m^2/s}$ | Angular momentum |
| $\delta$ | m | Displacement |
| $\epsilon$ | $-$ | Strain |

Greek Letters cont.

| Symbol | Unit | Description |
|---|---|---|
| $\theta$ | $^\circ$ | Angle about axis with respect to Caley-Gibbs-Rodriguez parameters |
| $\lambda$ | $-$ | Lagrange multiplier |
| $\nu$ | $-$ | Poisson's number |
| $\nu_t$ | $m^2/s$ | Kinematic viscosity |
| $\pi$ | $-$ | Mathematical constant pi |
| $\rho$ | $kg/m^3$ | Density |
| $\sigma$ | $N/m^2$ | Stress |
| $\tau$ | $N/m^2$ | Shear stress |
| $\phi$ | $-$ | Holonomic algebraic constraint |
| $\phi_q$ | $W/m^2$ | Heat flux |
| $\psi$ | $-$ | Non-holonomic algebraic constraint |
| $\omega$ | rad/s | Angular velocity |

Abbreviations

| Abbreviation | Description |
|---|---|
| AoA | Angle of Attack |
| API | Application Programming Interface |
| AWE | Airborne Wind Energy |
| CFD | Computational Fluid Dynamics |
| CGR | Caley-Gibbs-Rodriguez |
| cf. | from Latin: confer ("compare") |
| DAE | Differential-Algebraic Equation |
| DOF | Degrees Of Freedom |
| etc. | from Latin: et cetera ("and so forth") |
| e.g. | from Latin: exempli gratia ("for example") |
| FEM | Finite-Element-Method |
| FVM | Finite-Volume-Method |
| FSI | Fluid-Structure Interaction |

Abbreviations cont.

| Abbreviation | Description |
|---|---|
| GUI | Graphical User Interface |
| i.a. | from Latin: inter alia (“among other things”) |
| i.e. | from Latin: id est (“that is to say”) |
| KCU | Kite Control Unit |
| MBS | Multibody System |
| MPI | Messaging Passing Interface |
| SST | Shear Stress Transport |
| TU | Technical University |
| UML | Unified Modeling Language |
| VTK | Visualization Toolkit |

# 1. Introduction

Due to a continuously increasing demand for electricity and a widespread ecological awareness, renewable energies such as wind, solar or geothermal energy are on the rise. Especially wind power generation with conventional wind turbines is highly developed and already exists since the early 1970s. However, the achievable heights with the conventional design are limited. Since wind is generally stronger and steadier at higher altitudes, a potential successor to this technology are lightweight airborne wind energy (AWE) systems. One AWE concept which is also used at the Delft University of Technology (TU Delft) consists of a soft kite tethered to a generator on the ground. The kite produces high traction power while flying crosswind figures which leads to a tensile force on the cable. The material of the kite is highly flexible and exhibits large deformations resulting from aerodynamic loads. In turn, these deformations change the shape of the kite and therefore impact the fluid flow and the resulting loads. Hence, a fully coupled structural dynamic and aerodynamic problem establishes. This fluid-structure interaction (FSI) is essential for an accurate physical model in which the loads and displacements are considered interdependently.

### Previous Research

Several kite models regarding kite control and performance, e.g. from Dadd et al. [15], Terink et al. [60] and Williams et al. [64], are based on rigid body dynamics neglecting the interaction between fluid and structure. This assumption disregards aeroelastic responses such as flutter or wing bending which are omnipresent in reality, especially regarding additional steering input by manually changing the shape of the kite. In order to obtain a realistic representation, these effects have to be taken into account. First efforts in FSI simulations of kites have been made by De Wachter [16]. He captured the shape of a ram-air kite in a wind tunnel test and carried out computational fluid dynamics (CFD) simulations with the measured geometry and hence analyzing the interaction between fluid and structure separately. Based on De Wachter's results Bungart [12] performed further FSI simulations on 2.5D flexible ram-air kite sections with membrane-type elements. However, the results are not completely satisfactory since the final deformation is not as developed as he expected. A multibody model of a leading edge tube kite developed by Breukels [11] mimics the flexibility of soft kites by the use of rigid bodies connected by damping springs. Although this approach does capture aeroelastic effects to some extend, the idealizations, assumptions and empirical data leading to the model are highly specialized on the deployed kite making it difficult to apply this multibody model to other designs. The logical extension of the aforementioned models is a finite element (FE) representation of the kite. Schwoll [57] created a highly detailed discretization of a leading edge tube kite including tens of thousands of triangular membrane elements. This

allowed for a precise static simulation of the internal pressure loading; external forces were not considered. However, the model size and resolution result in a excessive increase of computational time. Bosch [9] and Bosch et al. [8] coupled a reduced quasi-static FE model of a flexible inflatable tube kite to a quasi-static aerodynamic model. The forces from the static FSI are then used as an input at specific points at the bridle lines of a dynamic model representing a simplified AWE system. Inertial forces are neglected in order to decrease the necessary computational time. An extension to this work was carried out by Geschiere [25] who adjusted the kite design and added a more complex bridle system. Recently, FSI simulations on semi-flexible membrane wings after the form finding at multiple angles of attack (AoA) were performed by Saeedi et al. [56] including a nonlinear static FE analysis of the structure and a vortex panel method for the CFD simulation.

The current FSI scheme used at TU Delft employs the open-source CFD solver OpenFoam coupled with an in-house structural solver via the coupling library preCICE [22, 23]. The aerodynamic model is based on the work by Breukels [11]. However, the structural solver is limited to static analyses. In order to capture dynamic characteristics in the simulation, such as gusts, sudden steering input or flutter effects, either the solver has to be adjusted or an existing solver that is capable of dynamic simulations has to be integrated. The latter option has been selected for this work.

### Goal and Structure of the Thesis

The goal of the thesis is to construct and validate essential modeling features of the open-source Multibody Dynamics analysis software MBDyn with respect to the mentioned soft kites and furthermore to eventually couple MBDyn with OpenFOAM. This work is focused on the structural side of the simulation. The coupling, i.e. establishing a communication pattern between the structure and the fluid solver, is a crucial aspect in order to eventually perform FSI simulations on soft kites. A realistic FSI approach with a dynamic solver can predict the motion and deformation of the kite during the flight phases and might aid the development and automation process of AWE systems in general.

The structure of the thesis is based on the chronological development of the above stated goal. The literature research results are summarized in Chapter 2. This includes a concise description of the current AWE system at TU Delft and an introduction to the software MBDyn and OpenFOAM. Moreover, the coupling library preCICE and the interaction between fluid and structure in general are described. Chapter 3 addresses MBDyn in detail with sections about the theory behind the software, the user's modeling possibilities and various validation tests to evaluate its performance. The coupling of the structural solver to the fluid solver is introduced in Chapter 4. Here, an adapter written in `Python` providing the communication channel between MBDyn and the coupling library preCICE is presented. Furthermore, an already existing adapter between OpenFOAM and preCICE and the configuration of a popular benchmark with respect to fluid-structure interaction are described. Lastly, the FSI scheme is applied to a simplified wing section of a ram-air soft kite. The results for the MBDyn validation tests, the FSI benchmark and the wing section are compiled in Chapter 5. Finally, the summary of the thesis and the interpretation of the results are given in Chapter 6. Furthermore, essential future tasks with respect to this work are suggested. Appendix A includes certain calculations whereas the input files for

MBDyn and `Python` scripts are collected in Appendix B.

# 2. Research

In this chapter the literature research at the start of the graduation research project is summarized. First, a concise review of AWE in general and in context of system at TU Delft is presented in Section 2.1. The structure and usage of the open-source software MBDyn and OpenFOAM are described in Section 2.2 and Section 2.3 respectively, followed by an explanation of the preCICE coupling library with respect to FSI in Section 2.4.

## 2.1. Airborne Wind Energy

AWE is the wind power generation resulting from airborne devices. Two different classes of concepts can be distinguished depending on the type of energy that is transfered from the airborne devices to the ground. For ground-generation concepts, the tether is used to transfer traction power to the ground, where it is converted into electricity. For airborne-generation concepts, the wind energy is converted on-board of the devices, using small wind turbines, and the electricity is conducted to the ground. With respect to conventional wind turbines, AWE systems offer basically two main advantages. First, due to the fact that they are not limited by a supporting foundation, they can reach considerable higher

Figure 2.1.: AWE system consisting of a leading edge inflatable kite, bridle lines and a kite control unit [33]

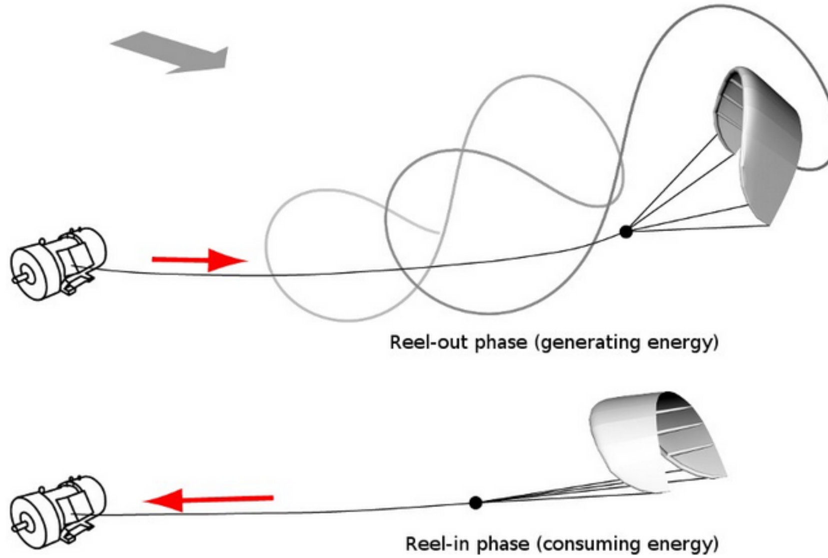



Figure 2.2.: TU Delft pumping cycle [18]

altitudes where the wind tends to be stronger and steadier [5]. This relates directly to the efficiency of the system. Second, for the same reason a remarkable amount of material can be saved which relates to a high power-to-mass ratio. A major drawback of this technology compared to conventional wind turbines are the difficulties in controlling the kite and thus the automation of the energy generation. However, realistic FSI simulations of the kite and its structural components might help to overcome these issues [17].

The current AWE system from Kitepower B.V. (formerly from the kite power research group at TU Delft) makes use of a soft kite as shown in Figure 2.1 tethered to the ground and connected to a generator. The traction on the tether is the result of a pumping cycle. This cycle, illustrated in Figure 2.2, consists of a traction and a retraction phase. In the traction phase the kite is reeled out, flying a Figures-of-eight pattern in crosswind maneuvers perpendicular to the wind direction and therefore generating high traction forces on the tether. The retraction phase utilizes the minimization of aerodynamic forces on the kite by decreasing the kite's AoA and omitting the crosswind motion. The generator is using some of the energy which was produced in the previous phase to reel-in the kite. This balance equals the net energy gain during one pumping cycle. The active steering input in order to control the kite is enforced by a kite control unit (KCU) which is represented by the black box connected to the kite by bridle lines in Figure 2.1. This powered device is able to change the flying direction or the angle of attack of the kite by adjusting the lines connected to the wing similar to the steering of a conventional kite. For a more detailed representation of this AWE system the reader is referred to Vlugt et al. [61].

As already mentioned above, the kite used in this concept is a soft kite. More precisely, it is a leading edge inflatable (LEI) kite which consists of an inflatable tubular edge at the front and cross struts. A thin canopy connects the structure and distributes the aerodynamic load to the structural elements. Bridle lines are connected to the edges in order to steer

and maneuver the kite. All these components can be seen in Figure 2.1.

## 2.2. MBDyn

MBDyn is an open-source software developed at the Department of Aerospace engineering of the university Politecnico di Milano in order to analyze multidisciplinary dynamics problems. The application field comprises and combines i.a. mechanical, aerospace and electrical engineering topics. The code was initially written in `FORTRAN 77`. However, due to the rising object-orientated programming and its advantages especially considering the potential growth of the software, the language was switched to `C++` in the early 2000s. Since then, the code grew thanks to its open-source status with the help of various researchers from many different fields [42].

### Multibody System

A multibody system (MBS) basically consists of mechanical structural bodies such as rods or beams, joints that constrain the lateral or rotational motion of rigid or flexible bodies and force elements. This system is described by the amount of degrees of freedom (DOF) of its components. Generally, bodies add three lateral and three rotational DOF and joints remove DOF depending on the number of constraints they impose. This relation can be expressed as

$$n_{\text{DOF}} = 6n_{\text{b}} - n_{\phi} \tag{2.1}$$

in which $n_{\text{DOF}}$ represents the total amount of DOF, $n_{\text{b}}$ the amount of bodies in the system and $n_{\phi}$ the number of constraints. One additional condition for the joints is, that they have to be independent; i.e. they are not allowed to constrain an already constrained DOF. With a complete description of the included bodies, joints and forces a MBS can be solved by assembling the translational and rotational Newton-Euler equations of motion for rigid bodies. Considering only one body they yield

$$\begin{pmatrix} m\boldsymbol{I} & 0 \\ 0 & \boldsymbol{J} \end{pmatrix} \begin{pmatrix} \ddot{\boldsymbol{x}} \\ \dot{\boldsymbol{\omega}} \end{pmatrix} = \begin{pmatrix} \boldsymbol{f} \\ \boldsymbol{m} - \boldsymbol{\omega} \times \boldsymbol{J}\boldsymbol{\omega} \end{pmatrix} \tag{2.2}$$

with mass $m$, identity matrix $\boldsymbol{I}$, inertia tensor $\boldsymbol{J}$, acceleration of the center of mass $\ddot{\boldsymbol{x}}$ and angular velocity $\boldsymbol{\omega}$ and acceleration $\dot{\boldsymbol{\omega}}$. The variables $\boldsymbol{f}$ and $\boldsymbol{m}$ represent the sum of all forces and moments respectively. By substituting

$$\begin{pmatrix} m\boldsymbol{I} & 0 \\ 0 & \boldsymbol{J} \end{pmatrix} = \boldsymbol{M}_i, \quad \begin{pmatrix} \ddot{\boldsymbol{x}} \\ \dot{\boldsymbol{\omega}} \end{pmatrix} = \dot{\boldsymbol{v}}_i, \quad \begin{pmatrix} \boldsymbol{f} \\ \boldsymbol{m} - \boldsymbol{\omega} \times \boldsymbol{J}\boldsymbol{\omega} \end{pmatrix} = \boldsymbol{g}_i \tag{2.3}$$

Equation (2.2) can also be written as

$$\boldsymbol{M}_i \dot{\boldsymbol{v}}_i = \boldsymbol{g}_i. \tag{2.4}$$

Thus, the equations for $n_\mathrm{b}$ bodies result in

$$\boldsymbol{M}\dot{\boldsymbol{v}} = \boldsymbol{g} \tag{2.5}$$

with

$$\boldsymbol{M} = \begin{pmatrix} M_1 & & & \\ & M_2 & & \\ & & \ddots & \\ & & & M_{n_b} \end{pmatrix}, \quad \dot{\boldsymbol{v}} = \begin{pmatrix} \dot{\boldsymbol{v}}_1 \\ \dot{\boldsymbol{v}}_2 \\ \vdots \\ \dot{\boldsymbol{v}}_{n_b} \end{pmatrix}, \quad \boldsymbol{g} = \begin{pmatrix} \boldsymbol{g}_1 \\ \boldsymbol{g}_2 \\ \vdots \\ \boldsymbol{g}_{n_b} \end{pmatrix}. \tag{2.6}$$

By adding constraints to Equation (2.5) the constrained equations of motions are written as

$$\boldsymbol{M}\dot{\boldsymbol{v}} - \boldsymbol{\phi}_{,x}^T \boldsymbol{\lambda} = \boldsymbol{g} \tag{2.7}$$

with the Jacobi matrix $\boldsymbol{\phi}_{,x}$ of the constraint condition $\boldsymbol{\phi}(\boldsymbol{x}, \dot{\boldsymbol{x}}, t) = 0$ and Lagrange multipliers $\boldsymbol{\lambda}$. This system is integrated in time and solved in an iterative manner [20, 41]. To summarize the actual implementation of the software, a system of implicit differential-algebraic equations (DAE) is used in the solution process consisting of the Newton-Euler equations of motion and algebraic holonomic and non-holonomic constraints. This DAE is integrated in time and solved for the kinematics of the model. A more detailed explanation of the theory behind MBDyn is given in Section 3.1.

### Modeling in MBDyn

Models in MBDyn can be built by the use of nodes and by elements such as nonlinear beams, shells or membranes. The simplified integration of these structural entities is shown in Figure 2.3 including a systematic black box which represents MBDyn. Since no preprocessing graphical user interface (GUI) is implemented yet, the user prepares an input file where all nodes and elements are enlisted and where the simulation data is set

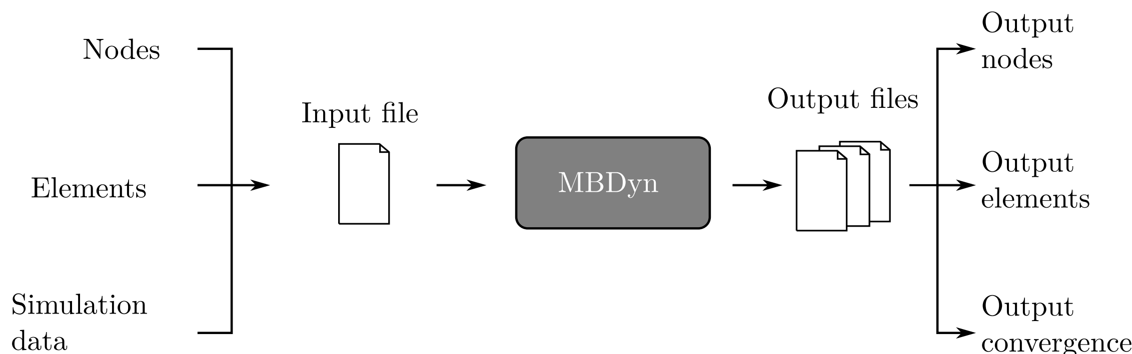



Figure 2.3.: Input and Output of MBDyn presented as a black box

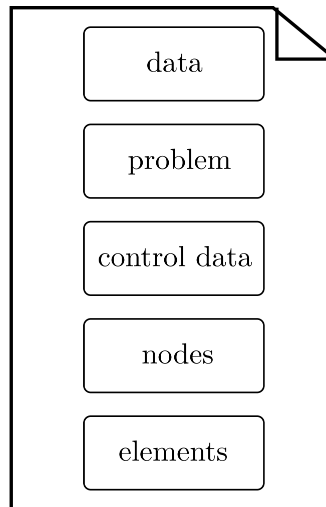



Figure 2.4.: Input file for MBDyn

according to a defined syntax. The input file consists of five main blocks as illustrated in Figure 2.4. The first two are a `data` block where the type of problem and a `problem` block where simulation data with respect to to the type of problem is defined. The next block represents the `control data` that lists solely the amount of all structural entities which are used in the model. Finally, the last two blocks define the actual `nodes` and `elements`. MBDyn reads the input file, processes the data and outputs various files with information about the simulation and the kinematics of the nodes and elements. More information about the syntax is provided in Section 3.2.

MBDyn is chosen as a structural solver due to its versatility and the ability to execute dynamic analyses. Furthermore, an open-source code and the lack of a preprocessing GUI provide a flexibility that can seldom be achieved by the use of commercial products. An equally important aspect of MBDyn is the already existing implementation of application programming interfaces (APIs) in `C`, `C++` and `Python` resulting in a simplified way of communicating with external solvers. This is a key point, especially in the context of FSI simulations [42, 54].

## 2.3. OpenFOAM

OpenFOAM is an open-source software for CFD analyses. A first draft was developed in 1989 by Henry Weller and OpenFOAM was released under its contemporary name in 2004. It consists of object-oriented libraries in `C++` implementing i.a. mesh generation and handling, physical models and their discretization and various finite volume solvers pertaining to a specific continuum mechanics problem. Furthermore, due to the modular structure of the software custom solvers can be easily implemented [31, 49].

#### Computational Fluid Dynamics

CFD is a method to approximate solutions to the governing partial differential equations of a fluid flow resulting from the conservation of mass, momentum and energy. The assumption for the conservation of mass leads to the statement that the change of mass in a control volume $V$ must be equal to the flux $\rho v$ through the boundary

$$\frac{\mathrm{d}}{\mathrm{d}t}\int\limits_{V} \rho \mathrm{d}V = -\int\limits_{\partial V} \rho v \mathrm{d}S \tag{2.8}$$

with density $\rho$ and bulk velocity $v$. Converting the surface integral into a volume integral using the divergence $\nabla\cdot$ of the flux results in

$$\int\limits_{V} \left(\rho_{,t} + \nabla\cdot(\rho v)\right)\mathrm{d}V = 0. \tag{2.9}$$

Assuming that the density and flux are continuously differentiable, the mass conservation as a differential equation for the integral form in Equation (2.9) for a domain $V$ yields

$$\rho_{,t} + \nabla\cdot(\rho v) = 0. \tag{2.10}$$

The conservation of momentum states that a change of momentum is equal to the sum of all surface and volume forces and the convection. This yields the integral momentum conservation equation

$$\frac{\mathrm{d}}{\mathrm{d}t}\int\limits_{V} \rho v \mathrm{d}V = -\int\limits_{\partial V} ((\rho v)\circ v)\mathrm{d}S - \int\limits_{\partial V} p\cdot\boldsymbol{n}\mathrm{d}S + \int\limits_{\partial V} \tau\cdot\boldsymbol{n}\mathrm{d}S + \int\limits_{V} f\mathrm{d}V \tag{2.11}$$

where $n$ is a vector orthogonal to the volume surface, $((\rho v)\circ v)$ is the convection, $p$ pressure, $\tau$ stress and $f$ an external force. The differential equation of Equation (2.11) is

$$(\rho v)_{,t} + \nabla\cdot((\rho v)\circ v) + \nabla p = \nabla\cdot\tau + f. \tag{2.12}$$

The conservation equation for energy is important when compressible fluids are considered. The equation equals the change of energy to the rate of energy transfer by the flux and by heat and work and yields

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}\int\limits_{V} e\mathrm{d}V = &-\int\limits_{\partial V} ve\cdot\boldsymbol{n}\mathrm{d}S - \int\limits_{\partial V} vp\cdot\boldsymbol{n}\mathrm{d}S + \int\limits_{\partial V} v\tau\cdot\boldsymbol{n}\mathrm{d}S + \int\limits_{V} vf\mathrm{d}V \\ &-\int\limits_{\partial V} \phi_q\cdot\boldsymbol{n}\mathrm{d}S + \int\limits_{\partial V} Q\mathrm{d}V\end{aligned} \tag{2.13}$$

with the energy transfer through convection $e$, heat flux $\phi_q$ and heat addition $Q$. Equation (2.13) can be written as a differential equation resulting in

$$e_{,t} + \nabla\cdot(v(e+p)) = \nabla\cdot\tau v + f\cdot v - \nabla\cdot\phi_q + Q \tag{2.14}$$

Equations (2.10), (2.12) and (2.14) are combined in the famous Navier-Stokes equations

$$\begin{aligned} \rho_{,t} + \nabla \cdot (\rho v) &= 0. \\ (\rho v)_{,t} + \nabla \cdot ((\rho v) \circ v) + \nabla p &= \nabla \cdot \tau + f \\ e_{,t} + \nabla \cdot (v(e + p)) &= \nabla \cdot \tau v + f \cdot v - \nabla \cdot \phi_q + Q. \end{aligned} \tag{2.15}$$

For most problems these equations can not be solved analytically and therefore have to be approximated numerically by selecting an appropriate mathematical model. Except for the extremely computational costly direct numerical simulation (DNS) of the Navier-Stokes equations, a mathematical model is always erroneous with respect to reality due to assumptions and simplifications. The translation of the mathematical model into a comprehensible format for computers is achieved by a discretization process. First, the space is discretized resulting in a structured or unstructured grid representing geometries, surfaces and boundaries of the fluid domain depending on the initial configuration. Afterwards, the equations of the model are discretized by a finite difference (FD), finite volume (FV) or finite element (FE) method. The most popular technique at the time is the FV method (FVM) where the domain is discretized and a finite volume is associated to each node of the grid. In contrast to the FD method, the integral conservation laws (Equations (2.9), (2.11) and (2.13)) are directly applied to the defined finite volumes and therefore automatically satisfy the conservative discretization. Since the solution and flux are only known at the cell average points (for an exemplary FVM) the reconstruction of the values at the grid points is performed by the use of numerical schemes which relate neighboring mesh values [24, 29, 47].

### OpenFOAM Case Structure

A case in OpenFOAM is set up by a specific directory structure as illustrated in Figure 2.5. Generally three subdirectories are created referring to the `system`, `constant` properties and `time directories`. Files in the `system` directory define properties with respect to the solution of the problem. The `controlDict` file contains main control parameters for the analysis. Numerical schemes e.g. time derivatives or gradient schemes are defined in `fvSchemes`. Solvers, preconditioners and tolerances are set in the `fvSolution` file. Lastly, in `blockMeshDict` the mesh is generated. The `constant` subdirectory includes `properties` with respect to the physical problem statement such as turbulence or transport properties. Furthermore, it contains a detailed description of the generated mesh in the directory `polyMesh`. In the `time directories` initial and boundary conditions at time $t = 0$ (in case of initial value problems) are defined. Moreover, OpenFOAM writes its results to this directory [13, 50].

### FOAM-FSI

In order to utilize already existing solvers for FSI simulations, a different set of libraries has to be used, namely from the project foam-extend. This is a branch of the OpenFOAM library for CFD with contributions from developers all over the community. Foam-extend contains extensions, additional features and general improvements regarding the original

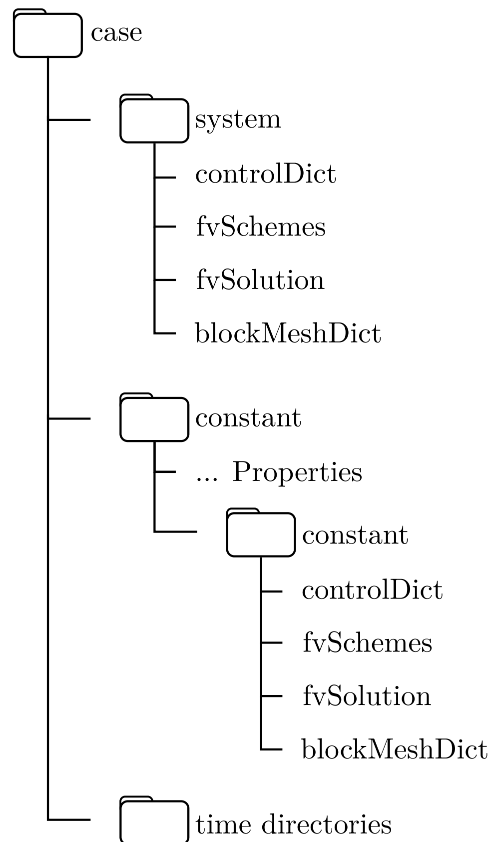



Figure 2.5.: Directory structure for OpenFOAM (cf. [13])

library [32]. A further extension to foam-extend is the FOAM-FSI library developed by Blom [6]. This library contains algorithms for an implicit coupling between fluid and structural solvers based on a partitioned approach and furthermore a coupling adapter for preCICE (see Section 2.4). FOAM-FSI is used as the fluid solver also due to the fact that a successful coupling between OpenFOAM and an in-house solver at TU Delft has already been achieved.

## 2.4. Fluid-Structure Interaction and preCICE

The term FSI is used, when two physical fields for problems are coupled; e.g. fluid and solid dynamics. This is the case when the fluid flow affects the structural component in such a way that it deforms. In turn, the deformation of the structure impacts the flow field and hence, a dependency between the fluid and the structure arises. Popular examples for FSI applications are the wing bending of aircraft due to lift in flight or the blood flow in vessels of the human body where a higher blood pressure leads to a widening of the channel and simultaneously to a change in the blood flow. FSI simulations are a relatively

new domain starting in the 1990s, whereas the sole computation of the structure and fluid dates way back. This is mostly due to the continuous development of more efficient and powerful computational resources since the simultaneous simulation of both fields is extremely challenging [55].

### Approaches

Generally, two type of approaches can be distinguished regarding FSI simulation. The computation of the structure and the fluid domain by two separate specialized solvers and synchronizing them at runtime is a so called partitioned approach. Here, the communication between the participants has to be established usually by a common interface. It is the natural and standard strategy since these solvers often already exist and are understood, tested and well developed for their domain. This fact is especially important in the industry. The partitioned coupling system of equations can be described as

$$\begin{aligned} \mathcal{F}(\boldsymbol{s}) &= \boldsymbol{f} \\ \mathcal{S}(\boldsymbol{f}) &= \boldsymbol{s} \end{aligned} \tag{2.16}$$

with a representation of the fluid solver $\mathcal{F}$ that calculates the forces and stresses $\boldsymbol{f}$ based on the kinematics $\boldsymbol{s}$ at the interface and the structural solver $\mathcal{S}$ that calculates the kinematics $\boldsymbol{s}$ based on the forces and stresses $\boldsymbol{f}$ at the interface. However, since two distinct models and the interface have to be described, a potential source of errors and inconsistency is created. The counterpart to the partitioned approach is the monolithic approach which uses a fully coupled mathematical model in one simulation and software and thus exhibits a more stable solution process. The coupled system of equations for a monolithic approach results in

$$\mathcal{A}(y_{\mathcal{F}}, y_{\mathcal{S}}) = 0 \tag{2.17}$$

with the flow and structural variable $y_{\mathcal{F}}$ and $y_{\mathcal{S}}$ respectively. Nevertheless, the focus in this work lies on the partitioned approach for the reasons stated above [1, 24, 55].

### Coupling Scheme

The coupling in a partitioned approach consists of a coupling scheme for the equations and the data mapping between the participants, i.e. the participating solvers. The coupling schemes are a method of approximating the system of equations of the partitioned approach in Equation (2.16) to the system of equations of the monolithic approach in Equation (2.17). However, in a logical sense, they define the execution order of the two participating solvers. They can be divided into explicit and implicit coupling schemes and moreover into a serial and a parallel variant respectively. If an explicit scheme is used, the participants compute their solution only once in one time step. A serial and parallel explicit scheme is shown in Figure 2.6 (a) and (b) respectively. In serial configuration the fluid solver calculates the stresses of the next time step $\mathcal{F}^{n+1}$ with the kinematics of the old time step $\boldsymbol{s}^n$. As soon as the fluid solver is executed exactly once, the structural solver starts and computes the kinematics of the next time step $\mathcal{S}^{n+1}$ with the stresses of the new time step $\boldsymbol{f}^{n+1}$. Note

that the execution sequence is interchangeable. A parallel explicit scheme establishes a bidirectional communication channel for data exchange at the beginning of each time step. Thus, the stresses and displacements of the new time step $\mathcal{F}^{n+1}$ and $\mathcal{S}^{n+1}$ are computed simultaneously using the old time step values $\boldsymbol{s}^n$ and $\boldsymbol{f}^n$ respectively.

Serial and parallel implicit schemes are illustrated in Figure 2.6 (c) and (d). The main difference is that here, both solvers are iterated over one time step until a predefined measure of convergence is reached. The iterations are illustrated as vertical small lines throughout the time step. Implicit methods are often times desired since they tend to be much more stable. However, for this to be the case, certain postprocessing methods such as constant underrelaxation, adaptive underrelaxation and several quasi-Newton schemes are applied. These methods are used in order to improve the overall stability of iterative methods by stabilizing data values at the interface, e.g. with values from previous iterations. A constant underrelaxation is providing a constant factor that is applied to the previous and the current dataset, where previous is associated with the previous iteration. In turn, the adaptive underrelaxation is accelerating the convergence by the use of a dynamic relaxation

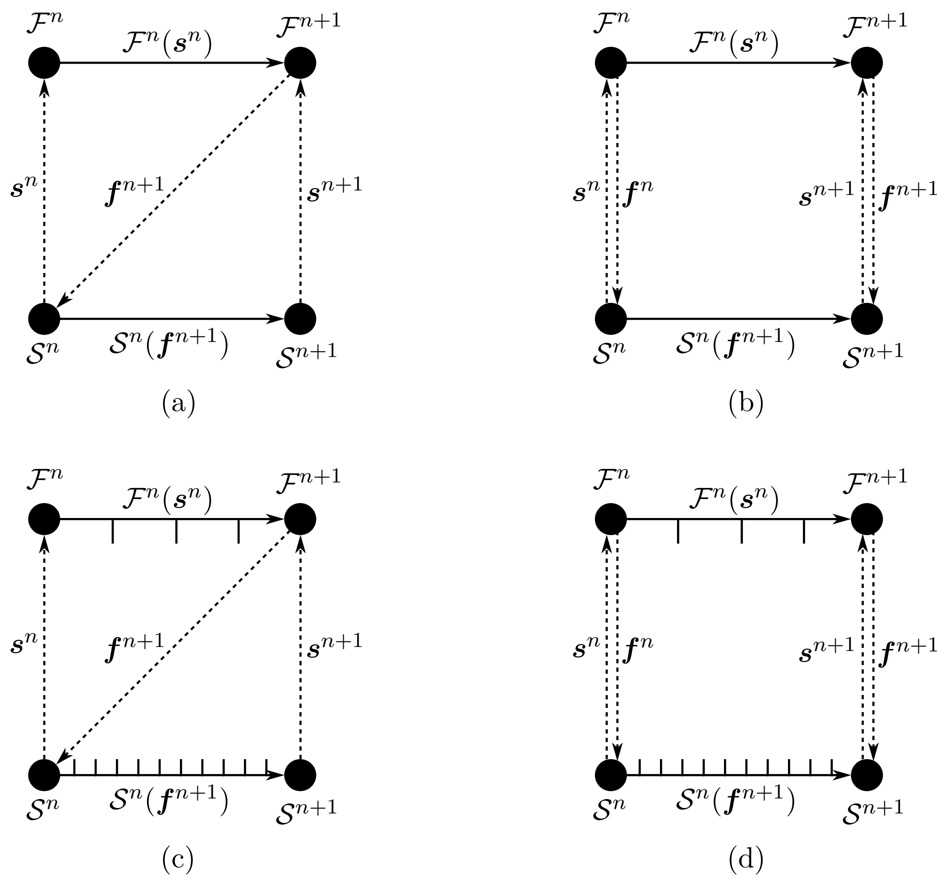



Figure 2.6.: Coupling schemes: serial (a) and parallel (b) explicit scheme and serial (c) and parallel (d) implicit scheme (cf. [24])

factor from two previous datasets which changes in each iteration. Quasi-Newton methods are using reduced models to obtain the Jacobian matrix in order to find the roots of the data function in a way that the residual goes to zero. If no postprocessing methods were used, the implicit scheme would suffer the same instabilities as the explicit scheme [24, 52]

### Data Mapping

Data mapping is another aspect in the coupling of a partitioned approach. Since most of the time the structural and fluid mesh do not match at the interface, mapping schemes are needed in order to exchange data from one solver to the other while conserving the correct values. A first distinction in mapping is dependent on the type of constraint, which can be conservative or consistent. The conservative mapping is applied when absolute values such as forces or masses are mapped from two meshes of a different size. In this case, the sum of the quantity at both sides has to be equal. A consistent mapping is used when normalized values such as temperatures or pressures are exchanged. Here, the value of a coarse node must comply with its matching fine node and not the sum of the fine nodes in its vicinity. Due to the fact that a various amount of data mapping methods exist, the description in this work consists of a simple nearest neighbor mapping, a more developed nearest projection mapping and a more complex radial basis function (RBF) mapping. In the nearest neighbor variant, data nodes are mapped simply by connecting all nodes of the first mesh to their closest neighbor of the second mesh and copying the data. While this is an efficient method for grids that are similar at the interface, it obviously has considerable deficits in a more distinct interface configuration. The nearest projection method projects data nodes to elements in a right angle to the surface or volume. Furthermore, the data nodes of the elements are interpolated linearly within the element to match the position of the nodes of the other mesh. In comparison to the nearest neighbor scheme the nearest projection scheme achieves a good accuracy on reasonably different grids. However, the mesh connectivity regarding the pertaining elements has to be known and documented. RBF is a mapping scheme that approximates point cloud data through a linear combination of a polynominal and radial basis functions. In this case the cloud data is associated to the nodes of one mesh. Hence, this approximation is mapped to the other mesh. Radial basis functions can be Guassians, thin-plate splines or multi-quadric splines. Generally, RBF mapping schemes with thin-plate splines are the most accurate variant out of the presented methods and are therefore used in the later described coupling mechanism of the solvers [24, 36, 52].

### Coupling Library

The Interface of choice is the software preCICE (precise Code Interaction Coupling Environment) which consists of a coupling library in `C++` based on a partitioned approach. It provides a generic coupling to multiple solvers by the use of adapters. Parameters concerning the equation coupling schemes and data mapping can be easily adjusted. Furthermore, an adapter for OpenFOAM already exists thus only an adapter for MBDyn has to be developed. A deeper insight into the implementation of preCICE is given in Chapter 4.

# 3. MBDyn

A detailed description of the open-source multibody dynamics analysis software MBDyn is presented in this chapter. This includes the theoretical assumptions and implementations in Section 3.1, the simulation with pre- and postprocessing in Section 3.2 and the introduction of a variety of validation tests in order to examine the performance of MBDyn in Section 3.3.

## 3.1. Theory

MBDyn's fundamental equations are outlined in this section. Furthermore, the structural entities which comprise structural nodes and elements such as shells, membranes, joints and forces are described and their specific syntax is illustrated.

### 3.1.1. Fundamental Equations

The fundamental equations behind the structural dynamics implemented in the software MBDyn consist of the Newton-Euler equations. They define the linear momentum $\boldsymbol{q}$ and the angular momentum $\boldsymbol{\gamma}$ for a set of rigid bodies

$$\boldsymbol{q} = \boldsymbol{M} \cdot \dot{\boldsymbol{x}} \tag{3.1}$$

$$\gamma = \boldsymbol{J}\boldsymbol{\omega} + S \times \dot{\boldsymbol{x}} \tag{3.2}$$

with mass matrix $\boldsymbol{M}$, velocity vector $\dot{\boldsymbol{x}}$, first order inertia moment $S$, second order inertia moment matrix $\boldsymbol{J}$ and angular velocity vector $\boldsymbol{\omega}$. Each structural node that is used in the simulation model (Section 3.1.2) is linked to its own equations for $\boldsymbol{q}$ and $\boldsymbol{\gamma}$. Furthermore, they create their own force and moment equilibrium equations

$$\dot{\boldsymbol{q}} = \boldsymbol{M} \cdot \ddot{\boldsymbol{x}} = \boldsymbol{f} \tag{3.3}$$

$$\dot{\boldsymbol{\gamma}} + \dot{\boldsymbol{x}} \times \boldsymbol{q} = \boldsymbol{m} \tag{3.4}$$

with acceleration vector $\ddot{\boldsymbol{x}}$, force vector $\boldsymbol{f}$ and torque vector $\boldsymbol{m}$ [38, 44].

#### Constrained Equations of Motion

In order to implement joints or boundary conditions, or generally speaking, dependencies between coordinates or their derivatives of different bodies, algebraic constraints in the form of $\boldsymbol{\phi}(\boldsymbol{x}, \dot{\boldsymbol{x}}, t) = 0$ are appended to Equations (3.1) to (3.4). These constraints can be holonomic and thus only dependent on the coordinates $\boldsymbol{x}_n$ and time $t$ (e.g. $\boldsymbol{\phi}(\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \ldots, \boldsymbol{x}_n, t) = 0$) or non-holonomic and thus dependent on any derivative of $\boldsymbol{x}$ with respect to $t$ (e.g. $\boldsymbol{\psi}(\dot{\boldsymbol{x}}_1, \dot{\boldsymbol{x}}_2, \dot{\boldsymbol{x}}_3, \ldots, \dot{\boldsymbol{x}}_n, \boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \ldots, \boldsymbol{x}_n, t) = 0$). These limitations

are, analogous to a mathematical optimization, implemented by the use of Lagrange multipliers. Therefore, in case of an initial value problem which is the only analysis used in this work, the problem is formulated as a system of DAE

$$\boldsymbol{M} \cdot \dot{\boldsymbol{x}} - \boldsymbol{q} = 0 \tag{3.5}$$

$$\dot{\boldsymbol{q}} + \boldsymbol{\phi}_{,\boldsymbol{x}}^T \boldsymbol{\lambda}_\phi + \boldsymbol{\psi}_{,\dot{\boldsymbol{x}}}^T \boldsymbol{\lambda}_\psi - \boldsymbol{f}(\boldsymbol{x}, \dot{\boldsymbol{x}}, t) = 0 \tag{3.6}$$

$$\boldsymbol{\phi}(\boldsymbol{x}, t) = 0 \tag{3.7}$$

$$\boldsymbol{\psi}(\boldsymbol{x}, \dot{\boldsymbol{x}}, t) = 0 \tag{3.8}$$

where $\boldsymbol{\phi}_{,\boldsymbol{x}}^T$ is the transpose of the partial derivative of the holonomic constraint $\boldsymbol{\phi}$ with respect to $\boldsymbol{x}$, $\boldsymbol{\psi}_{,\dot{\boldsymbol{x}}}^T$ is the transpose of the partial derivative of the non-holonomic constraint $\boldsymbol{\psi}$ with respect to $\dot{\boldsymbol{x}}$ and the vector $\boldsymbol{f}$ contains forces and moments [38, 42].

#### Time Integration

This system of DAE is merged into a residual function $\boldsymbol{r}(\boldsymbol{x}, \dot{\boldsymbol{x}}, t) = 0$ and integrated in time using an implicit A/L-stable multistep integration scheme

$$\boldsymbol{x}_k = \sum_{i=1,2} a_i \boldsymbol{x}_{k-i} + \Delta t \sum_{i=0,2} b_i \dot{\boldsymbol{x}}_{k-i} \tag{3.9}$$

where $a$ and $b$ are coefficients that determine the used scheme and $\Delta t$ is the time step. As can be seen by the limits of the summation operator this is a two-step algorithm which results in

$$\boldsymbol{x}_k = a_1 \boldsymbol{x}_{k-1} + a_2 \boldsymbol{x}_{k-2} + \Delta t \left( b_0 \dot{\boldsymbol{x}}_k + b_1 \dot{\boldsymbol{x}}_{k-1} + b_2 \dot{\boldsymbol{x}}_{k-2} \right) \tag{3.10}$$

and is second-order accurate. The solution to this polynomial is achieved by the use of a predictor-correction approach. In the predictor step an initial guess for $\boldsymbol{x}_k$ is predicted via the explicit formulation of Equation (3.10) where the coefficient $b_0 = 0$. Hence, $\boldsymbol{x}_k$ is approximated using an extrapolation of $\dot{\boldsymbol{x}}_k$ solely from previous points

$$\tilde{\dot{\boldsymbol{x}}}_k = \frac{c_1 \boldsymbol{x}_{k-1} + c_2 \boldsymbol{x}_{k-2}}{\Delta t} + d_1 \dot{\boldsymbol{x}}_{k-1} + d_2 \dot{\boldsymbol{x}}_{k-2}. \tag{3.11}$$

In this equation $c$ and $d$ are coefficients for the explicit formulation and $\tilde{\dot{\boldsymbol{x}}}_k$ denotes the predicted value. Inserting Equation (3.11) into Equation (3.10) yields

$$\tilde{\boldsymbol{x}}_k = a_1 \boldsymbol{x}_{k-1} + a_2 \boldsymbol{x}_{k-2} + \Delta t \left( b_0 \tilde{\dot{\boldsymbol{x}}}_k + b_1 \dot{\boldsymbol{x}}_{k-1} + b_2 \dot{\boldsymbol{x}}_{k-2} \right) \tag{3.12}$$

where $\tilde{\boldsymbol{x}}_k$ is an updated value for $\boldsymbol{x}_k$ calculated with a predicted $\dot{\boldsymbol{x}}_k$. In the correction step the Newton iteration

$$\Delta \dot{\boldsymbol{x}}^i = -(\Delta t b_0 \boldsymbol{r}_{,\boldsymbol{x}} + \boldsymbol{r}_{,\dot{\boldsymbol{x}}})^{-1} \boldsymbol{r}(\boldsymbol{x}_k^{i-1}, \dot{\boldsymbol{x}}_k^{i-1}, t_k) \tag{3.13}$$

$$(\Delta t b_0 \boldsymbol{r}_{,\boldsymbol{x}} + r_{,\dot{\boldsymbol{x}}}) \Delta \dot{\boldsymbol{x}}^i = -r(\boldsymbol{x}_k^{i-1}, \dot{\boldsymbol{x}}_k^{i-1}, t_k) \tag{3.14}$$

with the Jacobian Matrix $\boldsymbol{J}_f = (\Delta t b_0 \boldsymbol{r}_{,\boldsymbol{x}} + r_{,\dot{\boldsymbol{x}}})$ and the index $k$ which stands for the current convergence step is executed with the derivatives update

$$\dot{\boldsymbol{x}}_k^i = \dot{\boldsymbol{x}}_k^{i-1} + \Delta \dot{\boldsymbol{x}}^i \tag{3.15}$$

$$\boldsymbol{x}_k^i = \boldsymbol{x}_k^{i-1} + \Delta t b_0 \Delta \dot{\boldsymbol{x}}^i \tag{3.16}$$

and the corrections $\Delta \dot{\boldsymbol{x}}^i$ to $\dot{\boldsymbol{x}}^i$ [37, 38, 41, 42, 53].

#### Orientation Handling

Another important aspect in computational multibody dynamics is the orientation of each structural node due to finite rotations during the simulation. The parametrization of an incremental rotation in MBDyn is accomplished by the use of Cayley-Gibbs-Rodriguez (CGR) parameters. Their basis is the conventional rotation matrix $\boldsymbol{R}$ and its differentiation matrix $\boldsymbol{C}$

$$\boldsymbol{R} = \boldsymbol{I} + \sin(\theta)\boldsymbol{D} + (1 - \cos(\theta))\boldsymbol{D}^2 \tag{3.17}$$

$$\boldsymbol{C} = \boldsymbol{I} + \frac{1 - \cos(\theta)}{\theta}\boldsymbol{D} + \left(1 - \frac{\sin(\theta)}{\theta}\right)\boldsymbol{D}^2 \tag{3.18}$$

for a vector $\Theta = \theta d$ with an angle $\theta$ about the axis $d$. In order to eliminate the trigonometric functions in the calculation of these two matrices, the CGR parameters $g$ are defined as

$$\boldsymbol{g} = 2 \cdot \tan\left(\frac{\theta}{2}\right) d \tag{3.19}$$

yielding a modified $\boldsymbol{R}$ and $\boldsymbol{C}$

$$\boldsymbol{R}_{\mathrm{CGR}} = \boldsymbol{I} + \frac{4}{4 + \boldsymbol{g}^T\boldsymbol{g}}\left(\boldsymbol{C} + \frac{1}{2}\boldsymbol{C}^2\right) \tag{3.20}$$

$$\boldsymbol{C}_{\mathrm{CGR}} = \frac{4}{4 + \boldsymbol{g}^T\boldsymbol{g}}\left(\boldsymbol{I} + \frac{1}{2}\boldsymbol{C}\right). \tag{3.21}$$

The disappearance of the trigonometric functions improves the computational performance of the calculation of the rotation matrix considerably[38, 41, 42].

#### Solution Steps

The DEA consisting of Equations (3.5) to (3.8) is subsequently used in the solution process. As a first step the initial assembly of the model is examined for consistency pertaining to

holonomic and non-holonomic constraints. The equations

$$\boldsymbol{K}_p(\boldsymbol{x}-\boldsymbol{x}_0)+\boldsymbol{\phi}_{,\boldsymbol{x}}^T\boldsymbol{\lambda}_\phi=\boldsymbol{f}_p \tag{3.22}$$

$$\boldsymbol{K}_v(\dot{\boldsymbol{x}}-\dot{\boldsymbol{x}}_0)+\boldsymbol{\phi}_{,\boldsymbol{x}}^T\boldsymbol{\lambda}_\phi+\boldsymbol{\psi}_{,\dot{\boldsymbol{x}}}^T\boldsymbol{\lambda}_\psi=\boldsymbol{f}_v \tag{3.23}$$

$$\boldsymbol{\phi}\left(\boldsymbol{x},t\right)=0 \tag{3.24}$$

$$\boldsymbol{\psi}(\boldsymbol{x},\dot{\boldsymbol{x}},t)=0 \tag{3.25}$$

$$\boldsymbol{\phi}_{,\boldsymbol{x}}\dot{\boldsymbol{x}}+\boldsymbol{\phi}_{,t}=0 \tag{3.26}$$

are correcting the deviation of the model from the initial state by the matrices $\boldsymbol{K}_p$ in Equation (3.22) for the position and orientation and $\boldsymbol{K}_v$ in Equation (3.23) for the linear and angular velocities with respect to the external loads $\boldsymbol{f}_p$ and $\boldsymbol{f}_v$. If necessary, the three constraint functions in Equations (3.24) to (3.26) aid the solver in finding new values for the position and velocity to comply with the constraints and finally changing the initial configuration [42].

The second step consists of the computation of the initial value of the derivatives prior to any iteration in order to ensure that the differential equations are satisfied. Thus, the initial value problem, formulated as an implicit DAE $\boldsymbol{r}(\boldsymbol{x},\dot{\boldsymbol{x}},t)=0$, is solved for $\dot{\boldsymbol{x}}$ via a modified correction iteration of Equation (3.14)

$$(\Delta t b_0\boldsymbol{r}_{,\boldsymbol{x}}+\boldsymbol{r}_{,\dot{\boldsymbol{x}}})\Delta\dot{\boldsymbol{x}}^i=-\boldsymbol{r}(\boldsymbol{x}_0^{i-1},\dot{\boldsymbol{x}}_0,t_0).[42] \tag{3.27}$$

Next, the DAE is integrated by an implicit linear multistep integration scheme. Since this integration scheme is not able to start the first step on its own, a Crank-Nicolson scheme is used for the first step with the coefficients $a_1=1$ and $b_0=b_1=1/2$ for Equation (3.9). This results in

$$\boldsymbol{x}_k=\boldsymbol{x}_{k-1}+\frac{\Delta t}{2}\dot{\boldsymbol{x}}_k+\dot{\boldsymbol{x}}_{k-1}. \tag{3.28}$$

Hence, the subsequent steps can be integrated via the multistep scheme [42].

### 3.1.2. Structural Nodes

MBDyn implements various sorts of nodes such as electric, hydraulic or thermal variants. However, the most important nodes for this work are of structural type. They are a fundamental entity considering the architecture of a simulation model due to the fact that they provide the DOF and hence define the bodies' motion in space. Furthermore, as described in Section 3.1.1, structural nodes create and store the equilibrium equations in Equations (3.3) and (3.4) and, if applicable, the definitions for the linear and angular momentum in Equations (3.1) and (3.2). The corresponding equations and DOF can be accessed by elements (see Section 3.1.3) which in turn can contribute to the equations. There exist six different structural nodes, namely static, dynamic, modal, dummy and, resulting from the introduction of membrane elements, static displacement and dynamic displacement nodes.

### Dynamic and Modal Nodes

Dynamic nodes possess mass and thereby are affected by inertia. They can have up to six DOF. These include the lateral motion in all three directions and the rotation about all three axes ultimately describing a spatial rigid-body motion. Thus, they store information about their current position and orientation and properties regarding inertia. Modal nodes have the same properties and input variables as their dynamic counterpart. However, they can only be connected to a modal joint or attached to the nodes of a constitutive FEM model. A modal analysis focuses on the dynamic response of structural systems regarding frequencies. This type of node is only mentioned for the sake of completeness and is not used in the simulations.

### Dynamic Displacement Node

Constraining the rotational DOF converts a dynamic node into a so called dynamic displacement node. This variant describes the spatial point mass motion and thus does not create equations for the angular momentum and the moment equilibrium (Equations (3.2) and (3.4)). It was implemented in the code to support the use of membrane elements which by definition do not own rotational DOF. The dynamic displacement node is also referred to as a displacement-only node. It should be stressed that membrane elements can be connected to both displacement-only and conventional dynamic nodes. However, shell elements can in fact not be connected no rotationless nodes.

### Static and Dummy Nodes

If no inertia is needed or intended, a static or a static displacement node can be deployed. These variants do not own DOF and therefore have to be correctly constrained in the reference frame or linked to elements. Furthermore, for static nodes the left-hand side of Equations (3.3) and (3.4), i.e. the derivatives of the linear and angular momentum, is zero. Lastly, dummy nodes are deployed to directly output the motion of other preselected structural nodes. They represent an additional feature of MBDyn to facilitate the post-processing. Dummy nodes do not own any DOF either and have to be attached to other structural nodes that are not of dummy type.

### Syntax

The typical syntax with the essential parameter for the input file regarding a structural node with six DOF is given below.

```
structural: node label, node type,
    position,
    orientation,
    orientation description,
    velocity,
    angular velocity;
```

This type of node owns a unique node label, the desired type and data about its current position, velocity and angular velocity as a three-dimensional vector and its orientation, all with respect to the global reference frame. The orientation can be supplied in the form of euler angles, an orientation vector or a $3 \times 3$ orientation matrix. An example of the syntax for a dynamic node is

```
structural: 1, dynamic,
    1.0, 1.0, 0.0,
    eye,
    null,
    null;
```

with the node label 1, a vector for the position $(1\ 1\ 0)^T$, no rotation with respect to the global reference frame and no initial velocity nor angular velocity. The `eye` statement returns an identity matrix and `null` a vector filled with zeros. The simplicity of this example is due to the fact that for the simulations no initial rotation or velocity had to be defined. A more complex example is

```
set: real Alpha = pi/2;
set: real v = 2;
set: real Omega = 1;
structural: 2, dynamic,
    .4, .1, .8,
    1, 0., 1., 0., 2, cos(Alpha), 0., sin(Alpha),
    0., 0., v,
    Omega, 0., 0.;
```

with an initial rotation, expressed as an orientation matrix, an initial velocity in $z$-direction and an angular velocity about the $x$-axis. The specified orientation matrix defines a new orthonormal reference system with the description of two vectors. In this case the first vector points into $y$-direction of the global frame. The second represents a rotation of $\pi/2$ about the global $y$-direction. The third vector is normal to the plane that the two previous vectors are creating and is calculated from their cross product [40, 41, 42].

### 3.1.3. Elements

Alongside nodes, elements are the most important entity in constructing a simulation model in MBDyn. Amongst others, there exist structural, aerodynamic, electric, hydraulic or thermal elements. However, structural elements are by far the most important and interesting for this work. They include bodies, joints, beams, plate elements and forces. In general, elements can access nodal properties and write their own contributions to the equations. Additionally, they are also able to add specific equations such as holonomic and non-holonomic algebraic constraints.

#### Bodies

In order to define a rigid body, properties regarding the mass, the center of mass and inertia have to be assigned. This is accomplished by connecting a body element to each

non-static structural node. The syntax for this element considering six DOF nodes is

```
body: body label,
    node label,
    mass,
    relative center of mass,
    inertia matrix;
```

and considering three DOF nodes, i.e. displacement-only nodes,

```
body: body label,
    node label,
    mass;
```

with a value for the mass, a three-dimensional vector for the relative offset of the center of mass with respect to the referenced node and a $3 \times 3$ inertia matrix. Since no inertia is defined for displacement-only nodes, no center of mass is needed. The inertia matrix consists of the three principal moments of inertia around each axis in the main diagonal and the product moments in the secondary diagonal. For a six DOF node body element the syntax may result in

```
body: 10,
    1,
    8,
    0., 0., 0.,
    diag, .4, .4, .2;
```

with a mass of 8, no offset with respect to the center of gravity and an inertia matrix of

$$\boldsymbol{J} = \begin{pmatrix} 0.4 & & \\ & 0.4 & \\ & & 0.2 \end{pmatrix}$$

with the principal moments of inertia [40].

### Joints

The kinematics and the DOF of structural dynamic nodes can be constrained by the use of joint elements. MBDyn offers a great range of different joints that can impose the lateral and rotational motion of single nodes or the relative position and orientation between two discrete nodes. The three most important joints for this work, a clamp, a total pin joint and a deformable displacement joint, are discussed in the following.
A clamp is a particularly simple joint that fixes every DOF of one node. The syntax is

```
joint: joint label, joint type,
    node label,
    position,
    orientation;
```

including a position vector and an orientation matrix. However, most of the time the

clamp adopts the position and orientation of the node it fixes. This is accomplished by the keyword `node` as the exemplary syntax

```
joint: 100, clamp,
    2,
    node,
    node;
```

where a clamp element fixes all DOF of node 2 suggests.

The total pin joint can constrain the position and orientation of a structural dynamic node as well. However, its flexibility makes for an extremely useful joint element. The syntax of the total pin joint

```
joint: joint label, joint type,
    node label,
        position,
        position orientation,
        rotation orientation,
    position,
    position orientation,
    rotation orientation,
    position constraint,
        position status,
    orientation constraint,
        orientation status;
```

is more complex starting with the joint label and type followed by the node label the total pin joint refers to and the relative position, position orientation and rotation orientation in case a reference frame was used. The next three entries are the absolute position and orientation, i.e. a description in the global reference frame. When no local reference system was defined the following entries for the relative position and orientation are identical to the previous statements. A constraint regarding the position and orientation of the node can be triggered by the position and orientation status respectively. It includes the keywords `active` and `inactive` and these keywords are chosen separately for a translational motion in any axis and a rotation about any axis. This is illustrated by an exemplary total pin joint

```
joint: 101, total pin joint,
    3,
        position, null,
        position orientation, eye,
        rotation orientation, eye,
    position, null,
    position orientation, eye,
    rotation orientation, eye,
    position constraint,
        active, inactive, active, null,
    orientation constraint,
        inactive, active, inactive, null;
```

that constrains any displacements in $x$ and $z$-direction and any rotation about $y$ of node 3. The keyword `null` in the status entry prevents the solver to implement any kind of template driver to impose the absolute position and orientation of the node. A template driver is a highly flexible pattern for any kind of prescribed value along or about any axis.

A deformable displacement joint applies a configuration dependent force between two points that may be optionally offset from two nodes by means of a 3D constitutive law. The relative position of these nodes determines the absolute value of the force. In the simulations of this work this joint is used as a way of damping dynamic nodes with respect to a grounded static node. The syntax of this joint is

```
joint: joint label, deformable displacement joint,
    node 1 label,
        position,
        orientation,
    node 2 label,
        position,
        orientation,
    constitutive law;
```

where two points, their relative position and optionally their relative orientation are defined. A constitutive law is a characteristic material model. There exist several constitutive laws that can be looked up in the MBDyn Input File Format [40]. The one that is used in this work is a linear viscoelastic isotropic constitutive law

```
linear viscoelastic isotropic,
    stiffness,
    viscosity;
```

with a value for the stiffness and the viscosity. In MBDyn a constitutive law can not exist on its own but has to be instantiated by structural entities such as nodes or elements. A combined example is

```
joint: 102, deformable displacement joint,
    0,
        null,
    3,
        null,
    linear viscoelastic isotropic,
        0,
        5;
```

where the points 0 and 3 coincide with the position of the nodes respectively and certain values for stiffness and viscosity are defined [40].

### Beams

Deformable, slender beams are implemented in MBDyn by means of a FV approach. Currently, two and three-node beam elements are available. They are described by reference points linked to structural nodes and evaluation points in between, i.e. two

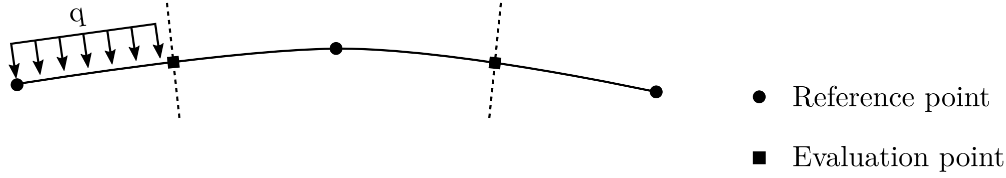



Figure 3.1.: Configuration of a three-node beam in MBDyn (cf. [27])

evaluation points for the three-node beam as shown in Figure 3.1 and one evaluation point for the two-node beam element. Each of the beam's nodes is linked to a structural node and optionally imposed by an offset or a relative orientation. Any external force, e.g. a uniform load in Figure 3.1, or moment is integrated over the appropriate beam section and finally converted into nodal forces. The reader is referred to the paper from Ghiringhelli et al. [27] for further information about the beam element [27].

### Shells

MBDyn offers a $C^0$ four-node geometrically non-linear shell element to construct 2D models. A $C^0$ compliance implies that if the domain deforms, no gap or overlap between elements must arise. Its formulation is derived from a shell element developed by Witkowski [65] which is based on the modified Hu-Washizu variational functional [27, 38, 40].
The virtual internal work $\delta W$ of the shell is

$$\delta W = \int_A \delta \boldsymbol{\epsilon}^T \boldsymbol{\sigma} \, \mathrm{d}S \tag{3.29}$$

where $\boldsymbol{\epsilon}$ represents the strain vector in Voigt notation

$$\boldsymbol{\epsilon} = \begin{pmatrix} \tilde{\boldsymbol{\epsilon}}_1 \\ \tilde{\boldsymbol{\epsilon}}_2 \\ \tilde{\boldsymbol{\kappa}}_1 \\ \tilde{\boldsymbol{\kappa}}_2 \end{pmatrix} \tag{3.30}$$

with strain $\tilde{\boldsymbol{\epsilon}}_i$ and curvature $\tilde{\boldsymbol{\kappa}}_i$ and the stress vector $\boldsymbol{\sigma}$

$$\boldsymbol{\sigma} = \begin{pmatrix} \boldsymbol{n}_1 \\ \boldsymbol{n}_2 \\ \boldsymbol{m}_1 \\ \boldsymbol{m}_2 \end{pmatrix} \tag{3.31}$$

with forces $\boldsymbol{n}_i$ and moments $\boldsymbol{m}_i$ per unit length. The strain is computed by the deviation between the deformed and undeformed derivative of the position $\boldsymbol{x}_{,k}$ and $\boldsymbol{x}_{0,k}$

$$\tilde{\boldsymbol{\epsilon}}_k = \boldsymbol{T}^T \boldsymbol{x}_{,k} - \boldsymbol{T}_0^T \boldsymbol{x}_{0,k} \tag{3.32}$$

where $k$ is the arc length coordinate $k = 1, 2$ and $\boldsymbol{T}$ is the orientation triad. Similarly, the equation for the curvature is

$$\tilde{\boldsymbol{\kappa}}_k = \boldsymbol{T}^T \boldsymbol{\kappa}_k - \boldsymbol{T}_0^T \boldsymbol{\kappa}_{0k} \tag{3.33}$$

where $\boldsymbol{\kappa}_k$ is a vector that characterizes the spatial derivative with respect to the arc length $k$ of the orientation tensor $\boldsymbol{T}$ [43]. For a detailed representation of the exact computation including among others the interpolation of the position and orientation of the shell and its nodes, the linearization of the virtual work and the determination of the Jacobian matrix and residual the reader is referred to MBDyn's Theory and Developer's Manual [41].

The finite element discretization for the four-node shell element in dimensionless coordinates $\xi$ and $\eta$ in the range $[-1, 1]$ is shown in Figure 3.2. The quadrilateral element consists of four nodes which are numbered counterclockwise. Moreover, the elements can adopt any quadrilateral shape and do not have to be quadratic as the figure might imply. According to that, the bilinear shape functions can be defined as

$$\begin{aligned} N_1(\xi, \eta) &= \frac{1}{2}(1+\xi) \cdot \frac{1}{2}(1+\eta) \\ &= \frac{1}{4}(1+\xi)(1+\eta) \\ N_2(\xi, \eta) &= \frac{1}{4}(1-\xi)(1+\eta) \\ N_3(\xi, \eta) &= \frac{1}{4}(1-\xi)(1-\eta) \\ N_4(\xi, \eta) &= \frac{1}{4}(1+\xi)(1-\eta)\,. \end{aligned} \tag{3.34}$$

Furthermore, MBDyn shell elements are modified with an enhanced assumed strain (EAS) and an assumed natural strain (ANS) formulation in order to reduced volumetric and shear locking effects respectively.

The relationship between stress and strain, i.e. the constitutive law of an isotropic shell, yields

$$\begin{pmatrix} n_{11} \\ n_{12} \\ n_{13} \\ n_{21} \\ n_{22} \\ n_{23} \\ m_{11} \\ m_{12} \\ m_{13} \\ m_{21} \\ m_{22} \\ m_{23} \end{pmatrix} = \begin{pmatrix} C & & & & C\nu & & & & & & & \\ & 2Gh & & & & & & & & & & \\ & & \alpha Gh & & & & & & & & & \\ & & & 2Gh & & & & & & & & \\ C\nu & & & & C & & & & & & & \\ & & & & & \alpha Gh & & & & & & \\ & & & & & & 2F & & & & & \\ & & & & & & & D & & -D\nu & & \\ & & & & & & & & \beta F & & & \\ & & & & & & & -D\nu & & D & & \\ & & & & & & & & & & 2F & \\ & & & & & & & & & & & \beta F \end{pmatrix} \begin{pmatrix} \epsilon_{11} \\ \epsilon_{12} \\ \epsilon_{13} \\ \epsilon_{21} \\ \epsilon_{22} \\ \epsilon_{23} \\ \kappa_{11} \\ \kappa_{12} \\ \kappa_{13} \\ \kappa_{21} \\ \kappa_{22} \\ \kappa_{23} \end{pmatrix} \tag{3.35}$$

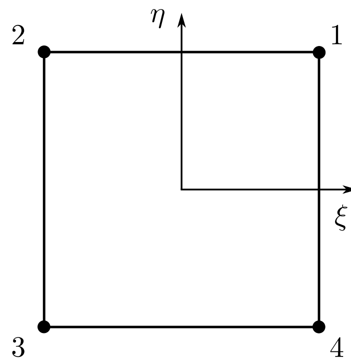



Figure 3.2.: FEM Discretization of a four-node shell or membrane element in MBDyn (cf. [40])

with the Young's modulus $E$, Poisson's number $\nu$, shell thickness $h$, shear modulus $G = E/\left(2\left(1+\nu\right)\right)$ and constants $C = Eh/\left(1-\nu^2\right)$, $D = Ch^2/12$ and $F = Gh^3/12$.
An example for the syntax of a shell with an isotropic constitutive law is

```
shell4easans: joint label,
    node 1 label, node 2 label, node 3 label, node 4 label,
    constitutive law;
```

with four nodes connected as illustrated in Figure 3.2 and a constitutive law that can either be isotropic or orthotropic. For the orthotropic type the matrix has to be inserted manually. The syntax for a typical isotropic shell element is

```
shell4easans: 0,
    1, 2, 3, 4
    isotropic,
        E, 270 000,
        nu, 0.4,
        thickness, 0.001;
```

which is constructed from four nodes with an element thickness of 0.001. Due to the fact that it is an isotropic shell, only two out of Young's modulus $E$, Poisson's number $\nu$ and shear modulus $G$ have to be specified [41, 43, 65].

### Membranes

Generally, membrane elements are used to model thin structures with next to none flexural stiffness. In contrast to shell elements that sustain transverse forces with bending stresses, membranes carry them through in-plane stresses. For extremely thin structures in MBDyn, the use of membrane elements instead of thin shell elements yields a reduction in computational cost. The formulation for the membrane element is similar to that of the shell. Additionally, it shares its FEM discretization and notation of Figure 3.2. However, the membrane strains are computed by the use of the nonlinear Green-Lagrange strain

tensor for large deformations

$$\boldsymbol{E} = \frac{1}{2}\left(\boldsymbol{F}^T\boldsymbol{F} - \boldsymbol{I}\right) \tag{3.36}$$

where $\boldsymbol{F}$ is the deformation gradient. Equation (3.36) can also be defined as

$$\epsilon_{ij} = \frac{1}{2}\left(u_{i,j} + u_{j,i} + \boldsymbol{u}_{,i}^T\boldsymbol{u}_{,j}\right) \tag{3.37}$$

with the deformation vector $\boldsymbol{u} = (u_x, u_y, u_z)^T = (u, v, w)^T$. Equation (3.37) yields for the strain vector

$$\epsilon = \begin{pmatrix} \epsilon_x \\ \epsilon_y \\ \gamma_{xy} \end{pmatrix} = \begin{pmatrix} \epsilon_{xx} \\ \epsilon_{yy} \\ 2\epsilon_{xy} \end{pmatrix} \tag{3.38}$$

with the strain components

$$\begin{aligned} \epsilon_x &= \frac{1}{2}\left(u_{,x} + u_{,x} + [u_{,x}\ v_{,x}\ w_{,x}][u_{,x}\ v_{,x}\ w_{,x}]^T\right) \\ &= u_{,x} + \frac{1}{2}\left(u_{,x}^2 + v_{,x}^2 + w_{,x}^2\right) \\ \epsilon_y &= v_{,y} + \frac{1}{2}\left(u_{,y}^2 + v_{,y}^2 + w_{,y}^2\right) \\ \gamma_{xy} &= 2 \cdot \frac{1}{2}\left(u_{,y} + v_{,x} + [u_{,x}\ v_{,x}\ w_{,x}][u_{,y}\ v_{,y}\ w_{,y}]^T\right) \\ &= u_{,y} + v_{,x} + u_{,x}u_{,y} + v_{,x}v_{,y} + w_{,x}w_{,y}. \end{aligned} \tag{3.39}$$

The constitutive law of an isotropic membrane is

$$\begin{pmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{pmatrix} = \begin{pmatrix} C & C\nu & \\ C\nu & C & \\ & & 2D \end{pmatrix} \begin{pmatrix} \epsilon_y \\ \epsilon_y \\ \gamma_{xy} \end{pmatrix} + \text{prestress} \tag{3.40}$$

with a constant $C = Eh/\left(1 - \nu^2\right)$ and $D = Gh$ including Young's modulus $E$, Poisson's ratio $\nu$, shear modulus $G$ and membrane thickness $h$. Furthermore, a MBDyn membrane element can be prestressed in span direction in order to improve the numerical stability when applying transversal forces due to the fact that per definition a membrane lacks flexural stiffness. Membrane elements are modified with an EAS formulation only. The syntax is almost identical to that of a shell element

```
membrane4eas: joint label,
    node 1 label, node 2 label, node 3 label, node 4 label,
    constitutive law,
        prestress;
```

and as an prestressed example

```
membrane4eas: 0,
```

```
        5, 6, 7, 8
        isotropic,
            E, 588 000 000,
            nu, 0.34,
            thickness, 0.0006,
            prestress, 1, 1, 0;
```

where the prestress values are contributions to the $x$, $y$, and $xy$ direction respectively, under the assumption that the membrane lies in the $x, y$-plane. [39, 40, 58].

### Forces

Forces and moments generally add a right-hand side to the equations of motion in Equations (3.3) and (3.4). Once again MBDyn offers a great range of force and torque elements and, in order to not go beyond the scope, only structural and external forces are described. Structural forces can be solely applied to structural dynamic nodes. The syntax of a structural force is

```
    force: force label, force type,
        node label,
            position,
            force;
```

with a force type, the node's label to which the force is applied to, the position of the force and finally its direction and value. Currently, follower and absolute type forces can be instantiated with the former being a force that follows the deformed shape of the structure and always stays perpendicular to it and the latter keeping its initial direction throughout the whole simulation. The position of the force describes the offset to the structural node in a $3 \times 1$ vector. The last entry sets the direction of the force, its value and furthermore can contain a drive caller which determines how the force is applied. This includes e.g. a constant force, a force that is ramped up in a certain period of time or a force that is applied in an incremental manner. An absolute force in MBDyn is defined by

$$f = \text{real} \tag{3.41}$$

and a follower force by

$$f = \boldsymbol{R} \cdot \text{real} \tag{3.42}$$

with a value of the force as a vector and the orientation matrix $\boldsymbol{R}$ of the corresponding node. Two examples for an absolute

```
    force: 1, absolute,
        1,
            position, null,
            0., 0., 1., constant, 2.;
```

and a follower force

```
force: 2, follower,
    2,
        position, null,
        0.707, 0.707, 0., ramp, 0.5, 0, 4, 1;
```

with different drive callers are presented. The position of the force relative to the node is zero in both examples. The first force is directed along the $z$-axis of the global reference frame with a constant value of 2 throughout the whole simulation. The second one is directed along the $x$ and $y$ axis of the global reference frame in a 45° angle and ramped up from 0 to 4 seconds with a slope of 0.5 and an initial value of 1. Thus, in the end the force has a value of 3.

The mentioned forces are exclusively compatible to structural dynamic nodes. In case of membrane elements and the corresponding displacement-only nodes MBDyn offers a special absolute displacement force that is identical to the structural absolute force except that the position parameter is omitted since no rotational DOF exist. Hence, the force can only be applied directly to the node. A simple example is

```
force: 3, absolute displacement,
    3,
        0., 0., 1., ramp, 20.5, 0., 1., 0.;
```

where the position parameter is discarded and a `ramp` function is applied. Due to the fact that displacement-only nodes do not own rotational DOF there exists no counterpart to the follower force in MBDyn [40].

An external structural force is used to prescribe forces that are calculated by an external software. The two universal communication options are file and socket based. A third option exists, however it is developed in particular for the CFD solver Edge and thus of no importance to this work. With respect to FSI the socket based option is by far the most suitable and is therefore the only external force described in detail in this section. For the other types the reader should consider the MBDyn Input File Format [40]. Generally, in network programming, a socket establishes the communication points of bidirectional data exchange between two processes [67]. The syntax for the local socket communication based force is typically

```
force: force label, external structural
    socket,
    create, (yes|no),
    path,
    coupling,
    orientation,
    accelerations,(yes|no),
    number of nodes,
        node label(s);
```

where the forces are sent through a transmission control protocol (TCP) socket to the nodes according to the listed node labels. The `create` entry determines whether MBDyn initiates the socket after the input file is read with a local path (`yes`) or whether it tries to

connect to a previously created socket at the path right away (`no`). Regarding the coupling between MBDyn and its peer, two different communication schemes can be selected, namely a loose and a tight coupling. For a loose coupling the communication occurs at each time step whereas for a tight coupling it occurs at each iteration. The keyword `orientation` followed by the manner in which the orientation of the nodes is defined, i.e. in euler angles, an orientation vector or an orientation matrix, is set to determine the output structure. The output of the accelerations of the nodes can be toggled once again by the keywords `yes` or `no`. Some optional parameters and not yet implemented features in the syntax have been omitted for the sake of convenience. An exemplary external structural force is

```
force: 2, external structural
    socket,
    create, yes,
    path, "\$MBSOCK",
    no signal,
    coupling, tight,
    orientation, orientation vector,
    accelerations, yes,
    4,
        1,
        2,
        3,
        4;
```

with forces defined by the peer, sent through the socket and applied to nodes 1, 2, 3 and 4. The additional statement `no signal` disables the system's SIGPIPE signal in case one of the participants' communication channel shuts down, which prevents the simulation from crashing completely [40].

Due to the fact that follower forces are of great importance when applying a uniform pressure, a solution for the lack of this type of force for membrane elements in MBDyn has been developed. The idea is to compute the current spatial orientation of the membrane surface with the position of its four nodes (consider Figure 3.2) and consequentially defining the normal vector to this plane. Hence, this normal vector represents the direction of the follower force. Since the position of each node at every time step and iteration can be accessed via a `Python` interface at runtime (Section 3.2), the force vector for each node is calculated separately at every time step and transmitted to MBDyn. The procedure printed in detail in Appendix B.5 is based on the method of Least Squares from the technical report of Miller [45] and extended to find the averaging plane. The coefficients $a$, $b$ and $c$ for the equation of a plane

$$ax + by + d = cz \tag{3.43}$$

are solved according to

$$\begin{pmatrix} a \\ b \\ d \end{pmatrix} = \left(\boldsymbol{A}^T \boldsymbol{A}\right)^{-1} \boldsymbol{A}^T \boldsymbol{b}. \tag{3.44}$$

The coefficient $c$ is arbitrarily set to 1 due to the fact that this equation with four nodes is overdetermined. The matrix

$$\boldsymbol{A} = \begin{pmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{pmatrix} \tag{3.45}$$

contains the $x$ and $y$ position of the four nodes and vector

$$\boldsymbol{b} = \begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} \tag{3.46}$$

contains the $z$ position. The normal vector $n$ to this plane

$$\boldsymbol{n} = \begin{pmatrix} a \\ b \\ c \end{pmatrix} \tag{3.47}$$

consists of the computed coefficients. In order to get a more accurate vector, the normals

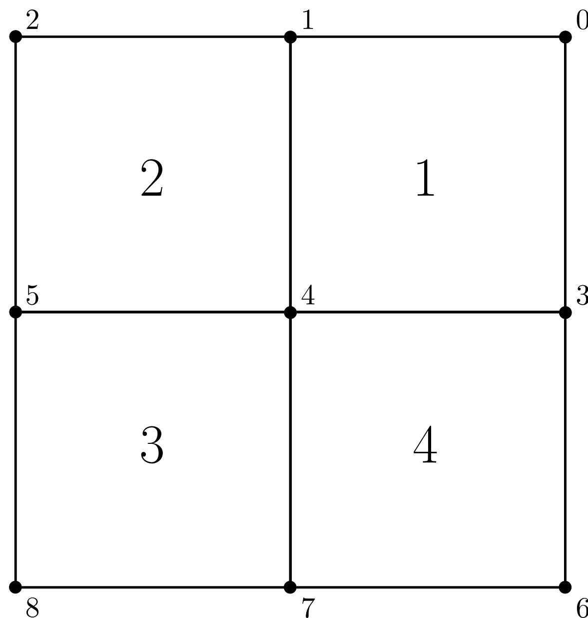



Figure 3.3.: Node surrounded by four membrane elements; the small numbers represent the node labels and the large numbers represent the element labels

to all membrane elements surrounding the concerned node are averaged according to

$$\overline{\boldsymbol{n}}_l = \frac{1}{i}\sum_{k=1}^{i}\boldsymbol{n}_k = \begin{pmatrix}\overline{a}\\ \overline{b}\\ \overline{c}\end{pmatrix} \tag{3.48}$$

where $l$ is the node label and $i$ the number of elements around the node. For node 4 in Figure 3.3 surrounded by elements 1, 2, 3 and 4 the averaged normal would be

$$\overline{\boldsymbol{n}}_4 = \frac{1}{4}\left(\boldsymbol{n}_1 + \boldsymbol{n}_2 + \boldsymbol{n}_3 + \boldsymbol{n}_4\right). \tag{3.49}$$

Hence, when the value of the applied force $p$ is known, the nodal force vector $\boldsymbol{f}_l$ can be determined by

$$\boldsymbol{f}_l = p \cdot \frac{\overline{\boldsymbol{f}}_l}{|\overline{\boldsymbol{f}}_l|} = p \cdot \hat{\overline{\boldsymbol{f}}}_l \tag{3.50}$$

where $\hat{\overline{\boldsymbol{f}}}_l$ is the normalized vector of $\overline{\boldsymbol{f}}_l$. This procedure is implemented in the validation tests of Hencky's problem (Section 3.3.3) and the airbag inflation (Section 3.3.4). Special attention has to be paid to the direction of the normal vector as it can easily point to the opposite direction. However, in order to use this modification for membrane elements, no displacement-only nodes can be utilized for the construction of the model due to the fact that external structural forces are not compatible with three DOF nodes. A workaround for this issue is to model membrane elements with dynamic structural six DOF nodes and constrain all rotational DOF of each node with a total pin joint (Section 3.1.3). Hence, follower forces can be applied to membrane elements. However, the advantages of the displacement-only nodes, e.g. a faster computation, are lost [45].

## 3.2. Simulation

In this section the necessary steps for the execution and the implementation of a MBDyn simulation are listed and described. This includes the preprocessing in Section 3.2.1 in form of an input file that is passed to MBDyn, the resulting output and its postprocessing in Section 3.2.2 and a `Python` interface in Section 3.2.3 to communicate at runtime. Furthermore, the implementation of the main classes of the source code and the solution process are outlined in a concise manner in Section 3.2.4.

### 3.2.1. Preprocessing

Since MBDyn does not offer an own preprocessing feature, the input file structure has to be assembled manually in a conventional text-file, usually but not necessarily with a .mbd extension. It consists of five different blocks, namely data, problem, control data, nodes and elements. Each block is encapsulated in a `begin: block` and an `end: block` statement. Furthermore, every line has to end with a semicolon. The syntax for one node (Section 3.1.2) or element (Section 3.1.3) is actually considered as one line and thus only

has one semicolon at the end of the definition.

### Data and problem related data

In the data block the type of problem is defined with the syntax

```
begin: data;
    problem;
end: data;
```

including currently two options, namely an initial value and an inverse dynamics problem. Generally, an initial value problem is solved by computing the displacements based on applied forces while an inverse dynamics problem is solved by reconstructing the forces which were necessary for a certain displacement. In this work only problems of the initial value type are considered.

The problem block consists of general and numerical data regarding the solution process of the chosen problem. A selection of important entries for an initial value problem with exemplary values is

```
begin: initial value;
    initial time: 0;
    final time: 1;
    time step: 0.01;
    method: ms, 0.6;
    tolerance: 1.e-6, 1.e-6;
    max iterations: 100;
    linear solver: umfpack;
    nonlinear solver: newton raphson;
    output: iterations;
end: initial value;
```

however many more exist and can be looked-up in MBDyn's Input File Format [40]. The `method` entry defines a linear multistep (`ms` ) integration scheme (see Equation (3.9)) with an algorithmic dissipation factor of 0.6 which results in good accuracy while maintaining a reasonable amount of numerical dissipation. The tolerances for the residual and solution tests respectively as well as the maximum amount of iterations are set. For the computation of linear systems and nonlinear problems a linear solver called "umfpack" and a Newton-Raphson scheme are chosen respectively. Finally, the `output: iterations` entry prints the convergence to the standard output of the system [40].

### Control data

In the control data block every instance of nodes or elements has to be listed due to the fact that the solver allocates memory according to the amount that is used. A simple syntax with exemplary values is

```
begin: control data;
    structural nodes: 261;
    beams: 261;
```

```
        plates: 236;
        joints: 48;
        rigid bodies: 261;
        forces: 213;
        output frequency: 10;
        # model: static;
    end: control data;
```

with all structural entities necessary for the model. Furthermore, global parameters such as the frequency of the output data or its precision can be defined here. Conveniently, if a dynamic analysis is to be changed to a static one, the entry `model: static` can be used instead of changing all structural dynamic nodes to static nodes [40].

#### Nodes and elements

The blocks for nodes and elements contain all structural entities including their distinct information that are used in the simulation and thus construct the actual model. The syntax is

```
    begin: nodes;
        node data;
    end: nodes;

    begin: elements;
        element data;
    end: elements;
```

with the data syntax as described in Section 3.1.2 and Section 3.1.3. An exemplary .mbd file is located in Appendix B.1, however due to the dimension of the file, only the first and last two entities of the nodes and elements are printed.

In order to reduce the writing effort the .mbd file is created by the use of a python script which is printed in Appendix B.2. It converts a mesh (.msh), created by Gmsh [26], into a text file (.mbd) ready to be executed by MBDyn. This script is fitted for Hencky's problem (consider Section 3.3.3) including a uniform pressure and membrane elements. However, a modification for shell elements and other force configurations is trivial.

The nodes and elements from the .msh file are read in and reformulated in order to resemble the syntax from Section 3.1.2 and Section 3.1.3. For the calculation of the uniform pressure, the areas of the elements are needed. Since Hencky's problem is initially a circular membrane, the mesh consists of unstructured quadrilaterals and one of them is schematically illustrated in Figure 3.4. It consists of four nodes 1, 2, 3 and 4 whose coordinates are available as a position vector $\boldsymbol{r}$ and four sides $l_1$, $l_2$, $l_3$ and $l_4$. First, the

length of the sides has to be calculated as the norm of the vector they are creating

$$\begin{aligned} l_1 = |\boldsymbol{r}_1 - \boldsymbol{r}_2| &= \left| \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} - \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} \right| \\ &= \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \\ l_2 &= \sqrt{(x_2 - x_3)^2 + (y_2 - y_3)^2 + (z_2 - z_3)^2} \\ l_3 &= \sqrt{(x_3 - x_4)^2 + (y_3 - y_4)^2 + (z_3 - z_4)^2} \\ l_4 &= \sqrt{(x_4 - x_1)^2 + (y_4 - y_1)^2 + (z_4 - z_1)^2}. \end{aligned} \tag{3.51}$$

Moreover, the area of an arbitrary quadrilateral is computed by splitting it into two triangles as shown in Figure 3.4. Hence, $\beta_1$ and $\beta_2$ are calculated as angles between two vectors

$$\begin{aligned} \beta_1 &= \arccos \frac{(\boldsymbol{r}_4 - \boldsymbol{r}_1) \circ (\boldsymbol{r}_2 - \boldsymbol{r}_1)}{l_4 \cdot l_1} \\ \beta_2 &= \arccos \frac{(\boldsymbol{r}_4 - \boldsymbol{r}_3) \circ (\boldsymbol{r}_2 - \boldsymbol{r}_3)}{l_3 \cdot l_2}. \end{aligned} \tag{3.52}$$

With one angle and two sides the area of the triangles results in

$$A = \frac{1}{2} \cdot l_1 \cdot l_4 \sin \beta_1 + \frac{1}{2} \cdot l_2 \cdot l_3 \sin \beta_2 \tag{3.53}$$

and thus the force for each node with a pressure $p$ yields

$$F = \frac{p \cdot A}{4}. \tag{3.54}$$

The force is put into the correct syntax and finally the .mbd file is created.

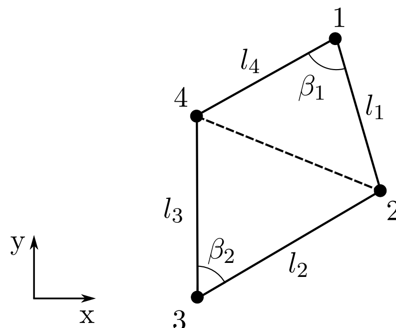



Figure 3.4.: The calculation of the area of an arbitrary quadrilateral

### 3.2.2. Postprocessing

MBDyn generates several output files and a short description for each file is provided in Table 3.1. The amount of output is defined by the output frequency which is set in the control data block. If the parameter equals 1 the files regarding structural entities (.mov, .ine, .frc, .jnt, .plt) contain output data for each time step and if it equals 10 for every tenth time step, etc. The most important files for postprocessing are the .mov and .frc file from which the position of the nodes and the forces with respect to the time step can be extracted respectively. For the exact output of the files consider [40].

The postprocessing software of choice is ParaView [2], a visualization tool for scientific data. With respect to the visualization toolkit (VTK) and the data representation a certain file format has to be met [34]. The format for scalar and vector data, e.g. nodes, cells and forces, is

```
# vtk DataFile Version x.x
Header
ASCII | BINARY
DATASET type

POINTS n datatype
x_1  y_1  z_1
...
x_n  y_n  z_n

CELLS n size
n_points_1 i  j  k  l  ...
...
n_points_n i  j  k  l  ...

CELL_TYPES n
type_1
...
type_n

POINT_DATA n
VECTORS name datatype
u_1  v_1  w_1
...
u_n  v_n  w_n
```

where the first line contains the identifier for .vtk files and the desired file version. The name of the dataset can be defined in the entry `Header`. Data characters can be encoded as ASCII or binary types. The `DATASET` parameter contains the type of dataset, e.g. structured or unstructured grids and points. The variable `n` in the defining lists represents the amount of entities the individual sections contain whereas the entry `datatype` determines the type of input data, e.g. integer or float. The size of the cells list is calculated according to size $= n \cdot 4 + n$. In this case, points are listed as coordinates $(x_i \;\; y_i \;\; z_i)^T$ and cells with a leading number of points, followed by the points' labels according to the points list. The type of the cell pertains to the structure of the used cells, e.g. triangular or quadrilateral

Table 3.1.: MBDyn output

| File extension | Data |
|---|---|
| .mov | Kinematics of nodes |
| .ine | Inertia of dynamic nodes |
| .frc | Forces |
| .out | General information about the simulation |
| .log | Supplementary information about the simulation |
| .jnt | Reaction forces |
| .pla | Information about shells/membranes |

cells which are defined by number 5 and 9 respectively. Vectors $(u_i \;\; v_i \;\; w_i)^T$ are included listing the vectors' components and furthermore can be labeled with a unique name. This type of .vtk file is created for each time step or rather for each output with respect to the output frequency. A reduced exemplary .vtk file for the first time step of Hencky's problem is included in Appendix B.3.

Again, in order to reduce the writing effort, a `Python` script has been developed and included in Appendix B.4. The initial position of the nodes and the membranes with their defining nodes are extracted from the .mbd file. Additionally, the position and forces of the nodes at each time or output step are extracted from the 2$^{\text{nd}}$, 3$^{\text{rd}}$ and 4$^{\text{th}}$ column of the .mov file and from the 3$^{\text{rd}}$, 4$^{\text{th}}$ and 5$^{\text{th}}$ column of the .frc file respectively. By means of a loop that iterates over each time step, the forces and the displacements, which result from the difference of the current and initial position, are obtained. In the same loop the syntax of the initial nodes (`POINTS` section from the definition above), the nodes assembling the quadrilateral membranes (`CELLS` and `CELL_Types`), the displacements and the forces (`VECTORS Displacement` and `VECTORS Forces`) are written to a .vtk file for each time or output step. Hence, the resulting files can be processed by ParaView [2, 34, 40].

### 3.2.3. Python Interface

MBDyn offers APIs for the programming languages `C`, `C++` and `Python`. This work is focused on the `Python` interface. In this part the basics of the communication with MBDyn are presented in a universal manner. However, a fully functional python script for the socket communication with MBDyn for Hencky's problem including comments is printed in Appendix B.5. The python wrapper is defined in MBDyn's library `mbpy`.

First, in Listing 3.1 the path for the `Python` wrapper is defined and the library for structural nodes `mbcNodal` is imported. Currently, only the libraries for dynamic nodes (`mbcNodal`) and modal nodes (`mbcModal`) are implemented. Afterwards, the script sets the path to the socket `$MBSOCK` which is created later in the input file by an external structural force as described in Section 3.1.3. Thereupon, the .mbd file is executed.

Listing 3.1: Set up of the environment

```python
import sys
import os
```

```
import tempfile

sys.path.append('/usr/local/mbdyn/libexec/mbpy')
from mbc_py_interface import mbcNodal

tmpdir = tempfile.mkdtemp('', '.mbdyn_')
path = tmpdir + '/mbdyn.sock'
os.environ['MBSOCK'] = path

os.system('input.mbd -o input > input.txt 2>&1 &')
```

The socket and pointers to the nodes are set up in Listing 3.2. The `Host` and `port` parameters are useful if an Internet socket is desired. If neither of them are defined, it defaults to the local host. The `timeout` entry states how long the socket will wait for connections and the value `-1` removes the limit. If `verbose` is true, a more detailed information about the decisions of the solver is provided. While true, the `data_and_next` parameter signals MBDyn to expect forces in the communication pattern. The `refnode` and `labels` entries are only important if a reference node was described in the .mbd file which instructs the peer to send the forces and their orientations relative to the reference node. The number of `nodes` must coincide with the number that was defined in the syntax of the external structural force Section 3.1.3. The `rot` and `accels` parameter describe the output of the orientation and acceleration of the nodes where the orientation can be written as a vector, a matrix or defined by euler angles and the acceleration can be toggled. However, the rotation parameter was identified to destabilize the script execution and therefore was often defined as zero. The listed parameters are all used in the initialization of the `mbcNodal` module. The method `negotiate()` sets pointers to the nodes in order to access or manipulate their properties.

Listing 3.2: Set up of the socket

```
host = ""
port = 0
timeout = -1          # -1 = forever (will not expire)
verbose = 0
data_and_next = 1     # true: MBDyn expects the packet to also contain forces
refnode = 0
nodes = n_nodes       # number of nodes
labels = 0
rot = 0x100           # orientation vector
#rot = 0x200;         # orientation matrix
#rot = 0x400;         # Euler 123
accels = 1            # output: accelerations

nodal = mbcNodal(path, host, port, timeout, verbose, data_and_next, refnode,
    nodes, labels, rot, accels)

nodal.negotiate()
```

The loop for the simulation is set up in Listing 3.3. With the call of the `recv()` method

MBDyn writes the kinematics to the external solver, which in this case is the `Python` script. From this point on, the script can access the properties of the nodes such as the position, velocity and acceleration by the use of the dot notation between the instance (`nodal`) and the attribute (e.g. `n_x`). It is important to notice that MBDyn stores its data, e.g. the position, in a one dimensional vector. Therefore, in order to access the position of the fifth node the entries 12, 13 and 14 of `nodal.n_x` have to be considered as illustrated in Listing 3.3. More accessible properties and their invocation are summarized in Table 3.2. Afterwards, the forces can be defined in the script, optionally dependent on any property sent by MBDyn. Upon calling the method `send()`, if the argument is true, the forces are sent to MBDyn and MBDyn iterates until convergence. However, if the argument is false, only one iteration is performed. The solver calculates the new kinematics and in the next iteration sends them back again. This whole iteration is carried out until the argument of the while loop is met. When the iteration is exited, the instance is destroyed and the temporary directory is removed.

Listing 3.3: Set up of the iteration

```
while (running = true):
    nodal.recv()

    x0, y0, z0 = nodal.n_x[0], nodal.n_x[1], nodal.n_x[2]
    x5, y5, z5 = nodal.n_x[12], nodal.n_x[13], nodal.n_x[14]

    nodal.n_f[0], nodal.n_f[1], nodal.n_f[2] = f_x0, f_y0, f_z0
    nodal.n_f[12], nodal.n_f[13], nodal.n_f[14] = f_x5, f_y5, f_z5

    nodal.send(converged)
nodal.destroy()
os.rmdir(tmpdir)
```

### 3.2.4. Implementation

A basic unified modeling language (UML) class diagram of the main classes of MBDyn is illustrated in Figure 3.5. The most fundamental ones, the `DataManager` and the `StepSolver`, are located in the middle of the diagram. The `DataManager` contains the

Table 3.2.: Accessible properties of the nodes

| Dot notation | Description |
|---|---|
| self.n_x | Position of the node |
| self.n_xp | Velocity |
| self.n_xpp | Acceleration |
| self.n_omega | Angular velocity |
| self.n_omegap | Angular acceleration |
| self.n_f | Force |
| self.n_m | Torque |

data of an arbitrary amount of nodes and elements and furthermore provides the member functions `AssRes()` to assemble the residual vector $\boldsymbol{r}$ and `AssJac()` to assemble the Jacobian matrix $\boldsymbol{J}_f$. Nodes and elements are children of `SimulationEntity` and inherit member functions that are important for every structural entity and that are called at every iteration (`Update()`) or every time step (`BeforePredict()`, `AfterPredict()`, `AfterConvergence()`). The other fundamental class, the `StepSolver`, is responsible for initiating the next time step when the previous one is finished by `Advance()` and therefore is able to access the residual and Jacobian matrix assembled by `DataManager` via the functions `Residual()` and `Jacobian()`. Additionally, the `StepSolver` class has a `NonlinearSolver` that solves the nonlinear problem by a provided interface, e.g. a Newton-Raphson solver. In turn, the `NonlinearSolver` class has a `LinearSolver` to avoid calculating the inverse of the Jacobian and solving the linear system of equations in Equation (3.14) instead. The desired linear and nonlinear solver can be selected in the problem block of the input file (see Section 3.2.1). The `Simulation` class owns all aforementioned classes and executes the simulation by a representative `Execute()` function.

A concept of the workflow of MBDyn is illustrated in Figure 3.6. The entries in the round rectangles are activities and the arrows are signifying the flow of these activities from the start to the end. The start is represented as a black circle and the end resembles a black circle in an empty circle. Diamonds are condition checks where the flow can take different routes depending on the checked variables. The conditions are included in square brackets. The simulation and model data is read from the problem block and the control data, and the nodes and elements block from the .mbd input file respectively. After the initial assembly, the derivatives phase and the first step integrated with a Crank-Nicolson scheme as described in Section 3.1.1, the second time step is started by the `Advance()` method. In the first condition check the time is examined. If the current time is greater or equal to the final time of the simulation, which is defined in the problem block of the input file, the end is reached and the simulation is terminated. Otherwise, a prediction of the kinematics is derived from time integration, followed by the entry of the first iteration of the time step. The residual of the system is assembled and checked for the residual tolerance which is also defined in the problem block of the input file. If the tolerance is met, i.e. the time step converged, the control flow returns to the `Advance()` method right away and a new time step is initiated. However, if the tolerance is not met, the next condition check is launched. Here, the iteration number is examined and while the maximum iteration number is not reached, the iteration continues. Otherwise, the simulation is terminated. In the next steps the Jacobi matrix is assembled if needed, the linear system of equations is solved (see Equation (3.14)) and the solution is updated. In the last condition check the current solution is compared to the solution of the previous time step and if the difference is smaller than the solution tolerance, the solution is declared “converged” and the next time step is initiated. Alternatively, the next iteration is invoked. The inner dotted rectangle defines the iteration loop and the outer dotted rectangle the time step loop.

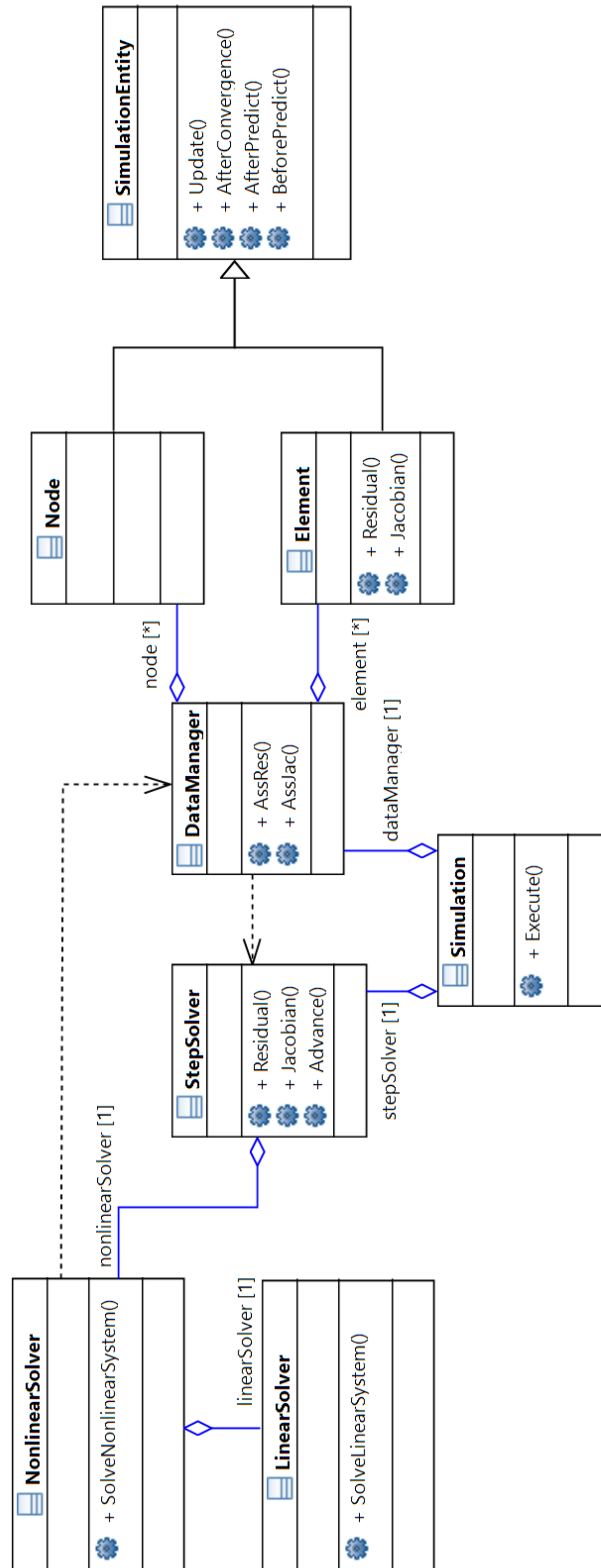



Figure 3.5.: UML class diagram for MBDyn (cf. [54])

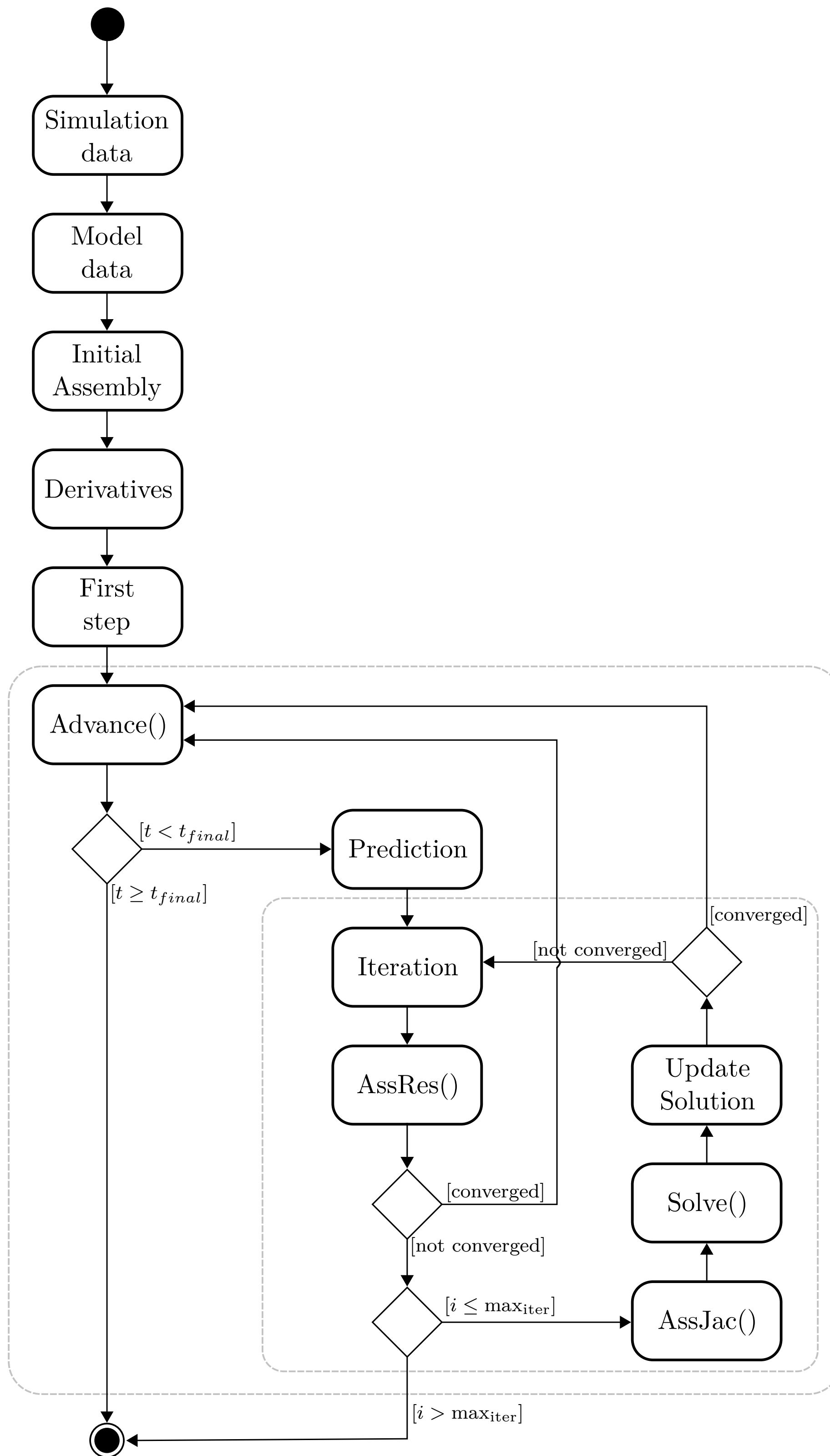



Figure 3.6.: UML activity diagram for MBDyn's solution process

## 3.3. Validation Tests

In order to examine the performance and the capabilities of MBDyn, several popular validation tests with shell and membrane elements are selected and executed. The employed benchmark problems are concisely described in the following subsections and the results are presented in Section 5.1. All meshes for the test cases are generated with Gmsh [26]. Furthermore, the test cases are simulated using at least two different mesh sizes.

### 3.3.1. Cantilever Beam

A cantilever beam clamped at one end as illustrated in the upper part of Figure 3.7 is subjected to a force $F$ [59]. The numerical data of the beam for this benchmark test is given in Table 3.3. In the lower part of Figure 3.7 the structure of one mesh is shown as eight four-node shell elements with the thickness $h$, whose vertices are defined by 18 dynamic nodes. The other mesh has a higher mesh density and consists of 16 shell elements. For this case the whole left side of the beam is fixed to a wall. Therefore, the nodes at $x = 0$ are grounded by means of clamp joints, constraining all six DOF. The force however is applied as an absolute type to the two nodes at $x = l$ and ramped up from zero to its maximum value within four seconds.

Table 3.3.: Numerical data for the cantilever beam

| Characteristic | Symbol | Value | Unit |
|---|---|---|---|
| Length | $l$ | 10 | m |
| Width | $b$ | 1 | m |
| Thickness | $h$ | 0.1 | m |
| Force | $F$ | 4 | N |
| Young's modulus | $E$ | $1.2 \cdot 10^6$ | $N/m^2$ |
| Poisson's ratio | $\nu$ | 0 | |

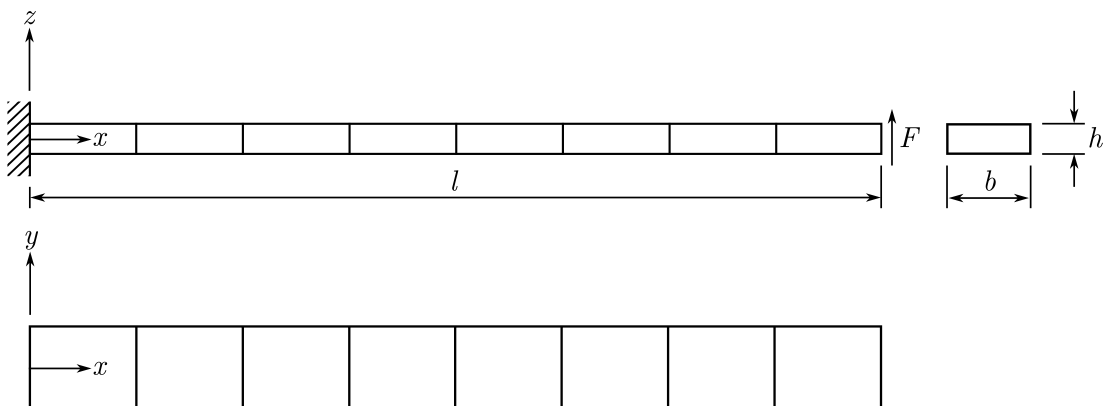



Figure 3.7.: Cantilever beam (top) as described by [59] and the employed mesh (bottom)

Table 3.4.: Numerical data for the cable

| Characteristic | Symbol | Value | Unit |
|---|---|---|---|
| Length | $l$ | 1 | m |
| Width | $b$ | 0.2 | m |
| Thickness | $h$ | 0.01 | m |
| Gravitational acceleration | $g$ | 9.81 | m/s$^2$ |
| Young's modulus | $E$ | $10 \cdot 10^9$ | N/m$^2$ |
| Poisson's ratio | $\nu$ | 0.4 | |
| Density | $\rho$ | 1000 | kg/m$^3$ |

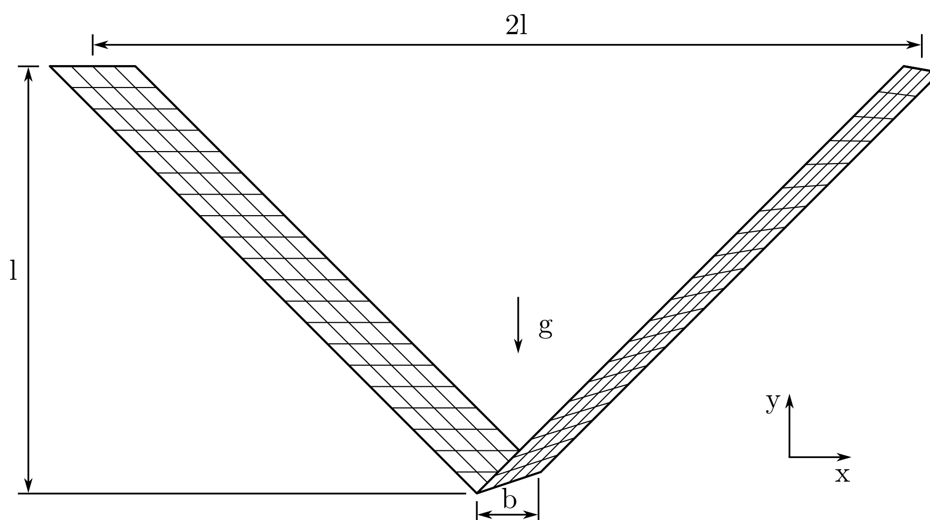



Figure 3.8.: Schematic drawing of the cable

### 3.3.2. Cable

The next benchmark test is a simulation of a cable, stretched across two supports at an equal height and the structure is solely affected by gravity [21]. The initial shape of the cable is an open-top triangle as displayed in Figure 3.8. In Table 3.4 the numerical data for the cable is presented. The structure is discretized with a mesh of 20 quadrilateral membrane elements in span-wise direction and four in thickness direction. Whenever a cable is exclusively loaded with its own weight, the shape of the structure at equilibrium turns into a catenary ([28], p. 169 ff.). The vertical displacement of the apex due to the dead load can be calculated by using the equations for the vertical position $y_a$ and cable length $s$

$$y_a = a \cdot \cosh\left(\frac{x_a}{a}\right) \tag{3.55}$$

$$s = 2a \cdot \sinh\left(\frac{x_a}{a}\right) \tag{3.56}$$

with the distance from the $x$-axis $a$ and the horizontal position $x_a$ of the apex. Considering Figure 3.8 and a coordinate system $x$, $y$ whose $y$-axis starts at the left support and whose $x$-axis is located at distance $a$ below the apex the equations

$$x_a = l \tag{3.57}$$

$$s = 2 \cdot \sqrt{2}\,\mathrm{m} \tag{3.58}$$

can be derived. Inserting Equations (3.57) and (3.58) into Equation (3.56) yields

$$0 = a \cdot \sinh\left(\frac{l}{a}\right) - \sqrt{2}\,\mathrm{m} \tag{3.59}$$

and can be solved via Newton's method with a starting point of $a_0 = 1$ and eight steps as performed in Appendix A.1 yielding

$$a = 0.6705\,\mathrm{m}. \tag{3.60}$$

This value for $a$ can finally be used for Equation (3.55) and results in

$$y_a = 1.565\,\mathrm{m} \tag{3.61}$$

where $y_a$ is the absolute vertical position of the apex. Subtracting the distance from the $x$-axis $a$ yields the height of the catenary

$$\delta = y_a - a = 1.565\,\mathrm{m} - 0.6705\,\mathrm{m} \approx 0.895\,\mathrm{m} \tag{3.62}$$

which is used to validate the results.

The gravitational force acting on each element $F_g$ is calculated by the use of a simple equation

$$F_g = mg = \rho V g \tag{3.63}$$

including density $\rho$ and volume $V$ of the particular element and the gravitational acceleration $g$. These forces are apportioned among the nodes, directed into negative $y$-direction and ramped up from zero to the maximum value within one second.

### 3.3.3. Circular Membrane Under Uniform Pressure

For this benchmark an initially round and flat membrane is inflated [21]. The configuration of this test, also known as Hencky's problem [19], is illustrated in Figure 3.9 and the corresponding data is summarized in Table 3.5. A schematic drawing of the circular membrane with a fixed edge and radius $r$ in the initial and final state is shown in Figure 3.9 (a) and (b) respectively. The uniform pressure is converted into nodal forces $F$ and applied to the nodes which are represented as black dots. Once again, the forces are ramped up from zero to their respective maximum value within one second. In contrast to the cantilever beam from Section 3.3.1 the force type is not absolute but follower. Consequently, the force changes its direction depending on the deformed structure. Unfortunately, as already stated in Section 3.1.3, MBDyn does not allow follower type forces to act on

Table 3.5.: Numerical data for Hencky's problem

| Characteristic | Symbol | Value | Unit |
|---|---|---|---|
| Radius | $r$ | 0.1425 | m |
| Thickness | $h$ | 0.01 | m |
| Uniform pressure | $p$ | $100 \cdot 10^3$ | Pa |
| Membrane modulus | $E_t$ | 311 488 | N/m |
| Poisson's ratio | $\nu$ | 0.34 | |
| Density | $\rho$ | 1000 | kg/m$^3$ |

Table 3.6.: Numerical data for the square airbag

| Characteristic | Symbol | Value | Unit |
|---|---|---|---|
| Length | $l$ | 1.2 | m |
| Thickness | $h$ | $6 \cdot 10^{-4}$ | m |
| Uniform pressure | $P$ | 5000 | Pa |
| Young's modulus | $E$ | $588 \cdot 10^6$ | N/m$^2$ |
| Poisson's ratio | $\nu$ | 0.4 | |
| Density | $\rho$ | 1000 | kg/m$^3$ |

membrane elements. Therefore, the forces and their vectors are calculated using the procedure introduced in Section 3.1.3. The employed mesh is depicted in Figure 3.9 (c) and consists of 261 structural nodes and 236 quadrilateral membrane elements.

### 3.3.4. Square Airbag

In order to simulate an airbag inflation, two square membranes are sewn together at their perimeters and a uniform internal pressure is applied. This benchmark is well tested [4, 21, 14, 35, 66, 30, 68]. Although it is mainly used to validate a wrinkling model, this test can be performed to examine the representation of various non-linear aspects such as large displacements and displacement-dependent forces. The material's characteristics and the acting pressure are compiled in Table 3.6. A schematic drawing of the configuration and one employed simple mesh with $10 \times 10$ membrane elements are shown in Figure 3.10. Due to symmetry in all three axes, with the correct symmetry boundary conditions for the line segments $\overline{\mathrm{AB}}$ and $\overline{\mathrm{AD}}$, only one eight of the airbag has to be modeled. Furthermore, line segments $\overline{\mathrm{BC}}$ and $\overline{\mathrm{CD}}$ are fixed in $z$-direction but free to move in $x$ and $y$-direction [35]. Once again the forces on the nodes are calculated from the applied uniform pressure, converted into follower forces and ramped up from zero to their maximum in two seconds.

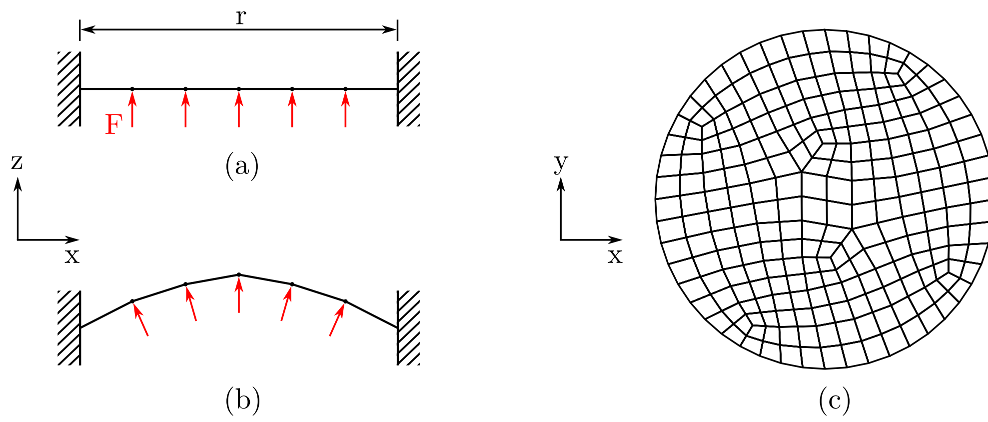



Figure 3.9.: Schematic drawing of Hencky's problem and the employed mesh

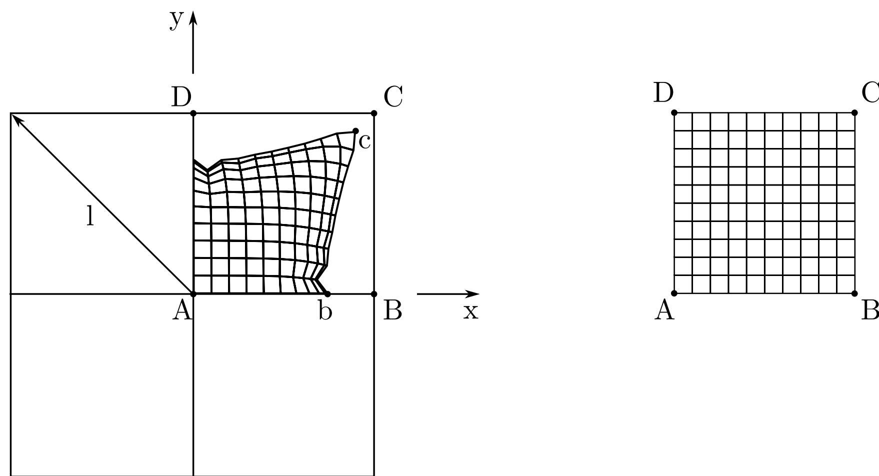



Figure 3.10.: Schematic drawing of the square airbag as described by [66] and the employed mesh

# 4. FSI

As already mentioned in Section 2.4 a FSI framework is created by connecting two solvers, in this case the fluid solver OpenFOAM and the structural solver MBDyn, with the coupling library preCICE. This relation is illustrated in Figure 4.1. An exemplary configuration file for preCICE is presented and described in Section 4.1. The communication through preCICE is achieved by adapters. Fortunately, a ready-to-use adapter for OpenFOAM (colored blue in Figure 4.1) already exists. Thus, only the MBDyn adapter has to be realized. This development is described in Section 4.2 followed by a concise review of the OpenFOAM adapter in Section 4.3. A modified version of a popular benchmark for CFD simulations, the lid-driven cavity, and a FSI simulation of a ram-air kite section are documented in Section 4.4.

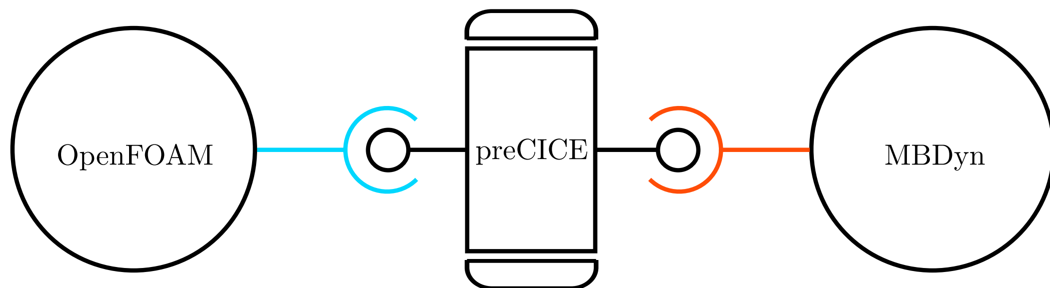



Figure 4.1.: Schematic drawing of the coupling mechanism for FSI simulations with the coupling library preCICE for MBDyn and OpenFOAM including adapters (colored)

## 4.1. preCICE

The coupling library preCICE is configured by a .xml file in which the solvers, their meshes and the desired mapping scheme, the exchanging data and the general coupling scheme are defined. In order to explain the structure of the preCICE configuration file, an example of a 2D coupling between the displacements and forces of an arbitrary structural and fluid solver with a RBF mesh mapping and a serial-implicit coupling scheme is given in this section.

The beginning of the .xml file in Listing 4.1 marks the current xml version and the statement that a `precice-configuration` is defined in the rest of the file. The `dimensions`

parameter determines whether a 2D or 3D problem is examined. Next, the data sets that are planned to be exchanged are defined, in this case stresses and displacements. The `name` tag, which is used throughout the file, is an identifier that is used in the code of the adapters. In order to generate the meshes, the vertices and elements at the interface are set carrying specified data. The coordinates of these vertices are determined by the solvers. In this example, the location of the nodes and the centers of the cells for the fluid and structural mesh that incorporate the displacements and stresses respectively are specified.

Listing 4.1: preCICE data and mesh

```
<?xml version="1.0"?>

<precice-configuration>

    <solver-interface dimensions="2">

        <data:vector name="Stresses" />
        <data:vector name="Displacements" />

        <mesh name="Fluid_Nodes">
            <use-data name="Displacements" />
        </mesh>

        <mesh name="Fluid_CellCenters">
            <use-data name="Stresses" />
        </mesh>

        <mesh name="Structure_Nodes">
            <use-data name="Displacements" />
        </mesh>
        <mesh name="Structure_CellCenters">
            <use-data name="Stresses" />
        </mesh>
```

In Listing 4.2 the two solvers that make use of the coupling environment preCICE are introduced as separate participants. For each participant the necessary meshes including the coordinates of the vertices and elements are supplied. If the solver possesses an own mesh, the `provide` statement is applied. If a mesh is created by another solver and has to be supplied, the `from` statement is triggered followed by the name tag of the providing solver. Furthermore, `write-data` and `read-data` are methods for the participants to write and read data belonging to a certain mesh. The data mapping in this case is executed by an RBF scheme (see Section 2.4) using thin-plate splines. Since the example implements a serial coupling scheme, the mesh mapping only occurs for one participant. The `direction` statement indicates, whether the mapping is executed before data is read from the mesh, or after data is written.

Listing 4.2: preCICE participants

```
        <participant name="Fluid_Solver">
            <use-mesh name="Fluid_Nodes" provide="yes" />
```

```
    <use-mesh name="Fluid_CellCenters" provide="yes" />
    <use-mesh name="Structure_Nodes" from="Structure_Solver" />
    <use-mesh name="Structure_CellCenters" from="Structure_Solver" />
    <write-data mesh="Fluid_CellCenters" name="Stresses" />
    <read-data mesh="Fluid_Nodes" name="Displacements" />
    <mapping:petrbf-thin-plate-splines direction="read"
        from="Structure_Nodes" to="Fluid_Nodes" constraint="consistent"/>
    <mapping:petrbf-thin-plate-splines direction="write"
        from="Fluid_CellCenters" to="Structure_CellCenters"
        constraint="consistent"/>
</participant>

<participant name="Structure_Solver">
    <use-mesh name="Structure_Nodes" provide="yes"/>
    <use-mesh name="Structure_CellCenters" provide="yes"/>
    <use-mesh name="Fluid_CellCenters" from="Fluid_Solver" />
    <write-data mesh="Structure_Nodes" name="Displacements" />
    <read-data mesh="Structure_CellCenters" name="Stresses" />
</participant>
```

The communication between the participants in Listing 4.3 is provided by a socket including the tag `m2n:sockets` with a direction and, when a serial scheme is used, the shown `distribution-type`. The exemplary serial-implicit coupling scheme (see Section 2.4) is defined within the `coupling-scheme` environment. General and self-explanatory properties, such as the length of the time step, the maximum allowed number of time steps, the execution order between the participants, the maximum allow number of iterations and the relative convergence measure are defined here. Moreover, the exchange of the corresponding data in the corresponding mesh is determined with respect to the coupling scheme. A quasi-Newton postprocessing method (see Section 2.4) based on the Least Squares method is applied in order to stabilize the iteration. For further information including the syntax for other coupling schemes the reader is referred to the preCICE Wiki [52]. This file is configured at runtime [52].

Listing 4.3: preCICE coupling scheme

```
<m2n:sockets from="Fluid_Solver" to="Structure_Solver"
    distribution-type="gather-scatter" />

<coupling-scheme:serial-implicit>
    <timestep-length value="0.005" />
    <max-timesteps value="20000" />
    <participants first="Fluid_Solver" second="Structure_Solver" />
    <exchange data="Stresses" from="Fluid_Solver"
        mesh="Structure_CellCenters" to="Structure_Solver" />
    <exchange data="Displacements" from="Structure_Solver"
        mesh="Structure_Nodes" to="Fluid_Solver" />
    <relative-convergence-measure limit="1.0e-4" data="Displacements"
        mesh="Structure_Nodes"/>
    <max-iterations value="200" />
    <extrapolation-order value="2" />
```

```
        <post-processing:IQN-ILS>
            <data mesh="Structure_Nodes" name="Displacements" />
            <initial-relaxation value="0.001" />
            <max-used-iterations value="10" />
            <timesteps-reused value="2" />
            <filter type="QR1" limit="1e-8" />
        </post-processing:IQN-ILS>
    </coupling-scheme:serial-implicit>

</solver-interface>

</precice-configuration>
```

## 4.2. MBDyn - preCICE Adapter

The adapter for the communication between preCICE and MBDyn is written in `Python`. The following description is a basic structure of the adapter. However, for an easier understanding, in this case, MBDyn receives stresses from an external fluid solver and sends back displacements based on the stresses using an implicit coupling scheme. With respect to the intentional use of this adapter, this will probably be the configuration most of the time. From the programming point of view this represents a mixture between a direct modification and the use of an adapter class.

The beginning of the adapter script is shown in Listing 4.4. Here, necessary libraries are imported with `PySolver Interface` being the `Python` API library for preCICE and `mpi4py` the message passing interface (MPI). Furthermore, the `Adapter` library includes methods that are used in the later section of the code. Initially, various name tags are stated coinciding with the definition in the preCICE configuration file. The `PySolverInterface()` with the name of the structural solver in the .xml file from the previous section is instantiated as `interface`. Finally, the interface is configured using the configuration file and the global dimension of the system is obtained by the use of a getter-method.

Listing 4.4: Set up of preCICE

```
import numpy as np
from numpy import *
import Adapter

# define name tags
MBDynFile = 'input.mbd'          # MBDyn file name prefix
configFile = "constant/preCICE.xml" # preCICE configuration file name
solver = "Structure_Solver"      # solver name tag in preCICE.xml
mesh1 = "Structure_Nodes"        # mesh name tag in preCICE.xml
mesh2 = "Structure_CellCenters"  # mesh name tag in preCICE.xml
dataStructure = "Displacements"  # data name tag in preCICE.xml
dataFluid = "Stresses"           # data name tag in preCICE.xml
```

```
# set up preCICE
print("Configure preCICE...")
interface = PySolverInterface(solver, 0, 1)
interface.configure(configFile)
print("preCICE configured...")
dim = interface.getDimensions()
```

The next code snippet in Listing 4.5 consists of reading nodes and cells from the MBDyn input file and the set up of MBDyn. The method `readMesh()` reads the input file and returns the nodes, cells and cell centers of the structural mesh. For a detailed description of the MBDyn set-up consider Section 3.2.3.

Listing 4.5: Set up of MBDyn

```
# read nodes and cells
nodes_list, cells_list, cell_centers = Adapter.readMesh(MBDynFile)

# set up MBDyn
Adapter.execute(MBDynFile)
from mbc_py_interface import mbcNodal
path, host, port, timeout, verbose, data_and_next, refnode, nodes, label, rot,
    accels = Adapter.init()
nodal = mbcNodal(path, host, port, timeout, verbose, data_and_next, refnode,
    nodes, labels, rot, accels)
nodal.negotiate()
```

The data exchange between the participants with regard to preCICE is initialized in Listing 4.6. First the mesh and data access is determined. The method `getMeshID()` takes the name of the mesh and returns the identification (ID) of the mesh pertaining to MBDyn. Afterwards, the name of the data and the mesh ID are passed to `getDataID()` and the method returns the data ID. In this case two grids are defined, namely one for the coordinates of the nodes and one for the coordinates of the cell centers. This is important due to the fact that the fluid solver calculates the stresses and writes them to the cell centers and MBDyn calculates the displacements and writes them to the nodes. The coordinates are set by the method `setMeshVertices()` which takes the ID of the mesh, the size, the positions in a 1D array and the vertex IDs. Since the iteration loop has not been entered yet, the initial displacements and stresses are assumed to be zero in this case. The communication and data structure of preCICE is set up by the function `initialize()`. Moreover, this method returns the maximal allowed time step for MBDyn with respect to the global time step for both participants, stored in `precice_dt`. The boolean methods `isActionRequired()`, `fulfilledAction()` and `isReadDataAvailable()` are triggers in order to execute actions or to let preCICE know that the action was fulfilled. The first `if` block instructs MBDyn to write the initial data, or initial displacements in this case, to the preCICE data structure. If a restart of the simulation from a previous state is needed, the `initializeData()` is called and the data list can be set accordingly. The second `if` block signifies that if block data is available, the interface reads the stresses using the `readBlockVectorData()` function.

Listing 4.6: Set up of the data exchange

```
# set up data exchange through preCICE
Nn = len(nodes_list)
nmeshID = interface.getMeshID(mesh1)
displacementsID = interface.getDataID(dataStructure, nmeshID)
nvertexIDs = Nn[0.0]
interface.setMeshVertices(nmeshID, len(nodes_list), np.ravel(nodes_list),
    nvertexIDs)
displacements = np.array(dim*Nn*[0.0])

Nc = len(cell_centers)
cmeshID = interface.getMeshID(mesh2)
stressesID = interface.getDataID(dataFluid, cmeshID)
cvertexIDs = Nc*[0.0]
interface.setMeshVertices(cmeshID, Nc, np.ravel(ccs), cvertexIDs)
stresses = np.array(dim*Nc*[0.0])

# coupling intialization
precice_dt = interface.initialize()

if (interface.isActionRequired(PyActionWriteInitialData())):
    interface.writeBlockVectorData(displacementsID, Nn, nvertexIDs,
        displacements)
    interface.fulfilledAction(PyActionWriteInitialData())

interface.initializeData()

if (interface.isReadDataAvailable()):
    interface.readBlockVectorData(stressesID, Nc, cvertexIDs, stresses)
```

Listing 4.7 consists of data definitions for the loop and the beginning of the loop itself. Parameter such as the time step for MBDyn `MBDyn_dt`, the `iteration` and `substep` or the displacement of the previous time step `prevd` are initialized. Furthermore, the boolean `converged` implies, whether the simulation converged in the current substep and is initially set to `False`. The iteration is set up by a while loop with the boolean argument `isCouplingOngoing()` through which preCICE can guide the beginning and termination of the simulation for both solvers. As soon as the loop is entered, an iteration checkpoint is written to the preCICE data structure. In the first iteration of the loop the stresses that have been initialized in Listing 4.6 are reshaped. For any successive iteration the stresses are then read from the fluid solver through preCICE in the last code line of Listing 4.7. Due to the fact that MBDyn only accepts forces, the stresses have to be converted. This is accomplished by the `forces()` method which takes the stresses, the nodes and cells and the current position of the nodes and returns equivalent nodal forces. The current position of the nodes can be accessed via MBDyn by `nodal.n_x` as described in Section 3.2.3. Similarly, the resulting forces are assigned with `nodal.n_f`. With the call of the `recv()` method from `nodal` the script receives the kinematics from MBDyn. With these updated kinematics, the displacements are calculated by a simple function `displacements()` which takes the initial and the current position of

the nodes, subtracts them and returns the displacement list `d` for each node. Furthermore, in order to measure the convergence of each substep, the previous displacement `prevd` is subtracted from the resulting displacement. This data is subsequently written to the preCICE data structure once again in a 1D array. The next time step is initiated by the method `advance()` which takes MBDyn's time step as an argument and returns the next possible maximum time step with respect to the value defined in the configuration file. This way, the structural solver can have a lower time step than the coupled fluid solver, which is also known as "subcycling". It is important to mention that data such as stresses or displacements is only exchanged at each time step and not at each substep. This fact is also illustrated in the coupling scheme in Figure 2.6.

Listing 4.7: Coupling loop 1

```
# data for the loop
MBDyn_dt = 0.001
time = precice_dt
iteration = 0
substep = 0
prevd = 0
converged = False

# coupling loop
while (interface.isCouplingOngoing()):

    if (interface.isActionRequired(PyActionWriteIterationCheckpoint())):
        interface.fulfilledAction(PyActionWriteIterationCheckpoint()

    s = np.reshape(stresses,(-1,2))

    # calculate and assign forces
    current_nodes = nodal.n_x
    forces = Adapter.forces(s, nodes_list, cells_list, current_nodes)
    nodal.n_f = forces

    nodal.recv()

    # calculate displacements
    current_nodes = nodal.n_x
    d = Adapter.displacements(nodes_list, current_nodes)
    displacements = d - prevd

    interface.writeBlockVectorData(displacementsID, Nn, nvertexIDs,
        np.ravel(displacements))

    precice_dt = interface.advance(MBDyn_dt)

    interface.readBlockVectorData(stressesID, Nc, cvertexIDs, stresses)
```

The last code snippet in Listing 4.8 addresses the convergence status of the substep. If the global time step of preCICE `time` matches the maximum available time step `precice_dt`

the convergence can be measured. Otherwise, MBDyn and the fluid solver still have to be synchronized. The `send()` method sends the forces to MBDyn only when the passed argument `converged` is true. This is only the case if the time step matches and if no action is required. Else, no forces are sent and the next substep is initiated. As soon as preCICE breaks the loop, the function `finalize()` terminates the communication [52].

Listing 4.8: Coupling loop 2

```
    if precice_dt == time:
        if (interface.isActionRequired(PyActionReadIterationCheckpoint())):
            interface.fulfilledAction(PyActionReadIterationCheckpoint())
            converged = False
            nodal.send(converged)
            substep += 1
            print 'Not converged'
        else:
            iteration += 1
            time = iteration*precice_dt
            prevd = d
            substep = 0
            print 'Converged! Time: {} iteration: {}'.format(time, substep)
            converged = True
            nodal.send(converged)
    else:
        converged = False
        nodal.send(converged)

# terminate
interface.finalize()
```

## 4.3. OpenFOAM - preCICE Adapter

The adapter between OpenFOAM and preCICE has already been developed by Blom [6] and is implemented in the FOAM-FSI library which is an extension to foam-extend (see also Section 2.3). An additional `FSI.YAML` file is added to the `constant` OpenFOAM directory of Section 2.3. This file includes information about the time-integration scheme and the type of the participating fluid and structural solvers. Furthermore, the employed coupling scheme and data mapping method are defined. This file is read by the adapter which subsequently couples FOAM-FSI to preCICE and thus to the structural solver. More information about the adapter can be found at the GitHub repository for FOAM-FSI [6]

## 4.4. Validation Tests

A FSI benchmark is performed in Section 4.4.1 in order to examine the general accuracy and stability of the developed adapter. Eventually, in Section 4.4.2, a FSI simulation of a ram air kite section is introduced. The results of these simulations are summarized in

Section 5.2.

### 4.4.1. 2D Lid-Driven Cavity with Flexible Bottom Membrane

This validation test is a modified version of the well-known CFD lid-driven cavity benchmark. The original benchmark is illustrated in Figure 4.2 (a) and consists of a solid cavity that is subject to a moving lid with a constant velocity parallel to the bottom wall of the cavity. Due to the movement of the lid, a circulating flow is created inside the cavity. The ambient pressure $p_0$ is assumed to be zero. All remaining boundaries are defined as “no-slip wall”, i.e. the velocity directly at the walls is zero. The modified variant for FSI problems in Figure 4.2 (b) was introduced by Wall [63] and further improved by Garcia [62]. It shares the same boundary conditions as the original benchmark. However, the constant velocity of the lid is replaced by a periodic velocity defined as

$$\overline{u} = \left(1 - \cos\left(\frac{2\pi t}{5}\right)\right) \text{m/s} \tag{4.1}$$

resulting in a harmonic oscillation between $\overline{u}_{\text{min}} = 0$ m/s and $\overline{u}_{\text{min}} = 2$ m/s. Furthermore, an inlet and an outlet are implemented in order to allow volume changes with respect to time. Lastly, a flexible membrane replaces the bottom wall. The simulation data for the fluid and the structural domain is summarized in Table 4.1 [46, 62, 63, 69].

#### Fluid domain

The fluid domain, which is everything but the membrane, is computed by the FOAM-FSI extension of foam-extend. Since this is a fully transient and laminar case the PIMPLE algorithm is used as a solver with a relaxation factor of 0.8 for the fluid velocity $u$ and pressure $p$. The fluid itself is considered “Newtonian” with a kinematic viscosity $\nu_t = 0.01$ N/m$^2$ and density $\rho_{\text{f}} = 1$ kg/m$^3$. The fluid domain is spatially discretized with $40 \times 40$ hexahedron in $x$ and $y$ and one in $z$ direction. A backward differentiation formula (BDF) is applied for the time integration and the time step is fixed to $\Delta t_{\text{f}} = 0.005$ s.

Table 4.1.: Numerical data for the lid-driven cavity

| Characteristic | Symbol | Value | Unit |
|---|---|---|---|
| Young’s modulus | $E$ | 250 | N/m$^2$ |
| Kinematic viscosity | $\nu_t$ | 0.01 | m$^2/s$ |
| Fluid density | $\rho_{\text{f}}$ | 1 | kg/m$^3$ |
| Fluid time step | $\Delta t_f$ | 0.005 | s |
| Poisson’s ratio | $\nu$ | 0.0 | |
| Structural density | $\rho_{\text{s}}$ | 500 | kg/m$^3$ |
| Structural time step | $\Delta t_s$ | 0.001 | s |
| Membrane thickness | $t$ | 0.002 | m |

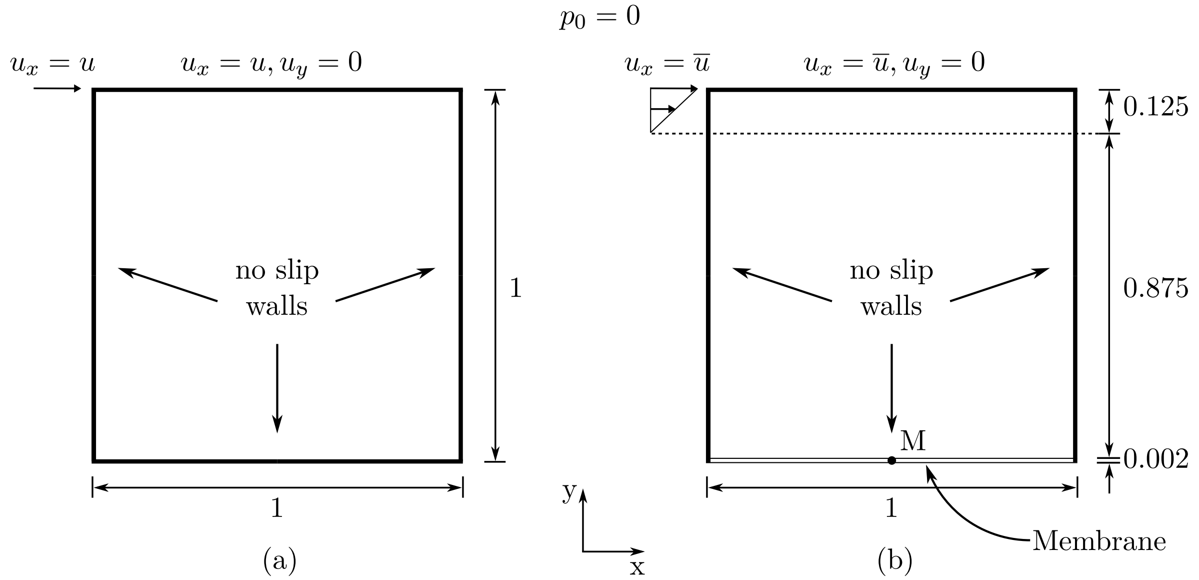



Figure 4.2.: Schematic drawing of the original lid-driven cavity benchmark (a) and a modified version (b) in order to include fluid-structure interaction

#### Structural domain

The structural domain consists solely of the membrane at the bottom of the cavity. 81 thin MBDyn membrane elements in $x$ direction with a thickness $t = 0.002$ m are used as a mesh. For the postprocessing of the result the middle point M of the membrane is defined. The structural properties are a low Young's modulus of $E = 250$ N/m$^2$, a Poisson's ratio of $\nu = 0$ and a density of $\rho_\mathrm{s} = 500$ kg/m$^3$. The time integration is accomplished by a linear multistep scheme (see Section 3.1.1) and the time step is $\Delta t_\mathrm{s} = 0.001$ s. Note, that the time step of the structural solver is considerably lower than than that of the fluid solver.

#### Coupling

In order to couple both solvers, preCICE is configured with a serial-implicit coupling scheme and a quasi-Newton postprocessing algorithm (IQN-ILS) based on the least squares method to approximate the Jacobi matrix. The data mapping between the two meshes is provided by the RBF method using thin-plate splines.

Due to the harmonic excitation and the expected oscillating motion of the membrane, this case is a suitable validation for the stability and the performance of the coupling mechanism including the deployed MBDyn-preCICE adapter. Furthermore, various reference solutions exist to validate against.

### 4.4.2. Ram-Air Kite Section

a section of a ram-air kite is considered, in order to examine the future field of application for this adapter. A ram-air kite is a soft kite which consists of cells and ribs and whose shape is inflated by an air inlet at the leading edge. The inlet is ideally located exactly at the stagnation point. Due to the high stagnation pressure of airfoils compared to the dynamic and ambient pressure in flight the kite is inflated. Furthermore, while airborne, the pressure inside the cells stays intact and provides internal stability. This investigation is based on a steady-state simulation by Folkersma et al. [23] of a ram-air kite cell, performed with FOAM-FSI and the TU Delft in-house solver mem4py. The coupling is achieved once again via preCICE. The deployed shape of the section is a MH 92 airfoil with a maximum thickness of 15.5% at 27.4% chord and a maximum camber of 1.4% at 15.1% chord as shown in Figure 4.3 [3]. This airfoil is often used in para-gliding and for general ram-air kites. The surface of the structure can be modeled as a membrane with the membrane modulus $E_t$ and Poisson's number $\nu$. At first, the section is inflated due to the difference of the stagnation pressure $p_\mathrm{t}$, resulting from an inlet at the leading edge of the kite and the flow pressure $p_\infty$. Furthermore, it is set at an AoA $\alpha$ to the air flow with a flow velocity $u_\infty$ and a flow density $\rho_\infty$ as illustrated in Figure 4.4 and extended in $z$-direction. Since

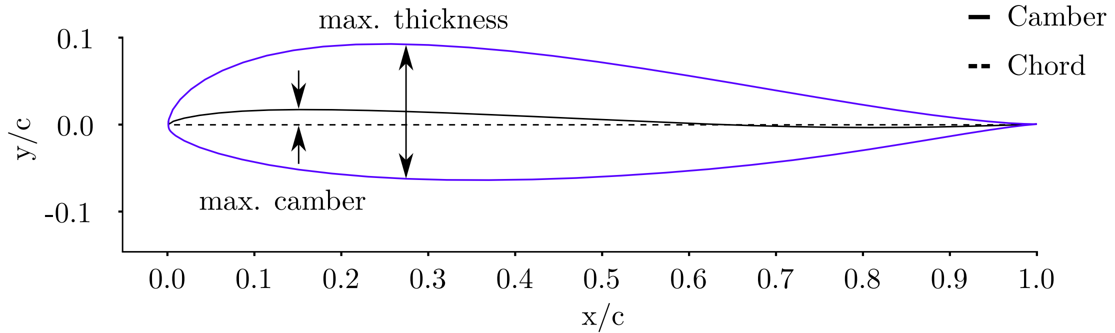



Figure 4.3.: Airfoil of the ram-air kite section (cf. [3])

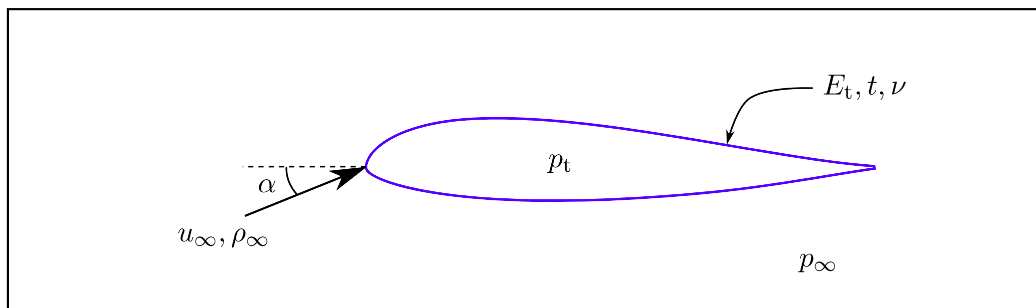



Figure 4.4.: Schematic drawing of the fluid and structural domain of the ram-air kite section (cf. [23])

the sharp trailing edge of the kite is bound to produce singularities and instabilities in the simulation, the structure is trimmed.

#### Fluid domain

For this case the PIMPLE algorithm is used again as a solver. Since the fluid flow is considered turbulent at the boundary layer at the airfoil the shear stress transport (SST) $k-\omega$ turbulence model is implemented with the the turbulent energy $k = 0.0006$ m$^2$/s$^2$ and the turbulence scale $\omega = 300$ 1/s. The fluid domain is discretized as a circular unstructured mesh and consists of more than 500 000 elements. In order to ensure an evolved flow the $x$ and $y$ dimensions of the grid are exceeding those of the structural mesh to a great extend. The flow strikes the airfoil with an AoA of $\alpha = 10°$ and a velocity of $u_\infty = 1$ m/s. This system is integrated in time by a BDF scheme. The Newtonian fluid has a density of $\rho_\mathrm{f} = 2000$ kg/m$^3$ and a kinematic viscosity $\nu_t = 2 \times 10^{-7}$.

#### Structural domain

The structural domain is discretized which results in a mesh with 1309 quadrilateral membrane elements and is illustrated in Figure 4.5. Due to a symmetry condition at the mid-plane of the kite section, only half of the structure is modeled. One side of the kite is clamped while the opposite side is imposed by a sliding boundary and symmetry condition. The material characteristics are a membrane modulus $E_t = 10000$ N/m and the Poisson's number $\nu = 0.3$. Prior to the FSI simulation, the shape of the membrane has to be determined due to inflation pressure resulting from the pressure inlet of the ram-air kite. This is also known as the form finding process. Therefore, the kite section is inflated by an internal stagnation pressure of 1000 Pa. The weight of the membrane is determined by the areal density $\rho_\mathrm{sA} = 80$ g/m$^2$. The time integration is accomplished by a linear multistep scheme. Consider Table 4.2 for a summarization of the numerical data.

#### Coupling

Similar to the lid-driven cavity, the coupling environment preCICE is configured with a serial-implicit coupling scheme and a quasi-Newton postprocessing algorithm IQN-ILS based

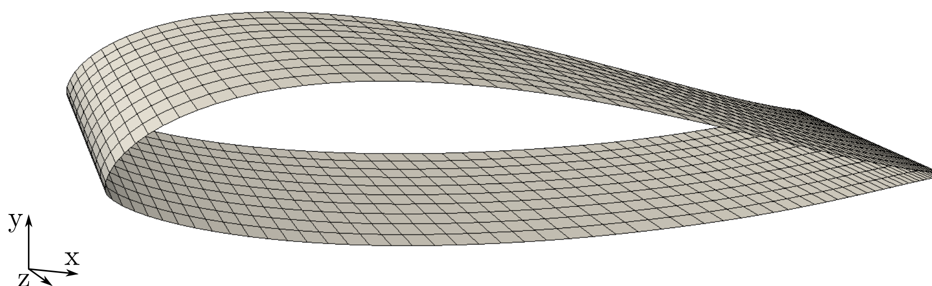



Figure 4.5.: Structural discretization of the ram-air kite section

Table 4.2.: Numerical data for the ram-air kite

| Characteristic | Symbol | Value | Unit |
|---|---|---|---|
| Membrane modulus | $E_t$ | 10000 | N/m |
| Kinematic viscosity | $\nu_t$ | 0.01 | $m^2/s$ |
| Poisson's ratio | $\nu$ | 0.3 | |
| Fluid density | $\rho_{\mathrm{f}}$ | 2000 | $kg/m^3$ |
| Areal density | $\rho_{\mathrm{sA}}$ | 500 | $kg/m^3$ |
| Flow velocity | $u_\infty$ | 1 | m/s |
| Turbulent energy | $k$ | 0.0006 | $m^2/s^2$ |
| Turbulent scale | $\omega$ | 300 | 1/s |
| Angle of Attack | $\alpha$ | 10 | ° |
| Stagnation pressure | $p_t$ | 1000 | Pa |

on the least squares method. The data mapping between the two meshes is accomplished by the RBF method using thin-plate splines.

# 5. Results

The results of the validation tests introduced in Section 3.3 and Section 4.4 are presented in this chapter. The structural simulations pertaining to MBDyn and those including fluid-structure interaction are gathered in Section 5.1 and Section 5.2 respectively.

## 5.1. MBDyn Validation Tests

In this section all validation tests for the MBDyn membrane and shell elements are analyzed. Whenever there exists an analytical solution to the benchmark problem, the results are compared to it. Otherwise numerical data ideally from multiple sources is used as a reference.

### 5.1.1. Cantilever Beam

In Figure 5.1 the deformation process of the cantilever beam from 0 to 4 seconds is visualized. The tip of the beam moves up in positive $z$-direction and to the left in negative

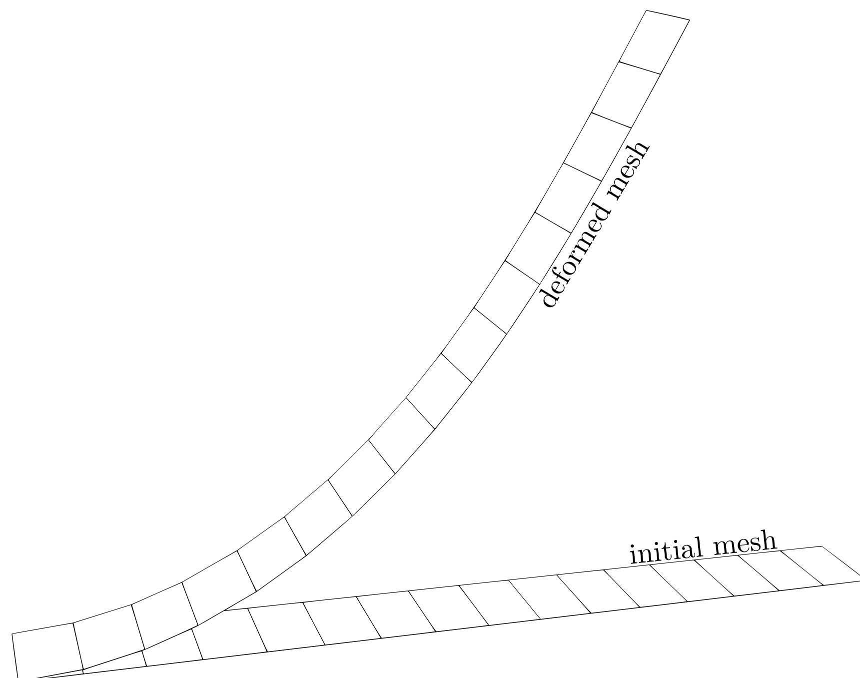



Figure 5.1.: The initial (bottom) and deformed (top) mesh of the cantilever beam (cf. [59])

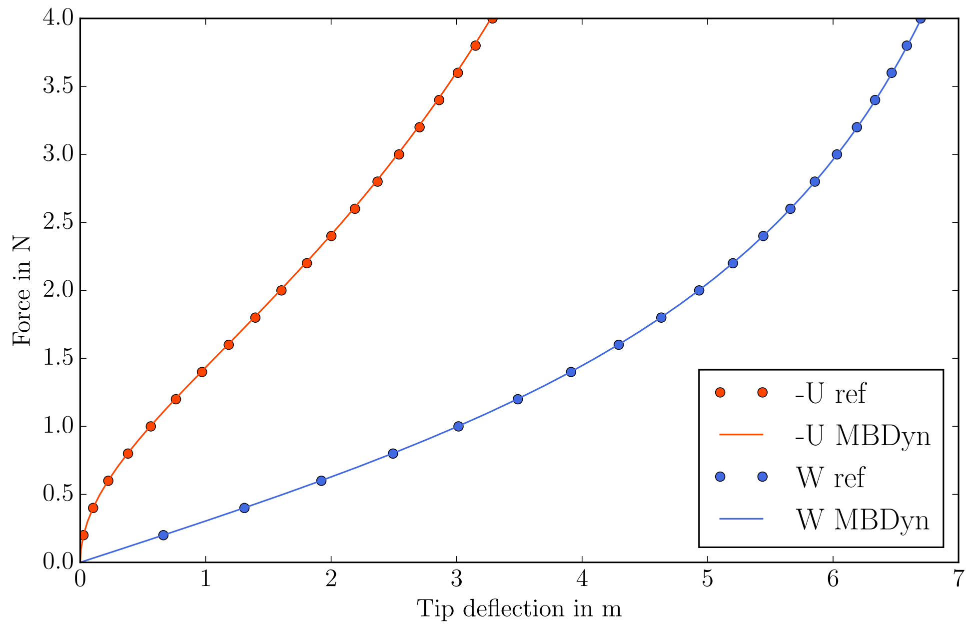



Figure 5.2.: Cantilever tip deflections with eight shell elements

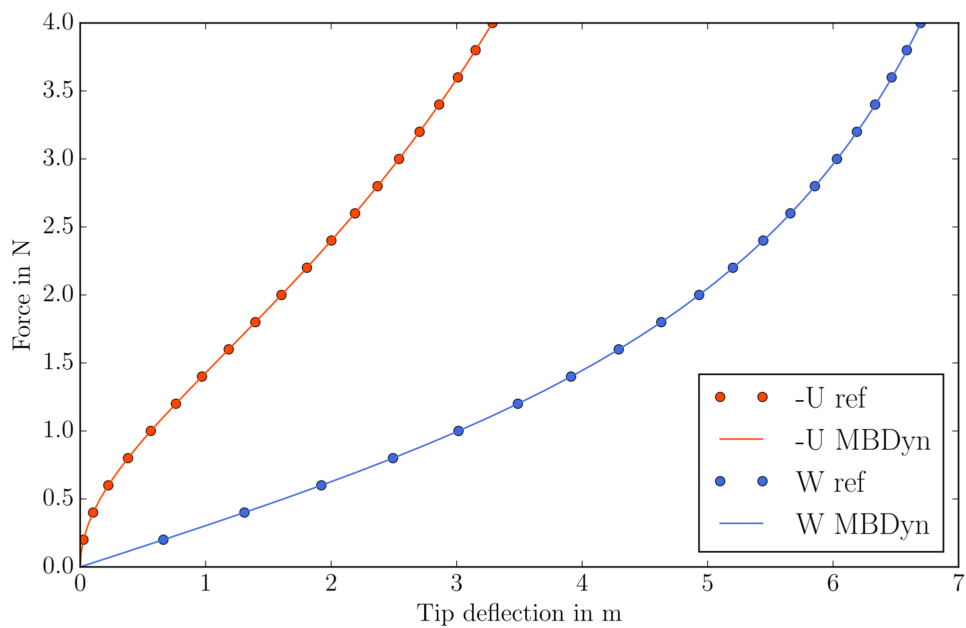



Figure 5.3.: Cantilever tip deflections with 16 shell elements

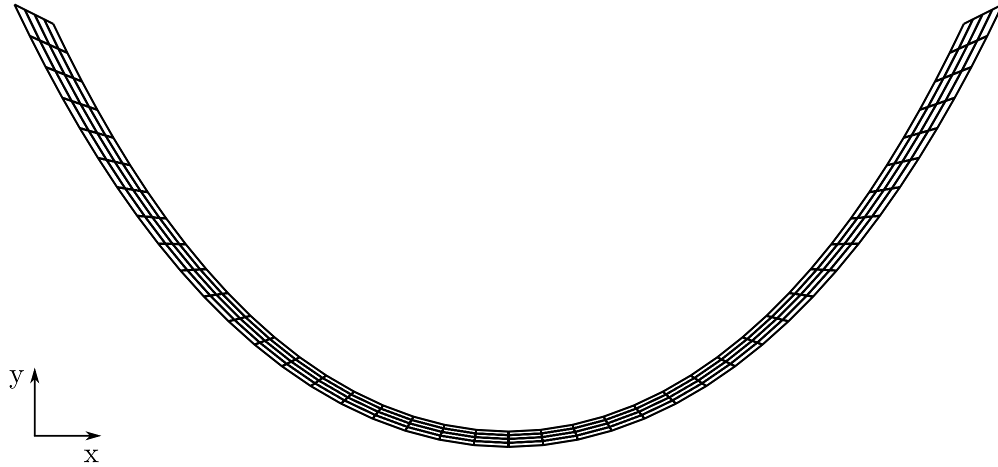



Figure 5.4.: The deformation of the cable due to dead load

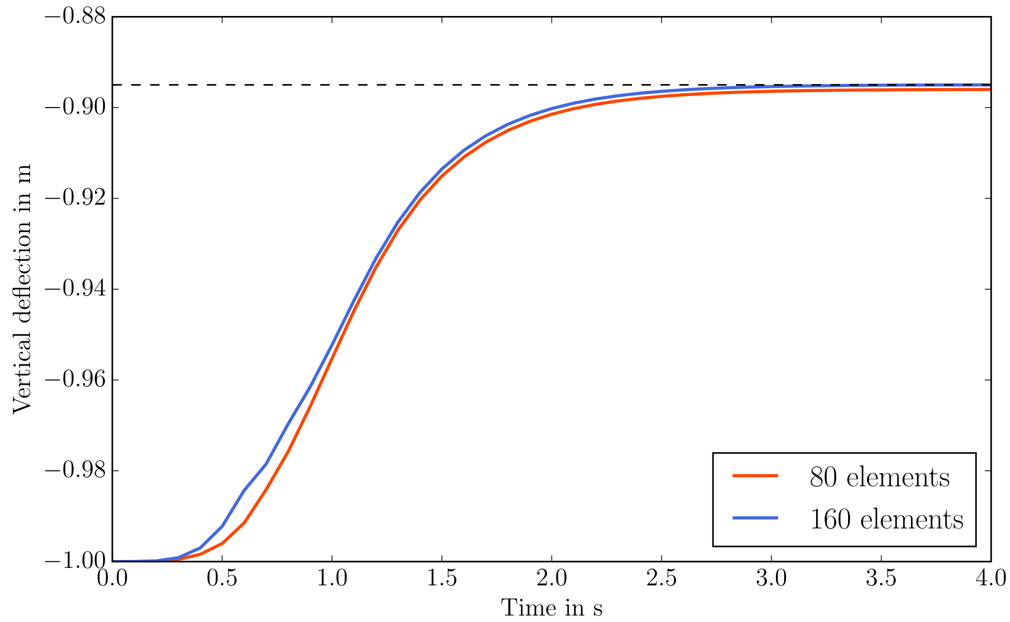



Figure 5.5.: The deflection at the apex of the cable

$x$-direction. This deformation can be traced in Figure 5.2 for the mesh with eight shell elements and in Figure 5.3 for the mesh with 16 elements. In these diagrams the axis of abscissae represents the tip displacement in $x$ which is denoted as $U$ and colored blue and in $z$ which is denoted as $W$ and colored red. The filled circles are data points from the reference solution which are calculated using a mesh with 16 shell elements. The lines display displacements provided from the MBDyn simulation. The force $F$ is plotted on the axis of ordinate. Furthermore, the algebraic signs for $U$ are inverted in order to fit both curves into the quadrant of the diagram.

The tip deflections of the cantilever with eight shell elements deviate marginally from the reference solution. A slight underestimation for $U$ and a slight overestimation for $W$ with increasing force can be observed which is most likely due to the fact that the reference solution uses a mesh with twice the amount of elements. With the same mesh density the results are almost identical.

### 5.1.2. Cable

The deformation of the cable at the end of the simulation with 160 membrane elements is shown in Figure 5.4. Due to the dead load the structure clearly forms a catenary as introduced in Chapter 3. In Figure 5.5 the vertical deflection of the apex is plotted against time. For the sake of clarity, the analytical result of the height of the catenary $\delta = 0.895\,\mathrm{m}$ is included in the diagram as a dashed black line. The red curve represents a cable with 80 elements and the blue curve represents the same structure with twice the amount of elements. Although both variants approach the dashed line almost asymptotically, after four seconds the structure with the higher mesh density coincides exceptionally well with the analytic solution.

### 5.1.3. Circular Membrane Under Uniform Pressure

The deflection $W$ of the mid-point of the membrane in $z$-direction for three different pressure levels at a converged state is plotted against the radius in Figure 5.6. Furthermore, an analytical reference solution was calculated according to Appendix A.2 and included in the diagram as black dashed lines. Especially at the inner region, i.e. near the midpoint, moderate differences between the analytical and the finite element displacements can be observed. Here, MBDyn's solution consistently underestimates the deflection although the shape of the curve compares quite well with the reference curve. This behavior can also be observed in a similar diagram by Bouzidi [10]. However, at the edges the compliance with the theoretical calculation is remarkable. In this context, it needs to be stressed that the simulation was executed applying follower forces and the analytical solution was calculated with absolute forces. Unfortunately, the referenced sources do not specify the employed force types. Typically, considering uniform pressure, follower forces are used. Additionally, Figure 5.7 extends the previous diagram by including the results for applied absolute forces. Here, in turn, the deflection with the use of follower forces tends to be higher at the edges and lower at the midpoint. This fact can be clearly observed when comparing the displacement curves for the highest pressure. In contrast to the absolute force vector in this case, which only possesses a $z$ contribution, the follower force vector

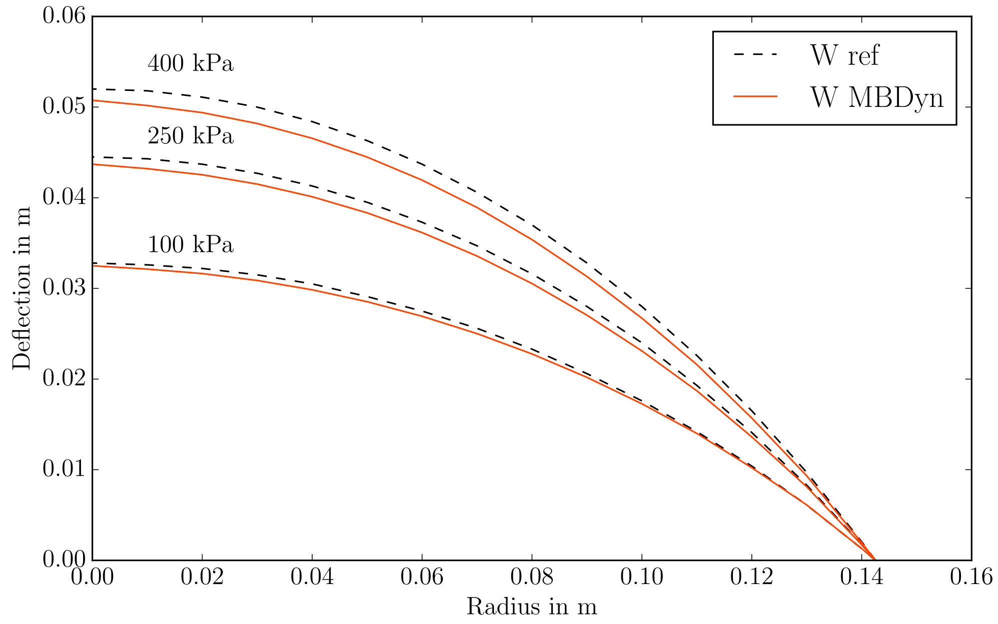



Figure 5.6.: The mid-point deflection of Hencky's problem

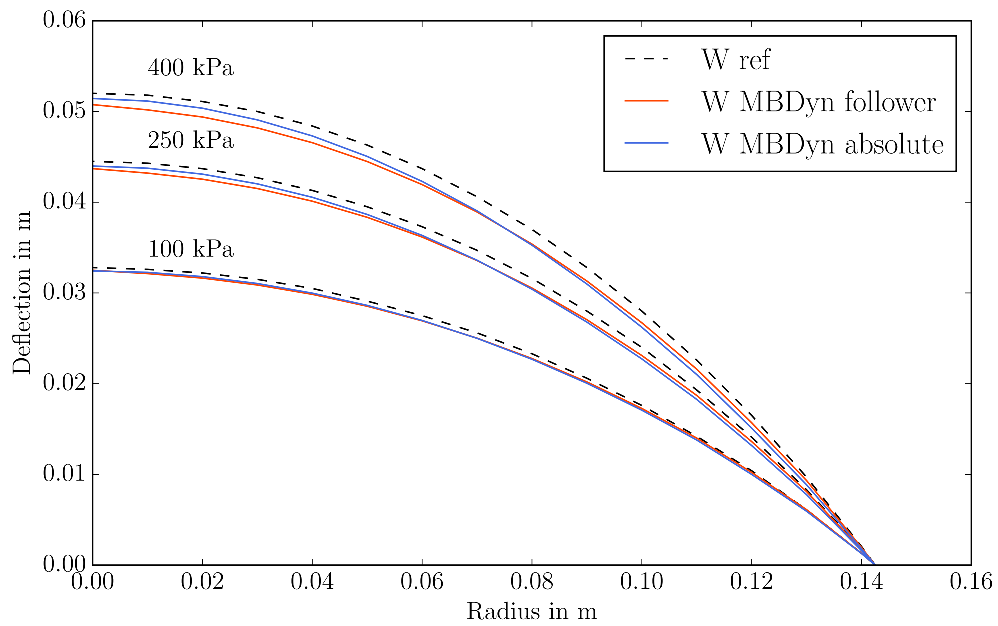



Figure 5.7.: The mid-point deflection of Hencky's problem for absolute and follower forces

Table 5.1.: Deflection values for the midpoint of Hencky's problem

| Source | Deflection in m | | |
|---|---|---|---|
| | 100 kPa | 250 kPa | 400 kPa |
| Pauletti (SATS) [51] | 0.0331 | - | - |
| Pauletti (Ansys) [51] | 0.0319 | - | - |
| Flores (PUMI_MEM) [21] | 0.0348 | - | - |
| MBDyn (follower) | 0.0325 | 0.0437 | 0.0507 |
| MBDyn (absolute) | 0.0324 | 0.0440 | 0.0514 |
| Hencky | 0.0328 | 0.0445 | 0.0520 |

Table 5.2.: Deflection values for the midpoint A of the square airbag

| Source | Deflection in m | | | |
|---|---|---|---|---|
| | 4x4 | 5x5 | 10x10 | 20x20 |
| Contri and Schrefler [14] | 0.209 | 0.217 | - | - |
| Lee [35] | - | - | 0.214 | - |
| Wüchner [30] | 0.2208 | 0.2189 | 0.2175 | - |
| Ziegler [68] | 0.215 | - | 0.216 | - |
| MBDyn | 0.204 | 0.212 | 0.221 | 0.238 |

also contains $x$ and $y$ components depending on the deformed membrane element (see Equation (3.50) in Section 3.1.3). Hence, the differences of these two simulations are physically comprehensible.

The exact values of the displacement at the mid-point and additional reference solutions for a pressure of 100, 250 and 400 kPa are collected in Table 5.1 [21, 51]. Regarding the listed solutions, MBDyn's result for follower and absolute forces is located in between the overall results and almost coincide with the analytical solution for a low pressure. However, with respect to a higher pressure, the results for follower forces are moderately lower than the absolute and analytical counterpart. A higher mesh density employing a mesh with twice the amount of elements did not change the solution noticeably.

### 5.1.4. Square Airbag

In Figures 5.8 to 5.11 the final shape of the airbag is illustrated for a mesh with 16, 25, 100 and 400 elements. The number of elements represents the amount which is used to model one fourth of the upper membrane in the simulation. Subplot (a) shows an isometric view while subplots (b), (c) and (d) display a top view and two side views respectively. The corresponding coordinate axes are labeled accordingly and all values are given in meters as indicated. The most outward grid line in these figures represents the initial square mesh with a side length of 0.84 m. Due to the fact that no wrinkling model is available or has been developed for MBDyn, the deformed structure shows sharp edges in various areas around the mid axes at the edge of the airbag. These irregularities in the shape seem to

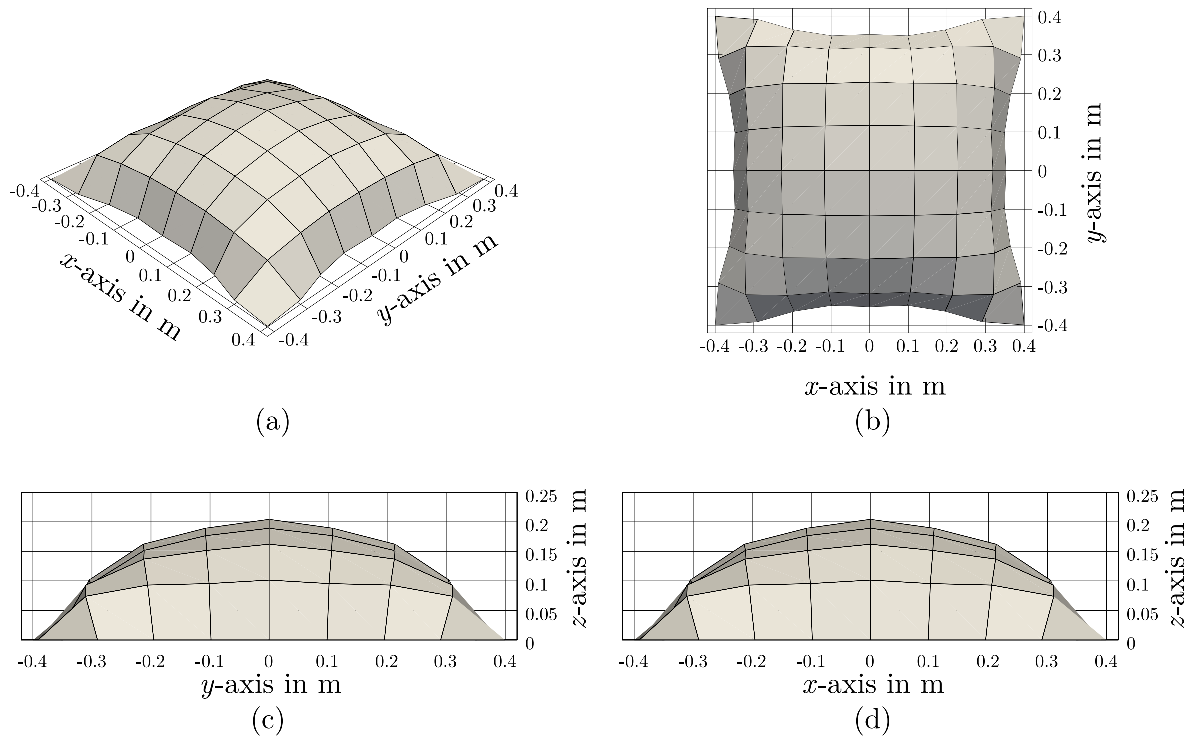



Figure 5.8.: Final shape of the airbag with 16 elements (cf. [66])

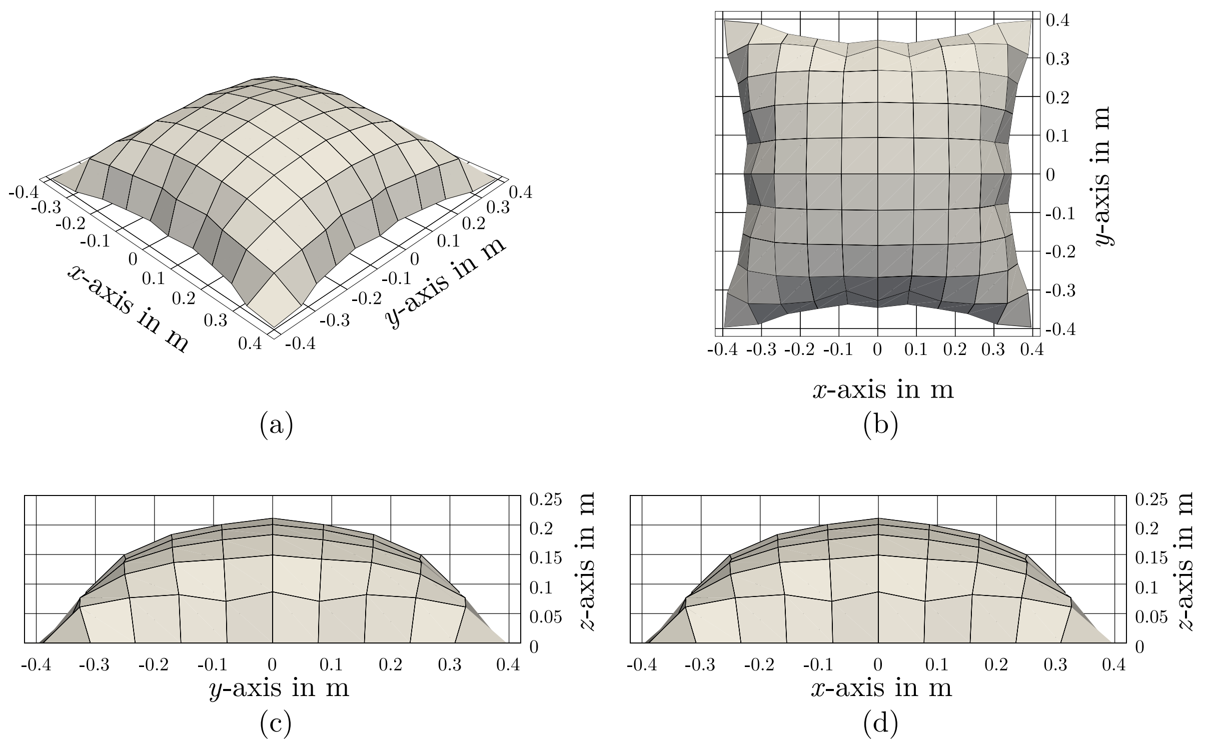



Figure 5.9.: Final shape of the airbag with 25 elements (cf. [66])

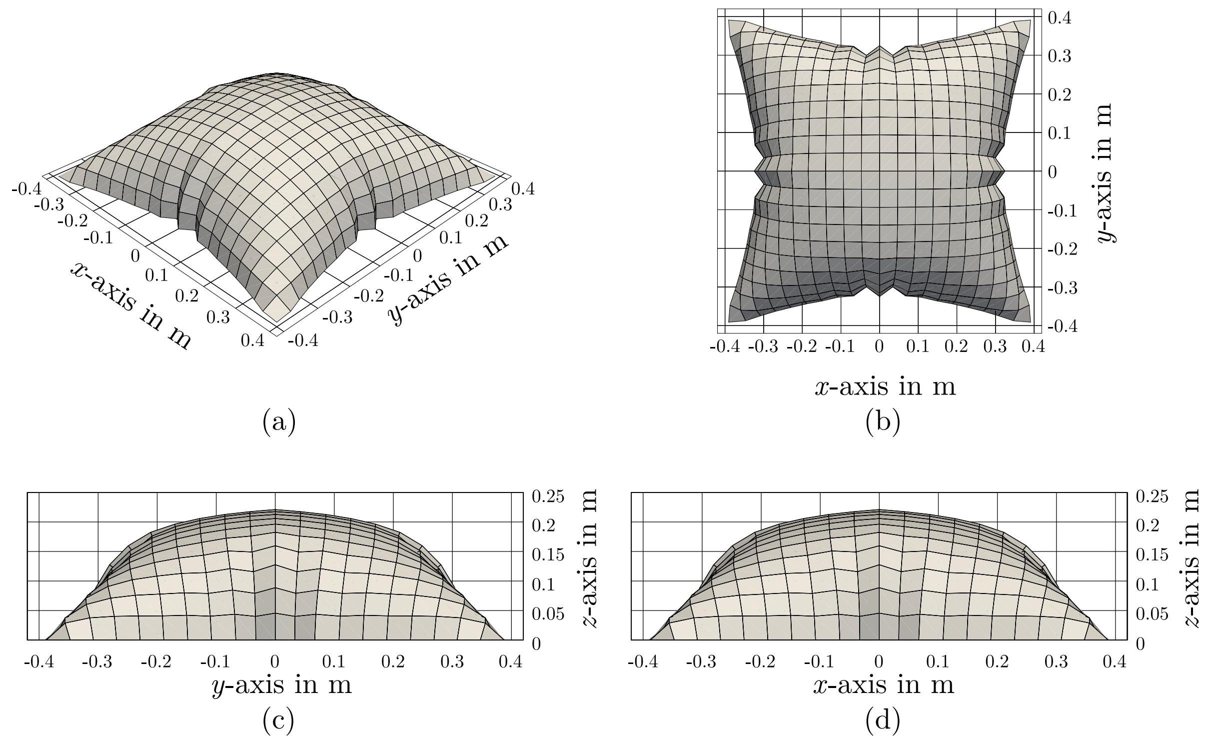



Figure 5.10.: Final shape of the airbag with 100 elements (cf. [66])

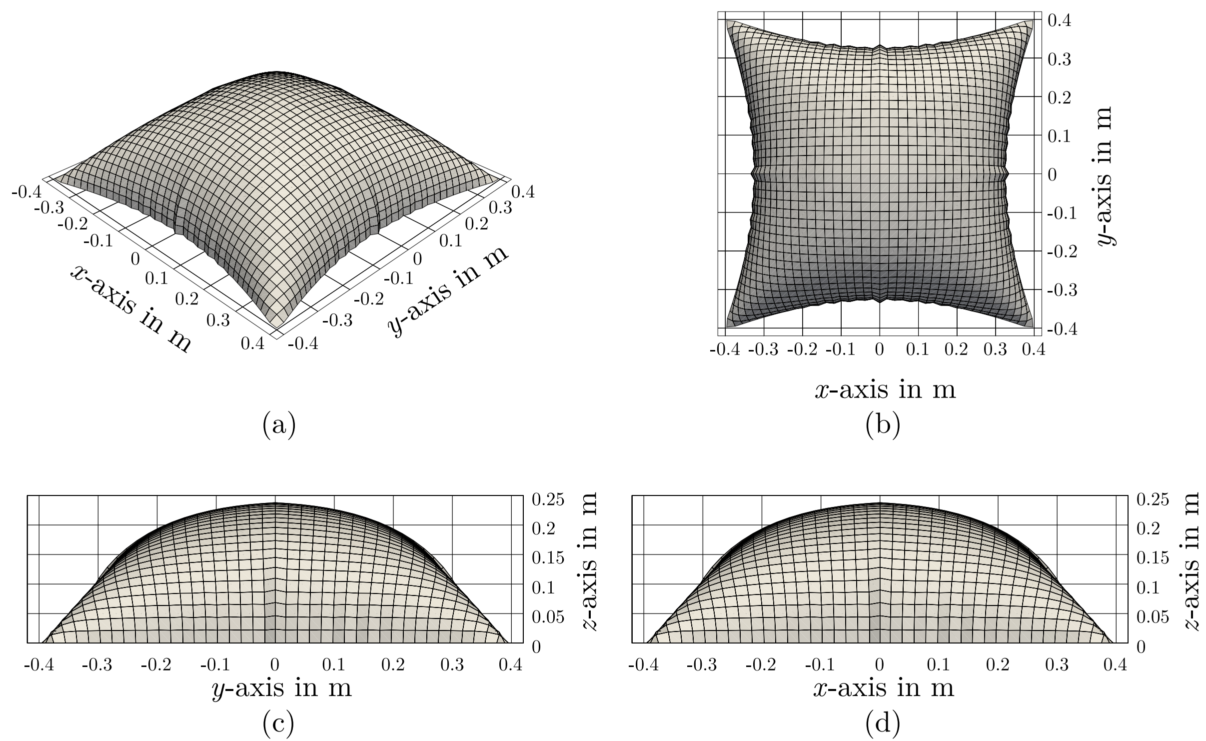



Figure 5.11.: Final shape of the airbag with 400 elements (cf. [66])

diminish with a higher mesh density and almost vanish in Figure 5.11.

Furthermore, results from selected references and MBDyn's solution for the deflection of the midpoint of the airbag are collected in Table 5.2. Although no wrinkling model has been deployed, the difference between these values is negligible. MBDyn's results are marginally lower for meshes with fewer elements and moderately higher with more elements. Unfortunately, no reference solution has been found for the mesh with the highest resolution.

## 5.2. FSI Validation Tests

The implementation and performance of the adapter and the coupling of the structural solver MBDyn and the fluid solver FOAM-FSI through preCICE are examined in this section. Therefore, the results of a popular benchmark in Section 5.2.1 and an experimental FSI simulation of a ram air wing section in Section 5.2.2 are evaluated.

### 5.2.1. 2D Lid-Driven Cavity with Flexible Bottom Membrane

In Figure 5.12 the displacement of the midpoint $M$ of the membrane illustrated in Figure 4.2 (b) for MBDyn and a TU Delft in-house solver specialized on membranes is plotted against time. Both solvers are coupled to FOAM-FSI. It is evident, that the time dependent displacements match well. Furthermore, the solution agreement is quite remarkable for the period up to nine seconds. However, by reaching the first peak oscillation at around ten seconds, a slight overshoot by MBDyn can be observed with respect to the in-house solver result which vanishes throughout the simulation. Similarly, at the bottom of the following amplitudes, MBDyn undershoots constantly. However, since both structural solvers are coupled to the same fluid solver, no distinct evaluation regarding the accuracy

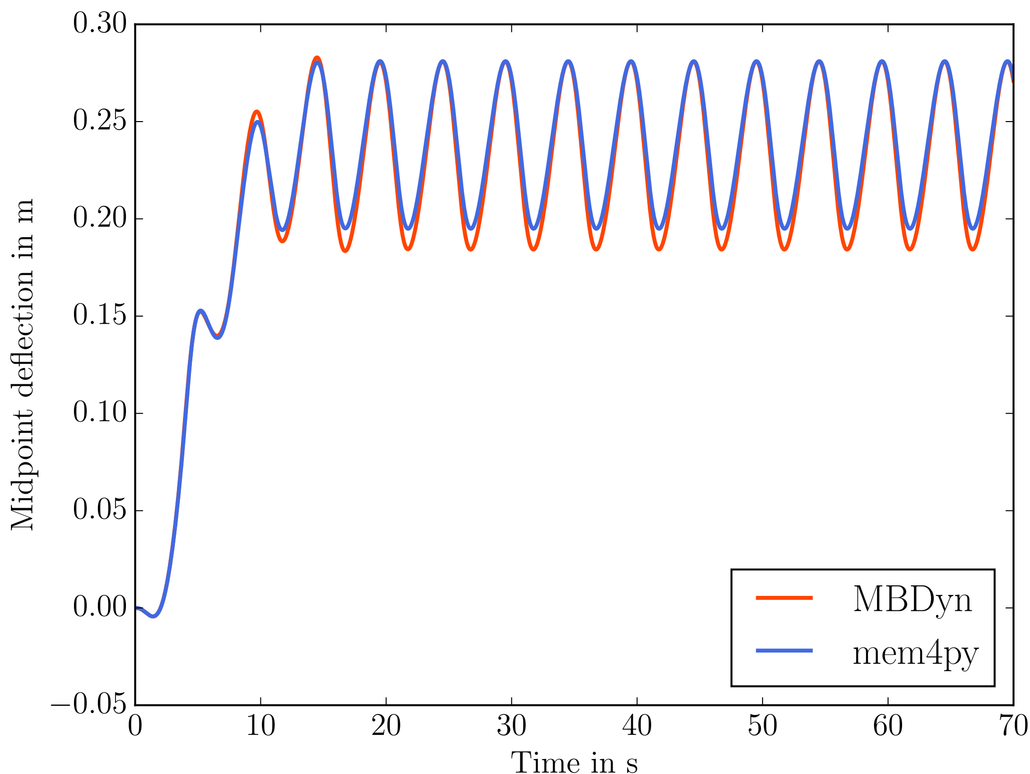



Figure 5.12.: Mid-point deflection of MBDyn and mem4py for the oscillating lid-driven cavity

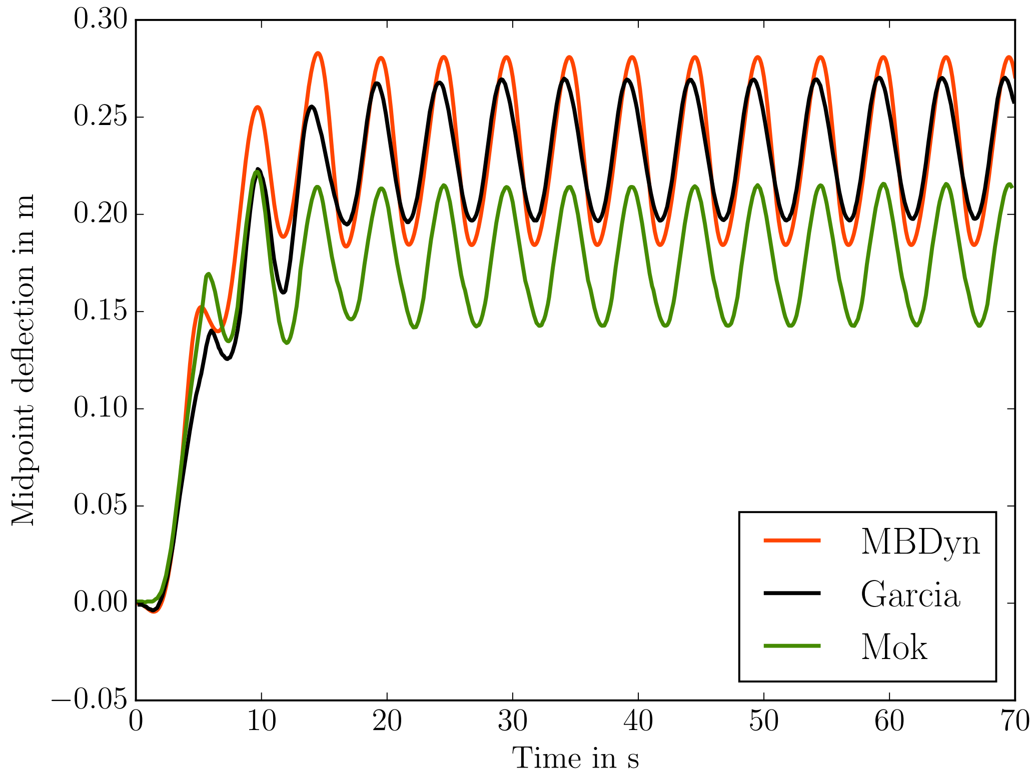



Figure 5.13.: Mid-point deflection of the present work (MBDyn) and results from Mok [46] and Garcia [62] for the oscillating lid-driven cavity

of the results can be made. Therefore, MBDyn's solution is compared to the results from Mok [46] and Garcia [62] in Figure 5.13. It becomes obvious, that there is not one exact solution of the problem probably due to the sheer amount of parameters on the fluid and structural side and in the coupling interface that have to be defined. However, the overall appearance is the same. One common matching property is the frequency of about 0.195 Hz. Although the amplitude of MBDyn's result is higher than that of Garcia, the mean value of the amplitude, i.e. the mean displacement of the membrane, is virtually equal with 0.23 m. Mok's solution possesses a similar amplitude compared to Garcia's. However, the mean displacement and therefore the location of the oscillation is offset from the other results by 0.054 m.

The pressure field of the fluid domain and the interrelated displacement of the membrane for one period is shown in Figure 5.14. This period was chosen since the time from 24 to 29 s comprises the whole oscillation from the maximum to the minimum displacement and back again. The oscillating velocity

$$\overline{u} = \left(1 - \cos\left(\frac{2\pi t}{5}\right)\right) \text{m/s} \tag{5.1}$$

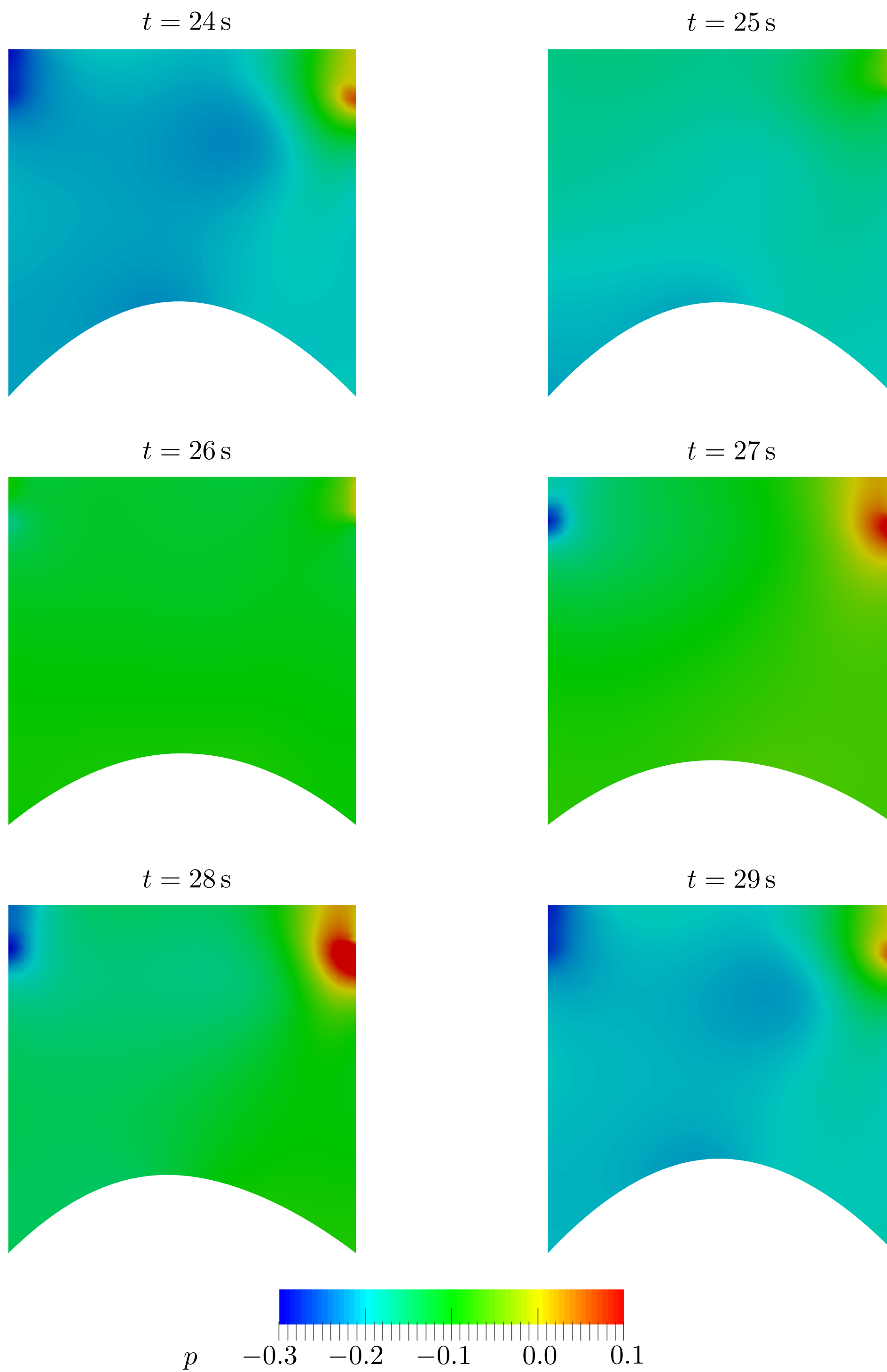



Figure 5.14.: The pressure field of the fluid domain and the deforming membrane within one period

is inducing a changing negative pressure which is responsible for the displacement of the membrane. When the velocity is at its peak, the pressure is the lowest and the maximum displacement is reached which in this period is at $t = 24$ s. With a lower velocity, the pressure gradually rises until the membrane displacement is at its lowest at $t = 26$ s. Since this is an oscillating process, the velocity rises again, the pressure declines and the membrane is deformed until the starting position at the end of the period at $t = 29$ s is regained.

### 5.2.2. Ram-Air Kite Section

As mentioned in Section 4.4.2, prior to the FSI simulation the kite section is inflated due to a stagnation pressure resulting from the inlet at the leading edge. The deformed shape as a result of the applied internal pressure is illustrated in Figure 5.15. It is important

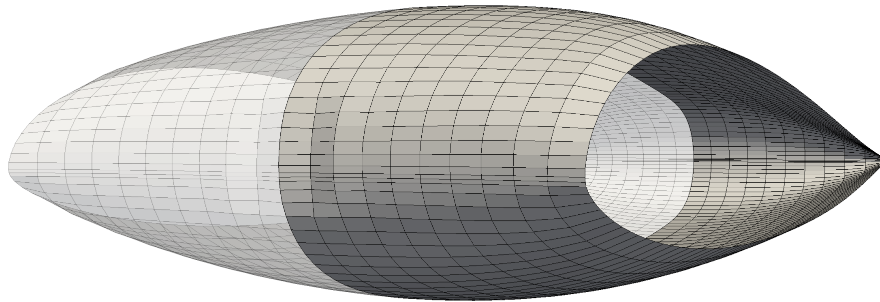


Figure 5.15.: Pressurized shape of the bisected kite section

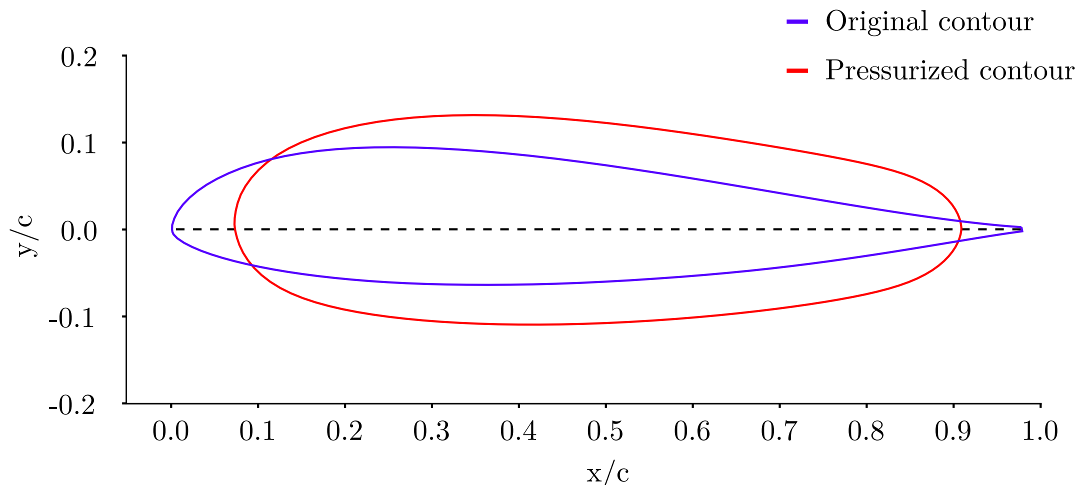



Figure 5.16.: Original and pressurized contour at the mid-plane of the kite

to notice that thanks to symmetry conditions only half of the section is modeled. This fact is clarified in the figure by printing the other half with a lower opacity. Since only one side of the section is clamped the deflection continuously increases until the middle of the structure is reached. For ram-air kites and elastic material this behavior is also known as “ballooning”. The dent in the middle region of the leading edge is physically not comprehensible and probably results from an incorrect modeling feature. A clearer comparison between the original and the pressurized contour at the mid-plane is shown in Figure 5.16. Here, the coordinates in $x$ and $y$ are normalized by the chord length $c$ of the airfoil. Due to the uniform internal pressure the resulting membrane contour resembles an ellipse rather than a conventional airfoil. The section is compressed with respect to the dimension in $x$ and stretched in $y$. The rounding of the leading edge and especially the trailing edge is evident. Moreover, the “ballooning” effect is displayed since the original contour is identical to the contour at the clamped edges.

From this state, the fluid flow is added to the simulation while the stagnation pressure and the resulting internal forces are continuously applied. The pressure distribution at the steady-state is shown in Figure 5.17. In this image the flow enters the domain from the left side and impacts the airfoil at the stagnation point at an AoA of 10°. From there, the flow splits into a low pressure area at the top of the foil and a higher pressure area at the bottom. At the trailing edge the separated fluid flows merge and again create a high pressure area. The deformations of the kite section due to the aerodynamic loads are shown in a more detailed fashion in Figure 5.18 and Figure 5.19 respectively. In the former image, the kite section and its symmetric counterpart are illustrated. In the latter, the normalized $y$ coordinate of the pressurized contour and the contour after the interaction with the fluid flow are printed against the $x$ coordinate. It is apparent, that the leading edge is compressed and dented inwards due to the impacting stream. Hence, a great part of the front half of the section is lifted upwards compared to the initial pressurized configuration.

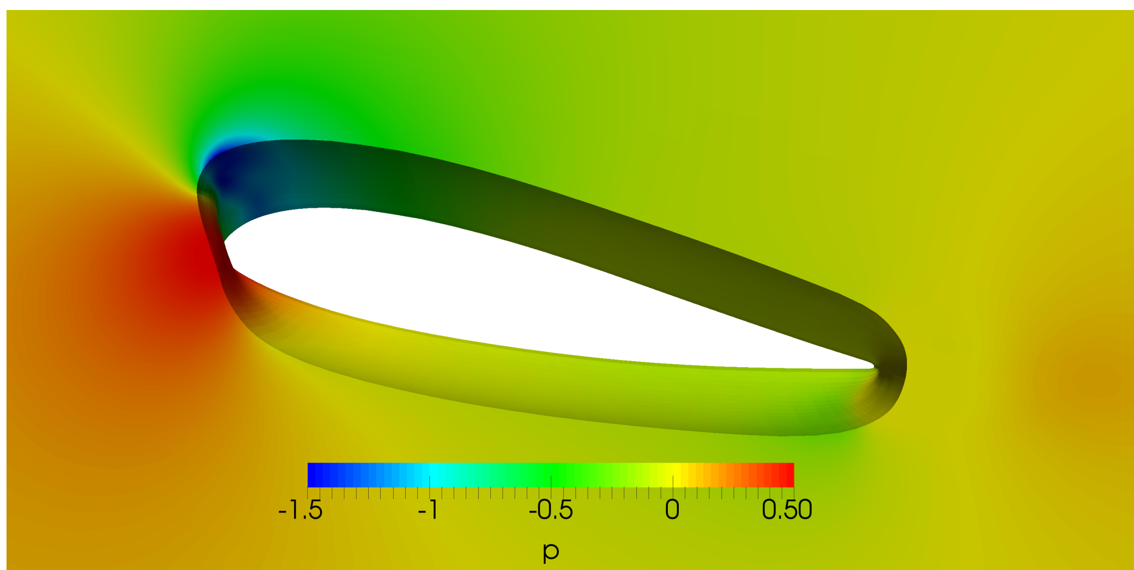



Figure 5.17.: Steady-state pressure distribution of the kite section

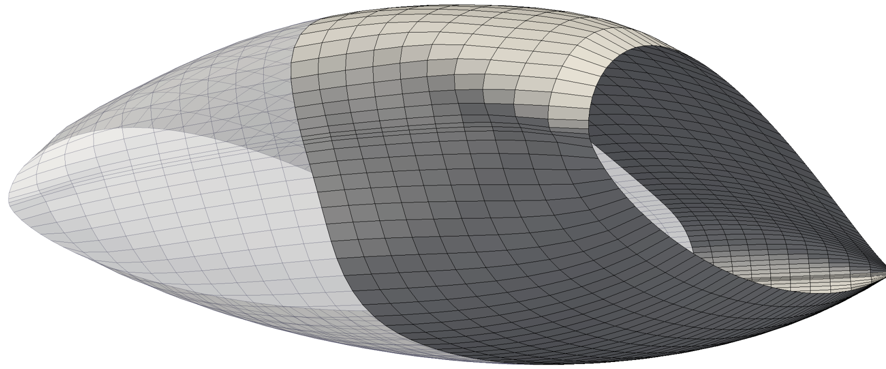


Figure 5.18.: Shape of the bisected kite section after the interaction with the fluid flow

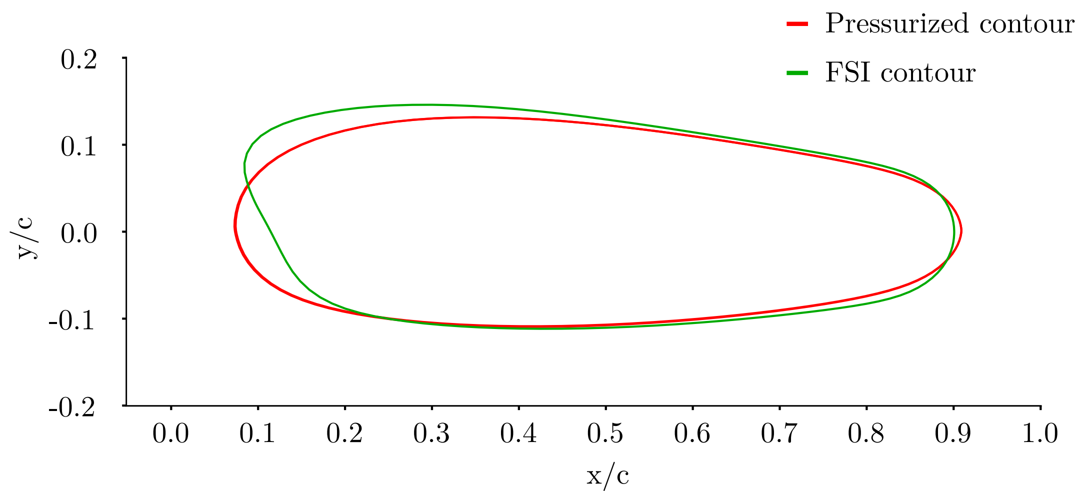



Figure 5.19.: Contour of the kite at the mid-plane after the inflation and after the interaction with the fluid flow

The rear half and especially the trailing edge are rounded off once more.

# 6. Conclusion

## Summary

In order to aid the development and automation of AWE systems, an approach to FSI simulations on soft kites has been created in this thesis. The general goal has been to establish the coupling between the open-source multibody dynamics software MBDyn and the extension to the open-source CFD software OpenFOAM, namely FOAM-FSI, via the coupling library preCICE. Therefore, Chapter 2 provided a concise introduction to AWE and an overview of all participating software. Since the focus of this work has been on the structural side of the FSI simulation, the theory and the handling of the structural solver MBDyn were described in detail in Chapter 3. Additionally, various validation tests to examine essential modeling features for soft kites, i.e. mostly membrane elements and important joints, and MBDyn's general performance were presented. Chapter 4 provided the coupling between the solvers via preCICE. The introduction to the coupling library was followed by the development of the MBDyn-preCICE adapter and some notes about the OpenFOAM-preCICE adapter. Eventually, two tests to assess the coupling mechanism were introduced. The first consisted of a modified benchmark for fluid dynamics in order to include structural dynamics and the second was a first attempt towards FSI simulations on flexible ram-air kite section. Chapter 5 addressed the results of the validation tests regarding the sole structural part and the coupled FSI simulations.

## Results

It is apparent, with respect to the validation tests for the structural solver, that the membrane element formulation and implementation in MBDyn is correct. This is a crucial statement since the canopy of a kite is best modeled with membranes. The overall relative compliance with the reference solutions was satisfactory. Boundary conditions such as clamps and, especially regarding the inflation of the square airbag, symmetry and sliding conditions were applied appropriately. Therefore, MBDyn was and still is considered a suitable structural solver in FSI simulations regarding soft kites.

With respect to the coupling aspect of the fluid and structural solver, an adapter to the preCICE environment for MBDyn has been successfully developed. The functional capability for this adapter and the interaction between MBDyn and FOAM-FSI were verified by the modified lid-driven cavity benchmark. An implicit-serial coupling scheme and a RBF data mapping showed a satisfying performance. Although differences between the references exist, the shape and relative range of the solution are acceptable. Unfortunately, regarding the ram-air kite section, no reference solution was available and thus, no reliable evaluation of this result could be given. However, the presented outcome of the steady-state FSI simulation makes sense in a physical aspect. The inward dent in the structure is

located exactly at the stagnation point where the fluid flow impacts. Moreover, the upward lift of the front half of the airfoil reflects the expected motion when a flexible airfoil is positioned at an AoA of 10° to a fluid flow. With these observations taken into account, the FSI simulation between MBDyn and FOAM-FSI can be considered successful as well.

### Limitations

Limitations of this work can be associated with both the structural solver MBDyn and the coupling adapter. For instance, MBDyn only supports bilinear quadrilateral shell or membrane formulations with respect to the modeling of plate elements. In this case, the formulation was satisfactory. Furthermore, since membrane elements lack the rotational DOF in the formulation, follower forces in their current implementation are not compatible. Therefore, an alternative way of calculating the forces based on the orientation of the membrane surfaces had to be developed (see Section 3.1.3). However, this definition is only tested for the shown test cases and is far from being sophisticated. Regarding the adapter, the serial-explicit/implicit coupling scheme was exclusively used in the simulations. Extending the adapter to a parallel scheme might be trivial, however, no effort has been made in this work. Moreover, only one FSI simulation of a soft ram-air kite was performed at exactly one AoA.

### Future Research

According to the mentioned limitations, future research on the one hand could comprise the extension of membrane elements in MBDyn. This might include membrane elements based on the current formulation that are compatible with the follower forces or even a completely new element formulation. On the other hand, a profound more detailed parameter study on FSI simulations of soft kites based on the presented approach and coupling mechanism could be an interesting research topic as well. Furthermore, the preCICE-MBDyn adapter should be developed into a fully generic version in order to eventually publish it on the preCICE website.

# Bibliography

[1] Ahrem, R.; Beckert, A.; Wendland, H.: A New Multivariant Interpolation Method for Large-scale Spatial Coupling Problems in Aeroelaticity. In: *Conference Proceedings to the International Forum on Aeroelasticity and Structural Dynamics (IFASD)* (2004).

[2] Ahrens, J.; Geveci, B.; Law, C.: ParaView: An End-User Tool for Large Data Visualization / Los Alamos National Laboratory. Los Alamos, NM, USA, 2005. – Technical report. – ISBN 978–0123875822.

[3] Airfoil Tools: *MH 92 15.49%, Martin Hepperle MH 92 parafoil.* http://airfoiltools.com/airfoil/details?airfoil=mh92-il, . – Accessed: 2018-09-11.

[4] Bauer, N.: *Zur Darstellung von Falten in Membranen mit Hilfe der Methode der finiten Elemente = About representation of wrinkles in membranes using the finite element method.* Stuttgart, Universität Stuttgart, Phd Thesis, 1976. http://slubdd.de/katalog?TN_libero_mab214755261. – Accessed: 2018-09-16.

[5] Bechtle, P.; Schelbergen, M.; Schmehl, R.; Zillmann, U.; Watson, S.: *Airborne Wind Energy Resource Analysis.* https://arxiv.org/abs/1808.07718, . – Accessed: 2018-11-11.

[6] Blom, D.: *FOAM-FSI.* https://github.com/davidsblom/FOAM-FSI, . – Accessed: 2018-10-24.

[7] Bosch, A.; Schmehl, R.; Tiso, P.; Rixen, D.: Nonlinear Aeroelasticity, Flight Dynamics and Control of a Flexible Membrane Traction Kite. In: Ahrens, U. (Ed.); Diehl, M. (Ed.); Schmehl, R. (Ed.): *Airborne Wind Energy.* Springer Berlin Heidelberg, 2013. – ISBN 978–3–642–39965–7, Chapter 17, p. 307 – 323.

[8] Bosch, A.; Schmehl, R.; Tiso, P.; Rixen, D.: Dynamic Nonlinear Aeroelastic Model of a Kite for Power Generation. In: *Journal of Guidance, Control, and Dynamics* 37 (2014), June, no. 5, p. 1426 – 1436.

[9] Bosch, H.A.: *Finite element analysis of a kite for power generation*, Delft University of Technology, Faculty of Aerospace Engineering, Thesis, April 2012.

[10] Bouzidi, R.; Ravaut, Y.; Wielgosz, C.: Finite elements for 2D problems of pressurized membranes. In: *Computers and Structures* 81 (2003), November, p. 2479–2490.

[11] Breukels, J.: *An Engineering Methodology for Kite Design.* Delft, Netherlands, Deft University of Technology, Phd Thesis, 2011.

[12] Bungart, M.: *Fluid-Struktur Kopplung an einem RAM-Air-Kiteschirm*, University of Stuttgart, Institute of Aerospace Thermodynamics, Thesis, February 2009.

[13] Chen, G.; Xiong, Q.; Morris, P.J.; Paterson, E.G.; Sergeev, A.; Wang, Y.-

C.: OpenFOAM for Computational Fluid Dynamics. In: *Notices of the American Mathematical Society* 61 (2004), April, no. 4, p. 354 – 363.

[14] Contri, P.; Schrefler, B.A.: A geometrically nonlinear finite element analysis of wrinkled membrane surfaces by a no-compression material model. In: *Communications in Applied Numerical Methods* 4 (1988), January, p. 5 – 15.

[15] Dadd, G.M.; Hudson, D.A.; Shenoi, R.A.: Determination of kite forces using three-dimensional flight trajectories for ship propulsion. In: *Renewable Energy* 36 (2011), October, no. 10, p. 2667 – 2678.

[16] De Wachter, A.: *Deformation and Aerodynamic Performance of a Ram-Air Wing*, Delft University of Technology, Faculty of Aerospace Engineering, Thesis, September 2008.

[17] Diehl, M.: Airborne Wind Energy: Basic Concepts and Physical Foundations. In: Ahrens, U. (Ed.); Diehl, M. (Ed.); Schmehl, R. (Ed.): *Airborne Wind Energy*. Springer Berlin Heidelberg, 2013. – ISBN 978–3–642–39965–7, Chapter 1, p. 3 – 22.

[18] Fechner, U.; Schmehl, R.: Design of a Distributed Kite Power Control System. In: *2012 IEEE International Conference on Control Applications (CCA)*, 2012, p. 800 – 805.

[19] Fichter, W.B.: Some Solutions for the Large Deflections of Uniformly Loaded Circular Membranes / NASA Langley Research Center. 1997 (L-17585). – Technical report.

[20] Flores, P.: *Concepts and Formulations for Spatial Multibody Dynamics*. Vol. 4. Springer, Cham, 2015. – ISBN 978–3–319–16190–7.

[21] Flores, R.; Ortega, E.; Oñate, E.: Explicit dynamic analysis of thin membrane structures. In: *International Centre for Numerical Methods in Engineering (CIMNE)* (2011), February, no. 351.

[22] Folkersma, M.A.M.; Schmehl, R.; Viré, A.: Fluid-Structure Interaction Simulations on Kites. In: *Airborne Wind Energy Conference*. Freiburg, Germany, October 2017.

[23] Folkersma, M.A.M.; Thedens, P.; Schmehl, R.: Fluid-Structure Interaction of Inflatable Wing Section. In: *The* 13$^{\text{th}}$ *OpenFOAM Workshop* (2018), 24 - 29 June, 251 - 253. `http://openfoamworkshop.org/wp-content/uploads/2018/06/OFW13-7-128.pdf`. – Accessed: 2018-11-02.

[24] Gatzhammer, B.: *Efficient and Flexible Partitioned Simulation of Fluid-Structure Interactions*. München, Germany, Technische Universität München, Phd Thesis, 2014.

[25] Geschiere, N.H.: *Dynamic modelling of a flexible kite for power generation*, Delft University of Technology, Faculty of Aerospace Engineering, Thesis, May 2014.

[26] Geuzaine, C.; Remacle, J.-F.: Gmsh: A 3-D Finite Element Mesh Generator with Built-in Pre- and Post-Processing Facilities. In: *International Journal for Numerical Methods in Engineering* 79 (2009), September, no. 11, p. 1309 – 1331.

[27] Ghiringhelli, G.L.; Masarati, P.; Mantegazza, P.: A Multi-Body Implementation of Finite Volumes $C^0$ Beams / Dipartimento di Ingegneria Aerospaziale Politecnico di Milano. Milano, Italy, October . – Technical report.

[28] Gross, D.; Hauger, W.; Wriggers, P.: *Springer-Lehrbuch*. Vol. 4: *Technische*

*Mechanik*. Springer Berlin Heidelberg, 2009. – ISBN 9783540893912.

[29] Hirsch, C.: *Numerical Computation of Internal and External Flows: Fundamentals of Computational Fluid Dynamics*. Butterworth-Heinemann, 2007. – ISBN 978–0–7506–6594–0.

[30] Jarasjarungkiat, A.; Wüchner, R.; Bletzinger, K.-U.: A wrinkling model based on material modification for isotropic and orthotropic membranes. In: *Computer Methods Applied Mechanics Engineering* 197 (2008), p. 773 – 788.

[31] Jasak, H.: OpenFOAM: Open source CFD in research and industry. In: *International Journal of Naval Architecture and Ocean Engineering* 1 (2009), December, no. 2, p. 89 – 94.

[32] Kessler, R.: *Release Notes for foam-extend-4.0*. `https://sourceforge.net/p/foam-extend/foam-extend-4.0/ci/master/tree/ReleaseNotes.txt`, December 2016. – Accessed: 2018-10-02.

[33] Kitepower: *Kitepower - Airborne Wind Energy - Plug & Play Mobile Wind Energy*. `https://kitepower.nl`, . – Accessed: 2018-10-03.

[34] *Chapter* 19.3. In: Kitware, Inc.: *The VTK User's Guide*. Vol. 11. – ISBN 978–1–930934–23–8, p. 469 – 493.

[35] Lee, E.-S.; Youn, S.-K.: Finite element analysis of wrinkling membrane structures with large deformations. In: *Finite Elements in Analysis and Design* 42 (2006), January, no. 8, p. 780 – 791.

[36] Lindner, F.; Mehl, M.; Uekermann, B.: Radial Basis Function Interpolation for Black-Box Multi-Physics Simulations. In: Papadrakakis, M. (Ed.); Oñate, E. (Ed.); Schrefler, B. (Ed.): *VII International Conference on Computational Methods for Coupled Problems in Science and Engineering*. Rhodes Island, Greece, 2017.

[37] Masarati, P.: *Multibody System Dynamics: MBDyn Overview*. Presentation. `https://www.mbdyn.org/userfiles/documents/MBDyn-Overview.pdf`. – Accessed: 2018-08-23.

[38] Masarati, P.: *Comprehensive Multibody AeroServoElastic Analysis of Integrated Rotorcraft Active Controls*. Milano, Italy, Dipartimento di Ingegneria Aerospaziale Politecnico di Milano, Phd Thesis, 1999.

[39] Masarati, P.: Real-time wing-vortex and pressure distribution estimation on wings via displacement and strains in unsteady and transitional flight conditions / Dipartimento di Ingegneria Aerospaziale Politecnico di Milano. Milano, Italy, September 2016. – Technical report.

[40] Masarati, P.: *MBDyn Input File Format Version 1.7.3*. Milano: Dipartimento di Ingegneria Aerospaziale Politecnico di Milano, October 2017.

[41] Masarati, P.: *MBDyn Theory and Developers's Manual Version 1.7.1*. Milano: Dipartimento di Ingegneria Aerospaziale Politecnico di Milano, February 2017.

[42] Masarati, P.; Morandini, M.; Mantegazza, P.: An Efficient Formulation for General-Purpose Multibody/Multiphysics Analysis. In: *Journal of Computational and Nonlinear Dynamics* 9 (2014), October.

[43] Masarati, P.; Morandini, M.; Quaranta, G.; Vescovini, R.: Multibody Analysis of a Micro-Aerial Vehicle Flapping Wing. In: Samin, J.C. (Ed.); Fisette, P. (Ed.):

*MULTIBODY DYNAMICS, ECCOMAS Thematic Conference*. Brussles, Belgium, July 2011.

[44] Masarati, P.; Sitaraman, J.: Coupled CFD/Multibody Analysis of NREL Unsteady Aerodynamic Experiment Phase VI Rotor. In: 49th *AIAA Aerospace Sciences Meeting* (2011), July.

[45] Miller, S.J.: The Method of Least Squares / Mathematics Department Brown University. `https://web.williams.edu/Mathematics/sjmiller/public_html/BrownClasses/54/handouts/MethodLeastSquares.pdf`. Providence, RI, USA, . – Technical report. – Accessed: 2018-09-12.

[46] Mok, D.P.: *Partitionierte Lösungsansätze in der Strukturdynamik und der Fluid-Struktur-Interaktion*. Stuttgart, Germany, University of Stuttgart, Institute for Structural Mechanics, Phd Thesis, 2001.

[47] Munz, C.-D.: *Computational Fluid Dynamics: I. Equations*. University Lecture, 2017.

[48] Neumann, S.: *Fluid-structure interaction of flexible lifting bodies with multi-body dynamics of order-reduced models and the actuator-line method*. Hamburg, Germany, Technische Universität Hamburg-Harburg, Phd Thesis, 2016.

[49] OpenFOAM: *About OpenFOAM*. `https://cfd.direct/openfoam/about/`, . – Accessed: 2018-09-16.

[50] OpenFOAM: *The Open Source CFD Toolbox User Guide*. `https://sourceforge.net/projects/openfoamplus/files/v1806/UserGuide.pdf`, July 2018. – Accessed: 2018-10-05.

[51] Pauletti, R.M.O.; Guirardi, D.M.; Deifeld, T.E.C.: Argyris' Natural Membrane Finite Element Revisited. In: Oñate, E. (Ed.); Kröplin, B. (Ed.): *International Conference on Textile Composites and Inflatable Structures*. Barcelona, April 2005.

[52] preCICE: *preCICE Wiki*. `https://github.com/precice/precice/wiki`, . – Accessed: 2018-10-01.

[53] *Chapter* 16.7. In: Press, W.H.; Teukolsky, S.A.; Vetterling, W.T.; Flannery, B.P.: *Numerical Recipes in C*. Vol. 2. Cambridge, Great Britain : Cambridge University Press, 1992. – ISBN 0521431085, p. 742 – 752.

[54] Quaranta, G.; Masarati, P.; Morandini, M.: Multibody Free Software for Teaching Purposes. In: *MULTIBODY DYNAMICS 2005, ECCOMAS Thematic Conference*. Madrid, Spain, June 2005.

[55] Richter, T.: *Lecture Notes in Computational Science and Engineering, Vol. 118*. Vol. 1: *Fluid-structure Interactions: Models, Analysis and Finite Elements*. Springer, Cham, 2017. – ISBN 978–3–319–63970–3.

[56] Saeedi, M.; Wüchner, R.; Bletzinger, K.-U.: Multi-fidelity Fluid-Structure Interaction Analysis of a Membrane Wing. In: *International Journal of Aerospace and Mechanical Engineering* 9 (2015), no. 1.

[57] Schwoll, J.: *Finite Element approach for statically loaded inflatable kite structures*, Delft University of Technology, Faculty of Aerospace Engineering, Thesis, June 2012.

[58] Solcia, T.; Masarati, P.; Morandini, M.: A Membrane Element for Micro-Aerial Vehicle Fluid-Structure Interaction. In: *The 2nd Joint International Conference on*

*Multibody System Dynamics*. Stuttgart, Germany, May 2012.

[59] Sze, K.Y.; Liu, X. H.; Lo, S. H.: Popular benchmark problems for geometric nonlinear analysis of shells. In: *Finite Elements in Analysis and Design* 40 (2004), July, no. 11, p. 1551–1569.

[60] Terink, E.J.; Breukels, J.; Schmehl, R.; Ockels, W.: Flight Dynamics and Stability of a Tethered Inflatable Kiteplane. In: *AIAA Journal of Aircraft* 48 (2011), no. 2, p. 503 – 513.

[61] Vlugt, R. van d.; Peschel, J.; Schmehl, R.: Design and Experimental Characterization of a Pumping Kite Power System. In: Ahrens, U. (Ed.); Diehl, M. (Ed.); Schmehl, R. (Ed.): *Airborne Wind Energy*. Springer Berlin Heidelberg, 2013. – ISBN 978–3–642–39965–7, Chapter 23, p. 403 – 425.

[62] Vàzquez, J.G.V.: *Nonlinear Analysis of Orthotropic Membrane and Shell Structures Including Fluid-Structure Interaction*. Barcelona, Spain, Universitat Politècnica de Catalunya, Phd Thesis, 2007.

[63] Wall, W.A.: *Fluid-Struktur-Interaktion mit stabilisierten Finiten Elementen*. Stuttgart, Germany, University of Stuttgart, Institute for Structural Mechanics, Phd Thesis, 1999.

[64] Williams, P.; Lansdorp, B.; Ruiterkamp, R.; Ockels, W.: Modeling, Simulation, and Testing of Surf Kites for Power Generation. In: *AIAA Modeling and Simulation Technologies Conference and Exhibit*, 2008.

[65] Witkowski, W.: 4-Node combined shell element with semi-EAS-ANS strain interpolations in 6-parameter shell theories with drilling degrees of freedom. In: *Computational Mechanics* 43 (2009), January, no. 2, p. 307–319.

[66] Wu, T.-Y.; Ting, E.C.: Large deflection analysis of 3D membrane structures by a 4-node quadrilateral intrinsic element. In: *Thin-Walled Structures* 46 (2008), November, no. 3, p. 261 – 275.

[67] *Chapter* 4.3. In: Zahn, M.: *Unix-Netzwerkprogrammierung*. Springer Berlin Heidelberg, 2006 (X.systems.press). – ISBN 10 3–540–00299–5, p. 179.

[68] Ziegler, R.; Wagner, W.; Bletzinger, K.-U.: A Finite Element Model for the Analysis of Wrinkled Membrane Structures. In: *International Journal of Space Structures* 18 (2003), no. 1, p. 1 – 14.

[69] Zorilla, R.: *KratosMultiphysics*. `https://github.com/KratosMultiphysics/Examples/tree/master/fluid_structure_interaction/validation/fsi_lid_driven_cavity`, . – Accessed: 2018-09-10.

# A. Appendix A

## A.1. Newton's Method

The equation

$$x \cdot \sinh\left(\frac{1}{x}\right) - \sqrt{2} = 0 \tag{A.1}$$

can be solved via a Newton's method

$$x_{n+1} = x_n - \frac{f\left(x_n\right)}{f'\left(x_n\right)} \tag{A.2}$$

with

$$f\left(x_n\right) = x_n \cdot \sinh\left(\frac{1}{x_n}\right) - \sqrt{2} \tag{A.3}$$

and

$$f'\left(x_n\right) = \sinh\frac{1}{x_n} - \frac{\cosh\left(\frac{1}{x_n}\right)}{x_n}. \tag{A.4}$$

With a starting point at $x_0 = 1$ and a residual tolerance of $1 \cdot 10^{-3}$ Newton's method results in six steps until convergence that are summarized in Table A.1.

Table A.1.: Newton iteration

| $n$ | $x_n$ | $f\left(x_n\right)$ | $f'\left(x_n\right)$ |
|---|---|---|---|
| 0 | 1 | -0.239012 | -0.367879 |
| 1 | 0.35 | 1.61796 | -16.2188 |
| 2 | 0.450055 | 0.637341 | -5.81105 |
| 3 | 0.559733 | 0.209401 | -2.58091 |
| 4 | 0.640867 | 0.0439520 | -1.60281 |
| 5 | 0.668289 | 0.00304513 | -1.38774 |
| 6 | 0.670483 | 0.00001701296 | -1.37218 |

## A.2. Hencky's Theory

According to Fichter [19] the displacement $w$ of a circular, isotropic membrane with a clamped edge under a uniform lateral loading, also known as Hencky's problem, is approximated by

$$w = a \cdot W(\rho) = q^{1/3} \sum_{n=0}^{\infty} a_{2n} \left(1 - \rho^{2n+2}\right) \tag{A.5}$$

where $W(\rho) = w/a$ is the dimensionless deflection, $\rho = r/a$ is the dimensionless coordinate relating the radial coordinate $r$ and the membrane radius $a$. The loading parameter $q$

$$q = \frac{pa}{E_t} \tag{A.6}$$

consists of the uniform lateral loading $p$, the membrane radius $a$ and the membrane modulus $E_t$. The coefficients for the power series $a_{2n}$ are used to approximate the deflection. For $n = 10$ the coefficients result in

$$\begin{aligned}
a_0 &= \frac{1}{b_0}\\
a_2 &= \frac{1}{2 \cdot b_0^4}\\
a_4 &= \frac{5}{9 \cdot b_0^7}\\
a_6 &= \frac{55}{72 \cdot b_0^{10}}\\
a_8 &= \frac{7}{6 \cdot b_0^{13}}\\
a_{10} &= \frac{205}{108 \cdot b_0^{16}}\\
a_{12} &= \frac{17\,051}{5\,292 \cdot b_0^{19}}\\
a_{14} &= \frac{2\,864\,485}{508\,032 \cdot b_0^{22}}\\
a_{16} &= \frac{103\,863\,265}{10\,287\,648 \cdot b_0^{25}}\\
a_{18} &= \frac{27\,047\,983}{1\,469\,664 \cdot b_0^{28}}\\
a_{20} &= \frac{42\,367\,613\,873}{1\,244\,805\,408 \cdot b_0^{31}}.
\end{aligned} \tag{A.7}$$

The value of $b_0$ depends on the Poisson's ratio and Fichter defined it as shown in Table A.2. The displacements for a circular membrane with radius $a = 0.1425$ m, membrane modulus $E_t = 311\,488$ N/m, Poisson's number $\nu = 0.34$ and a uniform lateral loading $p = 100$ kPa are compiled in Table A.3.

Table A.2.: Value for $b_0$

| Poisson's ratio | $b_0$ |
|---|---|
| 0.2 | 1.6827 |
| 0.3 | 1.7244 |
| 0.4 | 1.7769 |

Table A.3.: Exemplary displacements for Hencky's problem for a radius $a = 0.1425$ m, a membrane modulus $E_t = 311\,488$ N/m, a Poisson's number $\nu = 0.34$ and a uniform lateral loading $p = 100$ kPa

| Dimensionless coordinate $\rho$ | Dimensionless displacement $W(\rho)$ | Displacement $w$ in m |
|---|---|---|
| 1 | 0.0091 | 0.0 |
| 0.982 | 0.009 | 0.0013 |
| 0.912 | 0.043 | 0.0061 |
| 0.842 | 0.073 | 0.0104 |
| 0.772 | 0.1 | 0.0142 |
| 0.702 | 0.124 | 0.0176 |
| 0.632 | 0.145 | 0.0206 |
| 0.561 | 0.163 | 0.0233 |
| 0.491 | 0.179 | 0.0256 |
| 0.421 | 0.193 | 0.0275 |
| 0.351 | 0.204 | 0.0291 |
| 0.281 | 0.214 | 0.0305 |
| 0.211 | 0.221 | 0.0315 |
| 0.140 | 0.226 | 0.0322 |
| 0.07 | 0.229 | 0.0326 |
| 0.0 | 0.230 | 0.0328 |

# B. Appendix B

## B.1. Input File for Hencky's Problem

Listing B.1 illustrates an exemplary input file for MBDyn. The definition of the nodes and elements are built up with line breaks in order to improve the readability. In the actual input file these are all single lines.

Listing B.1: hencky.mbd

```
begin: data;
   problem: initial value;
end: data;

begin: initial value;
   initial time: 0;
   final time: 2;
   time step: 0.01;
   method: ms, 0.6;
   tolerance: 0.001,0.001;
   max iterations: 100;
   linear solver: umfpack;
   output: iterations;
   threads: disable;
end: initial value;

begin: control data;
   structural nodes: 261;
   beams: 0;
   plates: 236;
   joints: 261;
   rigid bodies: 261;
   forces: 213;
   output frequency: 10;
end: control data;

begin: nodes;
structural: 0, dynamic,
   0.1425, 0.0, 0.0,
   eye,
   null,
   null;
structural: 1, dynamic,
   0.141280892746, 0.0185999823914, 0.0,
   eye,
```

```
    null,
    null;
...
structural: 259, dynamic,
    0.0547564673631, 0.0945253436869, 0.0,
    eye,
    null,
    null;
structural: 260, dynamic,
    0.0913646204, -0.0551304856463, 0.0,
    eye,
    null,
    null;
end: nodes;


begin: elements;
joint: 2000, total pin joint,
    0,
        position, null,
        position orientation, eye,
        rotation orientation, eye,
    position, 0.1425, 0.0, 0.0,
    position orientation, eye,
    rotation orientation, eye,
    position constraint, active, active, active, null,
    orientation constraint, active, active, active, null;
joint: 2001, total pin joint,
    1,
        position, null,
        position orientation, eye,
        rotation orientation, eye,
    position, 0.141280892746, 0.0185999823914, 0.0,
    position orientation, eye,
    rotation orientation, eye,
    position constraint, active, active, active, null,
    orientation constraint, active, active, active, null;
...
joint: 2259, total pin joint,
    259
        position, null,
        position orientation, eye,
        rotation orientation, eye,
    position, 0.0547564673631, 0.0945253436869, 0.0,
    position orientation, eye,
    rotation orientation, eye,
    position constraint, inactive, inactive, inactive, null,
    orientation constraint, active, active, active, null;
joint: 2260, total pin joint,
    260,
        position, null,
```

```
        position orientation, eye,
        rotation orientation, eye,
    position, 0.0913646204, -0.0551304856463, 0.0,
    position orientation, eye,
    rotation orientation, eye,
    position constraint, inactive, inactive, inactive, null,
    orientation constraint, active, active, active, null;

body: 1000,
    0,
    0.00181930124246,
    null,
    diag, 0.0, 0.0, 0.0;
body: 1001,
    1,
    0.00191644662507,
    null,
    diag, 0.0, 0.0, 0.0;
...
body: 1259, 259,
    0.00151805777363,
    null,
    diag, 0.0, 0.0, 0.0;
body: 1260, 260,
    0.00173345878021,
    null,
    diag, 0.0, 0.0, 0.0;

membrane4eas: 0,
    18, 96, 99, 17,
    isotropic,
        E, 31148800.0,
        nu, 0.34,
        thickness, 0.01;
membrane4eas: 1,
    96, 75, 97, 99,
    isotropic,
        E, 31148800.0,
        nu, 0.34,
        thickness, 0.01;
...
membrane4eas: 234,
    260, 245, 83, 230,
    isotropic,
        E, 31148800.0,
        nu, 0.34,
        thickness, 0.01;
membrane4eas: 235,
    232, 260, 230, 92,
    isotropic,
        E, 31148800.0,
```

```
        nu, 0.34,
        thickness, 0.01;

force: 30000, follower,
    48,
        position, 0., 0., 0.,
        single, 0., 0., 1.,
        ramp, 20.2872473474, 0, 1, 0;
force: 30001, follower,
    49,
        position, 0., 0., 0.,
        single, 0., 0., 1.,
        ramp, 20.1685196483, 0, 1, 0;
...
force: 30211, follower,
    259,
        position, 0., 0., 0.,
        single, 0., 0., 1.,
        ramp, 15.1805777363, 0, 1, 0;
force: 30212, follower,
    260,
        position, 0., 0., 0.,
        single, 0., 0., 1.,
        ramp, 17.3345878021, 0, 1, 0;

end: elements;
```

## B.2. Mesh to Input File Conversion: Hencky's Problem

The Python script for the conversion of the mesh generated by Gmsh to an input file readable by MBDyn is presented in Listing B.2.

Listing B.2: hencky_msh_to_mbd.py

```
import numpy as np

# physical and numerical environment
T = 2                   # time [s]
T0 = 0                  # initial time [s]
dt = 0.01               # timeStep [s]
method = "ms, 0.6"      # integration (multistep: "ms, 0.6" | "hope, 0.4";
    backward euler: "bdf"; "crank nicolson"; "implicit euler")
linSolv = "umfpack"     # linear solver ("umfpack"; "naive"; "klu")
rho =  1000             # density [kg/m^3]
t = 0.01                # thickness [m]
E = 311488.e2           # E-Module [N/m^2]
nu = 0.34               # poisson
stress = 100.e3         # pressure [N/m^2]
toleranceR = 1.e-6      # residual test
toleranceS = 1.e-6      # solution error test
derTol = 1.e6           # derivatives tolerance
max_iterations = 100    # nr. of iterations before abortion
freq = 10               # output frequency

# read lines from msh-file to get nodes and elements
nodes = []              # nodes-List
cells = []             # cells-List
supports = []           # supports-List
with open('henky.msh') as fin:
    readNodes = False
    readElements = False
    for line in fin:
        if "$Nodes" in line:
            readNodes = True
            continue
        elif "$EndNodes" in line:
            readNodes = False
        elif "$Elements" in line:
            readElements = True
            continue
        elif "$EndElements" in line:
            readElements = False
        if readNodes:
            nodes.append(map(float,line.split()[1:]))
        elif readElements:
            element = map(int,line.split()[1:])
            if len(element) < 2:
                continue
```

```
            if element[0] == 1:
                supports.extend(element[-2:])
            elif element[0] == 15:
                pass
            else:
                cells.append(element[-4:])
nodes = np.array(nodes[1:])
supports = [x-1 for x in supports]          # mesh file starts from 1, python
    from 0
supports = list(set(supports))
nodes_free = np.delete(nodes, supports,axis=0) # nodes that are not supports
nodes_constrained = nodes[supports]          # nodes that are supports

# syntax nodes
nodess = "structural: {}, dynamic, {}, {}, {}, eye, null, null;"
mbnodes = '\n'.join([nodess.format(i,n[0],n[1],n[2]) for i, n in
    enumerate(nodes)])

# syntax joints
# joints for the supports
joints_constrained = "joint: {}, total pin joint, {}, position, null, position
    orientation, eye, rotation orientation, eye, position, {}, {}, {}, position
    orientation, eye, rotation orientation, eye, position constraint, active,
    active, active, null, orientation constraint, active, active, active, null;"
mbjoints_constrained =
    '\n'.join([joints_constrained.format(i+2000,i,n[0],n[1],n[2]) for i, n in
    enumerate(nodes_constrained)])
# joints for the rest of the nodes in order to use them with membrane elements
joints_free = "joint: {}, total pin joint, {}, position, null, position
    orientation, eye, rotation orientation, eye, position, {}, {}, {}, position
    orientation, eye, rotation orientation, eye, position constraint, inactive,
    inactive, inactive, null, orientation constraint, active, active, active,
    null;"
mbjoints_free =
    '\n'.join([joints_free.format(i+2000+len(nodes_constrained),i+len(nodes_constrained),n[0],n[1],n
    for i, n in enumerate(nodes_free)])
mbjoints = mbjoints_constrained + mbjoints_free

# syntax membranes
membranes = "membrane4eas: {}, {}, {}, {}, {}, isotropic, E, {}, nu, {},
    thickness, {};"
mbmembranes = '\n'.join([membranes.format(i, s[0]-1, s[1]-1, s[2]-1, s[3]-1, E,
    nu, t) for i, s in enumerate(cells)])

# calculation of forces and masses
forces = np.zeros(len(nodes))
masses = np.zeros(len(nodes))
areas = []
# compute the area of an arbitrary rectangle by splitting it into two triangles
length = lambda v: (v[0]**2+ v[1]**2+v[2]**2)**0.5
for s in cells:
```

```
    n1,n2,n3,n4 = s[0]-1,s[1]-1,s[2]-1,s[3]-1
    x1,x2,x3,x4 = nodes[n1],nodes[n2],nodes[n3],nodes[n4]
    # Distance formula
    l1 = length(x1-x2)
    l2 = length(x2-x3)
    l3 = length(x3-x4)
    l4 = length(x4-x1)
    a1 = np.arccos(np.dot((x4-x1)/l4,(x2-x1)/l1))
    a2 = np.arccos(np.dot((x4-x3)/l3,(x2-x3)/l2))
    A = 0.5*l1*l4*np.sin(a1)+0.5*l2*l3*np.sin(a2)
    areas.append(A)
    forces[[n1,n2,n3,n4]] += stress*A/4.
    masses[[n1,n2,n3,n4]] += t*A*rho/4.
    print x1, x2
    print l1
    print A
    quit()

# Syntax forces
forcesRamp = forces
nodal_forces = "force: {}, follower, {}, position, 0., 0., 0., single, 0., 0.,
    1., ramp, {}, 0, 1, 0;"
mbforces = '\n'.join([nodal_forces.format(i+30000, i, f) for i, f in
    enumerate(forcesRamp)])

# Syntax masses
bodies = "body: {}, {}, {}, null, diag, 0.0, 0.0, 0.0;"
mbbodies = '\n'.join([bodies.format(i+1000, i, m) for i, m in enumerate(masses)])

# Additional information for mbd-file
nnodes = len(nodes)          # number of nodes
njoints = len(nodes)         # number of joints
nplates = len(cells)         # number of cells
nrigid_bodies = len(nodes)   # number of rigid bodies
nforces = len(forces)        # number of forces

# Write mbd-file (different for membranes and cells)
with open('henky.mbd','w') as fout:
    fout.write('''begin: data;
    problem: initial value;
end: data;

begin: initial value;
    initial time: {T0};
    final time: {T};
    time step: {dt};
    method: {method};
    tolerance: {toleranceR},{toleranceS};
    max iterations: {max_iterations};
    derivatives tolerance: {derTol};
    linear solver: {linSolv};
```

```
    output: iterations;
    threads: disable;
end: initial value;

begin: control data;
    structural nodes: {nnodes};
    beams: 0;
    plates: {nplates};
    joints: {njoints};
    rigid bodies: {nrigid_bodies};
    forces: {nforces};
    output frequency: {freq};
end: control data;

begin: nodes;

{mbnodes}

end: nodes;

begin: elements;

{mbjoints}

{mbbodies}

{mbmembranes}

{mbforces}

end: elements;
'''.format(**locals()))
```

## B.3. VTK File Format: Hencky's Problem

Listing B.3 contains an exemplary .vtk file for Hencky's problem for the first time step to be read by ParaView.

Listing B.3: hencky1.vtk

```
# vtk DataFile Version 2.0
Hencky Unstructured Grid
ASCII
DATASET UNSTRUCTURED_GRID

POINTS 261 float
0.1425 0.0 0.0
0.141280892746 0.0185999823914 0.0
...
0.0547564673631 0.0945253436869 0.0
0.0913646204 -0.0551304856463 0.0

CELLS 236 1180
4  18  96  99  17
4  96  75  97  99
...
4  260  245  83  230
4  232  260  230  92

CELL_TYPES 236
9
9
...
9
9

POINT_DATA 261
VECTORS Displacement float
0.0 0.0 0.0
0.0 0.0 0.0
...
0.0002218726369 0.0003796063131 0.006756271
0.0003727496 -0.0002300943537 0.007082572

VECTORS Force float
0.0 0.0 0.0
0.0 0.0 0.0
...
0.1246057 0.214934 1.537743
0.231393 -0.1389717 1.753867
```

## B.4. Input File to VTK Conversion: Hencky's Problem

The Python script for the conversion of the input file to the .vtk file is presented in Listing B.4.

Listing B.4: hencky_mbd_to_vtk.py

```
import numpy as np

# generate lists
nodes = []
cells = []
ncells = []
position = []
forces = []
force_labels = []
Fzeros = []

# read lines from mbd-file to obtain nodes and elements
with open('henky.mbd', 'r+') as f:
    with open('henky.mbd') as fin:
        for line in fin:
            if "structural:" in line:
                nodes.append(line.split(',')[2:5])
            if "membrane" in line:
                cells.append(line.split(',')[1:5])
                ncells.append("9")
            if "force:" in line:
                frc.append(line.split(',')[9:10])
nodes = np.array(nodes)
nodes = nodes.astype(np.float)
cells = np.array(cells)
cells = cells.astype(np.int)
force_labels = np.array(force_labels)

# syntax nodes
node_positions = "{} {} {}"
mbnodes = '\n'.join([node_positions.format(n[0],n[1],n[2]) for i, n in
    enumerate(nodes)])

# syntax cells
cell_nodes = "4 {} {} {} {}"
mbcells = '\n'.join([cell_nodes.format(n[0],n[1],n[2],n[3]) for i, n in
    enumerate(cells)])
ncell_nodes = "{}"
mbncells = '\n'.join([ncell_nodes.format(n[0]) for i, n in enumerate(ncells)])

# additional information for vtk-file
P = len(nodes)
F = len(force_labels)
C = len(cells)
```

```
# if no forces for supports are defined, fill up with zeros
FP = P - F
Fzeros = np.zeros(FP)
Fzeros = [[x, x, x] for x in Fzeros]
C_size = C*4 + C      # 4 for quadrlateral elements

# syntax forcess
Fzero_nodes="{} {} {}"
mbFzeros = '\n'.join([Fzero_nodes.format(n[0],n[1],n[2]) for i, n in
    enumerate(Fzeros)])

# read lines from mov-file to get position of nodes
with open('henky.mov') as fin2:
    for line in fin2:
        position.append(line.split()[1:4])
position = np.array(position, dtype=float)

# read lines from frc-file to get forces
with open('henky.frc') as fin3:
    for line in fin3:
        forces.append(line.split()[2:5])
forces = np.array(forces)

# Calculate displacements and assign forces
for i, (ps, fs) in enumerate(zip(position[::P], forces[::F])): # enumerate every
    Pth/Fth element
    displacement = position[i*P:i*P+P] - nodes # calculate the displacement of
        the nodes
    displacement = np.array(displacement, dtype=float)
    dispString = '\n'.join(['{} {} {}'.format(*d) for d in displacement])
    forcess = forces[i*F:i*F+F]
    forcess = np.array(forcess, dtype=float)
    frcString = '\n'.join(['{} {} {}'.format(*f) for f in forcess])
    # Write vtk-files
    with open('henky{}.vtk'.format(i),'w') as fout:
        fout.write('''# vtk DataFile Version 2.0
Henky Unstructured Grid
ASCII
DATASET UNSTRUCTURED_GRID

POINTS {P} float
{mbnodes}

CELLS {C} {C_size}
{mbcells}

CELL_TYPES {C}
{mbncells}

POINT_DATA {P}
VECTORS Displacement float
```

```
{dispString}
VECTORS Force float
{mbFzeros}
{frcString}

'''.format(**locals()))
```

## B.5. Python Code for Hencky's Problem

Listing B.5 contains an exemplary `Python` script with respect to the API of MBDyn.

Listing B.5: hencky.py

```
import sys
import os
import tempfile
import numpy as np
from numpy import *

# path to MBDyn's python interface
sys.path.append('/usr/local/mbdyn/libexec/mbpy')
from mbc_py_interface import mbcNodal

# create temporary directory
tmpdir = tempfile.mkdtemp('', '.mbdyn_');

# full path of socket
path = tmpdir + '/mbdyn.sock'

# include path to environment
os.environ['MBSOCK'] = path

# execute MBDyn
os.system('henky.mbd -o henky > henky.txt 2>&1 &')

# set up socket
host = ""
port = 0
timeout = -1          # -1 = forever (will not expire)
verbose = 0
data_and_next = 1     # true: MBDyn expects the packet to also contain forces
refnode = 0
nodes = 261           # number of nodes
labels = 0
rot = 0               # orientation vector
accels = 1            # output: accelerations
nodal = mbcNodal(path, host, port, timeout, verbose, data_and_next, refnode,
    nodes, labels, rot, accels)

# set pointers to nodes
nodal.negotiate()

# read nodes and cells from mbd-file
nodes_list = []
cells_list = []
with open('henky.mbd') as fin:
    for line in fin:
        if "structural:" in line:
            nodes_list.append(line.split(',')[2:5])
```

```
        if "membrane4eas:" in line:
            cells_list.append(line.split(',')[1:5])
nodes_list = np.array(nodes_list, dtype=float)
cells_list = np.array(cells_list, dtype=int)

# timestep n and size dt, iteration i and the max. iteration number
n = 0
dt = 0.01
i = 0
max_i = 5

# set up loop
while 1:
    if (nodal.recv()): # recv: receive kinematics from peer (peer = MBDyn)
        break

    # increment timestep
    if (i == 0):
        n = n + 1;

    # compute forces based on the cells' areas (unstructured mesh)
    forces = np.zeros(len(nodes_list))
    masses = np.zeros(len(nodes_list))
    stress = 100.e3
    areas = []
    length = lambda v: (v[0]**2+ v[1]**2+v[2]**2)**0.5 # abs()
    for s in cells_list:
        n1,n2,n3,n4 = s[0],s[1],s[2],s[3]
        x1,x2,x3,x4 = nodes_list[n1],nodes_list[n2],nodes_list[n3],nodes_list[n4]
        l1 = length(x1-x2)
        l2 = length(x2-x3)
        l3 = length(x3-x4)
        l4 = length(x4-x1)
        a1 = np.arccos(np.dot((x4-x1)/l4,(x2-x1)/l1))
        a2 = np.arccos(np.dot((x4-x3)/l3,(x2-x3)/l2))
        A = 0.5*l1*l4*np.sin(a1)+0.5*l2*l3*np.sin(a2)
        areas.append(A)
        forces[[n1,n2,n3,n4]] += stress*A/4.

    # computation of averaged plane from four points
    normal = []   # normal vector to plane
    A_temp = []
    b_temp = []
    for s, c in enumerate(cells_list):
        n1, n2, n3, n4 = c[0], c[1], c[2], c[3]
        # Get nodes from MBDyn at runtime:
        # nodal.n_x[i*3]   x-value of node i
        # nodal.n_x[i*3+1] y-value of node i
        # nodal.n_x[i*3+2] z-value of node i
        x1, y1, z1 = nodal.n_x[n1*3], nodal.n_x[n1*3+1], nodal.n_x[n1*3+2]
        x2, y2, z2 = nodal.n_x[n2*3], nodal.n_x[n2*3+1], nodal.n_x[n2*3+2]
```

```
    x3, y3, z3 = nodal.n_x[n3*3], nodal.n_x[n3*3+1], nodal.n_x[n3*3+2]
    x4, y4, z4 = nodal.n_x[n4*3], nodal.n_x[n4*3+1], nodal.n_x[n4*3+2]
    xs = [x1, x2, x3, x4]
    ys = [y1, y2, y3, y4]
    zs = [z1, z2, z3, z4]
    for t in range(len(xs)):
        A_temp.append([xs[t], ys[t], 1])
        b_temp.append(zs[t])
    b = np.matrix(b_temp).T
    A = np.matrix(A_temp)
    fit = (A.T * A).I * A.T * b
    normal.append([-fit[0], -fit[1], 1])
    del A_temp[:] # clean up for next iteration
    del b_temp[:]
normal = np.reshape((np.array(normal, dtype=float)), (-1,3))

# averaging of normals of surrounding plane
quantity = []     # number of elements around node (1|2|4)
avg_normal_x = []
avg_normal_y = []
avg_normal_z = []
for u, d in enumerate(nodes_list):
    # extract all cells where node u is involved
    rows, cols = np.where(cells_list == u)
    quantity = float(len(rows))
    normal_u = normal[rows]  # normals associated with node r
    avg_normal_x.append(1/quantity*sum(normal_u[:,0]))
    avg_normal_y.append(1/quantity*sum(normal_u[:,1]))
    avg_normal_z.append(1/quantity*sum(normal_u[:,2]))
avg_normal = np.zeros(len(nodes_list)*3)
avg_normal = np.reshape(avg_normal,(-1,3))
avg_normal[:,0] = avg_normal_x
avg_normal[:,1] = avg_normal_y
avg_normal[:,2] = avg_normal_z

# assign forces
# ramping up from 0 to 1 s and a time step of dt = 0.01
if n <= (1/dt):   # n = 100 --> t = 1 s
    forces_x = np.zeros(nodes)
    forces_y = np.zeros(nodes)
    forces_z = np.zeros(nodes)
    # force at middle point
    for w, d in enumerate(nodes_list):
        if w <= 47:    # 0 - 47: clamped nodes
            forces_x[w] = 0
            forces_y[w] = 0
            forces_z[w] = 0
        else:
            p = forces[w]*dt*n
            abs_normal=np.linalg.norm(avg_normal[w])
            forces_x[w] = p*abs_normal*avg_normal_x[w]
```

```
                forces_y[w] = p*abs_normal*avg_normal_y[w]
                forces_z[w] = p*abs_normal*avg_normal_z[w]
    if n > (1/dt):
        forces_x = np.zeros(nodes)
        forces_y = np.zeros(nodes)
        forces_z = np.zeros(nodes)
        # force at middle point
        for w, d in enumerate(nodes_list):
            if w <= 47:    # 0 - 47: clamped nodes
                forces_x[w] = 0
                forces_y[w] = 0
                forces_z[w] = 0
            if w >= 48:
                p = forces[w]
                abs_normal=np.linalg.norm(avg_normal[w])
                forces_x[w] = p*abs_normal*avg_normal_x[w]
                forces_y[w] = p*abs_normal*avg_normal_y[w]
                forces_z[w] = p*abs_normal*avg_normal_z[w]
    forces = np.reshape(nodal.n_f,(-1,3))
    forces[:,0] = forces_x
    forces[:,1] = forces_y
    forces[:,2] = forces_z
    nodal.n_f = forces    # nodal.n_f: define forces for MBDyn

    print 'iteration =', i
    print 'timestep =', n
    print 'time     =', n*dt,'s', '\n'

    # increment iteration
    i = i + 1
    if (i == max_i):
        i = 0

    if (nodal.send(i == 0)): #send: send forces to peer
        break
nodal.destroy()
os.rmdir(tmpdir)
```