Partitioned Solution Strategies for Strongly-Coupled Fluid-Structure Interaction Problems in Maritime Applications

Vom Promotionsausschuss der
Technischen Universität Hamburg

zur Erlangung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

genehmigte Dissertation

von
Marcel König, M.Sc.

aus
Pinneberg

2018

Vorsitzender des Prüfungsausschusses
Prof. Dr.-Ing. Otto von Estorff

Gutachter
1. Gutachter: Prof. Dr.-Ing. habil. Alexander Düster
2. Gutachter: Prof. Dr.-Ing. Moustafa Abdel-Maksoud

Tag der mündlichen Prüfung: 6. Juli 2018

Acknowledgements

The present thesis has emerged from the joint program *Maritime safety aspects regarding installation and maintenance of offshore wind turbines*, funded by the *Hamburg Research and Science Foundation*, and the research project *Fluid-structure interaction and optimization of floating platforms for offshore wind turbines*, financed by the *Federal Ministry for Economic Affairs and Energy*, which I conducted at the Institute for Ship Structural Design and Analysis at Hamburg University of Technology from July 2013 to June 2017. Many great people have contributed to my work and I would like to say "thank you" to those without whom this thesis would not have been possible.

First and foremost, I would like to express my gratitude to Prof. Dr.-Ing. habil. Alexander Düster for the supervision of my PhD thesis. His broad expertise, his helpfulness, and our fruitful discussions guided me through my time at university during the past years. I would also like to thank Prof. Dr.-Ing. Moustafa Abdel-Maksoud for acting as a co-supervisor for this thesis.

Moreover, I am much obliged to the *Hamburg Research and Science Foundation* and the *Federal Ministry for Economic Affairs and Energy* for providing the necessary funding for the projects I have been working on and for sharpening the scope of my research.

I also owe credits to my colleagues at the Institute of Ship Structural Design and Analysis who have always been an inspiration for my work and who made my four years at the institute a pleasure. In particular, I would like to thank my colleague Lars Radtke for our lively discussions, his constant helpfulness, and his willingness to share his knowledge.

Last but not least, I am indebted to my wonderful wife Marieke, and I would like to express my deepest thanks to her – for her endless support, her patience, and her love.

Contents

List of Acronyms

AAA	Almost always use `auto`
AABB	Axis-aligned bounding box
ALE	Arbitrary Lagrangian-Eulerian
APDL	ANSYS parametric design language
API	Application programming interface
BEM	Boundary element method
BMWi	Bundesministerium für Wirtschaft und Energie (Federal Ministry for Economic Affairs and Energy)
CFD	Computational fluid dynamics
COG	Center of gravity
CPU	Central processing unit
FE	Finite element
FEM	Finite element method
FSI	Fluid-structure interaction
FV	Finite volume
FVM	Finite volume method
HF	High frequency
HyStOH	Hydrodynamische und strukturmechanische Optimierung eines Halbtauchers für Offshore-Windenergieanlagen (Hydrodynamic and structural optimization of a floating platform for offshore wind turbines)
JONSWAP	Joint North Sea Wave Project
LF	Low frequency
MPI	Message passing interface
MVP	Most vexing parse
ODR	One definition rule
OWT	Offshore wind turbine
pImpl	Pointer to implementation
POD	Plain old data
RAII	Resource acquisition is initialization
RANS	Reynolds-averaged Navier-Stokes
RAO	Response amplitude operator
RBF	Radial basis function
RVO	Return value optimization
SFINAE	Substitution failure is not an error
VOF	Volume of fluid

List of Symbols

Nomenclature
In this thesis, the following notation for the distinction of scalars, vectors, tensors, and matrices is introduced. Scalars s are denoted in italic font. Vectors $\boldsymbol{v}$ from the Euclidean space as well as second-order tensors $\boldsymbol{T}$ defined as linear maps between vector spaces are typeset in bold italic font. Bold calligraphic symbols are used to represent tensors $\boldsymbol{\mathcal{C}}$ of order higher than two. While "short" local discrete vectors $\boldsymbol{v}$ and matrices $\boldsymbol{M}$ are typeset in bold italic font, "long" global discrete vectors $\mathbf{v}$ and matrices $\mathbf{M}$ are denoted by an upright bold symbol.

Symbol	Description
$\operatorname{card}(\mathcal{S})$	Cardinality of the set $\mathcal{S}$
$\boldsymbol{\Phi} \circ \boldsymbol{\Psi}$	Composition of the transformations $\boldsymbol{\Phi}$ and $\boldsymbol{\Psi}$
$\boldsymbol{\Phi}^{-1}$	Inverse of the transformation $\boldsymbol{\Phi}$
δu	Variation of u
Δu	Increment of u
u_0	Initial value of u
u_{ref}	Reference value of u
$u\vert_{x=x^*}$	u evaluated at $x = x^*$
$\bar{u}$	Prescribed value of u (Chapter 2, 3) Mean value of u (Chapter 4, 6)
$\boldsymbol{u}^{\mathrm{T}}$	Transpose of the vector or tensor $\boldsymbol{u}$
$\mathbf{v}^{-1}$	$:= \mathbf{v}/\lVert\mathbf{v}\rVert_2^2$, Moore-Penrose inverse of the vector $\mathbf{v}$
$\mathrm{D}u \cdot \boldsymbol{v}$	Directional derivative of u in the direction of the vector $\boldsymbol{v}$
$\partial u/\partial x$	Partial derivative of u with respect to x
$\partial u/\partial t\vert_{\boldsymbol{X}}$	Derivative of u with respect to time t, with the material coordinate $\boldsymbol{X}$ held fixed
$\partial u/\partial t\vert_{\boldsymbol{\chi}}$	Derivative of u with respect to time t, with the referential coordinate $\boldsymbol{\chi}$ held fixed
$\mathrm{D}u/\mathrm{D}t$	$:= \partial u/\partial t\vert_{\boldsymbol{X}}$, material derivative of u
$\dot{u}$	$:= \mathrm{D}u/\mathrm{D}t$
$\ddot{u}$	$:= \mathrm{D}^2u/\mathrm{D}t^2$
$\operatorname{grad} s$	$:= \sum_{i=1}^{d} \partial s/\partial x_i \boldsymbol{e}_i$, gradient of the scalar s
$\operatorname{Grad} s$	$:= \operatorname{grad}_{\boldsymbol{X}} s$, gradient of the scalar s in the reference configuration
$\operatorname{grad} \boldsymbol{v}$	$:= \sum_{i,j=1}^{d} \partial v_j/\partial x_i \boldsymbol{e}_i \otimes \boldsymbol{e}_j$, gradient of the vector $\boldsymbol{v}$
$\operatorname{Grad} \boldsymbol{v}$	$:= \operatorname{grad}_{\boldsymbol{X}} \boldsymbol{v}$, gradient of the vector $\boldsymbol{v}$ in the reference configuration
$\operatorname{curl} \boldsymbol{v}$	$:= \boldsymbol{e}_1(\partial v_3/\partial x_2 - \partial v_2/\partial x_3) + \boldsymbol{e}_2(\partial v_1/\partial x_3 - \partial v_3/\partial x_1) + \boldsymbol{e}_3(\partial v_2/\partial x_1 - \partial v_1/\partial x_2)$, curl of the vector $\boldsymbol{v}$
$\operatorname{div} \boldsymbol{v}$	$:= \sum_{i=1}^{d} \partial v_i/\partial x_i$, divergence of the vector $\boldsymbol{v}$
$\operatorname{Div} \boldsymbol{v}$	$:= \operatorname{div}_{\boldsymbol{X}} \boldsymbol{v}$, divergence of the vector $\boldsymbol{v}$ in the reference configuration

$\operatorname{div} \boldsymbol{T}$	$:= \sum_{i,j=1}^{d} \partial T_{ij} / \partial x_i \boldsymbol{e}_j$, divergence of the tensor $\boldsymbol{T}$
$\operatorname{Div} \boldsymbol{T}$	$:= \operatorname{div}_{\boldsymbol{X}} \boldsymbol{T}$, divergence of the tensor $\boldsymbol{T}$ in the reference configuration
Δs	$:= \operatorname{div} \operatorname{grad} s$, Laplacian of the scalar s
$\boldsymbol{u} \cdot \boldsymbol{v}$	$:= \sum_{i=1}^{d} u_i v_i$, scalar product of the vectors $\boldsymbol{u}$ and $\boldsymbol{v}$
$\boldsymbol{S} \cdot \boldsymbol{T}$	$:= \sum_{i,j=1}^{d} S_{ij} T_{ij}$, scalar product of the second-order tensors $\boldsymbol{S}$ and $\boldsymbol{T}$
$\boldsymbol{v} \cdot \boldsymbol{T}$	$:= \sum_{i,j=1}^{d} v_j T_{ji} \boldsymbol{e}_i$, dot product of the vector $\boldsymbol{v}$ and the second-order tensor $\boldsymbol{S}$
$\boldsymbol{T}\boldsymbol{v}$	$:= \sum_{i,j=1}^{d} T_{ij} v_j \boldsymbol{e}_i$, dot product of the second-order tensor $\boldsymbol{S}$ and the vector $\boldsymbol{v}$
$\boldsymbol{u} \otimes \boldsymbol{v}$	$:= \boldsymbol{u}\boldsymbol{v}^{\mathrm{T}}$, dyadic product of the vectors $\boldsymbol{u}$ and $\boldsymbol{v}$
$\boldsymbol{S}\boldsymbol{T}$	$:= \sum_{i,j,k=1}^{d} S_{ij} T_{jk} \boldsymbol{e}_i \otimes \boldsymbol{e}_k$, tensor product of the second-order tensors $\boldsymbol{S}$ and $\boldsymbol{T}$
$\det \boldsymbol{T}$	Determinant of the tensor $\boldsymbol{T}$
$\operatorname{tr} \boldsymbol{T}$	$:= \sum_{i=1}^{d} T_{ii}$, trace of the tensor $\boldsymbol{T}$
$\lVert \boldsymbol{x} \rVert_p$	$:= \left(\sum_{i=1}^{d} \lvert x_i \rvert^p \right)^{1/p}$, p-norm of the vector $\boldsymbol{x}$
$\boldsymbol{H}^1(\Omega)$	Sobolev space of component-wise weak differentiable functions in $\boldsymbol{L}^2(\Omega)$ with derivatives in $\boldsymbol{L}^2(\Omega)$
$\boldsymbol{H}_0^1(\Omega)$	Set of functions in $\boldsymbol{H}^1(\Omega)$ with vanishing trace on the Dirichlet boundary of Ω
$\boldsymbol{L}^2(\Omega)$	Hilbert space of component-wise Lebesgue-measurable functions with integrable squares over Ω
$\boldsymbol{0}$	Zero vector
a	Axis
$\boldsymbol{a}$	Acceleration vector
$\mathrm{d}a$	Area element in the current configuration
$\mathrm{d}\boldsymbol{a}$	$:= \boldsymbol{n}\mathrm{d}a$, directed area element in the current configuration
A_k	Influence coefficient
$\mathrm{d}A$	Area element in the reference configuration
$\mathrm{d}\boldsymbol{A}$	$:= \boldsymbol{N}\mathrm{d}A$, directed area element in the reference configuration
$\mathcal{A}$	Convergence acceleration scheme
$\boldsymbol{b}$	Body force vector
B_k	Influence coefficient
$\mathcal{B}$	Set of bounding boxes
$\boldsymbol{B}$	$:= \boldsymbol{F}\boldsymbol{F}^{\mathrm{T}}$, left Cauchy-Green deformation tensor
$\hat{\boldsymbol{B}}$	$:= J^{2/3}\boldsymbol{I}$, dilatational (i.e., volume-changing or volumetric) part of the left Cauchy-Green deformation tensor $\boldsymbol{B}$
$\bar{\boldsymbol{B}}$	$:= J^{-2/3}\boldsymbol{B}$, distortional (i.e., volume-preserving or isochoric) part of the left Cauchy-Green deformation tensor $\boldsymbol{B}$
c	Speed of sound (Chapter 2) Damping constant (Chapter 4, 5)
c_p	Specific heat capacity
C	Child of a tree node (Chapter 4) Center of gravity (Chapter 6, 7)
C_k	Influence coefficient
$\boldsymbol{C}$	$:= \boldsymbol{F}^{\mathrm{T}}\boldsymbol{F}$, right Cauchy-Green deformation tensor

$\hat{\boldsymbol{C}}$	$:= J^{2/3}\boldsymbol{I}$, dilatational (i.e, volume-changing or volumetric) part of the right Cauchy-Green deformation tensor $\boldsymbol{C}$
$\bar{\boldsymbol{C}}$	$:= J^{-2/3}\boldsymbol{C}$, distortional (i.e., volume-preserving or isochoric) part of the right Cauchy-Green deformation tensor $\boldsymbol{C}$
$\boldsymbol{\mathcal{C}}$	Fourth-order material tensor
$\mathbf{C}$	Damping matrix
d	Dimension of space (Chapter 3)
	Distance (Chapter 4, 5)
$\boldsymbol{d}$	Displacement vector
$\mathbf{d}$	Discrete displacement vector
D	Diameter
$\boldsymbol{D}$	Rate-of-strain tensor
e	Error
$\boldsymbol{e}$	Cartesian basis vector
E	Young's modulus
$\boldsymbol{E}$	Green-Lagrange strain tensor
f	Generic function
$\boldsymbol{f}$	External force vector
$\mathbf{f}$	Discrete external force vector
F	External force
$\boldsymbol{F}$	Deformation gradient (Chapter 3)
	Distance function (Chapter 4)
$\hat{\boldsymbol{F}}$	$:= J^{1/3}\boldsymbol{I}$, dilatational (i.e, volume-changing or volumetric) part of the deformation gradient $\boldsymbol{F}$
$\bar{\boldsymbol{F}}$	$:= J^{-1/3}\boldsymbol{F}$, distortional (i.e., volume-preserving or isochoric) part of the deformation gradient $\boldsymbol{F}$
g	Gravitational acceleration
$\mathbf{g}$	Nonlinear field equation
G	Weak form (Chapter 3)
	Shear modulus (Chapter 6)
h	Height
H	Height
$\boldsymbol{H}$	Displacement gradient
$\mathrm{I}_{\boldsymbol{T}}, \mathrm{II}_{\boldsymbol{T}}, \mathrm{III}_{\boldsymbol{T}}$	Invariants of the second-order tensor $\boldsymbol{T}$
$\mathcal{I}$	Interpolation scheme
$\boldsymbol{I}$	Identity mapping (Chapter 2)
	Second-order identity tensor (Chapter 3)
$\boldsymbol{\mathcal{I}}$	Fourth-order identity tensor
$\mathbf{I}$	Identity matrix
j	Time step
J	$:= \det \boldsymbol{F}$, Jacobian determinant (Chapter 3)
	Advance ratio (Chapter 6)
$\boldsymbol{J}$	Jacobian matrix
$\mathbf{J}$	Discrete Jacobian matrix
k	Spring constant (Section 4.3)
	Iteration (Chapter 4 (except Section 4.3), 5, 6)

	Wave number (Chapter 7)
k_t	Thrust coefficient
k_q	Torque coefficient
K	Bulk modulus
$\mathbf{K}$	Tangent stiffness matrix
$\mathbf{K}_\mathrm{m}$	Material stiffness matrix
$\mathbf{K}_\mathrm{s}$	Initial stress matrix
ℓ	Tree level (Chapter 4)
	Length (Chapter 6)
$\boldsymbol{l}$	Angular momentum
L	Length
m	Number of finite volumes (Section 2.1)
	Number of boundary elements (Section 2.2)
	Number of finite elements (Chapter 3)
	Number of subdomains (Section 4.1, 4.2, Section 5.1)
	Number of query points (Section 4.5, 5.4, 5.5)
	Mass (Section 2.1.2, 3.1.2, 4.3, Chapter 6, 7)
m_s	Number of boundary segments (Chapter 2)
	Number of finite element surfaces (Chapter 3)
$\bar{m}$	Specific mass
$\boldsymbol{m}$	External moment
Ma	Mach number
$\mathbf{M}$	Mass matrix
n	Number of nodes (Chapter 3)
	Number of time steps (Section 4.4)
	Number of source points (Section 4.5)
	Rotational speed (Chapter 6)
	Number of mooring lines (Chapter 7)
$\boldsymbol{n}$	Outer unit normal in the spatial or current configuration
N	Shape function (Chapter 3)
	Tree node (Chapter 4)
$\boldsymbol{N}$	Outer unit normal in the reference configuration (Section 3.1.1, 3.1.2)
	Vector of shape functions (Section 3.1.5)
p	Pressure (Chapter 2, 6, 7)
	Polynomial order (Chapter 4, 6)
$\boldsymbol{p}$	Linear momentum (Chapter 3)
	Source point (Chapter 4, 5)
P	Point
$\mathcal{P}$	Predictor scheme (Section 4.2, 4.4)
	Set of source points (Section 4.5, Chapter 5)
$\boldsymbol{P}$	First Piola-Kirchhoff stress tensor
q	Torque (Section 6.14)
	Heat flux (Section 6.17)
$\boldsymbol{q}$	Query point
$\mathcal{Q}$	Set of query points
r	Radius
$\boldsymbol{r}$	Moment arm

$\mathbf{r}$	Residual or out-of-balance vector
R_φ	Joule heating term
$\mathcal{R}$	Set of nearest neighbor or nearest bounding box candidates
$\boldsymbol{R}$	Rotation tensor
s	Shrink factor (Chapter 4)
	Scaling factor (Chapter 6, 7)
S	Sequence
$\tilde{S}$	Transformed sequence
$\mathcal{S}$	Solver
$\boldsymbol{S}$	Second Piola-Kirchhoff stress tensor
t	Time (except Section 6.14)
	Thrust (Section 6.14)
t'	Ramp time
$\boldsymbol{t}$	Traction
T	Final time
u	Generic scalar-valued quantity
$\boldsymbol{u}$	Generic scalar-, vector-, or tensor-valued quantity
$\mathbf{u}$	Generic discrete quantity
U	Dilatational part of the strain energy density U
$\boldsymbol{U}$	Right stretch tensor
v	Volume in the current configuration (Chapter 3)
	Velocity (Chapter 2, 6, 7)
$\boldsymbol{v}$	Velocity
$\hat{\boldsymbol{v}}$	Mesh velocity
$\tilde{\boldsymbol{v}}$	$:= \boldsymbol{v} - \hat{\boldsymbol{v}}$, convective velocity
V	Volume in the reference configuration
$\boldsymbol{V}$	Left stretch tensor
w	Interpolation weight
$\boldsymbol{w}$	Material velocity in the referential configuration
W	Strain energy density (Section 3.1.3)
	Virtual work (Section 3.1.4)
	Width (Chapter 6, 7)
$\bar{W}$	Distortional part of the strain energy density W
$\mathcal{W}$	Set of interpolation weights
$\mathbf{W}$	Interpolation matrix
$\boldsymbol{x}$	Particle in the spatial or current configuration
$\boldsymbol{x}'$	Collocation point
$\boldsymbol{X}$	Particle in the material or reference configuration
$\boldsymbol{y}$	State vector
α	Angle of attack
α_ϑ	Thermal expansion coefficient
α_φ	Linear temperature coefficient
β	Newmark parameter
γ	Diffusion coefficient (Chapter 2)
	Newmark parameter (Chapter 3, 6, 7)

	Peak enhancement factor (Chapter 7)
Γ	$:= \partial\Omega$, boundary of the domain Ω
Δ	Thickness (Chapter 6)
	Draft (Chapter 7)
ε	Tolerance (except Section 6.17)
	Emissivity (Section 6.17)
$\boldsymbol{\varepsilon}$	Linear strain
ζ	Wave elevation
η_0	Efficiency
$\boldsymbol{\eta}$	Test function
$\boldsymbol{\eta}$	Discrete test function vector
θ	Angle between cell normals (Chapter 4)
	Misalignment angle θ (Chapter 7)
ϑ	Temperature
$\boldsymbol{\Theta}$	Inertia tensor in the inertial frame
$\hat{\boldsymbol{\Theta}}$	Inertia tensor in the body frame
λ_ϑ	Thermal conductivity
λ_φ	Electrical conductivity
$\boldsymbol{\lambda}$	Weighting vector
μ	Dynamic viscosity (Section 2.1, Chapter 6)
	Doublet strength (Section 2.2)
ν	Kinematic viscosity
$\xi, \boldsymbol{\xi}$	Coordinate in the parameter space of a finite element
ρ	Density
ρ_∞	Spectral radius
$\boldsymbol{\rho}$	Quaternion
σ	Source strength
σ_{sb}	Stefan-Boltzmann constant
σ_{v}	von Mises stress
$\boldsymbol{\sigma}$	Cauchy stress
ϕ	Generic scalar quantity (Chapter 2)
	Electric potential (Chapter 6)
	Phase angle (Chapter 7)
$\boldsymbol{\varphi}$	$:= \boldsymbol{\Phi} \circ \boldsymbol{\Psi}^{-1}$, bijective transformation from the material or reference configuration $\Omega_{\boldsymbol{X}}$ to the spatial or current configuration $\Omega_{\boldsymbol{x}}$ (Chapter 2, 3)
	Vector storing rotations about x, y, and z (Chapter 4)
Φ	Generic extensive property (Section 2.1)
	Velocity potential (Section 2.2)
$\boldsymbol{\Phi}$	Bijective transformation from the referential configuration $\Omega_{\boldsymbol{\chi}}$ to the spatial configuration $\Omega_{\boldsymbol{x}}$
$\boldsymbol{\chi}$	Particle in the reference configuration
$\boldsymbol{\Psi}$	Bijective transformation from the referential to the material configuration
ω	Relaxation factor (Chapter 4, 6)
	Angular frequency (Chapter 6, 7)
$\boldsymbol{\omega}$	Angular velocity pseudo-vector
Ω	$\subset \mathbb{R}^d$, domain in d-dimensional space $\mathbb{R}^d$

$\partial\Omega$	$\subset \mathbb{R}^{d-1}$, boundary of the domain $\Omega \subset \mathbb{R}^d$ in d-dimensional space $\mathbb{R}^d$

Abstract

A broad range of engineering applications are governed by coupled multifield phenomena. Due to their highly nonlinear nature, fluid-structure interaction problems belong to the most challenging problems in this area. Prominent examples for such problems can, in particular, be found in the maritime industry. In this thesis, emphasis is placed on the numerical investigation of the fluid-structure interaction of a floating offshore wind turbine and of the landing maneuver of a crew transfer vessel to an offshore wind turbine. Due to the ever increasing computational resources, even such highly complex problems have become amenable to a numerical analysis, which helps to provide a deeper insight into the governing physical processes, to reduce the number of expensive experiments, to increase the confidence in the final product, and, last but not least, to reduce costs by shortening the product development cycle.

In the present work, a partitioned solution approach is followed in order to split a coupled problem into separate subproblems, which are coupled by iteratively exchanging the relevant field quantities within a time increment. This procedure enables the use of different spatial and temporal discretization schemes in each of the subdomains. Existing specialized and efficient solvers can then be reused to solve the subproblems – which significantly enhances modularity, software reusability, and also performance. However, these advantages come at the expense of reduced stability of the solution process. Appropriate measures must hence be taken to circumvent stability problems and to accelerate the convergence of the partitioned solution procedure. Therefore, different predictors are proposed so as to provide a reasonable initial guess for the solution in the current time increment and to help to reduce the number of implicit iterations. Regarding the transfer of the relevant field quantities between possibly non-conforming discretizations, several mesh-independent and mesh-dependent interpolation schemes are presented and assessed with respect to accuracy and computational efficiency. Moreover, efficient convergence acceleration schemes, which are suitable to stabilize and accelerate the coupling procedure, are discussed in detail.

In order to simplify the computer implementation of customized coupling strategies for various kinds of multifield problems, the C++ software library *comana* is presented. It offers a vast range of modular and well-tested algorithmic building blocks, which can easily be combined to create a coupling algorithm tailored to the specific problem under consideration. Based on a master/slave architecture, *comana* allows the user to select from plenty of solvers for different physical phenomena and to simply exchange them in a black-box manner. Shared- and distributed-memory parallelized solvers can be integrated into a coupled computation without difficulty rendering even large computations possible. Preparing a solver for a coupled simulation only requires an adapter module and very little modifications in the solver code. Adapters for various solvers for thermodynamics, fluid and structural dynamics are readily provided; adapters not yet available can be implemented with little effort.

In the last part of this work, the software library is verified by means of numerous benchmark problems – and it is also applied to several advanced applications from the maritime industry. Exploiting the full versatility of the software library *comana*, it is demonstrated that the partitioned solution approach is well suited to solve even highly complex and strongly coupled problems efficiently. Particular focus is placed on the fluid-structure interaction of a floating offshore wind turbine and on the landing maneuver of a service ship to an offshore wind turbine, as specific applications from the maritime industry.

1 Introduction

Coupled multifield problems in general and fluid-structure interaction (FSI) problems in particular occur in a vast range of technical applications. Prominent examples are airfoil flutter, the wind-induced vibration of bridges, or the blood flow in arterial vessels. Such kind of problems can be categorized into weakly- and strongly-coupled problems. In the case of weakly-coupled problems, one of the subproblems affects the other subproblem, but not the other way round. For strongly-coupled problems, the situation is different: The subproblems interact with each other, and changes in one of the fields have influence on the respective other. In the context of FSI, problems are weakly coupled if the fluid flow induces a structural deformation that is, however, considered small enough to neglect the feedback on the surrounding flow. Strongly-coupled problems are much more complex, as the structural deformation is considered to have a notable impact on the behavior of the fluid flow and vice versa. In the analysis of the fluid problem, the deflection of the mechanical structure therefore needs to be taken into account.

In the maritime sector, a majority of problems is actually of coupled nature. Interactions between the ship hull and the waves (such as springing, slamming, or whipping) or vibrations in marine propulsion systems are well-known examples for strongly-coupled FSI problems from this area. In this thesis, the FSI of a floating offshore wind turbine and the landing maneuver of a crew transfer vessel to an offshore wind turbine will be of particular interest – as two interesting but also challenging practical applications. In the following, these highly complex, strongly-coupled FSI problems will be briefly introduced, serving to motivate the methodology for the analysis of FSI problems outlined in this work.

1.1 Motivation

Among several other countries in the world, the German government aims to reduce carbon dioxide emissions and to intensify the use of renewable, "green" energies to achieve the ambitious climate targets. Offshore wind energy is considered one of the primary resources to provide energy in an environmentally friendly and sustainable way. In recent years, floating offshore wind turbines (OWTs) have become an interesting alternative to fixed-foundation wind turbines, which are limited to on- or near-shore areas. Far away from the coast, the higher water depths render fixed-foundation platforms technically and economically unfeasible. In contrast, floating platforms can be assembled onshore and, afterwards, be installed offshore quite easily. Despite the higher cost for feeding the generated power into the land-based electricity grid, the stronger and steadier wind renders floating OWTs economically attractive. In order to increase the efficiency of the turbine and to ensure the structural integrity of the entire platform, it is vital to take the FSI into account. Neglecting the coupling effects can lead to a serious underestimation of the acting forces – or to an oversizing, which would impair the competitiveness of the concept.

As a second major problem, we consider the landing maneuver of a crew transfer vessel

to an OWT. Service ships are the preferred means of transportation to bring the servicing staff from the shore to the OWT to carry out maintenance or repair work. To allow the servicing staff to disembark safely, the vessel is equipped with a fender at its bow which is pressed towards the boat landing. In doing so, a vertical friction force is created, which counteracts the hydrodynamic forces acting on the ship hull and keeps the bow in position. Obviously, this process is particularly safety-critical, as a sudden movement of the ship's bow during the disembarkation could be a great danger for the servicing staff. A detailed analysis of the landing maneuver is therefore mandatory to study the forces acting on the ship and to judge under which sea conditions a safe transfer can still be ensured. Moreover, based on the findings from the analysis, it is possible to develop innovative crew transfer concepts that help to increase the safety of the service personnel and to widen the weather window for servicing, thus increasing the availability of the OWT.

Because of their high nonlinearity and complexity, such problems can seldom be solved analytically, and experimental measurements often turn out to be prohibitively expensive or impossible. Numerical simulations have therefore evolved as an important means to analyze all kinds of physical processes. Due to the ever rising availability of computational processing power, numerical simulations of even very complex phenomena have become feasible and increasingly attractive – not only for research but also for industrial applications. In recent years, such simulations have developed a great potential to accelerate the product development process by reducing the necessity to conduct expensive experiments, contributing to confidence in the final product.

1.2 State of the Art

Following the literature overview in [163, p. 1 sq.], first numerical investigations of FSI problems were carried out in the middle of the 1990s, see [10, 108, 132, 179, 28, 150, 143, 169], for instance. Since then, FSI problems in particular, but also other coupled multifield problems, have attracted much attention from researchers and the industry alike. One of the most prominent applications involving the interaction of a fluid and an elastic structure are aeroelastic problems, which, for example, appear in connection with the operation of aircraft [47], wind turbines [14, 13, 72], but also immovable light-weight structures such as tents [59, 182]. Another interesting application are parachutes [150, 151, 140] or inflatable structures like airbags [124]. Maritime applications are certainly another important field for FSI problems. Well-known examples are the interaction of hydrofoils [105] or propellers [118, 101] with the surrounding water flow, but also the interaction of the wind flow and the sails of sailing boats [127]. In view of cardiovascular diseases as one of the major causes of death in the industrial countries, biomedical applications such as the simulation of the human heart [90, 133] or the investigation of the blood flow in arterial vessels [159, 5, 135, 134] have come into focus. In recent years, FSI problems have been enhanced by other physical phenomena such as heat transfer [19, 114], acoustics [104, 141, 163], or coupling to control systems [145].

For the numerical analysis of coupled multifield problems, essentially two main strategies have emerged. The *monolithic* approach treats the entire physical system at once, exhibiting good stability characteristics if an implicit time stepping procedure is used. However, monolithic schemes lead to unsymmetric system matrices due to the presence of cross-derivatives stemming from the consistent linearization of each field's unknowns

with respect to the unknowns of the other fields. In addition, the system matrices can become very large because the unknowns of the entire physical problem enter the system of equations. Both these aspects may lead to systems of equations that are expensive to solve. Even more importantly, the monolithic approach tends to be rather inflexible as tailored solvers are required to solve a particular coupled problem. Existing specialized and fast black-box solvers can hence *not* be reused.

In contrast, the *partitioned* approach deals with each of the subproblems separately and realizes the coupling between the subproblems by iteratively exchanging the relevant field quantities within each time increment. This concept enables the use of different spatial and temporal discretization schemes for each of the subproblems. In addition, existing customized black-box solvers can be employed for the solution of the subproblems, which boosts software reusability, performance, and, above all, flexibility. However, the partitioned solution approach also poses some challenges. Partitioned schemes may converge slowly or even become unstable. Yet, several actions can be taken to circumvent these problems. For instance, the implicit coupling algorithm can be enhanced by a predictor that computes a reasonable initial solution for the current time increment based on the solutions from the previous time steps. No less important is the use of effective convergence acceleration schemes, which considerably reduce the number of required implicit iterations within a time increment. With these strategies, the partitioned approach becomes a very flexible but also effective tool for the solution of arbitrary coupled multifield problems.

1.3 Purpose and Scope of this Thesis

In the scope of this thesis, a generic partitioned solution procedure for the numerical analysis of weakly- or strongly-coupled multifield problems is developed. It is shown that the proposed framework can be easily applied to different kinds of problems without the need for major modifications. In addition, it already covers the possibility of using different time increments in each of the subproblems, adaptive time stepping, and the selection of different predictor, interpolation, and convergence acceleration schemes.

When it comes to the computer implementation of the partitioned approach, a dedicated software library providing the necessary algorithmic building blocks can be of great help to organize the data transfer between the subproblem solvers, and also to select and tune a coupling strategy. To this end, several software packages have already been developed, see [88, 165, 148, 67, 146, 26], for instance. Yet, these packages are either closed source and hence not extensible, do not offer much flexibility to tune the coupling scheme, or are focused on a specific application only. To overcome the limitations of existing software in this respect, the C++ library *comana* [96] is presented. It is based on a modular architecture and strongly focused towards high flexibility. Its algorithmic building blocks are well-tested and can be quickly exchanged as needed to apply the most suitable coupling strategy for the particular problem under consideration. Due to the fact that the subproblem solvers are addressed in a black-box manner, they can be quickly exchanged to employ an efficient numerical method for each of the subproblems. Thanks to a master/slave communication concept, shared- or distributed-memory parallelized solvers can be easily integrated into a partitioned solution strategy as well. Enabling a solver for a coupled simulation requires only a minimalistic interface to access the solver's database, which reduces the necessary modifications in the solver code to a minimum. Interfaces for

solvers from different disciplines – e.g., thermodynamics, fluid dynamics, rigid body and structural dynamics – are readily provided along with *comana*. Interfaces for solvers not yet supported can be implemented with little effort.

In order to demonstrate the versatility of the proposed generic partitioned solution procedure and its implementation in *comana*, several benchmark examples are presented, primarily focusing on FSI, but also on other multifield problems such as electro-thermo-mechanically coupled problems. Based on these examples, it is shown that – in contrast to the majority of available software tools – *comana* is not limited to a particular kind of multifield problem, but allows to apply the partitioned solution approach to arbitrary multifield problems. Exploiting its great flexibility, *comana* is used to treat problems involving single- and multiphase flows, large structural deformations and rotations, composite materials, plasticity, and contact. Different spatial discretizations such as finite element, finite volume, or boundary element discretizations are coupled to each other. Choosing different time step sizes for each of the subfields or adapting the time step size during the simulation helps to reduce the computational cost. Finally, the FSI of a floating OWT and the berthing maneuver of a crew transfer vessel to an OWT are simulated numerically using a partitioned approach, which provides interesting insight into the governing physical phenomena of these sophisticated problems from the maritime industry.

1.4 Outline of this Thesis

In Chapter 2 and 3, we first review the partial differential equations governing the fluid and the structural subproblem and summarize the most prevalent methods for their numerical treatment. Following that, we outline the integration of the subproblems into the partitioned solution approach and propose a generic framework suited not only for the analysis of FSI but also other multifield problems in Chapter 4. Here, we present several predictor schemes to provide an initial guess close to the solution in the current time increment before entering the implicit iteration. For the data transfer between the possibly non-matching discretizations of the subproblems, different mesh-independent and mesh-dependent interpolation schemes are discussed and compared with respect to their accuracy and computational efficiency. Furthermore, convergence acceleration schemes are recapitulated, adapted to the generic partitioned solution procedure, and eventually assessed with regard to their ability to reduce the number of implicit iterations. Subsequently, in Chapter 5, we sketch the computer implementation of the software library *comana*, which is then used to solve various benchmark problems in Chapter 6, before being applied to the simulation of the FSI of a floating OWT and the berthing maneuver of a service ship to an OWT in Chapter 7. Last but not least, Chapter 8 summarizes the main achievements of this work and gives an outlook on future research topics.

2 Fluid Problems

In this chapter, we discuss the governing fluid equations and present two different numerical methods for their solution. First of all, we derive the Navier-Stokes equations in a framework suitable for use in an FSI analysis. In particular, we focus on incompressible flows, which, especially with a view to maritime applications, are of notable interest to engineers. For the numerical solution of the Navier-Stokes equations, we briefly revisit the finite volume method (FVM), which, alongside finite difference and finite element methods, has emerged as one of the most widely-used numerical methods in computational fluid dynamics. Since the Navier-Stokes equations are expensive to solve numerically, we also discuss the potential flow equations, which are based on imposing the additional restraints of the flow being inviscid and irrotational. In many problems in the maritime sector, this assumption poses an adequate approximation of reality. Having derived the equations governing potential flow, we introduce the boundary element method (BEM) as a fast and efficient method for the numerical solution of potential problems. In contrast to the FVM, where the whole fluid domain needs to be discretized, it suffices to discretize the fluid boundary in the BEM, which leads to the significant boost in performance.

2.1 Incompressible Navier-Stokes Equations

In order to describe the motion of a continuous fluid in space, we usually employ an Eulerian description, which assumes the computational mesh to be fixed in space with the continuum moving relative to it, see Figure 2.1. A Lagrangian description, in contrast, follows a material particle as it moves through space. The latter approach is commonly used in structural mechanics, see also Chapter 3 and in particular Section 3.1. In the context of FSI, it is necessary to overcome the limitation of a fixed computational domain for the description of the fluid problem, and to allow the computational domain to deform and follow the structural deformation. Therefore, we introduce a formulation that combines the advantages of the Eulerian and the Lagrangian description, known as the *Arbitrary Lagrangian-Eulerian (ALE)* description [39, p. 2]. The ALE formulation emerged in the mid of the 1960s for the use in finite difference or finite volume methods, cf., among others, [121, 53, 162, 69]. Since then, the method has been considerably enhanced and, in particular, extended to finite element discretizations [38, 17, 16, 74].

2.1.1 Kinematics

For the derivation of the governing conservation equations in the ALE context, we primarily follow [39], but also [93, 172, 52, 102]. To begin with, let us introduce three different configurations as depicted in Figure 2.2, namely the referential configuration $\Omega_{\boldsymbol{\chi}}$, the material configuration $\Omega_{\boldsymbol{X}}$ following the particle motion in space, and the spatial configuration $\Omega_{\boldsymbol{x}}$, which corresponds to the Eulerian description. By means of the bijective transfor-

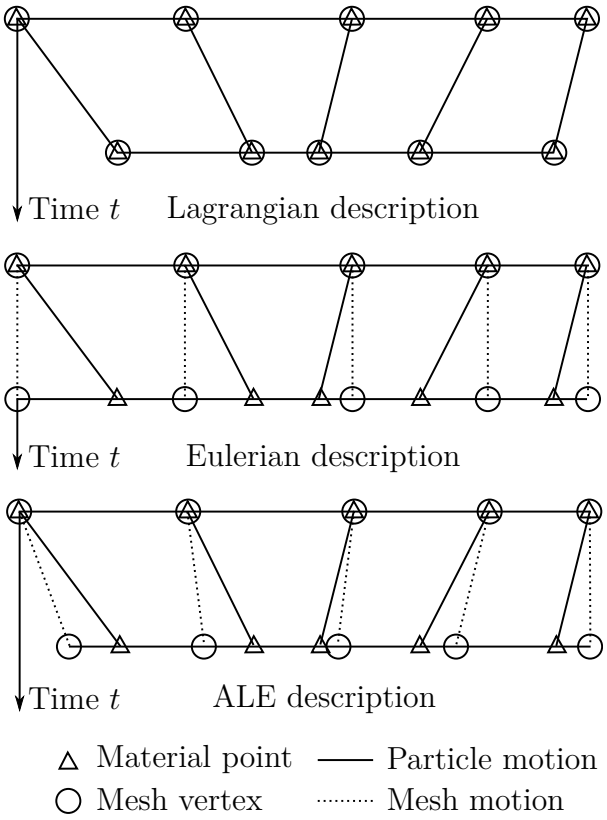


Figure 2.1: Mesh and particle motion in the Eulerian, Lagrangian, and ALE description [39, p. 2].

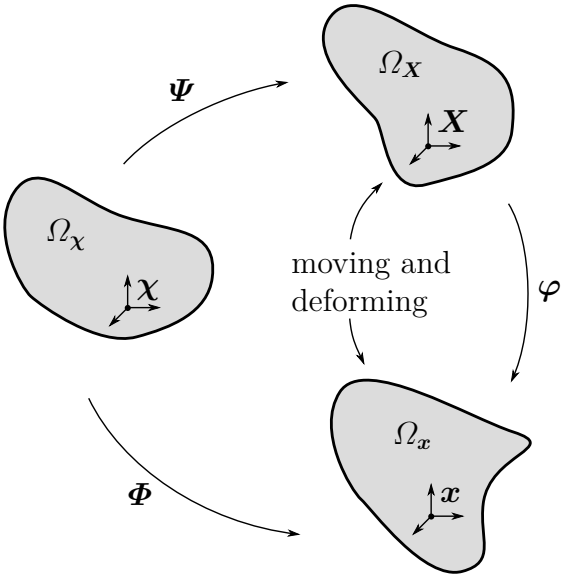


Figure 2.2: Referential configuration $\Omega_{\boldsymbol{\chi}}$, material configuration $\Omega_{\boldsymbol{X}}$, and spatial configuration $\Omega_{\boldsymbol{x}}$ in the ALE description of motion [39, p. 4].

mations $\boldsymbol{\Psi}$ and $\boldsymbol{\Phi}$, the referential configuration $\Omega_{\boldsymbol{\chi}}$ is mapped to the material and spatial configurations $\Omega_{\boldsymbol{X}}$ and $\Omega_{\boldsymbol{x}}$, respectively. These two transformations also serve to describe the mapping from the material to the spatial configuration by introducing the composition $\boldsymbol{\varphi} = \boldsymbol{\Phi} \circ \boldsymbol{\Psi}^{-1}$, which clearly indicates that the three mappings are not independent of each other. Let us formally define the transformation

$$\begin{aligned} \boldsymbol{\Phi} : \Omega_{\boldsymbol{\chi}} \times [t_0, T) &\to \Omega_{\boldsymbol{x}} \times [t_0, T) \ , \\ (\boldsymbol{\chi}, t) &\mapsto \boldsymbol{\Phi}(\boldsymbol{\chi}, t) = (\boldsymbol{x}, t) \ , \end{aligned} \tag{2.1}$$

where t denotes the time, while t_0 and T mark the beginning and the end of the time interval under consideration. The gradient of $\boldsymbol{\Phi}$ with respect to $(\boldsymbol{\chi}, t)$,

$$\frac{\partial \boldsymbol{\Phi}}{\partial(\boldsymbol{\chi}, t)} = \begin{pmatrix} \dfrac{\partial \boldsymbol{x}}{\partial \boldsymbol{\chi}} & \left.\dfrac{\partial \boldsymbol{x}}{\partial t}\right|_{\boldsymbol{\chi}} \\ \mathbf{0}^{\mathrm{T}} & 1 \end{pmatrix} \ , \tag{2.2}$$

involves the mesh velocity

$$\hat{\boldsymbol{v}}(\boldsymbol{\chi}, t) = \left.\frac{\partial \boldsymbol{x}}{\partial t}\right|_{\boldsymbol{\chi}} \ , \tag{2.3}$$

where the notation $|_{\boldsymbol{\chi}}$ implies that the referential coordinate $\boldsymbol{\chi}$ is held fixed. By $\mathbf{0}^{\mathrm{T}}$, we denote the row-vector of all zeros the length of which matches the number of columns in the spatial derivative $\partial \boldsymbol{x} / \partial \boldsymbol{\chi}$.

Regarding $\boldsymbol{\Psi}$, it makes sense to consider only its inverse $\boldsymbol{\Psi}^{-1}$, which will be useful to find an expression for $\boldsymbol{\varphi} = \boldsymbol{\Phi} \circ \boldsymbol{\Psi}^{-1}$. Let us now define the transformation

$$\begin{aligned} \boldsymbol{\Psi}^{-1} : \Omega_{\boldsymbol{X}} \times [t_0, T) &\to \Omega_{\boldsymbol{\chi}} \times [t_0, T) \ , \\ (\boldsymbol{X}, t) &\mapsto \boldsymbol{\Psi}^{-1}(\boldsymbol{X}, t) = (\boldsymbol{\chi}, t) \end{aligned} \tag{2.4}$$

from the material to the referential configuration. The gradient of $\boldsymbol{\Psi}^{-1}$ amounts to

$$\frac{\partial \boldsymbol{\Psi}^{-1}}{\partial(\boldsymbol{X}, t)} = \begin{pmatrix} \dfrac{\partial \boldsymbol{\chi}}{\partial \boldsymbol{X}} & \boldsymbol{w} \\ \mathbf{0}^{\mathrm{T}} & 1 \end{pmatrix} \ , \tag{2.5}$$

where we introduced the velocity

$$\boldsymbol{w} = \left.\frac{\partial \boldsymbol{\chi}}{\partial t}\right|_{\boldsymbol{X}} \ . \tag{2.6}$$

The expression (2.6) can be construed as the particle velocity in the referential configuration. Having derived the gradients of the mappings $\boldsymbol{\Phi}$ and $\boldsymbol{\Psi}^{-1}$, we have all the prerequisites to differentiate $\boldsymbol{\varphi} = \boldsymbol{\Phi} \circ \boldsymbol{\Psi}^{-1}$ and obtain the relations between the material velocity $\boldsymbol{v}$, the mesh velocity $\hat{\boldsymbol{v}}$, and the particle velocity $\boldsymbol{w}$ in the referential configuration:

$$\frac{\partial \boldsymbol{\varphi}}{\partial(\boldsymbol{X}, t)}(\boldsymbol{X}, t) = \frac{\partial \boldsymbol{\Phi}}{\partial(\boldsymbol{\chi}, t)}\left(\boldsymbol{\Psi}^{-1}(\boldsymbol{X}, t)\right) \frac{\partial \boldsymbol{\Psi}^{-1}}{\partial(\boldsymbol{X}, t)}(\boldsymbol{X}, t) = \frac{\partial \boldsymbol{\Phi}}{\partial(\boldsymbol{\chi}, t)}(\boldsymbol{\chi}, t) \frac{\partial \boldsymbol{\Psi}^{-1}}{\partial(\boldsymbol{X}, t)}(\boldsymbol{X}, t) \ . \tag{2.7}$$

In block matrix form, this reads

$$\begin{pmatrix} \dfrac{\partial \boldsymbol{x}}{\partial \boldsymbol{X}} & \boldsymbol{v} \\ \boldsymbol{0}^{\mathrm{T}} & 1 \end{pmatrix} = \begin{pmatrix} \dfrac{\partial \boldsymbol{x}}{\partial \boldsymbol{\chi}} & \hat{\boldsymbol{v}} \\ \boldsymbol{0}^{\mathrm{T}} & 1 \end{pmatrix} \begin{pmatrix} \dfrac{\partial \boldsymbol{\chi}}{\partial \boldsymbol{X}} & \boldsymbol{w} \\ \boldsymbol{0}^{\mathrm{T}} & 1 \end{pmatrix} . \tag{2.8}$$

From this, we extract the relation

$$\boldsymbol{v} = \hat{\boldsymbol{v}} + \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\chi}} \boldsymbol{w} . \tag{2.9}$$

After rearranging, this results in an expression for the convective velocity

$$\tilde{\boldsymbol{v}} := \boldsymbol{v} - \hat{\boldsymbol{v}} = \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\chi}} \boldsymbol{w} , \tag{2.10}$$

which represents the relative velocity between the material particles and the mesh. Note that $\tilde{\boldsymbol{v}}$ is distinct from $\boldsymbol{w}$; only if the mesh motion is purely translational, these velocities become identical.

It is also worth noting that the ALE description of motion covers both the Lagrangian as well as the Eulerian formulation. For a Lagrangian description, we choose $\boldsymbol{\Psi} = \boldsymbol{I}$, where $\boldsymbol{I}$ is the identity mapping, which implies $\boldsymbol{X} \equiv \boldsymbol{\chi}$. Consequently, the material and the mesh velocity coincide, meaning that $\boldsymbol{v} = \hat{\boldsymbol{v}}$, and the convective velocity $\tilde{\boldsymbol{v}}$ vanishes. An Eulerian description is recovered if $\boldsymbol{\Phi} = \boldsymbol{I}$ and $\boldsymbol{x} \equiv \boldsymbol{\chi}$. Then, the mesh velocity $\hat{\boldsymbol{v}}$ vanishes, and the convective velocity $\tilde{\boldsymbol{v}}$ becomes identical to the material velocity $\boldsymbol{v}$.

For the derivation of the conservation equations later on, it is required to establish a relation between the time derivative of a generic scalar quantity $\phi_{\boldsymbol{\chi}}$, $\phi_{\boldsymbol{X}}$, and ϕ – defined in the referential, material, and spatial frame, respectively. Relating the material and spatial description of a quantity by means of the transformation $\boldsymbol{\varphi}$, we obtain

$$\phi_{\boldsymbol{X}}(\boldsymbol{X}, t) = \phi\left(\boldsymbol{\varphi}(\boldsymbol{X}, t), t\right) \tag{2.11}$$

and, after differentiation, the gradient

$$\frac{\partial \phi_{\boldsymbol{X}}}{\partial (\boldsymbol{X}, t)}(\boldsymbol{X}, t) = \frac{\partial \phi}{\partial (\boldsymbol{x}, t)}(\boldsymbol{x}, t) \frac{\partial \boldsymbol{\varphi}}{\partial (\boldsymbol{X}, t)}(\boldsymbol{X}, t) . \tag{2.12}$$

Recast in matrix notation, this reads

$$\begin{pmatrix} \dfrac{\partial \phi_{\boldsymbol{X}}}{\partial \boldsymbol{X}} & \dfrac{\partial \phi_{\boldsymbol{X}}}{\partial t} \end{pmatrix} = \begin{pmatrix} \dfrac{\partial \phi}{\partial \boldsymbol{x}} & \dfrac{\partial \phi}{\partial t} \end{pmatrix} \begin{pmatrix} \dfrac{\partial \boldsymbol{x}}{\partial \boldsymbol{X}} & \boldsymbol{v} \\ \boldsymbol{0}^{\mathrm{T}} & 1 \end{pmatrix} . \tag{2.13}$$

From the second equation, the important relation

$$\frac{\partial \phi_{\boldsymbol{X}}}{\partial t} = \frac{\partial \phi}{\partial t} + \frac{\partial \phi}{\partial \boldsymbol{x}} \cdot \boldsymbol{v} . \tag{2.14}$$

is revealed. In the following, we simplify the notation by introducing the shorthand operators

$$\frac{\mathrm{D} \cdot}{\mathrm{D}t} := \left. \frac{\partial \cdot}{\partial t} \right|_{\boldsymbol{X}} \tag{2.15}$$

for the material time derivative and

$$\frac{\partial \cdot}{\partial t} := \left.\frac{\partial \cdot}{\partial t}\right|_{\boldsymbol{x}} \tag{2.16}$$

for the spatial time derivative, unless indicated otherwise. Using this, Equation (2.14) becomes

$$\frac{\mathrm{D}\phi}{\mathrm{D}t} = \frac{\partial \phi}{\partial t} + \boldsymbol{v} \cdot \operatorname{grad} \phi \, . \tag{2.17}$$

Let us now derive the relation between the material and the referential time derivative. With the inverse transformation $\boldsymbol{\Psi}^{-1}$ introduced in (2.4), the material and the referential representation of a quantity are related by

$$\phi_{\boldsymbol{X}} = \phi_{\boldsymbol{\chi}} \circ \boldsymbol{\Psi}^{-1} \, . \tag{2.18}$$

Differentiation leads us to

$$\frac{\partial \phi_{\boldsymbol{X}}}{\partial(\boldsymbol{X}, t)}(\boldsymbol{X}, t) = \frac{\partial \phi_{\boldsymbol{\chi}}}{\partial(\boldsymbol{\chi}, t)}(\boldsymbol{\chi}, t) \frac{\partial \boldsymbol{\Psi}^{-1}}{\partial(\boldsymbol{X}, t)}(\boldsymbol{X}, t) \, , \tag{2.19}$$

or, in matrix notation,

$$\begin{pmatrix} \dfrac{\partial \phi_{\boldsymbol{X}}}{\partial \boldsymbol{X}} & \dfrac{\partial \phi_{\boldsymbol{X}}}{\partial t} \end{pmatrix} = \begin{pmatrix} \dfrac{\partial \phi_{\boldsymbol{\chi}}}{\partial \boldsymbol{\chi}} & \dfrac{\partial \phi_{\boldsymbol{\chi}}}{\partial t} \end{pmatrix} \begin{pmatrix} \dfrac{\partial \boldsymbol{\chi}}{\partial \boldsymbol{X}} & \boldsymbol{w} \\ \boldsymbol{0}^{\mathrm{T}} & 1 \end{pmatrix} \, . \tag{2.20}$$

From the second block equation, we notice that

$$\frac{\partial \phi_{\boldsymbol{X}}}{\partial t} = \frac{\partial \phi_{\boldsymbol{\chi}}}{\partial t} + \frac{\partial \phi_{\boldsymbol{\chi}}}{\partial \boldsymbol{\chi}} \cdot \boldsymbol{w} \, , \tag{2.21}$$

or, taking (2.10) into account,

$$\frac{\partial \phi_{\boldsymbol{X}}}{\partial t} = \frac{\partial \phi_{\boldsymbol{\chi}}}{\partial t} + \frac{\partial \phi}{\partial \boldsymbol{x}} \cdot \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\chi}} \boldsymbol{w} = \frac{\partial \phi_{\boldsymbol{\chi}}}{\partial t} + \frac{\partial \phi}{\partial \boldsymbol{x}} \cdot \tilde{\boldsymbol{v}} \, . \tag{2.22}$$

Leaving the subscripts apart and employing the shorthand notation proposed in (2.15) and (2.16), this reads

$$\frac{\mathrm{D}\phi}{\mathrm{D}t} = \left.\frac{\partial \phi}{\partial t}\right|_{\boldsymbol{\chi}} + \frac{\partial \phi}{\partial \boldsymbol{x}} \cdot \tilde{\boldsymbol{v}} = \left.\frac{\partial \phi}{\partial t}\right|_{\boldsymbol{\chi}} + \tilde{\boldsymbol{v}} \cdot \operatorname{grad} \phi \, . \tag{2.23}$$

Combining (2.17) and (2.23) eventually reveals the important relation

$$\left.\frac{\partial \phi}{\partial t}\right|_{\boldsymbol{\chi}} = \frac{\partial \phi}{\partial t} - \hat{\boldsymbol{v}} \cdot \operatorname{grad} \phi \tag{2.24}$$

between the spatial and the referential time derivative.

2.1.2 Conservation Laws

Based on our findings from above, let us proceed to establishing the basic conservation laws, which serve to describe the motion of a fluid in the ALE context. Major parts of the following paragraphs are, apart from [39], adopted from the standard textbooks for fluid dynamics [49, 142, 128, 20, 175, 31]. Considering a generic intensive property $\phi(\boldsymbol{x}, t)$ in the spatial frame, the corresponding extensive property $\Phi(t)$ is obtained by integrating over the moving and time-varying material volume Ω_t:

$$\Phi(t) = \int_{\Omega_t} \rho\phi(\boldsymbol{x}, t) \, \mathrm{d}\Omega \, . \tag{2.25}$$

A change of the extensive property $\Phi(t)$ over time t is expressed by the material time derivative

$$\frac{\mathrm{D}\Phi}{\mathrm{D}t}(t) = \frac{\mathrm{D}}{\mathrm{D}t} \int_{\Omega_t} \rho\phi(\boldsymbol{x}, t) \, \mathrm{d}\Omega \, , \tag{2.26}$$

which is expanded as

$$\frac{\mathrm{D}}{\mathrm{D}t} \int_{\Omega_t} \rho\phi(\boldsymbol{x}, t) \, \mathrm{d}\Omega = \int_{\Omega_t} \frac{\partial \left(\rho\phi(\boldsymbol{x}, t)\right)}{\partial t} \, \mathrm{d}\Omega + \int_{\Gamma_t} \rho\phi(\boldsymbol{x}, t) \boldsymbol{v} \cdot \boldsymbol{n} \, \mathrm{d}\Gamma \, . \tag{2.27}$$

Here, $\Gamma_t := \partial\Omega_t$ is the boundary of the material volume Ω_t, which moves with the material velocity $\boldsymbol{v} = \boldsymbol{v}(\boldsymbol{x}, t)$ and $\boldsymbol{n}$ symbolizes the outer unit surface normal of the surface Γ_t. In essence, this equation states that the variation of the extensive property Φ with time is equal to the sum of the local change of the property within the moving material volume Ω_t and its net flux through the (likewise moving) boundary Γ_t. Note that the validity of this equation is still retained if ϕ is a vector- or tensor-valued quantity. Equation (2.27) represents a generic conservation equation and is known as the *Reynolds transport theorem*. In the ALE framework, Equation (2.27) is applied to an arbitrary volume Ω_t moving with the mesh velocity $\hat{\boldsymbol{v}}$:

$$\left.\frac{\partial}{\partial t}\right|_{\boldsymbol{\chi}} \int_{\Omega_t} \rho\phi(\boldsymbol{x}, t) \, \mathrm{d}\Omega = \int_{\Omega_t} \frac{\partial \left(\rho\phi(\boldsymbol{x}, t)\right)}{\partial t} \, \mathrm{d}\Omega + \int_{\Gamma_t} \rho\phi(\boldsymbol{x}, t) \hat{\boldsymbol{v}} \cdot \boldsymbol{n} \, \mathrm{d}\Gamma \, . \tag{2.28}$$

Note that, in this case, the characterizing velocity is no longer the material velocity $\boldsymbol{v}$ but the mesh velocity $\hat{\boldsymbol{v}}$.

In virtually all flows of engineering interest, the mass within the moving material volume Ω_t is conserved, which is conveyed by the equation

$$\frac{\mathrm{D}m}{\mathrm{D}t} = 0 \, . \tag{2.29}$$

Based on this, substituting $\phi = 1$ in (2.27) leads us to

$$\frac{\mathrm{D}}{\mathrm{D}t} \int_{\Omega_t} \rho \, \mathrm{d}\Omega = \int_{\Omega_t} \frac{\partial\rho}{\partial t} \, \mathrm{d}\Omega + \int_{\Gamma_t} \rho\boldsymbol{v} \cdot \boldsymbol{n} \, \mathrm{d}\Gamma = 0 \, . \tag{2.30}$$

From (2.28), we see (after rearranging) that

$$\int_{\Omega_t} \frac{\partial\rho}{\partial t} \, \mathrm{d}\Omega = \left.\frac{\partial}{\partial t}\right|_{\boldsymbol{\chi}} \int_{\Omega_t} \rho \, \mathrm{d}\Omega - \int_{\Gamma_t} \rho\hat{\boldsymbol{v}} \cdot \boldsymbol{n} \, \mathrm{d}\Gamma \, . \tag{2.31}$$

Plugging (2.31) into (2.30) and recalling the definition of the convective velocity $\tilde{\boldsymbol{v}}$ (2.10) results in the ALE form of the mass conservation or continuity equation

$$\left.\frac{\partial}{\partial t}\right|_{\boldsymbol{\chi}} \int_{\Omega_t} \rho \,\mathrm{d}V + \int_{\Gamma_t} \rho \tilde{\boldsymbol{v}} \cdot \boldsymbol{n} \,\mathrm{d}\Gamma = 0 \,. \tag{2.32}$$

Subsequent to deriving the integral form of the continuity equation, let us proceed with the differential form. Exploiting the identity (2.24), we restate (2.30) as

$$\int_{\Omega_t} \frac{\partial \rho}{\partial t} \,\mathrm{d}\Omega + \int_{\Gamma_t} \rho \boldsymbol{v} \cdot \boldsymbol{n} \,\mathrm{d}\Gamma = \int_{\Omega_t} \left.\frac{\partial \rho}{\partial t}\right|_{\boldsymbol{\chi}} - \hat{\boldsymbol{v}} \cdot \operatorname{grad} \rho \,\mathrm{d}\Omega + \int_{\Gamma_t} \rho \boldsymbol{v} \cdot \boldsymbol{n} \,\mathrm{d}\Gamma = 0 \,. \tag{2.33}$$

Applying Gauss's divergence theorem to the surface integral leaves us with

$$\int_{\Omega_t} \left.\frac{\partial \rho}{\partial t}\right|_{\boldsymbol{\chi}} - \hat{\boldsymbol{v}} \cdot \operatorname{grad} \rho \,\mathrm{d}\Omega + \int_{\Omega_t} \operatorname{div}(\rho \boldsymbol{v}) \,\mathrm{d}\Omega = 0 \,, \tag{2.34}$$

which, keeping in mind that

$$\operatorname{div}(\rho \boldsymbol{v}) = \boldsymbol{v} \cdot \operatorname{grad} \rho + \rho \operatorname{div} \boldsymbol{v} \,, \tag{2.35}$$

is more compactly written as

$$\int_{\Omega_t} \left.\frac{\partial \rho}{\partial t}\right|_{\boldsymbol{\chi}} - \hat{\boldsymbol{v}} \cdot \operatorname{grad} \rho + \boldsymbol{v} \cdot \operatorname{grad} \rho + \rho \operatorname{div} \boldsymbol{v} \,\mathrm{d}\Omega = \int_{\Omega_t} \left.\frac{\partial \rho}{\partial t}\right|_{\boldsymbol{\chi}} + \tilde{\boldsymbol{v}} \cdot \operatorname{grad} \rho + \rho \operatorname{div} \boldsymbol{v} \,\mathrm{d}\Omega = 0 \,. \tag{2.36}$$

Apparently, this equation must also hold for infinitesimal volumes Ω_t, which takes us to the differential form of the continuity equation (2.32):

$$\left.\frac{\partial \rho}{\partial t}\right|_{\boldsymbol{\chi}} + \tilde{\boldsymbol{v}} \cdot \operatorname{grad} \rho + \rho \operatorname{div} \boldsymbol{v} = 0 \,. \tag{2.37}$$

In many technical and most maritime applications, the Mach number Ma of the flow, which relates the maximum fluid velocity $v_{\max}$ to the speed of sound c, satisfies the relation

$$\mathrm{Ma} = \frac{v_{\max}}{c} < 0.3 \,. \tag{2.38}$$

Given this assumption, the flow can be considered as incompressible (or, more precisely, isochoric), which significantly simplifies the description of the fluid behavior. For the material time derivative of the density ρ, it then holds that

$$\frac{\mathrm{D}\rho}{\mathrm{D}t} = \left.\frac{\partial \rho}{\partial t}\right|_{\boldsymbol{\chi}} + \tilde{\boldsymbol{v}} \cdot \operatorname{grad} \rho = 0 \,, \tag{2.39}$$

which simplifies the continuity equation (2.36) to read

$$\int_{\Omega_t} \rho \operatorname{div} \boldsymbol{v} \,\mathrm{d}\Omega = 0 \,. \tag{2.40}$$

Similarly, the differential form (2.37) becomes

$$\operatorname{div} \boldsymbol{v} = 0 \,. \tag{2.41}$$

Newton's second law of motion teaches us that the change in momentum $\mathrm{D}(m\boldsymbol{v})/\mathrm{D}t$ of a material volume Ω_t equals the sum of external forces $\boldsymbol{f}(t)$ acting on the material volume such that

$$\frac{\mathrm{D}\,(m\boldsymbol{v})}{\mathrm{D}t} = \boldsymbol{f}(t)\ . \tag{2.42}$$

Substituting $\phi = \boldsymbol{v}$ and inserting (2.27) into (2.42) produces the momentum equation

$$\int_{\Omega_t} \frac{\partial(\rho\boldsymbol{v})}{\partial t}\,\mathrm{d}\Omega + \int_{\Gamma_t} \rho\boldsymbol{v}\boldsymbol{v}\cdot\boldsymbol{n}\,\mathrm{d}\Gamma = \boldsymbol{f}(t)\ . \tag{2.43}$$

Subdividing the sum of external forces into surface and volume forces, we obtain

$$\int_{\Omega_t} \frac{\partial(\rho\boldsymbol{v})}{\partial t}\,\mathrm{d}\Omega + \int_{\Gamma_t} \rho\boldsymbol{v}\boldsymbol{v}\cdot\boldsymbol{n}\,\mathrm{d}\Gamma = \int_{\Gamma_t} \boldsymbol{\sigma}\boldsymbol{n}\,\mathrm{d}\Gamma + \int_{\Omega_t} \rho\boldsymbol{b}\,\mathrm{d}\Omega\ , \tag{2.44}$$

where $\boldsymbol{\sigma}$ denotes the stress tensor and $\boldsymbol{b}$ represents the volume force per unit mass. Recalling Equation (2.28), we are led to the momentum equation in ALE integral form:

$$\left.\frac{\partial}{\partial t}\right|_{\boldsymbol{\chi}} \int_{\Omega_t} \rho\boldsymbol{v}\,\mathrm{d}\Omega + \int_{\Gamma_t} \rho\boldsymbol{v}\tilde{\boldsymbol{v}}\cdot\boldsymbol{n}\,\mathrm{d}\Gamma = \int_{\Gamma_t} \boldsymbol{\sigma}\boldsymbol{n}\,\mathrm{d}\Gamma + \int_{\Omega_t} \rho\boldsymbol{b}\,\mathrm{d}\Omega\ . \tag{2.45}$$

Let us now reformulate (2.44) in component form:

$$\int_{\Omega_t} \frac{\partial(\rho v_i)}{\partial t}\,\mathrm{d}\Omega + \int_{\Gamma_t} \rho v_i\boldsymbol{v}\cdot\boldsymbol{n}\,\mathrm{d}\Gamma = \int_{\Gamma_t} \sigma_{ij}n_j\,\mathrm{d}\Gamma + \int_{\Omega_t} \rho b_i\,\mathrm{d}\Omega\ . \tag{2.46}$$

Once again drawing on relation (2.24), we realize that (2.46) can also be written as

$$\int_{\Omega_t} \left.\frac{\partial(\rho v_i)}{\partial t}\right|_{\boldsymbol{\chi}} - \hat{\boldsymbol{v}}\cdot\operatorname{grad}(\rho v_i)\,\mathrm{d}\Omega + \int_{\Gamma_t} \rho v_i\boldsymbol{v}\cdot\boldsymbol{n}\,\mathrm{d}\Gamma = \int_{\Gamma_t} \sigma_{ij}n_j\,\mathrm{d}\Gamma + \int_{\Omega_t} \rho b_i\,\mathrm{d}\Omega\ . \tag{2.47}$$

Following this, we apply Gauss's divergence theorem on the surface integral terms, which results in

$$\int_{\Omega_t} \left.\frac{\partial(\rho v_i)}{\partial t}\right|_{\boldsymbol{\chi}} - \hat{\boldsymbol{v}}\cdot\operatorname{grad}(\rho v_i) + \operatorname{div}(\rho v_i\boldsymbol{v})\,\mathrm{d}\Omega = \int_{\Omega_t} \operatorname{div}(\sigma_{ij}\boldsymbol{e}_j) + \rho b_i\,\mathrm{d}\Omega\ , \tag{2.48}$$

where $\boldsymbol{e}_j$ is the Cartesian basis vector in the jth coordinate direction. Taking (2.35) into account, this changes to

$$\int_{\Omega_t} \left.\frac{\partial(\rho v_i)}{\partial t}\right|_{\boldsymbol{\chi}} - \hat{\boldsymbol{v}}\cdot\operatorname{grad}(\rho v_i) + \boldsymbol{v}\cdot\operatorname{grad}(\rho v_i) + \rho v_i \operatorname{div}\boldsymbol{v}\,\mathrm{d}\Omega = \int_{\Omega_t} \operatorname{div}(\sigma_{ij}\boldsymbol{e}_j) + \rho b_i\,\mathrm{d}\Omega\ , \tag{2.49}$$

or, with the definition of the convective velocity $\tilde{\boldsymbol{v}}$ in Equation (2.10),

$$\int_{\Omega_t} \left.\frac{\partial(\rho v_i)}{\partial t}\right|_{\boldsymbol{\chi}} + \tilde{\boldsymbol{v}}\cdot\operatorname{grad}(\rho v_i) + \rho v_i \operatorname{div}\boldsymbol{v}\,\mathrm{d}\Omega = \int_{\Omega_t} \operatorname{div}(\sigma_{ij}\boldsymbol{e}_j) + \rho b_i\,\mathrm{d}\Omega\ . \tag{2.50}$$

Exploiting the identity

$$\operatorname{grad}(\rho v_i) = \rho \operatorname{grad} v_i + v_i \operatorname{grad}\rho\ , \tag{2.51}$$

we notice that

$$\int_{\Omega_t} \left.\frac{\partial(\rho v_i)}{\partial t}\right|_{\boldsymbol{\chi}} + \tilde{\boldsymbol{v}} \cdot (\rho \operatorname{grad} v_i + v_i \operatorname{grad} \rho) + \rho v_i \operatorname{div} \boldsymbol{v} \, \mathrm{d}\Omega = \int_{\Omega_t} \operatorname{div}(\sigma_{ij} \boldsymbol{e}_j) + \rho b_i \, \mathrm{d}\Omega \, . \quad (2.52)$$

Inserting the relation

$$\left.\frac{\partial(\rho v_i)}{\partial t}\right|_{\boldsymbol{\chi}} = v_i \left.\frac{\partial \rho}{\partial t}\right|_{\boldsymbol{\chi}} + \rho \left.\frac{\partial v_i}{\partial t}\right|_{\boldsymbol{\chi}} \quad (2.53)$$

for the referential time derivative in (2.52) and rearranging the terms leads us to

$$\int_{\Omega_t} v_i \left(\left.\frac{\partial \rho}{\partial t}\right|_{\boldsymbol{\chi}} + \tilde{\boldsymbol{v}} \cdot \operatorname{grad} \rho + \rho \operatorname{div} \boldsymbol{v} \right) + \rho \left.\frac{\partial v_i}{\partial t}\right|_{\boldsymbol{\chi}} + \rho \tilde{\boldsymbol{v}} \cdot \operatorname{grad} v_i \, \mathrm{d}\Omega = \int_{\Omega_t} \operatorname{div}(\sigma_{ij} \boldsymbol{e}_j) + \rho b_i \, \mathrm{d}\Omega \, . \quad (2.54)$$

By virtue of the continuity equation (2.37), the term in parentheses on the left-hand side vanishes, and the remainder reads

$$\int_{\Omega_t} \rho \left.\frac{\partial v_i}{\partial t}\right|_{\boldsymbol{\chi}} + \rho \tilde{\boldsymbol{v}} \cdot \operatorname{grad} v_i \, \mathrm{d}\Omega = \int_{\Omega_t} \operatorname{div}(\sigma_{ij} \boldsymbol{e}_j) + \rho b_i \, \mathrm{d}\Omega \, , \quad (2.55)$$

or, again in tensor notation,

$$\int_{\Omega_t} \rho \left(\left.\frac{\partial \boldsymbol{v}}{\partial t}\right|_{\boldsymbol{\chi}} + \tilde{\boldsymbol{v}} \cdot \operatorname{grad} \boldsymbol{v} \right) \mathrm{d}\Omega = \int_{\Omega_t} \operatorname{div} \boldsymbol{\sigma} + \rho \boldsymbol{b} \, \mathrm{d}\Omega \, . \quad (2.56)$$

Evidently, this equation must also be fulfilled if the volume Ω_t approaches zero, which gives us the differential form of (2.56):

$$\rho \left(\left.\frac{\partial \boldsymbol{v}}{\partial t}\right|_{\boldsymbol{\chi}} + \tilde{\boldsymbol{v}} \cdot \operatorname{grad} \boldsymbol{v} \right) = \operatorname{div} \boldsymbol{\sigma} + \rho \boldsymbol{b} \, . \quad (2.57)$$

For Newtonian fluids, which are exclusively considered in this work, the stress tensor $\boldsymbol{\sigma}$ reads

$$\boldsymbol{\sigma} = - \left(p + \frac{2}{3} \mu \operatorname{div} \boldsymbol{v} \right) \boldsymbol{I} + 2\mu \boldsymbol{D} \, , \quad (2.58)$$

where p is the pressure, μ is the dynamic viscosity, and $\boldsymbol{D}$ is the rate-of-strain tensor, for which

$$\boldsymbol{D} = \frac{1}{2} \left(\operatorname{grad} \boldsymbol{v} + (\operatorname{grad} \boldsymbol{v})^{\mathrm{T}} \right) \quad (2.59)$$

holds. Respecting the continuity equation for isochoric flow (2.41), the second term in parentheses in Equation (2.58) vanishes, and the stress tensor reduces to

$$\boldsymbol{\sigma} = -p \boldsymbol{I} + 2\mu \boldsymbol{D} \, . \quad (2.60)$$

Substituting (2.59) into (2.60) and subsequently into (2.56) results in

$$\int_{\Omega_t} \rho \left(\left.\frac{\partial \boldsymbol{v}}{\partial t}\right|_{\boldsymbol{\chi}} + \tilde{\boldsymbol{v}} \cdot \operatorname{grad} \boldsymbol{v} \right) \mathrm{d}\Omega = \int_{\Omega_t} \mu \Delta \boldsymbol{v} - \operatorname{grad} p + \rho \boldsymbol{b} \, \mathrm{d}\Omega \, , \quad (2.61)$$

or, in a form better suited for the discussion later on,

$$\left.\frac{\partial}{\partial t}\right|_{\boldsymbol{\chi}} \int_{\Omega_t} \rho \boldsymbol{v} \,\mathrm{d}\Omega + \int_{\Gamma_t} \rho \boldsymbol{v} \tilde{\boldsymbol{v}} \cdot \boldsymbol{n} \,\mathrm{d}\Gamma = \int_{\Gamma_t} \mu \,\mathrm{grad}\, \boldsymbol{v} \cdot \boldsymbol{n} \,\mathrm{d}\Gamma - \int_{\Gamma_t} p \boldsymbol{n} \,\mathrm{d}\Gamma + \int_{\Omega_t} \rho \boldsymbol{b} \,\mathrm{d}\Omega \,. \tag{2.62}$$

In differential form, Equation (2.61) reads

$$\rho \left(\left.\frac{\partial \boldsymbol{v}}{\partial t}\right|_{\boldsymbol{\chi}} + \tilde{\boldsymbol{v}} \cdot \mathrm{grad}\, \boldsymbol{v} \right) = \mu \Delta \boldsymbol{v} - \mathrm{grad}\, p + \rho \boldsymbol{b} \,. \tag{2.63}$$

The sets of equations (2.40), (2.62) or (2.41), (2.63) represent the ALE form of the incompressible Navier-Stokes equations. The unknowns in these equations are the pressure p and the velocity $\boldsymbol{v}$. In order to constitute a well-posed initial-boundary value problem, the problem needs to be subjected to the Dirichlet boundary condition

$$\boldsymbol{v} = \bar{\boldsymbol{v}} \quad \text{on } \Gamma_{\boldsymbol{v},\boldsymbol{x}} := \partial\Omega_{\boldsymbol{v},\boldsymbol{x}} \tag{2.64}$$

and the Neumann boundary condition

$$\boldsymbol{\sigma}\boldsymbol{n} = \bar{\boldsymbol{t}} \quad \text{on } \Gamma_{\boldsymbol{t},\boldsymbol{x}} := \partial\Omega_{\boldsymbol{t},\boldsymbol{x}} \,; \tag{2.65}$$

where $\bar{\boldsymbol{v}}$ and $\bar{\boldsymbol{t}}$ denote the prescribed velocity and traction, respectively. In addition, an initial velocity $\boldsymbol{v}_0$ needs to be prescribed at $t = t_0$:

$$\boldsymbol{v}(t = t_0) = \boldsymbol{v}_0 \quad \text{in } \Omega_{\boldsymbol{x}} \,. \tag{2.66}$$

2.1.3 Discretization and Numerical Solution

For the numerical solution of the system of coupled nonlinear partial differential equations (2.41), (2.63) subject to the initial conditions (2.66) and the boundary conditions (2.64), (2.65), we employ the FVM, which is well-suited for complex geometries and conservative by construction [49, p. 36]. It has become one of the most popular methods for computational fluid dynamics and is also implemented in many open-source and commercial software packages, see [123, 3], for instance. Other frequently used numerical methods are the finite difference method or the finite element method, which, however, will not be discussed further in this work. The interested reader is referred to the standard literature on these methods, see [49, 136, 187], for instance. The derivation of the numerical schemes in the following paragraphs is closely oriented towards [49, pp. 73–79,136].

In the FVM, we subdivide the domain of interest into m finite control volumes such that

$$\Omega \approx \Omega_h = \bigcup_{\ell=1}^{m} \Omega_\ell \,. \tag{2.67}$$

Here, Ω_h denotes the discrete approximation of the continuous domain Ω. In each of the finite control volumes Ω_ℓ, the conservation equations (2.32) and (2.45) hold. A representative two-dimensional Cartesian finite volume (FV) discretization is depicted in Figure 2.3. Following the idea of the cell-centered FVM, the unknowns are located at the centers of the control volumes. In order to illustrate the discretization of the individual terms in the integral equations, we introduce the so-called *compass notation* [128, 49] and associate the

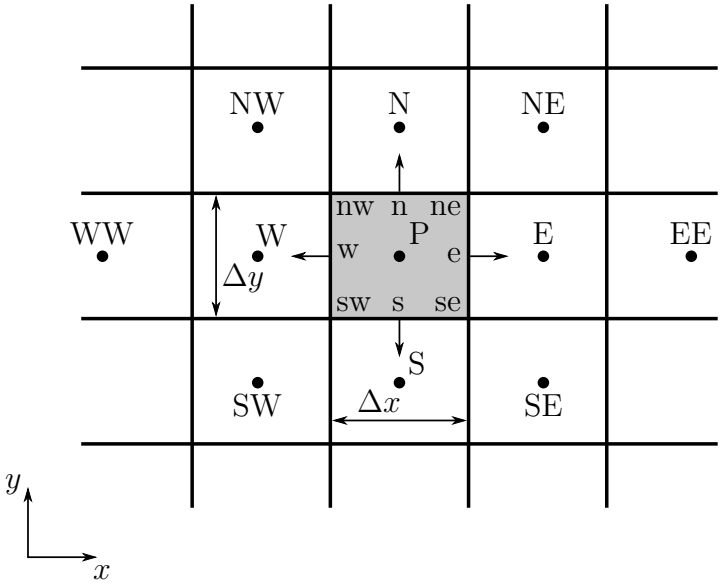


Figure 2.3: Control volume in an exemplary two-dimensional Cartesian FV discretization [49, p. 73].

superscripts W, E, N, S, NW, NE, SW, and SE to the quantities located at the centers of the control volumes adjacent to the considered control volume with center P. For the boundaries of the control volume, we utilize lowercase superscripts w, e, n, s, nw, ne, sw, and se. The extension to higher space dimensions and non-orthogonal meshes will be omitted here for the sake of conciseness.

In the momentum equation (2.62), the convective term $\int_{\Gamma_\ell} \rho \boldsymbol{v}\tilde{\boldsymbol{v}} \cdot \boldsymbol{n} \,\mathrm{d}\Gamma$, the diffusive term $\int_{\Gamma_\ell} \mu \,\mathrm{grad}\, \boldsymbol{v} \cdot \boldsymbol{n} \,\mathrm{d}\Gamma$, and the pressure term $\int_{\Gamma_\ell} p\boldsymbol{n} \,\mathrm{d}\Gamma$ are surface integrals of the type $\int_{\Gamma_\ell} f \,\mathrm{d}\Gamma$, where Γ_ℓ refers to the surface of the finite control volume Ω_ℓ. For their numerical treatment, these integrals are split into a sum of integrals over the $m_{\mathrm{s},\ell}$ surface segments $\Gamma_{\ell,k}$, $k = 1, \ldots, m_{\mathrm{s},\ell}$ of the control volume Ω_ℓ such that

$$\int_{\Gamma_\ell} f \,\mathrm{d}\Gamma = \sum_{k=1}^{m_{\mathrm{s},\ell}} \int_{\Gamma_{\ell,k}} f \,\mathrm{d}\Gamma \,. \tag{2.68}$$

Subsequently, each of the integrals must be integrated numerically. As an example, let us consider the surface segment "e" of the finite control volume depicted in Figure 2.3. Applying the midpoint rule results in

$$\int_{\Gamma_\mathrm{e}} f \,\mathrm{d}\Gamma \approx f_\mathrm{e} |\Gamma_\mathrm{e}| \,, \tag{2.69}$$

while the trapezoidal rule produces

$$\int_{\Gamma_\mathrm{e}} f \,\mathrm{d}\Gamma \approx \frac{|\Gamma_\mathrm{e}|}{2} (f_\mathrm{ne} + f_\mathrm{se}) \,. \tag{2.70}$$

By $|\Gamma_\mathrm{e}|$, we denote the length of the surface segment Γ_e. Both the midpoint and the trapezoidal rule are second-order accurate approximations. A fourth-order accurate scheme is given by the Simpson rule

$$\int_{\Gamma_\mathrm{e}} f \,\mathrm{d}\Gamma \approx \frac{|\Gamma_\mathrm{e}|}{2} (f_\mathrm{ne} + 4 f_\mathrm{e} + f_\mathrm{se}) \,. \tag{2.71}$$

Further, the term $\int_{\Omega_\ell} \rho \boldsymbol{v} \, \mathrm{d}\Omega$ and the integral over the volumetric forces $\int_{\Omega_\ell} \rho \boldsymbol{b} \, \mathrm{d}\Omega$ in Equation (2.62) are integrals of the kind $\int_{\Omega_\ell} q \, \mathrm{d}\Omega$. Here, we can also employ the midpoint rule to obtain the second-order accurate approximation

$$\int_{\Omega_\ell} q \, \mathrm{d}\Omega \approx q_\mathrm{P} \, |\Omega_\ell| \; . \tag{2.72}$$

By $|\Omega_\ell|$, we denote the area of the finite control volume Ω_ℓ. Higher-order approximations can be constructed by fitting a polynomial to the variable value in the center of the considered control volume and to the values in the centers of the neighboring control volumes. For instance, after determining the coefficients in the expression

$$q(x, y) = a_0 + a_1 x + a_2 y + a_3 x^2 + a_4 y^2 + a_5 xy + a_6 x^2 y + a_7 x y^2 + a_8 x^2 y^2 \; , \tag{2.73}$$

the integration on a regular Cartesian grid yields the fourth-order accurate approximation

$$\int_{\Omega_\ell} q \, \mathrm{d}\Omega \approx \frac{|\Omega_\ell|}{36} \left(16 q_\mathrm{P} + 4 q_\mathrm{s} + 4 q_\mathrm{n} + 4 q_\mathrm{w} + 4 q_\mathrm{e} + q_\mathrm{se} + q_\mathrm{sw} + q_\mathrm{ne} + q_\mathrm{nw} \right) \; . \tag{2.74}$$

Apparently, the numerical integration of the surface and the volume integrals require the evaluation of the integrand at points which may not coincide with the control volume centers. Hence, an interpolation scheme must be applied to interpolate the values from the control volume centers to the evaluation points. In the following, we presume that we want to interpolate the value ϕ_e of the variable ϕ on the boundary segment "e". A simple but also highly diffusive and, consequently, inaccurate interpolation scheme is the upwind differencing scheme

$$\phi_\mathrm{e} \approx \begin{cases} \phi_\mathrm{P} \text{ if } (\boldsymbol{v} \cdot \boldsymbol{n}) \geq 0 \\ \phi_\mathrm{E} \text{ if } (\boldsymbol{v} \cdot \boldsymbol{n}) < 0 \end{cases} \; . \tag{2.75}$$

A better interpolation is given by the central differencing scheme

$$\phi_\mathrm{e} \approx \lambda_\mathrm{e} \phi_\mathrm{E} + (1 - \lambda_\mathrm{e}) \phi_\mathrm{P} \; , \quad \lambda_\mathrm{e} = \frac{x_\mathrm{e} - x_\mathrm{P}}{x_\mathrm{E} - x_\mathrm{P}} \; , \tag{2.76}$$

which also provides a good approximation for the diffusive flux:

$$\left(\frac{\partial \phi}{\partial x} \right)_\mathrm{e} \approx \frac{\phi_\mathrm{E} - \phi_\mathrm{P}}{x_\mathrm{E} - x_\mathrm{P}} \; . \tag{2.77}$$

Higher-order interpolations can be constructed by fitting a polynomial to the values in the neighboring control volume centers. A fourth-order accurate interpolation is, for instance, given by fitting the function

$$\phi(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 \tag{2.78}$$

to the values ϕ_W, ϕ_P, ϕ_E, and ϕ_EE. For an equidistant Cartesian discretization, this leads us to

$$\phi_\mathrm{e} \approx \frac{27 \phi_\mathrm{P} + 27 \phi_\mathrm{E} - 3 \phi_\mathrm{W} - 3 \phi_\mathrm{EE}}{48} \; . \tag{2.79}$$

Differentiating the polynomial results in

$$\left(\frac{\partial \phi}{\partial x} \right)_\mathrm{e} = a_1 + 2 a_2 x + 3 a_3 x^2 \; , \tag{2.80}$$

which, for an equidistant Cartesian grid, can be used to approximate the diffusive flux as

$$\left(\frac{\partial \phi}{\partial x}\right)_{\mathrm{e}} \approx \frac{27\phi_{\mathrm{E}} - 27\phi_{\mathrm{P}} + \phi_{\mathrm{W}} - \phi_{\mathrm{EE}}}{24\Delta x} \ . \tag{2.81}$$

The above spatial discretizations finally lead to a system of equations, which contains the pressures p_ℓ and the velocities $\boldsymbol{v}_\ell$ at the control volume centers as unknowns.

Following the spatial discretization, Equation (2.62) must also be discretized in time. For this step, it suffices to consider an equation of the kind

$$\frac{\mathrm{D}(\phi(t))}{\mathrm{D}t} = f(t, \phi(t)) \ , \quad \phi(t = t_0) = \phi_0 \tag{2.82}$$

which resembles the nature of the momentum equation with respect to time. Starting from $t = t_j$, we aim to find the solution ϕ at time $t_{j+1} = t_j + \Delta t$. Commonly used methods for the integration of ordinary differential equations such as (2.82) are the explicit Euler scheme

$$\phi_{j+1} \approx \phi_j + \Delta t \, f\left(t_j, \phi_j\right) \ , \tag{2.83}$$

or the implicit Euler scheme (also known as first-order backward differencing scheme)

$$\phi_{j+1} \approx \phi_j + \Delta t \, f\left(t_{j+1}, \phi_{j+1}\right) \ . \tag{2.84}$$

While the explicit Euler scheme requires a time step size Δt smaller than a problem-dependent critical time step size Δt_{crit} for stability reasons, the implicit Euler scheme does not exhibit such a limitation – but is unconditionally stable. Other possible time integration schemes are the second-order backward differencing scheme [176, p. 190]

$$\phi_{j+1} \approx \frac{1}{3}\left(4\phi_j - \phi_{j-1} + 2\Delta t \, f\left(t_{j+2}, \phi_{j+2}\right)\right) \tag{2.85}$$

or the trapezoidal rule

$$\phi_{j+1} \approx \phi_j + \frac{\Delta t}{2}\left(f\left(t_j, \phi_j\right) + f\left(t_{j+1}, \phi_{j+1}\right)\right) \ . \tag{2.86}$$

As already mentioned above, the unknowns in the incompressible Navier-Stokes equations (2.41), (2.63) are the pressure p and the velocity $\boldsymbol{v}$. Apparently, the momentum equation (2.63) serves to determine the components of the velocity $\boldsymbol{v}$. The pressure p, however, cannot be obtained from the remaining continuity equation (2.41) as it does not even contain the pressure. We circumvent this problem by combining the continuity equation and momentum equation in a Poisson equation for the pressure, which, for the case of constant density and viscosity, reads [32, pp. 6–7]:

$$\operatorname{div}\left(\operatorname{grad} p\right) = \operatorname{div}\left(\rho \boldsymbol{b} - \tilde{\boldsymbol{v}} \cdot \operatorname{grad} \boldsymbol{v}\right) \ . \tag{2.87}$$

Note that the Laplacian on the left-hand side of the equation is the product of the divergence operator, stemming from the continuity equation and the gradient operator originating from the momentum equation. In a numerical approximation of this equation, it is important to retain the consistency of these operators and to define the discrete Laplacian as the product of the divergence and gradient approximation used in the basic equations [49, p. 167 sq.].

For the numerical solution of the discretized system of coupled partial differential equations (2.41), (2.63), we employ an enhanced version of the PISO (Pressure Implicit with Splitting of Operator) algorithm [84] to determine the pressure p and velocity $\boldsymbol{v}$. Following [84, 49, p. 178], the procedure can be summarized as follows:

1. In time step t_{j+1}, estimate the pressure p_{j+1} using the converged value p_j from the previous time step t_j. Assign $p^* := p_j$.
2. Assemble and solve the discretized momentum equation to obtain the velocity $\boldsymbol{v}^*$.
3. Assemble and solve the discretized pressure correction equation (2.87) to determine p'.
4. Update the pressure according to $p^* := p^* + p'$.
5. Determine a velocity correction $\boldsymbol{v}'$ such that the continuity equation is fulfilled.
6. Update the velocity according to $\boldsymbol{v}^* := \boldsymbol{v}^* + \boldsymbol{v}'$.
7. Repeat steps 2–6 until convergence.
8. Assign $p_{j+1} := p^*$, $\boldsymbol{v}_{j+1} := \boldsymbol{v}^*$ and proceed to the next time step.

In view of its application to FSI problems later on, we have introduced the ALE formulation of the Navier-Stokes equations. This formulation includes the mesh velocity $\hat{\boldsymbol{v}}$, which is computed from the mesh displacement. In an FSI context, the mesh displacement is prescribed at the fluid-structure interface and needs to propagate through the mesh while retaining the validity of the boundary conditions on the rest of the fluid boundary and the quality of the finite control volumes. Different methods have been proposed in the literature, see [156, 85, 171], for instance. In this work, we usually employ an approach based on the solution of the Laplace equation

$$\operatorname{div}(\gamma \operatorname{grad} \boldsymbol{d}) = 0 \tag{2.88}$$

for the displacement $\boldsymbol{d}$ of the mesh vertices, where the diffusion coefficient γ can be adapted to achieve that the mesh behaves relatively stiff in the vicinity of the fluid-structure interface, while becoming softer with increasing distance from the boundary. A common choice is $\gamma(r) = r^{-2}$, where r is the minimum distance between the mesh vertex and the boundary.

2.2 Potential Flow Equations

Although the availability of computational resources is steadily increasing, the numerical treatment of the Navier-Stokes equations, in particular in the context of FSI, may still prove to be prohibitively expensive for larger problems. If applicable, however, the Navier-Stokes equations can be simplified considerably by imposing certain restrictions on the behavior of the fluid flow, which leads to Laplace's equation as the governing equation. For the numerical treatment of this equation, it is possible to employ the BEM, which can also be easily integrated into a partitioned solution strategy for an FSI analysis.

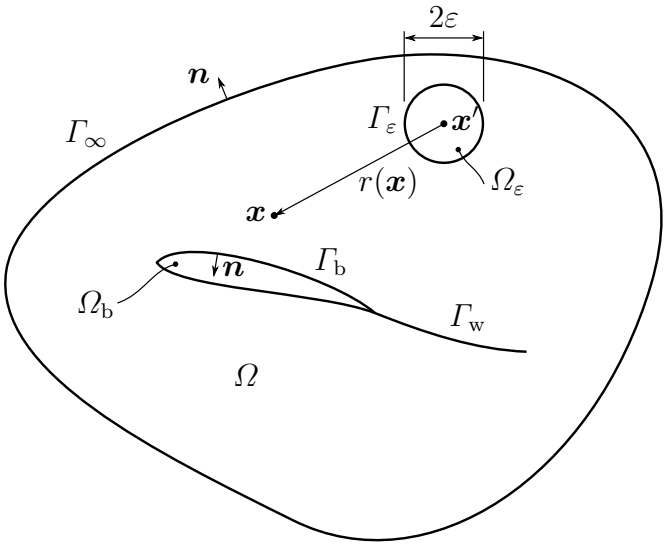


Figure 2.4: Potential flow problem [91, p. 53].

2.2.1 Problem Statement

In potential theory, we assume the fluid to be incompressible, which is conveyed by (2.39), inviscid, i.e., $\mu = 0$, and irrotational:

$$\operatorname{curl} \boldsymbol{v} = \boldsymbol{0} \,. \tag{2.89}$$

From vector calculus, we then deduce the existence of a scalar velocity potential Φ, the gradient of which again yields the velocity $\boldsymbol{v}$ such that

$$\operatorname{grad} \Phi = \boldsymbol{v} \,. \tag{2.90}$$

If we take the incompressibility condition (2.39) and the resulting continuity equation (2.41) into account, we are left with Laplace's equation

$$\operatorname{div}(\operatorname{grad} \Phi) = \Delta \Phi = 0 \tag{2.91}$$

for the velocity potential Φ. Equation (2.91) marks the starting point for the derivation of the governing equations of potential flow. In our discussion of the theoretical foundations, we primarily follow the standard textbooks [91, 87, 34], but also refer to [177, 168, 27, 157, 158, 1].

To begin with, let us consider the situation depicted in Figure 2.4, where a body $\Omega_\mathrm{b} \subset \mathbb{R}^d$ bounded by a sufficiently regular boundary $\Gamma_\mathrm{b} := \partial\Omega_\mathrm{b} \subset \mathbb{R}^{d-1}$ is completely submerged in a fluid. In the following, let us focus on the three-dimensional case, where $d = 3$. The fluid domain Ω is bounded by the boundary $\Gamma_\infty := \partial\Omega_\infty$ and the outer unit normal $\boldsymbol{n}$ points to the outside of the fluid domain Ω. Clearly, the directional derivative normal to the body's surface must vanish, since no flow can pass through the body [91, p. 423]:

$$(\operatorname{grad} \Phi - \boldsymbol{v}_\mathrm{b}) \cdot \boldsymbol{n} = 0 \quad \text{on } \Gamma_\mathrm{b} \,. \tag{2.92}$$

Here, $\boldsymbol{n}$ denotes the outer unit normal of the fluid domain Ω on the boundary Γ_b (that is, the inner unit normal of Ω_b) and $\boldsymbol{v}_\mathrm{b}$ is the surface velocity as seen from the inertial frame.

Note the presence of the minus sign in the inertial frame, which corresponds to a positive velocity in the body frame. In addition, the disturbance due to the motion of the body decreases with increasing distance r from the body – and vanishes in the limit case that r tends to infinity [91, p. 53]:

$$\lim_{r\to\infty}(\operatorname{grad}\Phi - \boldsymbol{v}_{\mathrm{rel}}) = \boldsymbol{0}\ , \tag{2.93}$$

where $r = r(\boldsymbol{x})$ is a function of the position $\boldsymbol{x}$ and $\boldsymbol{v}_{\mathrm{rel}} = \boldsymbol{v}_\infty - \boldsymbol{v}_{\mathrm{b}}$ represents the relative velocity between the undisturbed fluid $\boldsymbol{v}_\infty$ and the velocity $\boldsymbol{v}_{\mathrm{b}}$ of the body.

Following [91, pp. 53–57], we aim to represent the solution to Laplace's equation (2.91) for the fluid potential Φ by an integral involving the boundary conditions for Φ. For two scalar-valued functions $\Phi_1 = \Phi_1(\boldsymbol{x})$, $\Phi_2 = \Phi_2(\boldsymbol{x})$, it follows from applying Gauss's divergence theorem that

$$\int_\Gamma (\Phi_1 \operatorname{grad}\Phi_2 - \Phi_2 \operatorname{grad}\Phi_1)\cdot\boldsymbol{n}\,\mathrm{d}\Gamma = \int_\Omega (\Phi_1\Delta\Phi_2 - \Phi_2\Delta\Phi_1)\,\mathrm{d}\Omega\ , \tag{2.94}$$

which is also known as Green's second identity. The boundary Γ comprises the boundary Γ_{b} around the body, the wake surface Γ_{w}, which represents a surface across which the velocity or the velocity potential may be discontinuous, and the outer boundary Γ_∞ such that

$$\Gamma = \Gamma_{\mathrm{b}} \cup \Gamma_{\mathrm{w}} \cup \Gamma_\infty\ . \tag{2.95}$$

As indicated in Figure 2.4, let us place a source with a potential

$$\Phi_1(\boldsymbol{x}) = \frac{1}{r(\boldsymbol{x})} \tag{2.96}$$

at the point $\boldsymbol{x}'$, where $r(\boldsymbol{x}) := \|\boldsymbol{x} - \boldsymbol{x}'\|_2$ denotes the Euclidean distance of an arbitrary point $\boldsymbol{x}$ from the location $\boldsymbol{x}'$ of the source. Apparently, $\Phi_1(\boldsymbol{x})$ is unbounded as $\boldsymbol{x}$ approaches $\boldsymbol{x}'$, or, equivalently, as r tends to zero. Further, let us set

$$\Phi_2(\boldsymbol{x}) = \Phi(\boldsymbol{x})\ , \tag{2.97}$$

where $\Phi(\boldsymbol{x})$ is the velocity potential in Ω as introduced in Equation (2.90). In the situation where $\boldsymbol{x}' \notin \Omega$, i.e. the source is placed outside the fluid domain Ω, both Φ_1 and Φ_2 satisfy Laplace's equation – and it follows from (2.94) that

$$\int_\Gamma \left(\frac{1}{r}\operatorname{grad}\Phi - \Phi\operatorname{grad}\frac{1}{r}\right)\cdot\boldsymbol{n}\,\mathrm{d}\Gamma = 0\ . \tag{2.98}$$

If, on the other hand, $\boldsymbol{x}' \in \Omega$, i.e. the source is placed inside the fluid domain Ω, Φ_1 and Φ_2 still satisfy Laplace's equation in the region $\Omega \setminus \Omega_\varepsilon$ – and (2.94) becomes

$$\begin{aligned}\int_{\Gamma\cup\Gamma_\varepsilon} &\left(\frac{1}{r}\operatorname{grad}\Phi - \Phi\operatorname{grad}\frac{1}{r}\right)\cdot\boldsymbol{n}\,\mathrm{d}\Gamma \\ &= -\int_{\Gamma_\varepsilon}\left(\frac{1}{r}\operatorname{grad}\Phi - \Phi\operatorname{grad}\frac{1}{r}\right)\cdot\boldsymbol{n}\,\mathrm{d}\Gamma + \int_\Gamma\left(\frac{1}{r}\operatorname{grad}\Phi - \Phi\operatorname{grad}\frac{1}{r}\right)\cdot\boldsymbol{n}\,\mathrm{d}\Gamma = 0\ .\end{aligned} \tag{2.99}$$

For the evaluation of the integral over the sphere Γ_ε enclosing the point $\boldsymbol{x}'$, we take into account that $\operatorname{grad}(1/r)\cdot\boldsymbol{n} = -(1/r^2)$. Subsequently, we use Gauss's divergence theorem

and the mean value theorem, which states that the average potential over the spherical surface Γ_ε equals the potential $\Phi(\boldsymbol{x}')$ at the sphere center $\boldsymbol{x}'$ [75, p. 232, 1, p. 6]:

$$\begin{aligned}\int_{\Gamma_\varepsilon} \left(\frac{1}{r}\operatorname{grad}\Phi - \Phi\operatorname{grad}\frac{1}{r}\right)\cdot\boldsymbol{n}\,\mathrm{d}\Gamma &= \frac{1}{r}\int_{\Gamma_\varepsilon}\operatorname{grad}\Phi\cdot\boldsymbol{n}\,\mathrm{d}\Gamma + \frac{1}{r^2}\int_{\Gamma_\varepsilon}\Phi\,\mathrm{d}\Gamma \\ &= \frac{1}{r}\int_{\Omega_\varepsilon}\underbrace{\operatorname{div}\operatorname{grad}\Phi}_{=0}\,\mathrm{d}\Omega + \frac{1}{r^2}\underbrace{\int_{\Gamma_\varepsilon}\Phi\,\mathrm{d}\Gamma}_{4\pi r^2\Phi(\boldsymbol{x}')} = 4\pi\Phi(\boldsymbol{x}')\ .\end{aligned} \tag{2.100}$$

Note that the first integral vanishes due to the fact that Φ is a solution to Laplace's equation (2.91). Plugging the result (2.100) into Equation (2.99), we obtain

$$\Phi(\boldsymbol{x}') = \frac{1}{4\pi}\int_{\Gamma}\left(\frac{1}{r}\operatorname{grad}\Phi - \Phi\operatorname{grad}\frac{1}{r}\right)\cdot\boldsymbol{n}\,\mathrm{d}\Gamma\ . \tag{2.101}$$

Note that (2.101) represents the potential at any point $\boldsymbol{x}' \in \Omega$ in the flow domain in terms of the potential Φ and the normal derivative $\partial\Phi/\partial\boldsymbol{n}$ on the boundary Γ. Recalling (2.95), we replace the integral over Γ by integrals over the boundary segments Γ_b, Γ_w, and Γ_∞:

$$\begin{aligned}\Phi(\boldsymbol{x}') = \frac{1}{4\pi}\int_{\Gamma_\mathrm{b}}\left(\frac{1}{r}\operatorname{grad}\Phi - \Phi\operatorname{grad}\frac{1}{r}\right)\cdot\boldsymbol{n}\,\mathrm{d}\Gamma + \frac{1}{4\pi}\int_{\Gamma_\mathrm{w}}\left(\frac{1}{r}\operatorname{grad}\Phi - \Phi\operatorname{grad}\frac{1}{r}\right)\cdot\boldsymbol{n}\,\mathrm{d}\Gamma \\ + \frac{1}{4\pi}\int_{\Gamma_\infty}\left(\frac{1}{r}\operatorname{grad}\Phi - \Phi\operatorname{grad}\frac{1}{r}\right)\cdot\boldsymbol{n}\,\mathrm{d}\Gamma\ .\end{aligned} \tag{2.102}$$

Regarding the first integral, let us consider the situation in which some imaginary flow occurs within the boundary Γ_b. From (2.101), we deduce that the associated interior potential Φ_i, evaluated at a point $\boldsymbol{x}' \in \Omega_\mathrm{b}$, amounts to

$$\Phi_\mathrm{i}(\boldsymbol{x}') = \frac{1}{4\pi}\int_{\Gamma_\mathrm{b}}\left(\frac{1}{r}\operatorname{grad}\Phi_\mathrm{i} - \Phi_\mathrm{i}\operatorname{grad}\frac{1}{r}\right)\cdot\boldsymbol{n}\,\mathrm{d}\Gamma\ . \tag{2.103}$$

If, on the other hand, $\boldsymbol{x}' \in \Omega$, then, according to (2.98),

$$\Phi_\mathrm{i}(\boldsymbol{x}') = \frac{1}{4\pi}\int_{\Gamma_\mathrm{b}}\left(\frac{1}{r}\operatorname{grad}\Phi_\mathrm{i} - \Phi_\mathrm{i}\operatorname{grad}\frac{1}{r}\right)\cdot\boldsymbol{n}\,\mathrm{d}\Gamma = 0\ . \tag{2.104}$$

Accounting for the opposite orientation of the outer surface normal for the interior potential Φ_i and adding this expression to the first integral in (2.102) leads us to

$$\begin{aligned}&\frac{1}{4\pi}\int_{\Gamma_\mathrm{b}}\left(\frac{1}{r}\operatorname{grad}\Phi - \Phi\operatorname{grad}\frac{1}{r}\right)\cdot\boldsymbol{n}\,\mathrm{d}\Gamma \\ &\qquad = \frac{1}{4\pi}\int_{\Gamma_\mathrm{b}}\left(\frac{1}{r}\operatorname{grad}(\Phi-\Phi_\mathrm{i}) - (\Phi-\Phi_\mathrm{i})\operatorname{grad}\frac{1}{r}\right)\cdot\boldsymbol{n}\,\mathrm{d}\Gamma\ .\end{aligned} \tag{2.105}$$

For the contribution of the integral over the wake surface Γ_w, we assume the wake to be thin and, further, that $\partial\Phi/\partial\boldsymbol{n}$ is continuous over the wake (that is, no fluid-dynamic loads will be supported by the wake). Based upon these assumptions, we obtain

$$\frac{1}{4\pi}\int_{\Gamma_\mathrm{w}}\left(\frac{1}{r}\operatorname{grad}\Phi - \Phi\operatorname{grad}\frac{1}{r}\right)\cdot\boldsymbol{n}\,\mathrm{d}\Gamma = -\frac{1}{4\pi}\int_{\Gamma_\mathrm{w}}\Phi\boldsymbol{n}\cdot\operatorname{grad}\frac{1}{r}\,\mathrm{d}\Gamma\ . \tag{2.106}$$

For the remaining integral term over the boundary Γ_∞, we write

$$\Phi_\infty(\boldsymbol{x}') = \frac{1}{4\pi}\int_{\Gamma_\infty}\left(\frac{1}{r}\operatorname{grad}\Phi - \Phi\operatorname{grad}\frac{1}{r}\right)\cdot\boldsymbol{n}\,\mathrm{d}\Gamma\ . \tag{2.107}$$

Incorporating the relations (2.105)–(2.107) into (2.102) results in

$$\begin{aligned}\Phi(\boldsymbol{x}') = \frac{1}{4\pi}\int_{\Gamma_\mathrm{b}}\left(\frac{1}{r}\operatorname{grad}(\Phi-\Phi_\mathrm{i}) - (\Phi-\Phi_\mathrm{i})\operatorname{grad}\frac{1}{r}\right)\cdot\boldsymbol{n}\,\mathrm{d}\Gamma \\ - \frac{1}{4\pi}\int_{\Gamma_\mathrm{w}}\Phi\boldsymbol{n}\cdot\operatorname{grad}\frac{1}{r}\,\mathrm{d}\Gamma + \Phi_\infty(\boldsymbol{x}')\ .\end{aligned} \tag{2.108}$$

As already noted above, Equation (2.108) provides the solution for the potential $\Phi(\boldsymbol{x}')$, depending on Φ and $\partial\Phi/\partial\boldsymbol{n}$ on the boundaries. Substituting

$$-\mu = \Phi - \Phi_\mathrm{i} \tag{2.109}$$

for the difference between the external potential Φ and the internal potential Φ_i[1] and

$$-\sigma = \frac{\partial\Phi}{\partial\boldsymbol{n}} - \frac{\partial\Phi_\mathrm{i}}{\partial\boldsymbol{n}} \tag{2.110}$$

for the difference in the normal derivative $\partial\Phi/\partial\boldsymbol{n}$ of the external potential and the normal derivative $\partial\Phi_\mathrm{i}/\partial\boldsymbol{n}$ of the internal potential, Equation (2.108) is restated as

$$\begin{aligned}\Phi(\boldsymbol{x}') = -\frac{1}{4\pi}\int_{\Gamma_\mathrm{b}}\left(\sigma\left(\frac{1}{r}\right) - \mu\boldsymbol{n}\cdot\operatorname{grad}\left(\frac{1}{r}\right)\right)\mathrm{d}\Gamma \\ + \frac{1}{4\pi}\int_{\Gamma_\mathrm{w}}\left(\mu\boldsymbol{n}\cdot\operatorname{grad}\left(\frac{1}{r}\right)\right)\mathrm{d}\Gamma + \Phi_\infty(\boldsymbol{x}')\ .\end{aligned} \tag{2.111}$$

(2.109) is termed a *doublet*, while (2.110) represents a *source*. Note that both expressions approach zero as r tends to infinity. Following that, we replace $\boldsymbol{n}\cdot\operatorname{grad}(1/r)$ by $\partial(1/r)/\partial\boldsymbol{n}$ to eventually obtain

$$\Phi(\boldsymbol{x}') = -\frac{1}{4\pi}\int_{\Gamma_\mathrm{b}}\left(\sigma\left(\frac{1}{r}\right) - \mu\frac{\partial}{\partial\boldsymbol{n}}\left(\frac{1}{r}\right)\right)\mathrm{d}\Gamma + \frac{1}{4\pi}\int_{\Gamma_\mathrm{w}}\left(\mu\frac{\partial}{\partial\boldsymbol{n}}\left(\frac{1}{r}\right)\right)\mathrm{d}\Gamma + \Phi_\infty(\boldsymbol{x}')\ . \tag{2.112}$$

In order to determine a solution for the velocity potential Φ in the fluid domain Ω, the strengths of the doublet and source distribution on the surface must be determined. However, there is no unique solution to the problem unless some assumptions are made – which are usually based on the physics of the problem under consideration. For instance, we may choose

$$\frac{\partial\Phi_\mathrm{i}}{\partial\boldsymbol{n}} = \frac{\partial\Phi}{\partial\boldsymbol{n}} \quad \text{on } \Gamma_\mathrm{b}\ , \tag{2.113}$$

which results in the source term vanishing on the boundary. Then, only the contribution from the doublet will remain, and the problem is termed a *Neumann problem*. If, on the other hand, we impose

$$\Phi_\mathrm{i} = \Phi \quad \text{on } \Gamma_\mathrm{b}\ , \tag{2.114}$$

the doublet term vanishes on the boundary, and the source term is what remains; this problem is named a *Dirichlet problem*.

[1] The symbol μ has previously been used to represent the dynamic viscosity of the fluid. Since in potential theory, $\mu = 0$ is assumed and hence does not appear in any of the relevant equations, there is little chance of confusion with the doublet strength introduced at this point.

2.2.2 Discretization and Numerical Solution

A solution of the potential problem using analytical techniques is only possible for problems based on relatively simple geometries. For advanced applications involving more complicated geometries, numerical methods have to be applied. Following [91, pp. 237–250,421–431], in the BEM, the surface of the body Γ_b is subdivided into a set of m body panels, whereas the wake is discretized by m_w wake panels such that

$$\Gamma_\mathrm{b} \approx \Gamma_{\mathrm{b},h} = \bigcup_{k=1}^{m} \Gamma_k \quad \text{and} \quad \Gamma_\mathrm{w} \approx \Gamma_{\mathrm{w},h} = \bigcup_{\ell=1}^{m_\mathrm{w}} \Gamma_\ell \; . \tag{2.115}$$

By the symbols $\Gamma_{\mathrm{b},h}$ and $\Gamma_{\mathrm{w},h}$, we denote the discrete approximations of the body's surface Γ_b and the wake surface Γ_w, respectively. Based on these discretizations, the Dirichlet boundary condition (2.114) takes the following form:

$$\begin{aligned} &\sum_{k=1}^{m} \frac{1}{4\pi} \int_{\Gamma_k} \mu \boldsymbol{n} \cdot \operatorname{grad}\left(\frac{1}{r}\right) \mathrm{d}\Gamma \\ &\qquad + \sum_{\ell=1}^{m_\mathrm{w}} \frac{1}{4\pi} \int_{\Gamma_\ell} \mu \boldsymbol{n} \cdot \operatorname{grad}\left(\frac{1}{r}\right) \mathrm{d}\Gamma - \sum_{k=1}^{m} \frac{1}{4\pi} \int_{\Gamma_k} \sigma \left(\frac{1}{r}\right) \mathrm{d}\Gamma = 0 \; . \end{aligned} \tag{2.116}$$

For a panel Γ_k of constant source strength μ, the influence on the collocation point $\boldsymbol{x}'$ is

$$C_k = \frac{1}{4\pi} \int_{\Gamma_k} \frac{\partial}{\partial \boldsymbol{n}} \left(\frac{1}{r}\right) \mathrm{d}\Gamma \; . \tag{2.117}$$

For a panel of constant doublet strength σ, we have

$$B_k = -\frac{1}{4\pi} \int_{\Gamma_k} \left(\frac{1}{r}\right) \mathrm{d}\Gamma \; . \tag{2.118}$$

Having defined the influence coefficients C_k in (2.117) and B_k in (2.118), Equation (2.116) becomes

$$\sum_{k=1}^{m} C_k \mu_k + \sum_{\ell=1}^{m_\mathrm{w}} C_\ell \mu_\ell + \sum_{k=1}^{m} B_k \sigma_k = 0 \tag{2.119}$$

for each collocation point $\boldsymbol{x}'$. If the doublet strength σ_k on a panel Γ_k is selected as $\sigma_k = -\boldsymbol{n}_k \cdot \boldsymbol{v}_\mathrm{b}$ with $\boldsymbol{v}_\mathrm{b}$ being the surface velocity, the contributions $B_k \sigma_k$ are known and can be moved to the right-hand side of Equation (2.119).

The Kutta condition states that the wake doublet strength μ_w is constant and equals the doublet strength $\mu^\star$ at the trailing edge:

$$\mu_\mathrm{w} = \mu^\star = \text{const.} \tag{2.120}$$

Alternatively, this condition may also be written in terms of the doublet strengths μ_u and μ_l on the upper and lower surface at the trailing edge:

$$\mu_\mathrm{w} = \mu_\mathrm{u} - \mu_\mathrm{l} \; . \tag{2.121}$$

Based on the Kutta condition (2.121), the doublet strengths μ_ℓ associated to the wake panels Γ_ℓ can be related to the doublet strengths μ_k on the body panels Γ_k. For a pair

of arbitrary body panels Γ_u and Γ_l at the trailing edge, the doublet strength on the corresponding wake panel Γ_w evaluates to

$$C_w \mu_w = C_w(\mu_u - \mu_l) \; . \tag{2.122}$$

Substituting this algebraic relation into (2.119) enables us to replace the influence coefficients C_k related to the unknown body doublet strengths μ_k such that

$$\begin{aligned} A_k &= C_k \quad \text{if the panel } \Gamma_k \text{ is } \textit{not} \text{ at the trailing edge} \\ \text{and } A_k &= C_k \pm C_w \quad \text{if the panel } \Gamma_k \text{ is at the trailing edge} \; . \end{aligned} \tag{2.123}$$

In the second expression, the plus sign applies if the panel Γ_k is located at the upper side of the trailing edge, whilst the minus sign must be used for panels on the lower side of the trailing edge. With the abbreviations defined in Equation (2.123), we arrive at the algebraic relation

$$\sum_{k=1}^{m} A_k \mu_k = -\sum_{k=1}^{m} B_k \sigma_k \tag{2.124}$$

for each collocation point $\boldsymbol{x}'$. Note that Equation (2.124) is linear in the unknown doublet strengths μ_k. Hence, we obtain a linear system of equations by writing down (2.124) for each collocation point $\boldsymbol{x}_k, k = 1, \ldots, m$:

$$\begin{pmatrix} A_{11} & \cdots & A_{1m} \\ \vdots & \ddots & \vdots \\ A_{m1} & \cdots & A_{mm} \end{pmatrix} \begin{pmatrix} \mu_1 \\ \vdots \\ \mu_m \end{pmatrix} = - \begin{pmatrix} B_{11} & \cdots & B_{1m} \\ \vdots & \ddots & \vdots \\ B_{m1} & \cdots & B_{mm} \end{pmatrix} \begin{pmatrix} \sigma_1 \\ \vdots \\ \sigma_m \end{pmatrix} \; , \tag{2.125}$$

or, in compact notation,

$$\mathbf{A}\boldsymbol{\mu} = -\mathbf{B}\boldsymbol{\sigma} = \mathbf{b} \; . \tag{2.126}$$

After solving the linear system of equations for the unknowns $\mu_1, \ldots, \mu_m$, the tangential velocity components $\tilde{v}_\xi$, $\tilde{v}_\eta$ in the local (ξ, η, ζ) panel coordinate system can be obtained from

$$\tilde{v}_\xi = \frac{\partial \sigma}{\partial \xi} \quad \text{and} \quad \tilde{v}_\eta = \frac{\partial \sigma}{\partial \eta} \tag{2.127}$$

by numerical differentiation involving the neighboring panels. The normal component is computed from

$$\tilde{v}_\zeta = -\sigma \; . \tag{2.128}$$

Subsequently, the total velocity $\boldsymbol{v}_k$ in the panel coordinate system reads

$$\boldsymbol{v}_k = -\begin{pmatrix} v_{\mathrm{b},\xi} & v_{\mathrm{b},\eta} & v_{\mathrm{b},\zeta} \end{pmatrix}_\ell^{\mathrm{T}} + \begin{pmatrix} \tilde{v}_\xi & \tilde{v}_\eta & \tilde{v}_\zeta \end{pmatrix}_k^{\mathrm{T}} \; . \tag{2.129}$$

Finally, we calculate the panel pressure p_k from the Bernoulli equation

$$\frac{p_\infty - p_k}{\rho} = \frac{\|\boldsymbol{v}_k\|_2^2}{2} - \frac{\|\boldsymbol{v}_{\mathrm{b},k}\|_2^2}{2} + \frac{\partial \Phi}{\partial t} \; . \tag{2.130}$$

For the computation of the evolution of pressure over time, we can employ one of the time integration schemes for ordinary differential equations of first order introduced in Section 2.1.3.

3 Structural Problems

In this chapter, we consider the structural subproblem of the coupled problem. In the most general case, the mechanical structure is treated as a deformable body undergoing large displacements and large rotations. In Section 3.1, we derive the governing equations of motion and introduce the finite element method (FEM) for the efficient numerical analysis of the problem. If the body undergoes large displacements and large rotations, but only small deformations, and if the internal stress state is not of interest, it is sufficient to idealize the structure as a rigid body – which simplifies the governing equations considerably. Therefore, we will briefly state the rigid body equations and review the numerical methods for their solution in Section 3.2.

3.1 Deformable Body Equations

To study the dynamic behavior of a deformable structure, it is necessary to describe the configuration of the body at different instants of time. Following the discussion of the kinematical relations, we include the effect of external forces acting on the structure in order to derive the governing balance equations. Making these equations amenable to a numerical analysis, we transform the strong form of the equilibrium equations into a weak form. Subsequent to linearizing the weak form of the equilibrium equations, we outline the spatial and temporal discretization in the framework of the FEM. Our discussion is primarily based on the standard textbooks [70, 181, 21]; the interested reader is encouraged to consult these works for further details on the topic.

3.1.1 Kinematics

For the derivation of the kinematical relations describing the state of a deformable structure under the effect of large displacements and large rotations, we consider a body $\Omega \subset \mathbb{R}^d$ in d-dimensional Euclidean space $\mathbb{R}^d$ with a sufficiently regular boundary $\Gamma := \partial\Omega \subset \mathbb{R}^{d-1}$. In the following, let us restrict to $d = 3$, which certainly is the most relevant case for the numerical investigations carried out in the remainder of this work. In contrast to fluid mechanics, where an Eulerian or, if moving domains are considered, ALE description of motion is the most natural choice, a Lagrangian description is most commonly adopted in structural mechanics. Here, we introduce the reference configuration $\Omega_{\boldsymbol{X}}$ and the current configuration $\Omega_{\boldsymbol{x}}$ as depicted in Figure 3.1. By means of the nonlinear bijective mapping [39, p. 3]

$$\begin{aligned} \boldsymbol{\varphi} : \Omega_{\boldsymbol{X}} \times [t_0, T) &\to \Omega_{\boldsymbol{x}} \times [t_0, T) \\ (\boldsymbol{X}, t) &\mapsto \boldsymbol{\varphi}(\boldsymbol{X}, t) = (\boldsymbol{x}, t) \,, \end{aligned} \tag{3.1}$$

we relate the reference configuration $\Omega_{\boldsymbol{X}}$ at time $t = t_0$ to the current configuration $\Omega_{\boldsymbol{x}}$ at time $t \geq t_0$. In other words, the mapping $\boldsymbol{\varphi}$ transforms the material point $\boldsymbol{X}$ in the reference configuration $\Omega_{\boldsymbol{X}}$ to the point $\boldsymbol{x}$ in the current configuration $\Omega_{\boldsymbol{x}}$. Adopting

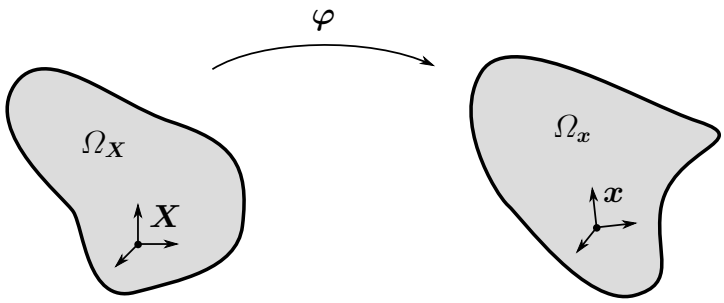


Figure 3.1: Reference configuration Ω_X and current configuration Ω_x in the Lagrangian description of motion [39, p. 3].

the notation in the standard textbooks of solid mechanics (see, e.g., [181, 21]), we use uppercase variables for quantities referring to the reference configuration, while lowercase variables are used for those related to the current configuration.

In order to characterize local deformation, we consider the transformation of the infinitesimal line element $\mathrm{d}\boldsymbol{X}$ from the reference configuration $\Omega_{\boldsymbol{X}}$ to the line element $\mathrm{d}\boldsymbol{x}$ in the current configuration $\Omega_{\boldsymbol{x}}$. The partial derivative

$$\boldsymbol{F} = \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{X}} \tag{3.2}$$

is the deformation gradient. Because $\boldsymbol{F}$ is regular, its inverse is well-defined and can hence be applied to obtain

$$\mathrm{d}\boldsymbol{X} = \boldsymbol{F}^{-1}\mathrm{d}\boldsymbol{x} \ . \tag{3.3}$$

Nanson's formula covers the change of a directed area element $\mathrm{d}\boldsymbol{A}$:

$$\mathrm{d}\boldsymbol{a} = \boldsymbol{n}\mathrm{d}a = J\boldsymbol{F}^{-\mathrm{T}}\boldsymbol{N}\mathrm{d}A = J\boldsymbol{F}^{-\mathrm{T}}\mathrm{d}\boldsymbol{A} \ , \tag{3.4}$$

which involves the Jacobian determinant

$$J = \det \boldsymbol{F} \neq 0 \ . \tag{3.5}$$

Note that the condition $J \neq 0$ holds due to the fact that $\boldsymbol{\varphi}$ represents an invertible mapping. The relation (3.4) can also be expressed using the cofactor

$$\mathrm{Cof}\,\boldsymbol{F} := J\boldsymbol{F}^{-\mathrm{T}} \tag{3.6}$$

of the deformation gradient $\boldsymbol{F}$ and then becomes

$$\mathrm{d}\boldsymbol{a} = \mathrm{Cof}\,\boldsymbol{F}\mathrm{d}\boldsymbol{A} \ . \tag{3.7}$$

Lastly, the transformation of an infinitesimal volume element $\mathrm{d}v$ from the reference to the current configuration is described by

$$\mathrm{d}v = J\mathrm{d}V \ . \tag{3.8}$$

For the difference between the current position of a material point $\boldsymbol{x} = \boldsymbol{\varphi}(\boldsymbol{X}, t)$ in the current configuration and the position $\boldsymbol{X}$ in the reference configuration, we introduce the displacement vector

$$\boldsymbol{d}(\boldsymbol{X}, t) = \boldsymbol{x}(\boldsymbol{X}, t) - \boldsymbol{X} \ . \tag{3.9}$$

Denoting the second-order identity tensor by $\boldsymbol{I}$ and recalling the definition of the deformation gradient $\boldsymbol{F}$ in Equation (3.2), the gradient of the displacement $\boldsymbol{d}(\boldsymbol{X}, t)$ is computed as

$$\boldsymbol{H} = \operatorname{Grad} \boldsymbol{d}(\boldsymbol{X}, t) = \boldsymbol{F} - \boldsymbol{I} \,, \tag{3.10}$$

where Grad denotes the gradient operator with respect to the reference configuration. The deformation gradient $\boldsymbol{F}$ can be decomposed into an orthogonal rotation tensor $\boldsymbol{R}$ and a symmetric stretch tensor $\boldsymbol{U}$ or $\boldsymbol{V}$, respectively, such that

$$\boldsymbol{F} = \boldsymbol{R}\boldsymbol{U} = \boldsymbol{V}\boldsymbol{R} \,. \tag{3.11}$$

The right stretch tensor $\boldsymbol{U}$ and the left stretch tensor $\boldsymbol{V}$ are related by the polar decompositions

$$\boldsymbol{U} = \boldsymbol{R}^{\mathrm{T}}\boldsymbol{V}\boldsymbol{R} \quad \text{and} \quad \boldsymbol{V} = \boldsymbol{R}\boldsymbol{U}\boldsymbol{R}^{\mathrm{T}} \,. \tag{3.12}$$

In order to formulate the constitutive relations in Section 3.1.3, suitable strain measures must be introduced. For a proper description of the local deformation, these measures should be independent of rigid body motions. A strain tensor that refers to the reference configuration is the Green-Lagrange strain tensor

$$\boldsymbol{E} = \frac{1}{2}\left(\boldsymbol{F}^{\mathrm{T}}\boldsymbol{F} - \boldsymbol{I}\right) \,. \tag{3.13}$$

Introducing the right Cauchy-Green deformation tensor

$$\boldsymbol{C} = \boldsymbol{F}^{\mathrm{T}}\boldsymbol{F} \,, \tag{3.14}$$

the Green-Lagrange strain tensor (3.13) reads

$$\boldsymbol{E} = \frac{1}{2}\left(\boldsymbol{C} - \boldsymbol{I}\right) \,. \tag{3.15}$$

In the small strain case, the Green-Lagrange strain tensor reduces to the linear strain measure

$$\boldsymbol{\varepsilon} = \frac{1}{2}\left(\boldsymbol{H} + \boldsymbol{H}^{\mathrm{T}}\right) \tag{3.16}$$

with the displacement gradient $\boldsymbol{H}$ from Equation (3.10). The invariants of the right Cauchy-Green deformation tensor $\boldsymbol{C}$ are given as [181, p. 515]

$$\mathrm{I}_{\boldsymbol{C}} = \operatorname{tr} \boldsymbol{C} \,, \quad \mathrm{II}_{\boldsymbol{C}} = \frac{1}{2}\left(\operatorname{tr}^2 \boldsymbol{C} - \operatorname{tr} \boldsymbol{C}^2\right) \,, \quad \text{and } \mathrm{III}_{\boldsymbol{C}} = \det \boldsymbol{C} \,. \tag{3.17}$$

In addition to the right Cauchy-Green deformation tensor, let us also introduce the left Cauchy-Green deformation tensor

$$\boldsymbol{B} = \boldsymbol{F}\boldsymbol{F}^{\mathrm{T}} \,, \tag{3.18}$$

which, due to its definition, has the same invariants as the right Cauchy-Green deformation tensor $\boldsymbol{C}$ [181, p. 42]. Later on, it will turn out useful to perform a multiplicative split of the deformation gradient $\boldsymbol{F}$ into a dilatational (i.e, volume-changing or volumetric) part $\hat{\boldsymbol{F}}$ and a distortional (i.e., volume-preserving or isochoric) component $\bar{\boldsymbol{F}}$, such that [50]

$$\boldsymbol{F} = \hat{\boldsymbol{F}}\bar{\boldsymbol{F}} \,. \tag{3.19}$$

For the dilatational component, we have $\hat{\boldsymbol{F}} = J^{1/3}\boldsymbol{I}$, whereas the distortional part amounts to $\bar{\boldsymbol{F}} = J^{-1/3}\boldsymbol{F}$. Building on this, the right Cauchy-Green deformation tensor $\boldsymbol{C}$ is decomposed into the dilatational part $\hat{\boldsymbol{C}}$ and the distortional component $\bar{\boldsymbol{C}}$, such that

$$\boldsymbol{C} = \hat{\boldsymbol{C}}\bar{\boldsymbol{C}} \,, \tag{3.20}$$

where $\hat{\boldsymbol{C}} = J^{2/3}\boldsymbol{I}$ and $\bar{\boldsymbol{C}} = J^{-2/3}\boldsymbol{C}$. In the forthcoming, we will also need the invariants

$$\begin{aligned} \mathrm{I}_{\bar{C}} &= \operatorname{tr}\bar{\boldsymbol{C}} = J^{-2/3}\operatorname{tr}\boldsymbol{C} = J^{-2/3}\mathrm{I}_{\boldsymbol{C}} \,, \\ \mathrm{II}_{\bar{C}} &= \frac{1}{2}\left(\operatorname{tr}^2\bar{\boldsymbol{C}} - \operatorname{tr}\bar{\boldsymbol{C}}^2\right) = \frac{1}{2}\left(\left(J^{-2/3}\operatorname{tr}\boldsymbol{C}\right)^2 - \operatorname{tr}(J^{-4/3}\boldsymbol{C}^2)\right) = J^{-4/3}\mathrm{II}_{\boldsymbol{C}} \,, \\ \mathrm{III}_{\bar{C}} &= \det\bar{\boldsymbol{C}} = J^{-6/3}\det\boldsymbol{C} = J^{-2}\mathrm{III}_{\boldsymbol{C}} = J^{-2}J^2 = 1 \,, \end{aligned} \tag{3.21}$$

of the distortional part $\bar{\boldsymbol{C}}$ of the right Cauchy-Green deformation tensor $\boldsymbol{C}$. Similar to (3.20), the left Cauchy-Green tensor $\boldsymbol{B}$ can be written as

$$\boldsymbol{B} = \hat{\boldsymbol{B}}\bar{\boldsymbol{B}} \tag{3.22}$$

using $\hat{\boldsymbol{B}} = J^{2/3}\boldsymbol{I}$ and $\bar{\boldsymbol{B}} = J^{-2/3}\boldsymbol{B}$.

For the description of the transient nature of motion, we introduce the velocity vector

$$\boldsymbol{v}(\boldsymbol{X},t) = \left.\frac{\partial\boldsymbol{x}}{\partial t}\right|_{\boldsymbol{X}}(\boldsymbol{X},t) = \frac{\mathrm{D}\boldsymbol{x}}{\mathrm{D}t}(\boldsymbol{X},t) = \dot{\boldsymbol{x}}(\boldsymbol{X},t) \,. \tag{3.23}$$

In accordance with Chapter 2, the derivative $\mathrm{D}\cdot/\mathrm{D}t$ indicates the material time derivative. Differentiating once more with respect to time produces the acceleration vector

$$\boldsymbol{a} = \dot{\boldsymbol{v}}(\boldsymbol{X},t) = \ddot{\boldsymbol{x}}(\boldsymbol{X},t) \,. \tag{3.24}$$

3.1.2 Balance Equations

For the derivation of the balance equations, we assume that the mass m of the body under consideration does not change with time, which is conveyed by

$$\frac{\mathrm{D}m}{\mathrm{D}t} = \dot{m} = 0 \,. \tag{3.25}$$

In terms of the density ρ, this relation is equivalently written as

$$m = \int_{\Omega_{\boldsymbol{x}}} \rho(\boldsymbol{x},t)\,\mathrm{d}v = \int_{\Omega_{\boldsymbol{X}}} \rho_{\boldsymbol{X}}(\boldsymbol{X})\,\mathrm{d}V = \text{const.} \tag{3.26}$$

Evidently, the balance of mass (3.26) must also hold for an infinitesimal mass element and, with $\mathrm{d}m(\boldsymbol{x}) = \rho\mathrm{d}v$ and $\mathrm{d}m(\boldsymbol{X}) = \rho_{\boldsymbol{X}}\mathrm{d}V$, leads us to

$$\rho\mathrm{d}v = \rho_{\boldsymbol{X}}\mathrm{d}V \,. \tag{3.27}$$

Recalling the transformation rule (3.8) from the infinitesimal volume element $\mathrm{d}V$ in the reference configuration to the volume element $\mathrm{d}v$ in the current configuration, Equation (3.27)

gives us a relation between the density $\rho_{\boldsymbol{X}}$ in the reference configuration and the density $\rho = \rho_{\boldsymbol{x}}$ in the current configuration:

$$\rho_{\boldsymbol{X}} = J\rho \; . \tag{3.28}$$

From Newton's second law of motion, we learn that the change in linear momentum $\boldsymbol{p}(t)$ equals the sum of external forces $\boldsymbol{f}(t)$ acting on the body:

$$\frac{\mathrm{D}\boldsymbol{p}}{\mathrm{D}t}(t) = \int_{\Omega_{\boldsymbol{x}}} \rho \frac{\mathrm{D}\boldsymbol{v}}{\mathrm{D}t} \, \mathrm{d}v = \int_{\Omega_{\boldsymbol{x}}} \rho \dot{\boldsymbol{v}} \, \mathrm{d}v = \boldsymbol{f}(t) \; , \tag{3.29}$$

where the external forces $\boldsymbol{f}(t)$ embody the surface traction $\boldsymbol{t}$ integrated over the surface $\Gamma_{\boldsymbol{x}}$ and the body force $\rho\boldsymbol{b}$ integrated over the volume $\Omega_{\boldsymbol{x}}$:

$$\boldsymbol{f}(t) = \int_{\Gamma_{\boldsymbol{x}}} \boldsymbol{t} \, \mathrm{d}a + \int_{\Omega_{\boldsymbol{x}}} \rho \boldsymbol{b} \, \mathrm{d}v \; . \tag{3.30}$$

Cauchy's theorem states that

$$\boldsymbol{t} = \boldsymbol{\sigma}\boldsymbol{n} \tag{3.31}$$

and, thus, provides a relation between the traction vector $\boldsymbol{t}$, the Cauchy stress tensor $\boldsymbol{\sigma}$, and the outer surface normal $\boldsymbol{n}$. Building on this, the balance of linear momentum (3.29) is restated as

$$\int_{\Omega_{\boldsymbol{x}}} \rho \dot{\boldsymbol{v}} \, \mathrm{d}v = \int_{\Gamma_{\boldsymbol{x}}} \boldsymbol{\sigma}\boldsymbol{n} \, \mathrm{d}a + \int_{\Omega_{\boldsymbol{x}}} \rho \boldsymbol{b} \, \mathrm{d}v \; . \tag{3.32}$$

Clearly, Equation (3.32) must also hold for a differential volume $\mathrm{d}v$, which, after employing Gauss's divergence theorem, leads us to the differential form of mechanical equilibrium in the current configuration

$$\rho \dot{\boldsymbol{v}} = \operatorname{div} \boldsymbol{\sigma} + \rho \boldsymbol{b} \; . \tag{3.33}$$

In order to express Equation (3.32) in the reference configuration, we first apply Nanson's formula to obtain

$$\int_{\Gamma_{\boldsymbol{x}}} \boldsymbol{\sigma}\boldsymbol{n} \, \mathrm{d}a = \int_{\Gamma_{\boldsymbol{X}}} \boldsymbol{\sigma} J \boldsymbol{F}^{-\mathrm{T}} \boldsymbol{N} \, \mathrm{d}A = \int_{\Gamma_{\boldsymbol{X}}} \boldsymbol{P}\boldsymbol{N} \, \mathrm{d}A \; . \tag{3.34}$$

Here, we introduced the first Piola-Kirchhoff stress tensor $\boldsymbol{P}$, which is related to the Cauchy stress tensor $\boldsymbol{\sigma}$ by

$$\boldsymbol{P} = J\boldsymbol{\sigma}\boldsymbol{F}^{-\mathrm{T}} \; . \tag{3.35}$$

Note that this tensor is unsymmetric. In the following, it will be more convenient to work with a symmetric stress tensor. By premultiplying (3.35) from the left by $\boldsymbol{F}^{-1}$, we obtain the second Piola-Kirchhoff stress tensor

$$\boldsymbol{S} = \boldsymbol{F}^{-1}\boldsymbol{P} = J\boldsymbol{F}^{-1}\boldsymbol{\sigma}\boldsymbol{F}^{-\mathrm{T}} \; , \tag{3.36}$$

which is symmetric. Recalling the transformation rule (3.27) and using (3.34), Equation (3.32) becomes

$$\int_{\Omega_{\boldsymbol{X}}} \rho_{\boldsymbol{X}} \dot{\boldsymbol{v}} \, \mathrm{d}V = \int_{\Gamma_{\boldsymbol{X}}} \boldsymbol{P}\boldsymbol{N} \, \mathrm{d}A + \int_{\Omega_{\boldsymbol{X}}} \rho_{\boldsymbol{X}} \boldsymbol{b} \, \mathrm{d}V \; . \tag{3.37}$$

In differential form, we have

$$\rho_{\boldsymbol{X}} \dot{\boldsymbol{v}} = \operatorname{Div} \boldsymbol{P} + \rho_{\boldsymbol{X}} \boldsymbol{b} \; , \tag{3.38}$$

or, with the second Piola-Kirchhoff stress tensor introduced in (3.36),

$$\rho_{\boldsymbol{X}} \dot{\boldsymbol{v}} = \mathrm{Div}\,(\boldsymbol{F}\boldsymbol{S}) + \rho_{\boldsymbol{X}} \boldsymbol{b}\,. \tag{3.39}$$

In addition to the balance of linear momentum (3.32), angular momentum must be conserved as well. It can be shown that the balance of angular momentum implies

$$\boldsymbol{\sigma} = \boldsymbol{\sigma}^{\mathrm{T}} \tag{3.40}$$

and, thus, the symmetry of the Cauchy stress tensor $\boldsymbol{\sigma}$. Equation (3.40) is commonly referred to as *Cauchy's second equation of motion* [149, p. 154].

Summarizing the above, our initial-boundary value problem, formulated in the reference configuration, reads as follows:

$$\rho_{\boldsymbol{X}} \dot{\boldsymbol{v}} = \mathrm{Div}\,(\boldsymbol{F}\boldsymbol{S}) + \rho_{\boldsymbol{X}} \boldsymbol{b} \qquad \text{in } \Omega_{\boldsymbol{X}} \tag{3.41}$$

$$\boldsymbol{d} = \bar{\boldsymbol{d}} \qquad \text{on } \Gamma_{\boldsymbol{d},\boldsymbol{X}} \tag{3.42}$$

$$\boldsymbol{F}\boldsymbol{S}\boldsymbol{N} = \bar{\boldsymbol{t}} \qquad \text{on } \Gamma_{\boldsymbol{t},\boldsymbol{X}} \tag{3.43}$$

$$\boldsymbol{v} = \boldsymbol{v}_0 \qquad \text{in } \Omega_{\boldsymbol{X}} \text{ at } t = t_0 \tag{3.44}$$

To make the problem well-posed, the mechanical equilibrium (3.41) was augmented by the displacement boundary condition (3.42) on the Dirichlet boundary $\Gamma_{\boldsymbol{d},\boldsymbol{X}} := \partial\Omega_{\boldsymbol{d},\boldsymbol{X}}$ and the traction boundary condition (3.43) on the Neumann boundary $\Gamma_{\boldsymbol{t},\boldsymbol{X}} := \partial\Omega_{\boldsymbol{t},\boldsymbol{X}}$ as well as the initial condition (3.44) for the velocity $\boldsymbol{v}(\boldsymbol{X}, t_0)$ at time $t = t_0$. Note that, for a transient problem, the Dirichlet boundary $\Gamma_{\boldsymbol{d},\boldsymbol{X}}$ may also be empty.

3.1.3 Constitutive Relations

To relate the stress inside the body to its deformation, a proper description of the specific material behavior must be found. In this work, we focus on elastic materials under the assumption of Green elasticity; the theoretical foundations of elasto-plastic, visco-elastic, and visco-plastic material behavior are omitted for the sake of conciseness. Instead, the interested reader is referred to the standard textbooks [181, 21], where such material laws are discussed in sufficient detail. For hyperelastic materials, the constitutive relation can be derived from a scalar potential W representing the strain energy stored in the body. Following [107, 122, 161, 181], the second Piola-Kirchhoff stress tensor $\boldsymbol{S}$ is given as the derivative of the strain energy density W with respect to the Green-Lagrange strain tensor $\boldsymbol{E}$ such that

$$\boldsymbol{S} = \frac{\partial W}{\partial \boldsymbol{E}} = 2\frac{\partial W}{\partial \boldsymbol{C}}\,. \tag{3.45}$$

Differentiating once more with respect to $\boldsymbol{C}$ produces the fourth-order material tensor

$$\mathcal{C} = 2\frac{\partial \boldsymbol{S}}{\partial \boldsymbol{C}} = 4\frac{\partial^2 W}{\partial \boldsymbol{C}\partial \boldsymbol{C}}\,. \tag{3.46}$$

In the small strain case, an appropriate description of the material behavior is provided by the St. Venant-Kirchhoff model

$$W(\boldsymbol{E}) = \frac{\lambda}{2}\,\mathrm{tr}^2\,\boldsymbol{E} + \mu \boldsymbol{E}^2\,, \tag{3.47}$$

where λ and μ are the Lamé parameters. This material model represents an extension of a linear-elastic material behavior to the case of large displacements and large rotations. Inserting (3.47) into the relations (3.45) and (3.46) for the second Piola-Kirchhoff stress tensor $\boldsymbol{S}$ and the material tensor $\boldsymbol{\mathcal{C}}$ yields

$$\boldsymbol{S} = \lambda \operatorname{tr}(\boldsymbol{E})\boldsymbol{I} + 2\mu\boldsymbol{E} \,, \tag{3.48}$$

$$\boldsymbol{\mathcal{C}} = \lambda \boldsymbol{I} \otimes \boldsymbol{I} + 2\mu\boldsymbol{\mathcal{I}} \,, \tag{3.49}$$

where $\boldsymbol{I} \otimes \boldsymbol{I}$ is the dyadic product of the second-order identity tensor with itself, while $\boldsymbol{\mathcal{I}}$ denotes the fourth-order identity tensor.

Once the body exhibits large deformations, the St. Venant-Kirchhoff model is no longer appropriate. On the contrary, it may cause material instabilities and lead to unrealistic and nonphysical stresses in the large strain regime. Hence, if strains become large, more elaborated constitutive models need to be applied. When considering the finite elasticity of rubber or foam materials, it is convenient to split the strain energy density W into a sum of a distortional part $\bar{W}(\bar{\boldsymbol{C}})$ and a dilatational part $U(J)$:

$$W(\bar{\boldsymbol{C}}, J) = \bar{W}(\bar{\boldsymbol{C}}) + U(J) \,. \tag{3.50}$$

For the distortional part, we assume a polynomial sum of the form

$$\bar{W}(\bar{\boldsymbol{C}}) = \sum_{i,j=0}^{n} C_{ij}(\mathrm{I}_{\bar{C}} - 3)^i(\mathrm{II}_{\bar{C}} - 3)^j \,, \tag{3.51}$$

where the coefficients C_{ij} are material constants and $\mathrm{I}_{\bar{C}}, \mathrm{II}_{\bar{C}}$ are the first and second invariant of the distortional component $\bar{\boldsymbol{C}}$ of the right Cauchy-Green deformation tensor from Equation (3.21). The dilatational component $U(J)$ can be interpreted as a penalty term to enforce the incompressibility constraint, and it is computed from the sum

$$U(J) = \sum_{k=1}^{m} D_k(J-1)^{2k} \tag{3.52}$$

with the constants D_k and the Jacobian determinant J from Equation (3.5). From the general forms (3.51) and (3.52), we recover, for instance, the compressible Neo-Hooke model [137] choosing $n = 0$, $m = 1$, and $C_{01} = C_{11} = 0$:

$$W(\bar{\boldsymbol{C}}, J) = C_{10}(\mathrm{I}_{\bar{C}} - 3) + D_1(J-1)^2 \,. \tag{3.53}$$

For reasons of consistency with small-strain theory, it holds that $C_{10} = \mu/2$ and $D_1 = \kappa/2$, where μ is the shear modulus and κ denotes the bulk modulus [8, p. 162]. Setting $n = 1$, $C_{01} = C_2$, $C_{11} = 0$, $C_{10} = C_1$, and $m = 1$ produces the compressible Mooney-Rivlin model [116, 137, 181, p. 43]

$$W(\bar{\boldsymbol{C}}, J) = C_{10}(\mathrm{I}_{\bar{C}} - 3) + C_{01}(\mathrm{II}_{\bar{C}} - 3) + D_1(J-1)^2 \,. \tag{3.54}$$

Based on (3.51) and (3.52), various other constitutive models can be derived. The constants C_{ij} and D_k can be determined by parameter identification to experimental measurements, for instance.

For a strain energy density function of the kind (3.50), the second Piola-Kirchhoff stress tensor reads

$$\boldsymbol{S} = 2\frac{\partial W}{\partial \boldsymbol{C}} = 2\frac{\partial \bar{W}}{\partial \bar{\boldsymbol{C}}}\frac{\partial \bar{\boldsymbol{C}}}{\partial \boldsymbol{C}} + 2\frac{\partial U}{\partial J}\frac{\partial J}{\partial \boldsymbol{C}} , \tag{3.55}$$

where the derivative $\partial J/\partial \boldsymbol{C}$ evaluates to

$$\frac{\partial J}{\partial \boldsymbol{C}} = \frac{\partial \sqrt{\det \boldsymbol{C}}}{\partial \boldsymbol{C}} = \frac{1}{2} J \boldsymbol{C}^{-1} \tag{3.56}$$

and the derivative $\partial \bar{\boldsymbol{C}}/\partial \boldsymbol{C}$ is computed from

$$\frac{\partial \bar{\boldsymbol{C}}}{\partial \boldsymbol{C}} = \frac{\partial \left(J^{-2/3} \boldsymbol{C} \right)}{\partial \boldsymbol{C}} = \frac{\partial J^{-2/3}}{\partial \boldsymbol{C}} \boldsymbol{C} + J^{-2/3} \frac{\partial \boldsymbol{C}}{\partial \boldsymbol{C}} = J^{-2/3} \left(\boldsymbol{\mathcal{I}} - \frac{1}{3} \boldsymbol{C}^{-1} \boldsymbol{C} \right) . \tag{3.57}$$

3.1.4 Variational Formulation and Linearization

Equation (3.41) stipulates mechanical equilibrium in a pointwise sense and is therefore commonly referred to as the *strong form* of dynamic equilibrium. If an approximation $\boldsymbol{d}_h$ instead of the exact solution $\boldsymbol{d}$ is used in (3.41), a residual $\boldsymbol{r}$ will remain, indicating the error in not fulfilling the balance equation, such that

$$\operatorname{Div} \boldsymbol{P}(\boldsymbol{d}_h) + \rho_{\boldsymbol{X}} \boldsymbol{b} - \rho_{\boldsymbol{X}} \dot{\boldsymbol{v}}_h = \boldsymbol{r} . \tag{3.58}$$

In what follows, the residual $\boldsymbol{r}$ will be reduced to zero in a weak sense by multiplying (3.58) by a test function $\boldsymbol{\eta}$ and by integrating over the domain $\Omega_{\boldsymbol{X}}$. The test functions $\boldsymbol{\eta}$ stem from the space

$$\boldsymbol{H}_0^1(\Omega_{\boldsymbol{X}}; \mathbb{R}^d) := \left\{ \boldsymbol{\eta}(\boldsymbol{X}) \in \boldsymbol{H}^1(\Omega_{\boldsymbol{X}}; \mathbb{R}^d) \middle| \boldsymbol{\eta} = \boldsymbol{0} \text{ on } \Gamma_{\boldsymbol{d},\boldsymbol{X}} \right\} , \tag{3.59}$$

which represents the restriction of the space

$$\boldsymbol{H}^1(\Omega_{\boldsymbol{X}}; \mathbb{R}^d) := \left\{ \boldsymbol{\eta}(\boldsymbol{X}) \in \boldsymbol{L}^2(\Omega_{\boldsymbol{X}}; \mathbb{R}^d) \middle| \frac{\partial \boldsymbol{\eta}}{\partial X}, \frac{\partial \boldsymbol{\eta}}{\partial Y}, \frac{\partial \boldsymbol{\eta}}{\partial Z} \in \boldsymbol{L}^2(\Omega_{\boldsymbol{X}}; \mathbb{R}^d) \right\} \tag{3.60}$$

to the set of functions vanishing on the Dirichlet boundary $\Gamma_{\boldsymbol{d},\boldsymbol{X}}$. As usual, $\boldsymbol{L}^2(\Omega_{\boldsymbol{X}}; \mathbb{R}^d)$ denotes the space of component-wise Lebesgue-measurable functions with integrable squares over $\Omega_{\boldsymbol{X}}$. Multiplying (3.58) by a test function $\boldsymbol{\eta} \in \boldsymbol{H}_0^1(\Omega_{\boldsymbol{X}}; \mathbb{R}^d)$ leads us to

$$\int_{\Omega_{\boldsymbol{X}}} \operatorname{Div} \boldsymbol{P}(\boldsymbol{d}_h) \cdot \boldsymbol{\eta} \, \mathrm{d}V + \int_{\Omega_{\boldsymbol{X}}} \rho_{\boldsymbol{X}} (\boldsymbol{b} - \dot{\boldsymbol{v}}_h) \cdot \boldsymbol{\eta} \, \mathrm{d}V = 0 . \tag{3.61}$$

Evidently, this equation must also hold for the exact solution $\boldsymbol{d}$:

$$\int_{\Omega_{\boldsymbol{X}}} \operatorname{Div} \boldsymbol{P}(\boldsymbol{d}) \cdot \boldsymbol{\eta} \, \mathrm{d}V + \int_{\Omega_{\boldsymbol{X}}} \rho_{\boldsymbol{X}} (\boldsymbol{b} - \dot{\boldsymbol{v}}) \cdot \boldsymbol{\eta} \, \mathrm{d}V = 0 . \tag{3.62}$$

Integrating the first term by parts, applying Gauss's divergence theorem and introducing the traction boundary condition (3.43) results in

$$G(\boldsymbol{\varphi}, \boldsymbol{\eta}) = \int_{\Omega_{\boldsymbol{X}}} \boldsymbol{P} \cdot \operatorname{Grad} \boldsymbol{\eta} \, \mathrm{d}V - \int_{\Omega_{\boldsymbol{X}}} \rho_{\boldsymbol{X}} (\boldsymbol{b} - \dot{\boldsymbol{v}}) \cdot \boldsymbol{\eta} \, \mathrm{d}V - \int_{\Gamma_{\boldsymbol{t},\boldsymbol{X}}} \bar{\boldsymbol{t}} \cdot \boldsymbol{\eta} \, \mathrm{d}A = 0 , \tag{3.63}$$

which represents the *weak form* of linear momentum, also known as the *principle of virtual work*. Noting that the first Piola-Kirchhoff stress tensor $\boldsymbol{P}$ can be substituted by $\boldsymbol{P} = \boldsymbol{F}\boldsymbol{S}$, the expression $\boldsymbol{P} \cdot \operatorname{Grad} \boldsymbol{\eta}$ may be recast as

$$\boldsymbol{P} \cdot \operatorname{Grad} \boldsymbol{\eta} = \boldsymbol{S} \cdot \boldsymbol{F}^{\mathrm{T}} \operatorname{Grad} \boldsymbol{\eta} = \boldsymbol{S} \cdot \frac{1}{2} \left(\boldsymbol{F}^{\mathrm{T}} \operatorname{Grad} \boldsymbol{\eta} + \operatorname{Grad}^{\mathrm{T}} \boldsymbol{\eta} \boldsymbol{F} \right) = \boldsymbol{S} \cdot \delta \boldsymbol{E} \, , \tag{3.64}$$

where $\delta \boldsymbol{E}$ denotes the variation of the Green-Lagrange strain tensor. It is obtained from the directional derivative

$$\begin{aligned} \mathrm{D}\boldsymbol{E} \cdot \boldsymbol{\eta} &= \frac{\mathrm{d}}{\mathrm{d}\alpha} \frac{1}{2} \left(\boldsymbol{F}^{\mathrm{T}}(\boldsymbol{\varphi} + \alpha\boldsymbol{\eta}) \boldsymbol{F}(\boldsymbol{\varphi} + \alpha\boldsymbol{\eta}) - \boldsymbol{1} \right) \Big|_{\alpha=0} \\ &= \frac{\mathrm{d}}{\mathrm{d}\alpha} \frac{1}{2} \left((\operatorname{Grad}(\boldsymbol{\varphi} + \alpha\boldsymbol{\eta}))^{\mathrm{T}} \operatorname{Grad}(\boldsymbol{\varphi} + \alpha\boldsymbol{\eta}) - \boldsymbol{1} \right) \Big|_{\alpha=0} \\ &= \frac{1}{2} \left((\operatorname{Grad} \boldsymbol{\eta})^{\mathrm{T}} \boldsymbol{F} + \boldsymbol{F}^{\mathrm{T}} \operatorname{Grad} \boldsymbol{\eta} \right) = \delta \boldsymbol{E} \, . \end{aligned} \tag{3.65}$$

This allows us to reformulate (3.63) as

$$G(\boldsymbol{\varphi}, \boldsymbol{\eta}) = \int_{\Omega_{\boldsymbol{X}}} \boldsymbol{S} \cdot \delta \boldsymbol{E} \, \mathrm{d}V - \int_{\Omega_{\boldsymbol{X}}} \rho_{\boldsymbol{X}} (\boldsymbol{b} - \dot{\boldsymbol{v}}) \cdot \boldsymbol{\eta} \, \mathrm{d}V - \int_{\Gamma_{\boldsymbol{t},\boldsymbol{X}}} \bar{\boldsymbol{t}} \cdot \boldsymbol{\eta} \, \mathrm{d}A = 0 \, . \tag{3.66}$$

In Section 3.1.5, it will be shown that the discretization of the weak form (3.63) or (3.66) results in a set of nonlinear algebraic equations. For the solution of this system of equations, we usually employ the Newton-Rapshon procedure by performing a Taylor series expansion about an already computed approximate solution. In the vicinity of the solution, the Newton-Rapshon method will converge quadratically. The Taylor expansion corresponds to a linearization of the weak form about a state of equilibrium $\bar{\boldsymbol{\varphi}}$. The linear part $L(G)_{\boldsymbol{\varphi}=\bar{\boldsymbol{\varphi}}}$ of the weak form $G(\boldsymbol{\varphi}, \boldsymbol{\eta})$ amounts to

$$L(G)_{\boldsymbol{\varphi}=\bar{\boldsymbol{\varphi}}} = G(\bar{\boldsymbol{\varphi}}, \boldsymbol{\eta}) + \mathrm{D}G(\bar{\boldsymbol{\varphi}}, \boldsymbol{\eta}) \cdot \Delta \boldsymbol{d} \, . \tag{3.67}$$

For the sake of simplicity, let us assume that the external surface and volume load do not depend on the state of deformation. In the context of FSI, this assumption is not valid, however, and the dependence of the external loads on the deformation must be included in the linearization. The reader is referred to [181, pp. 100–102, 142–148, 185, 65, 66] for details on this procedure. Under the assumption of deformation-independent loads, the directional derivative in (3.67) becomes

$$\mathrm{D}G(\bar{\boldsymbol{\varphi}}, \boldsymbol{\eta}) \cdot \Delta \boldsymbol{d} = \int_{\Omega_{\boldsymbol{X}}} (\mathrm{D}\boldsymbol{P}(\bar{\boldsymbol{\varphi}}) \cdot \Delta \boldsymbol{d}) \cdot \operatorname{Grad} \boldsymbol{\eta} \, \mathrm{d}V \, . \tag{3.68}$$

Recalling that $\boldsymbol{P} = \boldsymbol{F}\boldsymbol{S}$, the linearization of the first Piola-Kirchhoff stress tensor yields

$$\mathrm{D}G(\bar{\boldsymbol{\varphi}}, \boldsymbol{\eta}) \cdot \Delta \boldsymbol{d} = \int_{\Omega_{\boldsymbol{X}}} \left(\operatorname{Grad} \Delta \boldsymbol{d} \, \bar{\boldsymbol{S}} + \bar{\boldsymbol{F}} \, (\mathrm{D}\boldsymbol{S}(\bar{\boldsymbol{\varphi}}) \cdot \Delta \boldsymbol{d}) \right) \cdot \operatorname{Grad} \boldsymbol{\eta} \, \mathrm{d}V \, . \tag{3.69}$$

Quantities marked with a bar refer to the current deformation state $\bar{\boldsymbol{\varphi}}$. For the linearization of the second Piola-Kirchhoff stress tensor, we have

$$\mathrm{D}\boldsymbol{S}(\bar{\boldsymbol{\varphi}}) \cdot \Delta \boldsymbol{d} = \bar{\boldsymbol{\mathcal{C}}} \Delta \bar{\boldsymbol{E}} \, . \tag{3.70}$$

Here, $\bar{\boldsymbol{\mathcal{C}}}$ denotes the elasticity tensor, evaluated at $\boldsymbol{\varphi} = \bar{\boldsymbol{\varphi}}$:

$$\bar{\boldsymbol{\mathcal{C}}} = 4 \left. \frac{\partial^2 W}{\partial \boldsymbol{C} \partial \boldsymbol{C}} \right|_{\boldsymbol{\varphi}=\bar{\boldsymbol{\varphi}}} . \tag{3.71}$$

Inserting (3.71) into (3.70) and then into (3.69) leaves us with

$$\mathrm{D}G(\bar{\boldsymbol{\varphi}}, \boldsymbol{\eta}) \cdot \Delta \boldsymbol{d} = \int_{\Omega_{\boldsymbol{X}}} \left(\mathrm{Grad}\, \Delta \boldsymbol{d}\, \bar{\boldsymbol{S}} + \bar{\boldsymbol{F}} \bar{\boldsymbol{\mathcal{C}}} \Delta \bar{\boldsymbol{E}} \right) \cdot \mathrm{Grad}\, \boldsymbol{\eta}\, \mathrm{d}V , \tag{3.72}$$

or, more compactly,

$$\mathrm{D}G(\bar{\boldsymbol{\varphi}}, \boldsymbol{\eta}) \cdot \Delta \boldsymbol{d} = \int_{\Omega_{\boldsymbol{X}}} \mathrm{Grad}\, \Delta \boldsymbol{d} \bar{\boldsymbol{S}} \cdot \mathrm{Grad}\, \boldsymbol{\eta} + \delta \bar{\boldsymbol{E}} \cdot \bar{\boldsymbol{\mathcal{C}}} \Delta \bar{\boldsymbol{E}}\, \mathrm{d}V . \tag{3.73}$$

For the variation of the Green-Lagrange strain tensor, it holds that

$$\delta \bar{\boldsymbol{E}} = \frac{1}{2} \left(\bar{\boldsymbol{F}}^{\mathrm{T}} \mathrm{Grad}\, \boldsymbol{\eta} + \mathrm{Grad}^{\mathrm{T}}\, \boldsymbol{\eta} \bar{\boldsymbol{F}} \right) , \tag{3.74}$$

whereas the increment of the Green-Lagrange strain tensor takes the form

$$\Delta \bar{\boldsymbol{E}} = \frac{1}{2} \left(\bar{\boldsymbol{F}}^{\mathrm{T}} \mathrm{Grad}\, \Delta \boldsymbol{d} + \mathrm{Grad}^{\mathrm{T}}\, \Delta \boldsymbol{d} \bar{\boldsymbol{F}} \right) . \tag{3.75}$$

3.1.5 Spatial Discretization

In the FEM, we subdivide the body $\Omega_{\boldsymbol{X}}$ into a set of m non-overlapping finite elements $\Omega_\ell, \ell = 1, \ldots, m$ such that

$$\Omega_{\boldsymbol{X}} \approx \Omega_h = \bigcup_{\ell=1}^{m} \Omega_\ell . \tag{3.76}$$

In the general case of an arbitrarily curved geometry, Ω_h represents a discrete approximation of the original geometry $\Omega_{\boldsymbol{X}}$. For each of the finite elements, we interpolate the geometry by a linear combination of the coordinates $\boldsymbol{X}_{\ell,i}, i = 1, \ldots, n_\ell$ of the n_ℓ nodes of the element:

$$\boldsymbol{X}_\ell(\boldsymbol{\xi}) = \sum_{i=1}^{n_\ell} N_i(\boldsymbol{\xi}) \boldsymbol{X}_{\ell,i} . \tag{3.77}$$

The coefficients $N_i(\boldsymbol{\xi})$ in front of the nodal coordinates $\boldsymbol{X}_i$ are termed *shape functions* and are defined on a standard element $\hat{\Omega}$ in terms of the local coordinates $\boldsymbol{\xi}$. Depending on the element topology, the corresponding standard element $\hat{\Omega}$ and hence also the set of shape functions is different. Appendix A provides an excerpt of commonly used finite elements such as triangular and quadrilateral elements in the two-dimensional case or tetrahedral and hexahedral elements in the three-dimensional case. The transformation between the standard element $\hat{\Omega}$ and the element Ω is characterized by the Jacobian matrix

$$\boldsymbol{J}_\ell = \mathrm{Grad}_{\boldsymbol{\xi}}\, \boldsymbol{X}_\ell(\boldsymbol{\xi}) = \frac{\partial \boldsymbol{X}_\ell}{\partial \boldsymbol{\xi}}(\boldsymbol{\xi}) = \sum_{i=1}^{n_\ell} \boldsymbol{X}_{\ell,i} \otimes \frac{\partial N_i}{\partial \boldsymbol{\xi}}(\boldsymbol{\xi}) , \tag{3.78}$$

which relates the derivatives $\partial N_i / \partial \boldsymbol{\xi}$ and $\partial N_i / \partial \boldsymbol{X}$ with respect to the local coordinates $\boldsymbol{\xi}$ and the element coordinates $\boldsymbol{X}$, respectively. Based on (3.78), we obtain

$$\frac{\partial N_i}{\partial \boldsymbol{\xi}} = \frac{\partial \boldsymbol{X}_\ell}{\partial \boldsymbol{\xi}} \frac{\partial N_i}{\partial \boldsymbol{X}_\ell} = \boldsymbol{J}_\ell^{\mathrm{T}} \frac{\partial N_i}{\partial \boldsymbol{X}_\ell} \tag{3.79}$$

and the inverse relation

$$\frac{\partial N_i}{\partial \boldsymbol{X}_\ell} = \frac{\partial \boldsymbol{\xi}}{\partial \boldsymbol{X}_\ell}\frac{\partial N_i}{\partial \boldsymbol{\xi}} = \boldsymbol{J}_\ell^{-\mathrm{T}}\frac{\partial N_i}{\partial \boldsymbol{\xi}} \tag{3.80}$$

for $i = 1, \ldots, n_\ell$. Inserting (3.80) into the definition (3.2) for the deformation gradient $\boldsymbol{F}_\ell$ results in

$$\boldsymbol{F}_\ell = \operatorname{Grad}\boldsymbol{x}_\ell = \frac{\partial \boldsymbol{x}_\ell}{\partial \boldsymbol{X}_\ell} = \sum_{i=1}^{n_\ell} \boldsymbol{x}_i \otimes \frac{\partial N_i}{\partial \boldsymbol{X}_\ell} = \sum_{i=1}^{n_\ell} \boldsymbol{x}_i \otimes \left(\boldsymbol{J}_\ell^{-\mathrm{T}}\frac{\partial N_i}{\partial \boldsymbol{\xi}}\right) . \tag{3.81}$$

In this work, we shall adopt the isoparametric concept, which employs the same set of shape functions for the interpolation of the geometry as well as for the interpolation of the primary field variable. In the case of a structural problem, the primary field variable is the displacement $\boldsymbol{d}$. Hence, on element Ω_ℓ, the displacement and its gradient are interpolated as

$$\boldsymbol{d}(\boldsymbol{\xi}) = \sum_{i=1}^{n_\ell} N_i(\boldsymbol{\xi})\boldsymbol{d}_i = \boldsymbol{N}_\ell(\boldsymbol{\xi})\boldsymbol{d}_\ell \quad \text{and} \quad \operatorname{Grad}\boldsymbol{d} = \sum_{i=1}^{n_\ell} \boldsymbol{d}_i \otimes \frac{\partial N_i}{\partial \boldsymbol{X}_\ell}(\boldsymbol{\xi}) . \tag{3.82}$$

Here, the shape functions were gathered in the shape function vector $\boldsymbol{N}_\ell = \left(N_1, \ldots, N_{n_\ell}\right)$. Analogous to (3.82), the test function $\boldsymbol{\eta}$ and the corresponding derivative are interpolated as

$$\boldsymbol{\eta}(\boldsymbol{\xi}) = \sum_{i=1}^{n_\ell} N_i(\boldsymbol{\xi})\boldsymbol{\eta}_i = \boldsymbol{N}_\ell(\boldsymbol{\xi})\boldsymbol{\eta}_\ell \quad \text{and} \quad \operatorname{Grad}\boldsymbol{\eta} = \sum_{i=1}^{n_\ell} \boldsymbol{\eta}_i \otimes \frac{\partial N_i}{\partial \boldsymbol{X}_\ell}(\boldsymbol{\xi}) . \tag{3.83}$$

For the internal virtual work, we have

$$\delta W_\mathrm{i} = \int_{\Omega_{\boldsymbol{X}}} \boldsymbol{S} \cdot \delta\boldsymbol{E} \,\mathrm{d}\Omega \tag{3.84}$$

with the variation of the Green-Lagrange strain tensor $\delta\boldsymbol{E}$ from Equation (3.65). Recalling the relations for the discrete deformation gradient (3.81) and the discrete gradient of the test function (3.83), Equation (3.65), applied to a finite element Ω_ℓ, is recast as

$$\delta\boldsymbol{E}_\ell = \frac{1}{2}\sum_{i=1}^{n_\ell}\left(\boldsymbol{F}_\ell^{\mathrm{T}}\left(\boldsymbol{\eta}_i \otimes \frac{\partial N_i}{\partial \boldsymbol{X}_\ell}(\boldsymbol{\xi})\right) + \left(\frac{\partial N_i}{\partial \boldsymbol{X}_\ell}(\boldsymbol{\xi}) \otimes \boldsymbol{\eta}_i\right)\boldsymbol{F}_\ell\right) . \tag{3.85}$$

Taking advantage of the fact that the strain tensor is symmetric, the nine components of this second-order tensor can be condensed to Voigt notation, which, in the three-dimensional case, reads

$$\delta\boldsymbol{E}_\ell^{\mathrm{v}} = \begin{pmatrix}\delta E_{\ell,11} & \delta E_{\ell,22} & \delta E_{\ell,33} & 2\delta E_{\ell,12} & 2\delta E_{\ell,23} & 2\delta E_{\ell,13}\end{pmatrix}^{\mathrm{T}} . \tag{3.86}$$

It is straightforward to show that Equation (3.85) can be rewritten as

$$\delta\boldsymbol{E}_\ell^{\mathrm{v}} = \sum_{i=1}^{n_\ell}\boldsymbol{B}_i\boldsymbol{\eta}_i = \underbrace{\begin{pmatrix}\boldsymbol{B}_1 & \cdots & \boldsymbol{B}_{n_\ell}\end{pmatrix}}_{=\boldsymbol{B}_\ell}\underbrace{\begin{pmatrix}\boldsymbol{\eta}_1 \\ \vdots \\ \boldsymbol{\eta}_{n_\ell}\end{pmatrix}}_{=\boldsymbol{\eta}_\ell} , \tag{3.87}$$

where the nodal strain-displacement matrices $\boldsymbol{B}_i$ evaluate to

$$\boldsymbol{B}_i = \begin{pmatrix} F_{11}N_{i,X} & F_{21}N_{i,X} & F_{31}N_{i,X} \\ F_{12}N_{i,Y} & F_{22}N_{i,Y} & F_{32}N_{i,Y} \\ F_{13}N_{i,Z} & F_{23}N_{i,Z} & F_{33}N_{i,Z} \\ F_{11}N_{i,Y} + F_{12}N_{i,X} & F_{21}N_{i,Y} + F_{22}N_{i,X} & F_{31}N_{i,Y} + F_{32}N_{i,X} \\ F_{12}N_{i,Z} + F_{13}N_{i,Y} & F_{22}N_{i,Z} + F_{23}N_{i,Y} & F_{32}N_{i,Z} + F_{33}N_{i,Y} \\ F_{11}N_{i,Z} + F_{13}N_{i,X} & F_{21}N_{i,Z} + F_{23}N_{i,X} & F_{31}N_{i,Z} + F_{33}N_{i,X} \end{pmatrix} \tag{3.88}$$

and $N_{i,X}$ indicates the derivative of the ith shape function N_i with respect to X. Note that the $\boldsymbol{B}_i$ depend linearly on the displacement $\boldsymbol{d}$ as $\boldsymbol{F} = \boldsymbol{I} + \operatorname{Grad} \boldsymbol{d}$ holds for the deformation gradient. Rewriting the second Piola-Kirchhoff element stress tensor $\boldsymbol{S}_\ell$ in Voigt notation as $\boldsymbol{S}_\ell^{\mathrm{v}}$, we eventually approximate the variation δW_{i} of the internal virtual work as

$$\delta W_{\mathrm{i}} = \int_{\Omega_{\boldsymbol{X}}} \delta \boldsymbol{E} \cdot \boldsymbol{S} \,\mathrm{d}V \approx \bigcup_{\ell=1}^{m} \sum_{i=1}^{n_\ell} \boldsymbol{\eta}_i^{\mathrm{T}} \int_{\Omega_\ell} \boldsymbol{B}_\ell^{\mathrm{T}} \boldsymbol{S}_\ell^{\mathrm{v}} \,\mathrm{d}V = \bigcup_{\ell=1}^{m} \sum_{i=1}^{n_\ell} \boldsymbol{\eta}_i^{\mathrm{T}} \boldsymbol{r}_i(\boldsymbol{d}_\ell) = \boldsymbol{\eta}^{\mathrm{T}} \mathbf{r}(\mathbf{d}) \,. \tag{3.89}$$

For the variation of the external virtual work W_{e}, we have

$$\delta W_{\mathrm{e}} = -\int_{\Omega_{\boldsymbol{X}}} \rho_{\boldsymbol{X}} \ddot{\boldsymbol{v}} \cdot \boldsymbol{\eta} \,\mathrm{d}V + \int_{\Omega_{\boldsymbol{X}}} \rho_{\boldsymbol{X}} \boldsymbol{b} \cdot \boldsymbol{\eta} \,\mathrm{d}V + \int_{\Gamma_{\boldsymbol{t},\boldsymbol{X}}} \bar{\boldsymbol{t}} \cdot \boldsymbol{\eta} \,\mathrm{d}A \,. \tag{3.90}$$

The part related to the inertia terms amounts to

$$\int_{\Omega_{\boldsymbol{X}}} \rho_{\boldsymbol{X}} \ddot{\boldsymbol{v}} \cdot \boldsymbol{\eta} \,\mathrm{d}V \approx \bigcup_{\ell=1}^{m} \sum_{i=1}^{n_\ell} \sum_{j=1}^{n_\ell} \boldsymbol{\eta}_i^{\mathrm{T}} \int_{\Omega_\ell} \rho_{\boldsymbol{X}} N_i N_j \,\mathrm{d}V \ddot{\boldsymbol{v}}_j = \boldsymbol{\eta}^{\mathrm{T}} \mathbf{M} \ddot{\mathbf{v}} \,, \tag{3.91}$$

while the contribution from the external surface and body loads is

$$\begin{aligned} &\int_{\Omega_{\boldsymbol{X}}} \rho_{\boldsymbol{X}} \boldsymbol{b} \cdot \boldsymbol{\eta} \,\mathrm{d}V + \int_{\Gamma_{\boldsymbol{t},\boldsymbol{X}}} \bar{\boldsymbol{t}} \cdot \boldsymbol{\eta} \,\mathrm{d}A \\ &\qquad \approx \bigcup_{\ell=1}^{m} \sum_{i=1}^{n_\ell} \boldsymbol{\eta}_i^{\mathrm{T}} \int_{\Omega_\ell} N_i \rho_{\boldsymbol{X}} \boldsymbol{b} \,\mathrm{d}V + \bigcup_{k=1}^{m_{\mathrm{s}}} \sum_{j=1}^{n_{\mathrm{s},k}} \boldsymbol{\eta}_j^{\mathrm{T}} \int_{\Gamma_\ell} N_j \bar{\boldsymbol{t}} \,\mathrm{d}A = \boldsymbol{\eta}^{\mathrm{T}} \mathbf{f} \,. \end{aligned} \tag{3.92}$$

The assembly procedure for the surface traction operates on the m_{s} element surfaces subjected to a prescribed traction $\bar{\boldsymbol{t}}$ by summing up the contributions from the $n_{\mathrm{s},k}$ nodes of the surface Γ_k. In the state of equilibrium, $\delta W_{\mathrm{i}} - \delta W_{\mathrm{e}} = 0$. From (3.89), (3.91), and (3.92), it then follows that

$$\boldsymbol{\eta}^{\mathrm{T}} \left(\mathbf{M} \ddot{\mathbf{v}} + \mathbf{r}(\mathbf{d}) - \mathbf{f} \right) = 0 \,. \tag{3.93}$$

Due to the fact that the discrete test function $\boldsymbol{\eta}$ is arbitrary, we are led to the system of equations

$$\mathbf{M} \ddot{\mathbf{v}} + \mathbf{r}(\mathbf{d}) - \mathbf{f} = \mathbf{0} \,, \tag{3.94}$$

which is nonlinear in $\mathbf{d}$ because of the term $\mathbf{r}(\mathbf{d})$. For the linearization, we resort to Equation (3.73). Using (3.82) and (3.83), we obtain for the first part

$$\begin{aligned} \int_{\Omega_{\boldsymbol{X}}} \operatorname{Grad} \Delta \boldsymbol{d}\, \bar{\boldsymbol{S}} \cdot \operatorname{Grad} \boldsymbol{\eta} \,\mathrm{d}V &\approx \bigcup_{\ell=1}^{m} \sum_{i=1}^{n_\ell} \sum_{j=1}^{n_\ell} \int_{\Omega_\ell} \left(\Delta \boldsymbol{d}_j \otimes \frac{\partial N_j}{\partial \boldsymbol{X}_\ell} \right) \bar{\boldsymbol{S}}_\ell \cdot \left(\boldsymbol{\eta}_i \otimes \frac{\partial N_i}{\partial \boldsymbol{X}_\ell} \right) \mathrm{d}V \\ &= \bigcup_{\ell=1}^{m} \sum_{i=1}^{n_\ell} \sum_{j=1}^{n_\ell} \boldsymbol{\eta}_i^{\mathrm{T}} \int_{\Omega_\ell} \bar{G}_{ij} \boldsymbol{I} \,\mathrm{d}V \Delta \boldsymbol{d}_j = \boldsymbol{\eta}^{\mathrm{T}} \mathbf{K}_{\mathrm{s}} \Delta \mathbf{d} \,, \end{aligned} \tag{3.95}$$

where we introduced the abbreviation

$$\bar{G}_{ij} = \left(\frac{\partial N_i}{\partial \boldsymbol{X}_\ell}\right)^{\mathrm{T}} \bar{\boldsymbol{S}}_\ell \frac{\partial N_j}{\partial \boldsymbol{X}_\ell} = \begin{pmatrix} N_{i,X} & N_{i,Y} & N_{i,Z} \end{pmatrix} \begin{pmatrix} \bar{S}_{11} & \bar{S}_{12} & \bar{S}_{13} \\ \bar{S}_{21} & \bar{S}_{22} & \bar{S}_{23} \\ \bar{S}_{31} & \bar{S}_{32} & \bar{S}_{33} \end{pmatrix} \begin{pmatrix} N_{j,X} \\ N_{j,Y} \\ N_{j,Z} \end{pmatrix} \tag{3.96}$$

with the derivatives of the shape functions $N_{i,X}$, $N_{i,Y}$, and $N_{i,Z}$ with respect to X, Y, and Z, respectively. The matrix $\mathbf{K}_{\mathrm{s}}$ comprises only current stresses and is therefore termed *initial stress matrix*. For the second part of the linearization (3.73), we have

$$\int_{\Omega_{\boldsymbol{X}}} \delta\bar{\boldsymbol{E}} \cdot \bar{\boldsymbol{\mathcal{C}}} \Delta\bar{\boldsymbol{E}} \,\mathrm{d}V = \bigcup_{\ell=1}^{m} \sum_{i=1}^{n_\ell} \sum_{j=1}^{n_\ell} \boldsymbol{\eta}_i^{\mathrm{T}} \int_{\Omega_\ell} \bar{\boldsymbol{B}}_i^{\mathrm{T}} \bar{\boldsymbol{D}} \bar{\boldsymbol{B}}_j \,\mathrm{d}V \,\Delta\boldsymbol{d}_j = \boldsymbol{\eta}^{\mathrm{T}} \mathbf{K}_{\mathrm{m}} \Delta\mathbf{d} \tag{3.97}$$

with the material stiffness matrix $\mathbf{K}_{\mathrm{m}}$. Merging the expressions (3.95) for the initial stress matrix and (3.97) for the material stiffness matrix, we arrive at

$$\begin{aligned} &\int_{\Omega_{\boldsymbol{X}}} \mathrm{Grad}\,\Delta\boldsymbol{d}\,\bar{\boldsymbol{S}} \cdot \mathrm{Grad}\,\boldsymbol{\eta} + \delta\bar{\boldsymbol{E}} \cdot \bar{\boldsymbol{\mathcal{C}}} \Delta\bar{\boldsymbol{E}} \,\mathrm{d}V \\ &\qquad = \bigcup_{\ell=1}^{m} \sum_{i=1}^{n_\ell} \sum_{j=1}^{n_\ell} \boldsymbol{\eta}_i^{\mathrm{T}} \int_{\Omega_\ell} \bar{G}_{ij} \boldsymbol{I} + \bar{\boldsymbol{B}}_i^{\mathrm{T}} \bar{\boldsymbol{D}} \bar{\boldsymbol{B}}_j \,\mathrm{d}V \,\Delta\boldsymbol{d}_j = \boldsymbol{\eta}^{\mathrm{T}} \mathbf{K} \Delta\mathbf{d} \,. \end{aligned} \tag{3.98}$$

In this expression, $\mathbf{K}$ is termed the *tangent stiffness matrix*.

3.1.6 Temporal Discretization

In the previous section, we derived the semi-discrete equations of motion (3.93), which are usually stated as

$$\mathbf{M}\ddot{\mathbf{d}} + \mathbf{r}(\mathbf{d}) = \mathbf{f} \,, \tag{3.99}$$

where $\mathbf{M}$ represents the mass matrix, $\mathbf{r}$ symbolizes the internal force vector, and $\mathbf{f}$ embodies the time-dependent external surface and body loads. It is possible to include damping effects by adding a velocity-dependent damping term $\mathbf{C}\dot{\mathbf{d}}$ such that the equations of motion (3.99) become

$$\mathbf{M}\ddot{\mathbf{d}} + \mathbf{C}\dot{\mathbf{d}} + \mathbf{r}(\mathbf{d}) = \mathbf{f} \,. \tag{3.100}$$

In *explicit* integration schemes, the equations of motion are evaluated at time $t = t_j$ to obtain the displacement $\mathbf{d}_{j+1}$ at time $t = t_{j+1}$:

$$\mathbf{M}\ddot{\mathbf{d}}_j + \mathbf{C}\dot{\mathbf{d}}_j + \mathbf{r}(\mathbf{d}_j) = \mathbf{f}_j \,. \tag{3.101}$$

In the central difference method, we make the assumptions

$$\dot{\mathbf{d}} = \frac{1}{2\Delta t} \left(\mathbf{d}_{j+1} - \mathbf{d}_{j-1}\right) \,, \tag{3.102}$$

$$\ddot{\mathbf{d}} = \frac{1}{\Delta t^2} \left(\mathbf{d}_{j+1} - 2\mathbf{d}_j + \mathbf{d}_{j-1}\right) \tag{3.103}$$

for the velocity vector $\dot{\mathbf{d}}$ and the acceleration vector $\ddot{\mathbf{d}}$. Inserting the above into (3.101) produces

$$\mathbf{M}\left(\mathbf{d}_{j+1} - 2\mathbf{d}_j + \mathbf{d}_{j-1}\right) + \frac{\Delta t}{2}\mathbf{C}\left(\mathbf{d}_{j+1} - \mathbf{d}_{j-1}\right) + \Delta t^2 \mathbf{r}(\mathbf{d}_j) = \Delta t^2 \mathbf{f}_j \tag{3.104}$$

Rearranging the resulting expression then yields

$$\left(\mathbf{M} + \frac{\Delta t}{2}\mathbf{C}\right)\mathbf{d}_{j+1} = \Delta t^2\left(\mathbf{f}_j - \mathbf{r}(\mathbf{d}_j)\right) + \frac{\Delta t}{2}\mathbf{C}\mathbf{d}_{j-1} + \mathbf{M}(2\mathbf{d}_j - \mathbf{d}_{j-1})\ . \tag{3.105}$$

Apparently, Equation (3.105) can be solved efficiently by factorizing the term $\mathbf{M}+\Delta t/2\mathbf{C}$. However, explicit time integration schemes are not unconditionally stable but obey a critical time step size Δt_{crit}. In problems where a high frequency displacement response must be resolved, the time step size Δt might already be determined by the physics rather than the critical time step size Δt_{crit}. For problems where this is not the case, the requirement $\Delta t \leq \Delta t_{\text{crit}}$ might however render explicit time integration schemes prohibitively expensive. In such cases, it is preferable to resort to *implicit* time integration schemes which solve the equations of motion (3.100) at time $t = t_{j+1}$:

$$\mathbf{M}\ddot{\mathbf{d}}_{j+1} + \mathbf{C}\dot{\mathbf{d}}_{j+1} + \mathbf{r}(\mathbf{d}_{j+1}) = \mathbf{f}_{j+1}\ . \tag{3.106}$$

A popular representative of the class of implicit integration schemes is the Newmark method, which assumes

$$\mathbf{d}_{j+1} = \mathbf{d}_j + \Delta t\,\dot{\mathbf{d}}_j + \frac{\Delta t^2}{2}\left((1-2\beta)\ddot{\mathbf{d}}_j + 2\beta\ddot{\mathbf{d}}_{j+1}\right)\ , \tag{3.107}$$

$$\dot{\mathbf{d}}_{j+1} = \dot{\mathbf{d}}_j + \Delta t\left((1-\gamma)\ddot{\mathbf{d}}_j + \gamma\ddot{\mathbf{d}}_{j+1}\right) \tag{3.108}$$

for the displacement $\mathbf{d}_{j+1}$ and the velocity $\dot{\mathbf{d}}_{j+1}$ at time $t = t_{j+1}$. By adjusting the parameters $\gamma \geq 1/2$ and $\beta \geq (\gamma+1/2)^2/4$, it is possible to control the properties of the Newmark scheme. Introducing the integration constants

$$\begin{aligned} \alpha_1 &= \frac{1}{\beta\Delta t^2}\ , & \alpha_2 &= \frac{1}{\beta\Delta t}\ , & \alpha_3 &= \frac{1-2\beta}{2\beta}\ , \\ \alpha_4 &= \frac{\gamma}{\beta\Delta t}\ , & \alpha_5 &= \left(1-\frac{\gamma}{\beta}\right)\ , & \alpha_6 &= \left(1-\frac{\gamma}{2\beta}\right)\Delta t\ , \end{aligned} \tag{3.109}$$

the expressions

$$\ddot{\mathbf{d}}_{j+1} = \alpha_1(\mathbf{d}_{j+1} - \mathbf{d}_j) - \alpha_2\dot{\mathbf{d}}_j - \alpha_3\ddot{\mathbf{d}}_j\ , \tag{3.110}$$

$$\dot{\mathbf{d}}_{j+1} = \alpha_4(\mathbf{d}_{j+1} - \mathbf{d}_j) + \alpha_5\dot{\mathbf{d}}_j + \alpha_6\ddot{\mathbf{d}}_j \tag{3.111}$$

arise. Plugging (3.109) and (3.110)–(3.111) into (3.106) results in

$$\begin{aligned} \mathbf{g}(\mathbf{d}_{j+1}) = \mathbf{M}&\left(\alpha_1(\mathbf{d}_{j+1} - \mathbf{d}_j) - \alpha_2\dot{\mathbf{d}}_j - \alpha_3\ddot{\mathbf{d}}_n\right) \\ &+ \mathbf{C}\left(\alpha_4(\mathbf{d}_{j+1} - \mathbf{d}_j) + \alpha_5\dot{\mathbf{d}}_j + \alpha_6\ddot{\mathbf{d}}_j\right) \\ &+ \mathbf{r}(\mathbf{d}_{j+1}) - \mathbf{f}_{j+1} = \mathbf{0}\ , \end{aligned} \tag{3.112}$$

which is iteratively solved using the Newton-Raphson method within each time step:

$$\begin{aligned} \left(\alpha_1\mathbf{M} + \alpha_4\mathbf{C} + \mathbf{K}(\mathbf{d}_{j+1}^k)\right)\Delta\mathbf{d}_{j+1}^{k+1} &= -\mathbf{g}(\mathbf{d}_{j+1}^k) \\ \mathbf{d}_{j+1}^{k+1} &= \mathbf{d}_{j+1}^k + \Delta\mathbf{d}_{j+1}^{k+1}\ . \end{aligned} \tag{3.113}$$

Table 3.1: Parameters for different implicit time integration schemes as a function of the spectral radius ρ_∞ [100, p. 585].

Algorithm	α_m	α_f	β	γ
Newmark method [119]	0	0	$\frac{1}{(\rho_\infty+1)^2}$	$\frac{3-\rho_\infty}{2\rho_\infty+2}$
Bossak-α method [180]	$\frac{\rho_\infty-1}{\rho_\infty+1}$	0	$\frac{1}{4}\left(1-\alpha_\mathrm{m}\right)^2$	$\frac{1}{2}-\alpha_\mathrm{m}$
Hilber-α method [68]	0	$\frac{1-\rho_\infty}{\rho_\infty+1}$	$\frac{1}{4}\left(1+\alpha_\mathrm{f}\right)^2$	$\frac{1}{2}+\alpha_\mathrm{f}$
Generalized-α method [30]	$\frac{2\rho_\infty-1}{\rho_\infty+1}$	$\frac{\rho_\infty}{\rho_\infty+1}$	$\frac{1}{4}(1-\alpha_\mathrm{m}+\alpha_\mathrm{f})^2$	$\frac{1}{2}-\alpha_\mathrm{m}+\alpha_\mathrm{f}$

In the Generalized-α method, the terms in Equation (3.100) are evaluated at an intermediate time $t_j \leq t_{j+1-\alpha} \leq t_{j+1}$, where

$$t_{j+1-\alpha} = t_{j+1} - \alpha(t_{j+1} - t_j) = (1-\alpha)t_{j+1} + \alpha t_j \tag{3.114}$$

with the adjustable parameter α. Introducing the parameters α_m and α_f, the displacement, the velocity, and the acceleration are evaluated as

$$\begin{aligned} \mathbf{d}_{j+1-\alpha_\mathrm{f}} &= (1-\alpha_\mathrm{f})\mathbf{d}_{j+1} + \alpha_\mathrm{f}\mathbf{d}_j \ , \\ \dot{\mathbf{d}}_{j+1-\alpha_\mathrm{f}} &= (1-\alpha_\mathrm{f})\dot{\mathbf{d}}_{j+1} + \alpha_\mathrm{f}\dot{\mathbf{d}}_j \ , \\ \ddot{\mathbf{d}}_{j+1-\alpha_\mathrm{m}} &= (1-\alpha_\mathrm{m})\ddot{\mathbf{d}}_{j+1} + \alpha_\mathrm{m}\ddot{\mathbf{d}}_j \ . \end{aligned} \tag{3.115}$$

Based on these approximations, the semi-discrete equations of motion (3.93) become

$$\mathbf{M}\ddot{\mathbf{d}}_{j+1-\alpha_\mathrm{m}} + \mathbf{C}\dot{\mathbf{d}}_{j+1-\alpha_\mathrm{f}} + \mathbf{r}(\mathbf{d}_{j+1-\alpha_\mathrm{f}}) = \mathbf{f}_{j+1-\alpha_\mathrm{f}} \ . \tag{3.116}$$

In order to relate the velocity $\dot{\mathbf{d}}_{j+1-\alpha_\mathrm{f}}$ and the acceleration $\ddot{\mathbf{d}}_{j+1-\alpha_\mathrm{m}}$ to the displacement $\mathbf{d}_{j+1}$ as the only unknown, we resort to the Newmark approximations

$$\dot{\mathbf{d}}_{j+1} = \frac{\gamma}{\beta\Delta t}(\mathbf{d}_{j+1} - \mathbf{d}_j) - \frac{\gamma-\beta}{\beta}\dot{\mathbf{d}}_j - \frac{\gamma-2\beta}{2\beta}\Delta t\ddot{\mathbf{d}}_j \ , \tag{3.117}$$

$$\ddot{\mathbf{d}}_{j+1} = \frac{1}{\beta\Delta t^2}(\mathbf{d}_{j+1} - \mathbf{d}_j) - \frac{1}{\beta\Delta t}\dot{\mathbf{d}}_j - \frac{1-2\beta}{2\beta}\ddot{\mathbf{d}}_j \ . \tag{3.118}$$

From this, we obtain [100, p. 574]

$$\begin{aligned} \dot{\mathbf{d}}_{j+1-\alpha_\mathrm{f}} &= \frac{1-\alpha_\mathrm{m}}{\beta\Delta t}(\mathbf{d}_{j+1} - \mathbf{d}_j) - \frac{1-\alpha_\mathrm{m}}{\beta}\dot{\mathbf{d}}_j - \frac{1-\alpha_\mathrm{m}-2\beta}{2\beta}\Delta t\ddot{\mathbf{d}}_j \\ \ddot{\mathbf{d}}_{j+1-\alpha_\mathrm{m}} &= \frac{(1-\alpha_\mathrm{f})\gamma}{\beta\Delta t^2}(\mathbf{d}_{j+1} - \mathbf{d}_j) - \frac{(1-\alpha_\mathrm{f})\gamma-\beta}{\beta}\dot{\mathbf{d}}_j - \frac{(\gamma-2\beta)(1-\alpha_\mathrm{f})}{2\beta}\Delta t\ddot{\mathbf{d}}_j \ . \end{aligned} \tag{3.119}$$

Table 3.1 summarizes possible choices for the integration parameters in terms of the spectral radius ρ_∞. Note that we recover the Newmark method when choosing $\alpha_\mathrm{m} = \alpha_\mathrm{f} = 0$. The approximations (3.119) are then plugged into (3.116), which then takes the form [100, p. 575]

$$\begin{aligned} \mathbf{g}(\mathbf{d}_{j+1}) = \mathbf{M}&\left(\frac{1-\alpha_\mathrm{m}}{\beta\Delta t^2}(\mathbf{d}_{j+1} - \mathbf{d}_j) - \frac{1-\alpha_\mathrm{m}}{\beta\Delta t}\dot{\mathbf{d}}_j - \frac{1-\alpha_\mathrm{m}-2\beta}{2\beta}\ddot{\mathbf{d}}_j\right) \\ &+ \mathbf{C}\left(\frac{(1-\alpha_\mathrm{f})\gamma}{\beta\Delta t}(\mathbf{d}_{j+1} - \mathbf{d}) - \frac{(1-\alpha_\mathrm{f})\gamma-\beta}{\beta}\mathbf{d}_j - \frac{(\gamma-2\beta)(1-\alpha_\mathrm{f})}{2\beta}\Delta t\ddot{\mathbf{d}}_j\right) \\ &+ \mathbf{r}(\mathbf{d}_{j+1-\alpha_\mathrm{f}}) - \mathbf{f}_{j+1-\alpha_\mathrm{f}} = \mathbf{0} \ . \end{aligned} \tag{3.120}$$

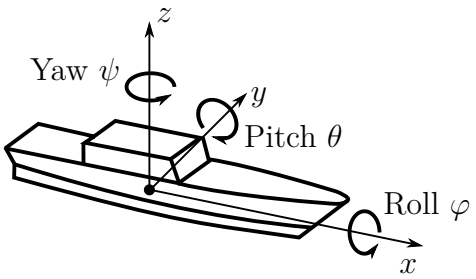


Figure 3.2: Euler angles yaw ψ, pitch θ, and roll φ.

In each time step, this equation is solved using the Newton-Raphson procedure

$$\left(\frac{1-\alpha_\mathrm{m}}{\beta\Delta t^2}\mathbf{M}+\frac{(1-\alpha_\mathrm{f})\gamma}{\beta\Delta t}\mathbf{C}+\mathbf{K}(\mathbf{d}_{j+1}^k)\right)=-\mathbf{g}(\mathbf{d}_{j+1}^k) \qquad \mathbf{d}_{j+1}^{k+1}=\mathbf{d}_{j+1}^k+\Delta\mathbf{d}_{j+1}^{k+1}\ . \tag{3.121}$$

3.2 Rigid Body Equations

If the elasticity of the structure is negligible and the internal stress state is not of interest, the structure can be considered as a rigid body. The configuration of the body is then completely described by six degrees of freedom, namely the displacement $\boldsymbol{d}$ of the center of mass and the Euler angles ϕ, θ, and ψ representing the orientation of the body-attached frame with respect to the inertial frame. In the context of (aero-)nautical applications, the Euler angles are termed *yaw* ψ, *pitch* θ, and *roll* φ, see Figure 3.2, and follow a $z-y'-x''$ sequence of rotation. In addition to displacement and rotation, the current state of the body is also characterized by the translational velocity $\boldsymbol{v}$ and the rotational velocity $\boldsymbol{\omega}$.

First, let us consider the translational motion, which is governed by Newton's second law of motion or conservation of linear momentum

$$\frac{\mathrm{D}\boldsymbol{p}}{\mathrm{D}t}(t)=\boldsymbol{f}(t)=m\frac{\mathrm{D}\boldsymbol{v}}{\mathrm{D}t}(t)\ , \tag{3.122}$$

where $\mathrm{D}\boldsymbol{p}/\mathrm{D}t$ denotes the material time derivative of linear momentum, $\boldsymbol{f}$ embodies the external forces acting on the body, m symbolizes the constant mass of the body, and $\mathrm{D}\boldsymbol{v}/\mathrm{D}t$ is the acceleration of the center of mass. Accordingly, angular motion is described by the Euler equations or conservation of angular momentum

$$\frac{\mathrm{D}\boldsymbol{l}}{\mathrm{D}t}(t)=\boldsymbol{m}(t)=\frac{\mathrm{D}}{\mathrm{D}t}(\boldsymbol{\Theta}(t)\boldsymbol{\omega}(t))\ . \tag{3.123}$$

Here, $\mathrm{D}\boldsymbol{l}/\mathrm{D}t$ represents the time derivative of angular momentum, $\boldsymbol{m}$ is the sum of external moments acting on the body, $\boldsymbol{\Theta}$ is the time-dependent inertia tensor in the inertial frame, and $\boldsymbol{\omega}=\boldsymbol{\Theta}^{-1}\boldsymbol{l}$ denotes the angular velocity pseudo-vector. The inertia tensor $\hat{\boldsymbol{\Theta}}$ defined in the body frame does not depend on time, and it serves to compute the time-dependent inertia tensor $\boldsymbol{\Theta}$ in the inertial frame by virtue of the transformation $\boldsymbol{\Theta}=\boldsymbol{R}\hat{\boldsymbol{\Theta}}\boldsymbol{R}^\mathrm{T}$ with the orthonormal rotation matrix $\boldsymbol{R}$ [6, p. 15]. From the inverse transformation $\hat{\boldsymbol{\Theta}}=\boldsymbol{R}^\mathrm{T}\boldsymbol{\Theta}\boldsymbol{R}$, it is possible to compute $\hat{\boldsymbol{\Theta}}$ from $\boldsymbol{\Theta}$. In computer implementations, it is often preferable to employ a quaternion $\boldsymbol{\rho}$ instead of the rotation matrix $\boldsymbol{R}$ to describe the orientations of

the body. This helps to avoid numerical drift, i.e., a loss of orthonormality of the rotation matrix, and the gimbal lock phenomenon [6, p. 20, 178, p. 21]. The force $\boldsymbol{f}$ and the moment $\boldsymbol{m}$ are obtained by integrating the external loads over the body's surface.

In order to integrate (3.122) and (3.123) in time, it is convenient to transform these equations into a state space representation, which produces the system of ordinary differential equations

$$\frac{\mathrm{D}\boldsymbol{y}}{\mathrm{D}t} = f(\boldsymbol{y}) \; . \tag{3.124}$$

Here,

$$\boldsymbol{y} = \begin{pmatrix} \boldsymbol{x} & \boldsymbol{\rho} & \boldsymbol{p} & \boldsymbol{l} \end{pmatrix}^{\mathrm{T}} \tag{3.125}$$

is the state vector, the derivative of which amounts to

$$\frac{\mathrm{D}\boldsymbol{y}}{\mathrm{D}t} = \begin{pmatrix} \boldsymbol{v} & \frac{1}{2}\boldsymbol{\omega} * \boldsymbol{\rho} & \boldsymbol{f} & \boldsymbol{m} \end{pmatrix}^{\mathrm{T}} \; . \tag{3.126}$$

The expression $\boldsymbol{\omega} * \boldsymbol{\rho}$ symbolizes the quaternion product of $(0, \boldsymbol{\omega})$ and $\boldsymbol{\rho}$ [6, p. 20 sq.]. To constitute a well-posed problem, the initial condition $\boldsymbol{y}(t = 0) = \boldsymbol{y}_0$ needs to be imposed.

Based on the state space description (3.124), we can apply a wide range of different time integration schemes for ordinary differential equations of first order. Possible choices are the explicit or implicit Euler scheme or more sophisticated time-adaptive Runge-Kutta methods.

4 Coupled Problems

Having reviewed the fluid and structural equations in Chapter 2 and 3, we will now focus on the interaction of multiple fields in the framework of a partitioned solution approach. Of course, the interaction of *two* fields as in the case of FSI problems will be of particular interest.

First of all, we will describe the geometrical setting of a general multifield problem and contrast a monolithic solution procedure with a partitioned approach, which is pursued in this work. Following that, we will present a generic partitioned solution procedure and introduce a simple coupled problem suited for a partitioned analysis, which will later serve to illustrate the integral parts of the solution procedure. As the individual components of the coupling procedure may have a significant influence on its stability and computational effort, they will be discussed in sufficient detail. Predictors, for instance, serve to stabilize the solution process and may reduce the number of implicit iterations by providing an initial solution at the beginning of each time increment that is closer to the converged solution in the current time increment than the converged solution from the previous time increment. By considering the displacement response of a simple mechanical system, the performance of the various predictors will be investigated; this analysis is expected to provide useful hints for the application of the predictor schemes also in more complex problems. In order to transfer the field quantities of interest between possibly non-conforming discretizations, a vast range of different interpolation schemes can be applied. Based on several benchmark examples, the interpolation schemes presented in this work will be compared regarding their accuracy and computational efficiency. Subsequently, we will briefly recapitulate different convergence criteria that are used to verify whether the coupling algorithm has converged. Last but not least, several convergence acceleration schemes are proposed and formulated in a manner that allows for a seamless integration into the generic partitioned solution procedure. Also here, a simple coupled problem will be studied to give a general impression of the performance of the different schemes.

4.1 Governing Equations

In a general multifield problem, we consider a finite number of separate open subdomains $\Omega_1, \ldots, \Omega_m$ such that

$$\bar{\Omega} = \bigcup_{i=1}^{m} \bar{\Omega}_i \tag{4.1}$$

holds for the closure $\bar{\Omega} := \Omega \cup \partial\Omega$ of Ω. Multifield problems can be classified into surface- or geometrically coupled and volume- or materially coupled problems [110, p. 27], see Figure 4.1. In surface-coupled problems, distinct subdomains Ω_i, Ω_j do not overlap each other, i.e., $\Omega_i \cap \Omega_j = \emptyset$ for $i \neq j$, but only share a common boundary $\Gamma_{i,j} = \Gamma_{j,i} := \Gamma_i \cap \Gamma_j \neq \emptyset$, where $\Gamma_i := \partial\Omega_i$ denotes the boundary of the domain Ω_i. In volume-coupled problems, in contrast, distinct subdomains Ω_i, Ω_j may overlap each other in part or in full,

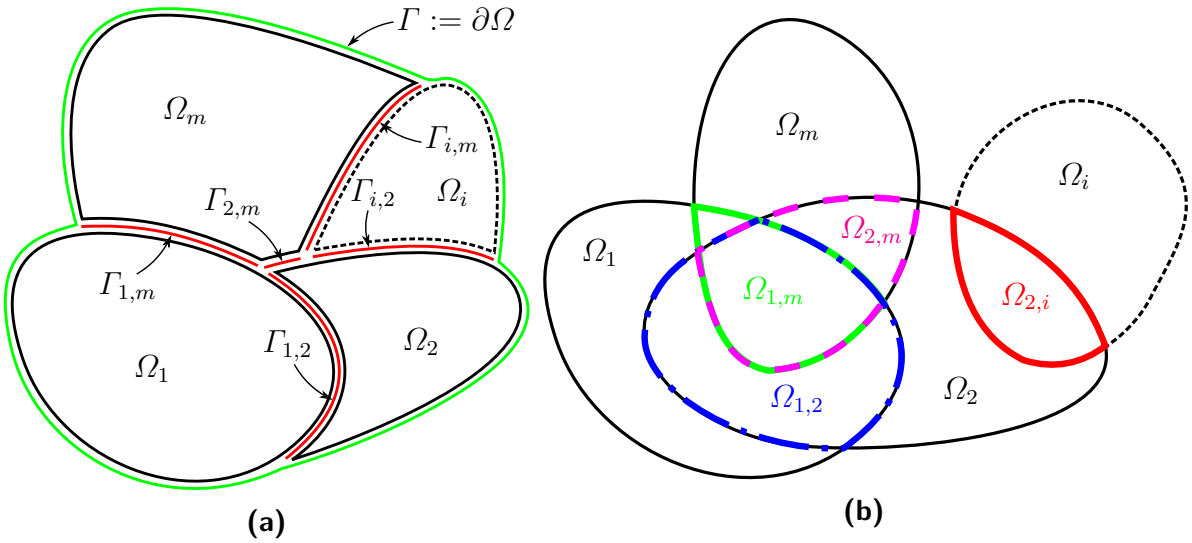


Figure 4.1: Comparison of (a) surface- or geometrically coupled and (b) volume- or materially coupled problems.

i.e., $\Omega_{i,j} = \Omega_{j,i} := \Omega_i \cap \Omega_j \neq \emptyset$ for $i \neq j$. For the sake of notation, we introduce the term *interface* and the symbol $I_{i,j}$ to denote the intersection $\Gamma_{i,j}$ of the boundaries $\Gamma_i, \Gamma_j, i \neq j$ in a surface-coupled problem or the intersection $\Omega_{i,j}$ of the domains $\Omega_i, \Omega_j, i \neq j$ in a volume-coupled problem.

A prominent example of surface-coupled problems are FSI problems, where the fluid flow causes a deformation of the mechanical structure, which in turn also leads to a change in the surrounding fluid flow. Further examples are acoustic-structure interaction[1] [109, p. 4] or purely mechanically coupled problems arising in contact mechanics. A typical example of volume-coupled problems are thermo-mechanically coupled problems, where the change in temperature leads to a thermal expansion of the mechanical structure, which in turn influences the thermal properties of the structure. Other applications include thermoelectricity (involving the interaction of heat conduction and electrodynamics) or reaction-diffusion systems, which are characterized by the interaction of chemical reactions and diffusive transport [109, p. 4].

Besides the classification into volume- and surface coupled problems, multifield problems can also be categorized either as weakly or one-way coupled problems or as strongly or two-way coupled problems. In weakly coupled two-field problems, a change in the first field leads to a change in the second field, whereas the influence of the second field on the first field is negligible. In contrast, the fields in a strongly coupled two-field problem interact with each other, i.e. a change in the first field causes the second field to change, which in turn affects the first field again. If more than two fields are involved, the overall problem may comprise weakly- as well as strongly coupled two-field problems. The terms *weak* and *strong* hence always refer to the mutual interaction of *two* fields. Strongly coupled problems are considerably more difficult to solve and are therefore mainly focused in this work.

[1] Precisely speaking, acoustic-structure interaction also involves the interaction of a fluid and a mechanical structure. However, the governing equations are different – while acoustic problems are governed by the Helmholtz equation, flow problems are described by the more difficult Navier-Stokes equations.

For the interaction of m fields, the coupled multifield problem may be stated as a discrete system of coupled nonlinear equations

$$\mathbf{g}_i(t, \mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_i, \ldots, \mathbf{u}_m) = \mathbf{0} \quad \text{on } \Omega_i \,, \quad i = 1, \ldots, m \,, \tag{4.2}$$

where $\mathbf{g}_i$ is a discrete nonlinear field equation defined on Ω_i, $\mathbf{u}_i$ is the corresponding state variable, and t denotes time. Introducing $\mathbf{g} := \begin{pmatrix} \mathbf{g}_1 & \ldots & \mathbf{g}_m \end{pmatrix}$ and $\mathbf{u} := \begin{pmatrix} \mathbf{u}_1 & \ldots & \mathbf{u}_m \end{pmatrix}$, the coupled problem (4.2) can be more compactly written as

$$\mathbf{g}(\mathbf{u}) = \mathbf{0} \quad \text{on } \Omega \,. \tag{4.3}$$

In a *monolithic* approach, Equation (4.3) is solved by applying a Newton-Raphson procedure using $\mathbf{u}^0$ as an initial guess and iterating for $k = 0, 1, 2, \ldots$:

$$\begin{aligned} \left.\frac{\partial \mathbf{g}}{\partial \mathbf{u}}\right|_{\mathbf{u}=\mathbf{u}^k} \Delta\mathbf{u}^k &= -\mathbf{g}\left(\mathbf{u}^k\right) \\ \mathbf{u}^{k+1} &= \mathbf{u}^k + \Delta\mathbf{u}^{k+1} \,, \end{aligned} \tag{4.4}$$

where the Jacobian matrix $\mathbf{J}_\mathbf{g} := {}^{\partial \mathbf{g}}/_{\partial \mathbf{u}}$ is computed from

$$\frac{\partial \mathbf{g}}{\partial \mathbf{u}} = \begin{pmatrix} \frac{\partial \mathbf{g}_1}{\partial \mathbf{u}_1} & \cdots & \frac{\partial \mathbf{g}_1}{\partial \mathbf{u}_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{g}_m}{\partial \mathbf{u}_1} & \cdots & \frac{\partial \mathbf{g}_m}{\partial \mathbf{u}_m} \end{pmatrix} \,. \tag{4.5}$$

Remarkable advantages of this approach are its good stability characteristics and quadratic convergence rates in the vicinity of the solution $\mathbf{u}^*$ which satisfies $\mathbf{g}(\mathbf{u}^*) = \mathbf{0}$. However, the computation of the cross-derivatives in the off-diagonal entries in the Jacobian matrix is complicated, and the resulting discrete system of equations may become large and expensive to solve. Moreover, the monolithic approach is rather inflexible as customized solvers are required for each particular coupled problem. Hence, it is out of the question to reusing existing specialized and efficient black-box solvers.

Partitioned solution approaches primarily aim at circumventing this lack of flexibility of the monolithic approach. By considering each of the subproblems separately and exchanging the relevant field quantities in an iterative manner within each time increment, the partitioned approach enables the use of different spatial and temporal discretizations for each of the subproblems, which entails the reuse of available specialized and fast black-box solvers. This does not only boost software modularity and reusability, but also promotes performance and the efficient use of computational resources. Yet, the partitioned approach also comes with some difficulties. More specifically, the solution procedure may become unstable – or a high number of implicit iterations within a time increment may render the approach prohibitively expensive. To tackle these problems, several predictor schemes are proposed in this work, providing a reasonable initial iterate close to the converged solution before entering the implicit iteration. To further improve the stability of the solution process and accelerate convergence, the use of an appropriate convergence acceleration scheme is indispensable. Numerous convergence acceleration schemes will therefore be discussed in the present thesis.

4.2 Generic Partitioned Coupling Algorithm

In a partitioned solution approach, the nonlinear equations (4.2) are solved separately, and the relevant field quantities are exchanged iteratively within a time increment across the interface. A generic staggered solution procedure, which applies to surface- as well as volume-coupled problems, is sketched in Algorithm 1. The term *staggered* reflects the

1: **function** PARTITIONEDSOLUTIONALGORITHM(initial solution $\mathbf{u}_0$, start time t_0, final time T, initial time increment Δt_j, convergence tolerance ε)
2: $j := 0$
3: **while** $t_j \leq T$ **do**
4: $k := 0$
5: $\tilde{\mathbf{u}}_{j+1}^k := \mathcal{P}(\mathbf{u}_j, \mathbf{u}_{j-1}, \ldots)$ ▷ Predictor
6: **while true do**
7: $\mathbf{u}_{j+1}^k := \mathcal{S}_m \left(\cdots \mathcal{S}_2 \left(\mathcal{I}_{1,2} \left(\mathcal{S}_1 \left(\tilde{\mathbf{u}}_{j+1}^k \right) \right) \right) \right)$ ▷ Subfield solution, interpolation
8: $\mathbf{r}_{j+1}^k := \mathbf{u}_{j+1}^k - \tilde{\mathbf{u}}_{j+1}^k$
9: **if** $\|\mathbf{r}_{j+1}^k\|_p \leq \varepsilon$ **then** ▷ Convergence check
10: $\mathbf{u}_{j+1} := \mathbf{u}_{j+1}^k, \mathbf{U}(:, j) := \mathbf{u}_{j+1}$
11: **break**
12: **end if**
13: $\tilde{\mathbf{u}}_{j+1}^{k+1} := \mathcal{A} \left(\mathbf{u}_{j+1}^k, \mathbf{u}_{j+1}^{k-1}, \ldots, \mathbf{r}_{j+1}^k, \mathbf{r}_{j+1}^{k-1}, \ldots \right)$ ▷ Convergence acceleration
14: $k := k + 1$
15: **end while**
16: $t_{j+1} := t_j + \Delta t_j$
17: $j := j + 1$
18: **end while**
19: **return U** ▷ Converged solutions
20: **end function**

Algorithm 1: Generic partitioned solution algorithm.

fact that the field problems are solved in a sequential manner, as opposed to solving them in parallel [25, p. 1121]. In the former case, the most recently computed field quantities can be employed for the solution of each of the subfields, whereas quantities from the previous iteration must be used in the latter case to permit a parallel execution of the subproblem solvers. Due to the fact that the subfield solvers work simultaneously and do not have to wait for each other, parallel schemes have a great potential with regard to saving computational cost. However, they are also prone to instability and require a potentially higher number of iterations per time increment, which may even wipe out their advantages regarding the computational effort. Yet, parallel schemes are still an active research topic, and the interested reader is referred to the recent publications in this field [25, 111, 163]. Besides staggered and parallel schemes, we furthermore distinguish between implicit and explicit schemes. In an implicit scheme, the relevant field quantities are exchanged multiple times per time increment, whereas an explicit scheme limits the number of iterations in a time increment to one – in other words, the implicit iteration within a time increment is omitted.

Let us now return to Algorithm 1. Prior to starting the solution procedure, we need to provide an initial value $\mathbf{u}_0$. At the beginning of each time increment, a predictor $\mathcal{P}$

Table 4.1: Parameters for the mass-spring-damper system depicted in Figure 4.2.

Parameter	Values
Masses	LF case: $m_1 = 2\,\mathrm{kg}$, $m_2 = 8\,\mathrm{kg}$, $m_3 = 4\,\mathrm{kg}$ HF case: $m_1 = 0.2\,\mathrm{kg}$, $m_2 = 0.8\,\mathrm{kg}$, $m_3 = 0.4\,\mathrm{kg}$
Spring constants	$k_1 = 500\,\mathrm{N/m}$, $k_2 = 240\,\mathrm{N/m}$, $k_{31} = 180\,\mathrm{N/m}$, $k_{32} = 120\,\mathrm{N/m^2}$, $k_4 = 350\,\mathrm{N/m}$
Damping coefficients	$c_1 = 0.015\,\mathrm{Ns/m}$, $c_2 = 0.1\,\mathrm{Ns/m}$, $c_3 = 0.08\,\mathrm{Ns/m}$, $c_4 = 0.05\,\mathrm{Ns/m}$
Forces	$F_i(t) = A_{1i}\sin(2\pi f_{1i}t)\sin(2\pi(B_i + A_{2i}\sin(2\pi t))t)$ $A_{11} = 50\,\mathrm{N}$, $f_{11} = 1.1\,\mathrm{Hz}$, $B_1 = 4\,\mathrm{Hz}$, $A_{21} = 0.05\,\mathrm{Hz}$ $A_{12} = 8.5\,\mathrm{N}$, $f_{12} = 2.4\,\mathrm{Hz}$, $B_2 = 4\,\mathrm{Hz}$, $A_{22} = 0.13\,\mathrm{Hz}$ $A_{13} = 16\,\mathrm{N}$, $f_{13} = 1.6\,\mathrm{Hz}$, $B_3 = 10\,\mathrm{Hz}$, $A_{23} = 0.1\,\mathrm{Hz}$

is applied to generate a reasonable initial solution $\tilde{\mathbf{u}}_{j+1}^0$ for the current time increment t_{j+1}. Following that, we enter the iterative solution procedure and supply the solution $\tilde{\mathbf{u}}_{j+1}^k$ to the first subfield solver $\mathcal{S}_1$. The result is then passed over to the interpolation operator $\mathcal{I}_{1,2}$ to interpolate the result from the surface discretization $\Gamma_{1,h}$ in the case of a surface-coupled problem or from the volume discretization $\Omega_{1,h}$ in the case of a volume-coupled problem to the discretization $\Gamma_{2,h}$ or $\Omega_{2,h}$, respectively. The procedure is repeated until the last subfield solver $\mathcal{S}_m$ delivers the solution $\mathbf{u}_{j+1}^k$. From the difference between the solutions $\mathbf{u}_{j+1}^k$ and $\tilde{\mathbf{u}}_{j+1}^k$, we compute the residual $\mathbf{r}_{j+1}^k$. If the p-norm $\|\mathbf{r}_{j+1}^k\|_p$ of the residual undershoots a predefined convergence tolerance ε, the algorithm is considered to be converged, and the most recent solution $\mathbf{u}_{j+1}^k$ is stored in the jth column of the solution matrix $\mathbf{U}$ before we proceed to the next time increment $t_{j+1} = t_j + \Delta t_j$. Certainly, also other convergence criteria or combinations thereof can be employed to check convergence. If the coupling algorithm is not yet converged, a convergence acceleration scheme $\mathcal{A}$ is applied to improve the solution $\mathbf{u}_{j+1}^k$ and to compute an updated solution $\tilde{\mathbf{u}}_{j+1}^{k+1}$.

4.3 Illustrative Coupled Problem

Before addressing the constituent parts of the generic coupling algorithm depicted in Algorithm 1, we will discuss a simple coupled problem that can be analyzed using the partitioned approach. Later, it will serve to illustrate the predictor and convergence acceleration schemes presented in the forthcoming sections.

Let us consider the simple three-degree-of-freedom mass-spring-damper system depicted in Figure 4.2. Each of the masses is connected to the fixed wall or its neighboring mass by a spring and a damper. The first, second, and fourth spring are linear springs, while the third spring is nonlinear and exhibits the force-displacement relationship $F_{\mathrm{s}3} = k_{31}(d_3 - d_2) + k_{32}(d_3 - d_2)^2$. The dampers are all linear; hence, the resulting damping force depends linearly on the velocity. In addition, every mass m_i is subjected to a time-dependent force $F_i(t)$. All relevant parameters of the system are listed in Table 4.1. To adjust the frequency of the displacement response, two different sets of masses are chosen. The first set of masses results in a low-frequency (LF) displacement response, whereas each mass

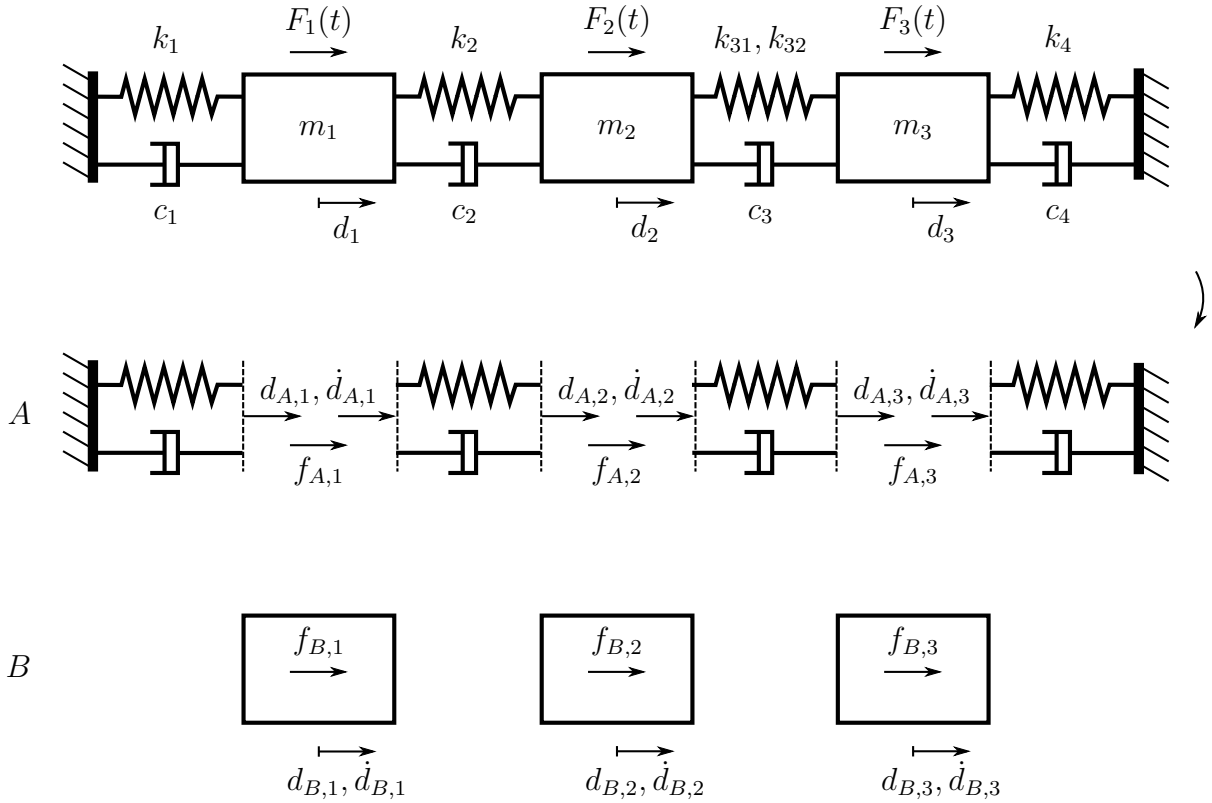


Figure 4.2: Mass-spring-damper system and partitioned subsystems A and B.

is reduced by a factor of $s = 10$ in the second set of masses to generate a high-frequency (HF) displacement response.

The system is amenable to an analysis by means of the partitioned approach if the system is, for instance, split into the subsystem A including the springs and dampers and the subsystem B consisting of the masses m_1, m_2, and m_3 subject to time-, displacement-, and velocity-dependent external forces. Establishing the equations of dynamic mechanical equilibrium leads to

$$\begin{aligned}
f_{A,1} &= -k_1 d_{A,1} - c_1 \dot{d}_{A,1} + k_2(d_{A,2} - d_{A,1}) + c_2(\dot{d}_{A,2} - \dot{d}_{A,1}) + F_1(t) \\
f_{A,2} &= -k_2(d_{A,2} - d_{A,1}) - c_2(\dot{d}_{A,2} - \dot{d}_{A,1}) + k_{31}(d_{A,3} - d_{A,2}) + k_{32}(d_{A,3} - d_{A,2})^2 \\
&\quad + c_3(\dot{d}_{A,3} - \dot{d}_{A,2}) + F_2(t) \\
f_{A,3} &= -k_{31}(d_{A,3} - d_{A,2}) - k_{32}(d_{A,3} - d_{A,2})^2 - c_3(\dot{d}_{A,3} - \dot{d}_{A,2}) - k_4 d_{A,3} - c_4 \dot{d}_{A,3} + F_3(t)
\end{aligned} \tag{4.6}$$

for subsystem A and

$$\begin{aligned}
m_1 \ddot{d}_1 &= f_{B,1} \\
m_2 \ddot{d}_2 &= f_{B,2} \\
m_3 \ddot{d}_3 &= f_{B,3}
\end{aligned} \tag{4.7}$$

for subsystem B. In matrix form, this is more compactly recast as

$$\mathbf{f}_A = \begin{pmatrix} -k_1 d_{A,1} - c_1 \dot{d}_{A,1} + k_2(d_{A,2} - d_{A,1}) \\ +c_2(\dot{d}_{A,2} - \dot{d}_{A,1}) + F_1(t) \\ -k_2(d_{A,2} - d_{A,1}) - c_2(\dot{d}_{A,2} - \dot{d}_{A,1}) + k_{31}(d_{A,3} - d_{A,2}) \\ +k_{32}(d_{A,3} - d_{A,2})^2 + c_3(\dot{d}_{A,3} - \dot{d}_{A,2}) + F_2(t) \\ -k_{31}(d_{A,3} - d_{A,2}) - k_{32}(d_{A,3} - d_{A,2})^2 \\ -c_3(\dot{d}_{A,3} - \dot{d}_{A,2}) - k_4 d_{A,3} - c_4 \dot{d}_{A,3} + F_3(t) \end{pmatrix} = \mathbf{f}_A(t, \mathbf{d}_A, \dot{\mathbf{d}}_A) \tag{4.8}$$

for subsystem A and

$$\mathbf{M}\ddot{\mathbf{d}}_B = \begin{pmatrix} m_1 & 0 & 0 \\ 0 & m_2 & 0 \\ 0 & 0 & m_3 \end{pmatrix} \begin{pmatrix} \ddot{d}_{B,1} \\ \ddot{d}_{B,2} \\ \ddot{d}_{B,3} \end{pmatrix} = \begin{pmatrix} f_{B,1} \\ f_{B,2} \\ f_{B,3} \end{pmatrix} = \mathbf{f}_B \tag{4.9}$$

for subsystem B. At the interface, the conditions

$$\mathbf{d}_A = \mathbf{d}_B \ , \quad \dot{\mathbf{d}}_A = \dot{\mathbf{d}}_B \ , \quad \text{and} \quad \mathbf{f}_A = \mathbf{f}_B \tag{4.10}$$

must hold. In subsystem B, the initial conditions

$$\mathbf{d}_B(t=0) = \mathbf{d}_{B,0} \ , \quad \dot{\mathbf{d}}_B(t=0) = \dot{\mathbf{d}}_{B,0} \ , \quad \text{and } \ddot{\mathbf{d}}_B(t=0) = \ddot{\mathbf{d}}_{B,0} \tag{4.11}$$

apply. Following the partitioned solution strategy outlined in Algorithm 1, the displacement $\mathbf{d}_A = \mathbf{d}_B$ and the velocity $\dot{\mathbf{d}}_A = \dot{\mathbf{d}}_B$ are first fed to subsystem A, which evaluates the displacement- and velocity-dependent spring and damper forces as well as the time-dependent forces to produce the force $\mathbf{f}_A = \mathbf{f}_A(t, \mathbf{d}_A, \dot{\mathbf{d}}_A)$. Subsequently, the force $\mathbf{f}_B = \mathbf{f}_A$ is passed over to subsystem B, which computes the displacement $\mathbf{d}_B$ and the velocity $\dot{\mathbf{d}}_B$ under the action of the external force $\mathbf{f}_B$. In each time increment, this procedure is repeated until the subsystems are in equilibrium with each other.

In subsystem B, we choose the Newmark scheme presented in Section 3.1.6 to advance in time. Equation (3.112) reduces to a linear system, which can easily be solved due to the diagonality of the mass matrix $\mathbf{M}$:

$$\alpha_0 \mathbf{M}\mathbf{d}_{B,j+1} = \mathbf{f}_{B,j+1} + \mathbf{M}\left(\alpha_0 \mathbf{d}_{B,j} + \alpha_2 \dot{\mathbf{d}}_{B,j} + \alpha_3 \ddot{\mathbf{d}}_{B,j}\right) \ . \tag{4.12}$$

The low- and high frequency displacement response of the system are shown in Figure 4.3. At $t = 0\,\mathrm{s}$, $d_{B,0,1} = 0\,\mathrm{m}$, $d_{B,0,2} = 0.5\,\mathrm{m}$, $d_{B,0,3} = 0.75\,\mathrm{m}$, $\dot{d}_{B,0,i} = 0\,\mathrm{m/s}$, and $\ddot{d}_{B,0,i} = f_{B,i}(t = 0\,\mathrm{s}, d_{B,0,i}, \dot{d}_{B,0,i})/m_i, i = 1, \ldots, 3$ are chosen as initial conditions. Obviously, the time signal is sufficiently complex to allow for a sound comparison of the various predictor and convergence acceleration schemes and allude to differences in their performance.

4.4 Predictors

Clearly, a reasonable initial value $\mathbf{u}_{j+1}^0$ at the beginning of the current time increment t_{j+1} will reduce the number of implicit iterations and, hence, have a notable effect on the efficiency of the entire solution procedure. To this end, we employ a predictor $\mathcal{P}$ that extrapolates the solutions $\mathbf{u}_j, \mathbf{u}_{j-1}, \ldots$ from the previous time steps $t_j, t_{j-1}, \ldots$ to compute an initial solution $\mathbf{u}_{j+1}^0$ for the current time increment t_{j+1}.

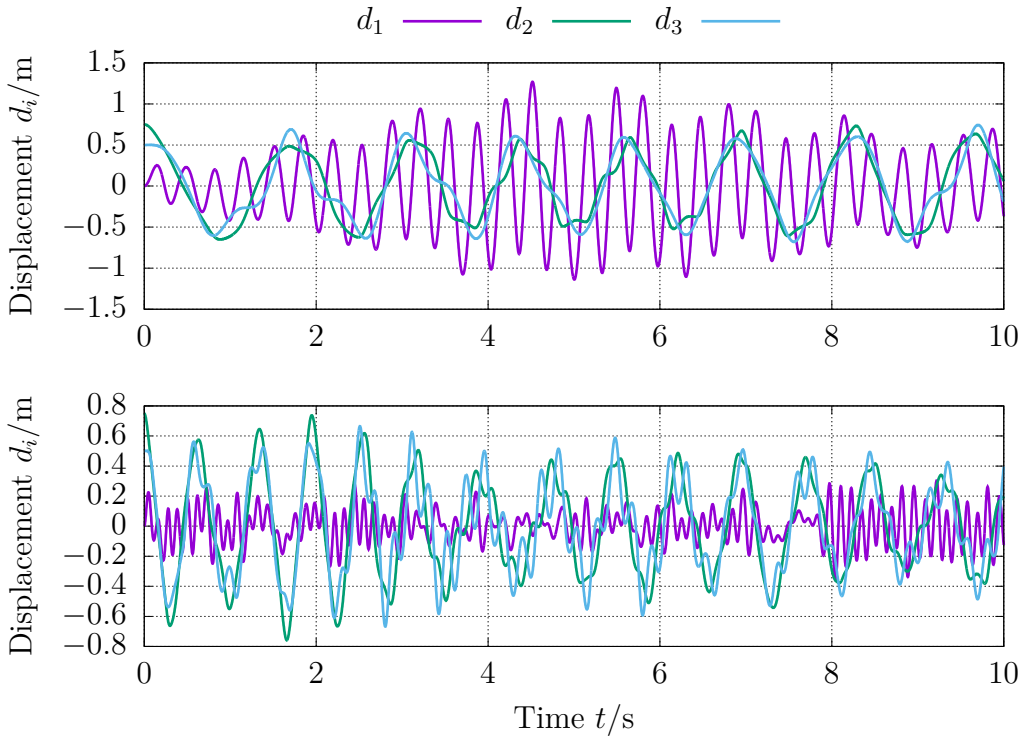


Figure 4.3: LF (top) and HF (bottom) displacement response of the mass-spring-damper system from Figure 4.2.

4.4.1 Polynomial Predictors

Polynomial predictors are based on the assumption that the solution $\mathbf{u}$ exhibits a polynomial behavior over time. From the solutions $\mathbf{u}_j, \mathbf{u}_{j-1}, \ldots$ in the previous time steps $t_j, t_{j-1}, \ldots$, a polynomial predictor of order p determines the ith component $u^0_{j+1,i}$ of the initial guess $\mathbf{u}^0_{j+1}$ from

$$u^0_{j+1,i} = \sum_{l=0}^{\min(p,j)} c_{j-l}\, t^l_{j+1} \,, \tag{4.13}$$

where the coefficients $c_j, \ldots, c_{j-\min(p,j)}$ are obtained from

$$\begin{pmatrix} c_j \\ c_{j-1} \\ \vdots \\ c_{j-\min(p,j)} \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & t_j & \cdots & t_j^{\min(p,j)} \\ 1 & t_{j-1} & \cdots & t_{j-1}^{\min(p,j)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & t_{j-\min(p,j)} & \cdots & t_{j-\min(p,j)}^{\min(p,j)} \end{pmatrix}^{-1}}_{=\boldsymbol{V}^{-1}} \begin{pmatrix} u_{j,i} \\ u_{j-1,i} \\ \vdots \\ u_{j-\min(p,j),i} \end{pmatrix} . \tag{4.14}$$

$\boldsymbol{V}^{-1}$ represents the inverse of the Vandermonde matrix $\boldsymbol{V}$ and can cheaply be computed explicitly as seldom $p > 3$. The term $\min(p, j)$ in the upper bound of the summation in (4.13) ensures that, in the first time increments, not more solutions than available at that time are taken into account. In the case that equidistant time increments $\Delta t = \text{const.}$ are used, Equation (4.13) produces the predictors listed in Table 4.2 up to an order $p = 3$. Note that a predictor of order j is applied in the case $j < p$. Predictors up to second order usually provide good initial solutions; higher-order polynomial prediction should be

Table 4.2: Polynomial predictors up to an order $p = 3$ for $\Delta t = \text{const.}$

Order p	Polynomial predictor
0	$\mathbf{u}_{j+1}^0 = \mathbf{u}_j$
1	$\mathbf{u}_{j+1}^0 = 2\mathbf{u}_j - \mathbf{u}_{j-1}$
2	$\mathbf{u}_{j+1}^0 = 3\mathbf{u}_j - 3\mathbf{u}_{j-1} + \mathbf{u}_{j-2}$
3	$\mathbf{u}_{j+1}^0 = 4\mathbf{u}_j - 6\mathbf{u}_{j-1} - \frac{20}{3}\mathbf{u}_{j-2} - \mathbf{u}_{j-3}$

used with due care, as the predicted values may become highly inaccurate due to Runge's phenomenon [139].

A polynomial predictor can also be combined with a linear extrapolation scheme. Assuming an equidistant time increment $\Delta t = \text{const.}$, we interpolate the results $\mathbf{u}_j$, $\mathbf{u}_{j-1}$, and $\mathbf{u}_{j-2}$ from the previous three time increments t_j, t_{j-1}, and t_{j-2} by a polynomial of second order and use the tangent at $\mathbf{u}_j$ for linear extrapolation to $t = t_{j+1}$, which leads us to

$$\mathbf{u}_{j+1}^0 = \frac{5}{2}\mathbf{u}_j - 2\mathbf{u}_{j-1} + \frac{1}{2}\mathbf{u}_{j-2} \,, \tag{4.15}$$

which has also been successfully applied in [36], for instance.

4.4.2 Predictors Based on Taylor Series Expansions

While polynomial predictors are only based on the solutions for the primary field variable from the previous time increments, predictors constructed from Taylor series expansions may enhance the quality of the prediction by taking additional physical considerations into account. For instance, in addition to the interface displacement $\mathbf{d}$, also the interface velocity $\dot{\mathbf{d}}$ and acceleration $\ddot{\mathbf{d}}$ can be evaluated as a result of the solution of the structural subproblem. From the Taylor series expansion about the previous displacement $\mathbf{d}_j$, we obtain the following approximation for the displacement $\mathbf{d}_{j+1}$ in the time increment t_{j+1}:

$$\mathbf{d}_{j+1} \approx \mathbf{d}_j + \Delta t \dot{\mathbf{d}}_j + \frac{\Delta t^2}{2}\ddot{\mathbf{d}}_j + \dots \,. \tag{4.16}$$

Truncating this Taylor series after the second term leads to the first-order predictor [130, p. 1218, 56, p. 52]

$$\mathbf{d}_{j+1}^0 = \mathbf{d}_j + \Delta t \dot{\mathbf{d}}_j \,. \tag{4.17}$$

A second-order predictor is obtained by truncating the series expansion (4.16) after the acceleration term [131, p. 3150, 56, p. 52]:

$$\mathbf{d}_{j+1}^0 = \mathbf{d}_j + \Delta t \dot{\mathbf{d}}_j + \frac{\Delta t^2}{2}\ddot{\mathbf{d}}_j \,. \tag{4.18}$$

4.4.3 Adaptive Predictors

The above predictors compute the initial solution from a linear combination of the primary field quantity and its derivatives from previous time steps, in which the coefficients are fixed throughout time. Adaptive predictors, in contrast, allow the coefficients to vary during the

simulation and, hence, may improve the quality of the prediction. To this end, the Taylor series-based predictors (4.17) and (4.18) are enhanced by premultiplying each of the terms by a constant, which is adjusted as we march forward in time. For instance, an adaptive second-order predictor reads [56, p. 52 sq.]

$$\mathbf{d}_{j+1}^0 = C_1\mathbf{d}_j + C_2\Delta t\dot{\mathbf{d}}_j + C_3\frac{\Delta t^2}{2}\ddot{\mathbf{d}}_j \ . \tag{4.19}$$

The coefficients C_i are determined by evaluating this expression at time t_j and by minimizing the expression

$$\underset{C_i}{\operatorname{argmin}} \left\| C_1\mathbf{d}_{j-1} + C_2\Delta t\dot{\mathbf{d}}_{j-1} + C_3\frac{\Delta t^2}{2}\ddot{\mathbf{d}}_{j-1} - \mathbf{d}_j \right\|_2 \ , \tag{4.20}$$

in which all kinematical quantities are known. In this context, it is assumed that the coefficients C_i do not vary much from one time step to another, Then, (4.19) generates a reasonable initial displacement solution $\mathbf{d}_{j+1}^0$ although the coefficients C_i are calculated by taking only quantities from previous time increments into account.

4.4.4 Comparison of Predictors

Based on the simple mechanical system introduced in Section 4.3 and the LF and HF displacement response depicted in Figure 4.3, we aim to assess the performance of the predictor schemes presented above. To investigate the predictor schemes in isolation, the displacement response is precomputed and afterwards the predictor scheme is applied to determine the prediction error in the jth time step from

$$e_{j+1} = \frac{\left\| \mathbf{d}_{j+1}^0 - \mathbf{d}_{j+1} \right\|_2}{\left\| \mathbf{d}_{j+1} \right\|_2} \ . \tag{4.21}$$

To obtain a scalar value characterizing the prediction error over the whole time range, we accumulate the errors e_{j+1} in each time increment for the n time increments $j = 0, \ldots, n-1$ and take the arithmetic mean of the resulting sum such that

$$e = \frac{1}{n}\sum_{j=0}^{n-1} e_{j+1} = \frac{1}{n}\sum_{j=0}^{n-1} \frac{\left\| \mathbf{d}_{j+1}^0 - \mathbf{d}_{j+1} \right\|_2}{\left\| \mathbf{d}_{j+1} \right\|_2} \ . \tag{4.22}$$

Results for the LF and the HF case are shown in Figure 4.4 and 4.5. In general, both diagrams indicate the same tendencies regarding the performance of the predictor schemes. As expected, all predictor schemes produce lower prediction errors e for decreasing time step sizes Δt. At least for the example considered here, a polynomial predictor of order $p = 0$ or $p = 1$ performs worse than the higher-order polynomial predictor of order $p = 2$. Except for large time step sizes, the linear extrapolation predictor (4.15) exhibits higher prediction errors than the polynomial predictor of second order. The Taylor series-based predictor of order $p = 1$ performs better than the polynomial predictors or the linear extrapolation predictor for larger time step sizes; however, as the time step size decreases, the first-order Taylor series-based predictor is outperformed by the second-order polynomial predictor, and the difference to the linear extrapolation predictor becomes marginal.

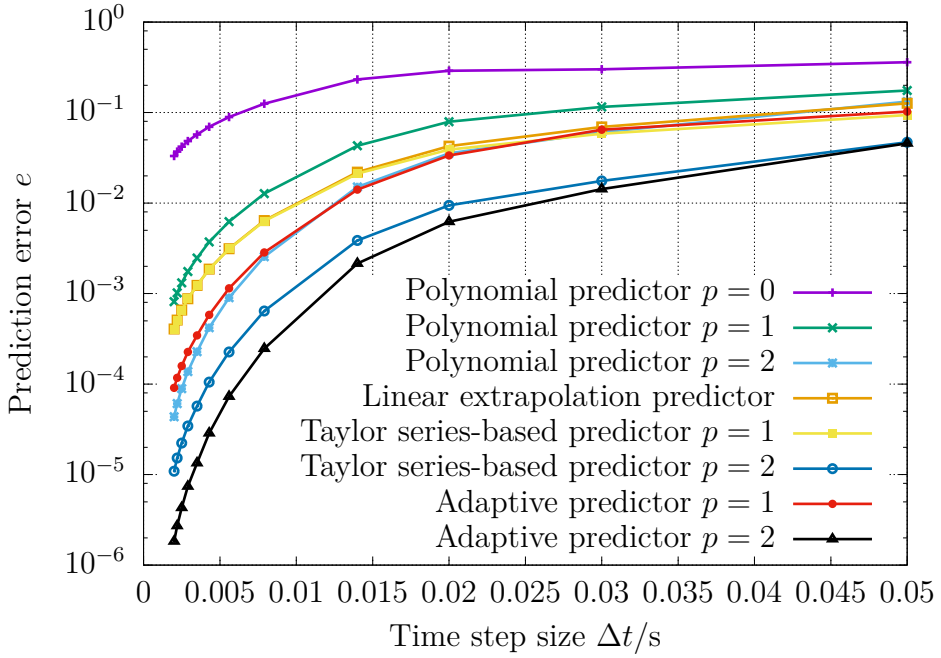


Figure 4.4: Comparison of the performance of the predictor schemes for the LF displacement response.

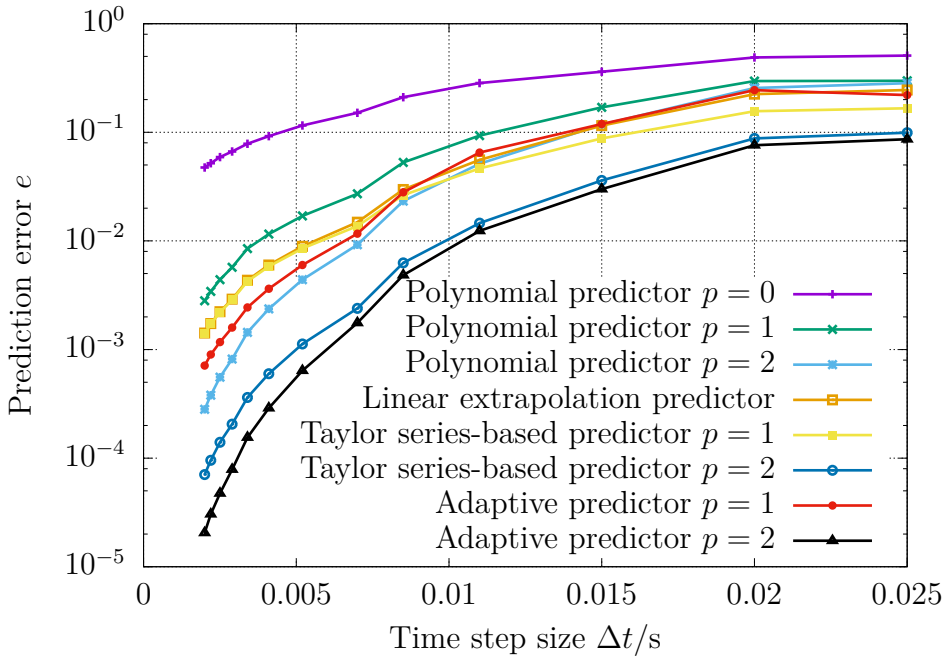


Figure 4.5: Comparison of the performance of the predictor schemes for the HF displacement response.

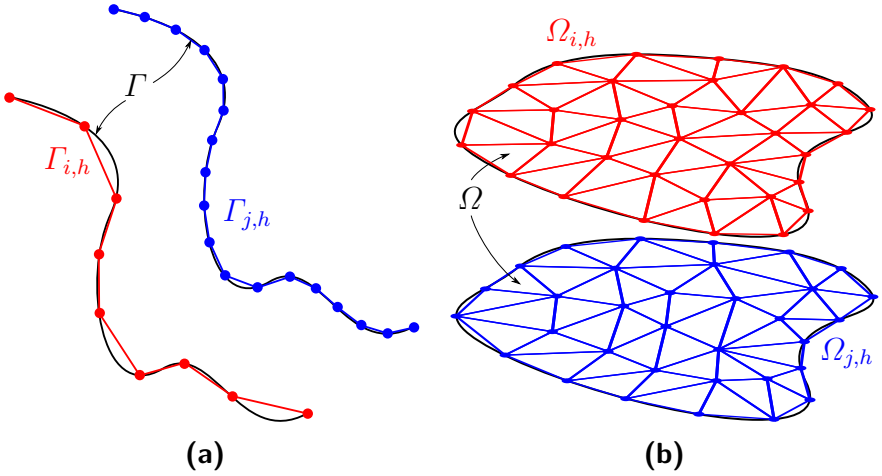


Figure 4.6: Non-conforming (a) boundary and (b) planar discretizations.

Throughout the whole range of considered time step sizes, the Taylor series-based predictor of order $p = 2$ yields a high-quality prediction and outperforms all previously mentioned predictor schemes. To further reduce the prediction error of the Taylor series-based predictors, we also investigate the adaptive variants of these predictors. The adaptive first-order predictor does not produce significantly better results than the first-order Taylor series-based predictor. For smaller time step sizes, the performance of the adaptive predictor is slightly better, while the opposite is true for larger time step sizes. In contrast, the adaptive predictor of order $p = 1$ may also lead to rather unsatisfactory results, especially for the HF displacement response. The adaptive predictor of second order, however, produces high-quality predictions throughout the entire range of considered time step sizes, and it is capable of reducing the already small prediction error of the second-order Taylor series-based predictor even further.

Although only a simple example has been considered in the comparison of the presented predictor schemes, it can be expected that the findings regarding their performance also apply to more complex problems involving a higher number of degrees of freedom. This is due to the fact that all considered predictor schemes operate on a degree-of-freedom basis and can hence be expected to obey a similar behavior independent of the number of degrees of freedom.

4.5 Interpolation Schemes

Since the spatial discretizations $\Gamma_{i,h}$ and $\Gamma_{j,h}$ in the case of a surface-coupled problem or $\Omega_{i,h}$ and $\Omega_{j,h}$ in the case of a volume-coupled problem are non-conforming in the general case, an interpolation scheme $\mathcal{I}_{i,j}$ needs to be applied to interpolate the quantities $\boldsymbol{u}(\boldsymbol{p}_1), \ldots, \boldsymbol{u}(\boldsymbol{p}_n)$ at the points $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n$ on the source discretization to the query points $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_m$ on the target discretization. Figure 4.6a depicts a typical scenario in a surface-coupled problem, where the boundaries Γ_i and Γ_j were discretized by non-conforming meshes. A situation representative for a two-dimensional volume-coupled problem is sketched in Figure 4.6b, where the coincident planar domains Ω_i and Ω_j were discretized by meshes of different size. In what follows, we consider the interpolation of a generic coupling quantity $\boldsymbol{u}$. In the case of an FSI problem, $\boldsymbol{u}$ equals the displacement $\boldsymbol{d}$ if the interpolation is performed

from the structural to the fluid domain – or the traction $\boldsymbol{t}$ if the interpolation is applied in the opposite direction.

Interpolation schemes can be categorized into mesh-independent and mesh-based interpolation schemes. The former group does not use any topological information of the spatial discretizations, whereas the interpolation schemes falling in the latter category aim to improve the quality of the interpolation by taking the mesh information into account. Consequently, access to specific mesh data structures is not required for mesh-independent interpolation schemes such as nearest neighbor interpolation, barycentric interpolation, inverse distance weighting, or interpolation by means of radial basis functions; only the evaluation points $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n$ on the source discretization and the query points $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_m$ on the target mesh are needed. Mesh-based interpolation schemes, in contrast, require the topological information from the computational mesh, and they need to be tailored to the data structure of each specific solver. However, as will be discussed in Section 4.5.7 in further detail, the improved interpolation accuracy usually outweighs this additional effort.

In addition to mesh-independent and mesh-based methods, interpolation schemes can also be classified into consistent and conservative schemes [164, p. 2421, 57, p. 683 sq.]. The difference between consistent and conservative schemes is best explained if the interpolation is written as a matrix-vector product

$$\begin{pmatrix} \boldsymbol{u}(\boldsymbol{q}_1) \\ \vdots \\ \boldsymbol{u}(\boldsymbol{q}_m) \end{pmatrix} = \underbrace{\begin{pmatrix} W_{11} & \cdots & W_{1n} \\ \vdots & \ddots & \vdots \\ W_{m1} & \cdots & W_{mn} \end{pmatrix}}_{=\mathbf{W}} \begin{pmatrix} \boldsymbol{u}(\boldsymbol{p}_1) \\ \vdots \\ \boldsymbol{u}(\boldsymbol{p}_n) \end{pmatrix} \tag{4.23}$$

involving the interpolation matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$. An interpolation method is termed *consistent* if the constant field $\boldsymbol{u}(\boldsymbol{p}_l) = \bar{\boldsymbol{u}}\ \forall\, l = 1, \ldots, n$ is retained by the interpolation such that $\boldsymbol{u}(\boldsymbol{q}_k) = \bar{\boldsymbol{u}}\ \forall\, k = 1, \ldots, m$. For this condition to hold, the sum of the entries in each *row* of $\mathbf{W}$ must be equal to one, in other words

$$\sum_{l=1}^{n} W_{kl} = 1 \qquad \forall\, k = 1, \ldots, m\ . \tag{4.24}$$

By contrast, a *conservative* interpolation scheme retains the sum of quantities such that $\sum_{l=1}^{n} \boldsymbol{u}(\boldsymbol{p}_l) = \sum_{k=1}^{m} \boldsymbol{u}(\boldsymbol{q}_k)$. In this case, the sum of the entries in each *column* of $\mathbf{W}$ are equal to one:

$$\sum_{k=1}^{m} W_{kl} = 1 \qquad \forall\, l = 1, \ldots, n\ . \tag{4.25}$$

Consistent interpolation schemes are typically used for the interpolation of continuous fields such as displacement fields or quantities scaled by length, area, or volume like traction, flux, or density. Conservative interpolation schemes, on the other hand, are applied for integral values such as forces or currents. In an FSI problem, the use of consistent interpolation schemes is appropriate if the displacement $\boldsymbol{d}$ and the traction $\boldsymbol{t}$ are chosen as coupling quantities. If the fluid solver integrates the traction on the coupling interface on its own, a conservative interpolation scheme is appropriate to interpolate the resulting force $\boldsymbol{f}$. It should be emphasized at this point that a consistent interpolation scheme should usually be preferred, as conservative interpolation schemes may introduce high local errors. Due

to this fact, consistent interpolation schemes are chosen for all the numerical examples presented in Chapter 6 and 7. The interpolation schemes presented in the remainder of this section are primarily constructed as consistent interpolation schemes; the corresponding conservative interpolation scheme, which is essentially shown for reasons of completeness, can often be derived from the consistent version.

4.5.1 Nearest Neighbor Interpolation

A simple yet effective and often-used interpolation scheme is the nearest neighbor interpolation scheme depicted in Algorithm 2. Given a query point $\boldsymbol{q}$, the set of source points

1: **function** NearestNeighborInterpolation(set of source points $\mathcal{P} = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n\}$, function values $\boldsymbol{u}(\boldsymbol{p}_1), \ldots, \boldsymbol{u}(\boldsymbol{p}_n)$, query point $\boldsymbol{q}$)
2: Find $\boldsymbol{p}_j \in \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n\}$ such that $d(\boldsymbol{p}_j, \boldsymbol{q})$ becomes minimal
3: **return** $\boldsymbol{u}(\boldsymbol{q}) := \boldsymbol{u}(\boldsymbol{p}_j)$
4: **end function**

Algorithm 2: Nearest neighbor interpolation.

$\mathcal{P} = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n\}$ is searched for the point $\boldsymbol{p}_j$, for which the distance metric $d(\boldsymbol{p}_j, \boldsymbol{q})$ assumes a minimum. For the scenarios considered in this work, the Euclidean norm $d(\boldsymbol{p}_j, \boldsymbol{q}) := \|\boldsymbol{p}_j - \boldsymbol{q}\|_2$ is the most natural choice. Once the nearest neighbor $\boldsymbol{p}_j$ has been determined, the function value $\boldsymbol{u}(\boldsymbol{q})$ is set to the function value $\boldsymbol{u}(\boldsymbol{p}_j)$. It is obvious that the quality of the interpolation increases with an increasing number of source points n.

Different strategies may be employed to determine the nearest neighbor $\boldsymbol{p}_j$ of a query point $\boldsymbol{q}$. A naive approach is to traverse the point set $\{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n\}$ in a sequential fashion, evaluate the distance metric $d(\boldsymbol{p}_l, \boldsymbol{q})$ for each source point $\boldsymbol{p}_l$, and to check whether $d(\boldsymbol{p}_l, \boldsymbol{q})$ undershoots any previously determined minimum distance. For the application in other interpolation schemes, it is useful to generalize the idea of the nearest neighbor search to the problem of determining the k-nearest neighbors. A possible implementation based on a naive linear search is outlined in Algorithm 3. First, we initialize $d_{\max} := \infty$ and $\mathcal{R} := \emptyset$. Looping through the source points $\{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n\}$, the distance $d(\boldsymbol{p}_l, \boldsymbol{q})$ is calculated for each point $\boldsymbol{p}_l$. If $d(\boldsymbol{p}_l, \boldsymbol{q}) < d_{\max}$, the point $\boldsymbol{p}_l$ is added to the set of k-nearest neighbor candidates $\mathcal{R}$ such that all elements in $\mathcal{R}$ are sorted by their distance to the query point $\boldsymbol{q}$ in ascending order. If the number of k-nearest neighbor candidates in $\mathcal{R}$ exceeds k, the source point with the greatest distance is dropped from the set. Next, the distance $d_{\max}$ is set to the greatest distance of a source point in $\mathcal{R}$ from the query point $\boldsymbol{q}$. Then, the cycle starts over again until all n source points have been examined.

Especially if a large number of query points is considered, the effort associated to the linear k-nearest neighbor search becomes prohibitively high – and it is wise to resort to more effective search strategies, which are usually based on space-partitioning methods. Probably one of the simplest is the k-d tree, first proposed in [18], which can, for instance, be generated by means of Algorithm 4. The application of a k-d tree to partition a two-dimensional space for an effective nearest-neighbor search is illustrated in Figure 4.7 for $n = 6$ source points $\mathcal{P} = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_6\}$. Given a query point $\boldsymbol{q}$, the k-d tree is then traversed to find the nearest neighbor in the source point set $\mathcal{P}$. The generic procedure is outlined in Algorithm 5. Initially, N is the root node of the k-d tree, the tree level is set to $\ell := 1$,

```
1: function LinearKNearestNeighborSearch(set of source points 𝒫 = {p_1, ..., p_n}, query point q)
2:     d_max := ∞, ℛ := ∅
3:     for l = 1 : n do
4:         if d(p_l, q) < d_max then
5:             ℛ := ℛ ∪ {p_l}
6:             Sort ℛ such that ∀ p_i, p_j ∈ ℛ it holds that d(p_i, q) ≤ d(p_j, q) if i < j
7:             if card(ℛ) > k then
8:                 Delete the last element of ℛ
9:                 d_max := max_{p_i ∈ ℛ} d(p_i, q)
10:            end if
11:        end if
12:    end for
13:    return ℛ
14: end function
```

Algorithm 3: Linear k-nearest neighbor search.

```
1: function KdTree(set of source points 𝒫 = {p_1, ..., p_n}, tree level ℓ)
2:     Determine axis a := mod(ℓ − 1, d) + 1 for sorting
3:     Sort points in 𝒫 along axis a
4:     Determine the median p̃ of 𝒫
5:     Split 𝒫 at the median p̃ to form a left subset 𝒫_left and a right subset 𝒫_right
6:     Increase the tree level ℓ := ℓ + 1
7:     Create a tree node N
8:     if 𝒫_left ≠ ∅ then
9:         Create left child node N.left := KdTree(𝒫_left, ℓ)
10:    end if
11:    if 𝒫_right ≠ ∅ then
12:        Create right child node N.right := KdTree(𝒫_right, ℓ)
13:    end if
14:    return N
15: end function
```

Algorithm 4: Function KdTree for the recursive setup of a k-d tree in d-dimensional space.

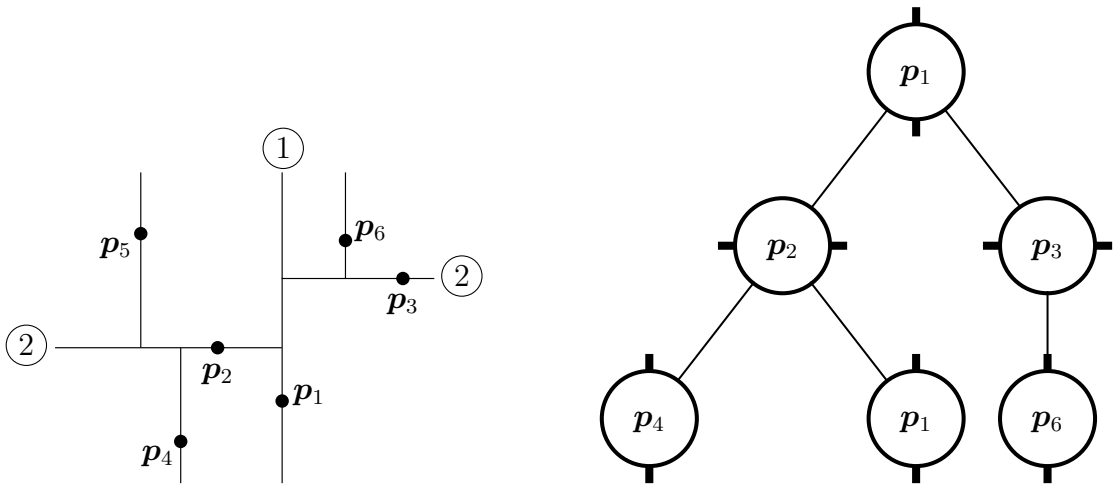


Figure 4.7: k-d tree.

for the set of nearest neighbors $\mathcal{R} := \emptyset$, and the maximum distance for a point $\boldsymbol{p}_l \in \mathcal{P}$ to be considered as a candidate for the k-nearest neighbors is initialized to $d_{\max} := \infty$. First, it is necessary to determine the axis a along which the source points have been sorted on the given tree level ℓ. Next, we compute the distance $d(\boldsymbol{p}_l, \boldsymbol{q})$ between the point $\boldsymbol{p}_l$ associated to the tree node N and the query point $\boldsymbol{q}$. If $d(\boldsymbol{p}_l, \boldsymbol{q}) < d_{\max}$, the point $\boldsymbol{p}_l$ is added to the current set of k-nearest neighbor candidates $\mathcal{R}$ such that the elements in $\mathcal{R}$ are sorted by their distance to the query point $\boldsymbol{q}$ in ascending order, i.e., the element closest to $\boldsymbol{q}$ comes first. If the number of candidates in $\mathcal{R}$ exceeds k, then the element farthest away from $\boldsymbol{q}$ is discarded from the set. Furthermore, the distance $d_{\max}$ is updated to the maximum distance of a source point in $\mathcal{R}$ to $\boldsymbol{q}$. Then, we calculate the distance d' between $\boldsymbol{p}_l$ and the query point $\boldsymbol{q}$ along axis a. If $d' < 0$, then the *left* branch is the closer one, whereas $d' \geq 0$ indicates that the *right* branch should be traversed first. Subsequently, the tree level ℓ is incremented – and the closer branch, if not empty, is traversed. The farther branch only needs to be investigated if it is non-empty and if the hypersphere of radius $d_{\max}$ intersects the splitting hyperplane. The procedure is graphically visualized in Figure 4.8 for the simple example with $n = 6$ source points, depicted in Figure 4.7, and the search for a single nearest neighbor, i.e., $k = 1$. In the first step illustrated in Figure 4.8a, the tree level is $\ell = 1$ and the distance $d_{\max}$ (initially set to $d_{\max} := \infty$) is updated to $d_{\max} := \|\boldsymbol{p}_1 - \boldsymbol{q}\|_2$. Since the sphere surrounding $\boldsymbol{q}$ intersects the left half-plane, the left branch of the k-d tree from Figure 4.7 must be investigated. As sketched in Figure 4.8b, we compute the distance $d(\boldsymbol{p}_2, \boldsymbol{q})$ between $\boldsymbol{p}_2$ and the query point $\boldsymbol{q}$ and update $d_{\max} := d(\boldsymbol{p}_2, \boldsymbol{q})$ as the current best distance. Descending further down the left branch of the k-d tree, as shown in Figure 4.8c, the points $\boldsymbol{p}_4$ and $\boldsymbol{p}_5$ are discarded due to the fact that the point $\boldsymbol{p}_2$, which has already been visited, exhibits a closer distance to $\boldsymbol{q}$. In the last step in Figure 4.8d, it is checked whether, once again starting from the root node, the right branch of the tree needs to be visited. Yet, since the current best-estimate sphere does not intersect the splitting plane corresponding to tree level $\ell = 1$, the right branch is discarded and $\boldsymbol{p}_2$ is eventually identified as the nearest neighbor.

The nearest neighbor interpolation outlined in Algorithm 2 is a *consistent* interpolation scheme. This fact is easily verified for the simple example sketched in Figure 4.9. If a constant function is to be interpolated and, thus, $\boldsymbol{u}(\boldsymbol{p}_l) = \bar{\boldsymbol{u}}\ \forall\, l = 1, \ldots, 4$, the function

1: **procedure** KdTreeKNearestNeighborSearch(tree node N, query point $\boldsymbol{q}$, number of nearest neighbors k, tree level ℓ, k-nearest neighbor candidates $\mathcal{R}$, distance $d_{\max}$)
2: Determine axis $a := \mathrm{mod}(\ell - 1, k) + 1$
3: Compute distance $d(\boldsymbol{p}_l, \boldsymbol{q})$, where $\boldsymbol{p}_l$ is the point associated to tree node N
4: **if** $d(\boldsymbol{p}_l, \boldsymbol{q}) < d_{\max}$ **then**
5: $\mathcal{R} := \mathcal{R} \cup \{\boldsymbol{p}_l\}$
6: Sort $\mathcal{R}$ such that $\forall\, \boldsymbol{p}_i, \boldsymbol{p}_j \in \mathcal{R}$ it holds that $d(\boldsymbol{p}_i, \boldsymbol{q}) \leq d(\boldsymbol{p}_j, \boldsymbol{q})$ if $i < j$
7: **if** $\mathrm{card}(\mathcal{R}) > k$ **then**
8: Delete the last element of $\mathcal{R}$
9: $d_{\max} := \max_{\boldsymbol{p}_i \in \mathcal{R}} d(\boldsymbol{p}_i, \boldsymbol{q})$
10: **end if**
11: **end if**
12: Compute distance in a-direction: $d' := p_{l,a} - q_a$
13: **if** $d' < 0$ **then**
14: $C_{\text{close}} := N$.left, $C_{\text{far}} := N$.right
15: **else**
16: $C_{\text{close}} := N$.right, $C_{\text{far}} := N$.left
17: **end if**
18: Increase the tree level $\ell := \ell + 1$
19: **if** $C_{\text{close}} \neq$ empty **then** ▷ Traverse closer branch
20: KdTreeKNearestNeighborSearch(C_{close}, $\boldsymbol{q}$, k, ℓ, $\mathcal{R}$, $d_{\max}$)
21: **end if**
22: **if** $C_{\text{far}} \neq$ empty $\wedge\, |d'| < d_{\max}$ **then** ▷ Traverse farther branch
23: KdTreeKNearestNeighborSearch(C_{far}, $\boldsymbol{q}$, k, ℓ, $\mathcal{R}$, $d_{\max}$)
24: **end if**
25: **end procedure**

Algorithm 5: Procedure KdTreeKNearestNeighborSearch for the recursive traversal of a k-d tree.

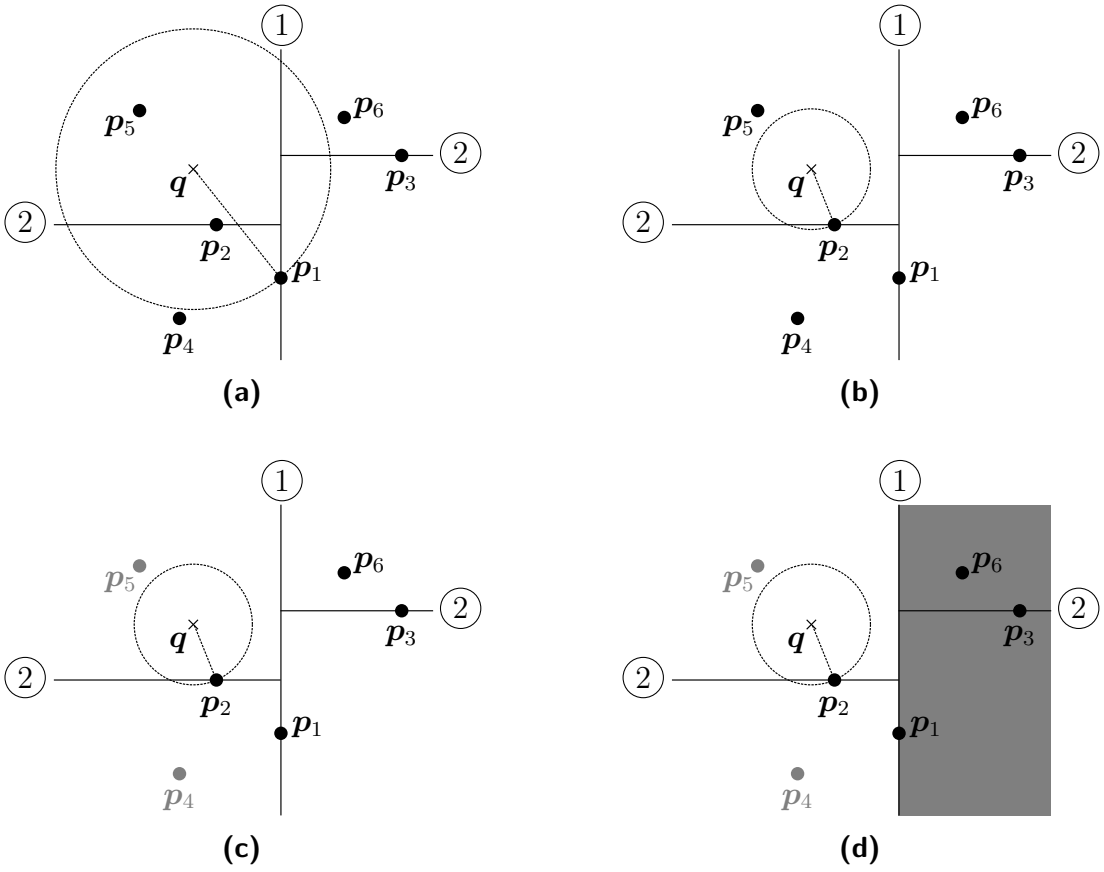


Figure 4.8: k-d tree traversal.

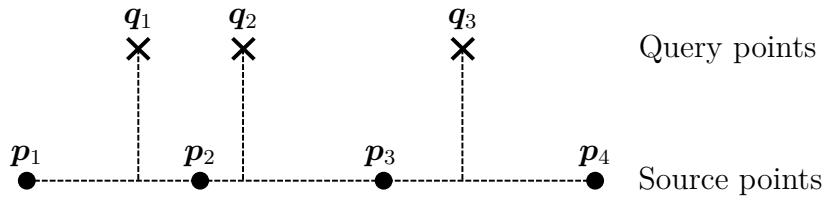


Figure 4.9: Consistent nearest neighbor interpolation.

values at the query points become $\boldsymbol{u}(\boldsymbol{q}_1) = \boldsymbol{u}(\boldsymbol{p}_2) = \bar{\boldsymbol{u}}$, $\boldsymbol{u}(\boldsymbol{q}_2) = \boldsymbol{u}(\boldsymbol{p}_2) = \bar{\boldsymbol{u}}$, and $\boldsymbol{u}(\boldsymbol{q}_3) = \boldsymbol{u}(\boldsymbol{p}_3) = \bar{\boldsymbol{u}}$; the constant function is hence retained by the interpolation scheme. In matrix notation, the interpolation scheme for this example reads

$$\begin{pmatrix} \boldsymbol{u}(\boldsymbol{q}_1) \\ \boldsymbol{u}(\boldsymbol{q}_2) \\ \boldsymbol{u}(\boldsymbol{q}_3) \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \boldsymbol{u}(\boldsymbol{p}_1) \\ \boldsymbol{u}(\boldsymbol{p}_2) \\ \boldsymbol{u}(\boldsymbol{p}_3) \\ \boldsymbol{u}(\boldsymbol{p}_4) \end{pmatrix} . \tag{4.26}$$

Apparently, the requirement (4.24) for consistent interpolation schemes – that the entries in each row of the interpolation matrix add up to one – is fulfilled. In the corresponding *conservative* interpolation scheme, the function values at the query points amount to $\boldsymbol{u}(\boldsymbol{q}_1) = \boldsymbol{u}(\boldsymbol{p}_1)$, $\boldsymbol{u}(\boldsymbol{q}_2) = \boldsymbol{u}(\boldsymbol{p}_2)$, $\boldsymbol{u}(\boldsymbol{q}_3) = \boldsymbol{u}(\boldsymbol{p}_3) + \boldsymbol{u}(\boldsymbol{p}_4)$. In matrix form, this reads

$$\begin{pmatrix} \boldsymbol{u}(\boldsymbol{q}_1) \\ \boldsymbol{u}(\boldsymbol{q}_2) \\ \boldsymbol{u}(\boldsymbol{q}_3) \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} \boldsymbol{u}(\boldsymbol{p}_1) \\ \boldsymbol{u}(\boldsymbol{p}_2) \\ \boldsymbol{u}(\boldsymbol{p}_3) \\ \boldsymbol{u}(\boldsymbol{p}_4) \end{pmatrix} . \tag{4.27}$$

Obviously, the criterion (4.25) is fulfilled, and the sum of the entries in each column is equal to one.

For computer implementation, it turns out useful to construct a conservative interpolation scheme by reusing the implementation for the consistent version. To this end, we reverse source and query points, determine the consistent interpolation weights, and transpose the resulting interpolation matrix to obtain the interpolation matrix for the conservative scheme. Of course, if the interpolation scheme has compact support, the interpolation matrix is never constructed explicitly but rather stored as a set of sets $\{\mathcal{W}_1, \ldots, \mathcal{W}_m\}$, where each set $\mathcal{W}_i = \{\ldots, (\boldsymbol{p}_j, w_j), \ldots\}$ contains pairs $(\boldsymbol{p}_j, w_j)$ associating an interpolation weight w_j to a source point $\boldsymbol{p}_j$ such that the function value $\boldsymbol{u}(\boldsymbol{q}_i)$ can be determined by summing the terms $w_j \boldsymbol{u}(\boldsymbol{p}_j)$ for all pairs $(\boldsymbol{p}_j, w_j)$ in the set $\mathcal{W}_i$. Algorithm 6 illustrates the reuse of the implementation for the calculation of the consistent interpolation weights to determine the interpolation weights of the corresponding conservative interpolation scheme.

4.5.2 Barycentric Interpolation

Barycentric interpolation schemes (see, e.g., [167, p. 116 sq.]) can be categorized into surface and volumetric interpolation schemes. For surface interpolation in two-dimensional space, we search the set of source points $\mathcal{P} = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n\}$ to determine the points $\boldsymbol{p}_a$ and $\boldsymbol{p}_b$, which span a line that exhibits minimum Euclidean distance to the query point $\boldsymbol{q}$ in the sense that no other pair of points $\boldsymbol{p}_i, \boldsymbol{p}_j \in \mathcal{P}, i \neq j$ spans a line closer to $\boldsymbol{q}$. It should be noted that the points $\boldsymbol{p}_a$ and $\boldsymbol{p}_b$ are *not* necessarily the nearest neighbors of $\boldsymbol{q}$, as illustrated in Figure 4.10.

If the source discretization, from which the source points $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n$ have been extracted, is convex, it suffices to perform a Delaunay triangulation of the source point set and to remove all inner edges such that only the outer edges remain, see Figure 4.11. A query point $\boldsymbol{q}$ is then successively projected to the outer edges of the Delaunay triangulation to identify the points $\boldsymbol{p}_a$ and $\boldsymbol{p}_b$, which will later be used for the interpolation. If the edges

1: **function** ConservativeInterpolationWeights(set of source points $\mathcal{P} = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n\}$, set of query points $\mathcal{Q} = \{\boldsymbol{q}_1, \ldots, \boldsymbol{q}_m\}$)
2: Reverse source and query points such that $\mathcal{P}' := \mathcal{Q}$ and $\mathcal{Q}' := \mathcal{P}$
3: Determine consistent interpolation weights for $\mathcal{P}', \mathcal{Q}'$
4: **for** $l = 1 : n$ **do**
5: Consider the set of interpolation weights $\mathcal{W}'_l = \{\ldots, (\boldsymbol{q}_i, w_i), \ldots\}$ associated to $\boldsymbol{p}_l$
6: **for all** $(\boldsymbol{q}_i, w_i) \in \mathcal{W}'_l$ **do**
7: Consider the set of interpolation weights $\mathcal{W}_i = \{\ldots, (\boldsymbol{p}_j, w_j), \ldots\}$ associated to $\boldsymbol{q}_i$, initialized to $\mathcal{W}_i := \emptyset$ on first access
8: Set $\mathcal{W}_i := \mathcal{W}_i \cup \{(\boldsymbol{p}_l, w_i)\}$
9: **end for**
10: **end for**
11: **return** $\{\mathcal{W}_1, \ldots, \mathcal{W}_m\}$
12: **end function**

Algorithm 6: Reuse of the implementation for the calculation of consistent interpolation weights to determine the interpolation weights of the corresponding conservative interpolation scheme.

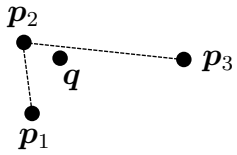


Figure 4.10: Situation where $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ are the nearest neighbors of $\boldsymbol{q}$ but the closest line is spanned by $\boldsymbol{p}_2$ and $\boldsymbol{p}_3$.

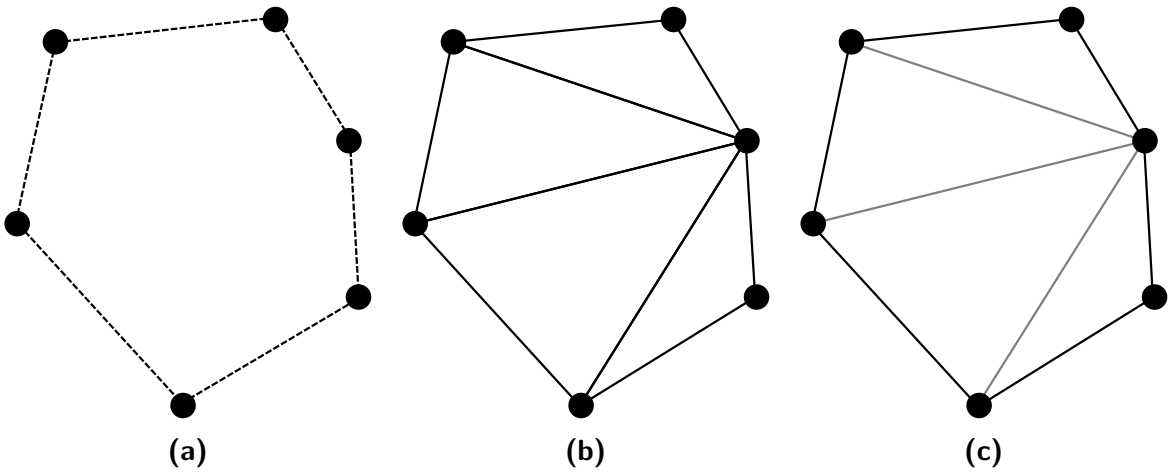


Figure 4.11: (a) Source points and original surface discretization indicated by dashed lines, (b) Delaunay triangulation of the convex hull of the source points, and (c) removal of the inner edges of the Delaunay triangulation (colored in gray).

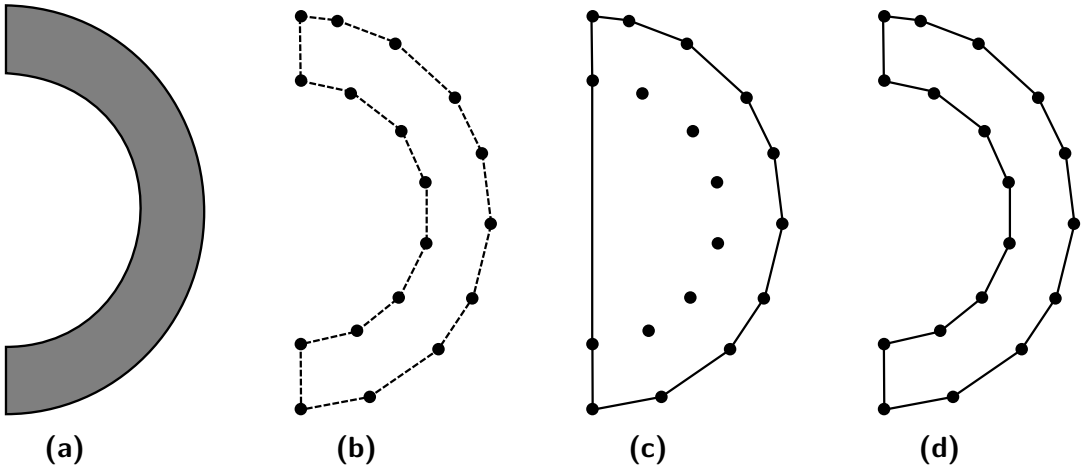


Figure 4.12: (a) Geometry, (b) boundary discretization, (c) α-shape for a shrink factor $s = 0$, and (d) α-shape for a shrink factor $s = 0.8$.

are parameterized by a local parameter $\xi \in [-1, 1]$, the projection technique outlined in Section 4.5.5 can readily be employed for this task. Since a Delaunay triangulation always triangulates the convex hull of a set of points, this method is inappropriate for concave surfaces. For sufficiently smooth boundaries, it helps to compute an α-shape [42] of the source point set and to apply a suitable shrink factor s. To illustrate the idea of α-shapes in $\mathbb{R}^d$, we imagine the construction of an infinite number of spheres of minimum radius $1/\alpha$ without enclosing any of the points in the considered point set. Next, we subtract the space occupied by these spheres from $\mathbb{R}^d$. The linear approximation of the remaining shape will then define the α-shape associated to the point set. Based on this, the role of the shrink factor s is explained as follows. First, we compute an α-shape for the source point set and determine the critical α-value α_{crit}, which still leads to a simply-connected region enclosing the considered point set. Subsequently, all α-values exceeding α_{crit} are extracted, and the shrink factor s is used to select a single $\alpha \in [\alpha_{\text{crit}}, \alpha_{\max}]$. Ranging from 0 to 1, a shrink factor $s = 0$ corresponds to the convex hull of the point set, and $s = 1$ produces a compact envelope around the points. The procedure is shown in Figure 4.12 for the source points stemming from the boundary discretization of a C-shaped domain. The reader is referred to the documentation of [160] for further details regarding the practical application of α-shapes.

In many situations, however, the boundary is not sufficiently smooth to expect a shrunk α-shape to resemble the shape of the original boundary discretization. Another approach to circumvent this substantial disadvantage is to find the $k \geq 2$ nearest neighbors and generate a local Delaunay triangulation of this point set. Here, k is a user-defined parameter. It should be chosen large enough to ensure the proper construction of a Delaunay triangulation, containing edges sufficiently close to the query point $\boldsymbol{q}$ and small enough so as not to compromise the computational efficiency of the interpolation scheme. For well-behaved boundary discretizations, $k = 3$ or $k = 4$ will usually suffice to construct an interpolation scheme that is both efficient and accurate. After the k-nearest neighbors have been triangulated, the query point $\boldsymbol{q}$ is projected to the closest edge resulting in the projected point $\boldsymbol{q}^*$, and the function value $\boldsymbol{u}(\boldsymbol{q})$ is determined by computing the function

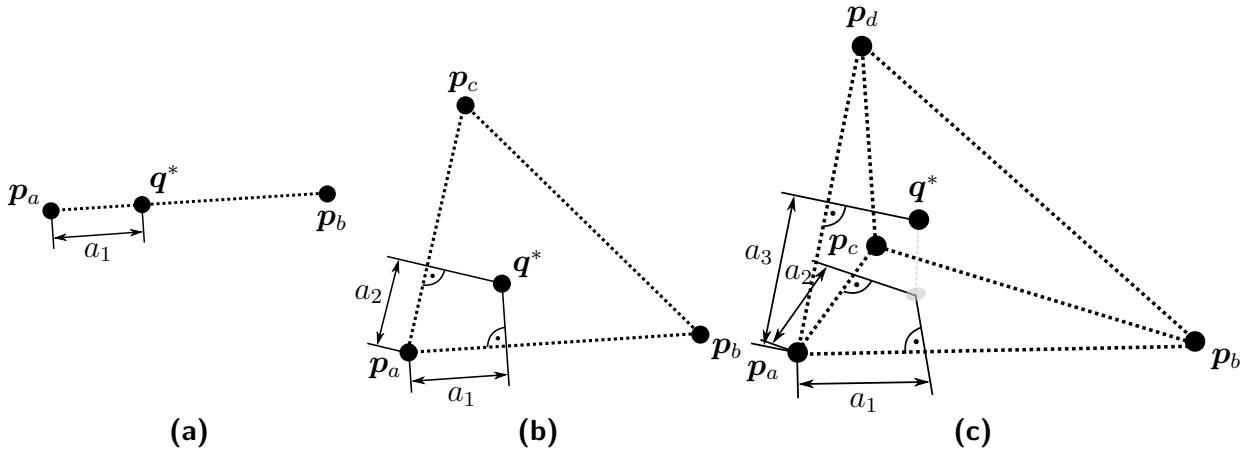


Figure 4.13: Barycentric interpolation on a (a) line (volume interpolation in one-dimensional space or surface interpolation in two-dimensional space), (b) triangle (volume interpolation in two-dimensional space or surface interpolation in three-dimensional space), or (c) tetrahedral (volume interpolation in three-dimensional space).

value $\boldsymbol{u}(\boldsymbol{q}^*)$ by linear interpolation between the points $\boldsymbol{p}_a, \boldsymbol{p}_b$ such that

$$\begin{aligned}\boldsymbol{u}(\boldsymbol{q}) = \boldsymbol{u}(\boldsymbol{q}^*) &= \boldsymbol{u}(\boldsymbol{p}_a) + a_1 \frac{\boldsymbol{u}(\boldsymbol{p}_b) - \boldsymbol{u}(\boldsymbol{p}_a)}{\|\boldsymbol{p}_b - \boldsymbol{p}_a\|_2} \\ &= \underbrace{\left(1 - \frac{a_1}{\|\boldsymbol{p}_b - \boldsymbol{p}_a\|_2}\right)}_{=\alpha} \boldsymbol{u}(\boldsymbol{p}_a) + \underbrace{\frac{a_1}{\|\boldsymbol{p}_b - \boldsymbol{p}_a\|_2}}_{=\beta} \boldsymbol{u}(\boldsymbol{p}_b) \ ,\end{aligned} \tag{4.28}$$

see also Figure 4.13a. In practice, the line spanned by $\boldsymbol{p}_a$ and $\boldsymbol{p}_b$ will not necessarily be the line with minimum Euclidean distance to the query point $\boldsymbol{q}$, but it will quite certainly suffice for an accurate interpolation of the function value $\boldsymbol{u}(\boldsymbol{q})$ due to the positive properties of the Delaunay triangulation (such as minimum angle maximization, for instance).

To perform a barycentric surface interpolation in three-dimensional space, we adopt a similar procedure to that in two-dimensional space. Instead of a line, we are searching for the triangle spanned by the points $\boldsymbol{p}_a$, $\boldsymbol{p}_b$, and $\boldsymbol{p}_c$, which has minimum Euclidean distance to the query point $\boldsymbol{q}$. Also here, in order to obtain a high-quality interpolation, it does *not* suffice to find the three nearest neighbors. Rather, the preferable approach is to search for the $k > 3$ nearest neighbors to generate a local Delaunay triangulation. Choosing k in the range from 7 to 10 will usually provide an acceptable compromise to ensure the construction of a proper Delaunay triangulation and to also retain computational efficiency. Having determined the k-nearest neighbors and their Delaunay triangulation, the query point $\boldsymbol{q}$ is successively projected to each of the triangles in the tessellation to obtain $\boldsymbol{q}^*$. The triangle with minimum Euclidean distance to $\boldsymbol{q}$ is then selected for interpolation:

$$\begin{aligned}\boldsymbol{u}(\boldsymbol{q}) = \boldsymbol{u}(\boldsymbol{q}^*) &= \boldsymbol{u}(\boldsymbol{p}_a) + a_1 \frac{\boldsymbol{u}(\boldsymbol{p}_b) - \boldsymbol{u}(\boldsymbol{p}_a)}{\|\boldsymbol{p}_b - \boldsymbol{p}_a\|_2} + a_2 \frac{\boldsymbol{u}(\boldsymbol{p}_c) - \boldsymbol{u}(\boldsymbol{p}_a)}{\|\boldsymbol{p}_c - \boldsymbol{p}_a\|_2} \\ &= \underbrace{\left(1 - \frac{a_1}{\|\boldsymbol{p}_b - \boldsymbol{p}_a\|_2} - \frac{a_2}{\|\boldsymbol{p}_c - \boldsymbol{p}_a\|_2}\right)}_{=\alpha} \boldsymbol{u}(\boldsymbol{p}_a) + \underbrace{\frac{a_1}{\|\boldsymbol{p}_b - \boldsymbol{p}_a\|_2}}_{=\beta} \boldsymbol{u}(\boldsymbol{p}_b) + \underbrace{\frac{a_2}{\|\boldsymbol{p}_c - \boldsymbol{p}_a\|_2}}_{=\gamma} \boldsymbol{u}(\boldsymbol{p}_c) \ .\end{aligned} \tag{4.29}$$

Ericson [46, p. 47 sq.] mentions an efficient method to compute the barycentric coordinates α, β, γ, which is based on solving a linear system using Cramer's rule and is close to optimal in terms of computational efficiency, see Algorithm 7.

1: **function** BarycentricTriangleCoordinates(source points $\boldsymbol{p}_a$, $\boldsymbol{p}_b$, $\boldsymbol{p}_c$, projected query point $\boldsymbol{q}^*$)
2: $\quad \boldsymbol{v}_1 := \boldsymbol{p}_b - \boldsymbol{p}_a$, $\boldsymbol{v}_2 := \boldsymbol{p}_c - \boldsymbol{p}_a$, $\boldsymbol{v}_3 := \boldsymbol{q}^* - \boldsymbol{p}_a$
3: $\quad D_{11} := \boldsymbol{v}_1 \cdot \boldsymbol{v}_1$, $D_{12} := \boldsymbol{v}_1 \cdot \boldsymbol{v}_2$, $D_{22} := \boldsymbol{v}_2 \cdot \boldsymbol{v}_2$, $D_{31} := \boldsymbol{v}_3 \cdot \boldsymbol{v}_1$, $D_{32} := \boldsymbol{v}_3 \cdot \boldsymbol{v}_2$
4: $\quad D := D_{11}D_{22} - D_{12}D_{12}$, $D^{-1} := 1/D$
5: $\quad \alpha := D^{-1}(D_{22}D_{31} - D_{12}D_{32})$
6: $\quad \beta := D^{-1}(D_{11}D_{32} - D_{12}D_{31})$
7: $\quad \gamma := 1 - \alpha - \beta$
8: $\quad$ **return** α, β, γ
9: **end function**

Algorithm 7: Efficient computation of the barycentric coordinates α, β, γ for the projected query point $\boldsymbol{q}^*$ with respect to the triangle spanned by the points $\boldsymbol{p}_a$, $\boldsymbol{p}_b$, $\boldsymbol{p}_c$.

The strategies followed for surface interpolation are, after minor modifications, also applicable to volume interpolation. For "volume" interpolation in two-dimensional space (or, in other words, planar interpolation), we again perform a local Delaunay tessellation of the k-nearest neighbors of a query point $\boldsymbol{q}$, where $k \geq 3$, and interpolate the function value according to Equation (4.29). As opposed to surface interpolation, where $\boldsymbol{q} \neq \boldsymbol{q}^*$ holds almost always if the surface is curved and the source and target discretizations are non-conforming, $\boldsymbol{q} = \boldsymbol{q}^*$ is the most likely case in volume interpolation. Only in the case where $\boldsymbol{q}$ lies outside the original source discretization, $\boldsymbol{q} \neq \boldsymbol{q}^*$ applies as $\boldsymbol{q}$ must be projected to the surface of the local Delaunay triangulation.

Eventually, barycentric volume interpolation in three-dimensional space is performed by local Delaunay triangulation of the $k \geq 4$ nearest neighbors of the query point $\boldsymbol{q}$ and by applying the interpolation formula

$$\begin{aligned}
\boldsymbol{u}(\boldsymbol{q}) = \boldsymbol{u}(\boldsymbol{q}^*) &= \boldsymbol{u}(\boldsymbol{p}_a) + a_1 \frac{\boldsymbol{u}(\boldsymbol{p}_b) - \boldsymbol{u}(\boldsymbol{p}_a)}{\|\boldsymbol{p}_b - \boldsymbol{p}_a\|_2} + a_2 \frac{\boldsymbol{u}(\boldsymbol{p}_c) - \boldsymbol{u}(\boldsymbol{p}_a)}{\|\boldsymbol{p}_c - \boldsymbol{p}_a\|_2} + a_3 \frac{\boldsymbol{u}(\boldsymbol{p}_d) - \boldsymbol{u}(\boldsymbol{p}_a)}{\|\boldsymbol{p}_d - \boldsymbol{p}_a\|_2} \\
&= \underbrace{\left(1 - \frac{a_1}{\|\boldsymbol{p}_b - \boldsymbol{p}_a\|_2} - \frac{a_2}{\|\boldsymbol{p}_c - \boldsymbol{p}_a\|_2} - \frac{a_3}{\|\boldsymbol{p}_d - \boldsymbol{p}_a\|_2}\right)}_{=\alpha} \boldsymbol{u}(\boldsymbol{p}_a) + \underbrace{\frac{a_1}{\|\boldsymbol{p}_b - \boldsymbol{p}_a\|_2}}_{=\beta} \boldsymbol{u}(\boldsymbol{p}_b) \\
&\quad + \underbrace{\frac{a_2}{\|\boldsymbol{p}_c - \boldsymbol{p}_a\|_2}}_{=\gamma} \boldsymbol{u}(\boldsymbol{p}_c) + \underbrace{\frac{a_3}{\|\boldsymbol{p}_d - \boldsymbol{p}_a\|_2}}_{=\delta} \boldsymbol{u}(\boldsymbol{p}_d) \, .
\end{aligned} \tag{4.30}$$

The barycentric coordinates α, β, γ, δ are efficiently computed by applying Algorithm 7 in a slightly modified version suitable for tetrahedra instead of triangles.

The barycentric interpolation scheme as described above is a *consistent* interpolation scheme. This is easily illustrated with the help of the simple example sketched in Figure 4.14a. In the case that $\boldsymbol{u}(\boldsymbol{p}_l) = \bar{\boldsymbol{u}} \ \forall \, l = 1, \ldots, n$, the function values at the query

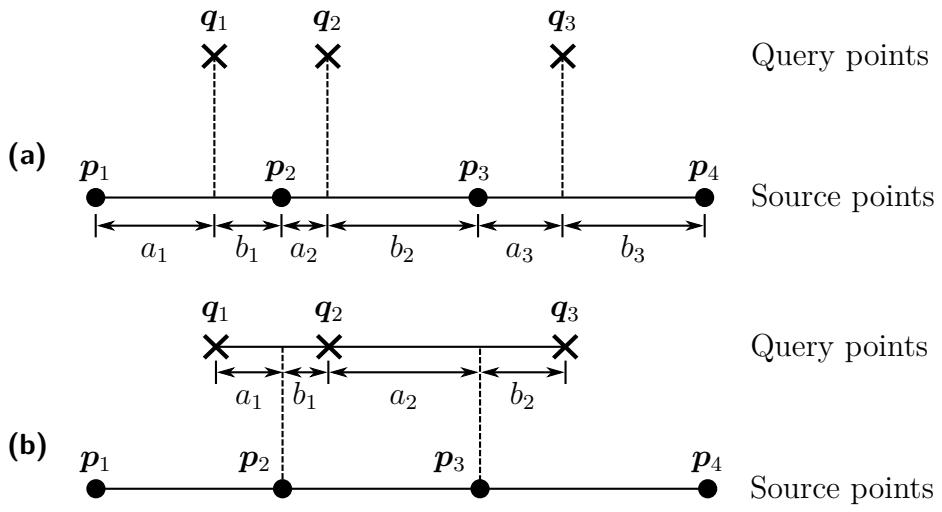


Figure 4.14: Barycentric interpolation in its (a) consistent and (b) conservative version.

points are computed as

$$\begin{aligned}
\boldsymbol{u}(\boldsymbol{q}_1) &= \alpha_1 \boldsymbol{u}(\boldsymbol{p}_1) + \beta_1 \boldsymbol{u}(\boldsymbol{p}_2) = \underbrace{(\alpha_1 + \beta_1)}_{=1} \boldsymbol{u} \ , \\
\boldsymbol{u}(\boldsymbol{q}_2) &= \alpha_2 \boldsymbol{u}(\boldsymbol{p}_2) + \beta_2 \boldsymbol{u}(\boldsymbol{p}_3) = \underbrace{(\alpha_2 + \beta_2)}_{=1} \boldsymbol{u} \ , \\
\boldsymbol{u}(\boldsymbol{q}_3) &= \alpha_3 \boldsymbol{u}(\boldsymbol{p}_3) + \beta_3 \boldsymbol{u}(\boldsymbol{p}_4) = \underbrace{(\alpha_3 + \beta_3)}_{=1} \boldsymbol{u} \ .
\end{aligned} \tag{4.31}$$

In matrix notation, these relations are recast as

$$\begin{pmatrix} \boldsymbol{u}(\boldsymbol{q}_1) \\ \boldsymbol{u}(\boldsymbol{q}_2) \\ \boldsymbol{u}(\boldsymbol{q}_3) \end{pmatrix} = \begin{pmatrix} \alpha_1 & \beta_1 & 0 & 0 \\ 0 & \alpha_2 & \beta_2 & 0 \\ 0 & 0 & \alpha_3 & \beta_3 \end{pmatrix} \begin{pmatrix} \boldsymbol{u}(\boldsymbol{p}_1) \\ \boldsymbol{u}(\boldsymbol{p}_2) \\ \boldsymbol{u}(\boldsymbol{p}_3) \\ \boldsymbol{u}(\boldsymbol{p}_4) \end{pmatrix} \ , \tag{4.32}$$

where the interpolation matrix obviously fulfills the requirement (4.24). For the conservative scheme, we deduce from Figure 4.14b that

$$\begin{aligned}
\boldsymbol{u}(\boldsymbol{q}_1) &= \boldsymbol{u}(\boldsymbol{p}_1) + \alpha_1 \boldsymbol{u}(\boldsymbol{p}_2) \ , \\
\boldsymbol{u}(\boldsymbol{q}_2) &= \beta_1 \boldsymbol{u}(\boldsymbol{p}_2) + \alpha_1 \boldsymbol{u}(\boldsymbol{p}_3) \ , \\
\boldsymbol{u}(\boldsymbol{q}_3) &= \beta_2 \boldsymbol{u}(\boldsymbol{p}_3) + \boldsymbol{u}(\boldsymbol{p}_4) \ ,
\end{aligned} \tag{4.33}$$

or, in matrix notation,

$$\begin{pmatrix} \boldsymbol{u}(\boldsymbol{q}_1) \\ \boldsymbol{u}(\boldsymbol{q}_2) \\ \boldsymbol{u}(\boldsymbol{q}_3) \end{pmatrix} = \begin{pmatrix} 1 & \alpha_1 & 0 & 0 \\ 0 & \beta_1 & \alpha_2 & 0 \\ 0 & 0 & \beta_2 & 1 \end{pmatrix} \begin{pmatrix} \boldsymbol{u}(\boldsymbol{p}_1) \\ \boldsymbol{u}(\boldsymbol{p}_2) \\ \boldsymbol{u}(\boldsymbol{p}_3) \\ \boldsymbol{u}(\boldsymbol{p}_4) \end{pmatrix} \ . \tag{4.34}$$

Obviously, it holds that

$$\boldsymbol{u}(\boldsymbol{p}_1)+\boldsymbol{u}(\boldsymbol{p}_2)+\boldsymbol{u}(\boldsymbol{p}_3)+\boldsymbol{u}(\boldsymbol{p}_4) = \boldsymbol{u}(\boldsymbol{p}_1)+\underbrace{(\alpha_1 + \beta_1)}_{=1} \boldsymbol{u}(\boldsymbol{p}_2)+\underbrace{(\alpha_2 + \beta_2)}_{=1} \boldsymbol{u}(\boldsymbol{p}_3)+\boldsymbol{u}(\boldsymbol{p}_4) \ , \tag{4.35}$$

which confirms the requirement (4.25) for conservative interpolation schemes. For the calculation of the conservative interpolation weights, one can again take advantage of Algorithm 6 to reuse the implementation already available for the consistent scheme.

4.5.3 Radial Basis Function Interpolation

Radial basis function interpolation schemes (see, e.g, [24]) are based on interpolating the function value $\boldsymbol{u}(\boldsymbol{q})$ at the query point $\boldsymbol{q}$ from the weighted sum

$$\boldsymbol{u}(\boldsymbol{q}) = \sum_{l=1}^{n} \boldsymbol{\lambda}_l \varphi\left(\|\boldsymbol{q} - \boldsymbol{p}_l\|_2\right) \ , \tag{4.36}$$

where φ represents a scalar-valued radial basis function (RBF). In matrix notation, this reads

$$\underbrace{\begin{pmatrix} \boldsymbol{u}(\boldsymbol{q}_1) \\ \vdots \\ \boldsymbol{u}(\boldsymbol{q}_m) \end{pmatrix}}_{=\mathbf{u}^\star} = \underbrace{\begin{pmatrix} \varphi(\|\boldsymbol{q}_1 - \boldsymbol{p}_1\|_2) & \cdots & \varphi(\|\boldsymbol{q}_1 - \boldsymbol{p}_n\|_2) \\ \vdots & \ddots & \vdots \\ \varphi(\|\boldsymbol{q}_m - \boldsymbol{p}_1\|_2) & \cdots & \varphi(\|\boldsymbol{q}_m - \boldsymbol{p}_n\|_2) \end{pmatrix}}_{=\boldsymbol{\Phi}_1} \underbrace{\begin{pmatrix} \boldsymbol{\lambda}_1 \\ \vdots \\ \boldsymbol{\lambda}_n \end{pmatrix}}_{=\boldsymbol{\lambda}} \ . \tag{4.37}$$

For the evaluation of the weighting vector $\boldsymbol{\lambda}$, we consider the special case $m = n$ and $\boldsymbol{q}_i = \boldsymbol{p}_i, i = 1, \ldots, n$, which results in

$$\underbrace{\begin{pmatrix} \boldsymbol{u}(\boldsymbol{p}_1) \\ \vdots \\ \boldsymbol{u}(\boldsymbol{p}_n) \end{pmatrix}}_{=\mathbf{u}} = \underbrace{\begin{pmatrix} \varphi(\|\boldsymbol{p}_1 - \boldsymbol{p}_1\|_2) & \cdots & \varphi(\|\boldsymbol{p}_1 - \boldsymbol{p}_n\|_2) \\ \vdots & \ddots & \vdots \\ \varphi(\|\boldsymbol{p}_n - \boldsymbol{p}_1\|_2) & \cdots & \varphi(\|\boldsymbol{p}_n - \boldsymbol{p}_n\|_2) \end{pmatrix}}_{=\boldsymbol{\Phi}_2} \begin{pmatrix} \boldsymbol{\lambda}_1 \\ \vdots \\ \boldsymbol{\lambda}_n \end{pmatrix} \tag{4.38}$$

and, thus,

$$\boldsymbol{\lambda} = \boldsymbol{\Phi}_2^{-1} \mathbf{u} \ . \tag{4.39}$$

It should be emphasized that the inverse in (4.39) should never be computed explicitly. Instead, it is more efficient to factorize $\boldsymbol{\Phi}_2$ and to solve (4.39) by forward and backward substitution. Inserting (4.39) into (4.37) leads us to

$$\mathbf{u}^\star = \boldsymbol{\Phi}_1 \boldsymbol{\Phi}_2^{-1} \mathbf{u} \tag{4.40}$$

and correlates the function values $\boldsymbol{u}(\boldsymbol{q}_1), \ldots, \boldsymbol{u}(\boldsymbol{q}_m)$ at the query points $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_m$ to the function values $\boldsymbol{u}(\boldsymbol{p}_1), \ldots, \boldsymbol{u}(\boldsymbol{p}_n)$ at the source points $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n$.

Possible choices for the RBF φ are listed in Table 4.3. The RBFs listed in the left half of Table 4.3 have global support – and they lead to fully populated matrices $\boldsymbol{\Phi}_1$ and $\boldsymbol{\Phi}_2$, which usually renders (4.40) prohibitively expensive to solve for large point sets. It is therefore often advisable to choose an RBF with compact support, which results in sparse matrices $\boldsymbol{\Phi}_1, \boldsymbol{\Phi}_2$ and significantly reduces the amount of required memory and computational time for the solution of (4.40). Several interesting compactly-supported RBFs were proposed in [173], for instance, which are listed on the right half of Table 4.3. Note that, in practice, the radius r in both types of RBFs is normally scaled by a parameter r_0, and the ratio r/r_0 instead of just the radius r is used in the evaluation of the RBFs. The parameter r_0 is problem-dependent; a small r_0 leads to a stronger influence of source

Table 4.3: Selection of global RBFs (left) and compactly-supported Wendland-type RBFs (right), where $r := \|\boldsymbol{y} - \boldsymbol{x}\|_2$ and $a_+ := \max\{a, 0\}$ [173, 29, p. 15].

Description	RBF $\varphi(r)$	Description	RBF $\varphi(r)$
Linear	r	Wendland (1)	$(1-r)_+^2$
Cubic	r^3	Wendland (2)	$(1-r)_+^4(4r+1)$
Gaussian	$\exp(-r^2)$	Wendland (3)	$(1-r)_+^6(35r^2+18r+3)$
Multiquadric	$\sqrt{1+r^2}$	Wendland (4)	$(1-r)_+^8(32r^3+25r^2+8r+1)$
Inverse quadratic	$(1+r^2)^{-1}$	Wendland (5)	$(1-r)_+^3$
Inverse multiquadric	$\sqrt{1+r^2}^{-1}$	Wendland (6)	$(1-r)_+^5(5r+1)$
Thin plate spline	$r^2\log(1+r)$	Wendland (7)	$(1-r)_+^7(16r^2+7r+1)$

points close to the query point $\boldsymbol{q}$, whereas a larger r_0 increases the region where the RBF is substantially different from zero.

For a conservative version of the consistent interpolation scheme outlined above, the source and query point set $\mathcal{P}$ and $\mathcal{Q}$ are reversed, resulting in the point sets $\mathcal{P}' := \mathcal{Q}$ and $\mathcal{Q}' := \mathcal{P}$. The matrices $\boldsymbol{\Phi}_1', \boldsymbol{\Phi}_2'$ are then constructed based on the consistent procedure, but instead of Equation (4.40), we solve

$$\mathbf{u}^\star = \left(\boldsymbol{\Phi}_1' \boldsymbol{\Phi}_2'^{-1}\right)^{\mathrm{T}} \mathbf{u} = \boldsymbol{\Phi}_2'^{-\mathrm{T}} \boldsymbol{\Phi}_1'^{\mathrm{T}} \mathbf{u} \,. \tag{4.41}$$

4.5.4 Inverse Distance Weighting

Inverse distance weighting is another interesting mesh-independent interpolation scheme, which dates back to Shepard [144] and is therefore also often referred to as *Shepard's method*. Given a set of source points $\mathcal{P} = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n\}$ and associated function values $\boldsymbol{u}(\boldsymbol{p}_1), \ldots, \boldsymbol{u}(\boldsymbol{p}_n)$, Shepard's method interpolates the function value $\boldsymbol{u}(\boldsymbol{q})$ at the query point $\boldsymbol{q}$ according to

$$\boldsymbol{u}(\boldsymbol{q}) = \begin{cases} \dfrac{\sum_{l=1}^n w_l(\boldsymbol{q})\boldsymbol{u}(\boldsymbol{p}_l)}{\sum_{l=1}^n w_l(\boldsymbol{q})} & , \quad \text{if } d(\boldsymbol{p}_l, \boldsymbol{q}) \neq 0 \text{ for all } l = 1, \ldots, n \\ \boldsymbol{u}(\boldsymbol{p}_l) & , \quad \text{if } d(\boldsymbol{p}_l, \boldsymbol{q}) = 0 \text{ for any } l = 1, \ldots, n \end{cases} . \tag{4.42}$$

Therein,

$$w_l(\boldsymbol{q}) = d(\boldsymbol{p}_l, \boldsymbol{q})^{-p} \tag{4.43}$$

is the interpolation weight associated to the source point $\boldsymbol{p}_l$ and $d(\boldsymbol{p}_l, \boldsymbol{q})$ is a suitable distance metric such as the Euclidean norm. The power parameter p is a user-defined parameter, usually chosen as $p = 2$. For larger data sets, it becomes expensive to include all source values $\boldsymbol{u}(\boldsymbol{p}_1), \ldots, \boldsymbol{u}(\boldsymbol{p}_n)$ to determine the interpolated value $\boldsymbol{u}(\boldsymbol{q})$. It is therefore advisable to include only source points within a cut-off radius r around the query point $\boldsymbol{q}$ to compute $\boldsymbol{u}(\boldsymbol{q})$ and to neglect all other source points. Since the weight (4.43) decays for increasing distances $d(\boldsymbol{p}_l, \boldsymbol{q})$, it can be expected that they have only a minor influence on $\boldsymbol{u}(\boldsymbol{q})$. The source points within the cut-off radius can efficiently be found by range search

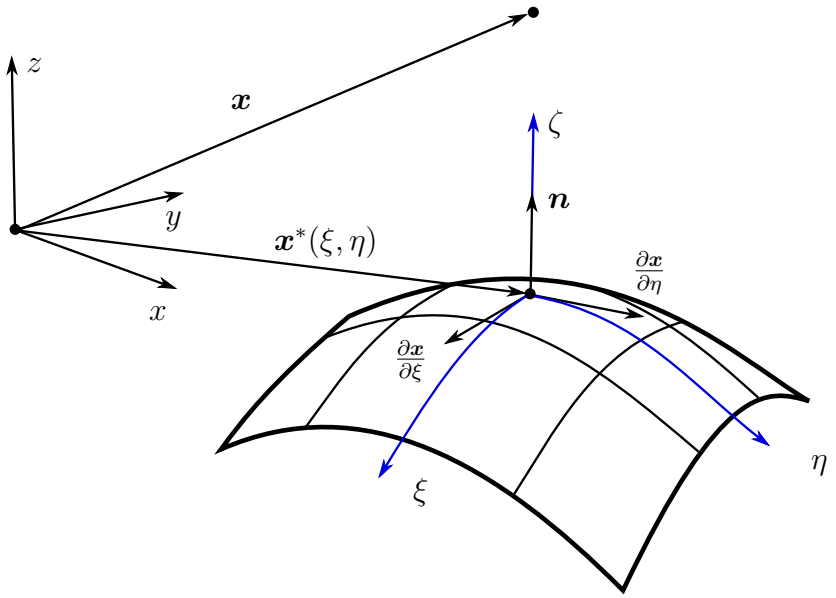


Figure 4.15: Nearest point projection for a surface in three-dimensional space [98, p. 38].

techniques, which rely on space partitioning methods comparable to those already used for the k-nearest neighbor search.

A conservative version of this consistent interpolation scheme is again easily constructed by means of Algorithm 6.

4.5.5 Interpolation on Finite Element Meshes

The previously discussed interpolation schemes all fall in the category of mesh-independent interpolation schemes. The term *mesh-independent* is due the fact that the topology information from the computational meshes of the source and target discretization is *not* taken into account for interpolation. In contrast, *mesh-dependent* interpolation schemes do use the mesh topology, usually with the aim to improve the quality of the interpolation. The reader is referred to Section 4.5.7, where the various mesh-independent interpolation schemes are compared to a mesh-dependent interpolation using several different benchmark discretizations. From a practical point of view, a notable drawback of a mesh-dependent interpolation scheme is the requirement to access the mesh information from a solver's database. Nonetheless, the effort of extracting the mesh connectivity in addition to the source point coordinates, which are also needed in mesh-independent interpolation schemes, is usually more than outweighed by the increased interpolation quality.

In an FE mesh, the elements are parameterized by $\boldsymbol{x}^*(\boldsymbol{\xi})$, where $\boldsymbol{x}^* \in \mathbb{R}^d$ represents a point in the element, expressed in global coordinates, and $\boldsymbol{\xi} \in \mathbb{R}^l$ stems from the element's l-dimensional parameter space. Figure 4.15 sketches a typical situation in three-dimensional space, where a point is projected on a surface.

It is possible to formulate the problem of projecting a point $\boldsymbol{x} \in \mathbb{R}^d$ to an element in d-dimensional space as an optimization problem [98, pp. 38–40]:

$$\underset{\boldsymbol{\xi} \in \Gamma}{\operatorname{argmin}} \left\| \boldsymbol{x} - \boldsymbol{x}^*(\boldsymbol{\xi}) \right\|_2 , \tag{4.44}$$

or, equivalently,

$$\underset{\boldsymbol{\xi}\in\Gamma}{\operatorname{argmin}} \left((\boldsymbol{x}-\boldsymbol{x}^*)\cdot(\boldsymbol{x}-\boldsymbol{x}^*)\right) \ . \tag{4.45}$$

Existence and uniqueness of a solution to the minimization problem (4.45) is guaranteed if the function

$$\boldsymbol{F} = \frac{1}{2}(\boldsymbol{x}-\boldsymbol{x}^*)\cdot(\boldsymbol{x}-\boldsymbol{x}^*) \tag{4.46}$$

is convex for $\boldsymbol{\xi} \in \Omega$, where Ω represents the domain occupied by the element. Hence, a Newton-Raphson procedure can be applied to solve (4.45) and can be expected to converge for any initial solution $\boldsymbol{\xi}^0 \in \Omega$.

For twice continuously differentiable elements, the Newton-Raphson procedure reads

$$\begin{aligned} \Delta\boldsymbol{\xi}^k &= -\left(\boldsymbol{F}''^k\right)^{-1}\boldsymbol{F}'^k \\ \boldsymbol{\xi}^{k+1} &= \boldsymbol{\xi}^k + \Delta\boldsymbol{\xi}^k \ . \end{aligned} \tag{4.47}$$

Given the local coordinates $\boldsymbol{\xi}$ corresponding to a projected point $\boldsymbol{x}^*$, the shape functions of the element can be evaluated at that point. Then, the value of a field quantity $\boldsymbol{u}(\boldsymbol{x}^*)$ is obtained from a linear combination of the products of the evaluated shape functions at $\boldsymbol{x}^*$ and the values of the field quantity at the element's degrees of freedom. In the case of isoparametric elements, which are primarily considered in this work, the element's degrees of freedom are situated at the nodes of the element. For FE meshes, an interpolation based on the above projection procedure can be considered to be optimal in terms of accuracy. For other discretization schemes such as the FVM or the BEM, the procedure can be slightly modified as outlined in Section 4.5.6.

Insofar as entire meshes are concerned, the question arises how to determine the elements that need to be considered for projection. Taking all elements into account for the iterative and, thus, comparably expensive Newton-Raphson procedure will certainly result in an unacceptably high numerical effort. It is therefore advisable to select only a small subset of elements and to consider only these elements as possible candidates for the closest element to a given query point $\boldsymbol{q}$. To this end, we propose an efficient spatial search strategy based on an axis-aligned bounding box (AABB) tree, similar to the method proposed in [99]. The basic idea behind AABB trees is a subdivision of the search space into partitions. Then, it is sufficient to visit only a small subset of elements, comparable to the k-d tree introduced in Section 4.5.1. In the first step, each element of the FE mesh is tightly enclosed by a bounding box, thus generating a set of bounding boxes $\mathcal{B} = \{B_1, \ldots, B_n\}$. A bounding box in d-dimensional space is fully characterized by a pair of points $(\boldsymbol{x}_{\min}, \boldsymbol{x}_{\max})$, reflecting the spatial extension of the enclosed object. The bounding boxes are then organized into a tree structure – the AABB tree – by subdividing the search space successively into half spaces until only a single element is left in a half space; these are then the leaves of the tree. Algorithm 8 illustrates a possible implementation based on a recursive tree setup. Once the AABB tree is created, it can be queried for a point $\boldsymbol{q}$ to return a set of nearest bounding boxes $\mathcal{R}$, enclosing the elements that represent the possible candidates for the nearest element to $\boldsymbol{q}$. The procedure is outlined in Algorithm 9. Starting at the root node, we first determine the axis a along which the bounding boxes were sorted in the current tree level ℓ. Then, the distance $d_{\max}(B_l, \boldsymbol{q}) := \max\{d(\boldsymbol{x}_{\min,l}, \boldsymbol{q}), d(\boldsymbol{x}_{\max,l}, \boldsymbol{q})\}$ is computed. It reflects the maximum distance a point inside B_l might have to the query point $\boldsymbol{q}$. If $d_{\max}(B_l, \boldsymbol{q})$ undershoots $d^\star$ (which is initially set to $d^\star := \infty$), the bounding box B_l is

```
1: function AabbTree(set of bounding boxes B = {B_1, ..., B_n}, tree level ℓ)
2:     Determine axis a := mod(ℓ − 1, d) + 1 for sorting
3:     Sort elements in B along axis a by considering the bounding box centers c_1, ..., c_n
4:     Determine the median c̃ of {c_1, ..., c_n}
5:     Split B at the bounding box B̃ corresponding to the median c̃ to form a left subset
       B_left and a right subset B_right
6:     Increase the tree level ℓ := ℓ + 1
7:     Create a tree node N
8:     if B_left ≠ ∅ then
9:         Create left child node N.left := AabbTree(B_left, ℓ)
10:    end if
11:    if B_right ≠ ∅ then
12:        Create right child node N.right := AabbTree(B_right, ℓ)
13:    end if
14:    return N
15: end function
```

Algorithm 8: Function AabbTree for the recursive setup of an AABB tree in d-dimensional space.

considered a candidate for one of the nearest bounding boxes to $\boldsymbol{q}$. B_l is added to the set $\mathcal{R}$ such that all elements in $\mathcal{R}$ are sorted according to their minimum distance $d_{\min}(B_i, \boldsymbol{q}) := \min\{d(\boldsymbol{x}_{\min,i}, \boldsymbol{q}), d(\boldsymbol{x}_{\max,i}, \boldsymbol{q})\}$ to the query point $\boldsymbol{q}$. Subsequently, all elements B_i for which $d_{\min}(B_i, \boldsymbol{q}) > d^\star$ are cleared from $\mathcal{R}$, and $d^\star$ is set to the largest distance $d_{\max}(B, \boldsymbol{q})$ of all elements $B \in \mathcal{B}$. Following that, the directed distance $d' := c_{l,a} - q_a$ is computed to decide which branch to descend next. The tree level ℓ is incremented and the closer branch is traversed. If the hypersphere of radius $d^\star$ intersects the splitting plane, the farther branch needs to be visited as well.

The procedure is graphically visualized in Figure 4.16 for a simple example. Starting at the root node, see Figure 4.16a, $d^\star := d_{\max}(B_1, \boldsymbol{q})$ is computed. Continuing with the left half-space as depicted in Figure 4.16b, the distance $d^\star$ is updated and set to $d^\star := d_{\max}(B_2, \boldsymbol{q})$. The sphere of radius $d^\star$ does not intersect B_1, and, hence, B_1 is discarded from the set $\mathcal{R}$ and B_2 is added instead. Descending further down the tree, as indicated in Figure 4.16c, B_4 is not included in $\mathcal{R}$ as it is not intersected by the current best-estimate sphere. On the contrary, B_5 must be included in $\mathcal{R}$. Ascending back to the root node, see Figure 4.16d, the right half-plane can be safely discarded from the search, as the splitting plane in the first tree level is not intersected by the current best-estimate sphere. Finally, $\mathcal{R} = \{B_2, B_5\}$, i.e. either B_2 or B_5 must enclose the nearest element to the query point $\boldsymbol{q}$. The projection procedure only has to be performed for the elements enclosed by these bounding boxes, which consequently reduces the numerical effort considerably as compared to taking all elements into consideration. In fact, the spatial search based on AABB trees is very generic and can be applied whenever neighborhood relations between potentially complex geometric objects need to be established.

The projection procedure outlined in Equation (4.47) is perfectly suited for all kinds of low- and also high-order continuum elements. However, a different approach must be adopted for structural elements such as beam or shell elements. For such elements, the coupling surface or volume does not coincide with the space occupied by the element,

1: **procedure** NearestBoundingBoxSearch(tree node N, query point $\boldsymbol{q}$, tree level ℓ, set of nearest bounding boxes $\mathcal{R}$, distance $d^\star$)
2: Determine axis $a := \mathrm{mod}(\ell - 1, k) + 1$
3: Compute distance $d_{\max}(B_l, \boldsymbol{q}) := \max\{d(\boldsymbol{x}_{\min,l}, \boldsymbol{q}), d(\boldsymbol{x}_{\max,l}, \boldsymbol{q})\}$, where B_l is the bounding box associated to tree node N
4: **if** $d_{\max}(B_l, \boldsymbol{q}) < d^\star$ **then**
5: $\mathcal{R} := \mathcal{R} \cup \{B_l\}$
6: Sort $\mathcal{R}$ such that $\forall\, B_i, B_j \in \mathcal{R}$ it holds that $d_{\min}(B_i, \boldsymbol{q}) := \min\{d(\boldsymbol{x}_{\min,i}, \boldsymbol{q}), d(\boldsymbol{x}_{\max,i}, \boldsymbol{q})\} \leq d_{\min}(B_j, \boldsymbol{q})$ if $i < j$
7: Delete all elements $B_i \in \mathcal{R}$ for which $d_{\min}(B_i, \boldsymbol{q}) > d^\star$
8: Set $d^\star := \max_{B_i \in \mathcal{R}} d_{\max}(B_i, \boldsymbol{q})$
9: **end if**
10: Compute distance in a-direction: $d' := c_{l,a} - q_a$
11: **if** $d' < 0$ **then**
12: $C_{\text{close}} := N$.left, $C_{\text{far}} := N$.right
13: **else**
14: $C_{\text{close}} := N$.right, $C_{\text{far}} := N$.left
15: **end if**
16: Increase the tree level $\ell := \ell + 1$
17: **if** $c_{\text{close}} \neq$ empty **then** ▷ Traverse closer branch
18: NearestBoundingBoxSearch(C_{close}, $\boldsymbol{q}$, ℓ, $\mathcal{R}$, $d_{\max}$)
19: **end if**
20: **if** $c_{\text{far}} \neq$ empty $\wedge\, |d'| < d^\star$ **then** ▷ Traverse farther branch
21: NearestBoundingBoxSearch(C_{far}, $\boldsymbol{q}$, ℓ, $\mathcal{R}$, $d_{\max}$)
22: **end if**
23: **end procedure**

Algorithm 9: Procedure NearestBoundingBoxSearch for the recursive traversal of an AABB tree.

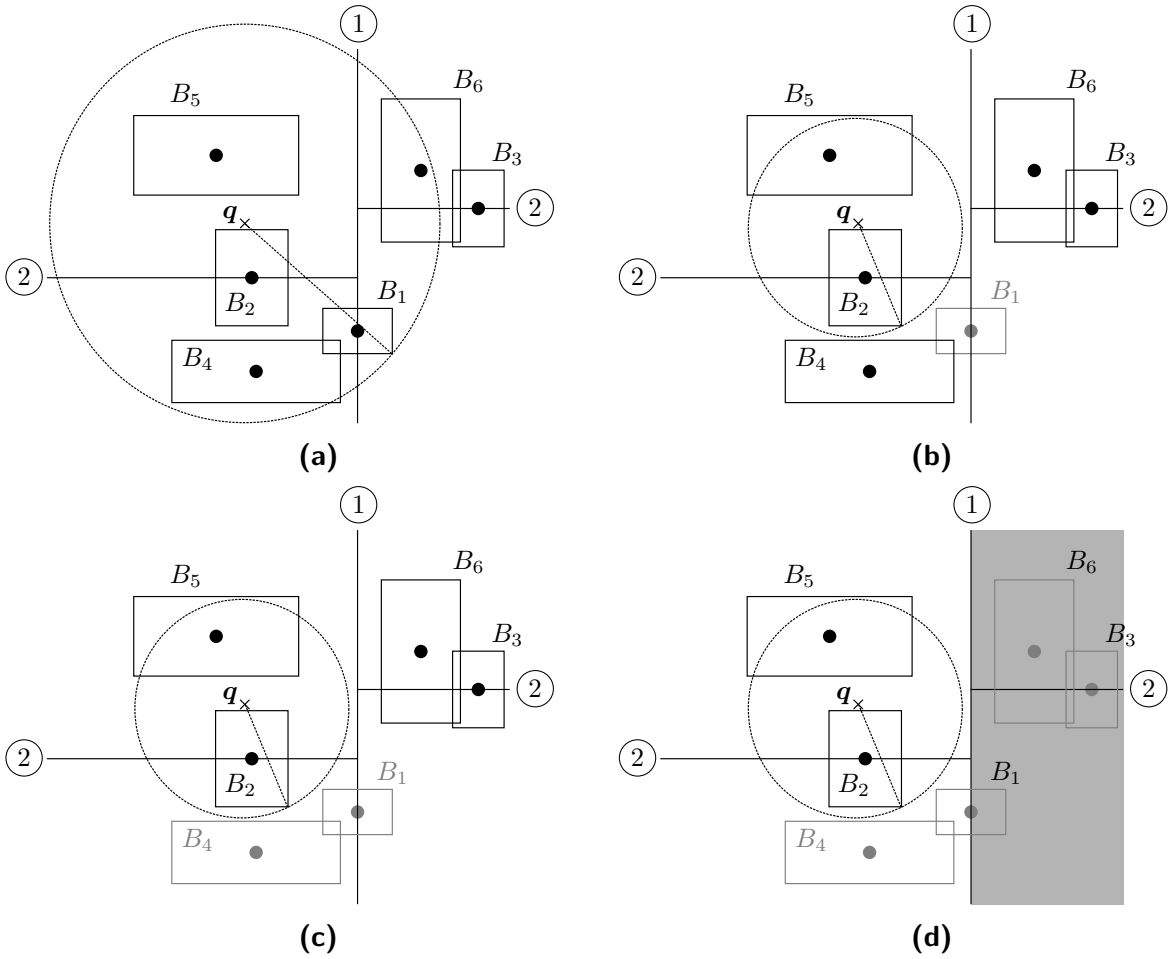


Figure 4.16: AABB tree traversal.

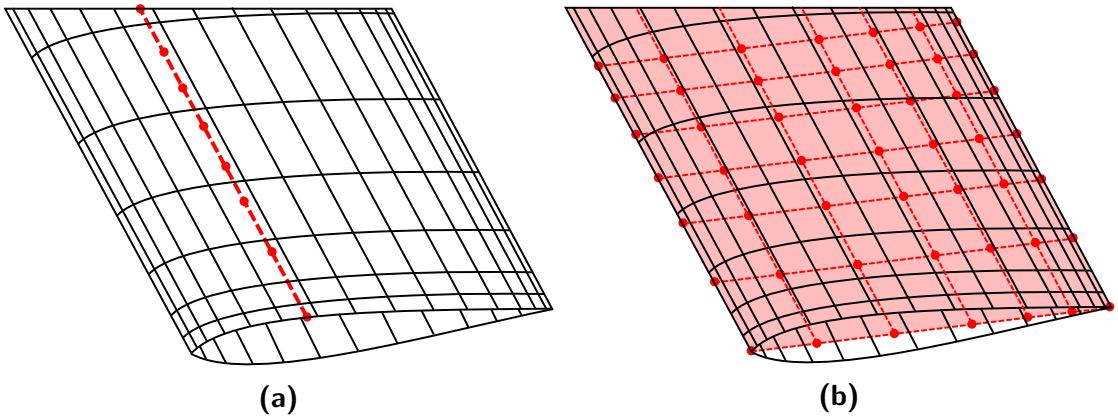


Figure 4.17: Slender structure discretized by (a) beam and (b) shell elements.

and the interpolation scheme must be tailored to still ensure an accurate data transfer across the coupling interface. For the sake of clarity, the discussion is limited to structural Timoshenko beams and Reissner-Mindlin shells applied in an FSI analysis. The general ideas, however, can, after some modifications, also be applied to other beam and shell element types such as Euler-Bernoulli beams or Kirchhoff shells, for instance.

In the first step, let us consider the interpolation of the displacement $\boldsymbol{d}$ from the structural to the fluid mesh. The situation is sketched in Figure 4.17, where a slender structure is discretized by beam elements (Figure 4.17a) or shell elements (Figure 4.17b). Regardless of whether the FVM, the FEM, or the BEM is used for the numerical treatment of the flow problem, the displacement needs to be interpolated from the nodes of the structural FE mesh to the vertices of the fluid discretization. Since structural elements do not only have translational but also rotational degrees of freedom, the interpolation procedure proposed for continuum elements must be modified to also include rotational deformation. To this end, we propose the following approach. Given a point $\boldsymbol{q}$ on the fluid mesh, where the displacement is required, that point is first projected to the nearest beam or shell element, resulting in the projected point $\boldsymbol{q}^*$. The distance vector between these two points is given by $\boldsymbol{r} = \boldsymbol{q} - \boldsymbol{q}^*$. At the projected point $\boldsymbol{q}^* = \boldsymbol{q}^*(\boldsymbol{\xi})$, the displacement is evaluated from a linear combination of the displacement values stored at the n nodes of the element:

$$\boldsymbol{d}(\boldsymbol{q}^*(\boldsymbol{\xi})) = \sum_{i=1}^{n} N_i(\boldsymbol{\xi}) \boldsymbol{d}_i \; . \tag{4.48}$$

Following that, the rotation matrix $\boldsymbol{R}(\boldsymbol{q}^*)$ signifying the orientation at the projected point $\boldsymbol{q}^*$ is analogously computed as

$$\boldsymbol{R}(\boldsymbol{q}^*(\boldsymbol{\xi})) = \sum_{i=1}^{n} N_i(\boldsymbol{\xi}) \boldsymbol{R}_i \; . \tag{4.49}$$

Based on (4.48) and (4.49), the displacement $\boldsymbol{d}(\boldsymbol{q})$ becomes

$$\boldsymbol{d}(\boldsymbol{q}) = \boldsymbol{d}(\boldsymbol{q}^*) + \boldsymbol{R}(\boldsymbol{q}^*) \boldsymbol{r} \; . \tag{4.50}$$

The rotation matrix $\boldsymbol{R}(\boldsymbol{q}^*)$, however, is not directly accessible and must be constructed incrementally from the nodal rotation increments in each time increment. Moreover, to

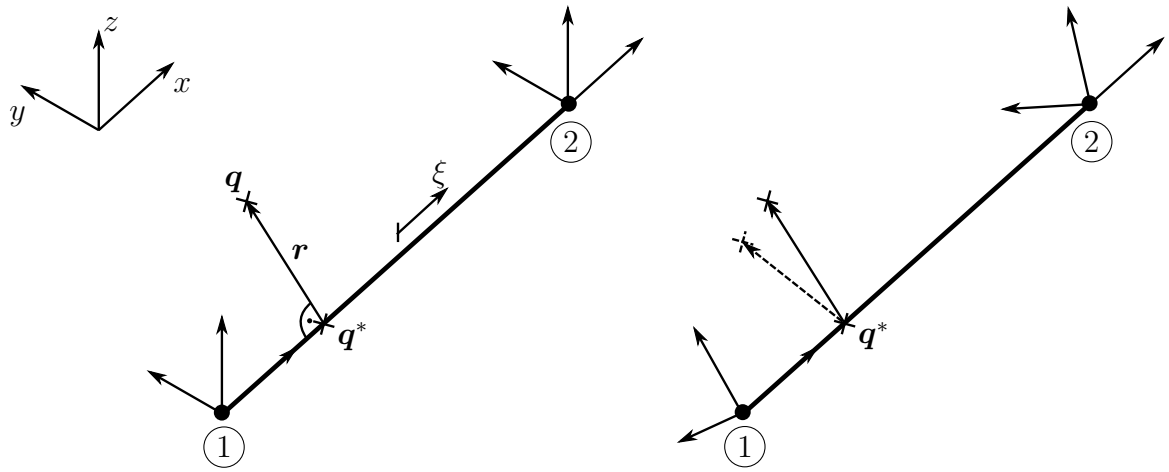


Figure 4.18: Displacement interpolation for a twisted Timoshenko beam element.

avoid numerical drift, we resort to a quaternion $\boldsymbol{\rho}_i$ instead of the rotation matrix $\boldsymbol{R}_i$ to represent the orientation at the ith element node. The rotation matrix $\boldsymbol{R}_i$ can be recaptured from the quaternion $\boldsymbol{\rho}_i = (s, \boldsymbol{v})$ by virtue of [6, p. 22]

$$\boldsymbol{R}_i = \begin{pmatrix} 1 - 2v_y^2 - v_z^2 & 2v_x v_y - 2sv_z & 2v_x v_z + 2sv_y \\ 2v_x v_y + 2sv_z & 1 - 2v_x^2 - 2v_z^2 & 2v_y v_z - 2sv_x \\ 2v_x v_z - 2sv_y & 2v_y v_z + 2sv_x & 1 - 2v_x^2 - 2v_y^2 \end{pmatrix} . \tag{4.51}$$

Given a rotation increment $\Delta\boldsymbol{\varphi}_{i,j} = \Delta\varphi_{i,j,1}\boldsymbol{e}_1 + \Delta\varphi_{i,j,2}\boldsymbol{e}_2 + \Delta\varphi_{i,j,3}\boldsymbol{e}_3$ in the jth time increment, the corresponding unit quaternion reads [147, p. 112 sq.]

$$\Delta\boldsymbol{\rho}_{i,j} = \left(\cos\frac{\|\Delta\boldsymbol{\varphi}_{i,j}\|_2}{2}, \frac{\Delta\boldsymbol{\varphi}_{i,j}}{\|\Delta\boldsymbol{\varphi}_{i,j}\|_2} \sin\frac{\|\Delta\boldsymbol{\varphi}_{i,j}\|_2}{2} \right) . \tag{4.52}$$

An updated, improved quaternion $\boldsymbol{\rho}_{i,j+1}$ is obtained from

$$\boldsymbol{\rho}_{i,j+1} = \Delta\boldsymbol{\rho}_{i,j} * \boldsymbol{\rho}_{i,j} , \tag{4.53}$$

where $*$ again implies the quaternion product. Following the outlined procedure, the displacement $\boldsymbol{d}(\boldsymbol{q})$ can then be determined according to (4.50).

In the second step, we consider the interpolation of the fluid traction to the structural FE mesh and the calculation of the resulting nodal forces required in the assembly of the global load vector. For this, let us assume that some discretization approximating the geometric surface of the structure be given. This discretization may also coincide with the discrete surface of the fluid domain at the FSI interface. Then, each element is furnished with a set of integration points suitable for the element's particular topology. If required, the fluid traction can be easily interpolated to these integration points by applying any of the aforementioned mesh-independent or mesh-based interpolation schemes. Considering an element A_l of the surface discretization and an integration point $\tilde{\boldsymbol{q}}_j$ with an associated local coordinate $\boldsymbol{\zeta}_j$ and integration weight w_j on A_l, we denote the (interpolated) traction at that point by $\boldsymbol{t}_j$. By projecting the integration point $\tilde{\boldsymbol{q}}_j$ to the closest structural element, we obtain the projected point $\tilde{\boldsymbol{q}}_j^*$, which corresponds to $\boldsymbol{\xi}_j$ in the parameter space of the structural element. The distance vector is again denoted by $\boldsymbol{r}_j = \tilde{\boldsymbol{q}}_j - \tilde{\boldsymbol{q}}_j^*$. Considering a

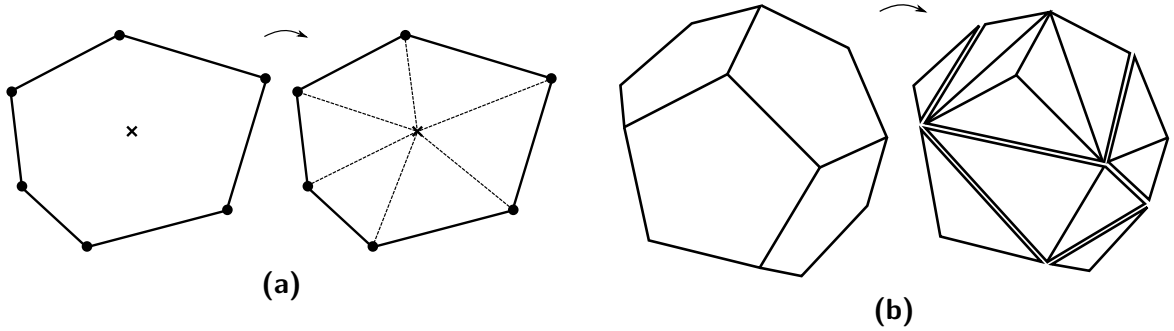


Figure 4.19: Decomposition of a (a) polygon or (b) polyhedron into triangles or tetrahedra, respectively.

particular structural element, the contribution of the traction $\boldsymbol{t}_j$ at the integration point $\tilde{\boldsymbol{q}}_j$ to the nodal force at the ith node of the structural element amounts to

$$\boldsymbol{f}_{i,j} = w_j N_i(\boldsymbol{\xi}_j)\boldsymbol{t}_j \det \boldsymbol{J}_l^{\mathrm{s}}(\boldsymbol{\zeta}_j) \tag{4.54}$$

and the nodal moment at the ith node is computed as

$$\boldsymbol{m}_{i,j} = w_j N_i(\boldsymbol{\xi}_j)(\boldsymbol{r}_j \times \boldsymbol{t}_j) \det \boldsymbol{J}_l^{\mathrm{s}}(\boldsymbol{\zeta}_j) \ , \tag{4.55}$$

where $\det \boldsymbol{J}_l^{\mathrm{s}}(\boldsymbol{\zeta}_j)$ signifies the determinant of the Jacobian matrix of the surface element A_l, evaluated at the local point $\boldsymbol{\zeta}_j$ of its parameter space. In order to increase the accuracy of the integration, it is also possible to apply a composed integration by splitting the surface elements A_l into smaller integration domains. Based on this, discontinuous integrands can be integrated accurately as well.

4.5.6 Interpolation on Polygonal and Polyhedral Meshes

The interpolation procedure for FE meshes outlined in the previous section is, after some modifications, also applicable to other discretization schemes where an interpolation of the values stored at the elements' degrees of freedom by means of shape functions is not available. Among others, the FVM and the BEM presented in Chapter 2 are examples for such schemes. Here, the computational mesh consists of polygonal or polyhedral cells. In order to determine the polygon closest to a given query point $\boldsymbol{q}$ in a surface-coupled problem or, in a volume-coupled problem, find the polyhedron $\boldsymbol{q}$ is located in, the polygons (or polyhedra) are first decomposed into triangles (or tetrahedra) as depicted in Figure 4.19. By applying the presented projection procedure for FE meshes, we then first determine the closest triangle or enclosing tetrahedron and, subsequently, the parent polygon or polyhedron by a table lookup. Once the polygon or polyhedron corresponding to a query point $\boldsymbol{q}$ has been determined, we identify the neighboring polygons or polyhedra (that is, polygons (polyhedra) sharing at least one vertex with the considered polygon (polyhedron)). By intention, we also consider the nearest polygon or polyhedron to be a neighbor of itself. In the FVM and the BEM, the result quantities (such as the traction $\boldsymbol{t}$, for instance) can be requested at the cell or panel centers. Denoting the number of neighbors by m and

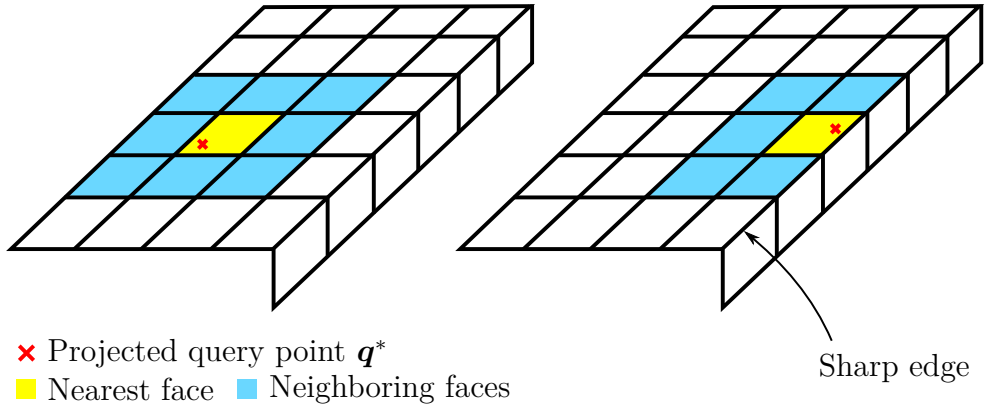


Figure 4.20: Surface interpolation on discretizations involving sharp edges.

indicating the center of the ith neighbor by $\boldsymbol{c}_i$, a sound interpolation scheme is given by

$$\boldsymbol{u}(\boldsymbol{q}) = \begin{cases} \dfrac{\sum_{i=1}^{m} w_i(\boldsymbol{q})\boldsymbol{u}(\boldsymbol{c}_i)}{\sum_{i=1}^{m} w_i(\boldsymbol{q})} , & \text{if } d(\boldsymbol{c}_i, \boldsymbol{q}) \neq 0 \text{ for all } i = 1, \ldots, m \\ \boldsymbol{u}(\boldsymbol{c}_i) & \text{if } d(\boldsymbol{c}_i, \boldsymbol{q}) = 0 \text{ for any } i = 1, \ldots, m \end{cases} . \tag{4.56}$$

similar to the inverse distance weighting scheme. Again, $d(\boldsymbol{c}_i, \boldsymbol{q})$ is a suitable distance metric such as the Euclidean norm, and the interpolation weight is $w_i(\boldsymbol{q}) = d(\boldsymbol{c}_i, \boldsymbol{q})^{-p}$ including the power parameter p.

Particular attention has to be paid to surface interpolation on discretizations involving sharp edges. In almost every case, an interpolation across a sharp edge produces inaccurate or nonphysical results – and should therefore best be avoided. Neighboring polygons with an outer normal $\boldsymbol{n}_i$ enclosing an angle θ_i greater than a user-defined threshold angle θ' with the outer normal $\boldsymbol{n}^*$ of the nearest polygon the projected query point $\boldsymbol{q}^*$ has been associated to are therefore excluded from the interpolation, see Figure 4.20. In most situations, the interpolation error can be reduced significantly as compared to mesh-independent interpolation techniques if the proposed mesh-based interpolation scheme is employed. A typical situation is sketched in Figure 4.21, where a floating cube is considered. Figure 4.21a depicts the interpolation result at the red cross if a nearest neighbor scheme is applied. Evidently, the resulting traction is directed in negative x-direction although one would expect a traction solely directed in y-direction. As shown in Figure 4.21b, a similar problem arises if a barycentric interpolation scheme is employed; the resulting traction still contains a component in negative x-direction. An accurate result resembling the expected value is only obtained if the topology of the mesh is taken into account, see Figure 4.21c.

4.5.7 Comparison of Interpolation Schemes

In order to compare the various interpolation schemes in terms of accuracy and computational effort, several benchmark problems are considered. The examples are deliberately based on functions evaluated on discrete geometry representations, which are better suited to give an account of the situation in a numerical analysis than the interpolation in continuous space. In all examples, the source discretization is an FE mesh consisting of isoparametric elements. A generic scalar quantity u is prescribed at the elements' nodes and shall be interpolated to a set of query points randomly distributed on the original

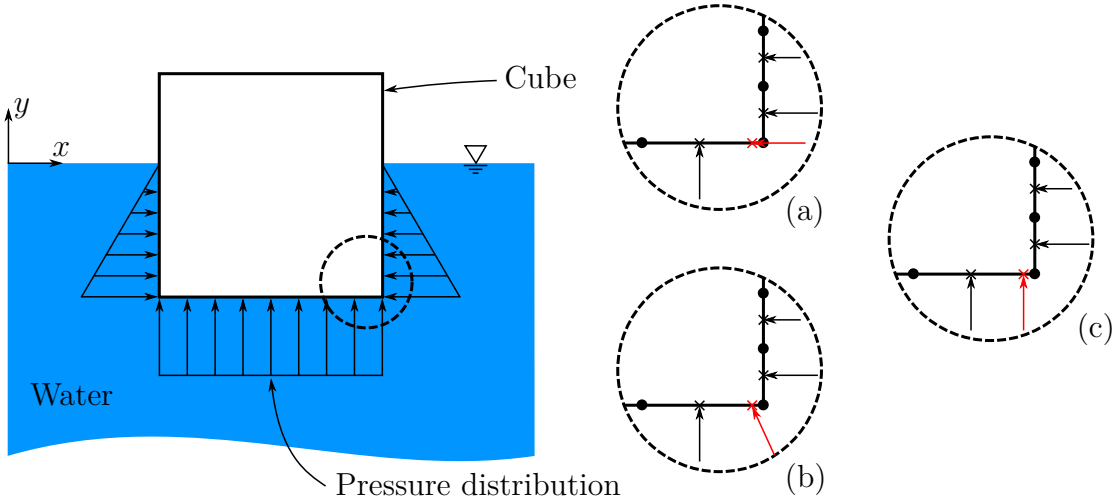


Figure 4.21: Sharp edge interpolation error.

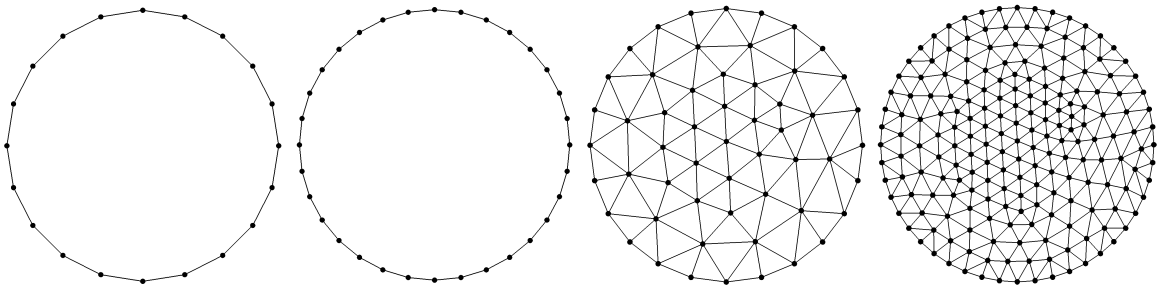


Figure 4.22: Discretizations of a circular line (a, b) and a circle (c, d).

geometry. Since all source discretizations are FE meshes, a reference value u_{ref} to judge about the quality of an interpolation scheme can be computed by applying the projection procedure outlined in Section 4.5.5. Apparently, this approximation is the best possible one. For the assessment of the interpolation error, we determine the relative global error measure

$$e = \frac{\sqrt{\sum_{k=1}^{m} \left(u(\boldsymbol{q}_k^*) - u_{\text{ref}}(\boldsymbol{q}_k^*)\right)^2}}{\sqrt{\sum_{i=1}^{m} u_{\text{ref}}^2(\boldsymbol{q}_k^*)}} \,. \tag{4.57}$$

Increasing the source mesh density successively while holding the number of query points m fixed enables us to analyze the error e as the number of source points n available for interpolation increases. Certainly, the quality of each of the proposed interpolation schemes can be expected to improve with an increasing number of source points n.

In the first example, we consider the interpolation on the boundary of a unit circle discretized by an FE mesh consisting of linear line elements, as depicted in Figure 4.22a and 4.22b. The mesh density and, hence, the number of elements are increased incrementally, whereas the number of $m = 10^3$ query points randomly distributed on the original

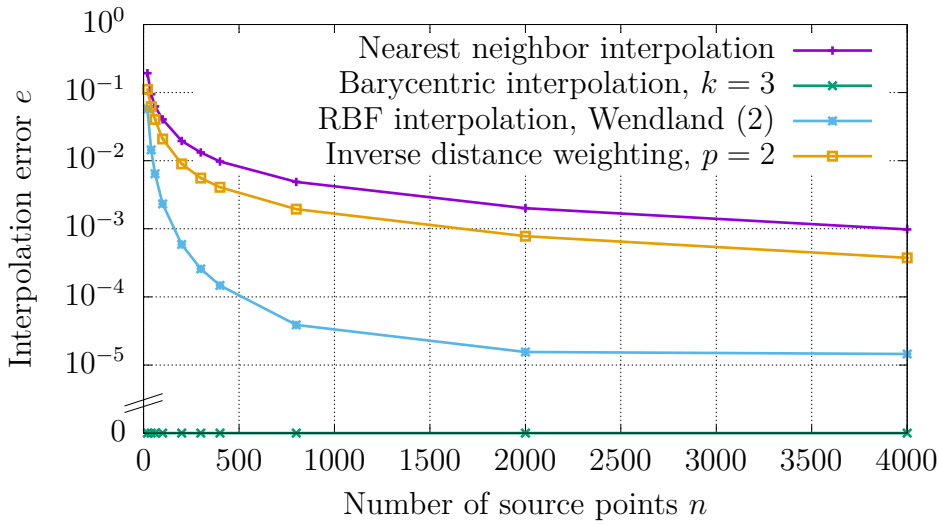


Figure 4.23: Interpolation errors of mesh-independent interpolation schemes on a discretized circular line.

continuous boundary of the circle remains unchanged. The function

$$u(x,y) = \bar{u}(x,y) = 3(1-x)^2 \exp\left(-x^2 - (y+1)^2\right) - 10\left(\frac{x}{5} - x^3 - y^5\right)\exp\left(-x^2 - y^2\right) - \frac{1}{3}\exp\left(-(x+1)^2 - y^2\right) \tag{4.58}$$

is evaluated at the nodes of the FE mesh; these nodal values are then used to produce the discrete reference solution by means of shape function interpolation as outlined in Section 4.5.5. The interpolation error for this example is depicted in Figure 4.23. The barycentric interpolation scheme resembles the reference solution almost exactly as, apparently, always two points forming a line coinciding with a line element from the boundary discretization have been selected for interpolation. The RBF interpolation scheme is also able to generate very accurate results; the interpolation error soon drops well below 0.1 % as the number of source points increases. Noticeably inferior results are obtained by inverse distance weighting and nearest neighbor interpolation.

In the next example, we investigate the performance of the interpolation schemes for the "volume" interpolation on a discretized circle – as depicted in Figure 4.22c and 4.22d – for two different exemplary mesh densities. As before, the function (4.58) is evaluated at the nodes of the FE mesh to generate the discrete function values at the source points for interpolation. Figure 4.24 delineates the interpolation error. As expectedly, the best results are again obtained by the barycentric interpolation scheme, although only a very small number of $k = 4$ nearest neighbors was used for the generation of the local Delaunay tessellation. The error produced by the other interpolation schemes is significantly higher. Similar to the previous example, the barycentric interpolation scheme is followed by the RBF interpolation and then by inverse distance weighting and nearest neighbor interpolation. Although the mesh density is already comparably high at the maximum refinement level, inverse distance weighting and the nearest neighbor interpolation still produce errors in the single-digit percentage area.

For surface interpolation in three-dimensional space, we consider the interpolation on

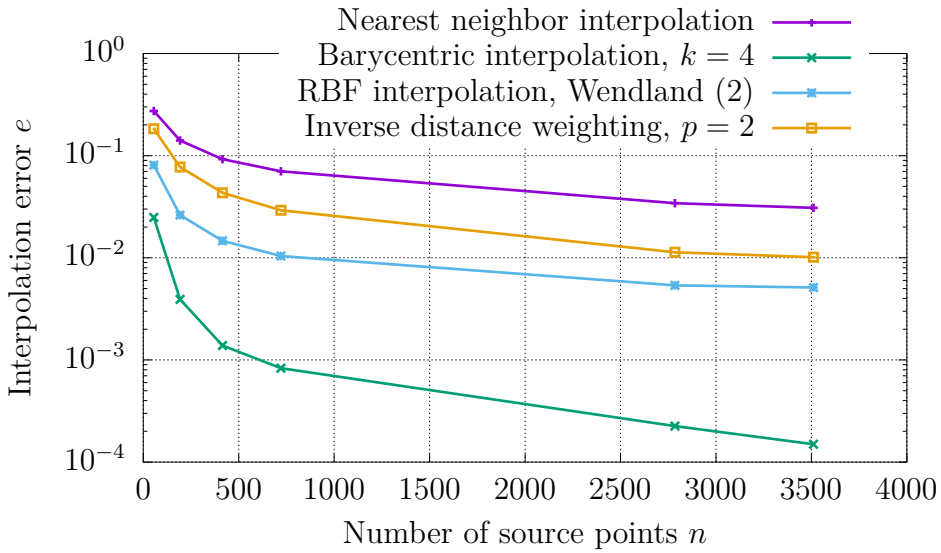


Figure 4.24: Interpolation errors of mesh-independent interpolation schemes on a discretized circle.

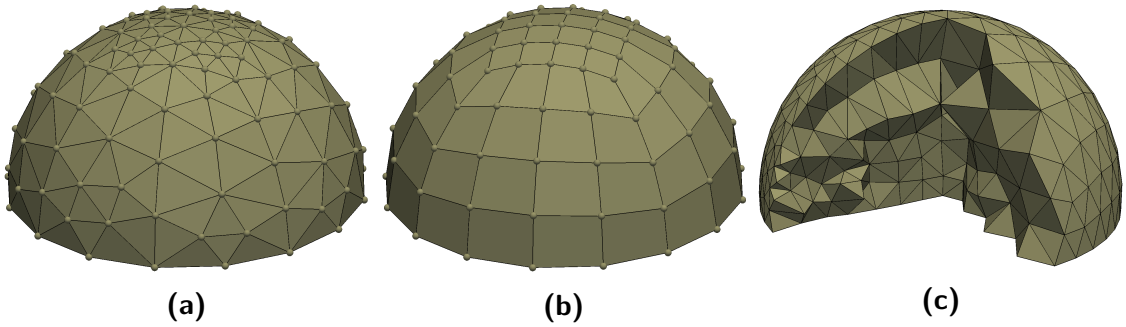


Figure 4.25: Discretizations of a spherical surface using linear (a) triangles and (b) quadrilaterals and (c) discretization of a half sphere using linear tetrahedra.

a discretized spherical surface as illustrated in Figure 4.25a and 4.25b. In the first case, linear triangular elements are used, while linear quadrilateral elements are employed in a second study. In order to generate the function values at the nodes of the FE mesh, the function (4.58) is augmented by a term to introduce a dependency on z, such that

$$u(x, y, z) = \bar{u}(x, y) - z^2 \, . \tag{4.59}$$

Figure 4.26 depicts the interpolation error for this example. Evidently, the errors associated to inverse distance weighting and nearest neighbor interpolation obey a similar tendency, irrespective of the element type the FE mesh consists of. On the contrary, the error in the RBF interpolation scheme is less if quadrilaterals are used, which is due to the fact that the average number of source points in the compact support of the RBF is different. As was to be expected too, the error in the barycentric interpolation scheme is slightly higher if a quadrilateral mesh is considered, which can be explained by the fact that the local Delaunay tessellation does not resemble the original discretization as accurately as in the case of the triangular mesh. Yet, the error for the barycentric interpolation scheme is again very small in absolute terms – and also in comparison to the other interpolation

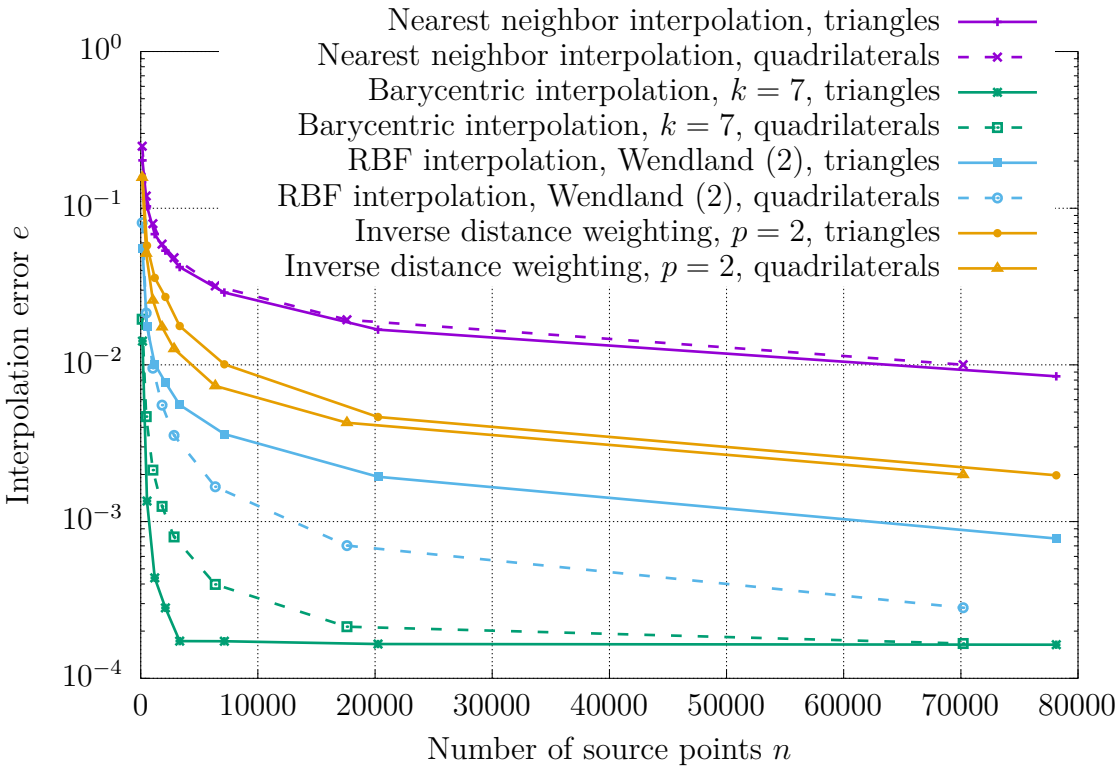


Figure 4.26: Interpolation errors of mesh-independent interpolation schemes on a discretized spherical surface.

schemes. RBF interpolation still leads to acceptable results, whereas the inverse distance weighting and the nearest neighbor interpolation do not prove to be competitive.

In the last example, we compare the interpolation schemes for the volume interpolation in a discrete half sphere in three-dimensional space, see Figure 4.25c. Like in the previous example, the function (4.59) is employed to generate the function values at the nodes of the tetrahedral FE mesh. Figure 4.27 visualizes the interpolation error. Again, the barycentric interpolation scheme performs best. Interestingly, RBF interpolation and inverse distance weighting obey a similar error, which can certainly be attributed to the average number of source points constituting the compact support of the interpolation schemes. Hence, it seems that the particular choice of the scaling parameter r_0 (RBF interpolation) or the cut-off radius r (inverse distance weighting) and the resulting number of source points contributing to the interpolation of the function value at a particular query point has a notable impact on the accuracy of these interpolation schemes. It can be assumed that including more source points to determine $u(\boldsymbol{q}^*)$ would have a positive effect on the accuracy of the interpolation. Yet, including too many source points will compromise the efficiency of these interpolation methods.

Summing up, the above examples provide a clear indication of the accuracy of the presented mesh-independent interpolation schemes. Irrespective of the considered dimension of space, the barycentric interpolation scheme exhibits by far the best performance. Even if only a moderate number of nearest neighbors are taken into account to generate the local Delaunay tessellation, the results are very accurate in all considered examples. Yet, the effort invested to find the nearest neighbors and to triangulate this point set is rather high and, very importantly, almost always higher than the cost associated to the projec-

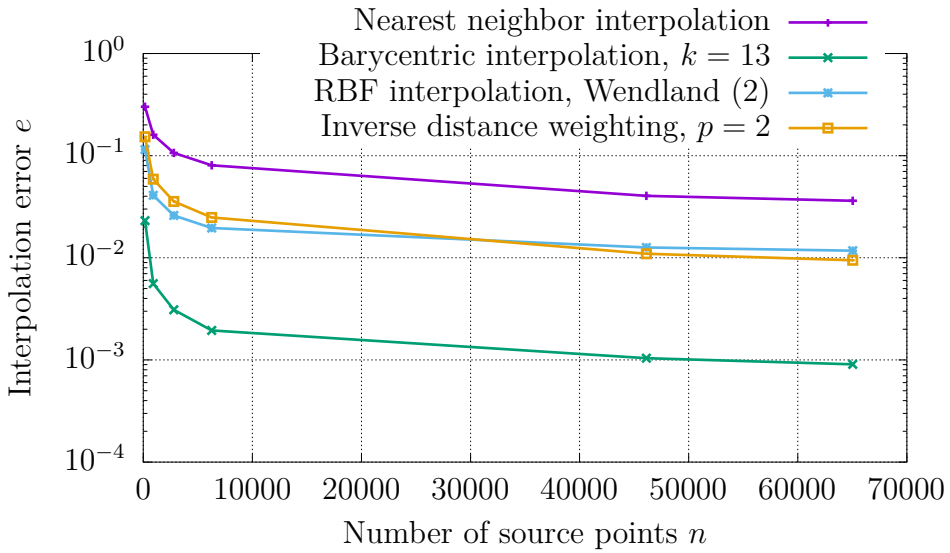


Figure 4.27: Interpolation errors of mesh-independent interpolation schemes on a discretized half sphere.

tion procedure outlined in Section 4.5.5. If it is possible to access the mesh topology, it is therefore advisable to take advantage of that information instead of trying to reconstruct the surface indirectly by a local Delaunay tessellation. RBF interpolation still produces acceptable results, which, however, are far less accurate than those of the barycentric interpolation. Inverse distance weighting performs moderately. It should be emphasized that the accuracy of both the RBF interpolation and the inverse distance weighting is strongly influenced by the number of source points forming the local support for interpolation; it can therefore be expected that the schemes produce different interpolation errors depending on the particular choice of the user-defined configuration parameters. It is, however, hard to make a reasonable choice for the scaling parameter r_0 or the cut-off radius r a priori, as these parameters depend on the mesh density (which may also vary locally) and on the local variation of the discrete function values. Due to the difficulty of adjusting the interpolation schemes in order to achieve a certain accuracy while at the same time retaining their efficiency, these mesh-independent interpolation schemes do not offer any significant advantage over the mesh-based projection procedure. Nearest neighbor interpolation is very efficient and robust, but also leads to high interpolation errors, which decrease only slowly with increasing source point density. For all examined examples, nearest neighbor interpolation can thus not be considered as a reasonable option. Summarizing the above, at least for the examples considered here, the mesh-independent interpolation methods do not exhibit any clear benefits as compared to mesh-based interpolation schemes.

Yet, the examples are all based on FE discretizations and on the interpolation of a continuous field $u = u(\boldsymbol{x})$. In this case, the interpolation by means of projection can be considered optimal in the sense of the underlying discretization scheme. However, the definition of a "good" interpolation scheme becomes diffuse if discretization schemes other than the FEM are considered. For instance, in the FVM or the BEM applied for fluid problems, the traction is available at the face or panel centers, but the *variation* of the traction across the surface is not sharply defined. From a physical point of view, a constant traction per face or panel clearly poses a comparably bad assumption. Rather,

Table 4.4: Comparison of the advantages and disadvantages of mesh-independent and mesh-based interpolation schemes.

	Mesh-independent interpolation schemes	**Mesh-dependent interpolation schemes**
Advantages	• No need to access mesh data structure • Generic in the sense that they can be applied to arbitrary discretizations • Robust (nearest neighbor and barycentric interpolation)	• Highly accurate • Resolves geometric features such as sharp edges • Robust
Disadvantages	• Rather inaccurate, in particular for coarse meshes • Inherent difficulty to resolve geometric features such as sharp edges • Sensitive to user parameters (inverse distance weighting, RBF interpolation)	• Need to access mesh data structure • Projection procedure required to associate query points to elements of the discretization

an interpolation involving the neighboring faces or panels as illustrated in Section 4.5.6 represents a better approximation of reality. In this light, a mesh-based interpolation again offers notable advantages over mesh-independent interpolation schemes. It features sharp edge detection capability, does not require any user input (except for the threshold angle θ^*, admittedly, but $\theta^* = \pi/3$ is most often a reasonable choice), and incorporates an inherent adaptivity to local changes in the mesh density.

Table 4.4 once again summarizes the advantages and disadvantages of mesh-independent and mesh-based interpolation schemes. Evidently, mesh-based interpolation is appropriate in most cases. The effort invested into the projection procedure is usually amortized over the number of time increments in the simulation. If no relative motion between source and target discretization occurs, it suffices to compute the local coordinates corresponding to the query points and the associated interpolation weights once at the beginning of the coupling procedure.

It remains to remark that our comparison only includes consistent interpolation schemes. Conservative interpolation schemes were deliberately excluded due to the fact that it is difficult to define an objective error measure for these schemes.

4.6 Convergence Criteria

In the generic partitioned coupling procedure outlined in Algorithm 1, the implicit iterations within a time increment are carried out until all the subfields are equilibrated with each other to sufficient accuracy. Hence, a convergence criterion needs to be defined to decide when to stop the implicit iteration and to proceed to the next time increment. Due to the fact that the subfield residuals are controlled by the individual solvers, it suffices to apply the convergence criterion (or a combination of different convergence criteria) to the interface residual $\mathbf{r}_{j+1}^k$. In addition, it may be necessary for the stability of the cou-

pling procedure to include the interface residuals of the other coupling quantities in the convergence check as well.

In general, absolute or relative convergence criteria can be used. Absolute criteria are based on the evaluation of the p-norm $\|\mathbf{r}_{j+1}^k\|_p$ of the interface residual, where $1 \leq p < \infty$. To make this scalar measure independent of the number of degrees of freedom n on the interface discretization, it is useful to divide by n in order to obtain

$$\frac{1}{n}\|\mathbf{r}_{j+1}^k\|_p \leq \varepsilon \ , \tag{4.60}$$

see [56, p. 49], for instance. Relative convergence criteria are independent of the magnitude of the coupling quantity by definition, as they are divided by an appropriate reference value:

$$\frac{\|\mathbf{r}_{j+1}^k\|_p}{\|\mathbf{r}_{j+1}^0\|_q} \quad \text{or} \quad \frac{\|\mathbf{r}_{j+1}^k\|_p}{\|\mathbf{r}_{j+1}^{k-1}\|_q} \ . \tag{4.61}$$

Note that $p \neq q$ in the general case, but $p = q$ is appropriate in most cases.

4.7 Convergence Acceleration Schemes

An essential part of the generic partitioned coupling procedure depicted in Algorithm 1 is the convergence acceleration scheme $\mathcal{A}$. If chosen properly, it does not only stabilize the solution procedure but also has significant influence on the efficiency by reducing the number of implicit iterations required to equilibrate the subfields with each other. A broad range of different convergence acceleration procedures has already been proposed in the literature, and it is merely impossible to give a complete overview. Therefore, the discussion is limited to the convergence acceleration schemes that are most often used in practice and are also applied to solve the numerical examples in Chapter 6 and 7.

In what follows, it will turn out useful to introduce the parameter $k_{\max}$, which indicates that a convergence acceleration scheme $\mathcal{A}$ is applied in only every $k_{\max}$th iteration. Otherwise, the solution $\mathbf{u}_{j+1}^k$ is assigned to $\tilde{\mathbf{u}}_{j+1}^{k+1}$ without modification:

$$\tilde{\mathbf{u}}_{j+1}^{k+1} = \tilde{\mathbf{u}}_{j+1}^k + \mathbf{r}_{j+1}^k = \tilde{\mathbf{u}}_{j+1}^k + \left(\mathbf{u}_{j+1}^k - \tilde{\mathbf{u}}_{j+1}^k\right) = \mathbf{u}_{j+1}^k \ . \tag{4.62}$$

If $k_{\max} = 1$, the solution is accelerated in *every* iteration.

Convergence acceleration schemes can basically be classified into two major categories. Following [22, p. 27], the first family of convergence acceleration schemes can be constructed from so-called *vector sequence acceleration methods*, which are based on the idea of transforming a given series

$$S := \{\mathbf{u}^0, \ldots, \mathbf{u}^{k-1}, \mathbf{u}^k\} \tag{4.63}$$

converging towards a limit $\mathbf{u}^*$ into a sequence

$$\tilde{S} := \{\tilde{\mathbf{u}}^0, \ldots, \tilde{\mathbf{u}}^{k-1}, \tilde{\mathbf{u}}^k\} \tag{4.64}$$

that converges faster to $\mathbf{u}^*$ than the original sequence S such that

$$\lim_{k\to\infty} \frac{\|\tilde{\mathbf{u}}^k - \mathbf{u}^*\|_2}{\|\mathbf{u}^k - \mathbf{u}^*\|_2} = 0 \ . \tag{4.65}$$

A sequence transformation like this can be generated by using a certain number of iterates $\mathbf{u}^k, \mathbf{u}^{k-1}, \ldots$ from the original series S to construct an element $\tilde{\mathbf{u}}^k$ from the transformed series $\tilde{S}$. In this work, we present several vector variants of Aitken's classical δ-squared process [2] as a typical representative of a sequence transformation. Furthermore, we reformulate numerous other vector sequence acceleration schemes in such a way that they can be applied for convergence acceleration in a partitioned solution approach.

From the formulation of the generic coupling scheme depicted in Algorithm 1, it is evident that the partitioned solution approach can also be interpreted as a fixed-point problem, in which we aim to solve the nonlinear algebraic system of equations

$$\mathbf{r}_{j+1}^k = \mathcal{S}_m \left(\cdots \mathcal{S}_2 \left(\mathcal{I}_{1,2} \left(\mathcal{S}_1 \left(\tilde{\mathbf{u}}_{j+1}^k \right) \right) \right) \right) - \tilde{\mathbf{u}}_{j+1}^k = \mathbf{0} \, . \tag{4.66}$$

If the Newton-Raphson method is applied to solve this system, we are led to the iterative process

$$\begin{aligned} \left. \frac{\partial \mathbf{r}_{j+1}^k}{\partial \tilde{\mathbf{u}}_{j+1}} \right|_{\tilde{\mathbf{u}}_{j+1} = \tilde{\mathbf{u}}_{j+1}^k} \Delta \tilde{\mathbf{u}}_{j+1}^k &= -\mathbf{r}_{j+1}^k \, , \\ \tilde{\mathbf{u}}_{j+1}^{k+1} &= \tilde{\mathbf{u}}_{j+1}^k + \Delta \tilde{\mathbf{u}}_{j+1}^k \, . \end{aligned} \tag{4.67}$$

Due to the fact that different discretization schemes and different solvers are involved, a direct evaluation of the Jacobian matrix in Equation (4.67) is not possible in the partitioned approach and, moreover, would also violate the black-box idea as *the* cornerstone of the partitioned approach. The problem that the Jacobian is not directly accessible leads to the idea of finding an approximation of the Jacobian matrix, which has motivated the class of inexact- or quasi-Newton methods. The Broyden method and the quasi-Newton least squares method belong to the most effective methods for convergence acceleration in the partitioned solution of even strongly coupled multifield problems, which is why they will be briefly described in the following.

4.7.1 Constant Relaxation

Probably the most simple convergence acceleration scheme is the static or constant relaxation. In each iteration k, an updated solution $\tilde{\mathbf{u}}_{j+1}^{k+1}$ is computed by adding the increment $\Delta \mathbf{u}_{j+1}^k = \omega \mathbf{r}_{j+1}^k$ to the modified solution $\tilde{\mathbf{u}}_{j+1}^k$ of the previous iteration $k-1$ such that

$$\tilde{\mathbf{u}}_{j+1}^{k+1} = \tilde{\mathbf{u}}_{j+1}^k + \omega \mathbf{r}_{j+1}^k \, . \tag{4.68}$$

The relaxation factor $\omega \in (0, 2]$ is held constant throughout the coupling algorithm and must therefore be chosen small enough to ensure a stable and convergent implicit iteration in each time increment. Consequently, it is rather unlikely that the constant relaxation scheme will provide optimal convergence characteristics in *every* time increment.

4.7.2 Aitken Relaxation and Related Methods

A possible remedy to the problems related to a constant relaxation factor ω is to employ a dynamic relaxation factor ω^k, which is recomputed in each iteration k. One of the most popular methods in this regard is Aitken's δ-squared process [2]. It was originally developed to accelerate scalar sequences but can easily be extended to the vector case. Following [106],

an accelerated sequence can be constructed given three subsequent iterates $\mathbf{u}^{k-2}$, $\mathbf{u}^{k-1}$, and $\mathbf{u}^{k}$:

$$\begin{aligned} \tilde{\mathbf{u}}^{k-1} &= \mathbf{u}^{k-2} + \omega^k \left(\mathbf{u}^{k-1} - \mathbf{u}^{k-2} \right) = \mathbf{u}^{k-2} + \omega^k \Delta \mathbf{u}^{k-1} \ , \\ \tilde{\mathbf{u}}^{k} &= \mathbf{u}^{k-1} + \omega^k \left(\mathbf{u}^{k} - \mathbf{u}^{k-1} \right) = \mathbf{u}^{k-1} + \omega^k \Delta \mathbf{u}^{k} \ . \end{aligned} \tag{4.69}$$

The relaxation factor ω^k is chosen such that it minimizes the expression

$$\begin{aligned} \underset{\omega^k}{\operatorname{argmin}} \left\| \tilde{\mathbf{u}}^{k} - \tilde{\mathbf{u}}^{k-1} \right\|_2 &= \underset{\omega^k}{\operatorname{argmin}} \left\| \mathbf{u}^{k-1} - \mathbf{u}^{k-2} + \omega^k \left(\Delta \mathbf{u}^{k} - \Delta \mathbf{u}^{k-1} \right) \right\|_2 \\ &= \underset{\omega^k}{\operatorname{argmin}} \left\| \Delta \mathbf{u}^{k-1} + \omega^k \Delta^2 \mathbf{u}^{k} \right\|_2 \ , \end{aligned} \tag{4.70}$$

where $\Delta^2 \mathbf{u}^k := \Delta \mathbf{u}^k - \Delta \mathbf{u}^{k-1}$. From this, we obtain

$$\omega^k = - \frac{\Delta \mathbf{u}^{k\,\mathrm{T}} \Delta^2 \mathbf{u}^{k-1}}{\| \Delta^2 \mathbf{u}^k \|_2^2} \ . \tag{4.71}$$

Herein, the Moore-Penrose inverse [22, p. 217]

$$\mathbf{v}^{-1} := \frac{\mathbf{v}}{\| \mathbf{v} \|_2^2} \tag{4.72}$$

of a vector $\mathbf{v}$ was used.

If adopted to the generic partitioned solution procedure, the update formula applied in every $k_{\max} = 3$rd iteration reads

$$\tilde{\mathbf{u}}_{j+1}^{k+1} = \mathbf{a}^k + \omega^k \mathbf{b}^k \ . \tag{4.73}$$

In this update rule, the parameters $\mathbf{a}^k$, $\mathbf{b}^k$, and ω^k were introduced to cover the several different variants of the classical Aitken relaxation, some of which are listed in Table 4.5.

A notable disadvantage of the convergence acceleration schemes based on the classical Aitken relaxation is the fact that a solution update is computed only in every third iteration, which can lead to a high number of iterations before convergence acceleration actually takes effect. A modification of the classical Aitken relaxation was therefore proposed by Irons et al. [83], who reformulated the scheme with the aim to apply a relaxation in *every* iteration. In the first iteration $k = 0$, a user-defined initial relaxation factor ω^0 must be provided. In subsequent iterations $k > 0$, the relaxation factor is then updated according to

$$\omega^k = -\omega^{k-1} \, \Delta \mathbf{r}_{j+1}^{k\,\mathrm{T}} \mathbf{r}_{j+1}^{k-1} \big/ \| \mathbf{r}_{j+1}^{k} \|_2^2 \ . \tag{4.74}$$

In a comparative study for different convergence acceleration schemes applied to FSI, Minami et al. [113] describe the line extrapolation method, which was first applied in [184] and can be understood as an extension to the scheme proposed by Irons et al. [83]. Likewise applicable in every coupling iteration, the update rule reads [113, p. 1133]

$$\tilde{\mathbf{u}}_{j+1}^{k+1} = \alpha^k \tilde{\mathbf{u}}_{j+1}^{k} + \left(1 - \alpha^k \right) \tilde{\mathbf{u}}_{j+1}^{k-1} + \beta \left(\alpha^k \mathbf{r}_{j+1}^{k} + \left(1 - \alpha^k \right) \mathbf{r}_{j+1}^{k-1} \right) \ . \tag{4.75}$$

The factor α^k can be interpreted as a relaxation parameter, which is set to $\alpha^0 := 1$ if $k = 0$ and amounts to

$$\alpha^k = - \Delta \mathbf{r}^{k\,\mathrm{T}} \mathbf{r}_{j+1}^{k-1} \big/ \| \Delta \mathbf{r}^{k} \|_2 \tag{4.76}$$

Table 4.5: Variants of the classical Aitken relaxation [106, 44, p. 92].

$\mathbf{a}^k$	ω^k	$\mathbf{b}^k$	Reference
$\tilde{\mathbf{u}}_{j+1}^k$	$-\frac{\mathbf{r}_{j+1}^{k-1\,\mathrm{T}}\Delta\mathbf{r}_{j+1}^k}{\|\Delta\mathbf{r}_{j+1}^k\|_2^2}$	$\mathbf{r}_{j+1}^k$	Aitken [2]
$\tilde{\mathbf{u}}_{j+1}^k$	$-\frac{\|\mathbf{r}_{j+1}^{k-1}\|_2^2}{\mathbf{r}_{j+1}^{k-1\,\mathrm{T}}\Delta\mathbf{r}_{j+1}^k}$	$\mathbf{r}_{j+1}^k$	Graves-Morris [60]
$\mathbf{u}_{j+1}^k$	$-\frac{\|\Delta\mathbf{r}^k\|_2^2}{\Delta\mathbf{r}^{k\,\mathrm{T}}\left(\mathbf{r}_{j+1}^k+\mathbf{r}_{j+1}^{k-1}\right)}$	$\mathbf{r}_{j+1}^k+\mathbf{r}_{j+1}^{k-1}$	Iguchi [78]
$\mathbf{u}_{j+1}^k$	$-\frac{\|\mathbf{r}_{j+1}^k\|_2^2}{\mathbf{r}_{j+1}^{k\,\mathrm{T}}\Delta\mathbf{r}_{j+1}^k}$	$\mathbf{r}_{j+1}^k$	Zienkiewicz et al. [186]
$\mathbf{u}_{j+1}^k$	$-\frac{\mathbf{r}_{j+1}^{k\,\mathrm{T}}\mathbf{r}_{j+1}^{k-1}}{\mathbf{r}_{j+1}^{k-1\,\mathrm{T}}\Delta\mathbf{r}_{j+1}^k}$	$\mathbf{r}_{j+1}^k$	Jennings [86]
$\mathbf{u}_{j+1}^k$	$-\frac{\|\mathbf{r}_{j+1}^k\|_2^2}{\|\Delta\mathbf{r}_{j+1}^k\|_2^2}$	$\Delta\mathbf{r}_{j+1}^k$	Arthur in [106]

in iterations $k > 0$, where

$$\Delta\mathbf{r}^k = \mathbf{r}_{j+1}^k - \mathbf{r}_{j+1}^{k-1} \tag{4.77}$$

denotes the difference of subsequent residuals. The second parameter $0 < \beta \leq 1$ is a user-defined line search parameter that serves to circumvent a situation in which the search space is limited to a single line.

4.7.3 Vector ε-Algorithm

Originally developed for scalar sequences by Wynn [183] in the mid of the 1950s, Wynn's ε-algorithm was later extended to vector sequences and henceforth termed *vector ε-algorithm* [22, p. 216]. It is constructed from the recursive formula

$$\begin{aligned} &\boldsymbol{\varepsilon}_{-1}^k = \mathbf{0}\,, \quad \boldsymbol{\varepsilon}_0^k = \mathbf{u}^k\,, && k = 0, 1, \ldots \\ &\boldsymbol{\varepsilon}_{\ell+1}^k = \boldsymbol{\varepsilon}_{\ell-1}^{k+1} + \left(\boldsymbol{\varepsilon}_\ell^{k+1} - \boldsymbol{\varepsilon}_\ell^k\right)^{-1}\,, && k, \ell = 0, 1, \ldots \end{aligned} \tag{4.78}$$

Given a convergent series $\mathbf{u}^{k-1}, \mathbf{u}^k, \ldots$, a simple vector sequence acceleration scheme is constructed by choosing $\ell = 1$ and by evaluating the terms needed to compute $\boldsymbol{\varepsilon}_2^{k-2}$. Trivially,

$$\boldsymbol{\varepsilon}_0^{k-2} = \mathbf{u}^{k-2}\,, \quad \boldsymbol{\varepsilon}_0^{k-1} = \mathbf{u}^{k-1}\,, \quad \boldsymbol{\varepsilon}_0^k = \mathbf{u}^k\,. \tag{4.79}$$

Next, we have

$$\boldsymbol{\varepsilon}_1^{k-2} = \underbrace{\boldsymbol{\varepsilon}_{-1}^{k-1}}_{=\mathbf{0}} + \left(\boldsymbol{\varepsilon}_0^{k-1} - \boldsymbol{\varepsilon}_0^{k-2}\right)^{-1}\,, \quad \boldsymbol{\varepsilon}_1^{k-1} = \underbrace{\boldsymbol{\varepsilon}_{-1}^{k}}_{=\mathbf{0}} + \left(\boldsymbol{\varepsilon}_0^{k} - \boldsymbol{\varepsilon}_0^{k-1}\right)^{-1}\,. \tag{4.80}$$

From the above expressions, we can finally compute

$$\tilde{\mathbf{u}}_{j+1}^{k+1} = \boldsymbol{\varepsilon}_2^{k-2} = \boldsymbol{\varepsilon}_0^{k-1} + \left(\boldsymbol{\varepsilon}_1^{k-1} - \boldsymbol{\varepsilon}_1^{k-2}\right)^{-1} \tag{4.81}$$

in every $k_{\max} = 3$rd iteration.

4.7.4 Topological ε-Algorithm

The topological ε-algorithm [22, p. 222] is based on the recursion

$$\begin{aligned} \boldsymbol{\varepsilon}_{-1}^{k} &= \mathbf{0}\,, \quad \boldsymbol{\varepsilon}_{0}^{k} = \mathbf{u}^{k}\,, && k = 0, 1, \ldots \\ \boldsymbol{\varepsilon}_{2\ell+1}^{k} &= \boldsymbol{\varepsilon}_{2\ell-1}^{k+1} + \frac{\mathbf{y}}{\mathbf{y}^{\mathrm{T}} \Delta \boldsymbol{\varepsilon}_{2\ell}^{k+1}}\,, && k, \ell = 0, 1, \ldots \\ \boldsymbol{\varepsilon}_{2\ell+2}^{k} &= \boldsymbol{\varepsilon}_{2\ell}^{k+1} + \frac{\Delta \boldsymbol{\varepsilon}_{2\ell}^{k+1}}{\Delta \boldsymbol{\varepsilon}_{2\ell+1}^{k+1\ \mathrm{T}} \Delta \boldsymbol{\varepsilon}_{2\ell}^{k+2}}\,, && k, \ell = 0, 1, \ldots \end{aligned} \tag{4.82}$$

Therein, $\mathbf{y}$ represents an arbitrary non-zero vector, which solely serves to avoid a zero denominator. Once again considering a convergent series $\mathbf{u}^{k-1}, \mathbf{u}^{k}, \ldots$, we construct an acceleration scheme using $\ell = 0$ and evaluating the terms required to determine $\boldsymbol{\varepsilon}_2^{k-2}$. The first required non-zero terms are

$$\boldsymbol{\varepsilon}_0^{k-2} = \mathbf{u}^{k-2}\,, \quad \boldsymbol{\varepsilon}_0^{k-1} = \mathbf{u}^{k-1}\,, \quad \boldsymbol{\varepsilon}_0^{k} = \mathbf{u}^{k}\,. \tag{4.83}$$

In the next recursion level, we compute

$$\boldsymbol{\varepsilon}_1^{k-1} = \underbrace{\boldsymbol{\varepsilon}_{-1}^{k}}_{=\mathbf{0}} + \frac{\mathbf{y}}{\mathbf{y}^{\mathrm{T}} \left(\boldsymbol{\varepsilon}_0^{k} - \boldsymbol{\varepsilon}_0^{k-1}\right)}\,, \quad \boldsymbol{\varepsilon}_1^{k-2} = \underbrace{\boldsymbol{\varepsilon}_{-1}^{k-1}}_{=\mathbf{0}} + \frac{\mathbf{y}}{\mathbf{y}^{\mathrm{T}} \left(\boldsymbol{\varepsilon}_0^{k-1} - \boldsymbol{\varepsilon}_0^{k-2}\right)}\,. \tag{4.84}$$

Following that, we eventually arrive at

$$\tilde{\mathbf{u}}_{j+1}^{k+1} = \boldsymbol{\varepsilon}_2^{k-2} = \boldsymbol{\varepsilon}_0^{k-1} + \frac{\boldsymbol{\varepsilon}_0^{k-1} - \boldsymbol{\varepsilon}_0^{k-2}}{\left(\boldsymbol{\varepsilon}_1^{k-1} - \boldsymbol{\varepsilon}_1^{k-2}\right)^{\mathrm{T}} \left(\boldsymbol{\varepsilon}_0^{k} - \boldsymbol{\varepsilon}_0^{k-1}\right)} \tag{4.85}$$

for the application in every $k_{\max} = 3$rd implicit iteration.

4.7.5 Vector θ-Algorithm

In the vector θ-algorithm proposed in [22, p. 249], the recursion

$$\begin{aligned} \boldsymbol{\theta}_{-1}^{k} &= \mathbf{0}\,, \quad \boldsymbol{\theta}_{0}^{k} = \mathbf{u}^{k}\,, && k = 0, 1, \ldots \\ \boldsymbol{\theta}_{2\ell+1}^{k} &= \boldsymbol{\theta}_{2\ell-1}^{k+1} + \left(\Delta \boldsymbol{\theta}_{2\ell}^{k+1}\right)^{-1}\,, && k, \ell = 0, 1, \ldots \\ \boldsymbol{\theta}_{2\ell+2}^{k} &= \boldsymbol{\theta}_{2\ell}^{k+1} + \frac{\Delta \boldsymbol{\theta}_{2\ell+1}^{k+2\ \mathrm{T}} \Delta^2 \boldsymbol{\theta}_{2\ell+1}^{k+2}}{\|\Delta^2 \boldsymbol{\theta}_{2\ell+1}^{k+2}\|_2^2} \Delta \boldsymbol{\theta}_{2\ell}^{k+2}\,, && k, \ell = 0, 1, \ldots \end{aligned} \tag{4.86}$$

is employed to generate the elements of a transformed, faster-converging series. Choosing $\ell = 0$, the element $\boldsymbol{\theta}_2^{k-3}$ is constructed from the elements of the original series as follows. First of all, we have

$$\boldsymbol{\theta}_0^{k-3} = \mathbf{u}^{k-3}\,, \quad \boldsymbol{\theta}_0^{k-2} = \mathbf{u}^{k-2}\,, \quad \boldsymbol{\theta}_0^{k-1} = \mathbf{u}^{k-1}\,, \quad \boldsymbol{\theta}_0^{k} = \mathbf{u}^{k}\,. \tag{4.87}$$

Noting that

$$\boldsymbol{\theta}_1^{k-3} = \underbrace{\boldsymbol{\theta}_{-1}^{k-2}}_{=\mathbf{0}} + \left(\Delta\boldsymbol{\theta}_0^{k-2}\right)^{-1} , \quad \boldsymbol{\theta}_1^{k-2} = \underbrace{\boldsymbol{\theta}_{-1}^{k-1}}_{=\mathbf{0}} + \left(\Delta\boldsymbol{\theta}_0^{k-1}\right)^{-1} , \quad \boldsymbol{\theta}_1^{k-1} = \underbrace{\boldsymbol{\theta}_{-1}^{k}}_{=\mathbf{0}} + \left(\Delta\boldsymbol{\theta}_0^{k}\right)^{-1} , \tag{4.88}$$

the updated solution is computed from $\boldsymbol{\theta}_2^{k-3}$ in every $k_{\max} = 4$th iteration as

$$\begin{aligned}
\tilde{\mathbf{u}}_{j+1}^{k+1} = \boldsymbol{\theta}_2^{k-3} &= \boldsymbol{\theta}_0^{k-2} + \frac{\Delta\boldsymbol{\theta}_1^{k-1\,\mathrm{T}}\Delta^2\boldsymbol{\theta}_1^{k-1}}{\|\Delta^2\boldsymbol{\theta}_1^{k-1}\|_2^2}\Delta\boldsymbol{\theta}_0^{k-1} \\
&= \boldsymbol{\theta}_0^{k-2} + \frac{\left(\boldsymbol{\theta}_1^{k-1} - \boldsymbol{\theta}_1^{k-2}\right)^{\mathrm{T}}\left(\Delta\boldsymbol{\theta}_1^{k-1} - \Delta\boldsymbol{\theta}_1^{k-2}\right)}{\left\|\Delta\boldsymbol{\theta}_1^{k-1} - \Delta\boldsymbol{\theta}_1^{k-2}\right\|_2^2}\Delta\boldsymbol{\theta}_0^{k-1} \\
&= \boldsymbol{\theta}_0^{k-2} + \frac{\left(\boldsymbol{\theta}_1^{k-1} - \boldsymbol{\theta}_1^{k-2}\right)^{\mathrm{T}}\left(\boldsymbol{\theta}_1^{k-1} - 2\boldsymbol{\theta}_1^{k-2} + \boldsymbol{\theta}_1^{k-3}\right)}{\left\|\boldsymbol{\theta}_1^{k-1} - 2\boldsymbol{\theta}_1^{k-2} + \boldsymbol{\theta}_1^{k-3}\right\|_2^2}\left(\boldsymbol{\theta}_0^{k-1} - \boldsymbol{\theta}_0^{k-2}\right) .
\end{aligned} \tag{4.89}$$

4.7.6 Generalized θ-Algorithm

A similar vector sequence acceleration method is the generalized θ-algorithm proposed in [22, p. 248]. It is based on the recursion

$$\begin{aligned}
\boldsymbol{\theta}_{-1}^{k} = \mathbf{0} , \quad \boldsymbol{\theta}_0^{k} = \mathbf{u}^{k} , \qquad & k = 0, 1, \dots \\
\boldsymbol{\theta}_{2\ell+1}^{k} = \boldsymbol{\theta}_{2\ell-1}^{k+1} + \frac{\mathbf{y}}{\mathbf{y}^{\mathrm{T}}\Delta\boldsymbol{\theta}_{2\ell}^{k+1}} , \qquad & k, \ell = 0, 1, \dots \\
\boldsymbol{\theta}_{2\ell+2}^{k} = \boldsymbol{\theta}_{2\ell}^{k+1} + \omega_\ell^k\boldsymbol{\delta}_{2\ell+1}^{k} , \quad \omega_\ell^k = -\frac{\mathbf{z}^{\mathrm{T}}\Delta\boldsymbol{\theta}_{2\ell}^{k+1}}{\mathbf{z}^{\mathrm{T}}\Delta\boldsymbol{\delta}_{2\ell+1}^{k+1}} , \quad \boldsymbol{\delta}_{2\ell+1}^{k} = \frac{\Delta\boldsymbol{\theta}_{2\ell}^{k+1}}{\Delta\boldsymbol{\theta}_{2\ell+1}^{k+1\,\mathrm{T}}\Delta\boldsymbol{\theta}_{2\ell}^{k+1}} , \\
k, \ell = 0, 1, \dots ,
\end{aligned} \tag{4.90}$$

where $\mathbf{y}$ and $\mathbf{z}$ are arbitrary non-zero vectors to avoid a zero denominator. By choosing $\ell = 0$, we are led to an acceleration scheme constructed as follows. First, we have

$$\boldsymbol{\theta}_0^{k-3} = \mathbf{u}^{k-3} , \quad \boldsymbol{\theta}_0^{k-2} = \mathbf{u}^{k-2} , \quad \boldsymbol{\theta}_0^{k-1} = \mathbf{u}^{k-1} , \quad \boldsymbol{\theta}_0^{k} = \mathbf{u}^{k} . \tag{4.91}$$

Building on this, the next elements of the transformed sequence become

$$\boldsymbol{\theta}_1^{k-3} = \underbrace{\boldsymbol{\theta}_{-1}^{k-2}}_{=\mathbf{0}} + \frac{\mathbf{y}}{\mathbf{y}^{\mathrm{T}}\left(\boldsymbol{\theta}_0^{k-2} - \boldsymbol{\theta}_0^{k-3}\right)} \tag{4.92}$$

$$\boldsymbol{\theta}_1^{k-2} = \underbrace{\boldsymbol{\theta}_{-1}^{k-1}}_{=\mathbf{0}} + \frac{\mathbf{y}}{\mathbf{y}^{\mathrm{T}}\left(\boldsymbol{\theta}_0^{k-1} - \boldsymbol{\theta}_0^{k-2}\right)} \tag{4.93}$$

$$\boldsymbol{\theta}_1^{k-1} = \underbrace{\boldsymbol{\theta}_{-1}^{k}}_{=\mathbf{0}} + \frac{\mathbf{y}}{\mathbf{y}^{\mathrm{T}}\left(\boldsymbol{\theta}_0^{k} - \boldsymbol{\theta}_0^{k-1}\right)} . \tag{4.94}$$

Finally, we compute the auxiliary quantities

$$\begin{aligned}
\boldsymbol{\delta}_1^{k-3} &= \frac{\boldsymbol{\theta}_0^{k-2} - \boldsymbol{\theta}_0^{k-3}}{\left(\boldsymbol{\theta}_1^{k-2} - \boldsymbol{\theta}_1^{k-3}\right)^{\mathrm{T}}\left(\boldsymbol{\theta}_0^{k-2} - \boldsymbol{\theta}_0^{k-3}\right)} , \quad \boldsymbol{\delta}_1^{k-2} = \frac{\boldsymbol{\theta}_0^{k-1} - \boldsymbol{\theta}_0^{k-2}}{\left(\boldsymbol{\theta}_1^{k-1} - \boldsymbol{\theta}_1^{k-2}\right)^{\mathrm{T}}\left(\boldsymbol{\theta}_0^{k-1} - \boldsymbol{\theta}_0^{k-2}\right)} \\
\omega_0^{k-3} &= -\frac{\mathbf{z}^{\mathrm{T}}\left(\boldsymbol{\theta}_0^{k-1} - \boldsymbol{\theta}_0^{k-2}\right)}{\mathbf{z}^{\mathrm{T}}\left(\boldsymbol{\delta}_1^{k-2} - \boldsymbol{\delta}_1^{k-3}\right)}
\end{aligned} \tag{4.95}$$

to obtain the update rule

$$\tilde{\mathbf{u}}_{j+1}^{k+1} = \boldsymbol{\theta}_2^{k-3} = \boldsymbol{\theta}_0^{k-2} + \omega_0^{k-3}\boldsymbol{\delta}_1^{k-3} \tag{4.96}$$

to be applied in every $k_{\max} = 4$th coupling iteration.

4.7.7 Vector w-Transformation

The vector w-transformation is another interesting vector sequence acceleration method. It was first proposed by Osada [125] and follows the recursion

$$\begin{aligned}
\mathbf{w}_0^k &= \mathbf{u}^k\,, \qquad k = 0, 1, \ldots \\
\mathbf{w}_\ell^k &= \mathbf{w}_{\ell-1}^{k-1} + \left(1 - \frac{\Delta\mathbf{w}_{\ell-1}^{k-1\,\mathrm{T}}\Delta\mathbf{w}_{\ell-1}^{k-2}}{\Delta\mathbf{w}_{\ell-1}^{k-2\,\mathrm{T}}\Delta\mathbf{w}_{\ell-1}^{k-2}}\right) \\
&\qquad\qquad \cdot \left(\left(\Delta\mathbf{w}_{\ell-1}^{k}\right)^{-1} - 2\left(\Delta\mathbf{w}_{\ell-1}^{k-1}\right)^{-1} + \left(\Delta\mathbf{w}_{\ell-1}^{k-2}\right)^{-1}\right)^{-1}\,, \\
&\qquad \ell = 1, 2, \ldots; k = 3\ell, 3\ell+1, \ldots
\end{aligned} \tag{4.97}$$

One possible acceleration scheme is constructed by selecting $\ell = 1$, which requires the following sequence members in the transformed series. First of all, we set

$$\mathbf{w}_0^{k-3} = \mathbf{u}^{k-3}\,, \quad \mathbf{w}_0^{k-2} = \mathbf{u}^{k-2}\,, \quad \mathbf{w}_0^{k-1} = \mathbf{u}^{k-1}\,, \quad \mathbf{w}_0^{k} = \mathbf{u}^{k}\,. \tag{4.98}$$

Expanding the differences

$$\Delta\mathbf{w}_0^{k-2} = \mathbf{w}_0^{k-2} - \mathbf{w}_0^{k-3}\,, \quad \Delta\mathbf{w}_0^{k-1} = \mathbf{w}_0^{k-1} - \mathbf{w}_0^{k-2}\,, \quad \Delta\mathbf{w}_0^{k} = \mathbf{w}_0^{k} - \mathbf{w}_0^{k-1}\,, \tag{4.99}$$

the update formula for every $k_{\max} = 4$th iteration

$$\begin{aligned}
\tilde{\mathbf{u}}_{j+1}^{k+1} &= \mathbf{w}_1^k \\
&= \mathbf{w}_0^{k-1} + \left(1 - \frac{\Delta\mathbf{w}_0^{k-1\,\mathrm{T}}\Delta\mathbf{w}_0^{k-2}}{\Delta\mathbf{w}_0^{k-2\,\mathrm{T}}\Delta\mathbf{w}_0^{k-2}}\right)^{\mathrm{T}} \left(\left(\Delta\mathbf{w}_0^{k}\right)^{-1} - 2\left(\Delta\mathbf{w}_0^{k-1}\right)^{-1} + \left(\Delta\mathbf{w}_0^{k-2}\right)^{-1}\right)^{-1}
\end{aligned} \tag{4.100}$$

is acquired.

4.7.8 Euclidean w-Transformation

A close relative to the aforementioned vector w-transformation is the Euclidean w-transformation, which was likewise proposed by Osada [125] and is constructed from the recursive formula

$$\begin{aligned}
\mathbf{w}_0^k &= \mathbf{u}^k\,, \qquad k = 0, 1, \ldots \\
\mathbf{w}_\ell^k &= \mathbf{w}_{\ell-1}^{k-1} - \frac{\Delta\mathbf{w}_{\ell-1}^{k\,\mathrm{T}}\Delta\mathbf{w}_{\ell-1}^{k-2}}{\Delta\mathbf{w}_{\ell-1}^{k\,\mathrm{T}}\Delta^2\mathbf{w}_{\ell-1}^{k-2} - \Delta\mathbf{w}_{\ell-1}^{k-2\,\mathrm{T}}\Delta^2\mathbf{w}_{\ell-1}^{k-1}}\Delta\mathbf{w}_{\ell-1}^{k-1}\,, \\
&\qquad \ell = 1, 2, \ldots; k = 3\ell, 3\ell+1, \ldots
\end{aligned} \tag{4.101}$$

Choosing $\ell = 1$, the following sequence members need to be computed. Obviously, as before, the first elements are

$$\mathbf{w}_0^{k-3} = \mathbf{u}^{k-3} \,, \quad \mathbf{w}_0^{k-2} = \mathbf{u}^{k-2} \,, \quad \mathbf{w}_0^{k-1} = \mathbf{u}^{k-1} \,, \quad \mathbf{w}_0^{k} = \mathbf{u}^{k} \,. \tag{4.102}$$

Subsequently, we write down the differences

$$\Delta\mathbf{w}_0^{k-2} = \mathbf{w}_0^{k-2} - \mathbf{w}_0^{k-3} \,, \quad \Delta\mathbf{w}_0^{k-1} = \mathbf{w}_0^{k-1} - \mathbf{w}_0^{k-2} \,, \quad \Delta\mathbf{w}_0^{k} = \mathbf{w}_0^{k} - \mathbf{w}_0^{k-1} \,, \tag{4.103}$$

which are then used to generate the update rule

$$\tilde{\mathbf{u}}_{j+1}^{k+1} = \mathbf{w}_1^k = \mathbf{w}_0^{k-1} - \frac{\Delta\mathbf{w}_0^{k\,\mathrm{T}}\Delta\mathbf{w}_0^{k-2}}{\Delta\mathbf{w}_0^{k\,\mathrm{T}}\Delta^2\mathbf{w}_0^{k-2} - \Delta\mathbf{w}_0^{k-2\,\mathrm{T}}\Delta^2\mathbf{w}_0^{k-1}}\Delta\mathbf{w}_0^{k-1} \tag{4.104}$$

used for convergence acceleration in every $k_{\max} = 4$th iteration.

4.7.9 Broyden Method

The Broyden method was originally proposed by Broyden [23] and belongs to the class of quasi-Newton methods. It is based on the iterative process

$$\begin{aligned} \mathbf{B}^k\Delta\tilde{\mathbf{u}}_{j+1}^k &= \mathbf{r}_{j+1}^k \\ \tilde{\mathbf{u}}_{j+1}^{k+1} &= \tilde{\mathbf{u}}_{j+1}^k + \Delta\tilde{\mathbf{u}}_{j+1}^k \,, \end{aligned} \tag{4.105}$$

where the Jacobian matrix in (4.67) was replaced by the Broyden matrix $\mathbf{B}^k$. In the first iteration $k = 0$, $\mathbf{B}^0 = \mathbf{I}$ is the typical choice. In subsequent iterations $k > 0$, the Broyden matrix is updated according to

$$\mathbf{B}^{k+1} = \mathbf{B}^k + \left(\Delta\mathbf{r}_{j+1}^k - \mathbf{B}^k\Delta\tilde{\mathbf{u}}_{j+1}^k\right)^{\mathrm{T}}\left(\Delta\tilde{\mathbf{u}}_{j+1}^k\right)^{-1} = \mathbf{B}^k + \mathbf{v}^{k\,\mathrm{T}}\mathbf{w}^k \,, \tag{4.106}$$

where the abbreviations

$$\mathbf{v}^k := \frac{\Delta\mathbf{r}_{j+1}^k - \mathbf{B}^k\Delta\tilde{\mathbf{u}}_{j+1}^k}{\|\Delta\tilde{\mathbf{u}}_{j+1}^k\|_2} \qquad \text{and} \qquad \mathbf{w}^k := \frac{\Delta\tilde{\mathbf{u}}_{j+1}^k}{\|\Delta\tilde{\mathbf{u}}_{j+1}^k\|_2} \tag{4.107}$$

were introduced. The inverse $\mathbf{H}^{k+1} := (\mathbf{B}^{k+1})^{-1}$ is computed by means of the Sherman-Morrison formula [7]

$$\mathbf{H}^{k+1} = \left(\mathbf{B}^k + \mathbf{v}^{k\,\mathrm{T}}\mathbf{w}^k\right)^{-1} = \left(\mathbf{I} - \frac{\left(\mathbf{H}^k\mathbf{v}^k\right)^{\mathrm{T}}\mathbf{w}^k}{1 + \mathbf{w}^{k\,\mathrm{T}}\mathbf{H}^k\mathbf{v}^k}\right)\mathbf{H}^k \tag{4.108}$$

and, according to [92, p. 125], the inverse $(\mathbf{B}^k)^{-1} = \mathbf{H}^k$ amounts to

$$\mathbf{H}^k = \prod_{i=0}^{k-1}\mathbf{I} + \frac{\Delta\tilde{\mathbf{u}}_{j+1}^{i+1}\left(\tilde{\mathbf{u}}_{j+1}^i\right)^{\mathrm{T}}}{\|\Delta\tilde{\mathbf{u}}_{j+1}^i\|_2^2} \,. \tag{4.109}$$

Evidently, by computing $\mathbf{H}^{k+1}$, the increment $\Delta\tilde{\mathbf{u}}_{j+1}^k$ required to update $\tilde{\mathbf{u}}_{j+1}^k$ is obtained by matrix-vector products only. A drawback of Broyden's method in its original formulation is the necessity to keep a potentially large matrix in memory. To alleviate this disadvantage, a restart version was proposed in [92, pp. 123–127, 113, p. 1134], which – as it is sufficient to update only a vector instead of a whole matrix in each iteration – reduces the storage requirements significantly. Algorithm 10 gives an outline of this procedure.

function BROYDENMETHOD(iteration k, modified solution $\tilde{\mathbf{u}}_{j+1}^k$, residual $\mathbf{r}_{j+1}^k$, relaxation factors $\omega^0, \ldots, \omega^{k+1}$, number of iterations until restart k^*)
  $\tilde{k} := \mathrm{mod}(k-1, k^*) + 1$
  if $\tilde{k} >$ **then**
    $\mathbf{p} := \mathbf{r}_{j+1}^k$
    for $i = 0, 1, \ldots, k-1$ **do**
      $\alpha := \omega^i / \omega^{i+1}$, $\beta := \omega^i - 1$
      $\mathbf{p} := \mathbf{p} + \mathbf{s}^{i\mathrm{T}}\mathbf{p} / \|\mathbf{s}^i\|_2^2 \, (\alpha \mathbf{s}^{i+1} + \beta \mathbf{s}^i)$
    end for
    $\mathbf{s}^{k+1} := \left(\mathbf{p} - (1-\omega^k)\mathbf{s}^k\right) / \left(1 - \omega^k \mathbf{s}^{k\mathrm{T}}\mathbf{p} / \|\mathbf{s}^k\|_2^2\right)$
  else
    $\tilde{\mathbf{s}}_{j+1}^{k+1} := \mathbf{r}_{j+1}^{k+1}$
  end if
  return $\tilde{\mathbf{u}}_{j+1}^{k+1} := \tilde{\mathbf{u}}_{j+1}^k + \omega^{k+1}\mathbf{s}^{k+1}$
end function

Algorithm 10: Restart version of the Broyden method [113, p. 1134].

4.7.10 Quasi-Newton Least Squares Method

A relatively new but promising convergence acceleration scheme is the interface quasi-Newton method with inverse Jacobian from a least squares model, proposed by Degroote et al. [36]. As the name suggests, it is based on the idea of approximating the exact Jacobian matrix in (4.67) by a reduced order method. For a fully-converged solution, we desire to minimize the residual

$$\mathbf{r}_{j+1}^{k+1} = \mathbf{r}_{j+1}^k + \Delta\mathbf{r}_{j+1}^k \tag{4.110}$$

in the current iteration. Following [36], the residual increment $\Delta\mathbf{r}_{j+1}^k$ is approximated by the linear combination

$$\Delta\mathbf{r}_{j+1}^k \approx \sum_{i=0}^{k-1} \alpha_i^k \Delta\mathbf{r}_{j+1}^i \tag{4.111}$$

of residual increments $\Delta\mathbf{r}_{j+1}^i$ from previous iterations $i = 0, \ldots, k-1$. Combining (4.110) and (4.111), this leads to the minimization problem

$$\underset{\alpha_i^k}{\mathrm{argmin}} \left\|\mathbf{r}_{j+1}^{k+1}\right\|_2 \approx \underset{\alpha_i^k}{\mathrm{argmin}} \left\|\mathbf{r}_{j+1}^k + \sum_{i=0}^{k-1} \alpha_i^k \Delta\mathbf{r}_{j+1}^i\right\|_2 . \tag{4.112}$$

In order to solve (4.112), the matrix

$$\mathbf{V}^k = \begin{pmatrix} \Delta\mathbf{r}_{j+1}^{k-1} & \cdots & \Delta\mathbf{r}_{j+1}^0 \end{pmatrix} \tag{4.113}$$

is set up, and the system of equations

$$\mathbf{V}^k \boldsymbol{\alpha}^k = -\mathbf{r}_{j+1}^k \tag{4.114}$$

is solved in a least squares sense. To this end, $\mathbf{V}^k \in \mathbb{R}^{m \times k}$ is QR-decomposed such that $\mathbf{V}^k = \mathbf{Q}^k\mathbf{R}^k$. The vector $\boldsymbol{\alpha}^k$, if $m \geq k-1$, is then obtained from the solution of

$$\mathbf{R}^k \boldsymbol{\alpha}^k = \mathbf{Q}^{k\mathrm{T}} \left(-\mathbf{r}_{j+1}^k\right) \tag{4.115}$$

by back substitution without inverting $\mathbf{R}^k$ explicitly. It suffices to perform an economy-size QR factorization of $\mathbf{V}^k$, where $\mathbf{R}^k \in \mathbb{R}^{k \times k}$. For an increasing number of iterations k, the matrix $\mathbf{V}^k$ is prone to becoming increasingly ill-conditioned. It is therefore advisable to remove the ith column from $\mathbf{V}^k$ if [36, 62, p. 12]

$$|R_{ii}| < \varepsilon \qquad \text{or} \qquad |R_{ii}| < \varepsilon \|\mathbf{R}\|_2 \tag{4.116}$$

with a suitable tolerance ε and to recompute the QR decomposition. If required, the procedure is repeated until no further columns are deleted.

Algorithm 11 depicts the quasi-Newton least squares procedure suited for the direct integration into the generic partitioned solution strategy. In the first iteration, $k = 0$ and

1: **function** QuasiNewtonLeastSquaresMethod(iteration k, unmodified solution $\mathbf{u}_{j+1}^k$, modified solution $\tilde{\mathbf{u}}_{j+1}^k$, residual $\mathbf{r}_{j+1}^k$, updated matrices $\mathbf{V}^k$, $\mathbf{W}^k$, static relaxation factor ω)
2: **if** $k > 0$ **then**
3: $\Delta\mathbf{r}^k := \mathbf{r}_{j+1}^k - \mathbf{r}_{j+1}^{k-1}$, $\mathbf{V}^k := \begin{pmatrix} \Delta\mathbf{r}^k & \mathbf{V}^{k-1} \end{pmatrix}$
4: $\Delta\mathbf{u}^k := \mathbf{u}_{j+1}^k - \mathbf{u}_{j+1}^{k-1}$, $\mathbf{W}^k := \begin{pmatrix} \Delta\mathbf{u}^k & \mathbf{W}^{k-1} \end{pmatrix}$
5: Decompose $\mathbf{V}^k$ such that $\mathbf{V}^k = \mathbf{Q}^k\mathbf{R}^k$
6: Solve $\mathbf{R}^k\boldsymbol{\alpha}^k = {\mathbf{Q}^k}^{\mathrm{T}}(-\mathbf{r}_{j+1}^k)$
7: $\Delta\tilde{\mathbf{u}}_{j+1}^k := \mathbf{W}^k\boldsymbol{\alpha}^k + \mathbf{r}_{j+1}^k$
8: **else**
9: Initialize $\mathbf{V}^k, \mathbf{W}^k$
10: $\Delta\tilde{\mathbf{u}}_{j+1}^k := \omega\mathbf{r}_{j+1}^k$
11: **end if**
12: **return** $\tilde{\mathbf{u}}_{j+1}^{k+1} := \tilde{\mathbf{u}}_{j+1}^k + \Delta\tilde{\mathbf{u}}_{j+1}^k$
13: **end function**

Algorithm 11: Quasi-Newton least squares method [36, p. 796].

the increment $\Delta\tilde{\mathbf{u}}_{j+1}^k$ is determined by performing a relaxation step using the constant relaxation factor ω:

$$\Delta\tilde{\mathbf{u}}_{j+1}^0 = \omega\mathbf{r}_{j+1}^0 \,. \tag{4.117}$$

In the following iterations, $k > 0$ and the current residual difference $\Delta\mathbf{r}^k$ and the current unmodified solution difference $\Delta\mathbf{u}^k$ are prepended to the matrices $\mathbf{V}^{k-1}$ and $\mathbf{W}^{k-1}$ to generate the updated matrices $\mathbf{V}^k$ and $\mathbf{W}^k$, respectively. $\mathbf{V}^k$ is then QR decomposed to determine the coefficients $\boldsymbol{\alpha}^k$. The increment $\Delta\tilde{\mathbf{u}}_{j+1}^k$ is then determined according to

$$\Delta\tilde{\mathbf{u}}_{j+1}^k = \mathbf{W}^k\boldsymbol{\alpha}^k + \mathbf{r}_{j+1}^k \,. \tag{4.118}$$

Finally, using the increment (4.117) for $k = 0$ or (4.118) for $k > 0$, an updated solution is obtained from

$$\tilde{\mathbf{u}}_{j+1}^{k+1} = \tilde{\mathbf{u}}_{j+1}^k + \Delta\tilde{\mathbf{u}}_{j+1}^k \,. \tag{4.119}$$

By reusing the information from previous time steps, it is also possible to construct the matrix

$$\mathbf{V'}^k = \begin{pmatrix} \mathbf{V}_{j+1}^k & \mathbf{V}_j & \cdots & \mathbf{V}_{j-\ell+1} \end{pmatrix} \tag{4.120}$$

from the most recent matrix $\mathbf{V}_{j+1}^k$ in the time increment t_{j+1} and the last matrices $\mathbf{V}_j, \ldots, \mathbf{V}_{j-\ell+1}$ from the previous ℓ time increments. Analogously, the coefficients $\boldsymbol{\alpha}^k$ are computed by first performing a QR factorization of $\mathbf{V}'^k$ before solving

$$\mathbf{R}'^k \boldsymbol{\alpha}^k = \mathbf{Q}'^{k\mathrm{T}} \left(-\mathbf{r}_{j+1}^k\right) \tag{4.121}$$

by back substitution. Here, QR filtering becomes even more important, and one is well-advised to apply the criterion (4.116) to successively remove columns from $\mathbf{V}'^k$ in order to obtain a stable solution procedure. For the sake of completeness, this version of the quasi-Newton least squares procedure is depicted in Algorithm 12.

1: **function** MultiTimeStepQuasiNewtonLeastSquaresMethod(time step j, iteration k, unmodified solution $\mathbf{u}_{j+1}^k$, modified solution $\tilde{\mathbf{u}}_{j+1}^k$, residual $\mathbf{r}_{j+1}^k$, updated matrices $\mathbf{V}_{j+1}^k$, $\mathbf{W}_{j+1}^k$, static relaxation parameter ω)
2: **if** $k > 0$ **then**
3: $\Delta\mathbf{r}^k := \mathbf{r}_{j+1}^k - \mathbf{r}_{j+1}^{k-1}$, $\mathbf{V}_{j+1}^k := \begin{pmatrix}\Delta\mathbf{r}^k & \mathbf{V}_{j+1}^{k-1}\end{pmatrix}$
4: $\mathbf{V}'^k := \begin{pmatrix}\mathbf{V}_{j+1}^k & \mathbf{V}_j & \cdots & \mathbf{V}_{j-\ell+1}\end{pmatrix}$
5: $\Delta\mathbf{u}^k := \mathbf{u}_{j+1}^k - \mathbf{u}_{j+1}^{k-1}$, $\mathbf{W}_{j+1}^k := \begin{pmatrix}\Delta\mathbf{u}^k & \mathbf{W}_{j+1}^{k-1}\end{pmatrix}$
6: $\mathbf{W}' := \begin{pmatrix}\mathbf{W}_{j+1}^k & \mathbf{W}_j & \cdots & \mathbf{W}_{j-\ell+1}\end{pmatrix}$
7: Decompose $\mathbf{V}'^k$ such that $\mathbf{V}'^k = \mathbf{Q}^k\mathbf{R}^k$
8: Solve $\mathbf{R}^k\boldsymbol{\alpha}^k = \mathbf{Q}^{k\mathrm{T}}(-\mathbf{r}_{j+1}^k)$
9: $\Delta\tilde{\mathbf{u}}_{j+1}^k := \mathbf{W}^k\boldsymbol{\alpha}^k + \mathbf{r}_{j+1}^k$
10: **else**
11: Initialize $\mathbf{V}_{j+1}^k, \mathbf{W}_{j+1}^k$
12: $\Delta\tilde{\mathbf{u}}_{j+1}^k := \omega\mathbf{r}_{j+1}^k$
13: **end if**
14: **return** $\tilde{\mathbf{u}}_{j+1}^{k+1} := \tilde{\mathbf{u}}_{j+1}^k + \Delta\tilde{\mathbf{u}}_{j+1}^k$
15: **end function**

Algorithm 12: Variant of the quasi-Newton least squares method reusing information from previous time increments [36, p. 796].

4.7.11 Comparison of Convergence Acceleration Schemes

In order to compare the convergence acceleration schemes proposed in the previous sections, a partitioned solution procedure is applied to solve the simple mechanical system introduced in Section 4.3. The convergence acceleration schemes serve to accelerate convergence and to reduce the number of required implicit iterations within a time increment. For the assessment of the numerical effort, the number of iterations in each time increment are accumulated over all time increments and afterwards divided by the number of time increments so as to obtain a mean number of iterations per time increment. Since the number of implicit iterations correlates directly with the number of subsystem evaluations for the convergence acceleration schemes proposed in this work, the mean number of iterations per time increment is an appropriate measure for the total computational cost of the entire solution procedure. In order to decouple the results from the influence of the predictor, a second-order polynomial predictor is chosen in all cases.

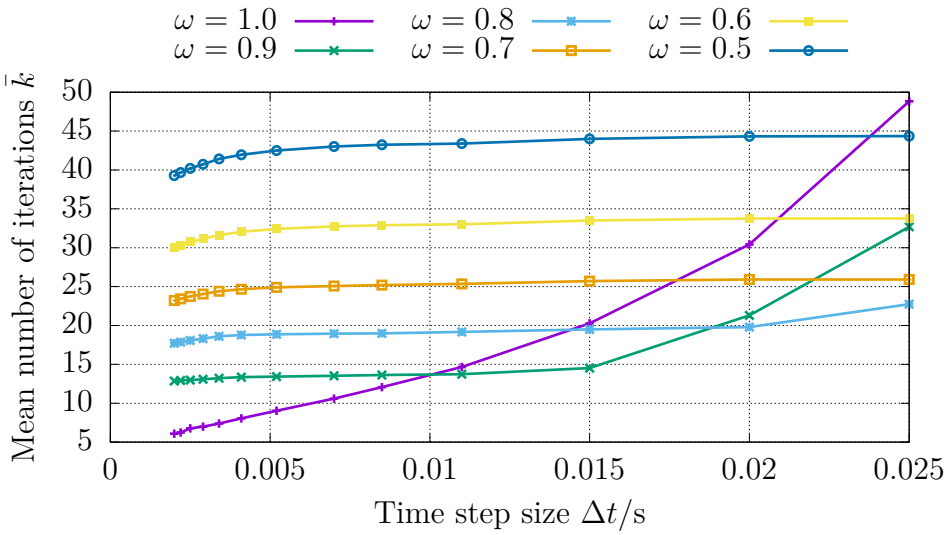


Figure 4.28: Influence of the relaxation factor ω on the performance of the constant relaxation scheme.

In Figure 4.28, the influence of the relaxation factor ω on the performance of the constant relaxation scheme is compared. A relaxation factor $\omega = 1$ corresponds to the unrelaxed case, where the modified displacement $\tilde{\mathbf{d}}_{j+1}^{k+1}$ equals the unmodified solution $\mathbf{d}_{j+1}^{k}$. Evidently, the mean number of iterations $\bar{k}$ is high for large time step sizes Δt, but it decreases rapidly with decreasing time step size. For the smallest investigated time step size, a relaxation factor $\omega = 1$ achieves the lowest mean number of iterations as compared to the other constant relaxation schemes with $\omega < 1$. For larger time step sizes Δt, we observe that the smaller the relaxation factor ω, the higher the reduction in the average number of iterations $\bar{k}$. However, this tendency diminishes with decreasing time step size, and the relaxation schemes with $\omega < 1$ eventually exhibit a worse performance than the unrelaxed scheme.

In the next step, we investigate the performance of the various Aitken-type relaxation schemes presented in Section 4.7.2. Figure 4.29 reveals that all relaxation schemes based on the classical Aitken relaxation are capable of reducing the mean number of iterations $\bar{k}$ as compared to the constant relaxation scheme using $\omega = 1$. Similar to the constant relaxation schemes, the number of iterations reduces notably as the time step size Δt becomes smaller. For the considered example, the variants of the classical Aitken relaxation scheme – likewise applied in every third coupling iteration, and listed in Table 4.5 – do not lead to a performance benefit as compared to the original version. In contrast, the modifications developed by Iguchi [78] and Arthur [106] deteriorate the performance conspicuously. Yet, the variant of Irons et al. [83], which can be applied in each coupling iteration instead of in every third, produces slightly better results, in particular for larger time step sizes. The advantage diminishes for smaller time step sizes, and almost no difference is observed for $\Delta t < 0.015\,\mathrm{s}$.

Subsequently, we investigate and compare the performance of the other vector sequence acceleration-based schemes presented in Section 4.7.3–4.7.8. The average number of coupling iterations per time increment is depicted in Figure 4.30. Apparently, the vector ε-algorithm and the topological ε-algorithm perform best. Regarding the mean number

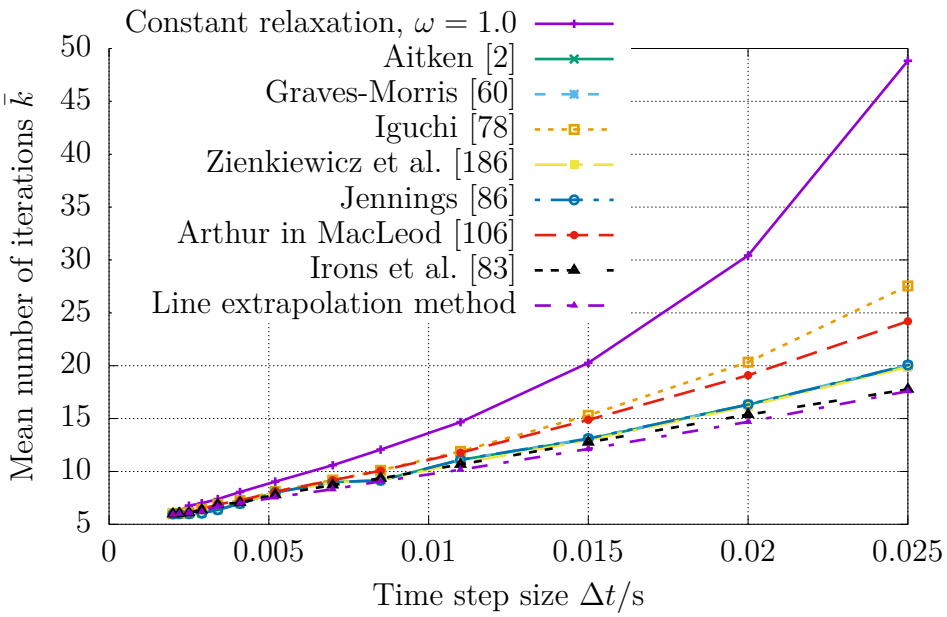


Figure 4.29: Comparison of the performance of Aitken-type relaxation schemes. The constant relaxation scheme using $\omega = 1$ is plotted for reference.

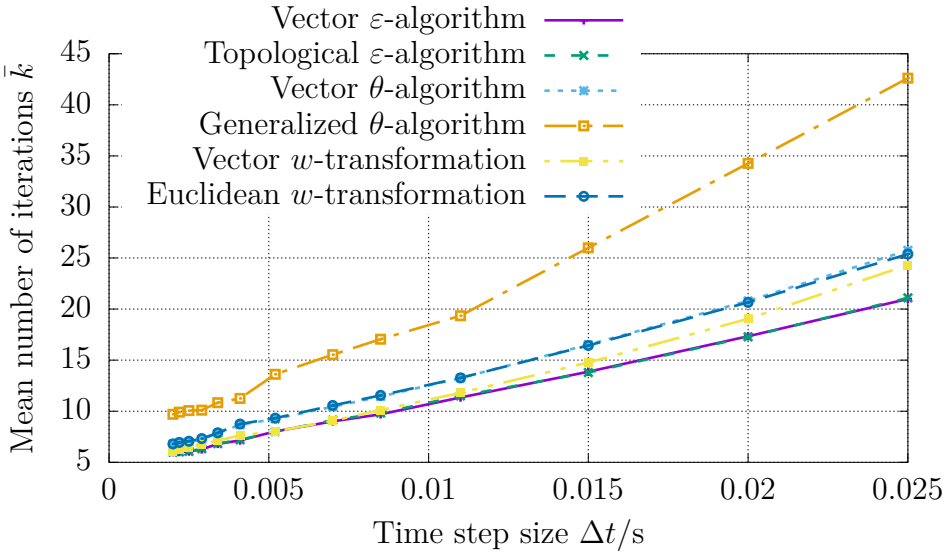


Figure 4.30: Comparison of the performance of the several vector-sequence acceleration-based schemes.

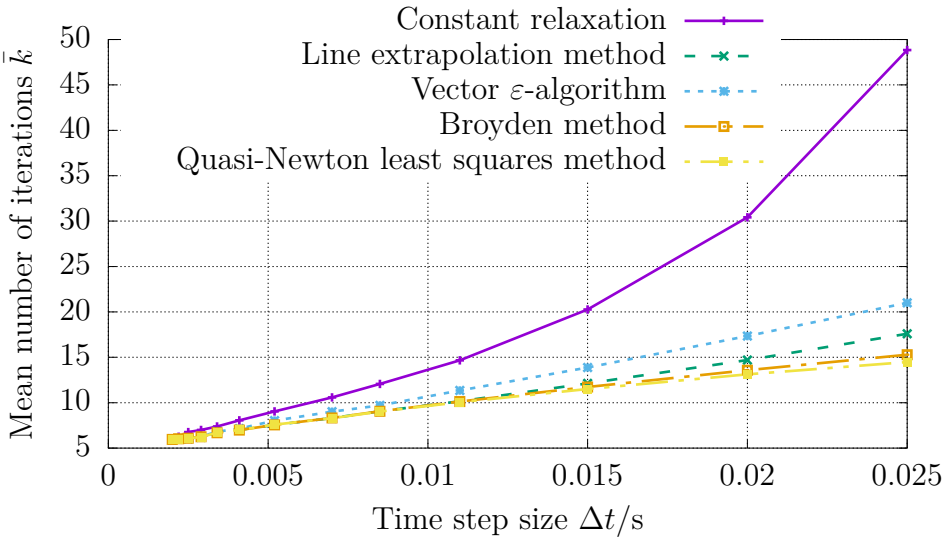


Figure 4.31: Comparison of the performance of various convergence acceleration schemes.

of required implicit iterations, hardly any difference between these schemes can be observed. The vector w-transformation exhibits a slightly worse performance throughout the entire range of considered time step sizes. The vector θ-algorithm and the Euclidean w-transformation exhibit an almost equal but slightly inferior performance as compared to the schemes mentioned before. The generalized θ-algorithm is not competitive for the considered example, as it requires a significantly higher number of iterations in average until convergence is achieved.

Finally, we compare the best performing convergence acceleration schemes based on vector sequence acceleration to those from the class of quasi-Newton methods. In Figure 4.31, we once again plot the mean number of iterations $\bar{k}$ against the time step size Δt. Most importantly, the graphs indicate that the use of a convergence acceleration scheme usually reduce the number of required coupling iterations significantly and, hence, have a notable impact on the computational efficiency of the entire solution procedure. The quasi-Newton least squares method obeys the best convergence characteristics throughout the entire range of considered time step sizes. Remarkably, the quasi-Newton least squares procedure requires only one third of the iterations of the unrelaxed scheme for the largest time step size, which represents a huge boost in performance. The Broyden method requires only slightly more iterations on average as compared to the quasi-Newton least squares method. The line extrapolation method exhibits a satisfactory convergence behavior as well. The vector ε-algorithm – as the best of the vector sequence acceleration-based schemes, apart from the classical Aitken relaxation – experiences a notably worse performance as compared to the schemes mentioned before. Nonetheless, the performance gain as compared to the constant relaxation using $\omega = 1$ is still remarkable, especially for larger time step sizes.

Summarizing the results, the quasi-Newton least squares method, the Broyden method, and the line extrapolation scheme belong to the best-performing convergence acceleration schemes. It should be noted, however, that the results only apply to a comparably simple problem originating from the partitioned analysis of the mechanical system presented in Section 4.3. It is difficult to generalize the findings to arbitrary two-field or multifield

problems involving even more than two fields. Yet, the conclusions drawn for this simple example can serve as useful hints for the choice of a convergence acceleration scheme for larger-scale problems.

5 Software Library *comana*

For the partitioned solution of a general multifield problem, separate dedicated solvers are used for each of the involved subproblems. Interaction between the subproblems is achieved by iteratively exchanging the relevant field quantities at the boundaries $\Gamma_1, \ldots, \Gamma_m$ in the case of a surface-coupled problem or in the subdomains $\Omega_1, \ldots, \Omega_m$ in the case of a volume-coupled problem within a time increment. If all subfields are equilibrated with each other, we proceed to the next time increment. In order to steer the coupled solution procedure and organize the data transfer between the solvers, we propose the C++ software library *comana*. It is based on a master/slave communication model suitable for the integration of serial as well as shared- or distributed-memory parallelized solvers into a partitioned analysis. Due to the modular architecture of the coupling extensions, modifications in the solvers are kept to a minimum. For the implementation of a custom partitioned solution strategy in a dedicated C++ program, *comana* offers a vast range of different modular, reusable, and extensible algorithmic building blocks. This way, the user may exploit the full power of the C++ language in order to set up the solution procedure, to easily manipulate the relevant field quantities, or to alter the order in which the subproblems are to be solved. Shared-memory parallelized building blocks contribute to reducing the time spent on executing the coupling algorithm.

In this chapter, we provide a concise overview of the general concepts on which *comana* is based. Then, we will present the most important data structures and their efficient use for the algorithmic tasks arising in the context of a partitioned solution procedure. Particular emphasis is placed on the discussion of the data structures required to prepare a solver for participation in a partitioned analysis. The simple and straightforward integration of solvers implemented in C/C++, Fortran, Python, MATLAB/Octave, or the ANSYS Parametric Design Language (APDL) can be considered as one of the most outstanding strengths of *comana*. Last but not least, a simple coupled problem is analyzed with the help of *comana* to illustrate the simulation setup and to demonstrate the usefulness of our software library for the implementation of a tailored partitioned solution strategy.

5.1 General Concepts

To conveniently address the different solvers involved in the partitioned solution of a general multifield problem and to organize the data transfer of the relevant field quantities, an elaborate communication concept is inevitable. In *comana*, we decided on the master/slave model sketched in Figure 5.1. The master process steers the solution process, and it serves to organize the data transfer of the field quantities between the subproblem solvers $\mathcal{S}_1, \ldots, \mathcal{S}_m$. Basically, it executes a coupling procedure similar to the one outlined in Algorithm 1, possibly tailored to the particular problem under consideration. The slave processes are responsible for the solution of the subproblems. If a subproblem solver is distributed-memory parallelized and the solution of the corresponding subproblem is

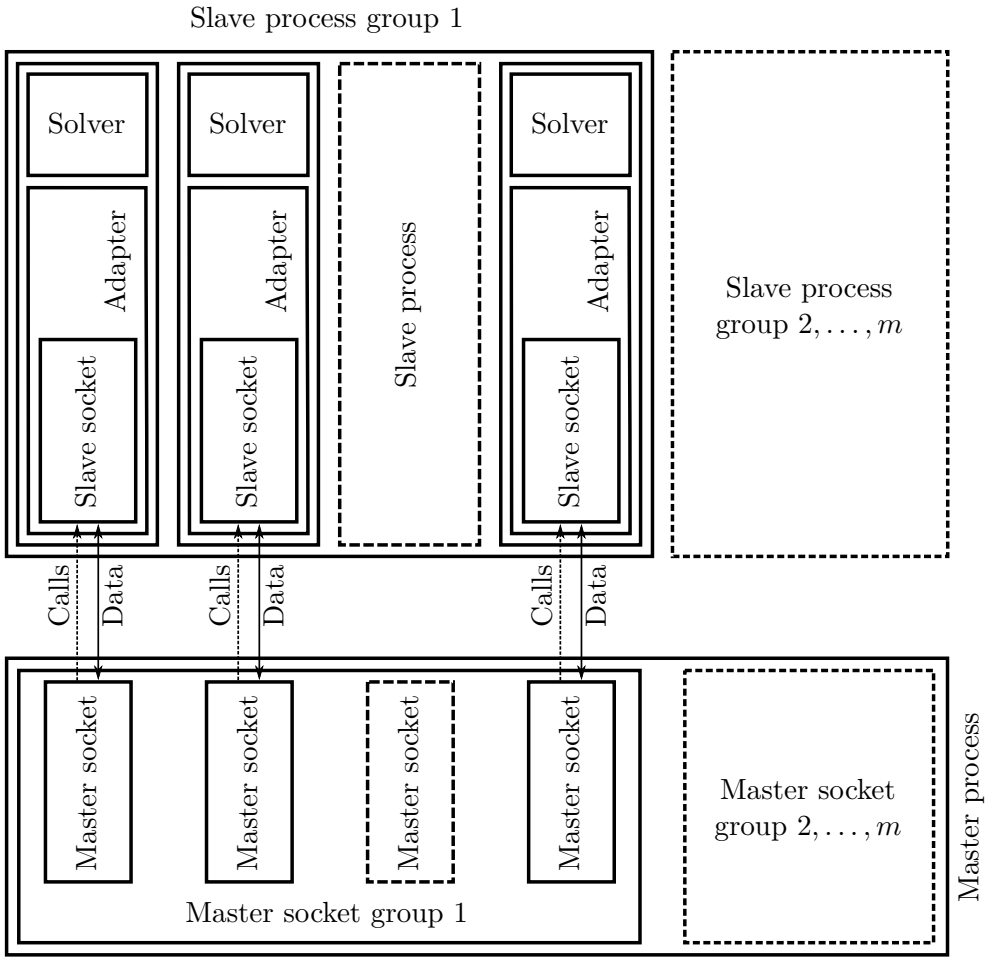


Figure 5.1: Master/slave communication concept.

1: $j := 0$
2: **while** $t_j \leq T$ **do**
3: Solve $\mathbf{g}_i(t_j, \mathbf{u}_1, \ldots, \mathbf{u}_i, \ldots, \mathbf{u}_m) = \mathbf{0}$
4: $j := j + 1$
5: $t_{j+1} := t_j + \Delta t_j$
6: **end while**

Algorithm 13: Generic solver $\mathcal{S}_i$.

1: Initialize driver
2: Initialize communication
3: $j := 0$
4: **while** $t_j \leq T$ **do**
5: $k := 0$
6: **while true do**
7: Data transfer
8: Solve $\mathbf{g}_i(t_j, \mathbf{u}_1, \ldots, \mathbf{u}_i, \ldots, \mathbf{u}_m) = \mathbf{0}$
9: Data transfer
10: **if** converged **then**
11: **break**
12: **end if**
13: $k := k + 1$
14: **end while**
15: $j := j + 1$
16: $t_{j+1} := t_j + \Delta t_j$
17: **end while**
18: Exit communication
19: Clear driver

Algorithm 14: Modified generic solver $\mathcal{S}_i$.

accomplished by several solver processes, this terminology must be generalized further. Then, the individual slave processes, which in this case are responsible for solving only part of the subproblem, constitute a slave process group. If a solver is single-threaded or shared-memory parallelized, the slave process group consequently contains just a single slave process. In the master process, a master socket group bundles several master sockets into a single data structure. From an algorithmic point of view, each master socket group and its corresponding slave process group represent a subproblem solver, independent of whether one or multiple solver instances are actually used to solve the subproblem. Later on, it will be shown that the concept of process groups substantially simplifies the implementation of a coupling algorithm. Each of the master sockets can be seen as a communication peer for a slave socket for sending and receiving data. The slave socket is part of an adapter, which prepares a solver for participation in a partitioned analysis and that allows to access and modify the solver's database during the simulation. In addition, some modifications in the solver's source code are required to enable multiple implicit iterations within a time increment and to facilitate the data transfer between the solver and the adapter as well as, subsequently, between the adapter and the master process.

To illustrate the necessary changes, let us consider the generic solver $\mathcal{S}_i$ for the solution of the ith subproblem sketched in Algorithm 13. Practically all transient solvers should fit the depicted scheme. For steady-state solvers, time can be interpreted as pseudo-time or as load steps. To begin with, the time increment counter j is initialized to zero, and the solver enters the time stepping procedure. In each loop cycle, the discrete field equations $\mathbf{g}_i(t_j, \mathbf{u}_1, \ldots, \mathbf{u}_i, \ldots, \mathbf{u}_m)$ are solved, j is incremented by one, and the time is increased by the time increment Δt_j. If time adaptivity is supported by the solver, Δt_j may also be

different between time steps. The procedure continues until the final time T is reached.

If a solver is to be integrated into a partitioned solution strategy, the modifications outlined in Algorithm 14 are required. In addition, the solver must offer the possibility to access and modify its database during the solution process. To this end, so-called *drivers* are introduced. In order to retain state during the simulation, the driver's data structures must be initialized to begin with (line 1). After that, the connection to the master process is established (line 2). In an implicit partitioned solution strategy, the relevant field quantities are, within a time increment, iteratively exchanged between the solvers until all subfields are equilibrated with each other. Consequently, the solution process of a single-field solver $\mathcal{S}_i$ must be embedded into an implicit iteration such that the set of discrete field equations $\mathbf{g}_i$ can be solved several times within the same time increment (lines 6 and 14). Furthermore, the program flow before and after the solution process needs to be modified (lines 7 and 9–12). Before solving the subfield equations $\mathbf{g}_i$, the relevant field quantities are received from the master process and applied as boundary conditions (line 7). After changing the boundary conditions, the subfield solution yields an updated set of discrete result variables, which need to be passed over to the other solvers involved in the partitioned solution process. In the case of surface-coupled problems, these are the solvers operating on domains sharing at least a part of the boundary Γ_i. For volume-coupled problems, the updated solution must be supplied to all solvers for subproblems on domains that partly or entirely overlap the domain Ω_i. This data transfer is carried out in line 9. Specifically, the solution is first sent to the master process, interpolated to the other solvers' discretizations (if required), and then forwarded to those solvers. Following the second data transfer, the master process informs the solver about the convergence status, indicating whether the subfields have already been equilibrated with each other up to a user-defined tolerance. If this is the case, the implicit iteration stops, and the solver continues with the next time increment (line 10–12). Otherwise, the iteration counter is incremented (line 13), and a further implicit iteration is carried out. Finally, the connection to the master process is terminated (line 18) and, if required, the resources acquired for the driver's data structures are released (line 19).

From the comparison of Algorithm 13 and 14, it becomes apparent that the modifications required to prepare a single-field solver for the integration into a partitioned solution procedure are minimally invasive. In fact, external functions are only called to establish the communication to the master process before entering the solution process (line 2), to transfer the relevant field quantities before and after the solution of the discrete field equations (lines 7 and 9), and eventually to finalize the communication to the master process (line 18). Since these functions interrupt the regular program flow of the solver, they will be referred to as *interrupt functions* in the following. In addition, the functions to initialize and clear the driver, which is used to access the solver's database, must be called (line 1 and 19). Clearly, any logic behind these functions should be hidden from the solver. Therefore, the implementation details are shifted into an adapter library as depicted in Figure 5.2. It furnishes a solver modified according to Algorithm 14 with all the necessary functionalities to participate in a partitioned solution process. In Algorithm 14, the first function called by the solver initializes the driver's global data structure (line 1). In contrast to the rest of the adapter library, the driver is implemented in the same programming language as the solver to avoid passing specific solver data structures across language barriers. To make these data structures accessible to the driver functions during the solution process, the driver stores references to them internally. To this end, a global data

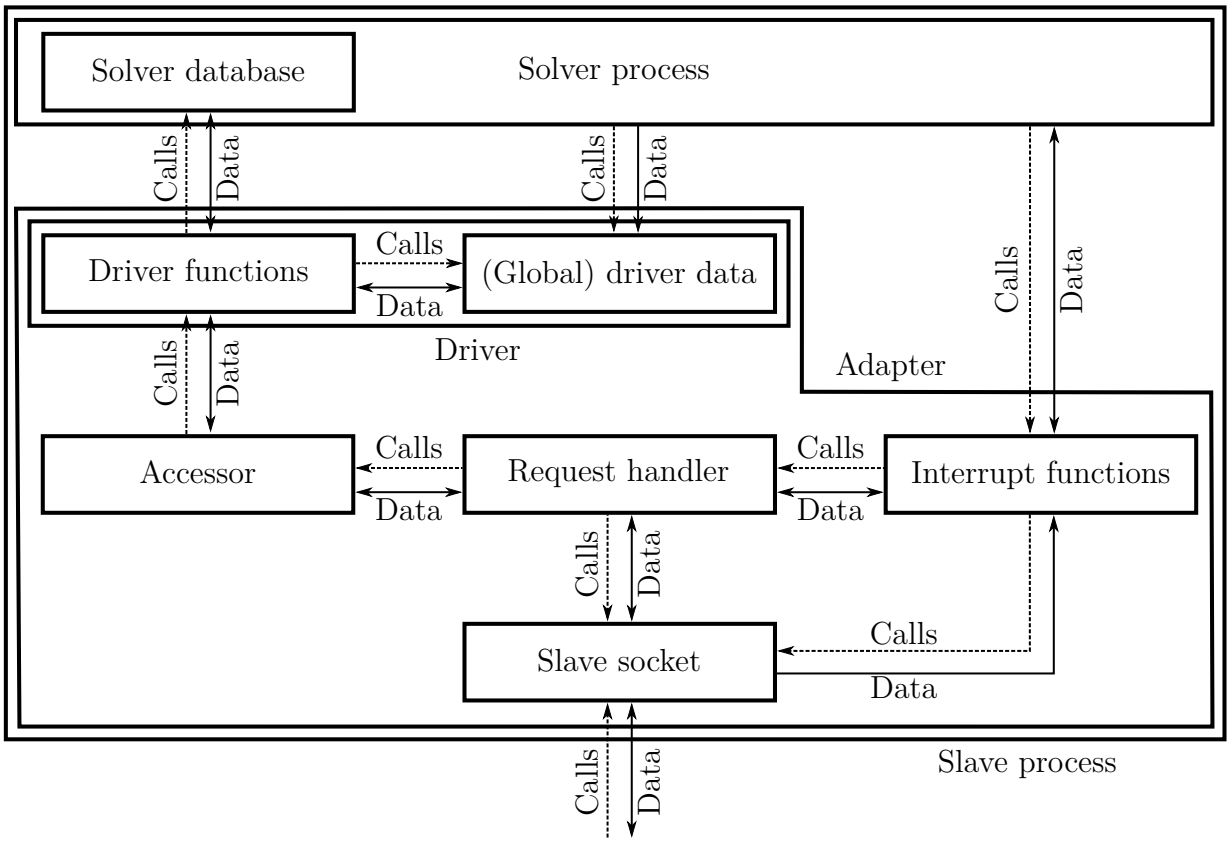


Figure 5.2: Adapter library enhancing a solver for the integration into a partitioned solution strategy.

structure must be introduced to retain state between subsequent function calls, which can involve a language barrier between C++ and the language the solver is implemented in. In line 1, this global data structure is initialized. Further explanations regarding the drivers are shifted to Section 5.6. Subsequent to initializing the driver data structures, the communication to the master process is initialized (line 2). Once the connection is established, the solver proceeds to the first data transfer (line 7). Most commonly, solvers receive and apply updated boundary conditions from the master process at this stage. However, it is also possible that different actions are required – which is why the generic concept of requests was introduced. In the data transfer phase, the solver polls for requests sent to it by the master process before carrying out different actions depending on the particular type of request. In that sense, the solver can be seen as being remote-controlled by the master process. A request consists of a request tag and request data. The request tag uniquely defines a request, the set of associated actions, and the request data to be sent to or received from the master process. Subsequent to receiving a request tag via the slave socket, the appropriate request handler responsible for processing the request is invoked. In the request handler, the accessor is addressed to access or modify the solver's database. Instead of accessing or modifying the solver data directly, however, a driver function is internally called. In essence, the set of driver functions represents a concise interface to the solver's database. The data from the solver's database, if any, is then passed back to the accessor and, subsequently, to the request handler. If data is to be sent to the master process, the data is forwarded to the slave socket. In turn, if data is to be received from the master process, the data is received by the slave socket and then passed over to the request handler. In the second data transfer (line 9), the same procedure applies – except that, usually, different requests than in first data transfer are received from the master process. In most cases, the master process will request the updated field solution at this point. At the end of the solution process, the slave socket terminates the connection to the master process (line 18). Finally, the driver data structures need to be cleaned up (line 19). If, for instance, heap memory was allocated upon driver initialization, these resources must again be released at the end of the simulation to prevent a memory leak.

Adapter libraries and modified source code files for various open source and commercial solvers for fluid dynamics, structural mechanics and thermodynamics implemented in C/C++, Fortran, Python, MATLAB/Octave, or APDL are readily available in *comana*. An overview of the supported solvers is given in Table 5.1. If an adapter library is not yet available for a solver, it can be implemented fairly easily using the vast range of generic data structures provided for this task in *comana*; the reader is referred to the following sections for further information regarding this topic.

Solvers to be integrated into the partitioned solution procedure are usually based on numerical methods that employ a discrete approximation of a continuous geometry. In *comana*, parts of the discretization associated to certain parts of the original geometry that are necessary to apply, for example, boundary conditions are referred to as *patches*. In the master/slave communication model, however, a slave process, if part of a distributed-memory parallelized process group, may only be responsible for a part of that patch, which leads to the concept of process patches. A process patch denotes the part of a patch occupied by a slave process, see also Figure 5.3 for an illustrative example.

In the partitioned solution procedure, the relevant field quantities need to exchanged at the coupling interface of a subdomain. In the general case, the coupling interface comprises several patches, which are termed a *patch group*. Therefore, a patch group is the relevant

Table 5.1: Adapter libraries currently available in *comana*.

Solver	Availability	Applications	Discretization	Language
AdhoC [41, 40]	In-house	Structural mechanics	FEM	C
ANSYS [3]	Commercial	Structural mechanics, thermodynamics	FEM	APDL/Fortran
Code_Aster [43]	Open source	Structural mechanics, thermodynamics	FEM	Fortran
foam-extend [61]	Open source	Fluid dynamics	FVM	C++
MSC Marc [117]	Commercial	Structural mechanics	FEM	Fortran
oct6dof	In-house	Rigid body dynamics	–	MATLAB/Octave
OpenFOAM [123]	Open source	Fluid mechanics	FVM	C++
*pan*MARE [12]	In-house	Fluid mechanics	BEM	Python

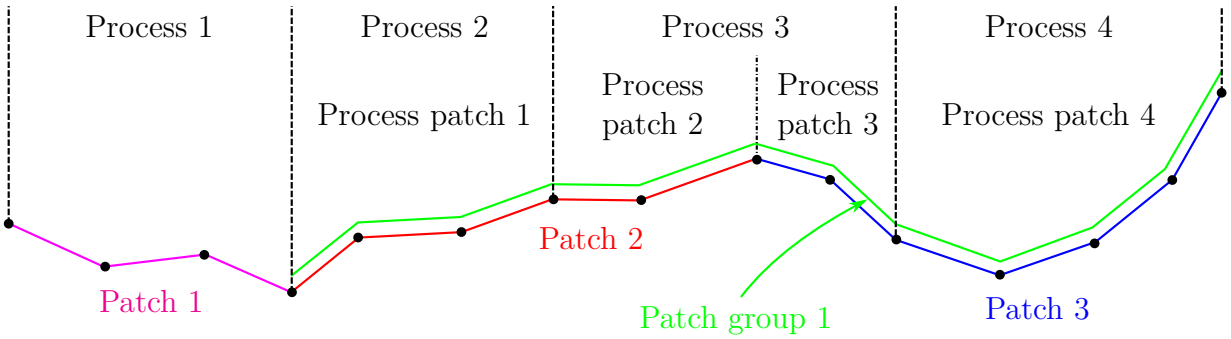


Figure 5.3: Patches, patch groups, and process patches.

entity in the master process. In contrast, the process patches are the entities a slave process operates on. In *comana*, a patch group stores a reference to the master socket group it was associated to. Field quantities to be applied as boundary conditions on a patch group are scattered to the master sockets in the master socket group, from where they are sent to the slave sockets. In a slave process, the received field quantities are then forwarded to the request handler, the accessor, and finally the driver, which modifies the solver's database. In the other direction, a master socket group gathers the field quantities from the individual process patches by going through the patch identifiers and requesting the field quantities associated to the process patches from each of the slave processes. By interacting only with the patch groups and master socket groups, the user can conveniently implement a coupling strategy irrespective of the particular solvers and irrespective of the number of distributed processes used for the subfield solution.

In order to request certain field quantities from a slave, the slave needs to be informed about the type of the quantity and its location, where the discrete quantity is available. In an FSI analysis, the type of the quantity could be *displacement* or *traction*, for instance. The location of these quantities could either be *vertex* in the case of displacements on a finite element discretization, or *cell* if tractions at the face centers of a finite volume discretization or at the collocation points of a boundary element mesh are considered. In addition to these standard point locations, there is also the concept of custom points, which serves to evaluate a field quantity at a set of user-defined points. In an FSI analysis, one would typically request the displacement at the vertices of the fluid discretization from the structural solver. In the opposite direction, it is convenient to define the structural integration points as evaluation points for the traction in the fluid solver. If custom points are used, the slave employs the projection procedure outlined in Section 4.5.5 and 4.5.6 to associate the custom points to the discretization in order to interpolate the field quantities from their standard locations to the custom points. Due to the fact that these mesh-dependent interpolation schemes need access to a solver's specific mesh data structure, the interpolation is performed in the adapter library instead of passing the information required for the interpolation to the master process, as in the case of mesh-independent interpolation schemes. In most situations, exchanging the field quantities at custom points is the preferred way of organizing the data transfer, as mesh-based interpolation schemes usually provide the highest interpolation accuracy.

5.2 Generic Data Structures

Our software framework *comana* was purposefully implemented in the C++ programming language. C++ is a general-purpose and feature-rich programming language, which is, among other applications, particularly well suited for the implementation of numerical software. As opposed to older languages like C or Fortran, it provides a lot of programming concepts that do not only make programming in C++ more convenient, but also faster and safer. At the same time, the execution speed of programs written in C++ is still comparable to codes implemented in C or Fortran. C++ is a multiparadigm language, i.e., in addition to functional and object-oriented paradigms, it also supports generic and metaprogramming. It also provides the standard library, which offers several generic container types, algorithms to manipulate or operate on these containers, function objects, strings and streams, or mathematical functions. Many common programming tasks can

be accomplished using the standard library and its well-tested and efficient classes and functions. In the course of the introduction of the recent C++ standards C++11 [80], C++14 [81], and the upcoming C++17 [82] standard, a vast range of interesting features have been added to the language. Bjarne Stroustrup, the inventor of the C++ programming language, remarks that "C++11 feels like a new language" [154]. In what follows, the most essential added features are therefore briefly summarized:

- *Move semantics [112, pp. 157–214]:* In older C++ standards, the only means to transfer state from one object to another was a deep and potentially costly copy operation. In many situations, however, the copied-from value is no longer used – hence making the copy superfluous as the temporary could be modified directly instead. This led to the concept of rvalue references. Consider, for example, a `std::vector<Type>` that is returned from a function. In the absence of return value optimization (RVO), assigning the result of this function to a variable involved a deep copy of the `std::vector<Type>` until C++03 [79]. In C++11, this copy is avoided due to the fact that `std::vector<Type>` defines a move constructor that accepts an rvalue reference and creates a new `std::vector<Type>` instance by "reusing" the data members from the moved-from vector.

- *Automatic type deduction [112, pp. 37–48]:* In C++03 and former C++ standards, it was necessary to specify the type of a variable explicitly. In C++11, this necessity was mitigated by introducing the `auto` keyword, which allows to infer the type from the right-hand side of an assignment. For instance,

  ```
  auto number = 3;
  ```

  deduces the type of `number` to `int` since the type of the literal `3` is `int`. The `auto` keyword makes production code safer, easier to read and to maintain. In order to underline the usefulness of the `auto` keyword, Herb Sutter coined the acronym AAA ("almost always `auto`") [112, p. XIII]. A further mechanism for automatic type deduction is provided by the keyword `decltype`, which enables to infer the type from another variable at compile time:

  ```
  int number1;
  decltype(number1) number2;
  ```

  Here, `number2` will be of type `int` – just as `number1`.

- *Generalized constant expressions [112, pp. 97–103]:* Before C++11, a constant expression must not involve a function call or an object constructor. Using the newly introduced `constexpr` keyword, an expression can now be designated a compile-time constant. For instance, the function

  ```
  constexpr int factorial(int n) {
      return n <= 1 ? 1 : (n * factorial(n - 1));
  }
  ```

  evaluates the factorial of `n` at compile-time [126, p. 141].

- *Initializer lists [112, pp. 52–58]:* Initializer lists have been available in C++ ever since, but only for plain old data (POD) types such as arrays or structs. C++11 generalized the concept of initializer lists and introduced the template `std::initial`

izer_list. This way, constructors or other functions may accept initializer lists as arguments:

```
struct Foo {
    Foo(std::initializer_list<int> numbers);
};
```

An instance of the class Foo can then be constructed from a list of integers as follows:

```
Foo foo{ 1, 2, 6, 8 };
```

- *Uniform initialization [112, pp. 50–51]:* C++11 extends the initializer list syntax to a fully uniform type initialization applicable to any kind of objects. Consider the structs

```
struct Pair1 {
    int a, b;
};

struct Pair2 {
    Pair2(int a, int b)
private:
    int a, b;
};
```

which can, in C++11, both be initialized in the same manner:

```
Pair1 pair1{1, 2};
Pair2 pair2{3, 4};
```

Uniform initialization removes any distinction between `()`, `{}`, and initialization with no braces by means of the default constructor. This syntax also helps to avoid repeating the typename in function arguments or returns. Moreover, it solves the most vexing parse (MVP) problem.

- *Lambda expressions [112, pp. 215–240]:* C++11 also introduced the concept of anonymous or lambda functions. For example, the lambda function

```
[](int x, int y) -> int { return x + y; }
```

adds the integers `x` and `y`. The return type is implicitly deduced from the type of the returned expression (in this case: `decltype(x + y)`).

- *Range-based for loops [112, p. 41]:* Another useful feature in recent C++ standards is the availability of range-based for loops. For C-style arrays, initializer lists, and any data types that define the functions `begin()` and `end()`, iterating through these data structures is as simple as

```
for (auto &x : someArray)
    x += 5;
```

Here, each element in `someArray` is incremented by 5.

- *Variadic templates [153, p. 66 sq.]:* C++11 introduces templates that can take a variable number of arguments. This allows the implementation of type-safe variadic

functions, as will also be shown in the remainder of this chapter. To illustrate the concept, let us consider a C++-style `print` function that accepts a variable number of arguments of different type:

```
template<class Type>
void print(const Type &message) {
    std::cout << message << " ";
}

template<class Head, class ...Tail>
void print(Head head, Tail ...tail) {
    print(head);
    print(tail...);
}
```

Note that the first `print` function is primarily required to end the recursion once only one argument is left in the argument list.

- *Strongly typed enumerations [112, pp. 67–74]:* In former C++ standards, enumerations are not type-safe. Essentially, they directly translate to integers, which allows to substitute an integer wherever an enumeration type is actually expected. C++11 introduces strongly typed enumerations expressed by the `enum class` keyword to be used in situations where an automatic conversion to an integer is not desired.

- *Alias declarations [112, pp. 63–67]:* C++11 introduces the `using` keyword to declare a synonym for a previously declared type. In this context, it essentially replaces the `typedef` keyword. In addition, an alias can also be defined for a template, for example:

  ```
  template<class Type, class Allocator = std::allocator<Type>>
  using Vector = std::vector<Type, Allocator>;
  ```

  `Vector` is termed an *alias template*.

- *General-purpose smart pointers [112, pp. 117–156]:* Recent C++ standards provide the smart pointer types `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`. These types simulate a pointer but add additional features that make them safer to use than raw pointers. For instance, a `std::unique_ptr` releases its memory automatically if it goes out of scope, without the need to `delete` as in the case of raw pointers. Smart pointers thus avoid memory leaks and bugs that are difficult to find, but at the same time essentially retain the efficiency of raw pointers.

The above list is by no means complete, but sets the outline for the discussion of the most important data structures in *comana*. For further details regarding the recent C++ language features, the interested reader is referred to the standard textbooks [155, 153, 112, 89].

Practically all classes in *comana* are based on a set of generic data structures. One of the most important programming concept are type traits, which serve to retrieve type information at compile-time. Type traits are used excessively throughout *comana*, contributing significantly to a generic and reusable code.

For the data transfer of the relevant field quantities between the subproblem solvers and the master process, and for the implementation of extensible algorithmic building blocks, it is convenient to introduce custom container classes that are not included in the C++ standard library. In particular, we will discuss enhanced versions of the matrix classes provided in the Eigen linear algebra library as well as a class for the storage of physical quantities.

5.2.1 Traits

Type traits provide a powerful mechanism in template metaprogramming. By means of type traits, it becomes possible to provide information about types at compile-time – one of *the* cornerstones of reusable, generic, and type-safe software. Recent C++ standards and C++11 in particular have added a lot of type traits to the standard library. Hence, it suffices to provide only a small number of additional traits for the use in *comana*.

Exemplary, let us consider the trait `IsIterator`, which provides the functionality to test whether `Type` is an iterator or not:

```
template<class Type>
struct IsIterator {
private:
    template<class Iterator>
    static constexpr std::true_type test(typename
        std::iterator_traits<Iterator>::iterator_category *);

    template<class Iterator>
    static constexpr std::false_type test(...);

public:
    static constexpr bool value = decltype(test<Type>(nullptr))::value;
};

template<class Type>
constexpr bool IsIterator<Type>::value;
```

If the compiler detects a specialization of `std::iterator_traits` for `Type`, the first `test` function is instantiated and `value` becomes `true`. Otherwise, the second `test` function is instantiated and `value` becomes `false`. Note the additional definition of the `constexpr` member `value` is necessary as `value` is ODR-used [80, sec. 9.4.2]. Another useful trait is the `IsIteratorCategory` trait

```
template<class Iterator, typename IteratorTag>
using IsIteratorCategory = std::is_same<
    typename std::iterator_traits<Iterator>::iterator_category, IteratorTag>;
```

to match an `IteratorTag` against the `iterator_category` field in the `std::iterator_traits` template and identify a certain class of iterators. Based on this, iterators providing random access to a range of values are, for instance, identified by the `IsRandomAccessIterator` trait

```
template<class Iterator, class Enable = void>
struct IsRandomAccessIterator: public std::false_type {
};

template<class Iterator>
struct IsRandomAccessIterator<Iterator, typename std::enable_if<
    IsIterator<Iterator>::value && IsIteratorCategory<Iterator,
    std::random_access_iterator_tag>::value>::type> : public std::true_type {
};
```

Here, the idiom *substitution failure is not an error (SFINAE)* has been used to conditionally enable the specialization of the `IsRandomAccessIterator` struct and to initialize its `value` property to `true` by inheriting `std::true_type`. As a last iterator trait, we present the `RequireInputIterator` trait, which is useful to selectively enable a template if the given template argument represents an input iterator[1]:

```
template<class InputIterator>
using RequireInputIterator = typename std::enable_if<std::is_convertible<
    typename std::iterator_traits<InputIterator>::iterator_category,
    std::input_iterator_tag>::value>::type;
```

Similar traits are provided to check whether a `Type` provides certain type aliases, which are present in the container types in the C++ standard library. For example, consider the `HasValueType` trait, which sets its member `value` to `true` if `Type` declares the typedef `value_type`:

```
template<class Type>
struct HasValueType {
private:
    template<class Container>
    static constexpr std::true_type test(typename Container::value_type *);

    template<class Container>
    static constexpr std::false_type test(...);

public:
    static constexpr bool value = decltype(test<Type>(nullptr))::value;
};
```

Further traits of the kind `Has`*`Member`*`<Type>` are available. Based on these traits, one is able to implement

```
template<class Type>
struct IsContainer: public std::integral_constant<bool,
    HasValueType<Type>::value &&
    HasPointer<Type>::value && HasConstPointer<Type>::value &&
    HasReference<Type>::value && HasConstReference<Type>::value &&
    HasIterator<Type>::value && HasConstIterator<Type>::value &&
```

[1] In fact, this trait is a minor modification of the `std::_RequireInputIter` trait distributed along with the GCC 5.4.0 [54].

```
    HasSizeType<Type>::value &&
    HasDifferenceType<Type>::value> {
};
```

to check whether `Type` is a container type. Several other traits are provided to check whether a container's underlying memory is contiguous, whether a container is resizable, or if it is a nested container (that is, a container of containers).

As already mentioned, all these traits serve to conditionally enable and instantiate the correct template for a certain type. Several applications of this concept will be shown in the forthcoming sections.

5.2.2 Containers

The C++ standard library provides a broad range of different container types that – as they are generic, robust, and fast – also prove useful in numerical software. For the use in *comana*, we define type aliases such as

```
template<class Type, std::size_t sizeAtCompileTime>
using Array = std::array<Type, sizeAtCompileTime>;

template<std::size_t sizeAtCompileTime>
using DoubleArray = Array<double, sizeAtCompileTime>;

template<class Type, class Allocator = std::allocator<Type>>
using DynamicArray = std::vector<Type, Allocator>;

using DoubleDynamicArray = DynamicArray<double>;
```

to ease typing for often-used container types and to increase the readability of the source code. Yet, the standard library lacks container classes providing mathematical operations by means of operator overloading. Instead, numerical operations on containers have been shifted to separate free functions for the sake of genericity. These functions operate on iterators instead of containers. Thus, the implementations are independent of specific container types and may also be used for custom container types. However, as soon as a lot of mathematical operations need to be performed, the readability decreases and the maintainability deteriorates. In *comana*, standard container types are therefore only used to store objects, plain numbers not intended for the use in mathematical expressions, and other containers, but do not represent mathematical entities. For linear algebra, we resort to the high-level header-only Eigen library [55], which provides a vast range of different dense and sparse matrix types, as well as plain array and tensor types. Eigen makes heavy use of expression templates and generates expression trees at compile-time, surrounded by custom code for their evaluation. Depending on the platform and compiler settings, the Eigen library unrolls loops at compile-time and performs automatic vectorizations. Because of their efficiency and easy use, *comana* relies on Eigen containers wherever numerical operations have to be performed on containers. To increase the compatibility of the Eigen containers with the standard library, numerous enhancements have been added by means of Eigen plugins and other utility functions and traits. For instance, let us consider an excerpt from the changes injected in the `Eigen::Matrix` class by defining the `EIGEN_MATRIX_PLUGIN` macro:

```
// In <COMANA_ROOT>/comana/kernel/container/matrix.h:
#define EIGEN_MATRIX_PLUGIN "comana/kernel/container/eigen_matrix_plugin.def"

// In <COMANA_ROOT>/comana/kernel/container/eigen_matrix_plugin.def:
template<class InputIterator, class = RequireInputIterator<InputIterator>>
EIGEN_STRONG_INLINE Matrix(InputIterator begin, InputIterator end) : Base() {
    Base::_check_template_params();
    const auto numberOfElements = std::distance(begin, end);

    if (base().size() == 0)
        this->resize(numberOfElements, 1);

    std::size_t index = 0;
    for (auto iterator = begin; iterator != end; ++iterator, ++index)
        coeffRef(index) = *iterator;
}

EIGEN_STRONG_INLINE Matrix(const InitializerList<Scalar> initializerList)
    : Matrix(std::begin(initializerList), std::end(initializerList)) {
}

EIGEN_STRONG_INLINE Matrix(
    const InitializerList<InitializerList<Scalar>> initializerList) : Base() {
    // Initialize from nested initializer list; if the storage order is
    // row-major, the matrix is initialized row-wise, while if the storage
    // order is column-major, the matrix is initialized column-wise
}

EIGEN_STRONG_INLINE const Scalar * begin() const noexcept {
    return this->data();
}

EIGEN_STRONG_INLINE const Scalar * end() const noexcept {
    return this->data() + this->size();
}
```

Enhancing the `Eigen::Matrix` class with these constructors, it becomes possible to construct an `Eigen::Matrix` from an iterator range, an initializer list, or a nested initializer list in the same manner as a standard container. Adding the functions `begin` and `end` furnishes the `Eigen::Matrix` class with iterator access and, moreover, allows to use the free functions `std::begin` and `std::end` on `Eigen::Matrix` objects. Note that the non-`const` versions of `begin` and `end` are implemented completely analogously. Similar enhancements also exist for other Eigen types.

One of the core classes in *comana* is the `Field` class for the storage of physical quantities:

```
class Field {
public:
    using value_type = QuantityView;
    using ComponentVector = DynamicDoubleVector;
    // Other type aliases, omitted here
```

```
    Field() noexcept;
    explicit Field(const QuantityType quantityType,
        const std::size_t size = 0) noexcept;
    explicit Field(const std::size_t quantitySize,
        const std::size_t size = 0) noexcept;
    Field(const QuantityType quantityType, const ComponentVector &components);
    Field(const std::size_t quantitySize, const ComponentVector &components);
    Field(const QuantityType quantityType, const std::size_t size,
        const QuantityView::Scalar value) noexcept;
    Field(const std::size_t quantitySize, const std::size_t size,
        const QuantityView::Scalar value) noexcept;
    template<class Quantity, class = RequireQuantity<Quantity>>
    Field(const Quantity &quantity, const std::size_t size) noexcept;
    template<class Quantity, class = RequireQuantity<Quantity>>
    Field(const InitializerList<Quantity> initializerList);
    template<class InputIterator, class = RequireInputIterator<InputIterator>>
    Field(InputIterator first, InputIterator last);
    Field(const Field &field) noexcept;
    Field(Field &&field) noexcept;
    Field & operator=(const Field &field) noexcept;
    Field & operator=(Field &&field) noexcept;
    ~Field() noexcept = default;
    // Iterator access, omitted here
    // Requesting and changing capacity, omitted here
    // Element and component access, omitted here
private:
    using QuantityViewArray = DynamicArray<value_type>;
    std::size_t quantitySize_;
    ComponentVector components_;
    QuantityViewArray quantityViews_;
    static QuantityViewArray createQuantityViewArray(
        const std::size_t quantitySize, ComponentVector &components);
};
```

In *comana*, a quantity is basically an array of values equipped with mathematical operations such as addition, subtraction, multiplication, or division. If a certain quantity type is requested from a solver, it is convenient to store the quantities in an array-like structure, which, however, provides some additional functionality for memory management and data access. In a `Field`, the quantities' real-valued components are stored in a contiguous `DynamicDoubleVector`. On top of this, the `Field` holds a `QuantityViewArray` member `quantityViews_` gathering so-called *views* to the quantities' components. Internally, views essentially only store the memory address to their components, whereas, seen from outside, they behave just as a regular quantity. That way, the components of a `Field` can be manipulated either directly through the component vector or indirectly through the quantity views. In the implementation of the partitioned solution procedure, this data representation offers several advantages. On the one hand, the user can request a `Field` from a slave – and operate on quantity views and, hence, on a data representation that is close to the physical interpretation. For instance, in a `field` of type `Field`, accessing `field[1]` returns the second quantity in the field, which refers to the field components

3 to 5 (note that C++ uses zero-based indexing). This avoids the need for tedious and erroneous indexing in the component vector. On the other hand, the direct access to the underlying components turns out beneficial in the application of the predictor or convergence acceleration methods that manipulate a plain numeric vector instead of a nested one. Moreover, the implementation of these classes then becomes much easier, and the required mathematical operations execute significantly faster if the underlying memory is contiguous. Also, the low-level data transfer functions used in communication between the master and the slave process expect the data to be organized in contiguous memory.

Several constructors for the `Field` class are provided; it can be default-constructed, constructed from quantity type and field size, or from the size of a single quantity and the entire field size. Instead of providing a field size, the field components can also be provided directly. Further constructors that accept initializer lists or ranges of quantities are implemented as well. Moreover, it offers iterator access, functions to request and change its capacity, and members for element and component access. Hence, the `Field` class allows to easily access and manipulate physical quantities, for simple and convenient indexing, and for a nested or flat data representation of its components, each of which prove to be expedient, depending on the context the `Field` class is used in.

5.3 Communication Data Structures

For the data transfer between the master process and the slave processes, the Message Passing Interface (MPI) is employed, which provides an Application Programming Interface (API) to exchange messages across a network in a standardized and portable manner. Due to the fact that the MPI C++ bindings are deprecated since the MPI 2.2 release, *comana* provides its own C++ interface to the MPI functionality, which essentially serves three important purposes. First of all, the MPI functions become much easier to use in a C++ application if a dedicated C++ interface is provided. Secondly, the MPI C bindings are not `const`-correct, which leads to the necessity of frequent `const_cast`s, cluttering the production code and deteriorating its maintainability. Lastly, C++ templates can be used to increase type-safety, and they remedy the need to individually specify the address of the data to be sent or received, its size, and its type. For instance, let us consider the function

```
int MPI_Send(void *buffer, int count, MPI_Datatype type, /* ... */)
```

to send a `buffer` of length `count` and type `type`. If we intend to send a `std::vector<int>` `v`, for example, we would have to call

```
MPI_Send(const_cast<int *>(v.data()), v.size(), MPI_INT, /* ... */)
```

As passing the latter two arguments separately is potentially error-prone, this is best avoided by introducing a function template

```
template<class Type>
void send(/* ... */, const Type &data) noexcept {
    Sender<Type>::send(/* ... */, data);
}
```

Here, `Sender<Type>` is a class template specialized for all data types that need to be exchanged across the network. SFINAE can again be leveraged to conditionally enable

a particular specialization depending on the template argument. For `std::vector<int>`, the specialization

```
template<class Container>
struct Sender<Container, typename
        std::enable_if<IsContiguousContainer<Container>::value>::type> {
    static void send(/* ... */, const Container &container) noexcept {
        sendRange(/* ... */, std::begin(container), std::end(container));
    }
};
```

is instantiated. A second version of the `send` functions allows to send only part of a vector by providing an iterator range:

```
template<class InputIterator, class = RequireInputIterator<InputIterator>>
void send(const Socket &socket, InputIterator first, InputIterator last)
        noexcept {
    Sender<InputIterator>::send(socket, first, last);
}
```

In this case, the specialization

```
template<class Iterator>
struct Sender<Iterator,
        typename std::enable_if<
                IsArithmeticRandomAccessIterator<Iterator>::value>::type> {
    static void send(const Socket &socket, Iterator first, Iterator last)
            noexcept {
        // random_access_iterators can be safely dereferenced and converted to
        // an ordinary pointer
        sendRange(socket, convertToPointer(first), convertToPointer(last));
    }
};
```

is instantiated. Internally, specializations of `Sender<Type>` dispatch their arguments to the function

```
template<class Type>
void sendRange(/* ... */, const Type *first, const Type *last) {
    MPI_Send(const_cast<Type *>(first), std::distance(first, last),
        DataType<Type>::type, /* ... */);
}
```

which finally invokes `MPI_Send`. `DataType<Type>` is a class template specialized for all primitive data types:

```
template<class Type>
struct DataType {
};

template<>
struct DataType<int> {
```

```
    static constexpr MPI_Datatype type = MPI_INT;
};

// Further specializations for all built-in data types
```

Note that the variants of the `send` function template still use `MPI_Send` in the end, but make it a lot easier and safer to use.

A look at the full declaration of `MPI_Send` reveals that, in addition to the data to be sent, its destination has to be specified as well:

```
int MPI_Send(const void *buffer, int count, MPI_Datatype type, int rank,
    int tag, MPI_Comm communicator)
```

Together, MPI `communicator` and `rank` uniquely identify the destination for a message. In essence, the `communicator` addresses a *group* of processes and the rank represents the local process number inside that process group. Evidently, it makes sense to merge these two entities into a single data structure. In *comana*, this data structure is termed a *socket*. It essentially offers the following interface:

```
class Socket {
public:
    virtual ~Socket() noexcept = 0;
    virtual const Mpi::Communicator & getCommunicator() const noexcept = 0;
    Mpi::Communicator & getCommunicator() noexcept;
    virtual std::size_t getPeerRank() const noexcept = 0;
};
```

Here, the `getCommunicator` method simply returns the `Mpi::Communicator` instance associated to the `Socket`, which is nothing but a thin wrapper around `MPI_Comm`. Not to be confused with the rank of the process in which the `Socket` is used, the function `getPeerRank` returns the rank of the process a `Socket` communicates to. From the `Socket` class, the classes `MasterSocket` and `SlaveSocket` are then derived. First, let us consider the `MasterSocket` class:

```
class MasterSocket: public Socket {
public:
    MasterSocket(const Mpi::Communicator &communicator,
        const std::size_t peerRank = 0) noexcept;
    const Mpi::Communicator & getCommunicator() const noexcept override;
    std::size_t getPeerRank() const noexcept override;
private:
    const Mpi::Communicator &communicator_;
    std::size_t peerRank_;
};
```

In its constructor, it expects an `Mpi::Communicator` and the `peerRank`, which are supplied by the parent `MasterSocketGroup`, which internally creates as many `MasterSockets` as there are slave processes in the slave process group it is associated to. Each `MasterSocket` instance is assigned the rank of the slave process it is responsible for. In doing so, a one-to-one correspondence between master sockets and slave sockets is accomplished,

which simplifies the communication substantially. In contrast to the `MasterSocket` constructor, the `SlaveSocket` constructor does not take any arguments, but initializes the `Mpi::Communicator` on its own:

```
class SlaveSocket: public Socket {
public:
    SlaveSocket() noexcept;
    const Mpi::Communicator & getCommunicator() const noexcept override;
    std::size_t getPeerRank() const noexcept override;
private:
    UniquePointer<Mpi::Communicator> communicator_;
};
```

Further, the constructor also does not require a peer rank, as a `SlaveSocket` instance always has rank 0 inside its communicator. Consequently, the `getPeerRank` method always returns 0. For the `communicator_` member, we use the *pointer to implementation (pImpl)* idiom to remove the compile-time dependency of the `SlaveSocket` class on the MPI headers.

Due to the fact that the `send` and `receive` functions presented above accept a `const Socket &` as their first argument, the same set of functions can be used for both `MasterSockets` and `SlaveSockets` alike. Based on the presented functions and classes, the following concise and type-safe interface to the original MPI functions can be supplied. For the task of sending data, there is a variant of the `send` function that accepts a single argument and one that accepts an iterator range:

```
template<class Type>
void send(const Socket &socket, const Type &data) noexcept;

template<class InputIterator, class = RequireInputIterator<Iterator>>
void send(const Socket &socket, InputIterator first,
    InputIterator last) noexcept;
```

Similarly, different variants of the `receive` function exist:

```
template<class Iterator>
typename std::enable_if<IsIterator<Iterator>::value, void>::type receive(
        const Socket &socket, Iterator result) noexcept;

template<class Type>
typename std::enable_if<!IsIterator<Type>::value, void>::type receive(
        const Socket &socket, Type &data) noexcept;

template<class Type>
Type receive(const Socket &socket) noexcept;
```

In the first version, the received values are inserted at the memory address pointed to by `result`. In the second variant, a single argument of type `Type &` to be modified in-place is expected. Lastly, the third version returns the received message by value.

5.4 Mesh Data Structures

Most numerical schemes for the solution of partial differential equations involve a computational mesh. For the application of the projection procedure outlined in Section 4.5.5 and 4.5.6, the user-defined custom points need to be distributed from the master process to the slave processes of a slave process group. In the following, we will therefore briefly discuss the strategy followed in this regard. Subsequent to receiving the custom points in a slave process, the mesh needs to be traversed in order to identify the nearest element to a given custom point and, after that, to interpolate the requested field quantities to this point. In order to reduce code duplication in the implementation of this procedure in the adapter library to a minimum, several generic mesh data structures are provided along with *comana*. Last but not least, generic classes and functions for performing the integration on a computational mesh are presented. These prove, for example, particularly useful for the load integration on finite element meshes in the case that a solver does not allow to specify a distributed load at the integration points directly.

5.4.1 Custom Point Scattering

In Section 4.5, it was outlined that mesh-based interpolation schemes offer substantial advantages over mesh-independent interpolation techniques. In *comana*, a mesh-based interpolation is performed in the adapter library of a *slave* process, so as to avoid the need to handle heterogeneous mesh data structures in the master process. This is opposed to the generic mesh-independent interpolation schemes, which need to be applied explicitly in every implicit iteration of the coupling algorithm executed by the *master* process.

If the user chooses a mesh-based interpolation scheme, the set of user-defined custom points, at which the interpolated field quantities are desired, needs to be associated to a patch group and supplied to the subfield solver. As already outlined in Section 5.1, a patch group may comprise multiple patches, each of which may be distributed across several processes. It is therefore required to also distribute the custom points such that each custom point is associated to the part of the mesh it is closest to. In *comana*, the strategy pursued in this regard is the following. First, the vertices of the process patches belonging to the patch group are requested from each slave process. For each of the process patches, we determine the bounding box enclosing its set of vertices. Based on Algorithm 9 implementing the AABB tree traversal, a nearest bounding box search is then performed to associate a set of bounding boxes $\mathcal{R}_i = \{B_{i,1}, \dots B_{i,N}\}$ to each custom point $\boldsymbol{q}_i$, which need to be considered as possible candidates to contain the part of the discretization closest to $\boldsymbol{q}_i$. For each custom point $\boldsymbol{q}_i$, we initialize a current best distance $d_i^\star$ to the maximum distance $d_{\max}(B_{i,1}, \boldsymbol{q}_i)$ between the custom point $\boldsymbol{q}_i$ and a point in its closest bounding box $B_{i,1}$. For each process patch bounding box B_l, the closest custom points are then gathered and sent to the associated slave process in batch to perform the projection procedure discussed in Section 4.5.5 for FE meshes or Section 4.5.6 for polygonal and polyhedral meshes. Next, the distances between the custom points and their projection to the lth process patch are passed over to the master process. Here, the current best distances $d_i^\star$ of the custom points last considered are updated, and bounding boxes $B_{i,j}$ with distance $d_{\min}(B_{i,j}, \boldsymbol{q}_i)$ exceeding $d_i^\star$ are deleted from the set of candidates $\mathcal{R}_i$. The procedure is repeated until no further possible candidates are left for any of the custom points $\boldsymbol{q}_i$, i.e, $\mathcal{R}_i = \emptyset \; \forall \, i = 1, \dots, m$.

5.4.2 Basis Functions, Cell Topology, and Cell Geometry

For the FEM, the concept of isoparametric elements and the use of shape functions for the interpolation of nodal quantities to the interior of the element were already discussed in Chapter 3. This concept is not restricted to the FEM, though, but can be deployed for any computational meshes that consist of elements exhibiting the same topology as the standard element types available in the FEM. In *comana*, the notion *cell* is used as an umbrella term for ordinary finite elements and elements for which the same interpolation can be applied, but which are not necessarily associated to an FE discretization. If the geometry and the field variables are not interpolated with the same set of shape functions, it is convenient to distinguish between the mapping functions for the interpolation of the geometry and the shape functions for the interpolation of the field variables. In the following, mapping and shape functions are subsumed under the term *basis functions*. Due to the fact that mapping and shape functions are not necessarily identical, it is convenient to separate the basis from a specific cell geometry and to factor it out into its own class:

```
template<class LocalSpace1, std::size_t numberOfFunctions1>
struct Basis final {
    using LocalSpace = LocalSpace1;

    static constexpr std::size_t localSpaceDimension { LocalSpace::dimension };

    using LocalPoint = typename LocalSpace::Point;

    static constexpr std::size_t numberOfFunctions { numberOfFunctions1 };

    using Functions = DoubleVector<numberOfFunctions>;

    static Functions evaluate(const LocalPoint &localPoint);

    using Derivatives = DoubleVector<numberOfFunctions>;

    using FirstOrderDerivatives = Array<Derivatives, localSpaceDimension>;

    static FirstOrderDerivatives evaluateFirstOrderDerivatives(
            const LocalPoint &localPoint);

    using SecondOrderDerivatives = Array<Derivatives,
        computeNumberOfSecondOrderDerivatives(localSpaceDimension)>;

    static SecondOrderDerivatives evaluateSecondOrderDerivatives(
        const LocalPoint &localPoint);

    template<class Quantities>
    static typename StorageType<typename Quantities::value_type>::type
        interpolate(const Quantities &quantities, const LocalPoint &localPoint);
};
```

Here, the local space represents the domain, on which the basis is defined. For quadrilateral domains, for instance, we have:

```
struct QuadrilateralLocalSpace final {
    static constexpr std::size_t localSpaceDimension { 2 };

    using Point = Comana::Point<dimension>;

    static bool isInside(const Point &point) noexcept;
};
```

In the context of the projection procedure, the function `isInside` becomes relevant to judge whether the local coordinates $\boldsymbol{\xi}$ of a projected global point are still inside the local space of the cell. In the above example, `isInside` returns `true` if $\xi \in [-1, 1] \wedge \eta \in [-1, 1]$. In the `Basis` class, the basis function vector $\boldsymbol{N}(\boldsymbol{\xi})$ and its first- or second-order derivatives $\partial \boldsymbol{N}/\partial\xi$, $\partial \boldsymbol{N}/\partial\eta$ or $\partial^2 \boldsymbol{N}/\partial\xi^2$, $\partial^2 \boldsymbol{N}/\partial\eta^2$, and $\partial^2 \boldsymbol{N}/\partial\xi\partial\eta$ are represented as `DoubleVectors` of fixed size and can be evaluated at a `LocalPoint` $\boldsymbol{\xi}$ of the parameter space using the evaluation methods, which are fully specialized for each particular basis[2]:

```
using Quadrilateral4Basis = Basis<QuadrilateralLocalSpace, 4>;

template<>
Quadrilateral4Basis::Functions Quadrilateral4Basis::evaluate(
        const LocalPoint &localPoint) {
    const auto r = localPoint[R], rm = 1 - r, rp = 1 + r, //
               s = localPoint[S], sm = 1 - s, sp = 1 + s;
    return { //
        0.25 * rm * sm, // (1)
        0.25 * rp * sm, // (2)
        0.25 * rp * sp, // (3)
        0.25 * rm * sp  // (4)
    };
}

template<>
Quadrilateral4Basis::FirstOrderDerivatives
        Quadrilateral4Basis::evaluateFirstOrderDerivatives(
            const LocalPoint &localPoint) {
    const auto r = localPoint[R], rp = 1 + r, rm = 1 - r, //
               s = localPoint[S], sp = 1 + s, sm = 1 - s;
    return { { //
        // Derivatives ∂/∂r
        { //
            -0.25 * sm, // (1)
            +0.25 * sm, // (2)
            +0.25 * sp, // (3)
            -0.25 * sp  // (4)
        }, //
        // Derivatives ∂/∂s
        { //
            -0.25 * rm, // (1)
```

[2]Note that, in *comana*, the local variables are named r, s, and t rather than ξ, η, and ζ.

```
            -0.25 * rp, // (2)
            +0.25 * rp, // (3)
            +0.25 * rm  // (4)
        } //
    } };
}

template<>
Quadrilateral4Basis::SecondOrderDerivatives
        Quadrilateral4Basis::evaluateSecondOrderDerivatives(
            const LocalPoint &) {
    return { { //
        // Derivatives ∂²/∂r²
        { //
            0, // (1)
            0, // (2)
            0, // (3)
            0  // (4)
        }, //
        // Derivatives ∂²/∂s²
        { //
            0, // (1)
            0, // (2)
            0, // (3)
            0  // (4)
        }, //
        // Derivatives ∂²/∂r∂s
        { //
            +0.25, // (1)
            -0.25, // (2)
            +0.25, // (3)
            -0.25  // (4)
        } //
    } };
}
```

A field quantity $\boldsymbol{u}(\boldsymbol{\xi})$ can then be evaluated at a particular point $\boldsymbol{\xi}$ of the parameter space by means of the `interpolate` method, which first evaluates the basis functions at $\boldsymbol{\xi}$ and subsequently accumulates the products of the evaluated shape functions $N_i(\boldsymbol{\xi})$ and the values u_i stored at the cell's degrees of freedom.

In order to gather topological cell information, we introduce the `CellTopology` class:

```
template<class Basis1, std::size_t numberOfSurfaces1>
class CellTopology final {
public:
    using Basis = Basis1;

    using LocalSpace = typename Basis::LocalSpace;

    static constexpr std::size_t localSpaceDimension { LocalSpace::dimension };
```

```
    using LocalPoint = typename LocalSpace::Point;

    static constexpr std::size_t numberOfVertices { Basis::numberOfFunctions };

    using LocalVertexCoordinates = Array<LocalPoint, numberOfVertices>;

    static const LocalVertexCoordinates localVertexCoordinates;

    static constexpr std::size_t numberOfSurfaces { numberOfSurfaces1 };

    using SurfaceIndices = Array<SizeTypeDynamicArray, numberOfSurfaces>;

    static const SurfaceIndices surfaceIndices;

    static constexpr std::size_t surfaceLocalSpaceDimension {
            localSpaceDimension - 1 };

    using SurfaceLocalPoint = Point<surfaceLocalSpaceDimension>;

    // ...
};
```

Its members are likewise specialized for each particular cell topology, for instance:

```
template<>
const Quadrilateral4Topology::LocalVertexCoordinates
            Quadrilateral4Topology::localVertexCoordinates { { //
        { -1, -1 }, // (1)
        { +1, -1 }, // (2)
        { +1, +1 }, // (3)
        { -1, +1 }, // (4)
    } };

template<>
const Quadrilateral4Topology::SurfaceIndices
            Quadrilateral4Topology::surfaceIndices { { //
        { { 0, 1 } }, // (1)
        { { 1, 2 } }, // (2)
        { { 2, 3 } }, // (3)
        { { 3, 0 } }  // (4)
    } };

// ...
```

Finally, the `CellGeometry` class connects the cell topology to the global space the cell is defined in by accepting the dimension of the global space as a template parameter:

```
template<class Topology, std::size_t globalSpaceDimension>
class CellGeometry final {
public:
```

```
    using GlobalPoint = Point<globalSpaceDimension>;

    using GlobalVector = DoubleVector<globalSpaceDimension>;

    template<class VertexCoordinates>
    static BoundingBox computeBoundingBox(
        const VertexCoordinates &vertexCoordinates) noexcept;

    template<class VertexCoordinates, class LocalPoint1>
    static double computeJacobianDeterminant(
        const VertexCoordinates &vertexCoordinates,
        const LocalPoint1 &localPoint);

    template<class VertexCoordinates, class LocalPoint1>
    static DoubleVector<globalSpaceDimension> computeNormal(
        const VertexCoordinates &vertexCoordinates,
        const LocalPoint1 &localPoint);

    using ProjectionResult = Comana::ProjectionResult<LocalPoint, GlobalPoint>;

    template<class VertexCoordinates, class GlobalPoint1>
    static ProjectionResult projectPoint(
        const VertexCoordinates &vertexCoordinates,
        const GlobalPoint1 &globalPoint);

    // ...

private:
    using ProjectionFunctionValue = DoubleVector<localSpaceDimension>;

    using ProjectionJacobianMatrix = DoubleMatrix<localSpaceDimension,
            localSpaceDimension>;

    using ProjectionOutput = Tuple<ProjectionFunctionValue,
            ProjectionJacobianMatrix, GlobalPoint, GlobalVector>;

    template<class VertexCoordinates, class GlobalPoint1>
    static ProjectionOutput projectPointToVolumeHelper(
            const VertexCoordinates &vertexCoordinates,
            const GlobalPoint1 &globalPoint, const LocalPoint &localPoint =
                    LocalPoint::Zero());
};
```

In doing so, essentially the same set of functions can, for instance, be used for quadrilateral cells in two- or three-dimensional space. Note that the `CellGeometry` class does not require any specializations as any cell-specific information was shifted to the specializations of the `Basis` and `CellTopology` template, which promotes a very generic, modular, and extensible design.

5.4.3 Projection Procedure

For the projection procedure outlined in Section 4.5.5, an iterative Newton-Raphson procedure is required as one essential building block. To this end, we provide a `solve` function, which accepts a functor, an initial iterate $\boldsymbol{x}^0$, a convergence tolerance, and a maximum number of iterations after which to stop the iterative procedure irrespective of whether convergence has been achieved. Given an argument $\boldsymbol{x}$, the functor is expected to return a tuple consisting of at least the function value $\boldsymbol{f}(\boldsymbol{x})$ and the Jacobian matrix $\boldsymbol{J}(\boldsymbol{x})$. In order to avoid recomputing the functor output in the projection procedure, the `solve` function does not only return the converged solution $\boldsymbol{x}^*$, for which $\boldsymbol{f}(\boldsymbol{x}^*) = \boldsymbol{0}$ to sufficient accuracy, but also the last function output, which contains the function value $\boldsymbol{f}(\boldsymbol{x}^*)$ and the Jacobian matrix $\boldsymbol{J}(\boldsymbol{x}^*)$:

```
template<class Functor, class Argument>
using FunctionOutput = typename std::result_of<Functor(Argument)>::type;

template<class Functor, class Argument>
using SolutionOutput = Pair<Argument, FunctionOutput<Functor, Argument>>;

template<class Functor, class Vector>
SolutionOutput<Functor, Vector> solve(Functor functor,
        Vector startVector, const double tolerance = VERY_TINY_VALUE,
        const std::size_t maximumNumberOfIterations = 1e2) {
    FunctionOutput<Functor, Vector> functionOutput;
    for (auto iteration : range(maximumNumberOfIterations)) {
        static_cast<void>(iteration); // unused
        functionOutput = functor(startVector);
        const auto &functionValue = std::get<0>(functionOutput);
        if (functionValue.norm() <= tolerance) break;
        const auto &jacobianMatrix = std::get<1>(functionOutput);
        const Vector increment(jacobianMatrix.inverse() * functionValue);
        if (increment.norm() <= tolerance) break;
        startVector -= increment;
    }
    return {startVector, functionOutput};
}
```

Based on this, the `projectPoint` method of the `CellGeometry` class can be implemented as follows:

```
template<class Topology, std::size_t globalSpaceDimension>
template<class VertexCoordinates, class GlobalPoint1>
typename CellGeometry<Topology, globalSpaceDimension>::ProjectionResult
        CellGeometry<Topology, globalSpaceDimension>::projectPoint(
            const VertexCoordinates &vertexCoordinates,
            const GlobalPoint1 &globalPoint) {
    // Project to volume first
    auto functor = std::bind(&projectPointToVolume<VertexCoordinates,
            GlobalPoint1>, vertexCoordinates, globalPoint,
            std::placeholders::_1);
```

```
    auto solutionOutput = solve(functor, LocalPoint(LocalPoint::Zero()));
    ProjectionResult projectionResult { std::move(solutionOutput.first),
        std::move(std::get<2>(solutionOutput.second)),
        std::move(std::get<3>(solutionOutput.second)) };
    if (Topology::LocalSpace::isInside(projectionResult.localPoint))
        return projectionResult;

    // Successively project to surfaces if local point is NOT inside the
    // cell volume...
}
```

Herein, the given global point $\boldsymbol{x}$ is first projected to the "volume" of the cell (that is, the cell itself). If the local coordinates $\boldsymbol{\xi}$ of the projected global point $\boldsymbol{x}^*$ are outside the local space of the cell, the global point is subsequently projected to the surfaces of the cell. Finally, the recursion stops at zero-dimensional surfaces (or, in other words, the vertices of the cell) and the projected point is exactly that point of the cell, which is closest to the global point $\boldsymbol{x}$ in the sense of the Euclidean distance.

5.4.4 Integration

Promoting a modular design, integration rules in *comana* are not bound to a specific cell type, but implemented as separate classes. For example, let us consider an integration rule for quadrilateral cells based on Gauss-Legendre integration:

```
template<std::size_t numberOfIntegrationPoints_>
class QuadrilateralIntegration {
public:
    static constexpr std::size_t numberOfIntegrationPoints {
        numberOfIntegrationPoints_ };

    static constexpr std::size_t localSpaceDimension { 2 };

    using IntegrationPoint = Comana::IntegrationPoint<localSpaceDimension>;

    using IntegrationPoints = Array<IntegrationPoint,
        numberOfIntegrationPoints>;

    static const IntegrationPoints & getIntegrationPoints() noexcept;
};
```

An `IntegrationPoint` combines the coordinates of the sampling points on the unit square $[-1, 1]^2$ and the associated integration weight into a single structure. The `QuadrilateralGaussLegendreIntegration` class is a template that can be specialized for different numbers of integration points. In the `getIntegrationPoints` method, the integration points are initialized as the tensor product of the one-dimensional Gauss-Legendre integration points upon first access following the RAII idiom.

For the integration of a field over a cell, the class `Integrator` is provided:

```
template<class IntegrationRule, class MappingFunctions,
        class ShapeFunctions = MappingFunctions>
```

```
class Integrator {
public:
    template<class VertexCoordinates>
    static Field integrate(const VertexCoordinates &vertexCoordinates,
        Field::const_iterator &quantityIterator) noexcept;

private:
    using WeightedShapeFunctionsAtIntegrationPoint
        = typename ShapeFunctions::ValueArray;

    // We use a dynamic array to save stack space
    using WeightedShapeFunctionsAtAllIntegrationPoints
        = DynamicArray<WeightedShapeFunctionsAtIntegrationPoint>;

    using MappingFunctionDerivativesAtIntegrationPoint
        = Array<typename MappingFunctions::ValueArray,
            MappingFunctions::localSpaceDimension>;

    // We use a dynamic array to save stack space
    using MappingFunctionDerivativesAtAllIntegrationPoints
        = DynamicArray<MappingFunctionDerivativesAtIntegrationPoint>;

    static WeightedShapeFunctionsAtAllIntegrationPoints
        initializeWeightedShapeFunctionsAtAllIntegrationPoints() noexcept;

    static const WeightedShapeFunctionsAtAllIntegrationPoints &
        getWeightedShapeFunctionsAtAllIntegrationPoints() noexcept;

    static MappingFunctionDerivativesAtAllIntegrationPoints
        initializeMappingFunctionDerivativesAtAllIntegrationPoints() noexcept;

    static const MappingFunctionDerivativesAtAllIntegrationPoints &
        getMappingFunctionDerivativesAtAllIntegrationPoints() noexcept;
};
```

`Integrator` represents a generic class that can be specialized for a particular integration rule and for particular sets of mapping and shape functions. By default, the shape functions are assumed to be identical to the mapping functions. In this case, the cell, on which the integration is performed, is isoparametric. In its public interface, the `Integrator` class exposes only the `integrate` method. For the computation of the Jacobian determinant, it requires the vertices of the cell as an input argument. Furthermore, it takes a `quantityIterator` as a second argument, which refers to the range of function values evaluated at the integration point locations. Consequently, the range the `quantityIterator` operates on must contain at least as many quantities as there are integration points as determined by the integration rule. Private member functions are responsible for the evaluation of the mapping and shape functions at the integration points. This avoids having to recompute these values upon each call to `integrate`. Also here, the RAII idiom is applied, and the data structures are initialized locally inside the member function on first access. Based on this, the implementation of the `integrate` method then becomes as simple as

```
template<class IntegrationRule, class MappingFunctions, class ShapeFunctions>
template<class VertexCoordinates>
Field Integrator<IntegrationRule, MappingFunctions, ShapeFunctions>::integrate(
        const VertexCoordinates &vertexCoordinates,
        Field::const_iterator &quantityIterator) noexcept {
    const auto numberOfVertices = vertexCoordinates.size();
    const auto quantitySize = quantityIterator->size();
    Field result(quantitySize, numberOfVertices, 0);
    const auto &weightedShapeFunctionsAtAllIntegrationPoints =
            getWeightedShapeFunctionsAtAllIntegrationPoints();
    const auto &mappingFunctionDerivativesAtAllIntegrationPoints =
            getMappingFunctionDerivativesAtAllIntegrationPoints();
    for (auto integrationPointIndex : range(
            IntegrationRule::numberOfIntegrationPoints)) {
        const auto jacobianMatrix = computeJacobianMatrix(
            mappingFunctionDerivativesAtAllIntegrationPoints[
                integrationPointIndex], vertexCoordinates);
        const auto jacobianDeterminant
            = computeJacobianDeterminant(jacobianMatrix);
        for (auto vertexIndex : range(numberOfVertices))
            result[vertexIndex] += weightedShapeFunctionsAtAllIntegrationPoints[
                integrationPointIndex][vertexIndex] * jacobianDeterminant *
                *quantityIterator;
        ++quantityIterator;
    }
    return result;
}
```

5.5 Algorithmic Data Structures

One of the most notable strengths of *comana* is the flexibility to customize a partitioned solution strategy to the particular problem under consideration. For each coupled multifield problem, the user implements the coupling algorithm in a dedicated C++ program, which is then compiled and launched as the master process. In order to simplify the implementation of a coupling strategy as much as possible, *comana* provides a vast range of well-tested, modular, and extensible algorithmic building blocks. These building blocks enable the user to develop a coupling algorithm in C++, which is hardly distinguishable from pseudocode notation. An illustrative example of a full simulation setup is provided in Section 5.7 for the simple coupled problem considered in Section 4.3. Before that, the implementation of the constituent parts of the coupling algorithm is briefly outlined in the following.

5.5.1 Predictors

In every coupled problem, it is advisable to employ a predictor at the beginning of each time increment so as to provide an initial solution to the first solver $\mathcal{S}_1$, which is closer to the (initially unknown) converged solution in the current time increment than the converged solution from the previous time increment. All predictor classes accept the converged unmodified solution $\mathbf{u}_j$ from the previous time increment and compute the initial modified

solution $\tilde{\mathbf{u}}_{j+1}^{0}$ for the current time increment. In addition, a `RunInfo` instance needs to be supplied. `RunInfo` objects store the time increment counter, time, time step size, and iteration index – and they serve to control the execution of the coupling algorithm and terminate the simulation once all time increments have been computed. For a predictor, the `RunInfo` object is needed, due to the fact that it contains the information about the current time step size, which is required for prediction. One of the simplest predictors is the `PolynomialPredictor`. It is constructed from the polynomial order p, and it stores the converged solutions $\mathbf{u}_j, \mathbf{u}_{j-1}, \ldots$ from the previous time increments upon each invocation of its `predict` method:

```
class PolynomialPredictor {
public:
    PolynomialPredictor(const std::size_t polynomialOrder);
    void predict(const RunInfo &runInfo, DoubleDynamicVector &solution);
private:
    std::size_t polynomialOrder_;
    Deque<DoubleDynamicVector> solutionSeries_;
    DoubleDeque timeStepSizes_;
};
```

For the storage of the solutions $\mathbf{u}_j, \mathbf{u}_{j-1}, \ldots$, the use of a `Deque` is of particular advantage. If the size of the deque exceeds $p + 1$, the foremost entry can cheaply be deleted by the `Deque`'s `pop_front` method. In addition, the `PolynomialPredictor` keeps track of the varying time step sizes to compute the Vandermonde matrix $\boldsymbol{V}$ and then its inverse $\boldsymbol{V}^{-1}$. Also here, the use of a `Deque` proves beneficial. Further predictors such as the `LinearExtrapolationPredictor`, `TaylorSeriesBasedPredictor`, and `AdaptiveNewmarkPredictor`, cf. Section 4.4, are provided and used in almost the same manner.

5.5.2 Interpolation Schemes

As the next constituent parts of a coupling algorithm, let us consider the implementation of the various interpolation schemes available in *comana*. In this context, it is important to remark that only mesh-independent interpolation schemes are used explicitly in the implementation of a coupling algorithm. Mesh-based interpolation schemes, in contrast, are employed implicitly in the adapter library of a slave if the user supplies a set of custom points. If custom points are used, a subfield's boundary conditions or result quantities can be directly requested at these points – thus eliminating the need for an explicit interpolation scheme in the coupling algorithm. For most interpolation schemes, it proves useful to store the interpolation weights in a nested list $\mathcal{W} = \{\mathcal{W}_1, \ldots, \mathcal{W}_m\}$ of m sets $\mathcal{W}_i = \{\ldots, (\boldsymbol{p}_j, w_j), \ldots\}$ of pairs of source points $\boldsymbol{p}_j$ and weights w_j associated to each of the m query points $\boldsymbol{q}_i$. That way, the computation of the interpolation matrix is avoided and the interpolation can be performed as efficient as possible. In *comana*, a generic interpolation weight storing the source point index and an associated weight is provided for this task:

```
struct InterpolationWeight {
    const std::size_t index;
    double weight;
};
```

Convenient typedefs for the sets $\mathcal{W}_i$ and the entire nested set $\mathcal{W}$ are also available:

```
using InterpolationWeightDynamicArray = DynamicArray<InterpolationWeight>;

using InterpolationWeightDynamicArrayDynamicArray
    = DynamicArray<InterpolationWeightDynamicArray>;
```

Employing these auxiliary data types, the interface of the consistent barycentric surface interpolation in three-dimensional space becomes as simple as

```
class BarycentricConsistentSurface3DInterpolation final {
public:
    BarycentricConsistentSurface3DInterpolation(const Field &sourcePoints,
            const Field &targetPoints);

    void apply(const Field &sourceQuantities, Field &targetQuantities) const;

    Field apply(const Field &sourceQuantities) const;

private:
    InterpolationWeightDynamicArrayDynamicArray interpolationWeights_;
};
```

Similar to other interpolation schemes, this barycentric interpolation scheme is constructed from a `Field` of source points $\{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n\}$ and query points $\{\boldsymbol{q}_1, \ldots, \boldsymbol{q}_m\}$. Like the predictor classes, the interpolation schemes support two variants of the `apply` method responsible for actually performing the interpolation. While the first one works in-place and modifies existing `targetQuantities` without allocating additional memory, the second variant returns the interpolated field quantities by value. Further barycentric and the other mesh-independent interpolation schemes exhibit a similar interface. For each mesh-independent interpolation scheme, a consistent as well as a conservative version is available.

5.5.3 Convergence Acceleration Schemes

It has already been discussed in Chapter 4 that a convergence acceleration scheme is essential to stabilize the partitioned solution procedure and to reduce the number of implicit iterations by improving the convergence behavior. In *comana*, a convergence acceleration scheme expects the current modified solution $\tilde{\mathbf{u}}_{j+1}^k$, the unmodified solution $\mathbf{u}_{j+1}^k$, and the residual $\mathbf{r}_{j+1}^k$ to compute an updated modified solution $\tilde{\mathbf{u}}_{j+1}^{k+1}$. Providing an almost identical interface to the user, the convergence acceleration schemes offer the `updateSolution` method, which either modifies the modified solution $\tilde{\mathbf{u}}_{j+1}^k$ in-place to produce $\tilde{\mathbf{u}}_{j+1}^{k+1}$ or to return it by value. During the simulation, the convergence acceleration schemes keep track of the coupling quantities from previous iterations and time increments by storing them as private members.

For example, consider the `QuasiNewtonLeastSquaresMethod` class

```
class QuasiNewtonLeastSquaresMethod final {
public:
    QuasiNewtonLeastSquaresMethod(const std::size_t numberOfReusedTimeSteps = 0,
        const double staticRelaxationFactor = 0.2);
```

```
    void updateSolution(const RunInfo &runInfo,
            DoubleDynamicVector &modifiedSolution,
            const DoubleDynamicVector &unmodifiedSolution,
            const DoubleDynamicVector &residual);

private:
    std::size_t numberOfReusedTimeSteps_;
    double staticRelaxationFactor_;
    Deque<DoubleDynamicMatrix> unmodifiedSolutionDifferences_;
    Deque<DoubleDynamicMatrix> residualDifferences_;
};
```

For the construction of an instance of this class, the user needs to supply the maximum number of time steps ℓ to be used for the construction of the matrices $\mathbf{V}_{j+1}^k$ and $\mathbf{W}_{j+1}^k$. By default, only the current time increment is used. In addition, it is possible to specify the static relaxation factor ω for the first iteration $k = 0$ in each time increment. In each iteration within a time increment, the `updateSolution` method is invoked to compute the modified solution $\tilde{\mathbf{u}}_{j+1}^{k+1}$. As arguments, the method takes a `RunInfo` object storing information about the current time increment and iteration, the current modified solution $\tilde{\mathbf{u}}_{j+1}^k$, the unmodified solution $\mathbf{u}_{j+1}^k$, and the residual $\mathbf{r}_{j+1}^k$. Of course, the latter three coupling quantities are not independent of each other. but already available in the coupling algorithm. Hence, they can be used by this function to avoid recomputing them. For the evaluation of the matrices $\mathbf{V}_j, \mathbf{V}_{j-1}, \ldots$ and $\mathbf{W}_j, \mathbf{W}_{j-1}, \ldots$ from the previous time increments and the matrices $\mathbf{V}_{j+1}^k$ and $\mathbf{W}_{j+1}^k$ from the current time increment, cf. Algorithm 12, the private members `unmodifiedResidualDifferences_` and `unmodifiedSolutionDifferences_` have been introduced and are updated upon each call to the `updateSolution` member function.

The other convergence acceleration schemes presented in Section 4.7 have been implemented in *comana* as well, and they can be used in just about the same manner as the `QuasiNewtonLeastSquaresMethod` class.

5.6 Adapter Data Structures

In order to enable a solver to participate in a partitioned solution procedure, a dedicated, solver-specific adapter library is required. It provides the ability to communicate to the master process and implements the interface to access or modify the solver's database. Oriented towards modularity, it is divided into a generic, solver-independent part (implemented in C++) and a solver-specific part (partly implemented in C++ and in the language of the solver code). Referring again to Figure 5.2, the slave socket used as a communication hub to the master process and the request handler responsible for processing requests received from the master process are integrated into the generic component. As the interrupt functions are generic too, they are implemented in the C++ language as well. However, the interrupt functions are invoked from the solver code, which may be implemented in a programming language other than C++. Hence, language-specific interfaces to the interrupt functions must be provided. Currently, *comana* supports interfacing solvers implemented in Fortran, MATLAB/Octave, Python, or APDL. Apart from

the generic solver-independent components, the adapter library consists of a solver-specific part tailored to access and modify a solver's data structures during the partitioned solution procedure. Accessor and process patches serve as a generic interface to the solver's database for the request handler. For these entities, we introduce generic base classes to be inherited by the concrete solver-specific implementations. Aiming at reducing code duplication and entailing the reuse of generic mesh data structures, the accessor and process patch base and child classes are implemented in C++. The last remaining component of the adapter library is the driver, which manages the data transfer between the solver's database and the adapter library. In order to simplify access from the accessor or the process patch (both of which are implemented in C++) another interface layer is introduced between the low-level driver functions and the rest of the adapter library. This interface serves to define a dedicated set of functions to manipulate a solver's database from C++ and to shield the specific implementations of the driver functions in the native solver language from the rest of the adapter library – thus promoting modularity and simplified testing.

In the following, we first present the generic interrupt functions as well as the accessor and process patch base classes. Furthermore, we will outline the interplay between this part of the adapter library and the request handler and the slave socket presented in Section 5.3. Next, we discuss the language-specific interfaces and helper classes to invoke the interrupt functions from the solver code on the one hand and to manipulate the solver's database by means of the driver functions on the other hand.

5.6.1 Generic Functions and Classes

In Section 5.1, we discussed the general architecture of an adapter library used to enhance a solver for a partitioned solution strategy. It was outlined that a major part of the adapter library is generic in the sense that no solver-specific modifications are required. In order to reduce code duplication and the effort invested into testing, the solver-specific part may use generic data structures as well.

As illustrated in Algorithm 14 in lines 2, 7, 9, and 18, the modified solver prepared for a partitioned solution procedure needs to invoke functions to initiate the connection to the master process, transfer data to or receive data from the master process, and finally terminate the connection to the master process. By calling the function

```
void comana_initialize_coupling() {
    if (!Mpi::isCoupledProcess())
        return;

    Mpi::initialize(); // if ‘Mpi::initialized()‘ is ‘true‘,
                       // ‘Mpi::finalizeMpi = false‘
    getSlaveSocket().connect();
}
```

the solver establishes the connection to the master process. Before actually connecting, it is checked whether the current process is run as part of a coupled solution procedure. If this is *not* the case, the function returns directly. This way, a solver executable modified according to Algorithm 14 and linked to an adapter library can still be used for uncoupled simulations. It behaves in the same manner as an unmodified solver as sketched in Algorithm 13, for

instance. By setting the environment variable `COMANA_PROCESS_GROUP` to a non-empty value before launching the solver executable, the subfield solver is signaled that it is used in a coupled solution procedure. In that case, the MPI session is initialized and the internal variable `Mpi::finalizeMpi` is set to true. If the solver has already initialized the MPI session, no further initialization is required and, hence, also the finalization of the MPI session is left to the solver itself; `Mpi::finalize` is set to `false`. Once the MPI session has been initialized, the slave socket is connected. Again following the RAII principle, it is initialized upon first access:

```
SlaveSocket & getSlaveSocket() {
    static SlaveSocket socket;
    return socket;
}
```

As the next interrupt function, the solver calls

```
int comana_do_transfer(const int stage) {
    if (!Mpi::isCoupledProcess())
        return true;

    auto &accessor = getAccessor();
    accessor.initializeTransfer(stage);

    int status;

    while (true) {
        const auto &slaveSocket = getSlaveSocket();
        const auto requestTag = receive<RequestTag>(slaveSocket);

        switch (requestTag) {
        case RequestTag::proceed:
            status = RequestHandler::proceed(slaveSocket);
            break;
        case RequestTag::initializePatch:
            RequestHandler::initializePatch(slaveSocket, accessor);
            break;
        // Further requests, omitted here
        default:
            // Error: unhandled request
        }
    }

    accessor.finalizeTransfer(stage);

    return status;
}
```

Again, it is first checked whether the process is actually a coupled process. Similar to the slave socket, the accessor is obtained from a function `getAccessor()` initializing a local static `Accessor` instance upon first access according to the RAII idiom. Following

that, the accessor is notified about the fact that a data transfer takes place by invoking the `initializeTransfer` method. As an argument, we pass the `stage` variable indicating at which point in the solution process the data transfer is performed. In line with Algorithm 14, `stage` takes the value `0` in the first data transfer and the value `1` in the second data transfer. Subsequently, the slave process successively receives requests from the master process. Based on the `requestTag`, the appropriate request handler is selected. Most request handlers expect the accessor and the slave socket as arguments. While the accessor is required to manipulate the solver's database through the driver functions, the slave socket is used to receive request data from or send request data to the master process. A special request handler is `RequestHandler::proceed`, which receives a status from the master process indicating whether to continue the implicit iteration within the current time increment (`status` takes the value `0`) or terminate the implicit iteration and proceed to the next time increment (`status` takes the value `1`). In the first data transfer, the return value of `comana_do_transfer` is unused and, hence, always set to `0`, while both values are possible in the second data transfer. Recalling again the modified generic solver in Algorithm 14, the return value of `comana_do_transfer` after the second data transfer in line 9 serves to decide whether to stop the implicit iteration in line 10–12.

As a last interrupt function, we have

```
void comana_exit_coupling() {
    if (!Mpi::isCoupledProcess())
        return;

    getSlaveSocket().disconnect();
    Mpi::finalize(); // Only invokes 'MPI_Finalize' if 'finalizeMpi'
                     // was previously set to 'false'
}
```

After the obligatory check for a coupled process, the slave socket is disconnected from the master process, and the MPI session is finalized. If the solver itself has initialized the MPI session before calling `comana_initialize_coupling`, the last step is omitted and the solver is expected to also finalize the MPI session after calling `comana_finalize_coupling`.

Having discussed the implementation of the generic interrupt functions, let us consider the accessor and process patch classes serving as the base classes for the concrete solver-specific implementations of these concepts. In the `Accessor` base class, the following interface is provided:

```
class Accessor {
public:
    virtual ~Accessor() noexcept = 0;

    virtual void initializeTransfer(const int stage);

    virtual void finalizeTransfer(const int stage);

    virtual void initializePatch(const String &label,
        const Topology topology);

    virtual const ProcessPatch & getPatch(const String &label) const;
```

```
    ProcessPatch & getPatch(const String &label);

    virtual Quantity getGlobalQuantity(const QuantityType quantityType) const;

    virtual void setGlobalQuantity(const QuantityType quantityType,
        const Quantity &quantity);
};
```

Because `Accessor` is a pure virtual base class, the destructor must also be declared as pure virtual. Signaling the initiation or the end of a data transfer to an `Accessor` instance, the methods `initializeTransfer` and `finalizeTransfer` are called in `comana_do_transfer`. For the initialization of a patch and associated data structures, the `initializePatch` member function is supplied. Next, the `getPatch` function returns a patch of base type `ProcessPatch`. As will be explained in a moment, the `ProcessPatch` class provides member functions to manipulate discrete field data. Last but not least, the `Accessor` class provides the member functions `getGlobalQuantity` and `setGlobalQuantity` to query or modify global quantities such as the time step size Δt, for instance, in a solver's database. All member functions of the `Accessor` class are equipped with a default implementation such that children do not necessarily need to override all member functions. In case a non-overridden member function is called on a solver-specific `Accessor` instance, the base class implementation throws an exception signaling the missing functionality to the caller.

A second important base class providing high-level access to a solver's database is the `ProcessPatch` class. It provides the following public interface:

```
class ProcessPatch {
public:
    virtual ~ProcessPatch() noexcept = 0;

    virtual void initializeCustomPoints(const std::size_t pointSetIndex,
        const SizeTypeDynamicArray &pointIndices,
        const Field &globalCoordinates);

    virtual void selectCustomPoints(const std::size_t pointSetIndex,
        const SizeTypeDynamicArray &pointIndices);

    virtual const SizeTypeDynamicArray & getCustomPointIndices(
        const std::size_t pointSetIndex) const;

    std::size_t getFieldSize(const Location location) const;

    std::size_t getCustomFieldSize(const std::size_t pointSetIndex) const;

    virtual Field getField(const Location location,
        const QuantityType quantityType) const;

    virtual Field getCustomField(const std::size_t pointSetIndex,
            const QuantityType quantityType) const;

    virtual void setField(const Location location,
```

```
        const QuantityType quantityType, const Field &field);

    virtual void setCustomField(const std::size_t pointSetIndex,
            const QuantityType quantityType, const Field &field);
};
```

Similar to the `Accessor` class, the `ProcessPatch` class is a pure virtual base class and, hence, needs to provide a pure virtual destructor. By overriding the `initializeCustomPoints` method, children may implement the projection procedure outlined in Section 4.5.5 or 4.5.6. As its arguments, the method expects the index of the set of custom points, the individual indices of the custom points in the entire user-supplied array of custom points in the master process, and their coordinates. For the custom point scattering procedure discussed in Section 5.4.1, a `selectCustomPoints` method is available to select and keep previously initialized custom points and discard all custom points from the given point set with indices that are not part of the array of custom point indices. In addition, the `getCustomPointIndices` member function was introduced to be able to associate the custom points of a slave process to the custom points originally provided by the user based on their index in the entire set of user-supplied custom points. Further, the `getFieldSize` function returns the size of a field at a particular set of points defined by the `location` argument. Lastly, the `getField` and `setField` method are intended to access or modify a certain type of quantity at a particular set of points of the discretization. Analogous methods are available for custom points. Like the `Accessor` class, the `ProcessPatch` class provides default implementations for each of its member functions, throwing an exception in case a child does not override these functions. In the implementation of its member functions, children of the `ProcessPatch` base class may use any of the generic mesh data structures presented in Section 5.4 or shift the actual work to the various generic utility functions provided in *comana*. That way, the effort to implement a derived solver-specific `ProcessPatch` class is reduced to a minimum. As a bonus, code duplication is avoided, and the need for testing is limited to a small amount of added functionality only.

5.6.2 C/C++ Solvers

For C and C++ solvers, the interface to the generic interrupt functions is as simple as

```
#ifdef __cplusplus
extern "C" {
#endif
void comana_initialize_coupling();
int comana_do_transfer(const int stage);
void comana_exit_coupling();
#ifdef __cplusplus
}
#endif
```

This header is also part of the implementation of the generic interrupt functions as outlined in the previous section. If a C++ compiler is used to compile a source file including this interface, the `extern "C"` construct prevents the function names from being mangled, but to have C linkage instead. Consequently, both C and C++ solvers can include the same header and refer to the same implementation of the generic interrupt functions.

No special measures must be taken for the data transfer between the driver and the solver database, as all data types can be exchanged natively.

5.6.3 Fortran Solvers

Fortran solvers use the same implementation of the generic interrupt functions, but they do not call them directly. Instead, an additional interface layer is provided, which wraps the calls to the C++ implementation of the interrupt functions:

```
#ifdef __cplusplus
extern "C" {
#endif
void comana_initialize_coupling_();
int comana_do_transfer_(const int stage);
void comana_exit_coupling_();
#ifdef __cplusplus
}
#endif
```

Note the underscores appended to the end of the function names, which are necessary due to the fact that most Fortran compilers expect external symbol names to have an underscore appended to them. A symbol *without* an underscore would result in a linker error.

Data between the driver and the solver database can be exchanged by passing their memory address. Yet, only fundamental data types such as character arrays for the representation of strings or integers and floating point numbers or arrays thereof are used to exchange information to keep the interface as transparent and simple as possible and to reduce portability issues to a minimum. On most platforms, these data types can be safely exchanged without using the intrinsic `iso_c_binding` module available since Fortran 2003. Moreover, there is hardly any information that cannot be represented by any of the previously mentioned data types or a combination thereof in numerical computations.

5.6.4 MATLAB/Octave Solvers

For solvers implemented in MATLAB or Octave, the interrupt functions need to provide an interface that is callable from a scripting language. Both MATLAB and Octave solvers call the anonymous functions

```
comana_initialize_coupling ...
    = @() <solver_adapter_library>('comana_initialize_coupling');
comana_do_transfer ...
    = @(stage) <solver_adapter_library>('comana_do_transfer', stage);
comana_exit_coupling = @() <solver_adapter_library>('comana_exit_coupling');
```

where `<solver_adapter_library>` is substituted by the adapter library name without file extension. The anonymous functions redirect to a shared library `<solver_adapter_library>.mex` that implements the function

```
void mexFunction(int number_of_output_arguments, mxArray *output_arguments[],
        int number_of_input_arguments, const mxArray *input_arguments[]);
```

All anonymous functions invoke the same `mexFunction` and pass their given arguments to that function. Based on the first input argument, `mexFunction` decides which interrupt function to call.

For the data exchange between the driver and the solver database, C++ data structures must be converted to MATLAB/Octave data structures – and vice versa. From C/C++, MATLAB/Octave functions are called through the function

```
int mexCallMATLAB(int number_of_output_arguments, mxArray *output_arguments[],
    int number_of_input_arguments, mxArray *input_arguments[],
    const char *command_name);
```

declared in `mex.h`. The called command populates the resulting array of `mxArrays` by allocating dynamic memory for each output argument and by storing the pointer in `output_arguments`. Likewise, `input_arguments` is an array of dynamically allocated `mxArrays`. For both input and output arguments, the caller must ensure that the dynamic memory is again properly released after the function call. To this end, variadic templates and the automatic memory management of smart pointers prove particularly useful.

First of all, a file `mex_type.def` is created to associate the MATLAB/Octave data types to their corresponding C++ data types:

```
MEX_TYPE_DEFINE(mxLOGICAL_CLASS, bool)
MEX_TYPE_DEFINE(mxCHAR_CLASS, char)
MEX_TYPE_DEFINE(mxINT8_CLASS, int8_t)
// Further fixed-size integer types; omitted here
MEX_TYPE_DEFINE(mxSINGLE_CLASS, float)
MEX_TYPE_DEFINE(mxDOUBLE_CLASS, double)
```

Making these type associations available in C++, the template

```
template<class cppType>
struct CppToMexType {
};
```

is introduced and fully specialized using the definitions from `mex_type.def`:

```
#define MEX_TYPE_DEFINE(mexType, cppType) \
    template<> \
    struct CppToMexType<cppType> { \
        static constexpr auto type = mexType; \
    };
#include "mex_type.def"
#undef MEX_TYPE_DEFINE
```

In order to pass a C++ variable to MATLAB/Octave, it needs to be converted to a `mxArray` that can be passed as an argument to `mexCallMATLAB`. For this purpose, the `MexAllocator` class is introduced:

```
template<class Type, class Enable = void>
struct MexAllocator {
};
```

This class is then specialized for different types, for instance:

```
template<>
struct MexAllocator<bool> {
    static mxArray * allocate(const bool value) noexcept;
};
```

If required, SFINAE can be employed to conditionally enable certain specializations based on the template argument:

```
template<class Container>
struct MexAllocator<Container,
        typename std::enable_if<std::is_arithmetic<
            typename Container::value_type>::value>::type> {
    static mxArray * allocate(const Container &container) noexcept {
        const mwSize size[] = { static_cast<mwSize>(container.size()), 1 };
        auto object = mxCreateNumericArray(2, size,
                CppToMexType<typename Container::value_type>::type, mxREAL);
        std::copy(std::begin(container), std::end(container),
                static_cast<typename Container::pointer>(mxGetData(object)));
        return object;
    }
};
```

Similar to the `MexAllocator` class, we provide a `MexDeallocator` class, which provides the `operator()` that takes an array of `mxArrays` and deletes every element in that array before destroying the outer array:

```
template<std::size_t size>
struct MexDeallocator {
    void operator()(mxArray *object[]) noexcept {
        std::for_each(object, object + size,
            [](mxArray *element) { mxDestroyArray(element); });
        delete[] object;
    }
};
```

This class can then be used as a custom deleter for a `UniquePointer` that manages a dynamically allocated `mxArray`:

```
template<std::size_t size>
using MexUniquePointer = UniquePointer<mxArray *[], MexDeallocator<size>>;
```

Filling an `mxArray *[]` before supplying it to `mexCallMATLAB` is possible by means of the `mexAllocateHelper`, which is based on a variadic template to process its input arguments recursively at compile-time:

```
template<std::size_t index>
void mexAllocateHelper(mxArray *) {
}

template<std::size_t index = 0, class Head, class ...Tail>
void mexAllocateHelper(mxArray *object[], const Head &head, Tail &&...tail) {
    object[index] = MexAllocator<Head>::allocate(head);
    mexAllocateHelper<index + 1>(object, std::forward<Tail>(tail)...);
}
```

In the implementation, the `mxArray` array at `index` is allocated using the appropriate specialization of `MexAllocator`. Next, the argument counter `index` is incremented by one so as to instantiate `mexAllocateHelper` for the next recursion level. The remaining input arguments are passed over to `mexAllocateHelper<index + 1>` using perfect forwarding. Based on this, the function `mexAllocate` can be implemented accepting a variable number of arguments to allocate an array of dynamic `mxArray`s managed by a `UniquePointer` to prevent hard-to-detect memory leaks:

```
template<class ...Types>
MexUniquePointer<sizeof...(Types)> mexAllocate(Types &&...arguments) {
    constexpr auto numberOfArguments = sizeof...(Types);
    MexUniquePointer<numberOfArguments> object(new mxArray
        ↪  *[numberOfArguments]);
    mexAllocateHelper(object.get(), std::forward<Types>(arguments)...);
    return object;
}
```

Note that a specialization for an empty argument list is required, due to the fact that an array of zero length is forbidden by the C++ language standard:

```
template<>
MexUniquePointer<0> mexAllocate();
```

Having discussed the processing of the input arguments, let us proceed with the conversion of the output arguments populated by `mexCallMATLAB`. To this end, it turns out useful to introduce a function `mexCopyTypedRange` that takes the address of an `mxArray` object and an `Iterator` the contents of the `mxArray` should be copied to:

```
template<class Type, class Iterator>
void mexCopyTypedRange(const mxArray * const object, Iterator result) noexcept {
    const auto source = static_cast<Type *>(mxGetData(object));
    const auto length = getMexObjectSize(object);
    std::copy(source, source + length, result);
}
```

Note that `mexCopyTypedRange` expects the C++ data type as an additional argument to be able to cast the result of `mxGetData` to a pointer of correct size. Following that, `mexCopyTypedRange` can be employed to define `mexCopyRange`, which has a similar signature as `mexCopyTypedRange` but does not require the `Type` template argument:

```
template<class Iterator>
void mexCopyRange(const mxArray * const object, Iterator result) {
    switch (mxGetClassID(object)) {
#define MEX_TYPE_DEFINE(mexType, cppType) case mexType: \
    mexCopyTypedRange<cppType>(object, result); return;
#include "mex_type.def"
#undef MEX_TYPE_DEFINE
    default:
        throw std::runtime_error("Unknown primitive data type.");
    }
}
```

Instead, a macro and the file `mex_type.def` are used to generate a `switch-case` statement required for the conversion of an `mxArray` object at runtime. The function `mexCopyRange` is then used by specializations of the class

```
template<class Type, class Enable = void>
struct MexConverter {
};
```

For instance, consider the specialization of the `MexConverter` class for `Lists` of arithmetic types (`bool`, `int`, `double`, ...):

```
template<class Type, class Allocator>
struct MexConverter<List<Type, Allocator>,
        typename std::enable_if<std::is_arithmetic<Type>::value>::type> {
    static List<Type, Allocator> convert(const mxArray * const object) {
        List<Type, Allocator> list;
        mexCopyRange(object, std::back_inserter(list));
        return list;
    }
};
```

Next, let us introduce a `mexConvertHelper` function that recursively converts arrays of `mxArrays` to C++ objects stored in a `Tuple<Types...>` of possibly different types:

```
template<std::size_t index>
void mexConvertHelper(const mxArray * const *) {
}

template<std::size_t index = 0, class ...Types>
typename std::enable_if<index < sizeof...(Types), void>::type
mexConvertHelper(const mxArray * const *object, Tuple<Types...> &tuple) {
    std::get<index>(tuple) = MexConverter<typename std::tuple_element<index,
        Tuple<Types...> >::type>()(object[index]);
    mexConvertHelper<index + 1, Types...>(object, tuple);
}
```

Finally, it is possible to define the function

```
template<class Type>
Type mexConvert(const mxArray * const *object) {
    return MexConverter<Type>::convert(*object);
}
```

for single return types, as well as a variant for multiple return types relying on `mexConvertHelper`:

```
template<class First, class Second, class ...Tail>
Tuple<First, Second, Tail...> mexConvert(const mxArray * const *object) {
    Tuple<First, Second, Tail...> tuple;
    mexConvertHelper(object, tuple);
    return tuple;
}
```

In order to reserve space for the output arguments populated by the `mexCallMATLAB` function, the function

```
template<std::size_t numberOfOutputs>
MexUniquePointer<numberOfOutputs> mexCreateOutput() noexcept {
    return MexUniquePointer<numberOfOutputs>(new mxArray *[numberOfOutputs]);
}
```

is introduced. For the case that no output arguments are returned by the called MATLAB/Octave function, `mexCreateOutput` must be specialized to avoid the creation of an array of zero length, which is forbidden by the C++ language standard:

```
template<>
MexUniquePointer<0> mexCreateOutput<0>() noexcept {
    return nullptr;
}
```

Further, we implement the function

```
template<std::size_t numberOfOutputs, class ...InputTypes>
MexUniquePointer<numberOfOutputs> mexCallHelper(const String &functionName,
        InputTypes &&...cppInput) {
    static constexpr std::size_t numberOfInputs { sizeof...(cppInput) };
    auto mexInput = mexAllocate(std::forward<InputTypes>(cppInput)...);
    auto mexOutput = mexCreateOutput<numberOfOutputs>();
    mexCallMATLAB(numberOfOutputs, mexOutput.get(), numberOfInputs,
            mexInput.get(), functionName.c_str());
    return mexOutput;
}
```

which generates the input arguments as expected by `mexCallMATLAB`, calls the MATLAB/Octave function, and returns the output arguments wrapped in a `MexUniquePointer`. Building on this, the function `mexCall` without any output arguments reads

```
template<class ...InputTail>
void mexCall(const String &functionName, InputTail &&...cppInput) {
    mexCallHelper<0>(functionName, std::forward<InputTail>(cppInput)...);
}
```

For a single return value, the function

```
template<class OutputType, class ...InputTail>
OutputType mexCall(const String &functionName, InputTail &&...cppInput) {
    static constexpr std::size_t numberOfOutputs { 1u };
    return mexConvert<OutputType>(
            mexCallHelper<numberOfOutputs>(functionName,
                std::forward<InputTail>(cppInput)...).get());
}
```

is provided. Multiple return values are packed into a variadic tuple:

```
template<class First, class Second, class ... OutputTail, class ...InputTail>
Tuple<First, Second, OutputTail...> mexCall(const String &functionName,
        InputTail &&... cppInput) {
    static constexpr std::size_t numberOfOutputs { sizeof...(OutputTail) + 2 };
    return mexConvert<First, Second, OutputTail...>(
        mexCallHelper<numberOfOutputs>(functionName,
            std::forward<InputTail>(cppInput)...).get());
}
```

In the implementation of the C++ wrappers for the MATLAB/Octave driver functions, `mexCall` can then conveniently and safely be used as follows:

```
const auto output = mexCall<int, DoubleDynamicArray>("function",
    DoubleDynamicArray { 1.0, 2.0, 3.0 });
```

Note that `mexCall` reduces the following tasks to a single call: type-safe conversion of C++ data structures to their MATLAB/Octave equivalents, invocation of a MATLAB/Octave function, and automatic conversion of MATLAB/Octave data structures back to C++. This way, the data exchange across the language barrier in the driver is significantly simplified. In addition, type-safety and maintainability of the driver code are increased as well.

5.6.5 Python Solvers

For Python solvers, *comana* provides a Python interface to the interrupt functions in `comana_interrupt.py`:

```
from ctypes import CDLL, RTLD_GLOBAL, c_int

solver_adapter_library = CDLL("<solver_adapter_library>", mode = RTLD_GLOBAL)

# Define arguments to C interrupt functions
solver_adapter_library.comana_initialize_coupling.argtypes = []
solver_adapter_library.comana_do_transfer.argtypes = [c_int]
solver_adapter_library.comana_exit_coupling.argtypes = []

# Define Python interrupt functions
def comana_initialize_coupling():
    solver_adapter_library.comana_initialize_coupling()

def comana_do_transfer(stage):
    return solver_adapter_library.comana_do_transfer(stage)

def comana_exit_coupling():
    solver_adapter_library.comana_exit_coupling()
```

Python comes with the `ctypes` module, which allows to interface a shared library from a Python script. The `CDLL` command loads the shared library `<solver_adapter_library>` and creates the object `solver_adapter_library` through which the library functions can be called. Note that `<solver_adapter_library>` is to be substituted by the specific

adapter library name. In the adapter library, the function names all have C linkage, and the function arguments are hence *not* encoded in the function symbol. It is therefore required to define the arguments to the library functions. For convenient use in a Python script, the rather lengthy interrupt functions are wrapped by another layer of ordinary Python functions.

Regarding the data transfer between the solver and the driver, data must be passed across a language barrier between C++ and Python. Although there are readily available interfaces to transfer even complex objects and data structures from C++ to Python (see [15], for instance), we prefer to provide our own interface to keep it as small and as simple as possible.

First, let us consider the transfer of C++ data structures to Python. It proves useful to introduce a generic `PythonAllocator` template that can then be partially specialized for different data types:

```
template<class Type, class Enable = void>
struct PythonAllocator {
};
```

In some situations, it is convenient to leverage SFINAE to conditionally enable a struct for a certain group of types. For floating point types such as `float` and `double`, for instance, `PythonAllocator` can be specialized as follows:

```
template<class Type>
struct PythonAllocator<Type,
        typename std::enable_if<std::is_floating_point<Type>::value>::type> {
    static PyObject * allocate(const Type value) noexcept {
        return PyFloat_FromDouble(value);
    }
};
```

As a counterpart for the `PythonAllocator` class, we introduce the `PythonDeallocator` class:

```
struct PythonDeallocator {
    void operator()(PyObject *object) noexcept;
};
```

`PythonDeallocator` provides the `operator()` member function that takes a `PyObject` for destruction. In the implementation, the reference count for that object is decremented to signal the Python memory manager that the object is no longer needed and that any associated resources can be released back to the system. In order to avoid memory leaks, any dynamically allocated Python object should be managed by a smart pointer. For the proper release of memory upon destruction of a Python object, `PythonDeallocator` is used as a custom deleter:

```
using PythonUniquePointer = UniquePointer<PyObject, PythonDeallocator>;
```

When calling a Python function from C++, the input arguments are passed as a Python tuple. Introducing `pythonAllocateHelper`, a variable number of arguments can be converted to Python objects and stored in a Python tuple:

```
template<std::size_t index>
void pythonAllocateHelper(PyObject *) noexcept {
}

template<std::size_t index = 0, class Head, class ...Tail>
void pythonAllocateHelper(PyObject *object, const Head &head,
        Tail &&...tail) noexcept {
    PyTuple_SetItem(object, index, PythonAllocator<Head>::allocate(head));
    pythonAllocateHelper<index + 1>(object, std::forward<Tail>(tail)...);
}
```

Note the use of a variadic template to achieve compile-time recursion. Here, `index` acts as a counter to indicate the position of an allocated Python object in the tuple. Appropriate specializations of the `PythonAllocator` class are automatically instantiated based on the type of `head`. Further arguments to `pythonAllocateHelper` are passed as universal references, which are then recursively passed to `pythonAllocateHelper` using perfect forwarding. Now, a `pythonAllocate` function can be implemented as follows:

```
template<class ...Types>
PyObject * pythonAllocate(Types &&...arguments) noexcept {
    auto object = PyTuple_New(sizeof...(Types));
    pythonAllocateHelper(object, std::forward<Types>(arguments)...);
    return object;
}
```

Accepting a variable number of arguments as universal references, the function creates a Python tuple and uses `pythonAllocateHelper` to fill the tuple.

Next, the conversion from Python data structures to C++ needs to be discussed. Similar to the `PythonAllocator`, we introduce a `PythonConverter` class, which may be specialized for different data types:

```
template<class Type, class Enable = void>
struct PythonConverter {
};
```

For instance, consider the conversion of a Python floating point number to a C++ floating point number of type `float`, `double`, or `long double`:

```
template<typename Type>
struct PythonConverter<Type,
        typename std::enable_if<std::is_floating_point<Type>::value>::type> {
    static Type convert(PyObject *object) {
        if (PyFloat_Check(object))
            return PyFloat_AsDouble(object);
        else
            throw std::runtime_error(
                "Invalid conversion to floating point type.");
    }
};
```

If a Python function returns a tuple instead of a single return value, the Python tuple is converted to a C++ tuple. To this end, it is again convenient to implement a helper function to convert the tuple elements recursively and to store them in a C++ tuple:

```
template<std::size_t index = 0, class ...Types>
void pythonConvertHelper(PyObject *object, Tuple<Types...> &tuple) {
    using Type = typename std::tuple_element<index, Tuple<Types...>>::type;
    std::get<index>(tuple) = PythonConverter<Type>::convert(
        PyTuple_GetItem(object, index));
    pythonConvertHelper<index + 1, Types...>(object, tuple);
}
```

The variadic function template accepts a Python tuple object and a C++ tuple of elements of possibly different types. Inferring the type of the tuple element at position `index` becomes possible using `std::tuple_element`. Appropriate specializations of `PythonConverter` are instantiated to convert the Python tuple element to C++; the result of the conversion is then assigned to the corresponding tuple element. The next level of recursion is started by incrementing the positional index by one. For the conversion of a single return value, a first variant of the function `pythonConvert` is implemented as follows:

```
template<class Type>
Type pythonConvert(PyObject *object) {
    return PythonConverter<Type>::convert(object);
}
```

Multiple return values are stored in a C++ tuple, and a second variant of `pythonConvert` hence needs to be supplied:

```
template<class First, class Second, class ...Tail>
Tuple<First, Second, Tail...> pythonConvert(PyObject *object) {
    Tuple<First, Second, Tail...> tuple;
    pythonConvertHelper(object, tuple);
    return tuple;
}
```

Combining the functions `pythonAllocate` and `pythonConvert`, we can implement a function `pythonCall` to call a Python function and handle all the necessary data conversion internally. Before this function is discussed, let us consider the variadic function template `pythonCallHelper`:

```
template<class ...InputTypes>
PythonUniquePointer pythonCallHelper(const String &moduleName,
        const String &functionName, InputTypes &&...cppInput) {
    auto state = pythonEnsureState();
    const auto function = pythonLoad(moduleName, functionName);
    const auto pythonInput = pythonAllocate(
        std::forward<InputTypes>(cppInput)...);
    const auto pythonOutput = PyObject_CallObject(function.get(),
        pythonInput.get());
    pythonReleaseState(state);
    return pythonOutput;
}
```

It accepts the mandatory Python module and function name and a variable number of further arguments of possibly different type as input arguments. Before a Python function

is called, `pythonEnsureState` ensures that the Python thread is ready to call the Python C API, irrespective of the current state of Python. Next, the Python function is loaded by supplying the module and function name to `pythonLoad`. Subsequently, a Python tuple object is created by means of `pythonAllocate`. Note again the beneficial use of perfect forwarding. Following that, the Python function is called and, after the Python thread is released by `pythonReleaseState`, the resulting Python object is returned. Based on this, three different versions of the `pythonCall` function can be implemented. Firstly, the version without any output arguments:

```
template<class ...InputTypes>
void pythonCall(const String &moduleName, const String &functionName,
        InputTypes &&...cppInput) {
    pythonCallHelper(moduleName, functionName,
        std::forward<InputTypes>(cppInput)...);
}
```

Secondly, the version with a single output argument:

```
template<class OutputType, class ...InputTypes>
OutputType pythonCall(const String &moduleName, const String &functionName,
        InputTypes &&...cppInput) {
    return pythonConvert<OutputType>(
            pythonCallHelper(moduleName, functionName,
                std::forward<InputTypes>(cppInput)...).get());
}
```

And, thirdly, a version with multiple output arguments wrapped in a C++ tuple:

```
template<class First, class Second, class ...OutputTail, class ...InputTypes>
Tuple<First, Second, OutputTail...> pythonCall(const String &moduleName,
        const String &functionName, InputTypes &&...cppInput) {
    return pythonConvert<First, Second, OutputTail...>(
            pythonCallHelper(moduleName, functionName,
                std::forward<InputTypes>(cppInput)...).get());
}
```

Now, the user is able to conveniently call a Python function from C++ without the need for manual data conversion, for instance:

```
auto output = pythonCall<int, DoubleDynamicArray>("module", "function",
    DoubleDynamicArray{ 1.0, 2.0, 3.0 });
```

5.6.6 APDL Solvers

The ANSYS software suite includes the scripting language *ANSYS parametric design language (APDL)* to interact with the ANSYS pre- and post-processing utilities and the ANSYS solver. To invoke the interrupt functions from an APDL script, the following interface is provided:

```
#ifdef __cplusplus
extern "C" {
#endif
int comana_initialize_coupling_(const char * const command);
int comana_do_transfer_(const char * const command);
int comana_exit_coupling_(const char * const command);
#ifdef __cplusplus
}
#endif
```

Note that, as opposed to the other language interfaces, the interrupt functions accept a `const char *` argument. In APDL, the entire command string is passed as an argument to the external command, and function arguments need to be processed manually. Unfortunately, APDL does not allow return values for external commands. Instead of returning the convergence status, `comana_do_transfer_` therefore defines the variable `convergence_status_`, which can then be used to query the convergence status in an APDL script. The underscore appended to the variable name serves to prevent name clashes with any existing variables. In order to inform ANSYS about the presence of a shared library providing external commands for the use in an APDL script, the file `ans_ext.tbl` needs to be visible to the ANSYS process. In addition to the full path to the shared library and the function symbol, it is also necessary to define a shorthand command preceded by a tilde representing the command in APDL:

```
<install_dir>/libansys_adapter.so ~initco comana_initialize_coupling_
<install_dir>/libansys_adapter.so ~dotran comana_do_transfer_
<install_dir>/libansys_adapter.so ~exitco comana_exit_coupling_
```

For the data transfer between the solver's database and the driver, thin wrappers around the ANSYS Fortran API are created to isolate ANSYS variable and function definitions from the C++ part of the driver, which does not only keep the C++ code from being cluttered by third-party code, but also simplifies modular testing.

5.7 Simulation Setup

In order to demonstrate the use of *comana* for the partitioned analysis of a coupled multifield problem, we once again consider the simple mechanical system introduced in Section 4.3. Splitting this system into two subsystems A and B, the equations

$$\mathbf{f}_A = \begin{pmatrix} -k_1 d_{A,1} - c_1 \dot{d}_{A,1} + k_2(d_{A,2} - d_{A,1}) \\ +c_2(\dot{d}_{A,2} - \dot{d}_{A,1}) + F_1(t) \\ -k_2(d_{A,2} - d_{A,1}) - c_2(\dot{d}_{A,2} - \dot{d}_{A,1}) + k_{31}(d_{A,3} - d_{A,2}) \\ +k_{32}(d_{A,3} - d_{A,2})^2 + c_3(\dot{d}_{A,3} - \dot{d}_{A,2}) + F_2(t) \\ -k_{31}(d_{A,3} - d_{A,2}) - k_{32}(d_{A,3} - d_{A,2})^2 \\ -c_3(\dot{d}_{A,3} - \dot{d}_{A,2}) - k_4 d_{A,3} - c_4 \dot{d}_{A,3} + F_3(t) \end{pmatrix} = \mathbf{f}_A(t, \mathbf{d}_A, \dot{\mathbf{d}}_A) \,, \tag{5.1}$$

$$\mathbf{M}\ddot{\mathbf{d}}_B = \begin{pmatrix} m_1 & 0 & 0 \\ 0 & m_2 & 0 \\ 0 & 0 & m_3 \end{pmatrix} \begin{pmatrix} \ddot{d}_{B,1} \\ \ddot{d}_{B,2} \\ \ddot{d}_{B,3} \end{pmatrix} = \begin{pmatrix} f_{B,1} \\ f_{B,2} \\ f_{B,3} \end{pmatrix} = \mathbf{f}_B \,.$$

are acquired, see also Equation (4.8) and (4.9). In addition, the interface conditions

$$\mathbf{d}_A = \mathbf{d}_B \,, \quad \dot{\mathbf{d}}_A = \dot{\mathbf{d}}_B \,, \quad \text{and} \quad \mathbf{f}_A = \mathbf{f}_B \tag{5.2}$$

and the initial conditions

$$\mathbf{d}_B(t=0) = \mathbf{d}_{B,0} \,, \quad \dot{\mathbf{d}}_B(t=0) = \dot{\mathbf{d}}_{B,0} \,, \quad \text{and } \ddot{\mathbf{d}}_B(t=0) = \ddot{\mathbf{d}}_{B,0} \tag{5.3}$$

are prescribed. Building on a partitioned solution approach, we iteratively, within a time increment, prescribe the displacement $\mathbf{d}_A$ and velocity $\dot{\mathbf{d}}_A$ in subsystem A, evaluate the displacement-, velocity- and time-dependent force $\mathbf{f}_A(t, \mathbf{d}_A, \dot{\mathbf{d}}_A)$, pass the result over to subsystem B, solve for the displacement $\mathbf{d}_B$ and the velocity $\dot{\mathbf{d}}_B$, and repeat the cycle all over again.

For the numerical treatment of the subsystems A and B, we use the dedicated force solver $\mathcal{S}_A$ and the motion solver $\mathcal{S}_B$. The force solver $\mathcal{S}_A$ comprises the header file `force_solver.h`

```
struct Problem {
    DoubleArray3 displacement, velocity, force;
};
```

and the implementation file `force_solver.cpp`

```
#include "force_solver.h"

// Time-dependent force contribution
double timeDependentForce(const double a0, const double a1, const double b,
        const double f, const double time) noexcept {
    return a0 * std::sin(2 * M_PI * f * time) * std::sin(2 * M_PI
        * (b + a1 * std::sin(2 * M_PI * time)) * time);
}

int main() {
    // Transient analysis parameters
    constexpr auto stopTime = 10., timeStepSize = 5e-3;
    constexpr std::size_t numberOfTimeSteps = stopTime / timeStepSize;
    auto time = 0.;

    // Data structure storing displacement, velocity, and force
    Problem problem;

    // Spring, damper, and external force constants
    constexpr auto k0 = 5e2, k1 = 2.4e2, k21 = 1.8e2, k22 = 1.2, k3 = 3.5e2;
    constexpr auto c0 = 1.5e-2, c1 = 1e-1, c2 = 8e-2, c3 = 5e-2;
    constexpr auto a00 = 50., a10 = .05, b0 = 4., f0 = 1.1;
    constexpr auto a01 = 8.5, a11 = .12, b1 = 4., f1 = 2.4;
    constexpr auto a02 = 16., a12 = .1, b2 = 10., f2 = 1.6;

    // Loop through time steps
    for (std::size_t timeStep = 0; timeStep < numberOfTimeSteps;
            timeStep++, time += timeStepSize) {
```

```
        problem.force[0] =
            - k0 * problem.displacement[0]
            - c0 * problem.velocity[0]
            + k1 * (problem.displacement[1] - problem.displacement[0])
            + c1 * (problem.velocity[1] - problem.velocity[0])
            + timeDependentForce(a00, a10, b0, f0, time);
        problem.force[1] =
            - k1 * (problem.displacement[1] - problem.displacement[0])
            - c1 * (problem.displacement[1] - problem.displacement[0])
            + k21 * (problem.displacement[2] - problem.displacement[1])
            + k22 * std::pow(problem.displacement[2]
                - problem.displacement[1], 2)
            + c2 * (problem.velocity[2] - problem.velocity[1])
            + timeDependentForce(a01, a11, b1, f1, time);
        problem.force[2] =
            - k21 * (problem.displacement[2] - problem.displacement[1])
            - k22 * std::pow(problem.displacement[2]
                - problem.displacement[1], 2)
            - c2 * (problem.velocity[2] - problem.velocity[1])
            - k3 * problem.displacement[2] - c3 * problem.velocity[2]
            + timeDependentForce(a02, a12, b2, f2, time);
    }
}
```

For the motion solver $\mathcal{S}_B$, we have the header file `motion_solver.h`

```
struct Problem {
    DoubleVector3 displacement {{ 0, 0.75, 0.5 }};
    DoubleVector3 velocity {{ 0, 0, 0 }};
    DoubleVector3 acceleration {{ 5.06250e1, -1.85156e1, 7.81250e-1 }};
    DoubleVector3 force;
};
```

and the source file `motion_solver.cpp`, which implements the linear version of the Newmark time integration scheme (3.112):

```
#include "motion_solver.h"

int main() {
    // Transient analysis parameters
    constexpr auto stopTime = 10., timeStepSize = 5e-3;
    constexpr std::size_t numberOfTimeSteps = stopTime / timeStepSize;
    auto time = 0.;

    // Newmark parameters and integration constants
    constexpr auto beta = .25, gamma = .5;
    constexpr auto alpha0 = 1 / (beta * timeStepSize * timeStepSize),
        alpha2 = 1 / (beta * timeStepSize), alpha3 = 1 / (2 * beta) - 1,
        alpha6 = timeStepSize * (1 - gamma), alpha7 = gamma * timeStepSize;

    // Data structure storing displacement, velocity, acceleration, and force
```

```
    Problem problem;

    // Masses
    constexpr DoubleVector3 masses {{ 2., 8., 4. }};

    auto previousDisplacement = problem.displacement;
    auto previousVelocity = problem.velocity;
    auto previousAcceleration = problem.acceleration;

    // Loop through time steps
    for (std::size_t timeStep = 0; timeStep < numberOfTimeSteps;
            timeStep++, time += timeStepSize) {
        const auto effectiveForce = problem.force + masses * (alpha0
            * previousDisplacement + alpha2 * previousVelocity + alpha3
            * previousAcceleration);

        problem.displacement = effectiveForce / (alpha0 * masses);

        problem.acceleration = alpha0 * (problem.displacement
            - previousDisplacement) - alpha2 * previousVelocity
            - alpha3 * previousAcceleration;
        problem.velocity = previousVelocity + alpha6 * previousAcceleration
            + alpha7 * problem.acceleration;

        previousDisplacement = problem.displacement;
        previousVelocity = problem.velocity;
        previousAcceleration = problem.acceleration;
    }
}
```

In order to integrate these solvers into a partitioned solution strategy, some modifications are required.

For the force solver $\mathcal{S}_A$, we insert the lines highlighted in yellow:

```
#include "force_solver.h"
#include "force_solver_adapter.h"

int main() {
    // Transient analysis parameters
    constexpr auto stopTime = 10., timeStepSize = 1e-3;
    constexpr std::size_t numberOfTimeSteps = stopTime / timeStepSize;
    auto time = 0.;

    // Data structure storing displacement, velocity, and force
    Problem problem;

    // Spring, damper, and external force constants...

    force_solver_initialize_driver(problem);
    comana_initialize_coupling();
```

```
    // Loop through time steps
    for (std::size_t timeStep = 0; timeStep < numberOfTimeSteps;
            timeStep++, time += timeStepSize) {
        while (true) {
            comana_do_transfer(0);
            problem.force[0] = // ...
            problem.force[1] = // ...
            problem.force[2] = // ...
            if (comana_do_transfer(1))
                break;
        }
    }

    comana_exit_coupling();
    force_solver_clear_driver(problem);
}
```

As the first modification, the header `force_solver_adapter.h` is included, which again includes `comana_interrupt.h` and `force_solver_driver_setup.h`. The first header `comana_interrupt.h` was already introduced in Section 5.6.2. It provides the C/C++ interface to the generic implementations of the interrupt functions `comana_initialize_coupling`, `comana_do_transfer`, and `comana_exit_coupling`. The second header comprises the interface to the functions initializing and clearing the global driver data structures:

```
#include "force_solver_driver_setup.h"
#include "force_solver_global.h"

void force_solver_initialize_driver(Problem &problem) {
    const auto &forceSolverGlobal = getForceSolverGlobal();
    forceSolverGlobal.problem = &problem;
}

void force_solver_clear_driver() {
}
```

In `force_solver_initialize_driver`, the function `getForceSolverGlobal` returns the global data structure `forceSolverGlobal` responsible for retaining the driver's state across several external function calls. The address of the `problem` object is assigned to the `problem` field of the `ForceSolverGlobal` instance. As no dynamic memory was allocated upon driver initialization in `force_solver_clear_driver`, no cleanup is required. For reasons of completeness, let us consider the header `force_solver_global.h`:

```
struct Problem;

struct ForceSolverGlobal {
    Problem *problem;
};
```

```
ForceSolverGlobal & getForceSolverGlobal() noexcept;
```

In the first line, we forward-declare the `Problem` struct. Equipped with a pointer to a `Problem` instance initialized upon the call to `force_solver_initialize_driver`, a `ForceSolverGlobal` instance is constructed as a local static object in the `getForceSolverGlobal` on first access following the RAII idiom. In the implementation of the driver functions, this global data structure is required to access the `problem` instance instantiated in the force solver's `main` function before being passed on to `force_solver_initialize_driver` through its memory address:

```
#include "force_solver_driver.h"
#include "force_solver_global.h"

void force_solver_set_displacement(const DoubleArray3 &displacement) noexcept {
    const auto &forceSolverGlobal = getForceSolverGlobal();
    forceSolverGlobal.displacement = displacement;
}

void force_solver_set_velocity(const DoubleArray3 &velocity) noexcept {
    const auto &forceSolverGlobal = getForceSolverGlobal();
    forceSolverGlobal.velocity = velocity;
}

DoubleArray3 force_solver_get_force() noexcept {
    const auto &forceSolverGlobal = getForceSolverGlobal();
    return forceSolverGlobal.force;
}
```

Similar to the force solver, the motion solver $\mathcal{S}_B$ is modified as well – as indicated by the lines highlighted in yellow:

```
#include "motion_solver.h"
#include "motion_solver_adapter.h"

int main() {
    // Transient analysis parameters
    constexpr auto stopTime = 10., timeStepSize = 1e-3;
    constexpr std::size_t numberOfTimeSteps = stopTime / timeStepSize;
    auto time = 0.;

    // Newmark parameters and integration constants...

    // Data structure storing displacement, velocity, acceleration, and force
    Problem problem;

    // Masses
    constexpr DoubleVector3 masses {{ 2., 8., 4. }};

    auto previousDisplacement = problem.displacement;
```

```
    auto previousVelocity = problem.velocity;
    auto previousAcceleration = problem.acceleration;

    motion_solver_initialize_driver(problem);
    comana_initialize_coupling();

    // Loop through time steps
    for (std::size_t timeStep = 0; timeStep < numberOfTimeSteps;
            timeStep++, time += timeStepSize) {
        while (true) {
            comana_do_transfer(0);
            const auto effectiveForce = problem.force + masses * (alpha0
                * previousDisplacement + alpha2 * previousVelocity + alpha3
                * previousAcceleration);

            problem.displacement = effectiveForce / (alpha0 * masses);

            problem.acceleration = alpha0 * (problem.displacement
                - previousDisplacement) - alpha2 * previousVelocity
                - alpha3 * previousAcceleration;
            problem.velocity = previousVelocity + alpha6 * previousAcceleration
                + alpha7 * problem.acceleration;

            if (comana_do_transfer(1))
                break;
        }

        previousDisplacement = problem.displacement;
        previousVelocity = problem.velocity;
        previousAcceleration = problem.acceleration;
    }

    comana_exit_coupling();
    motion_solver_clear_driver(problem);
}
```

Strong similarities to the changes conducted for the force solver are not mistakable. Since the implementation of the functions `motion_solver_initialize_driver` and `motion_solver_clear_driver` and the driver functions is completely analogous to those for the force solver $\mathcal{S}_A$, it is omitted here.

Functions to initialize the driver data structures as well as the driver functions themselves are compiled into the adapter libraries `libforce_solver_adapter.so` and `libmotion_solver_adapter.so` and then linked to the corresponding executables `force_solver` and `motion_solver`. This completes the modifications required to use these solvers in a partitioned analysis.

For the implementation of the partitioned solution procedure to solve the simple mechanical system from Section 4.3, a dedicated C++ program needs to be written and compiled into an executable. Although this involves some programming, the implemen-

tation is hardly more difficult than setting up an input file as required in other software packages implementing the partitioned solution approach. On the contrary, due to the fact that all the constructs available in the C++ language can be used, the simulation setup often even becomes simpler. In the first step, a file `mass_spring_damper_system.cpp` is created, and the *comana* header is included:

```
#include "comana/kernel/comana.h"
```

Functions and classes from the `Comana` namespace can be brought into the current scope to avoid typing by using

```
using namespace Comana;
```

Following that, we open the `main` function:

```
int main() {
```

Next, the master socket groups `forceSolver` and `motionSolver`, which are used to communicate to the slave process groups `forceSolver` and `motionSolver`, are created:

```
    MasterSocketGroup forceSolver("forceSolver"),
        motionSolver("motionSolver");
```

Note that the names of the slave process groups do not necessarily need to be identical to the file names of the solver executables or to the variable names of the master socket groups. Upon construction of the master socket groups, the connections to the slave process groups are established. Subsequently, we create the patch group `springsDampersAndForces`, which binds the master socket group `forceSolver` to the mesh label and the topology of the patch group representing the coupling surface in the force solver:

```
    PatchGroup springsDampersAndForces(forceSolver, "", Topology::point);
```

The first argument to the `PatchGroup` constructor is the master socket group, the second one designates the mesh label of the patch. In this particular case, the mesh label is an empty string, which is due to the fact that the force solver does not support addressing patches by their mesh label – and there is just a single one. As a third argument, we pass the topology of the patch. Since the coupling surface is one-dimensional – springs and dampers and masses are only connected at points – we choose `Topology::point`. Next, the patch group must be initialized using

```
    Request::initializePatchGroup(springsDampersAndForces);
```

Thereafter, the patch group in the motion solver is set up and initialized by

```
    PatchGroup masses(motionSolver, "", Topology::point);
    Request::initializePatchGroup(masses);
```

In order to start the coupled solution process, we need to request the initial condition for the displacement $\mathbf{d}_B$ and the velocity $\dot{\mathbf{d}}_B$ from the motion solver:

```
    auto displacement = Request::getField(masses, Location::vertex,
        QuantityType::displacement); // d^0_{B,0}
    auto velocity = Request::getField(masses, Location::vertex,
        QuantityType::velocity); // ḋ^0_{B,0}
```

During the solution process, we will iteratively modify the components of the displacement and the velocity field to equilibrate the subfields with each other. The convergence acceleration schemes applied to speedup the convergence operate on references to the displacement and velocity field's components:

```
auto &unmodifiedDisplacementComponents = displacement.getComponents();
auto modifiedDisplacementComponents = unmodifiedDisplacementComponents;

auto &unmodifiedVelocityComponents = velocity.getComponents();
auto modifiedVelocityComponents = unmodifiedVelocityComponents;
```

Now, we create polynomial predictors to obtain reasonable initial guesses for the displacement and the velocity at the beginning of each time increment:

```
PolynomialPredictor displacementPredictor(2), velocityPredictor(2);
```

In order to check whether the subfields are equilibrated with each other up to an acceptable tolerance, the convergence criteria

```
ConvergenceCriterion absoluteConvergenceCriterion(1e-12); // ε_abs = 10^-12
ConvergenceCriterion relativeConvergenceCriterion(1e-6); // ε_rel = 10^-6
```

are required. For convergence acceleration, we use the quasi-Newton least squares method introduced in Section 4.7.10:

```
QuasiNewtonLeastSquaresMethod displacementSolutionUpdate,
    velocitySolutionUpdate;
```

In this example, the simulation runs for $T = 10\,\mathrm{s}$ in steps of $\Delta t = 5 \cdot 10^{-3}\,\mathrm{s}$. To control the solution process, we create a run info object

```
RunInfo runInfo(0, 10, 5e-3);
```

After that, we enter the coupled solution procedure:

```
while (runInfo.run()) {
```

At the beginning of each time step, we predict the displacement and the velocity to obtain a good starting value for the inner iteration. The fields' components are modified in-place to avoid unnecessary reallocations:

```
    predictor.predict(runInfo,
        modifiedDisplacementComponents); // d̃^k_{B,j+1} := P_d(d_{B,j})
    predictor.predict(runInfo,
        modifiedVelocityComponents); // ḋ̃^k_{B,j+1} := P_ḋ(ḋ_{B,j})
```

Then, we enter the implicit iteration:

```
    while (runInfo.iterate()) {
```

Subsequently, we instruct the force solver to apply the displacement $\mathbf{d}^k_{A,j+1} = \tilde{\mathbf{d}}^k_{B,j+1}$ and the velocity $\dot{\mathbf{d}}^k_{A,j+1} = \tilde{\dot{\mathbf{d}}}^k_{B,j+1}$ as an updated boundary condition, solve for the resulting force $\mathbf{f}^k_{A,j+1}$, and pass it back to the master process:

```
Request::setField(springsDampersAndForces, Location::vertex,
    QuantityType::displacement, displacement); // d^k_{A,j+1} = d̃^k_{B,j+1}
Request::setField(springsDampersAndForces, Location::vertex,
    QuantityType::velocity, displacement); // ḋ^k_{A,j+1} = d̃̇^k_{B,j+1}
Request::proceed(forceSolver);
auto force = Request::getField(springsDampersAndForces,
    Location::vertex, QuantityType::force); // f^k_{A,j+1}
```

In turn, the motion solver is supplied with the updated force $\mathbf{f}^k_{B,j+1} = \mathbf{f}^k_{A,j+1}$ used to solve for the displacement $\mathbf{d}^k_{B,j+1}$ and the velocity $\dot{\mathbf{d}}^k_{B,j+1}$ at the end of the current time increment. The result is then sent over to the master process:

```
Request::setField(masses, Location::vertex, QuantityType::force,
    force); // f^k_{B,j+1} = f^k_{A,j+1}
Request::proceed(motionSolver);
displacement = Request::getField(masses, Location::vertex,
    QuantityType::displacement); // d^k_{B,j+1}
velocity = Request::getField(masses, Location::vertex,
    QuantityType::velocity); // ḋ^k_{B,j+1}
```

Following that, the displacement residual $\mathbf{r}^k_{\boldsymbol{d},B,j+1} = \mathbf{d}^k_{B,j+1} - \tilde{\mathbf{d}}^k_{B,j+1}$ and the velocity residual $\mathbf{r}^k_{\dot{\boldsymbol{d}},B,j+1} = \dot{\mathbf{d}}^k_{B,j+1} - \tilde{\dot{\mathbf{d}}}^k_{B,j+1}$ are computed – and it is checked whether the absolute or the relative convergence criterion are fulfilled:

```
displacementResidual = unmodifiedDisplacementComponents
        - modifiedDisplacementComponents; // r^k_{B,j+1} := d^k_{B,j+1} - d̃^k_{B,j+1}
velocityResidual = unmodifiedVelocityComponents
        - modifiedVelocityComponents; // r^k_{ḋ,B,j+1} := ḋ^k_{B,j+1} - d̃̇^k_{B,j+1}

// if (‖r^k_{d,B,j+1}‖_2 < ε_abs) ∨ (‖r^k_{d,B,j+1}‖_2 / ‖r^0_{d,B,j+1}‖_2 < ε_rel) then
if (absoluteConvergenceCriterion.fulfilled(displacementResidual)
        || relativeConvergenceCriterion.fulfilled(runInfo,
            displacementResidual)) {
    Request::proceed(forceSolver);
    Request::proceed(motionSolver);
    break;
} else {
    Request::iterate(forceSolver);
    Request::iterate(motionSolver);

    // d̃^{k+1}_{B,j+1} := A_d (d̃^k_{B,j+1}, d^k_{B,j+1}, r^k_{d,B,j+1})
    displacementSolutionUpdate.updateSolution(runInfo,
            modifiedDisplacementComponents,
            unmodifiedDisplacementComponents,
            displacementResidual);

    // d̃̇^{k+1}_{B,j+1} := A_ḋ (d̃̇^k_{B,j+1}, ḋ^k_{B,j+1}, r^k_{ḋ,B,j+1})
    velocitySolutionUpdate.updateSolution(runInfo,
```

```
                        modifiedVelocityComponents,
                        unmodifiedVelocityComponents,
                        unmodifiedVelocityComponents
                            - modifiedVelocityComponents);
            }
```

If any of the convergence criteria fulfills the user-defined tolerances, the solvers are instructed to stop the implicit iteration and proceed to the next time increment. Otherwise, the solvers are informed that the subfields are not yet equilibrated with each other, and that at least another implicit iteration is required. Moreover, the solution updates modify $\tilde{\mathbf{d}}^{k}_{B,j+1}$ in-place to compute $\tilde{\mathbf{d}}^{k+1}_{B,j+1}$ and, analogously, $\tilde{\dot{\mathbf{d}}}^{k}_{B,j+1}$ to obtain $\tilde{\dot{\mathbf{d}}}^{k+1}_{B,j+1}$. Finally, we close the scope for the implicit iteration, the time stepping procedure, and the `main` function:

```
        }
    }
}
```

6 Benchmark Problems

In this chapter, we present several numerical examples in order to verify and illustrate the effectiveness of the partitioned solution approach. Depending on the particular problem under consideration, different spatial and temporal discretization schemes are employed for each of the subproblems. In order to stabilize and accelerate the solution procedure, the predictor and convergence acceleration schemes presented in Section 4.4 and 4.7 are applied. If not mentioned separately, a second-order polynomial predictor and the quasi-Newton least squares method are used, which have already been found to perform particularly well for the simple benchmark problem in Section 4.3. For the interpolation of the exchanged field quantities, a mesh-dependent interpolation according to Section 4.5.5 and 4.5.6 is performed unless stated otherwise. This ensures both an effective and accurate transfer of the relevant fields between the possibly non-conforming subfield discretizations. All problems are solved using our versatile software library *comana*, introduced in Chapter 5. Due to the modular architecture of *comana*, the appropriate subproblem solvers can be integrated into the partitioned solution strategy with little effort. A vast range of available algorithmic building blocks enables to easily steer and tune the solution procedure as necessary – and to exchange the predictor, interpolation technique, or convergence acceleration scheme if required.

6.1 Two-Dimensional Lid-Driven Cavity Flow with Flexible Bottom

In the first example, we consider a lid-driven cavity flow in a square domain of side length $L = H = 1\,\mathrm{m}$ with a flexible membrane of thickness $h = 2 \times 10^{-3}\,\mathrm{m}$ at the bottom, see Figure 6.1. This benchmark problem has already been discussed in [169, pp. 190 sq., 115, pp. 99–103 and 135–139, 166, pp. 160–162], for instance. For the fluid, we choose a density $\rho_\mathrm{f} = 10^3\,\mathrm{kg/m^3}$ and a kinematic viscosity $\nu_\mathrm{f} = 0.01\,\mathrm{m^2/s}$. For the membrane, we assume a St. Venant-Kirchhoff material with a density $\rho_\mathrm{s} = 500\,\mathrm{kg/m^3}$, Young's modulus $E = 250\,\mathrm{N/m^2}$, and Poisson's ratio $\nu_\mathrm{s} = 0$. At the top of the flow domain, a time-varying velocity $v_x = \bar{v} = (1 - \cos(2\pi t/5\,\mathrm{s}))\,\mathrm{m/s}$ is prescribed, whereas $v_y = 0\,\mathrm{m/s}$. At the inlet, the velocity v_x varies linearly from $v_x = 0\,\mathrm{m/s}$ at $y = 0.875\,\mathrm{m}$ to $v_x = \bar{v}$ at $y = 1\,\mathrm{m}$. At the outlet, a fixed reference pressure $p = 0\,\mathrm{Pa}$ is imposed. At the fixed walls and at the bottom membrane, we prescribe a no-slip condition, i.e. $v_x = v_y = 0\,\mathrm{m/s}$. The membrane is clamped at both ends, and $d_x = d_y = 0\,\mathrm{m}$ at $x = 0\,\mathrm{m}$ and $x = L$ is enforced.

The pseudo-three-dimensional fluid domain is discretized by $40 \times 40 \times 1$ hexahedral finite volumes and solved using the pimpleDyMFoam solver, which is part of the open-source CFD package OpenFOAM [123]. For the structure, we employ 16×1 geometrically nonlinear biquadratic plane-stress elements, available in the in-house FEM code AdhoC [41, 40]. For comparative purposes, the OpenFOAM pimpleDyMFoam solver is replaced by its equivalent from the OpenFOAM fork foam-extend [61] in a second study. Concerning

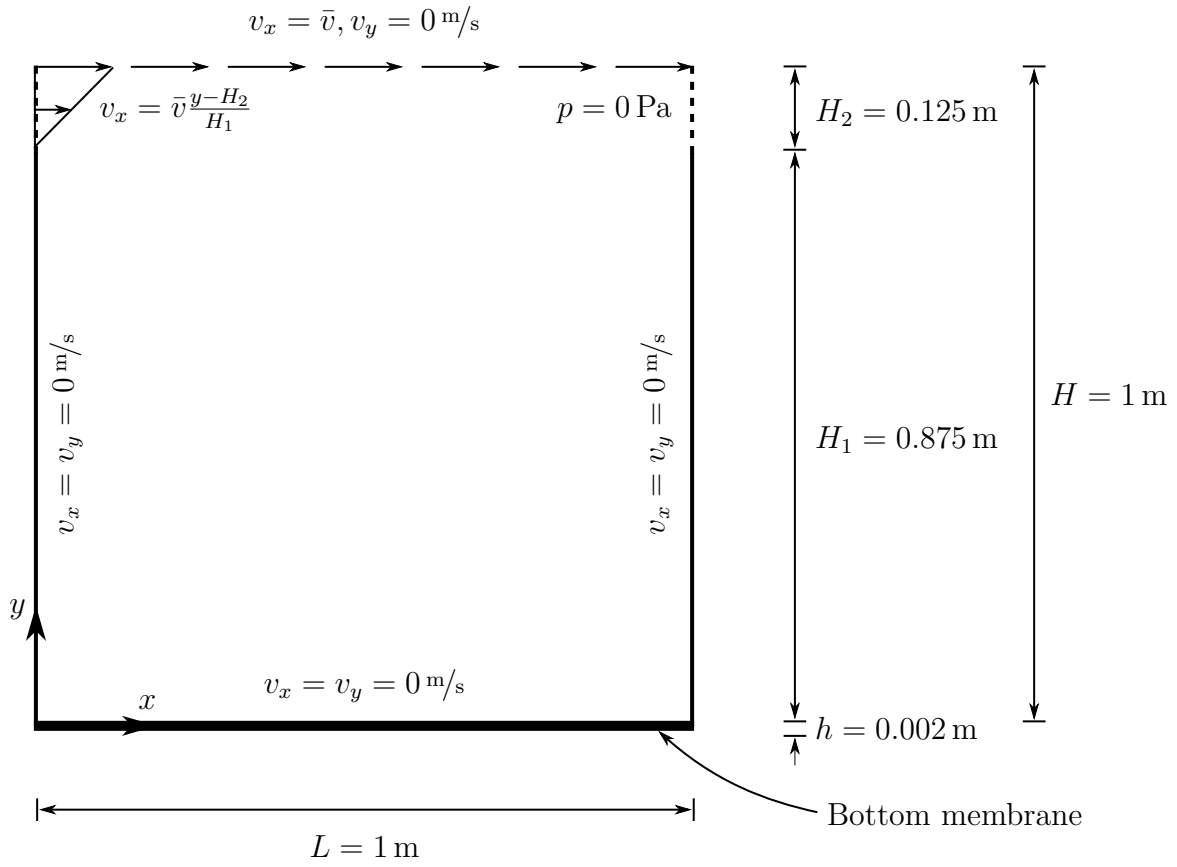


Figure 6.1: Geometry and boundary conditions for the two-dimensional lid-driven cavity flow with flexible bottom.

the temporal discretization, we apply the second-order accurate Euler backward method for the fluid problem, and the generalized-α scheme with spectral radius $\rho_\infty = 0.8$ for the structural field. The time step size is taken as $\Delta t = 0.01\,\mathrm{s}$, kept constant throughout the simulation. In the coupled solution procedure, the absolute convergence criterion $\|\mathbf{r}_{j+1}^k\|_2/\sqrt{n} < \varepsilon_{\mathrm{abs}} = 10^{-9}$ is applied to check whether the fluid and the structural field are equilibrated to sufficient accuracy, where n denotes the number of degrees of freedom at the fluid-structure interface.

Snapshots of the pressure and the velocity field at time $t = 35.39\,\mathrm{s}$ and $t = 69.77\,\mathrm{s}$ are shown in Figure 6.2. The vertical displacement at the bottom midpoint is depicted in Figure 6.3, showing a fair agreement with the reference results reported in [115, p. 135, 166, p. 161].

In the next step, we investigate the performance of the various convergence acceleration schemes proposed in Section 4.7. For this study, the convergence tolerance is further tightened to $\varepsilon_{\mathrm{abs}} = 10^{-15}$. This way, many iterations per time increment will be necessary until the convergence criterion is fulfilled – and it is especially the convergence acceleration schemes which cannot be applied in every coupling iteration that have better chances to unveil their potential to accelerate the solution process. In order to compare the convergence acceleration schemes with each other, the average number of iterations $\bar{k}$ per time increment is computed by accumulating the iterations in each time increment and dividing by the total number of time increments afterwards. Evidently, this is a direct measure

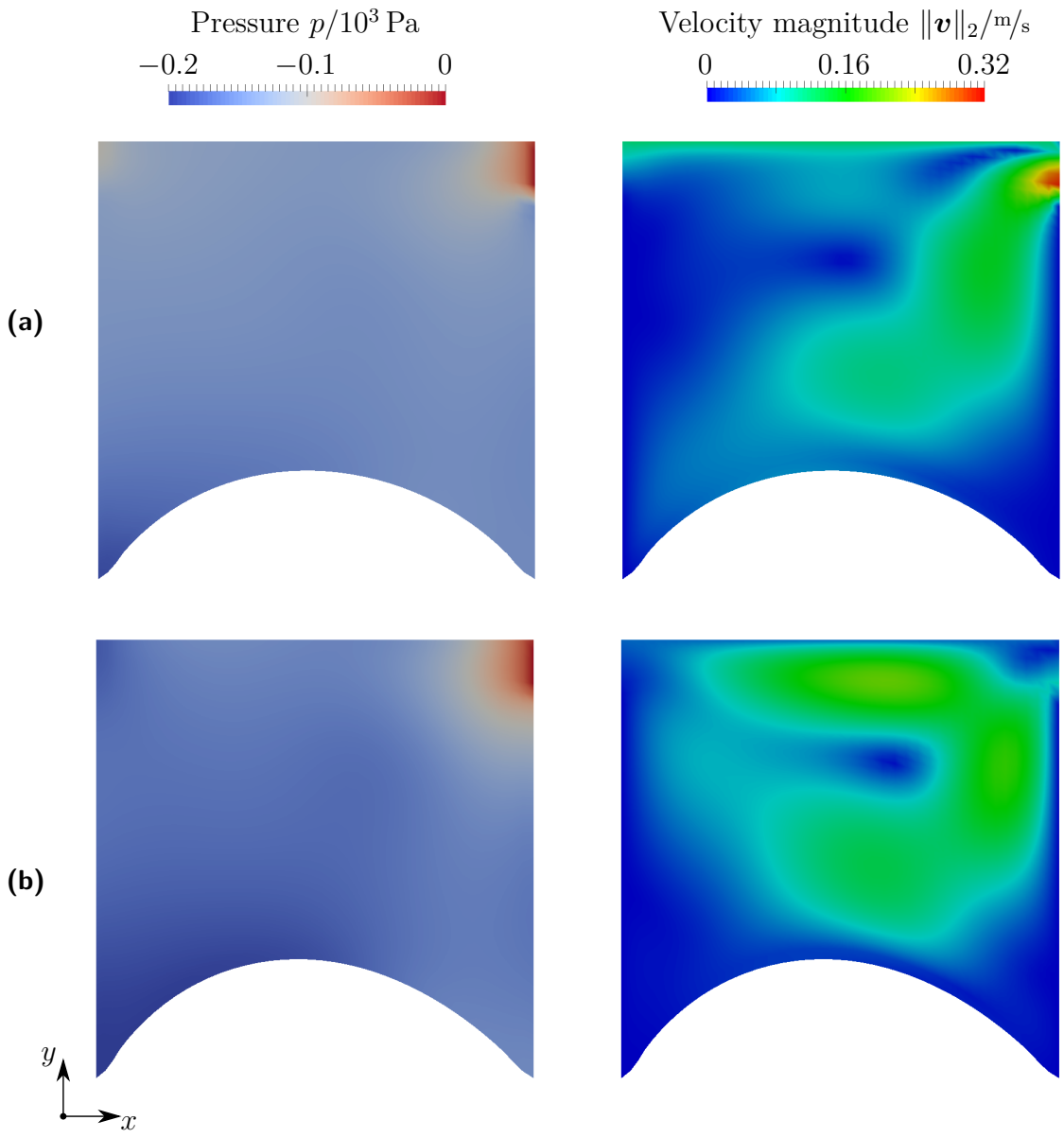


Figure 6.2: Pressure and velocity field for the two-dimensional lid-driven cavity flow at (a) $t = 35.39\,\mathrm{s}$ and (b) $t = 69.77\,\mathrm{s}$.

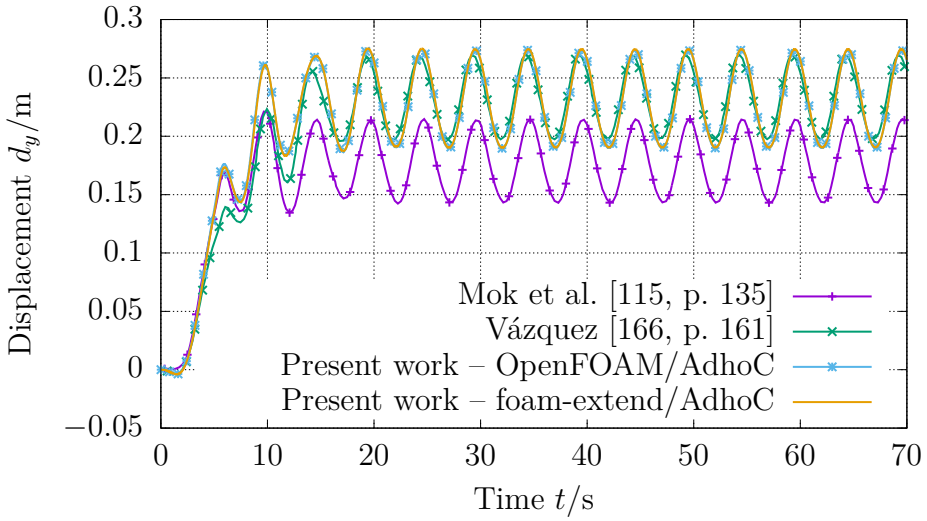


Figure 6.3: Membrane midpoint vertical displacement d_y versus time t for the two-dimensional lid-driven cavity flow.

for the overall computational effort associated to the solution process. In addition to the mean number of iterations $\bar{k}$, we also measure the CPU time $t_{\text{CPU,f}}$ and $t_{\text{CPU,s}}$ spent on the solution of the fluid and the structural field, as well as the time $t_{\text{CPU,alg}}$ required for the remaining algorithmic tasks in the coupling algorithm, such as the execution of the predictor or convergence acceleration scheme. In practice, $t_{\text{CPU,alg}}$ is obtained by subtracting the subfield solution time $t_{\text{CPU,f}} + t_{\text{CPU,s}}$ from the overall CPU time spent on the coupled solution procedure. From the results listed in Table 6.1, we notice that, regarding the number of iterations, the quasi-Newton least squares procedure exhibits the best performance. It requires the least number of iterations per time increment; $\bar{k} = 8.44$ iterations are, on average, required until convergence is achieved. It is followed by the modified version of the classical Aitken relaxation scheme proposed by Irons et al. [83] with $\bar{k} = 12.54$. Next comes the line extrapolation method, which, however, requires already more than 4 additional implicit iterations as compared to the Irons-Tuck relaxation scheme, and it almost doubles the computational effort associated to the quasi-Newton least squares method. Between 20 and 25 iterations are needed by the classical Aitken relaxation, the vector-ε algorithm, and the modification of the conventional Aitken relaxation proposed by Jennings [86] and Graves-Morris [60]. All other considered convergence acceleration schemes take up over 25 iterations per time increment on average and, thus, require at least three times the computational cost as compared to the quasi-Newton least squares method. Although these results cannot necessarily be generalized to other problems, this study serves as a valuable indicator of which convergence acceleration schemes seem to be preferable for the solution of coupled multifield problems using a partitioned approach. Furthermore, it is interesting to note that the CPU time spent on the solution of the fluid and the structural subproblem mostly takes up more than 99 % of the total CPU time. In this regard, the quasi-Newton least squares method is the only exception as here $t_{\text{CPU,alg}} \approx 1.5\,\%$ – due to the fact that this convergence acceleration scheme involves the solution of a least squares problem using a QR decomposition, which is slightly more expensive than other convergence acceleration schemes. Yet, the fraction $t_{\text{CPU,alg}}$ is almost negligible as compared to the subfield solution

Table 6.1: Comparison of the performance of various convergence acceleration schemes for the two-dimensional lid-driven cavity flow. Convergence acceleration schemes marked by † apply a constant relaxation with $\omega = 0.5$ in iterations $\mathrm{mod}(k, k_{\max}) \neq 0$ to avoid divergence.

Convergence acceleration scheme	Mean number of iterations $\bar{k}$	Fluid field solution time $t_{\mathrm{CPU,f}}/\%$	Struct. field solution time $t_{\mathrm{CPU,s}}/\%$	Remaining alg. tasks time $t_{\mathrm{CPU,alg}}/\%$
Constant relaxation, $\omega = 0.5$	25.34	82.04	17.76	0.21
Aitken relaxation†	20.92	80.93	18.83	0.24
Graves-Morris relaxation†	22.16	87.47	12.34	0.19
Iguchi relaxation†	31.70	88.06	11.74	0.20
Zienkiewicz relaxation†	36.17	87.72	12.06	0.21
Jennings relaxation†	22.16	83.30	16.49	0.21
Arthur relaxation†	22.64	83.44	16.29	0.25
Irons-Tuck relaxation, $\omega_0 = 0.5$	12.54	78.42	21.30	0.28
Line extrapolation method, $\beta = 1$	16.71	86.71	13.01	0.27
Vector ε-algorithm†	21.57	83.49	16.28	0.23
Topological ε-algorithm†	34.04	84.50	15.28	0.23
Vector θ-algorithm†	47.99	89.29	10.49	0.23
Generalized θ algorithm†	43.05	89.35	10.24	0.41
Vector w-transformation†	26.32	84.28	15.47	0.25
Euclidean w-transformation†	34.93	85.97	13.82	0.21
Broyden method, $\omega_0 = 1, k^* = 10$	28.56	88.60	10.95	0.45
Quasi-Newton least squares method, $\omega_0 = 0.2, \ell = 1$	8.44	86.68	11.83	1.49

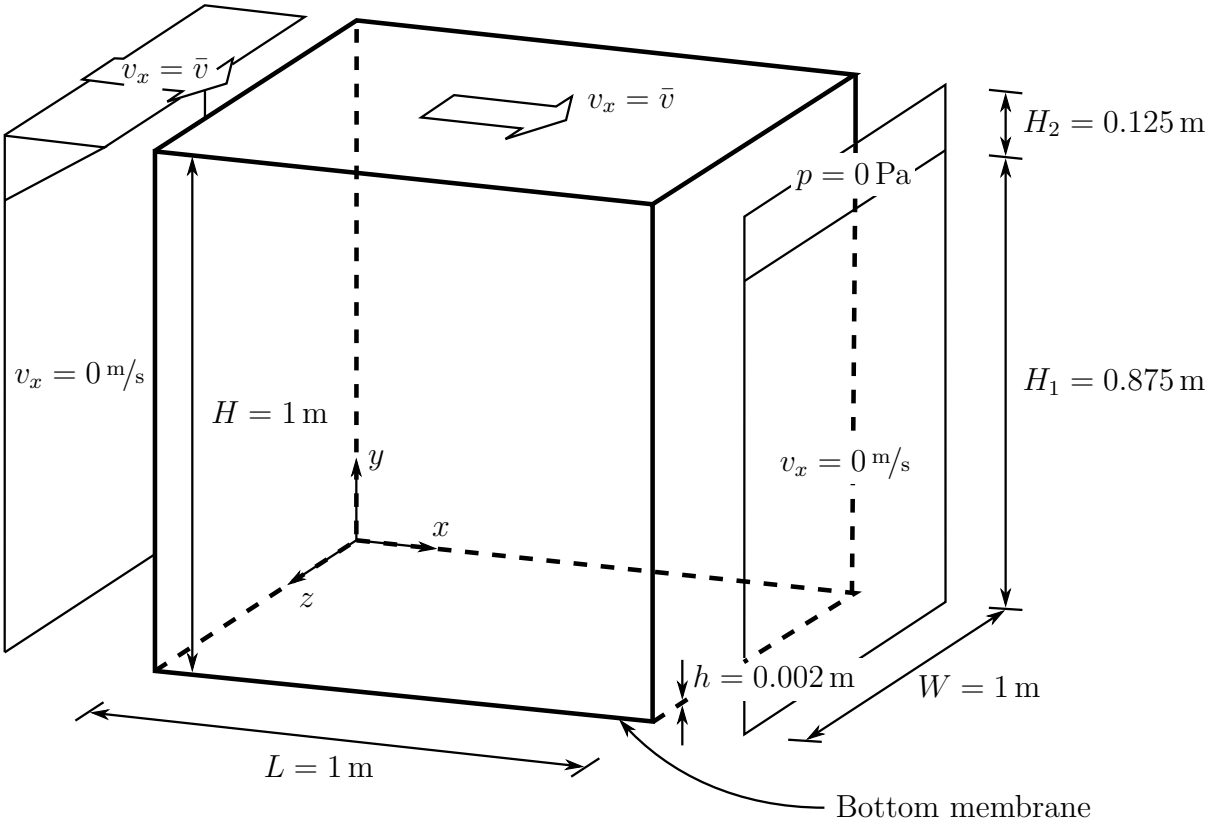


Figure 6.4: Geometry and boundary conditions for the three-dimensional lid-driven cavity flow with flexible bottom.

time. Also, it can be expected that this tendency does not change for larger problems. From these time measurements, it can be concluded that the implementation of the algorithmic building blocks available in *comana* is efficient enough to have only minor influence on the overall solution time.

6.2 Three-Dimensional Lid-Driven Cavity Flow with Flexible Bottom

As an extension to the previous example, we now consider a cuboidal domain of side length $L = W = H = 1\,\mathrm{m}$ with a flexible membrane at the bottom, cf. Figure 6.4. This benchmark problem has already been considered in [115, pp. 153–156, 166, pp. 163–166], for instance, and it serves to verify the partitioned solution procedure for a three-dimensional setting. Geometry and material parameters are chosen as before. Regarding the boundary conditions, we apply no-slip conditions at the front and rear side and prescribe an oscillatory lid velocity $v_x = \bar{v} = 1 - \cos(2\pi t/5\,\mathrm{s})$ at the top.

For the fluid part, we choose a hexahedral mesh consisting of $24\times24\times24$ hexahedral finite volumes, whereas the structural domain is discretized by 24×24 bilinear underintegrated membrane SHELL181 elements with hourglass control, available in the commercial FEM software suite ANSYS [3]. Following the benchmark definition, the time step size is changed to $\Delta t = 0.1\,\mathrm{s}$, and the structural time integration scheme uses a slightly higher spectral

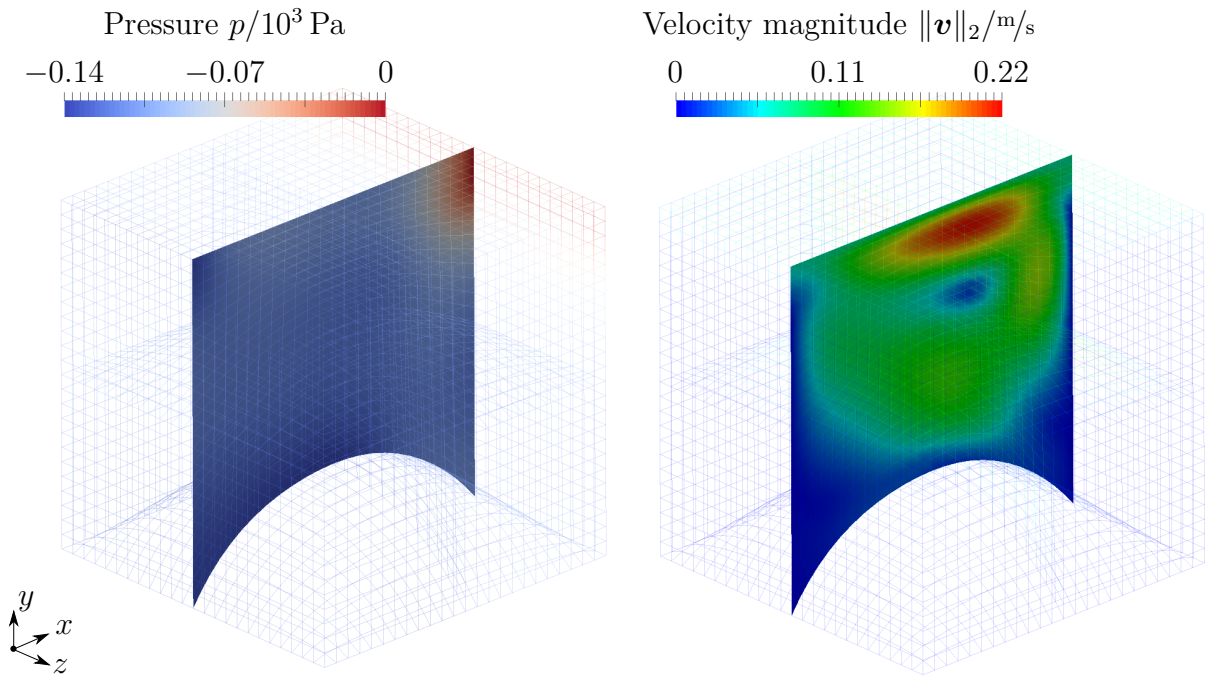


Figure 6.5: Pressure and velocity field in the three-dimensional cavity at time $t = 24.7\,\mathrm{s}$.

radius $\rho_\infty = 0.9$ to reduce numerical damping as compared to the two-dimensional example. Identical settings are chosen for the coupling procedure.

An impression of the pressure and velocity field at time $t = 24.7\,\mathrm{s}$ is given in Figure 6.5. The deformed membrane at different instants of time is depicted in Figure 6.6. In order to compare the numerical results to the results provided in [166, p. 164], we track the shell midpoint displacement d_y over time t as illustrated in Figure 6.7. Obviously, the results are again in fair agreement with the reference solution.

6.3 Round Cylinder with Flexible Membrane in Channel Flow

In this numerical example, we analyze the deflections of an elastic membrane attached to a rigid cylinder, which is subjected to a laminar incompressible channel flow, see Figure 6.8. Originally proposed in [71], this problem serves as another benchmark. The channel is of rectangular shape with side lengths $L = 2.5\,\mathrm{m}$ and $H = 0.41\,\mathrm{m}$. Measured from the bottom left corner of the channel, the cylinder of diameter $D = 0.1\,\mathrm{m}$ is located at $C_x = C_y = 0.2\,\mathrm{m}$. A membrane is symmetrically attached to the cylinder; it exhibits a length $\ell = 0.35\,\mathrm{m}$, a height $h = 0.02\,\mathrm{m}$, and extends to $x = 0.6\,\mathrm{m}$. In what follows, three different configurations as listed in Table 6.2 are considered. The fluid is again assumed to be Newtonian, and incompressible with density ρ_f and kinematic viscosity ν_f. For the elastic membrane, we choose a St. Venant-Kirchhoff material with density ρ_s, Young's modulus E, and Poisson's ratio ν_s. At the inlet, we prescribe a parabolic velocity profile

$$v_x = \frac{3\bar{v}y(H-y)}{2(H/2)^2}\,, \tag{6.1}$$

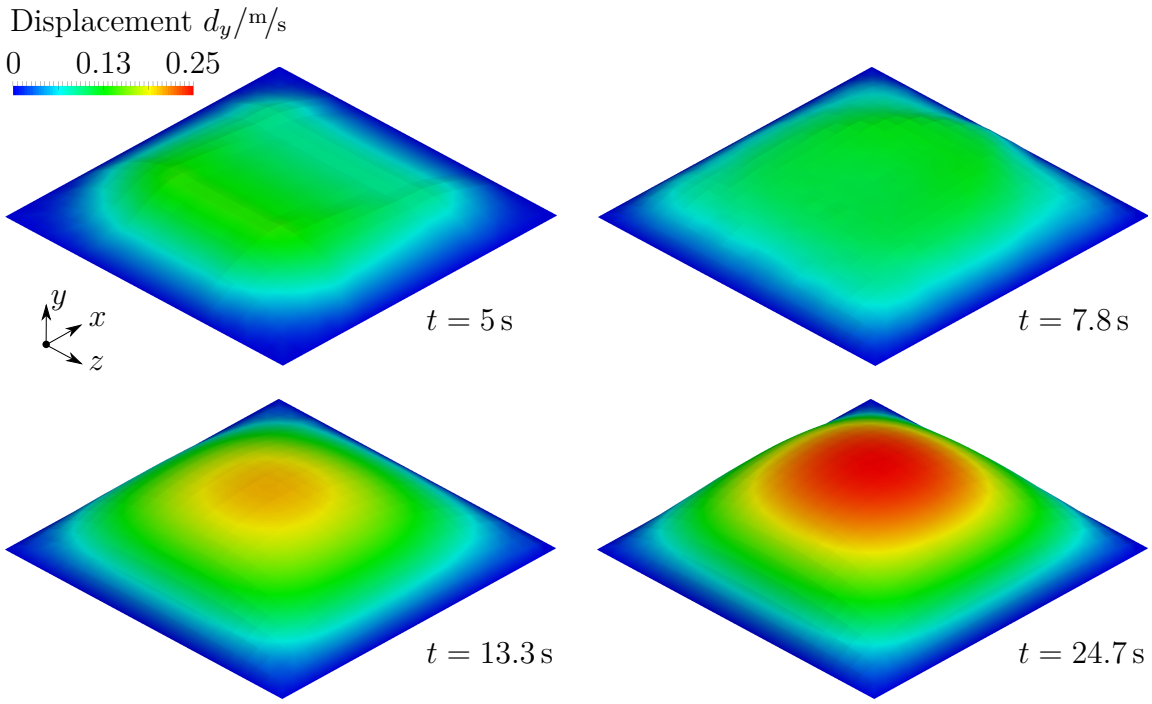


Figure 6.6: Deformed membrane at different instants of time.

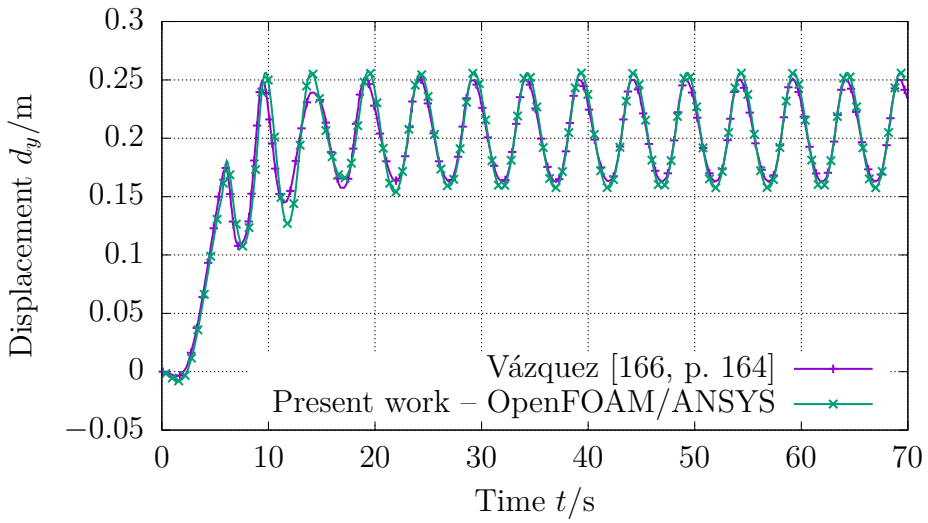


Figure 6.7: Membrane midpoint vertical displacement d_y versus time t for the three-dimensional lid-driven cavity flow.

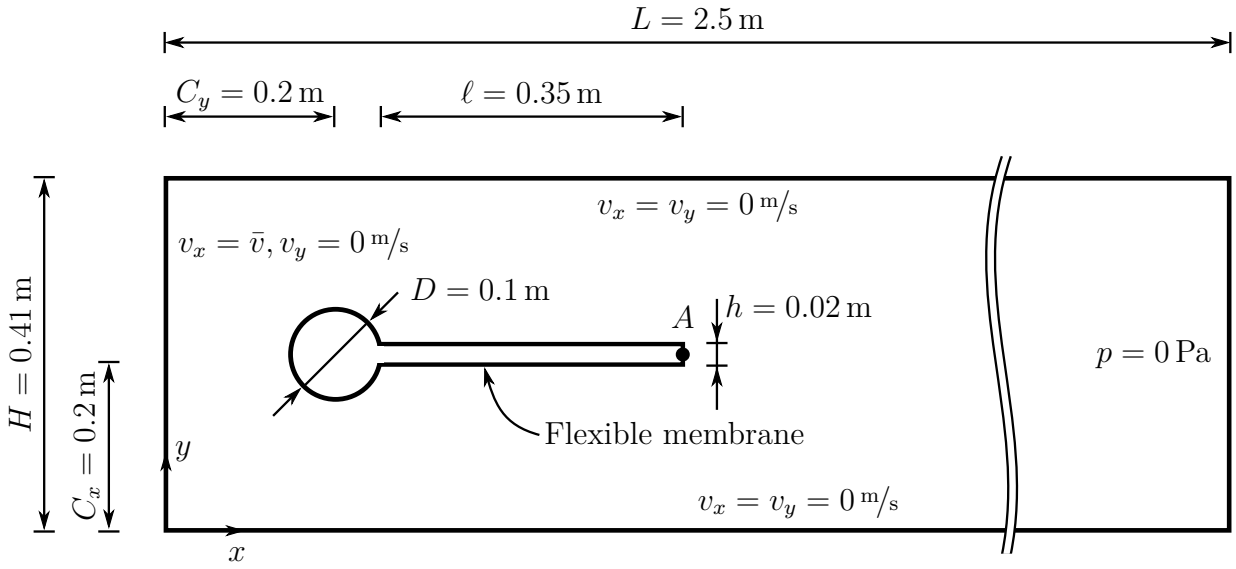


Figure 6.8: Geometry and boundary conditions for the round cylinder with flexible membrane in channel flow.

Table 6.2: Parameter settings for the round cylinder with flexible membrane in channel flow.

	Parameter	FSI1	FSI2	FSI3
Fluid	Density $\rho_\mathrm{f}/\mathrm{kg/m^3}$	10^3	10^3	10^3
	Kinematic viscosity $\nu_\mathrm{f}/\mathrm{m^2/s}$	10^{-3}	10^{-3}	10^{-3}
	Mean inflow velocity $\bar{v}/\mathrm{m/s}$	0.2	1	2
	Reynolds number $\mathrm{Re} = \bar{v}D/\nu_\mathrm{f}$	20	100	200
Structure	Density $\rho_\mathrm{s}/\mathrm{kg/m^3}$	10^3	10^4	10^3
	Young's modulus $E/\mathrm{N/m^2}$	1.4×10^6	5.6×10^6	1.4×10^6
	Poisson's ratio ν_s	0.4	0.4	0.4

such that the mean inflow velocity becomes $\bar{v}_x = \bar{v}$ and the maximum inflow velocity amounts to $\hat{v}_x = 3\bar{v}/2$. At the outlet, we impose $p = 0\,\mathrm{Pa}$ as a fixed reference pressure. We apply a no-slip condition for the fluid at the bottom and top wall, at the cylinder, and at the membrane. In the first parameter set, the flexible membrane is chosen as being light and stiff with a density $\rho_\mathrm{s} = 10^3\,\mathrm{kg/m^3}$, Young's modulus $E = 1.4 \times 10^6\,\mathrm{N/m^2}$, and Poisson's ratio $\nu_\mathrm{s} = 0.4$. For the mean inflow velocity, we take $\bar{v} = 0.2\,\mathrm{m/s}$. As will be shown in a moment, this setting results in a steady-state solution with constant membrane deflection. In the second parameter set, the density is increased to $\rho_\mathrm{s} = 10^4\,\mathrm{kg/m^3}$ and a higher Young's modulus $E = 5.6 \times 10^6\,\mathrm{N/m^2}$ is used, while the Poisson's ratio is left unchanged. This set of parameters, together with a higher mean inflow velocity $\bar{v} = 1\,\mathrm{m/s}$, leads to a transient membrane displacement inducing a von Kármán vortex street in the channel. Last but not least, the third parameter set uses the same structural properties as the steady-state case, while a significantly higher inlet velocity $\bar{v} = 2\,\mathrm{m/s}$ is applied. Here, a transient displacement response is to be expected as well. In all three cases, identical properties for the fluid are assumed: the density is taken as $\rho_\mathrm{f} = 10^3\,\mathrm{kg/m^3}$ and the kinematic viscosity amounts to $\nu_\mathrm{f} = 10^{-3}\,\mathrm{m^2/s}$. For the transient cases, the inflow velocity is smoothly increased

according to

$$v_x(t,y) = \frac{3y(H-y)}{2(H/2)^2}\bar{v}^*(t)\ , \quad \bar{v}^*(t) = \begin{cases} \bar{v}\frac{1-\cos(\pi t/t')}{2} & \text{if } t < t' = 2\,\text{s} \\ \bar{v} & \text{otherwise} \end{cases}\ . \tag{6.2}$$

For the solution of the fluid field, we employ the FVM and a computational mesh of 20,940 finite volumes, which are strongly graded towards the cylinder and the membrane in order to properly resolve the flow phenomena of most interest. Again, we apply the pimpleDyMFoam solver distributed along with the open-source CFD package OpenFOAM [123] to accomplish the flow solution. For the solution of the structural field, we use 3×1 anisotropic high-order quadrilateral elements of order $p_x = 8$ and $p_y = 4$, available in the high-order FEM software AdhoC [41, 40]. The transient problems are solved by applying the implicit second-order accurate backward differencing scheme for the fluid field, and the undamped Newmark scheme with $\beta = 0.25$ and $\gamma = 0.5$ for the structural part. For the first two parameter sets, the time step size is chosen as $\Delta t = 10^{-3}\,\text{s}$, whereas $\Delta t = 10^{-2}\,\text{s}$ for the third setting. In the coupled solution procedure, we apply an absolute convergence criterion $\|\mathbf{r}_{j+1}^k\|_2 < \varepsilon_{\text{abs}} = 10^{-8}$ and a relative criterion $\|\mathbf{r}_{j+1}^k\|_2 / \|\mathbf{r}_{j+1}^0\|_2 < \varepsilon_{\text{rel}} = 10^{-6}$ to decide when to leave the implicit iteration and proceed to the next time increment.

Figure 6.9 depicts the steady-state pressure and velocity field for the first parameter set. Apparently, the deflection of the membrane is fairly small for this case. Table 6.3 lists the most important result quantities. Both the steady-state deflection of the point

Table 6.3: Results for the round cylinder with flexible membrane in channel flow compared to the reference solution from [71, pp. 383 sq.]. For the transient problems, the oscillation frequency at the point A is given in square brackets.

	Parameter	FSI1	FSI2	FSI3
Displacement $d_x/10^{-3}\,\text{m}$	Present work	0.0227	-14.58 ± 12.44 [3.8]	-2.69 ± 2.53 [10.9]
	[71, pp. 383 sq.]	0.0221	-15.34 ± 13.03 [3.8]	-2.89 ± 2.72 [10.9]
	Rel. error e/%	7.6	-5.0 ∓ 4.5 [0.0]	-6.9 ∓ 7.0 [0.0]
Displacement $d_y/10^{-3}\,\text{m}$	Present work	0.8209	1.23 ± 80.60 [2.0]	1.48 ± 34.38 [5.3]
	[71, pp. 383 sq.]	0.8001	1.37 ± 80.34 [2.0]	1.51 ± 35.10 [5.3]
	Rel. error e/%	2.6	-10.2 ± 0.3 [0.0]	-2.0 ∓ 2.1 [0.0]
Drag force F_d/N	Present work	14.295	208.83 ± 73.75 [3.8]	457.3 ± 22.66 [10.9]
	[71, pp. 383 sq.]	14.159	219.10 ± 77.00 [3.8]	462.4 ± 25.42 [10.9]
	Rel. error e/%	1.0	-4.7 ∓ 4.2 [0.0]	-1.1 ∓ 10.9 [0.0]
Lift force F_l/N	Present work	0.7638	0.88 ± 234.20 [2.0]	2.22 ± 149.78 [5.3]
	[71, pp. 383 sq.]	0.7627	-1.55 ± 256.74 [2.0]	2.05 ± 156.45 [5.3]
	Rel. error e/%	0.1	-156.8 ∓ 8.8 [0.0]	8.3 ∓ 4.3 [0.0]

$A(t = 0\,\text{s}) = (0.6\,\text{m}, 0.2\,\text{m})$ at the tip of the membrane as well as the total lift force F_l and drag force F_d across the cylinder and the membrane are in good agreement with the reference solution.

Figure 6.10 provides a visual impression of the pressure and velocity field for the second parameter set. In this case, the flexible membrane is strongly deformed and induces a von Kármán vortex street in the flow. In order to compare our results to the reference results in [71, p. 383], we trace the displacements at the deflected point A as well as the total lift force F_l and total drag force F_d on the cylinder and the membrane over time, as shown in Figure 6.11. Here, the results are in good agreement with the reference solution as well.

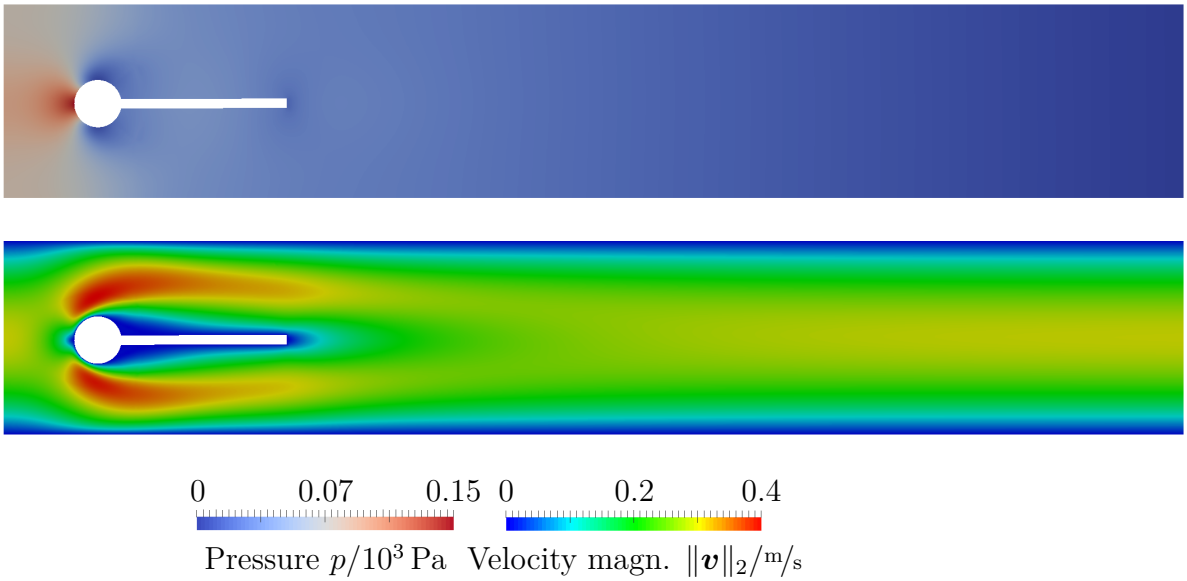


Figure 6.9: Steady-state pressure and velocity field for the round cylinder with membrane in channel flow and parameter set FSI1.

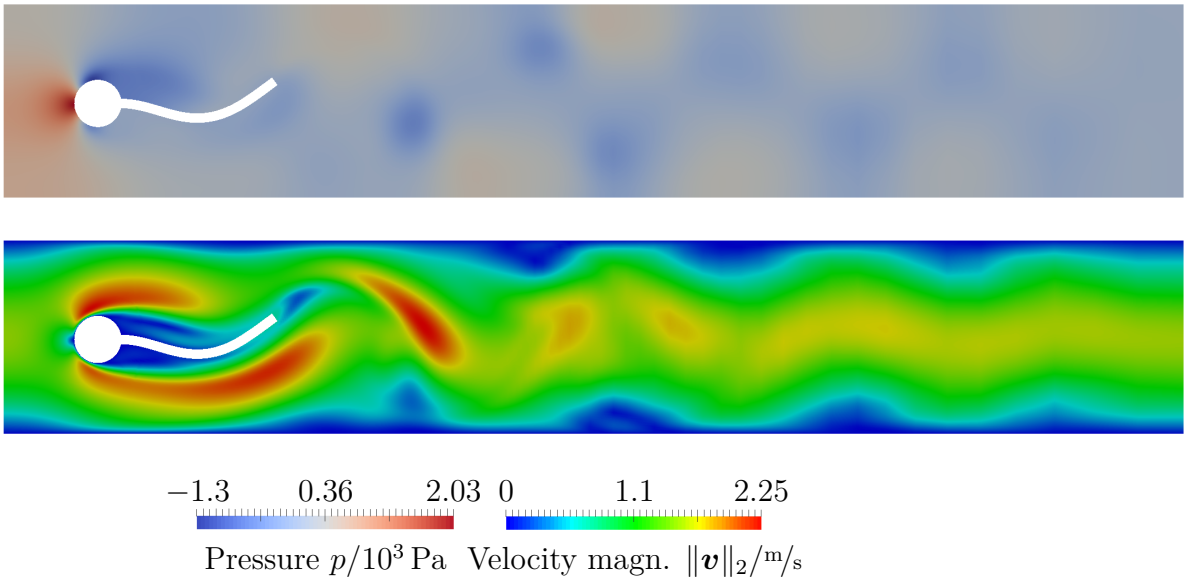


Figure 6.10: Pressure and velocity field for the round cylinder with membrane in channel flow and parameter set FSI2 at time $t = 9.7\,\mathrm{s}$.

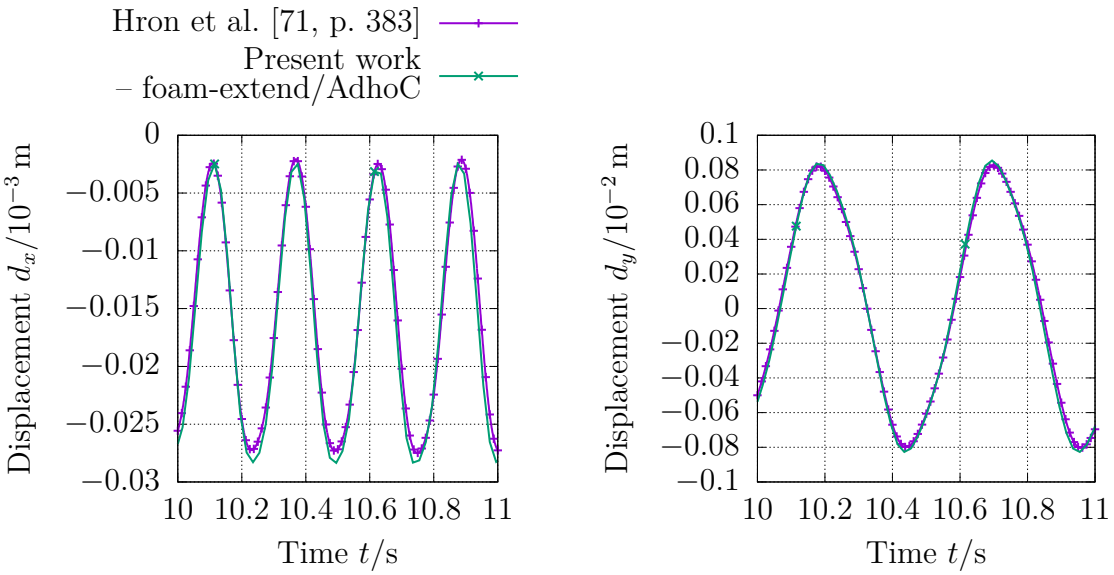


Figure 6.11: Displacement d_x and d_y at point A for the round cylinder with membrane in channel flow and parameter set FSI2.

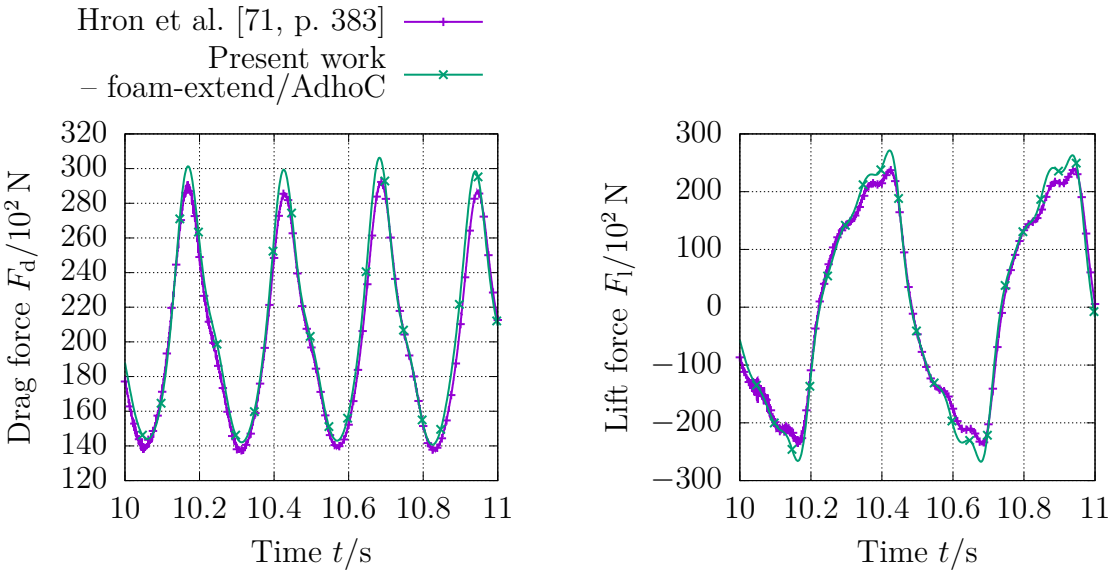


Figure 6.12: Lift force F_l and drag force F_d for the round cylinder with membrane in channel flow and parameter set FSI2.

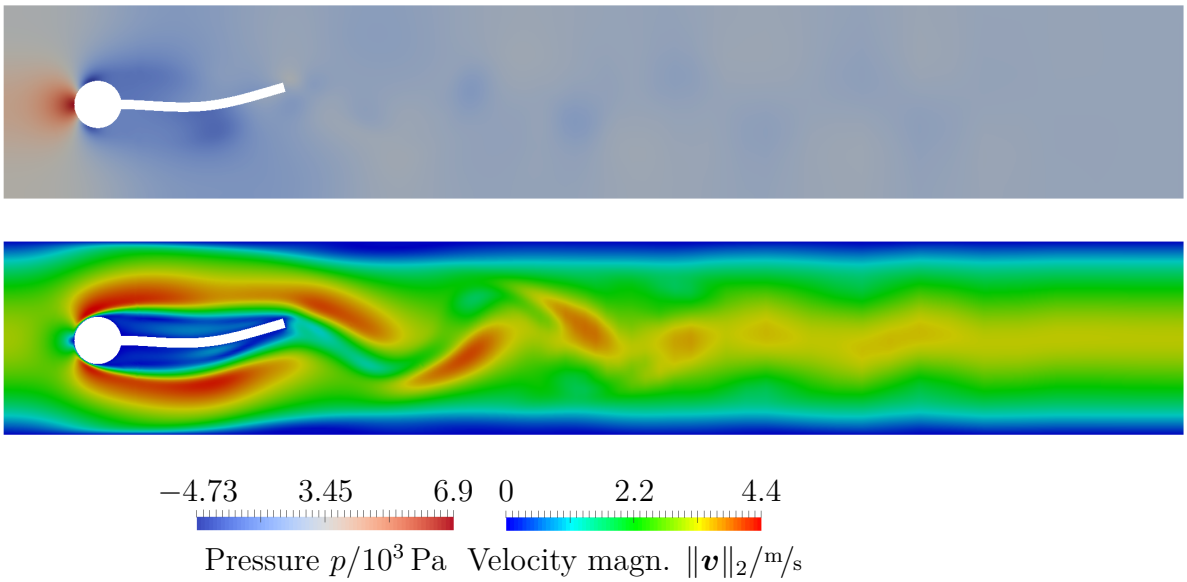


Figure 6.13: Pressure and velocity field for the round cylinder with membrane in channel flow and parameter set FSI3 at time $t = 9.58\,\mathrm{s}$.

The same holds for the lift and drag force depicted in Figure 6.12. As before, the mean and maximum displacement and forces are listed in Table 6.3.

The pressure and velocity field for the last parameter set FSI3 are depicted in Figure 6.13 for the moment where the flexible membrane exhibits maximum deflection. As illustrated in Figure 6.14 and 6.15, the numerical results again resemble the reference solution fairly well. Minor deviations can be attributed to the different discretization scheme for the fluid problem and the different spatial and temporal resolution as compared to the reference.

6.4 Square Cylinder with Flexible Membrane in Channel Flow

Next, we consider a deformable thin elastic shell attached to a square cylinder, subjected to an incompressible Newtonian flow. Originally proposed in [170, pp. 14 sq.], this problem was later also computed in [152, pp. 116–123, 37, pp. 258–270, 73, pp. 2100-2102, 166, pp. 152–159], for instance. As sketched in Figure 6.16, the computational domain is of rectangular shape and exhibits a length $L = 0.21\,\mathrm{m}$ and a height $H = 0.12\,\mathrm{m}$. Symmetrically placed in the channel, the bluff body has a side length $a = 0.01\,\mathrm{m}$ and extends to $L_1 = 0.055\,\mathrm{m}$. At the right end of the square cylinder, an elastic membrane with length $\ell = 0.04\,\mathrm{m}$ and thickness $h = 6 \times 10^{-4}\,\mathrm{m}$ is attached.

In what follows, we consider two different parameter sets with different material properties and inflow velocities. In parameter set #1, we choose a fluid density $\rho_\mathrm{f} = 1.18\,\mathrm{kg/m^3}$ and a dynamic viscosity $\mu = 1.82 \times 10^{-5}\,\mathrm{Pa\,s}$. For the elastic membrane, we assume a St. Venant-Kirchhoff material with density $\rho_\mathrm{s} = 2 \times 10^3\,\mathrm{kg/m^3}$, Young's modulus $E = 2 \times 10^5\,\mathrm{N/m^2}$, and Poisson's ratio $\nu_\mathrm{s} = 0.35$. At the inlet, the flow velocity in x-direction is taken as $v_x \equiv \bar{v} = 0.315\,\mathrm{m/s}$. At the bottom and top wall, we apply a slip condition and impose $v_y = 0\,\mathrm{m/s}$. At the outlet, we enforce a fixed reference pressure $p = 0\,\mathrm{Pa}$.

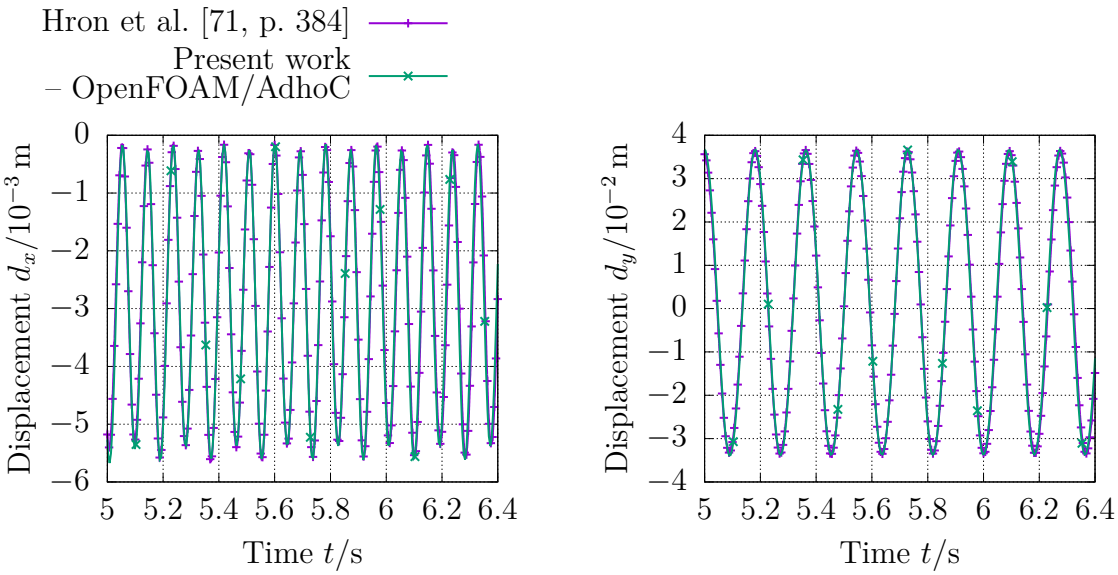


Figure 6.14: Displacement d_x and d_y at point A for the round cylinder with membrane in channel flow and parameter set FSI3.

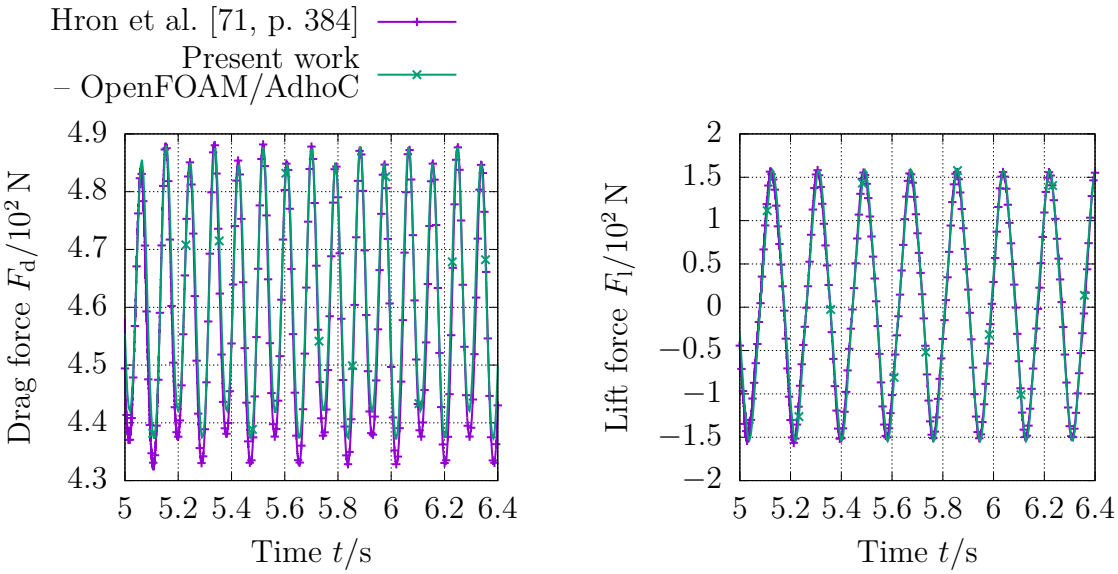


Figure 6.15: Lift force F_l and drag force F_d for the round cylinder with membrane in channel flow and parameter set FSI3.

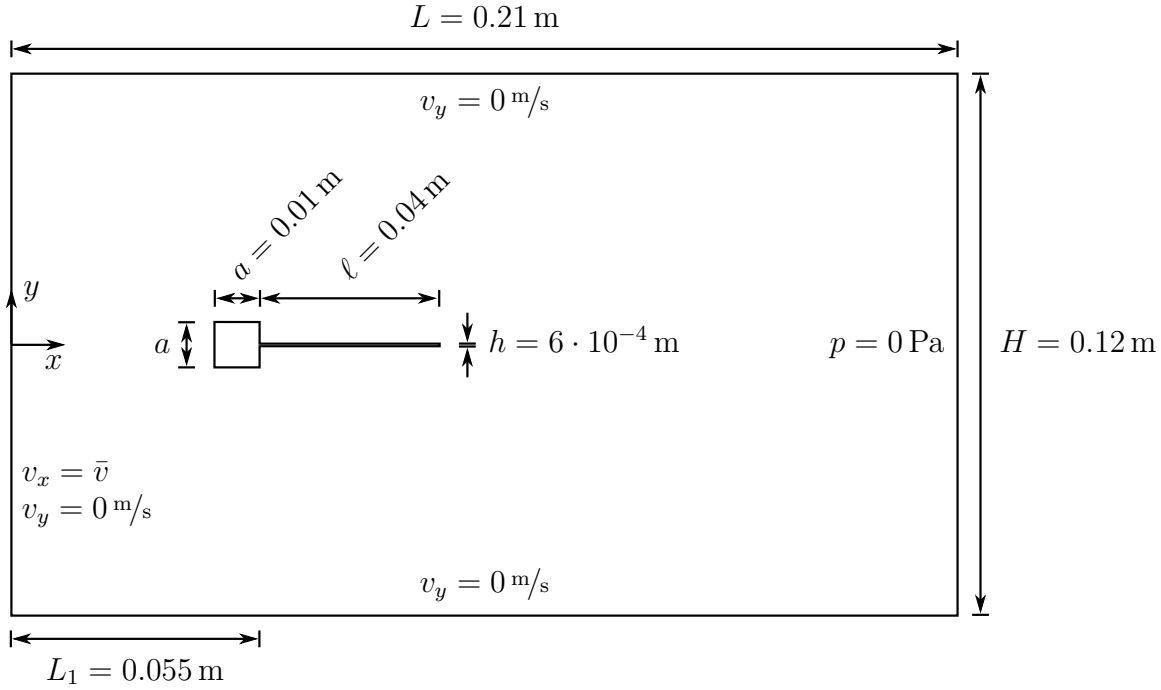


Figure 6.16: Square cylinder with a flexible membrane in channel flow.

In parameter set #2, we change the density and the Young's modulus of the structure to $\rho_\mathrm{s} = 10^2\,\mathrm{kg/m^3}$ and $E = 2.5 \times 10^5\,\mathrm{N/m^2}$, respectively. In addition, the inlet velocity is increased to $\bar{v} = 0.513\,\mathrm{m/s}$.

We use an identical spatial and temporal discretization for both parameter sets. For the spatial discretization of the fluid field, we employ an FV mesh consisting of 25,394 cells. The flow problem is solved by applying the pimpleDyMFoam solver available from the open-source CFD toolbox OpenFOAM [123]. For the discretization of the elastic membrane, we choose an FE mesh consisting of 18×1 anisotropic high-order elements of polynomial order $p_x = 8$ and $p_y = 4$. This problem is solved utilizing the in-house high-order FEM software AdhoC [41, 40]. Regarding the temporal discretization of the fluid problem, we use the second-order backward differencing scheme. For time integration of the structural subproblem, we choose the undamped Newmark scheme with $\beta = 0.25$ and $\gamma = 0.5$. In the coupled solution process, we apply an absolute convergence criterion $\|\mathbf{r}_{j+1}^k\|_2 < \varepsilon_\mathrm{abs} = 10^{-8}$ and a relative criterion $\|\mathbf{r}_{j+1}^k\|_2 / \|\mathbf{r}_{j+1}^0\|_2 < \varepsilon_\mathrm{rel} = 10^{-6}$ to judge whether the subfields are equilibrated to sufficient accuracy.

Figure 6.17 shows the pressure and velocity fields for parameter set #1 at different instants of time and gives an impression of the large deflections of the elastic structure. Clearly, the large deformations require appropriate mesh deformation techniques to deform the FV mesh according to the structural displacement response. Due to the unsteady dissolution of vortices from the membrane, a vortex street evolves in downstream direction. In order to compare the results to the reference solution available from [166, p.154], the displacement d_y at the membrane tip is tracked over time. This leads to the graph depicted in Figure 6.18, indicating that the periodic oscillation is in acceptable agreement with the reference solution. Yet, it should be mentioned that the frequency obtained in this work is slightly higher than that of the reference result, which is due to the fact that, as opposed to the reference case, undamped time integration schemes have been used for the fluid and

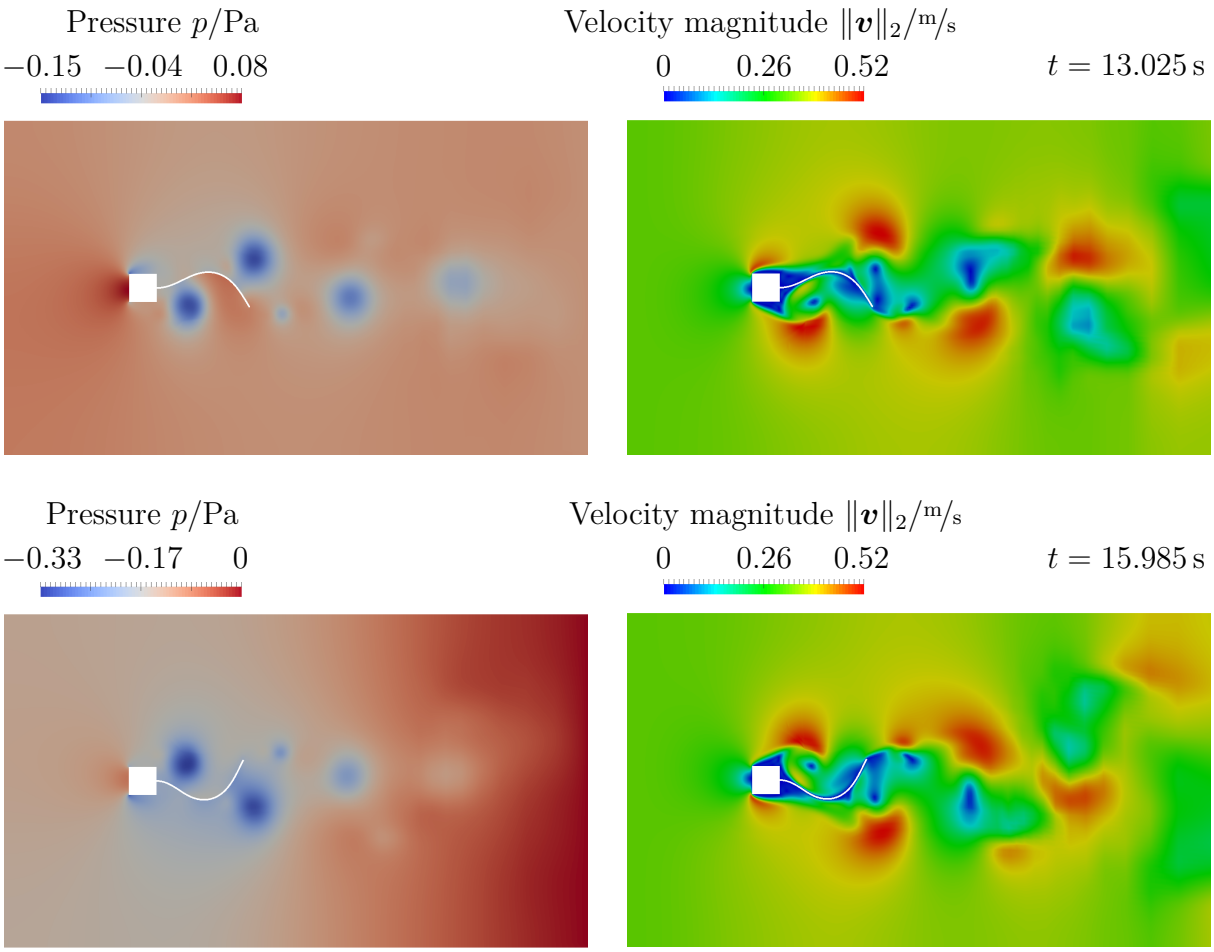


Figure 6.17: Pressure and velocity fields for the square cylinder with flexible membrane in channel flow using parameter set #1.

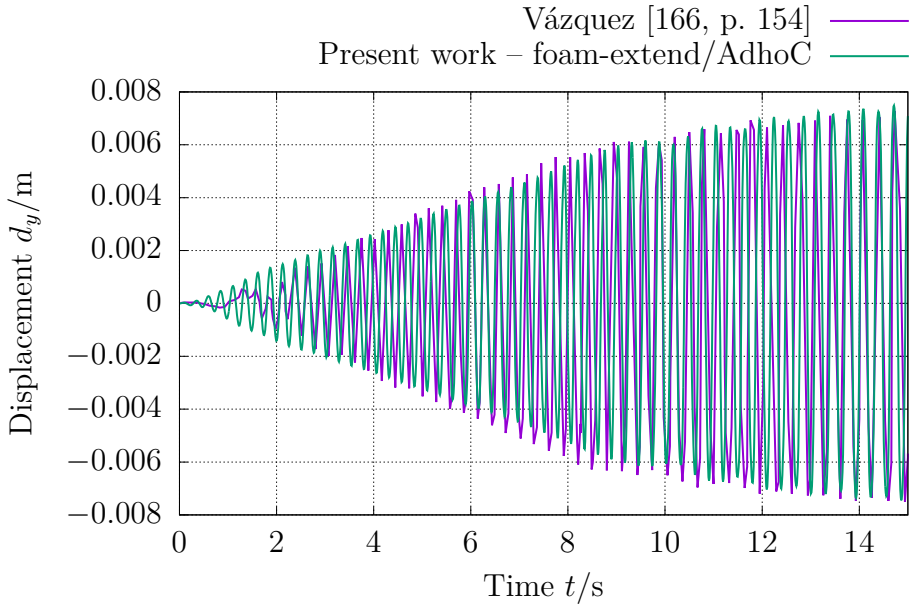


Figure 6.18: Membrane tip displacement d_y versus time t for the square cylinder in channel flow and parameter set #1.

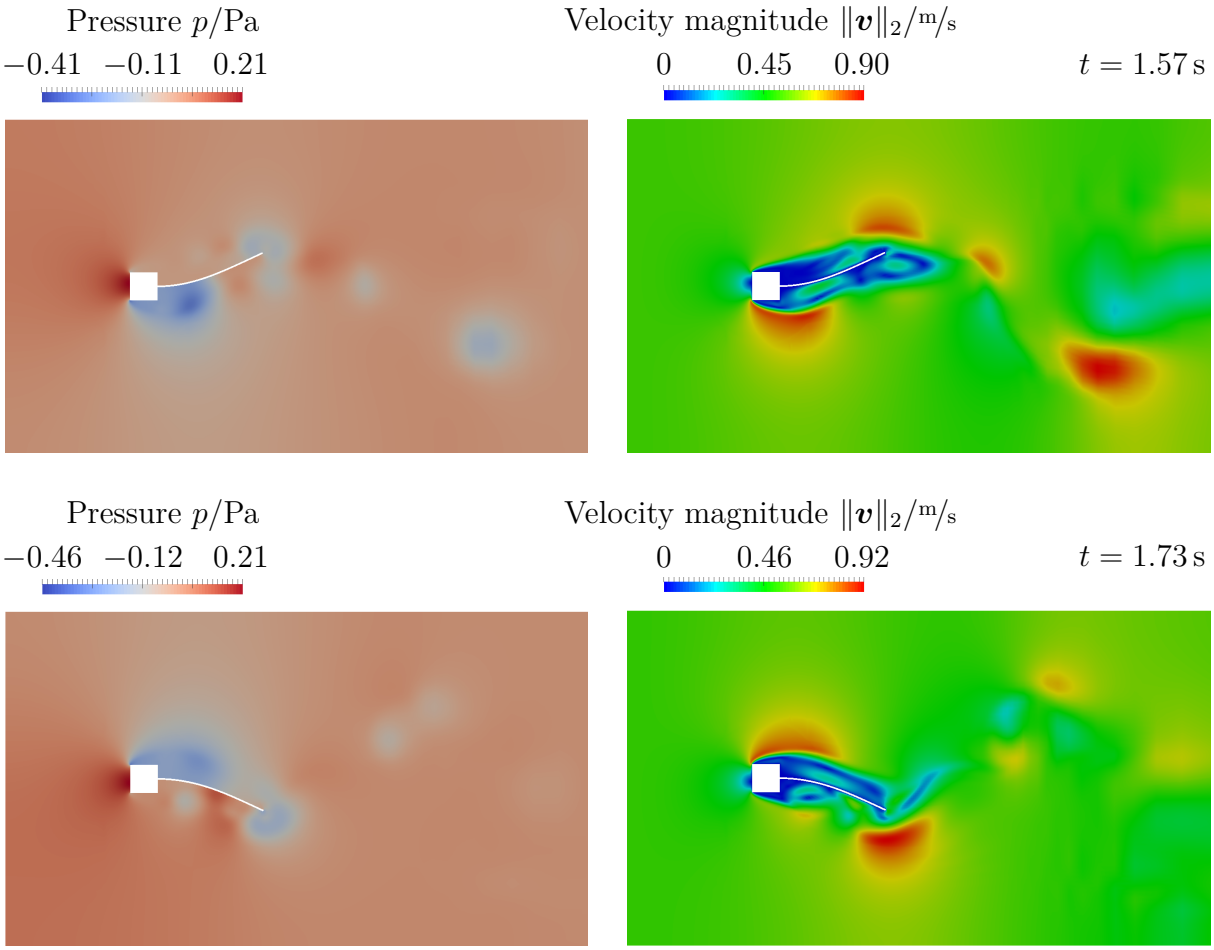


Figure 6.19: Pressure and velocity fields for the square cylinder with flexible membrane in channel flow using parameter set #2.

the structural subproblem.

With parameter set #2, the pressure and velocity fields depicted in Figure 6.19 are obtained. Here, a notable deformation of the elastic membrane can be observed as well, underlining the necessity of an appropriate mesh deformation technique for the FV mesh. Figure 6.20 graphs the tip displacement of the elastic membrane – obviously, the amplitude of the oscillation is in good agreement with the reference results. Due to the use of undamped time integration schemes for the fluid and the structural subproblem, also here the frequency is a little bit higher as compared to the reference solution.

6.5 Flapping Console in Channel Flow

Another interesting benchmark problem – as depicted in Figure 6.21, taken from a tutorial accompanying the open-source CFD toolbox OpenFOAM [123] – is an elastic console subjected to a channel flow. Here, we consider a channel section of total length $L = 6\,\mathrm{m}$ and height $H = 1\,\mathrm{m}$. At $L_1 = 2\,\mathrm{m}$, a flexible flap of thickness $\ell = 0.05\,\mathrm{m}$ is installed, forcing the flow to pass through the remaining gap between the flap tip and the channel top. Material properties for the incompressible Newtonian fluid are the density $\rho_\mathrm{f} = 1\,\mathrm{kg/m^3}$ and the

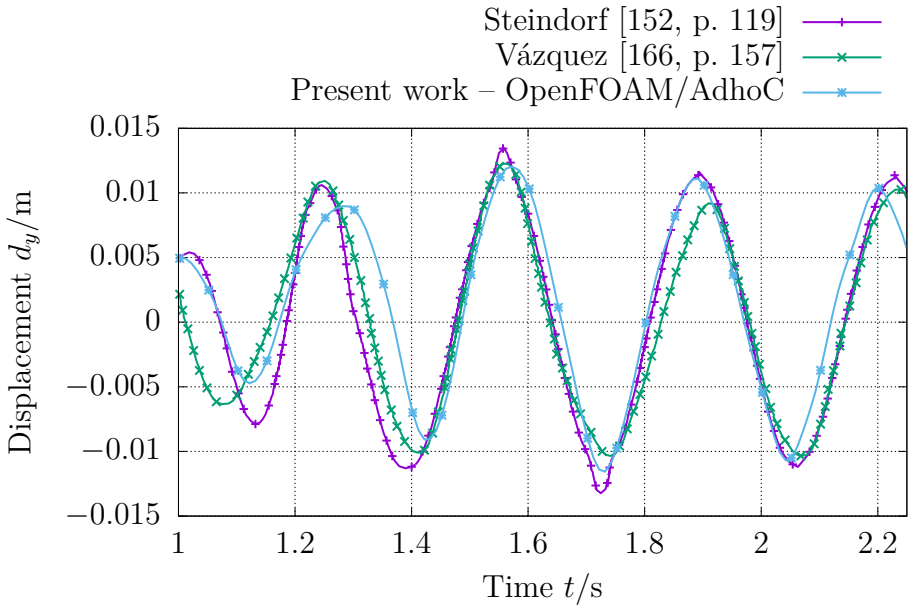


Figure 6.20: Membrane tip displacement d_y versus time t for the square cylinder in channel flow and parameter set #2.

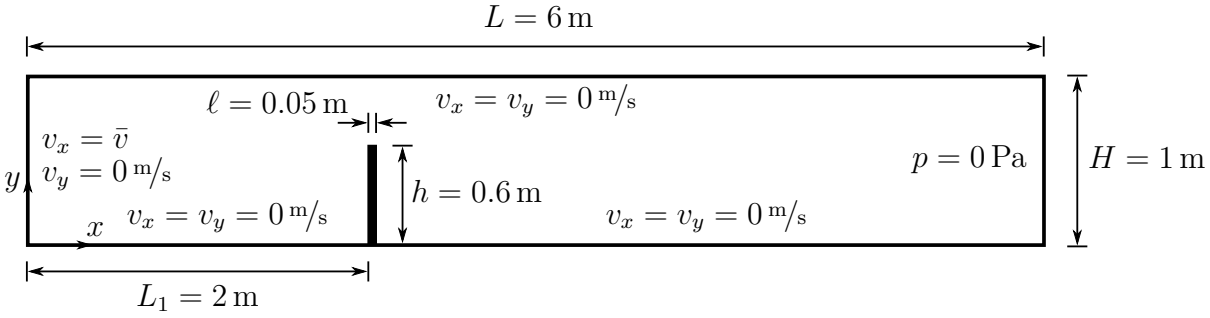


Figure 6.21: Flapping console in channel flow.

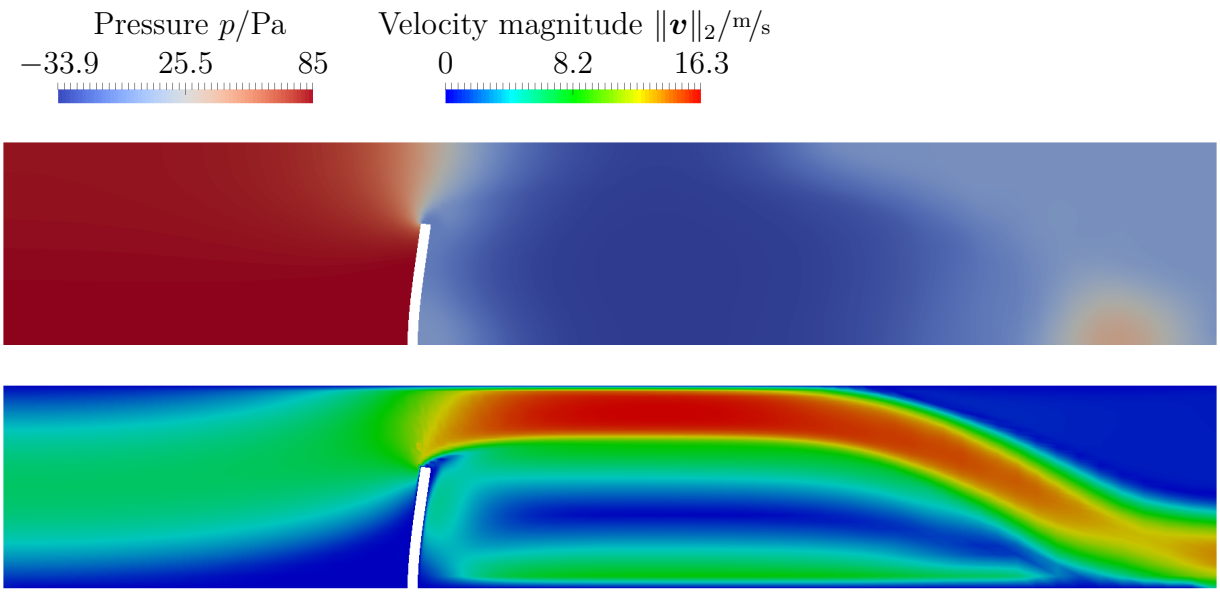


Figure 6.22: Steady-state pressure and velocity field for the flapping console in channel flow.

kinematic viscosity $\nu_\mathrm{f} = 10^{-3}\,\mathrm{m/s^2}$. For the structure, we assume a St. Venant-Kirchhoff material with density $\rho_\mathrm{s} = 10^3\,\mathrm{kg/m^3}$, Young's modulus $E = 2 \times 10^6\,\mathrm{N/m^2}$, and Poisson's ratio $\nu_\mathrm{s} = 0.3$. At the inlet of the channel, a parabolic velocity profile is applied, which is smoothly increased according to the ramp function

$$v_x(t,y) = \frac{3y(H-y)}{2(H/2)^2}\bar{v}^*(t)\,, \quad \bar{v}^*(t) = \begin{cases} \bar{v}\frac{1-\cos(\pi t/t')}{2} & \text{if } t < t' \\ \bar{v} & \text{otherwise} \end{cases}, \tag{6.3}$$

where $\bar{v} = 4\,\mathrm{m/s}$ is the final mean inlet velocity, and $t' = 5\,\mathrm{s}$ represents the ramp time. A no-slip condition is imposed at the top and the bottom wall, while a fixed reference pressure $p = 0\,\mathrm{Pa}$ is applied at the outlet.

For the spatial discretization of the fluid field, we employ an FV mesh comprising 4,900 cells, strongly graded towards the structure. This subproblem is again solved using the pimpleDyMFoam solver from the OpenFOAM [123] package. An FE mesh consisting of 1×3 plane-stress anisotropic high-order finite elements of polynomial degree $p_x = 4$ and $p_y = 8$ is chosen for the spatial discretization of the elastic flap. The structural problem is solved using the high-order FEM software AdhoC [41, 40]. Regarding the temporal discretization of the fluid problem, we use the Euler backward scheme and a time step size $\Delta t = 0.01\,\mathrm{s}$ to advance in time. For time integration of the structural subproblem, we select the undamped standard Newmark scheme with $\beta = 0.25$ and $\gamma = 0.5$. In the coupled solution process, the relevant field quantities are iteratively exchanged within a time increment until either the absolute convergence criterion $\|\mathbf{r}_{j+1}^k\|_2 < \varepsilon_\mathrm{abs} = 10^{-3}$ or the relative criterion $\|\mathbf{r}_{j+1}^k\|_2/\|\mathbf{r}_{j+1}^0\|_2 < \varepsilon_\mathrm{rel} = 10^{-2}$ is fulfilled.

Figure 6.22 illustrates the steady-state pressure and velocity field for this problem. Behind the flap, the flow separates and a vortex evolves. At the tip of the flap, a steady-state deflection $d_x = 0.066\,\mathrm{m}$ and $d_y = -0.004\,\mathrm{m}$ is observed.

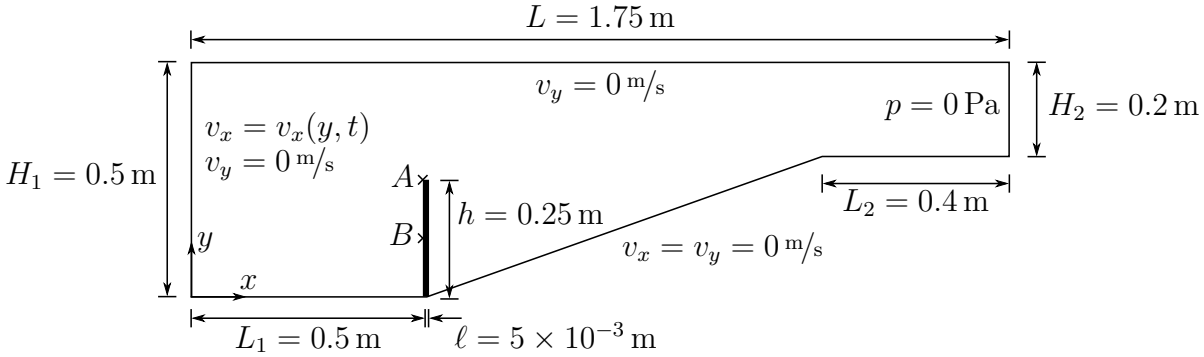


Figure 6.23: Flexible restrictor in converging channel.

6.6 Flexible Restrictor in Converging Channel

In the next benchmark problem, we consider a flexible restrictor in a converging channel, as illustrated in Figure 6.23. Mok et al. [115, pp. 147–152], Vázquez [166, pp. 167–170], and Degroote et al. [36, pp. 799 sq.] already computed this problem in order to verify their subproblem solvers and FSI setup. The channel exhibits an overall length $L = 1.75\,\mathrm{m}$, a height $H_1 = 0.5\,\mathrm{m}$ at the inlet, and a height $H_2 = 0.2\,\mathrm{m}$ at the outlet. For the fluid, we assume a density $\rho_\mathrm{f} = 956\,\mathrm{kg/m^3}$ and a dynamic viscosity $\mu = 0.145\,\mathrm{Pa\,s}$. For the structure, we choose a St. Venant-Kirchhoff material with density $\rho_\mathrm{s} = 1.5 \times 10^3\,\mathrm{kg/m^3}$, Young's modulus $E = 2.3 \times 10^6\,\mathrm{N/m^2}$, and Poisson's ratio $\nu_\mathrm{s} = 0.45$. At the inlet, the parabolic velocity profile

$$v_x(y,t) = \left(1 - \left(\frac{y-H}{H}\right)^2\right)\hat{v}^*(t)\,, \quad \hat{v}^*(t) = \begin{cases} \hat{v}\frac{1-\cos(\pi t/t')}{2} & \text{if } t < t' \\ \hat{v} & \text{otherwise} \end{cases} \tag{6.4}$$

is imposed, where the peak inflow velocity is taken as $\hat{v} = 0.06067\,\mathrm{m/s}$ and $t' = 10\,\mathrm{s}$ for the ramp time. For the bottom walls, we prescribe a no-slip condition, while we apply a symmetry boundary condition at the top wall. For the spatial discretization of the flow problem, we employ the FVM and a mesh of 16,080 cells. Here again, the pimpleDyMFoam solver from the CFD toolbox OpenFOAM [123] is used for the solution of the flow problem. The structure is discretized by anisotropic high-order plane-stress finite elements with polynomial orders $p_x = 2$ and $p_y = 4$, available as part of the in-house high-order FEM code AdhoC [41, 40]. Regarding the temporal discretization, we employ the second-order backward differencing scheme for the fluid, whereas we apply the undamped Newmark scheme with the parameters $\beta = 0.25$ and $\gamma = 0.5$ for the structural problem. A time step size $\Delta t = 0.1\,\mathrm{s}$ is chosen and kept constant throughout the total number of 250 calculated time steps. In the coupled solution procedure, an absolute convergence criterion $\|\mathbf{r}_{j+1}^k\|_2 < \varepsilon_\mathrm{abs} = 10^{-7}$ and a relative criterion $\|\mathbf{r}_{j+1}^k\|_2 / \|\mathbf{r}_{j+1}^0\|_2 < \varepsilon_\mathrm{rel} = 10^{-3}$ serve to decide whether the subfields are equilibrated to sufficient accuracy within a time increment.

Figure 6.24 illustrates the evolution of the pressure and velocity field over time until a steady state is reached. In order to compare the numerical results to the reference solution, we evaluate the horizontal displacement d_x and the pressure p at the point A at the top left corner, as well as at the point B at the center of the left edge of the restrictor over time. Figure 6.25 graphs the results. Apparently, the displacement response of the elastic structure and the pressure on the moving surface are in perfect agreement with the reference solutions.

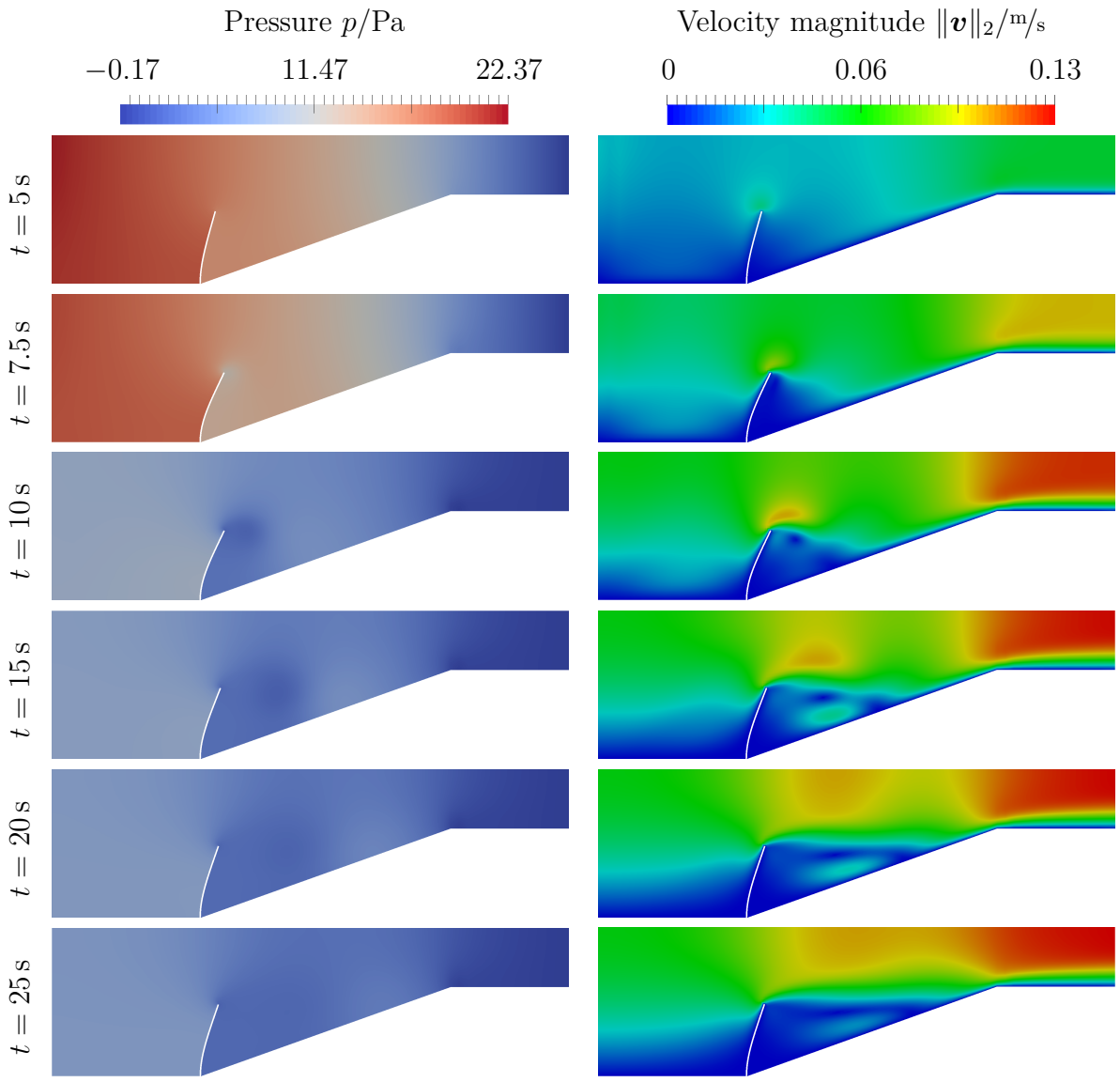


Figure 6.24: Pressure and velocity field for the flexible restrictor in the converging channel at different instants of time.

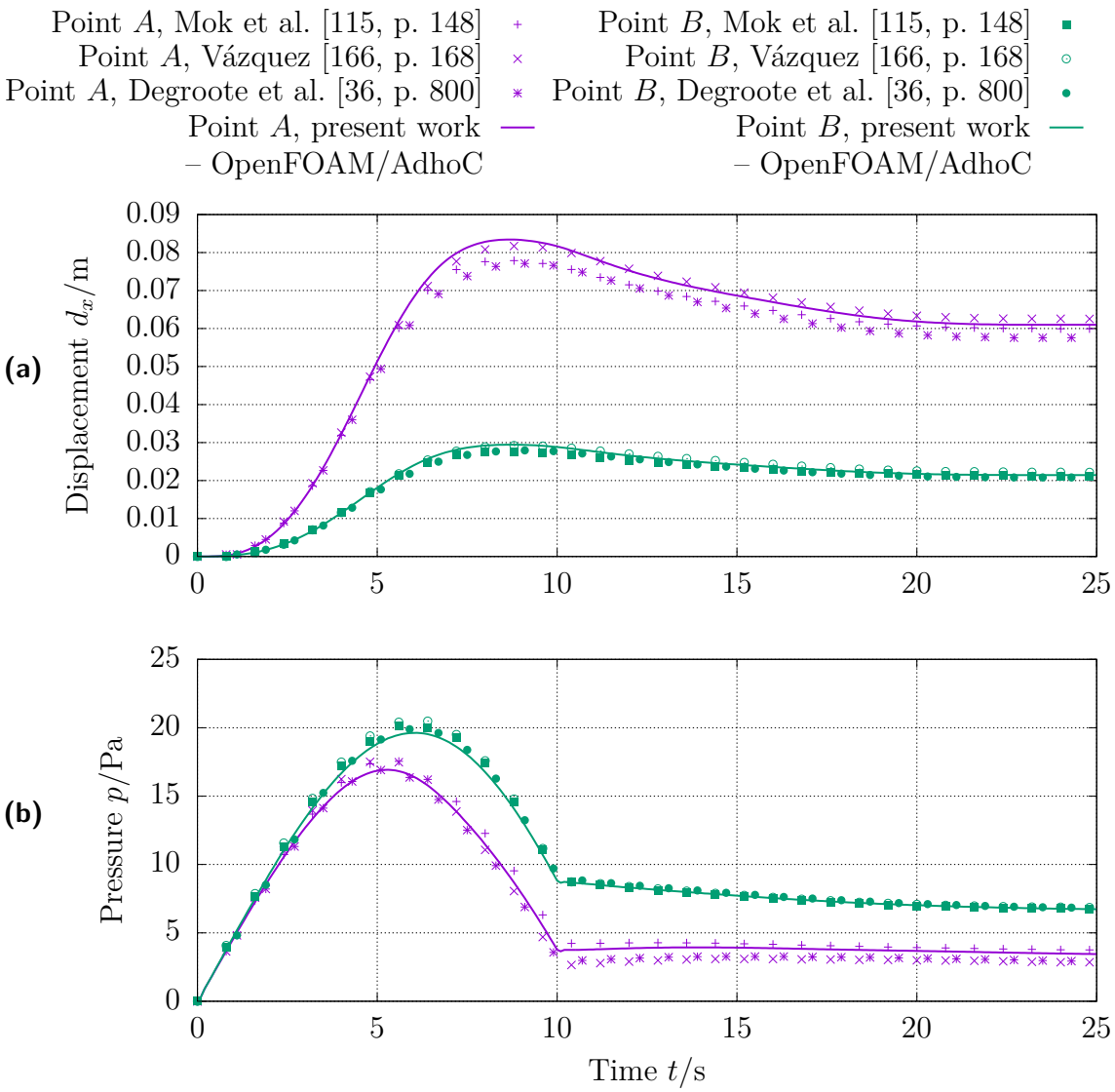


Figure 6.25: (a) Horizontal displacement d_x and (b) pressure p versus time t at the points A and B for the flexible restrictor in the converging channel.

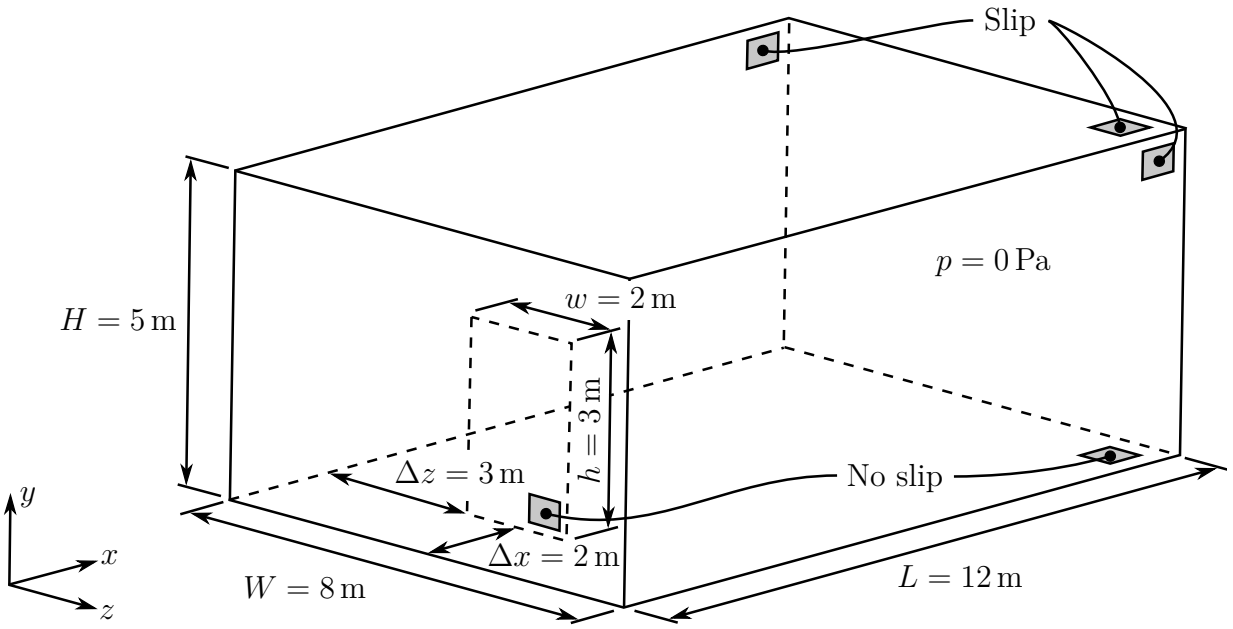


Figure 6.26: Shell in steady-state cross-flow.

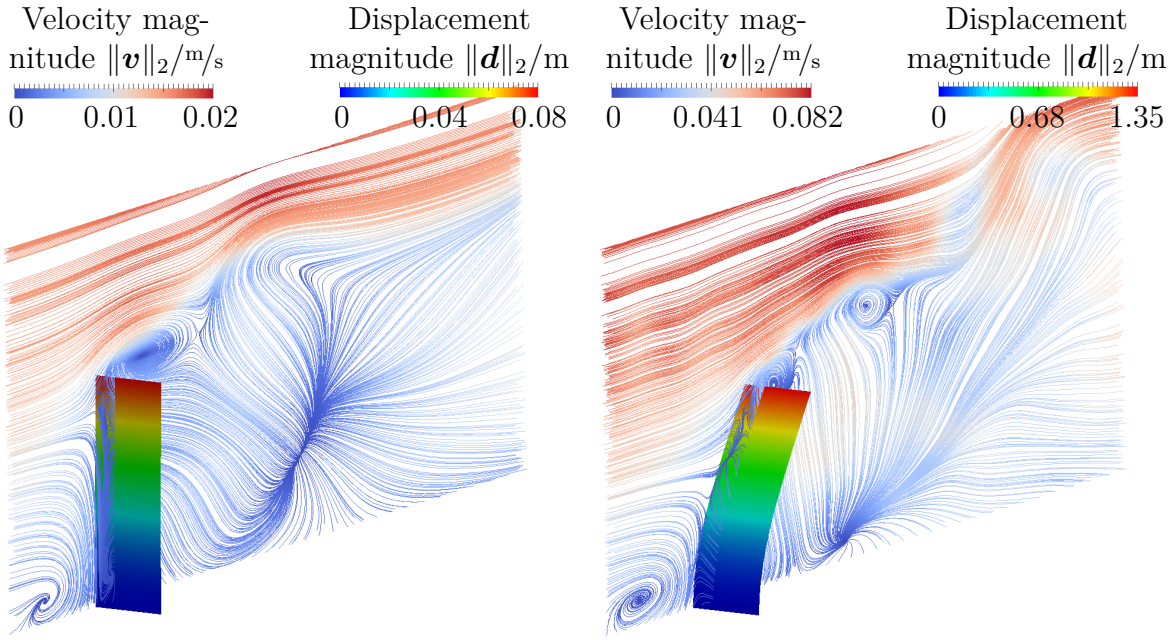
6.7 Shell in Steady-State Cross-Flow

In the next example, which is adopted from [9, pp. 638,640, 11, pp. 609,614, 36, pp. 798–799, 58, pp. 84–89, 94, pp. 92-100], we consider a shell in a steady-state cross-flow. As illustrated in Figure 6.26, the rectangular computational domain under consideration has a length $L = 12\,\mathrm{m}$, width $W = 8\,\mathrm{m}$, and height $H = 5\,\mathrm{m}$. At a distance $\Delta x = 2\,\mathrm{m}$ from the inlet, a thin flexible shell of thickness $t = 1.25 \times 10^{-3}\,\mathrm{m}$ is installed. For the fluid, we choose a density $\rho_\mathrm{f} = 10^3\,\mathrm{kg/m^3}$ and a dynamic viscosity $\mu = 0.1\,\mathrm{Pa\,s}$. For the shell, we use the St. Venant-Kirchhoff material model with a Young's modulus $E = 70 \times 10^9\,\mathrm{N/m^2}$ and Poisson's ratio $\nu_\mathrm{s} = 0.3$. At the inlet, a parabolic velocity profile

$$v(y) = \frac{3\bar{v}}{2H^2}\left(2Hy - y^2\right)\,, \tag{6.5}$$

is prescribed. In what follows, the same example is computed for varying mean inflow velocities $0.01\,\mathrm{m/s} \leq \bar{v} \leq 0.1\,\mathrm{m/s}$ in steps of $\Delta\bar{v} = 0.01\,\mathrm{m/s}$. On the shell and at the bottom, a no-slip condition is imposed, while a slip condition is applied at the back, front, and top wall. At the outlet, $p = 0\,\mathrm{Pa}$ is taken as a fixed reference pressure. In order to save computational costs, we consider only half of the problem, and apply symmetry boundary conditions at the midplane at $z = 0\,\mathrm{m}$.

For the spatial discretization of the fluid domain, we employ a rather coarse FV mesh consisting of 8,064 cells, graded towards the shell to resolve the flow phenomena of most interest. The flow solution is accomplished using the pimpleDyMFoam solver from the open-source CFD package foam-extend. For the sake of comparison, the structural problem is computed using two different discretizations. In the first case, the structure is discretized by an FE mesh of 12×6 (refinement level 1) or 24×12 (refinement level 2) eight-noded,



(a) $\bar{v} = 0.01\,\mathrm{m/s}$ **(b)** $\bar{v} = 0.05\,\mathrm{m/s}$

Figure 6.27: Displaced shell in steady-state cross-flow and velocity streamlines at $z = 0\,\mathrm{m}$ for different inlet velocities.

quadratic SHELL281 elements based on Reissner-Mindlin theory, which exhibit three translational and three rotational degrees of freedom per node. For the structural problem, we apply the commercial FEM software ANSYS [3]. In a second study, the structure is discretized by 3 anisotropic high-order hexahedral elements over the height of the shell with a polynomial degree $p_x = 1$, $p_y = 4$, and $p_z = 2$, available in the in-house p-FEM code AdhoC [41, 40]. Regarding the coupled solution procedure, we employ the absolute criterion $\|\mathbf{r}_{j+1}^k\|_2 < \varepsilon_{\mathrm{abs}} = 10^{-6}$ and the relative criterion $\|\mathbf{r}_{j+1}^k\|_2 / \|\mathbf{r}_{j+1}^0\|_2 < \varepsilon_{\mathrm{rel}} = 10^{-2}$ to judge whether convergence is achieved.

Figure 6.27 illustrates the deformed structure and the velocity streamlines on the symmetry plane at $z = 0\,\mathrm{m}$. For the higher inflow velocities, significant displacements of the shell can be observed. In order to compare the numerical results to the reference solution available from [58], the tip displacement d_x in flow direction is graphed versus the Reynolds number $\mathrm{Re} = \bar{v}\rho_{\mathrm{f}} H/\mu$ for increasing values of the mean inflow velocity $\bar{v}$. As depicted in Figure 6.28, the results for the different structural discretizations are in good agreement with each other. They also match the available reference solutions for $\bar{v} = 0.01\,\mathrm{m/s}$ and $\bar{v} = 0.05\,\mathrm{m/s}$ quantitatively.

6.8 Spherical Dome in Channel Flow

All the previous numerical examples involved flat deformable structures only. In order to demonstrate that an FSI analysis can also be carried out for more complex, curved three-dimensional structures, a spherical dome in a channel flow is now considered.

Geometry and main dimensions for this example are given in Figure 6.29a. Due to the symmetry of the problem, we consider only one half of the channel of length $L = 36\,\mathrm{m}$,

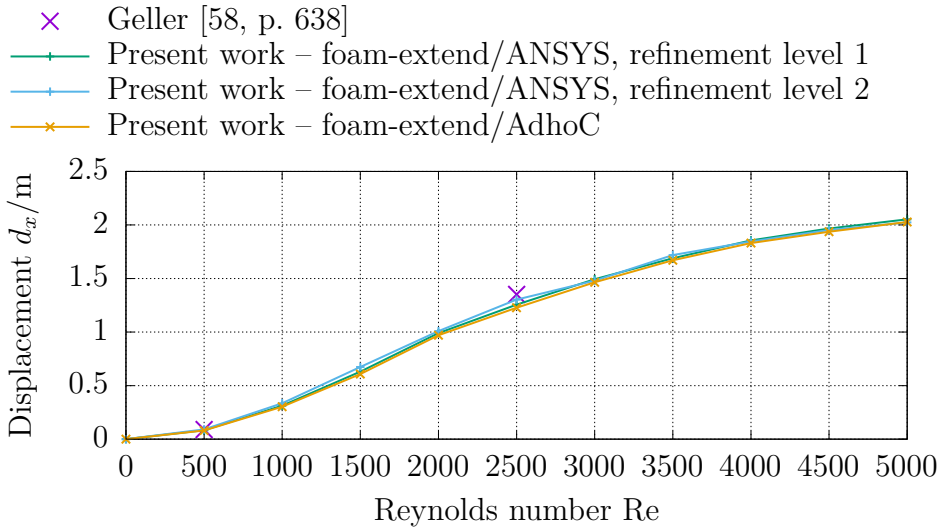


Figure 6.28: Displacement d_x of the tip of the shell in steady-state cross flow for different Reynolds numbers Re.

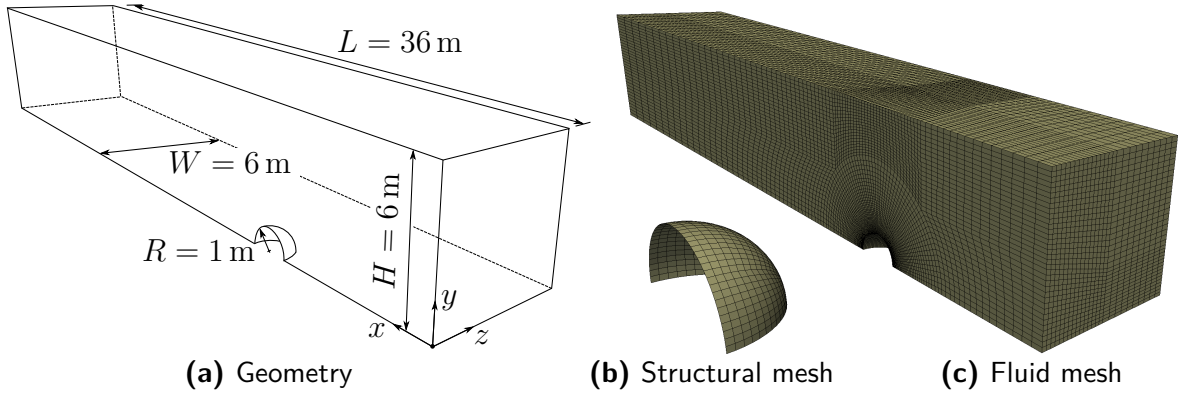
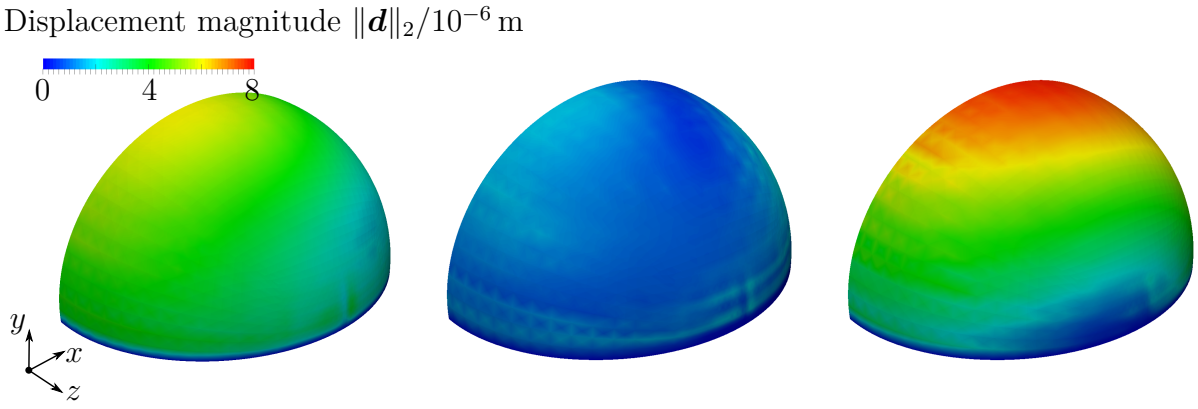


Figure 6.29: (a) Geometry, (b) structural mesh, and (c) fluid mesh for the spherical dome in channel flow.



(a) $t = 0.4\,\mathrm{s}$ **(b)** $t = 0.8\,\mathrm{s}$ **(c)** $t = 1.6\,\mathrm{s}$

Figure 6.30: Deflections of the spherical dome at different instant of time t.

width $W = 6\,\mathrm{m}$, and height $H = 6\,\mathrm{m}$. At $\Delta x = 9\,\mathrm{m}$ from the channel inlet, a spherical dome of radius $R = 1\,\mathrm{m}$ and thickness $t = 0.05\,\mathrm{m}$ is located. For the incompressible Newtonian fluid, we assume a density $\rho_\mathrm{f} = 10^3\,\mathrm{kg/m^3}$ and a kinematic viscosity $\nu_\mathrm{f} = 10^{-3}\,\mathrm{m^2/s}$. For the structure, we use a St. Venant-Kirchhoff material with density $\rho_\mathrm{s} = 10^3\,\mathrm{kg/m^3}$, Young's modulus $E = 7\times10^7\,\mathrm{N/m^2}$, and Poisson's ratio $\nu_\mathrm{s} = 0.35$. At the midplane plane at $z = 0\,\mathrm{m}$, symmetry boundary conditions are applied. At the inlet, we prescribe a parabolic velocity profile

$$v(z) = \frac{\hat{v}}{H^2}\left(2Hz - z^2\right)\,, \tag{6.6}$$

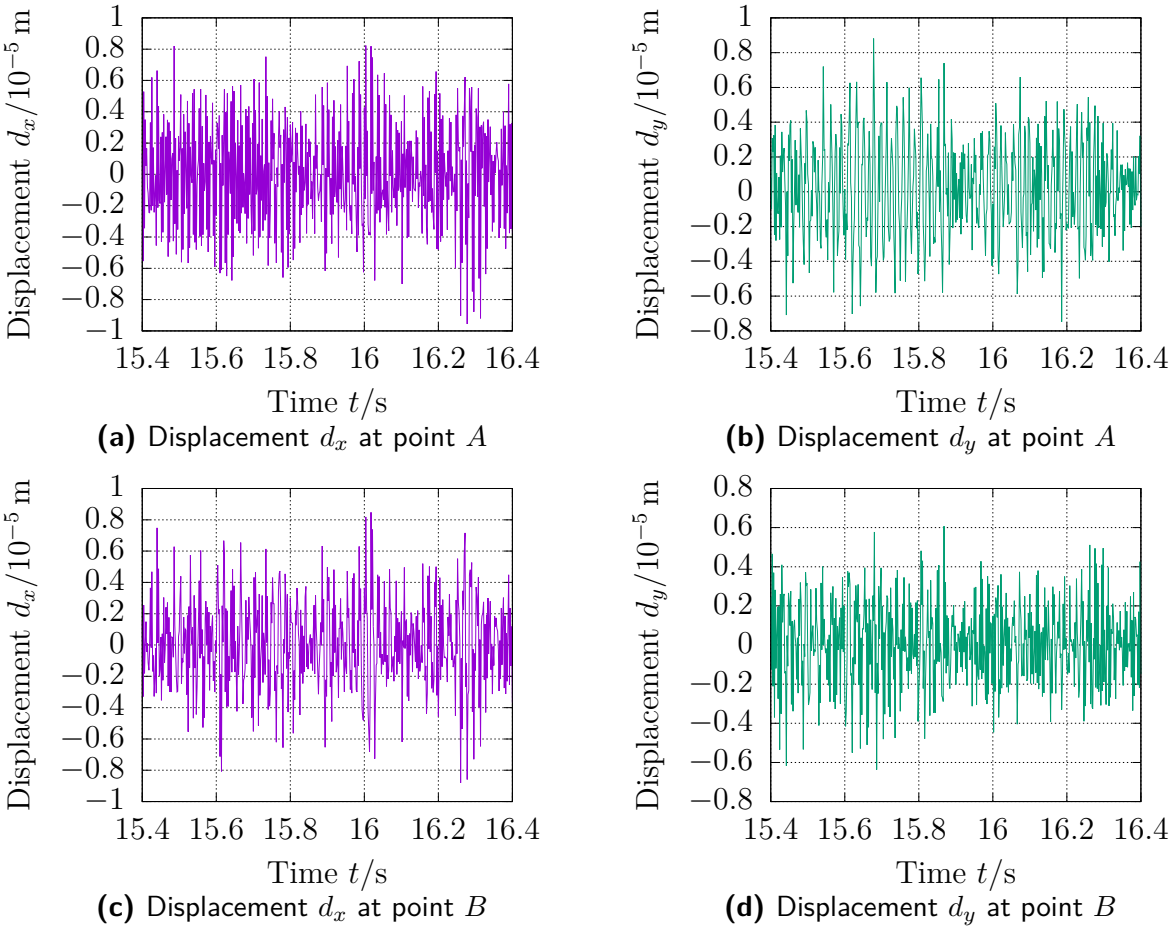
where $\hat{v} = 1\,\mathrm{m/s}$ denotes the maximum velocity at $z = H$. At the outlet, a fixed reference pressure $p = 0\,\mathrm{Pa}$ is imposed. Slip conditions are applied on all other boundaries.

For the solution of the flow problem, we employ a hex-dominant FV mesh comprising 144,125 cells and the open-source CFD package OpenFOAM [123]. As usual, the FEM is chosen for the spatial discretization of the structure. Due to the thin walls of the membrane, the ratio of the thickness to the lateral dimension is high and the use of shell elements is advisable. Here, the structure is discretized by bilinear four-noded SHELL181 elements based on Reissner-Mindlin theory with three translational and three rotational degrees of freedom per node available in the commercial FEM software ANSYS [3]. Regarding the temporal discretization, the fluid solver uses the backward Euler scheme, while the undamped Newmark scheme with $\beta = 0.25$ and $\gamma = 0.5$ is applied for the structural problem. Both schemes use the same time step size $\Delta t = 2\times10^{-3}\,\mathrm{s} = \mathrm{const}$. An absolute criterion $\|\mathbf{r}^k_{j+1}\|_2 < \varepsilon_\mathrm{abs} = 10^{-4}$ and a relative criterion $\|\mathbf{r}^k_{j+1}\|_2/\|\mathbf{r}^0_{j+1}\|_2 < \varepsilon_\mathrm{rel} = 10^{-3}$ are used for checking convergence.

Figure 6.30 depicts the deflection of the spherical dome at different instants of time. Figure 6.31 graphs the displacement components d_x and d_y versus time for the points $A(\Delta x, R, 0)$ at the top of the spherical dome and $B(\Delta x - R\cos(\pi/4), R\sin(\pi/4), 0)$ at an azimuthal angle $\theta = \pi/4$.

6.9 Pressure Pulse in a Straight Elastic Vessel

In the next example, we consider a laminar flow of an incompressible Newtonian fluid through a straight elastic vessel. Originally proposed by Nobile [120] and Formaggia et al.



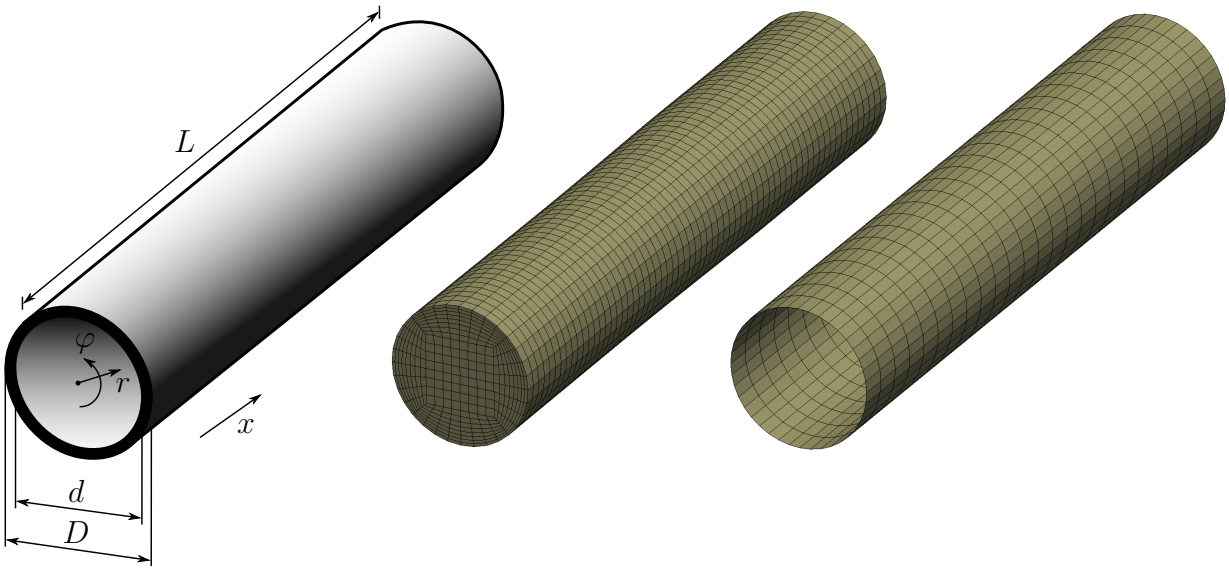
(a) Displacement d_x at point A

(b) Displacement d_y at point A

(c) Displacement d_x at point B

(d) Displacement d_y at point B

Figure 6.31: Displacement components d_x and d_y at the points A and B of the spherical dome over time t.



(a) Geometry **(b)** Fluid mesh **(c)** Structural mesh

Figure 6.32: Geometry, fluid, and structural discretization of the straight elastic vessel.

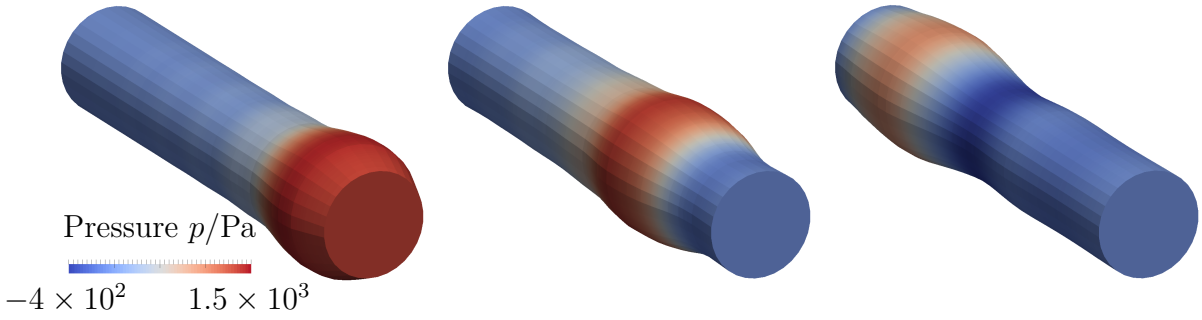
[51], this problem was later also computed in [48, 166, 36], for instance, and represents a simplified model for the FSI in large arterial vessels.

As depicted in Figure 6.32, the vessel has a length $L = 0.05\,\mathrm{m}$, the inner diameter is $d = 10^{-2}\,\mathrm{m}$, and the outer diameter amounts to $D = 1.2 \times 10^{-2}\,\mathrm{m}$ – corresponding to a wall thickness of $\Delta = (D-d)/2 = 10^{-3}\,\mathrm{m}$. For the fluid, we choose a density $\rho_\mathrm{f} = 10^3\,\mathrm{kg/m^3}$ and a dynamic viscosity $\mu = 3 \times 10^{-3}\,\mathrm{Pa\,s}$. The vessel wall is modeled using a St. Venant-Kirchhoff material with density $\rho_\mathrm{s} = 1.2 \times 10^3\,\mathrm{kg/m^3}$, Young's modulus $E = 3 \times 10^5\,\mathrm{N/m^2}$, and Poisson's ratio $\nu_\mathrm{s} = 0.3$. The structure is fully clamped at the inlet and outlet. Initially at rest, the fluid is subjected to a pressure pulse $p = 1.3332 \times 10^3\,\mathrm{Pa} = 10\,\mathrm{mmHg}$ in the time interval $0\,\mathrm{s} \leq t \leq 3 \times 10^{-3}\,\mathrm{s}$. From $t > 3 \times 10^{-3}\,\mathrm{s}$ on, the pressure is reduced to $p = 0\,\mathrm{Pa}$.

The fluid region is discretized by 1,600 finite control volumes of hexahedral shape and is solved using the pimpleDyMFoam solver, available in the open-source CFD package OpenFOAM [123], while the structural mesh consists of 1,200 bilinear Reissner-Mindlin SHELL181 elements with three translational and three rotational degrees of freedom per node, available in the commercial FEM software ANSYS [3]. The fluid solver employs the second-order accurate backward differencing scheme, while the structural solver uses the generalized-α scheme and a spectral radius $\rho_\infty = 0.8$. The simulation covers a time period of $T = 0.018\,\mathrm{s}$, discretized into equally spaced time steps of size $\Delta t = 10^{-4}\,\mathrm{s}$. In the implicit iteration of the coupled solution procedure, we use an absolute convergence criterion $\|\mathbf{r}_{j+1}^k\|_2 < \varepsilon_\mathrm{abs} = 10^{-6}$ and a relative criterion $\|\mathbf{r}_{j+1}^k\|_2 / \|\mathbf{r}_{j+1}^0\|_2 < \varepsilon_\mathrm{rel} = 10^{-3}$ to decide whether the fluid and the structural field are equilibrated with each other to sufficient accuracy.

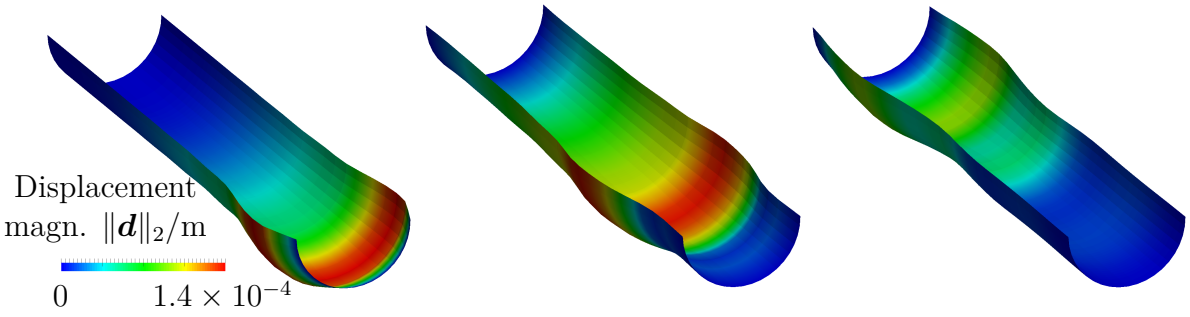
Figure 6.33 shows snapshots of the pressure wave propagating through the compliant vessel at different instants of time t. For illustrative purposes, the displacement was magnified by a scaling factor $s = 10$. Figure 6.34 shows the vessel in its deformed configuration at different instants of time.

Lastly, Figure 6.35 graphs the pressure p and the radial displacement d_r at fixed loca-



(a) $t = 0.0025\,\mathrm{s}$ **(b)** $t = 0.005\,\mathrm{s}$ **(c)** $t = 0.01\,\mathrm{s}$

Figure 6.33: Pressure wave propagating through the elastic vessel.



(a) $t = 0.0025\,\mathrm{s}$ **(b)** $t = 0.005\,\mathrm{s}$ **(c)** $t = 0.01\,\mathrm{s}$

Figure 6.34: Deformed configuration of the elastic vessel at different instants of time t.

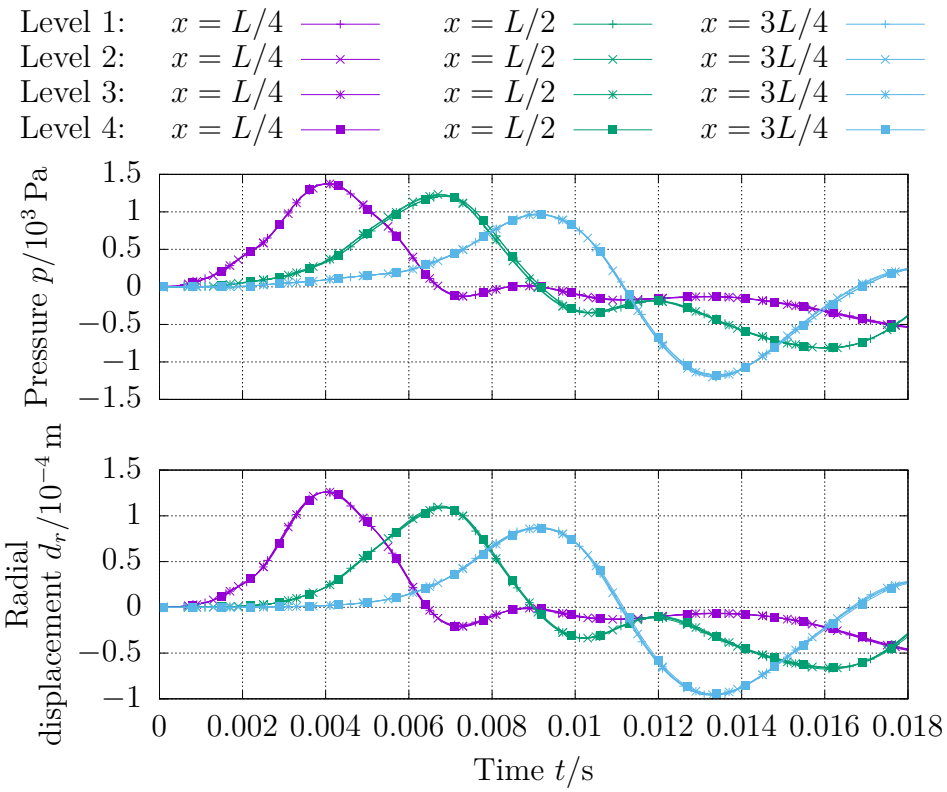


Figure 6.35: Variation of pressure p and radial displacement d_r at different positions $x \in \{L/4, L/2, 3L/4\}$ over time t.

tions $x \in \{L/4, L/2, 3L/2\}$ over time t for different refinement levels of the fluid and the structural mesh. No significant deviations between the solutions can be observed; hence, the solution can be considered converged in space.

6.10 Floating Object in Free-Surface Flow

Especially for maritime applications, the numerical simulation of free-surface flows is of particular interest. In this benchmark problem, which is adopted from a tutorial provided along with the open-source CFD package OpenFOAM [123], we therefore study the motion of a rigid floating object in a free-surface flow. Referring to Figure 6.36, the considered cuboidal computational domain has a side length $W = H = 1\,\mathrm{m}$. Up to $H_1 = 0.5368\,\mathrm{m}$, the domain is filled with water. At the front right edge, a fluid soil of side lengths $a = 0.3\,\mathrm{m}$ and $b = 0.2\,\mathrm{m}$ and height $c = 0.1132\,\mathrm{m}$ is located and dropped at the beginning of the simulation. A rigid floating body of side length $w = 0.3\,\mathrm{m}$ and height $h = 0.1333\,\mathrm{m}$ is placed in the center of the computational domain.

We assume a density $\rho_\mathrm{w} = 998\,\mathrm{kg/m^3}$ and a kinematic viscosity $\nu_\mathrm{w} = 10^{-6}\,\mathrm{m^2/s}$ for the water phase, while we choose a density $\rho_\mathrm{a} = 1\,\mathrm{kg/m^3}$ and a kinematic viscosity $\nu_\mathrm{a} = 1.48 \times 10^{-5}\,\mathrm{m^2/s}$ for the air phase filling the rest of the computational domain. For the rigid floating object, we take a mass $m = 9.6\,\mathrm{kg}$ and moments of inertia $\Theta_x = \Theta_y = 0.086\,\mathrm{kg\,m^2}$ and $\Theta_z = 0.144\,\mathrm{kg\,m^2}$ about the center of mass at $C(0.5, 0.5, 0.5)$.

At the stationary walls and on the floating object, we use a fixed flux pressure boundary

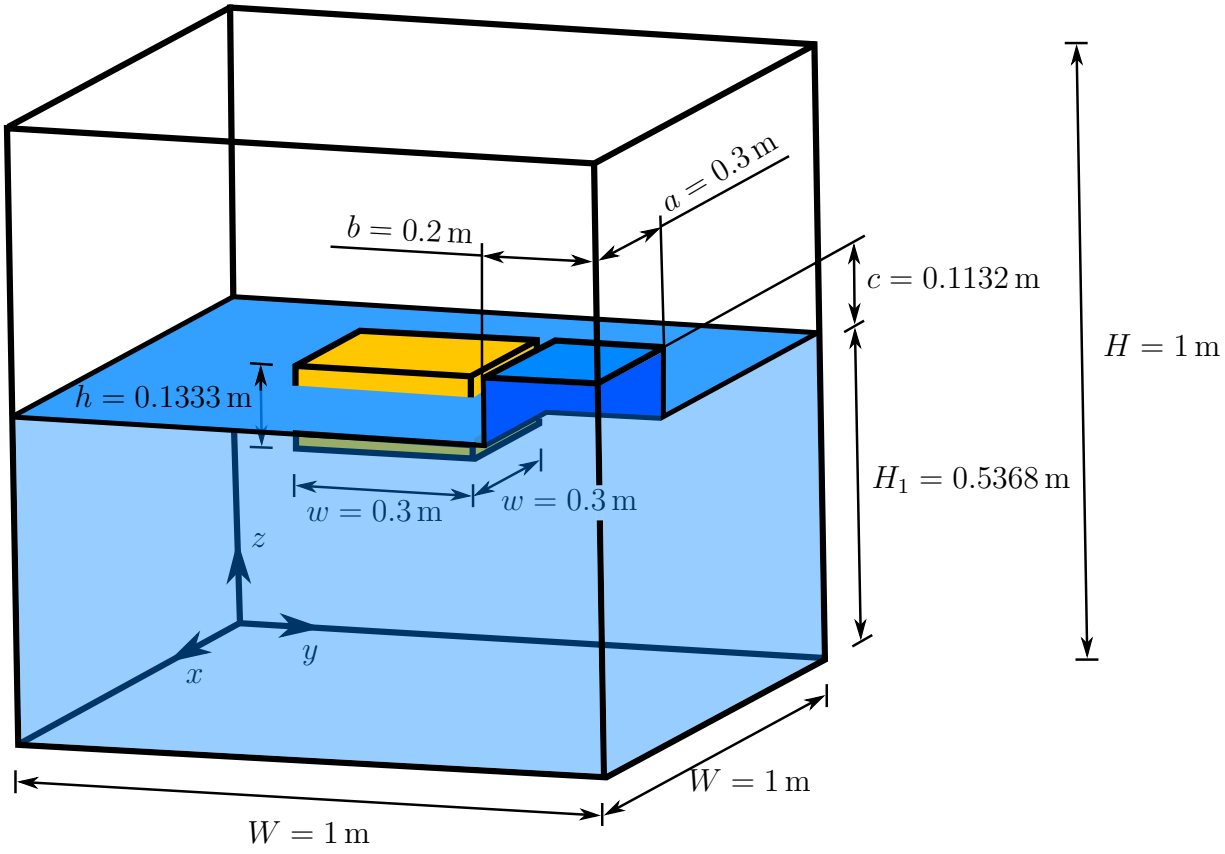


Figure 6.36: Floating object in free-surface flow.

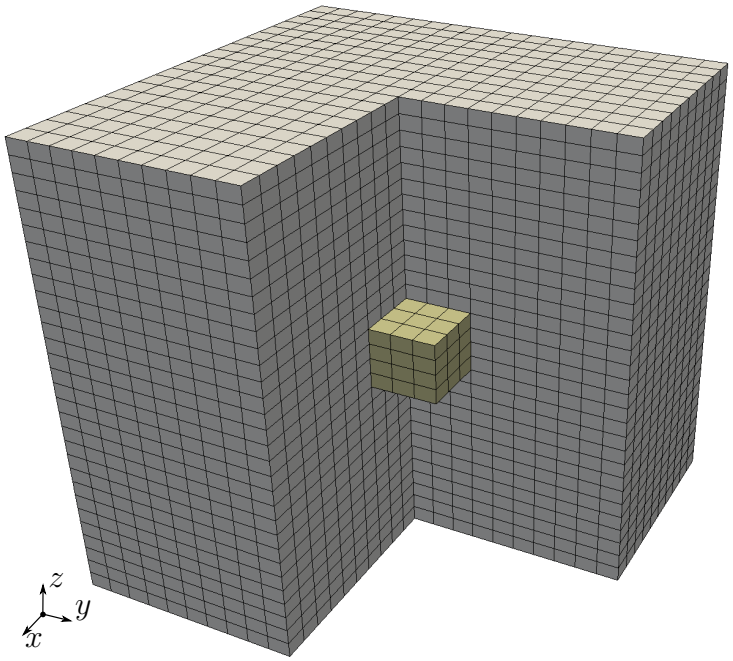
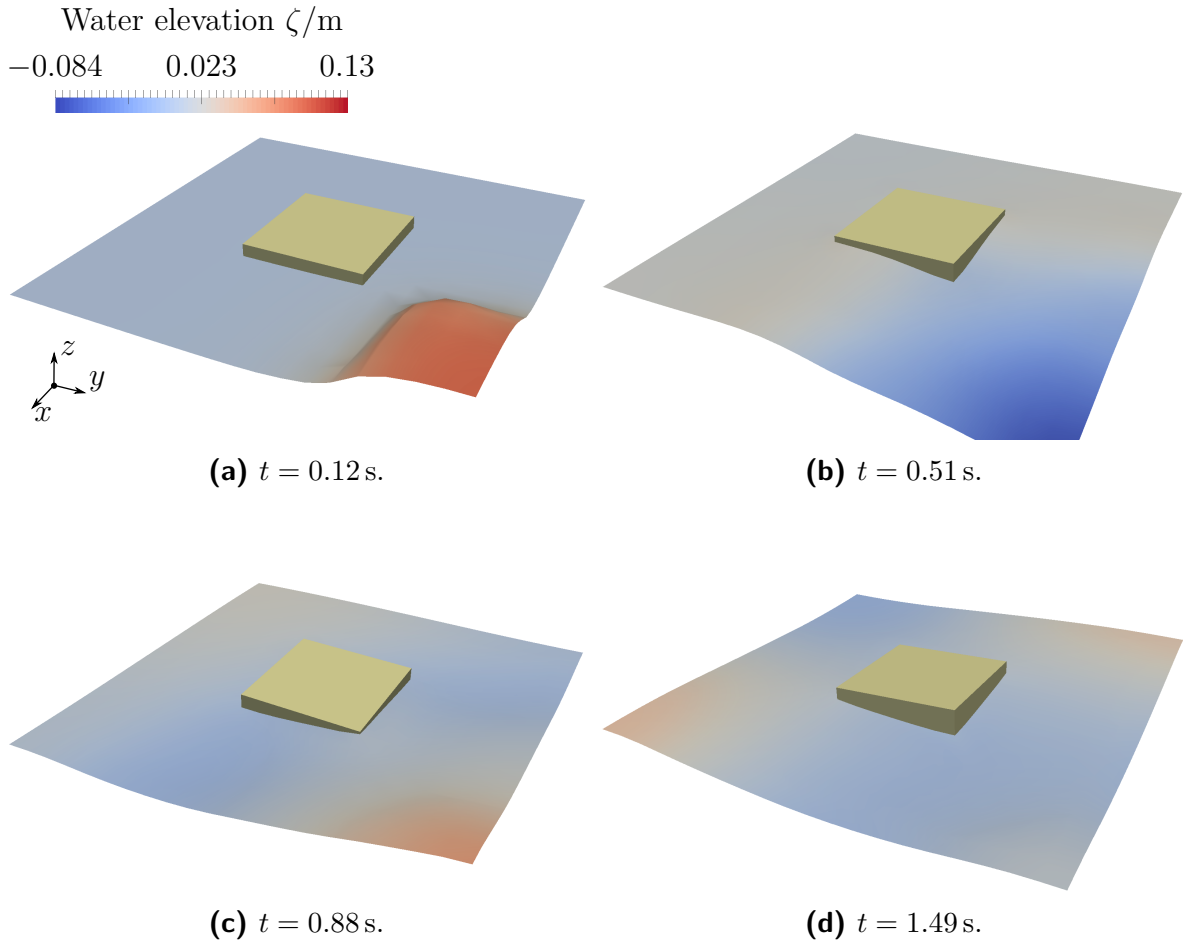


Figure 6.37: Discretization of the fluid and the structural region for the floating object in free-surface flow.

condition, adjusting the gradient of the pressure in such a way that the boundary flux matches the one specified by the velocity boundary condition. At the top of the domain, we impose a fixed uniform atmospheric pressure $p = 0\,\text{Pa}$. Following the notion in 3.2, surge, sway, yaw, and roll motion for the floating object are constrained to zero; heave and pitch are left free.

For the fluid, we utilize the FVM and a purely hexahedral mesh comprising 11,856 cells, as illustrated in Figure 6.37. The fluid problem is solved employing the volume of fluid (VOF) technique available in the interDyMFoam multiphase solver as part of the open-source CFD package OpenFOAM [123]. For the structural problem, we use the rigid body solver distributed along with the commercial software ANSYS [3]. For the sole purpose of integrating the fluid traction over the floating object's surface, a surface mesh consisting of 170 quadrilateral cells is introduced. Regarding the time stepping procedures, we utilize the Euler backward scheme for the fluid problem, whereas we apply the Newmark method with the parameters $\beta = 0.2525$ and $\gamma = 0.5050$ for the structural problem.

Figure 6.38 gives an impression of the evolution of the free surface and the motion response of the floating object at different instants of time. In order to verify the results obtained for this problem, we track the heave and pitch motion of the floating object over time and compare it to the solution produced by OpenFOAM only, which likewise employs a partitioned solution procedure internally to couple the fluid and the rigid body problem. As shown in Figure 6.39, both motion plots apparently are in perfect agreement with each other.



(a) $t = 0.12\,\mathrm{s}$. **(b)** $t = 0.51\,\mathrm{s}$.

(c) $t = 0.88\,\mathrm{s}$. **(d)** $t = 1.49\,\mathrm{s}$.

Figure 6.38: Displaced floating object in free-surface flow at different instants of time t.

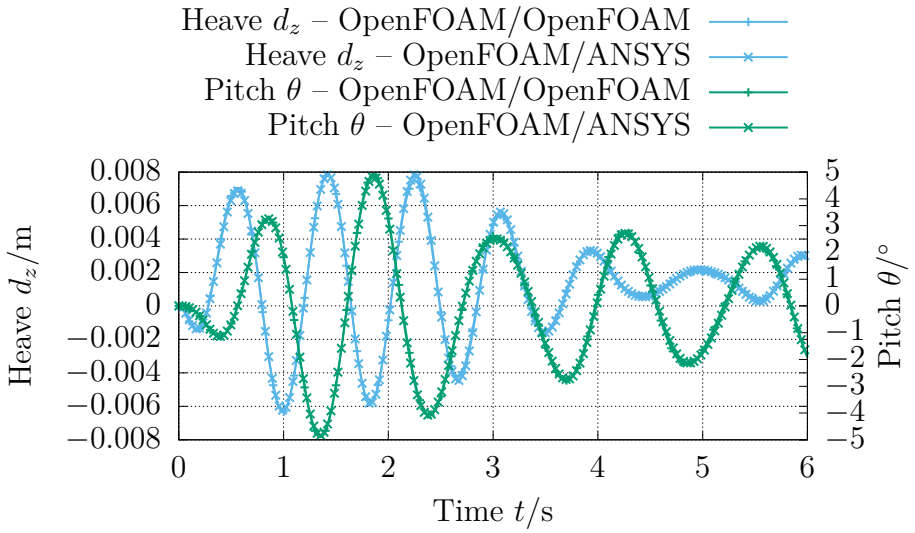
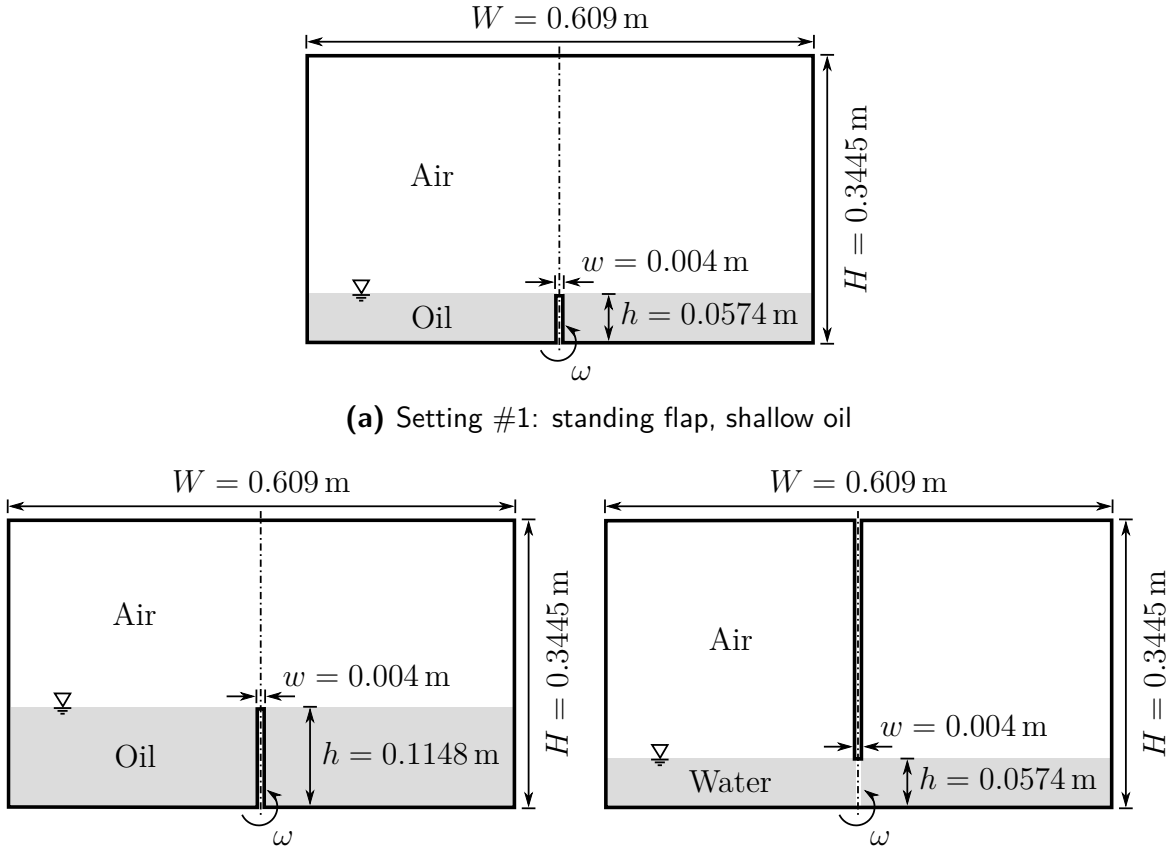


Figure 6.39: Heave d_z and pitch θ of the floating object versus time t.



(a) Setting #1: standing flap, shallow oil

(b) Setting #2: standing flap, deep oil

(c) Setting #3: hanging flap, shallow water

Figure 6.40: Different settings for the partly-filled rolling tank.

6.11 Sloshing Effects in Partly-Filled Tank

Sloshing effects in partly-filled tanks are of great importance in the maritime industry. Wave-induced ship motions are likely to induce resonance effects in the fluid, which can lead to high local impact loads on the tank and may also affect the global ship motion [76, p. 406]. In order to demonstrate that the FSI of a complex free-surface flow and an elastic structure can be computed accurately, a rolling tank with a flexible flap is considered in this example. Originally proposed by Idelsohn et al. [77], this benchmark problem was later also computed by Degroote [35, p. 202–210] to verify the coupling code and the subproblem solvers. As pictured in Figure 6.40, three different settings are considered, in each of which the fluid interacts with a flexible flap installed in the middle of the tank. In all cases, the tank exhibits a width $W = 0.609\,\mathrm{m}$ and a height $H = 0.3445\,\mathrm{m}$. In setting #1, the flexible structure has a width $w = 0.004\,\mathrm{m}$ and a height $h = 0.0574\,\mathrm{m}$, and the tank is filled with oil of density $\rho_\mathrm{l} = 917\,\mathrm{kg/m^3}$ and dynamic viscosity $\mu_\mathrm{l} = 0.04585\,\mathrm{Pas}$ up to a filling height h coinciding with the height of the flap, which has a density $\rho_\mathrm{s} = 1.1 \times 10^3\,\mathrm{kg/m^3}$, Young's modulus $E = 6 \times 10^6\,\mathrm{N/m^2}$, and Poisson's ratio $\nu_\mathrm{s} = 0.49$. In setting #2, the height of the structure and the filling level are increased to $h = 0.1148\,\mathrm{m}$. In case #3, we consider a hanging flap of density $\rho_\mathrm{s} = 1.1 \times 10^3\,\mathrm{kg/m^3}$, Young's modulus $E = 6 \times 10^6\,\mathrm{N/m^2}$, and Poisson's ratio $\nu_\mathrm{s} = 0.49$. Here, the tank is partly filled with water of density $\rho_\mathrm{l} = 998.2\,\mathrm{kg/m^3}$ and kinematic viscosity $\mu_\mathrm{l} = 0.001003\,\mathrm{Pa\,s}$. In all three settings, the rest of the tank is filled

Table 6.4: Material parameters for the liquid and gas phase and the structure in the rolling tank example.

	Parameter	Standing flap, shallow oil	Standing flap, deep oil	Hanging flap, shallow water
Liquid phase	Density $\rho_\mathrm{l}/\mathrm{kg/m^3}$	917	917	998.2
	Dynamic viscosity $\mu_\mathrm{l}/\mathrm{Pa\,s}$	0.04585	0.04585	0.001003
Gas phase	Density $\rho_\mathrm{g}/\mathrm{kg/m^3}$	1.225	1.225	1.225
	Dynamic viscosity $\mu_\mathrm{g}/\mathrm{Pa\,s}$	1.79×10^{-5}	1.79×10^{-5}	1.79×10^{-5}
Structure	Density $\rho_\mathrm{s}/\mathrm{kg/m^3}$	1.1×10^3	1.1×10^3	1.9×10^3
	Young's modulus $E/\mathrm{N/m^2}$	6×10^6	6×10^6	4×10^6
	Poisson's ratio ν_s	0.49	0.49	0.49

with air of density $\rho_\mathrm{g} = 917\,\mathrm{kg/m^3}$ and a dynamic viscosity $\mu_\mathrm{g} = 0.04585\,\mathrm{Pas}$. Table 6.4 once again summarizes the material parameters for this example.

At the bottom of the tank, a rolling motion of angular frequency

$$\omega = \sqrt{\frac{\pi g}{W} \tanh \frac{\pi h}{W}} \tag{6.7}$$

is imposed. Herein, $g = 9.81\,\mathrm{m/s^2}$ is the gravitational acceleration. On the side and bottom walls, we impose a no-slip condition for the velocity and a fixed flux boundary condition for the pressure, where the gradient of the pressure is adjusted such that the boundary flux matches the velocity boundary condition. On the top wall, a fixed reference pressure $p = 0\,\mathrm{Pa}$ is prescribed. A no-slip condition for the velocity and a zero gradient condition for the pressure are applied on the flexible flap.

As illustrated in Figure 6.41, the fluid region is discretized by a structured FV grid consisting of 10,554 (standing flap, shallow oil), 10,240 (standing flap, deep oil), or 10,228 (hanging flap, shallow water) hexahedral control volumes. Since this is a multiphase problem, it is solved using the interDyMFoam solver from the OpenFOAM [123] CFD package. For the structural domain, we employ an FE mesh comprising 3 high-order plane-stress elements of polynomial order $p_x = 2$ and $p_y = 6$, solved using the in-house p-FEM code AdhoC [41, 40]. In the flow field, we apply the Euler backward scheme for time integration, whereas the Newmark scheme with $\beta = 0.25$ and $\gamma = 0.5$ is chosen for the structure. Both problems are integrated using a time step size $\Delta t = 2.5 \times 10^{-3}\,\mathrm{s}$ (standing flap, shallow oil and standing flap, deep oil) or $\Delta t = \times 10^{-3}\,\mathrm{s}$ (standing flap, shallow water).

A comparison between the experiments reported by Idelsohn et al. [77] and the numerical results obtained in this work is shown in Figure 6.42–6.44. In general, a fair agreement between the measurements and the numerical solution can be observed. In addition, Figure 6.45 graphs the horizontal displacement $d_{\hat{x}}$ of the flag tip in the rotating reference frame of the tank over time t. Here, we can observe an excellent correspondence between the numerical results reported by Degroote [35, p. 209] and the solution obtained in the present work.

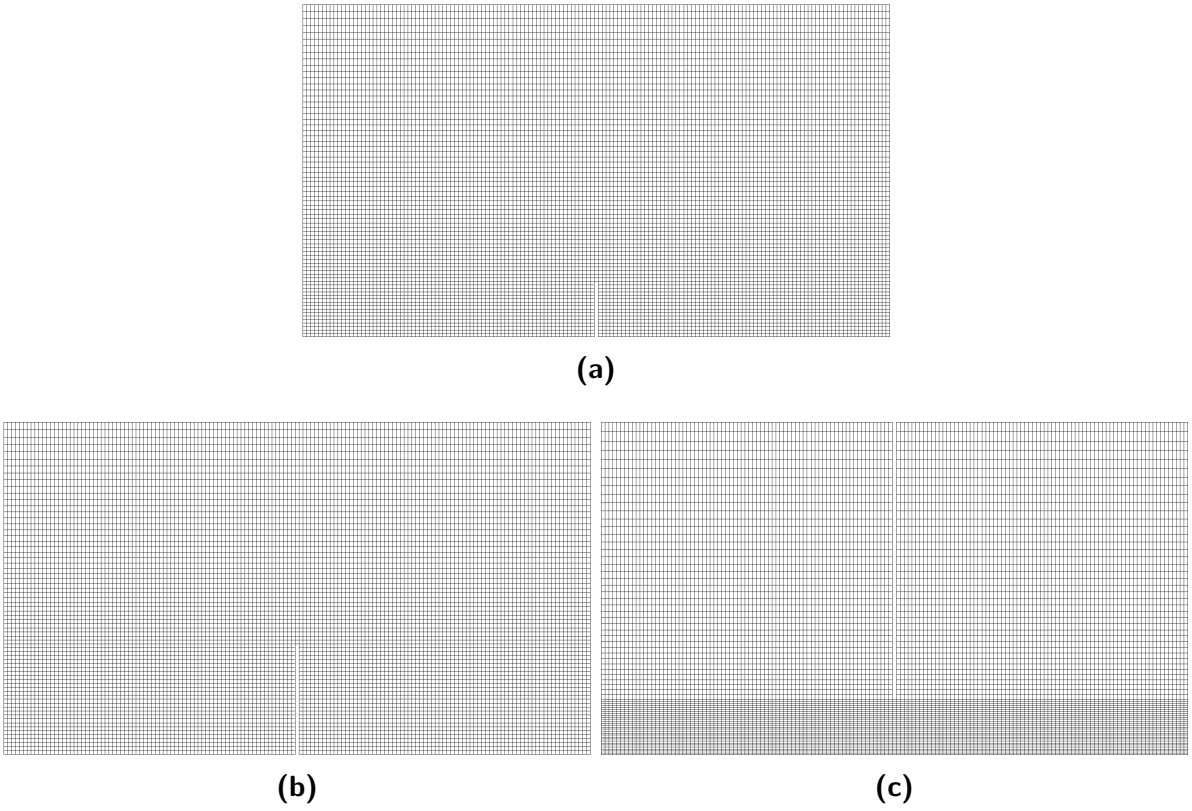
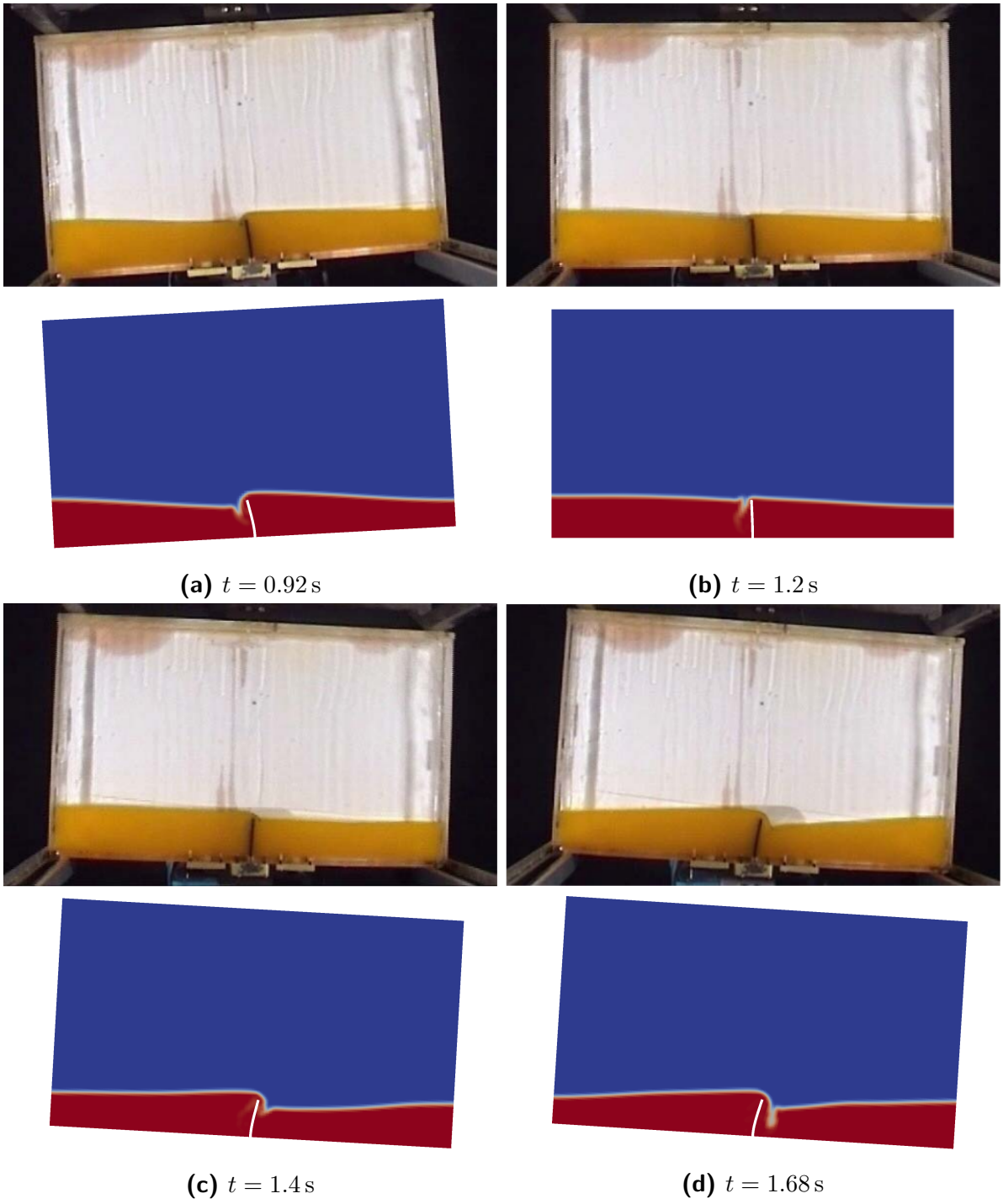


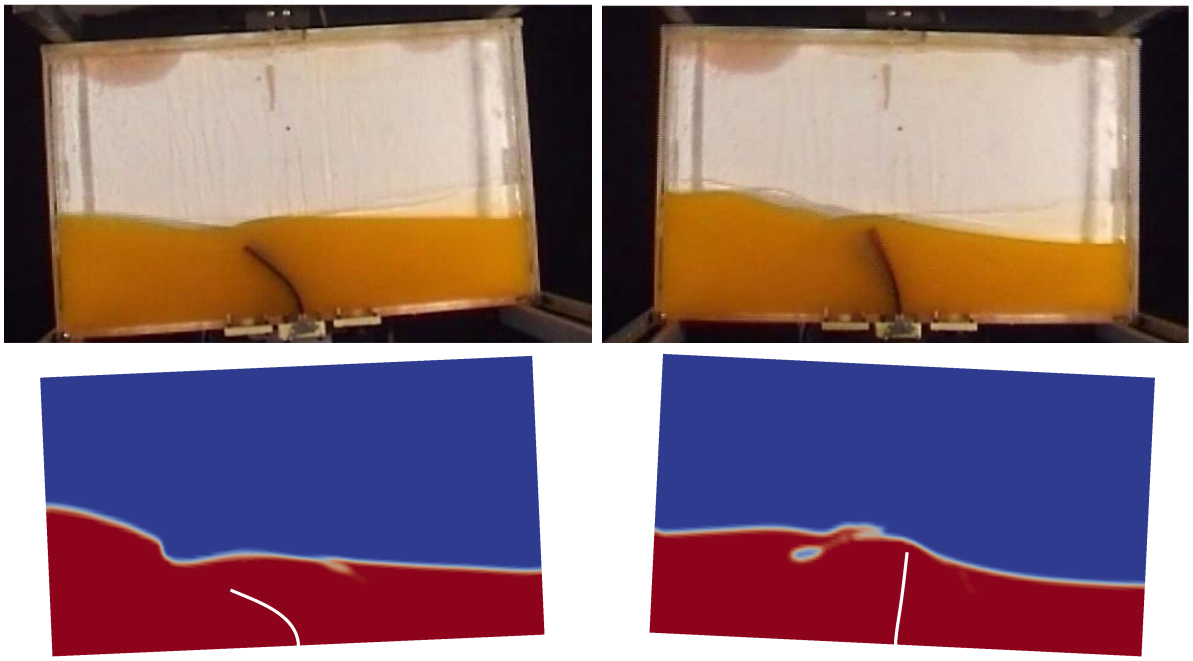
Figure 6.41: Fluid meshes for the rolling tank with (a) standing flap and shallow oil, (b) standing flap and deep oil, and (c) hanging flap and shallow water.



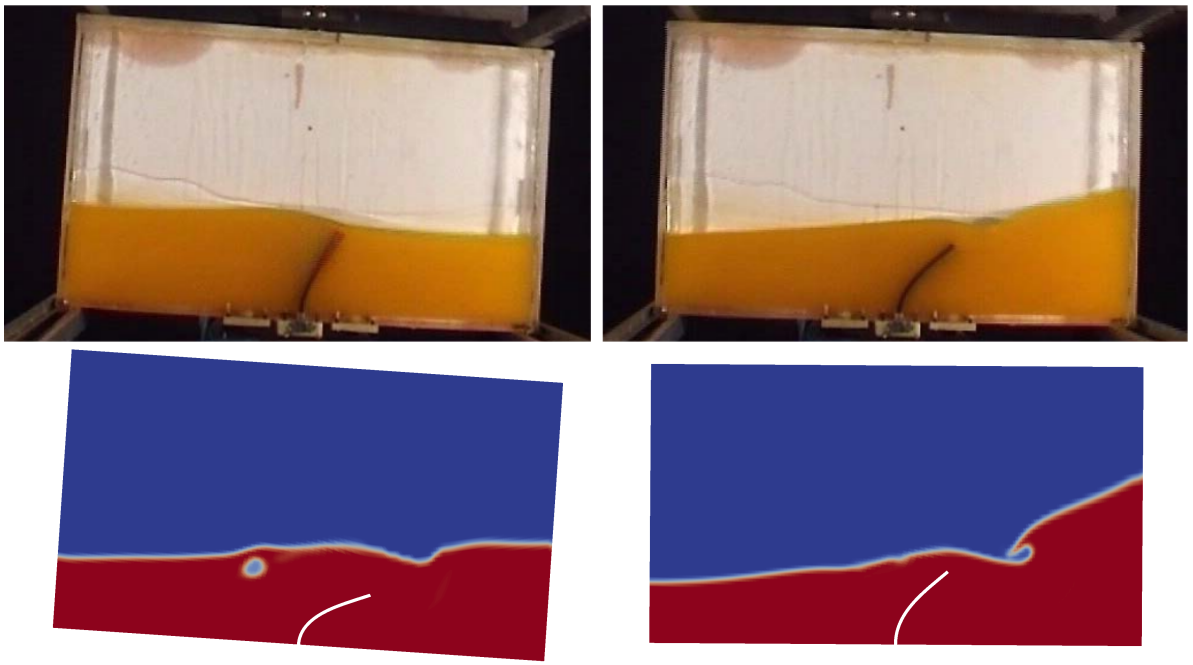
(a) $t = 0.92\,\mathrm{s}$ **(b)** $t = 1.2\,\mathrm{s}$

(c) $t = 1.4\,\mathrm{s}$ **(d)** $t = 1.68\,\mathrm{s}$

Figure 6.42: Comparison of experimental measurements and numerical results for the rolling tank with standing flap and shallow oil at different instants of time t.

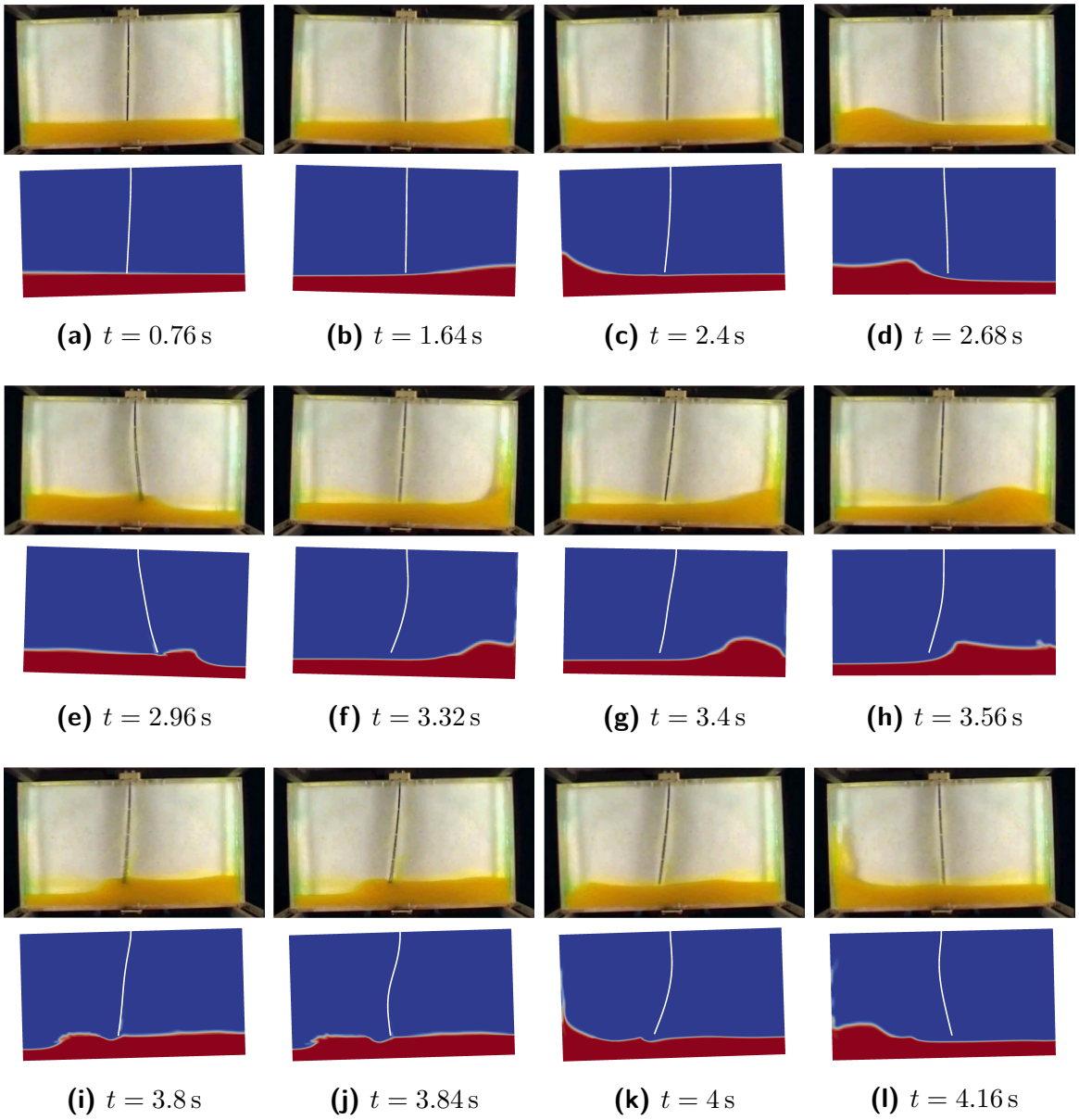


(a) $t = 1.84\,\mathrm{s}$ **(b)** $t = 2.12\,\mathrm{s}$



(c) $t = 2.32\,\mathrm{s}$ **(d)** $t = 2.56\,\mathrm{s}$

Figure 6.43: Comparison of experimental measurements and numerical results for the rolling tank with standing flap and deep oil at different instants of time t.



(a) $t = 0.76\,\mathrm{s}$ **(b)** $t = 1.64\,\mathrm{s}$ **(c)** $t = 2.4\,\mathrm{s}$ **(d)** $t = 2.68\,\mathrm{s}$

(e) $t = 2.96\,\mathrm{s}$ **(f)** $t = 3.32\,\mathrm{s}$ **(g)** $t = 3.4\,\mathrm{s}$ **(h)** $t = 3.56\,\mathrm{s}$

(i) $t = 3.8\,\mathrm{s}$ **(j)** $t = 3.84\,\mathrm{s}$ **(k)** $t = 4\,\mathrm{s}$ **(l)** $t = 4.16\,\mathrm{s}$

Figure 6.44: Comparison of experimental measurements and numerical results for the rolling tank with hanging flap and shallow water at different instants of time t.

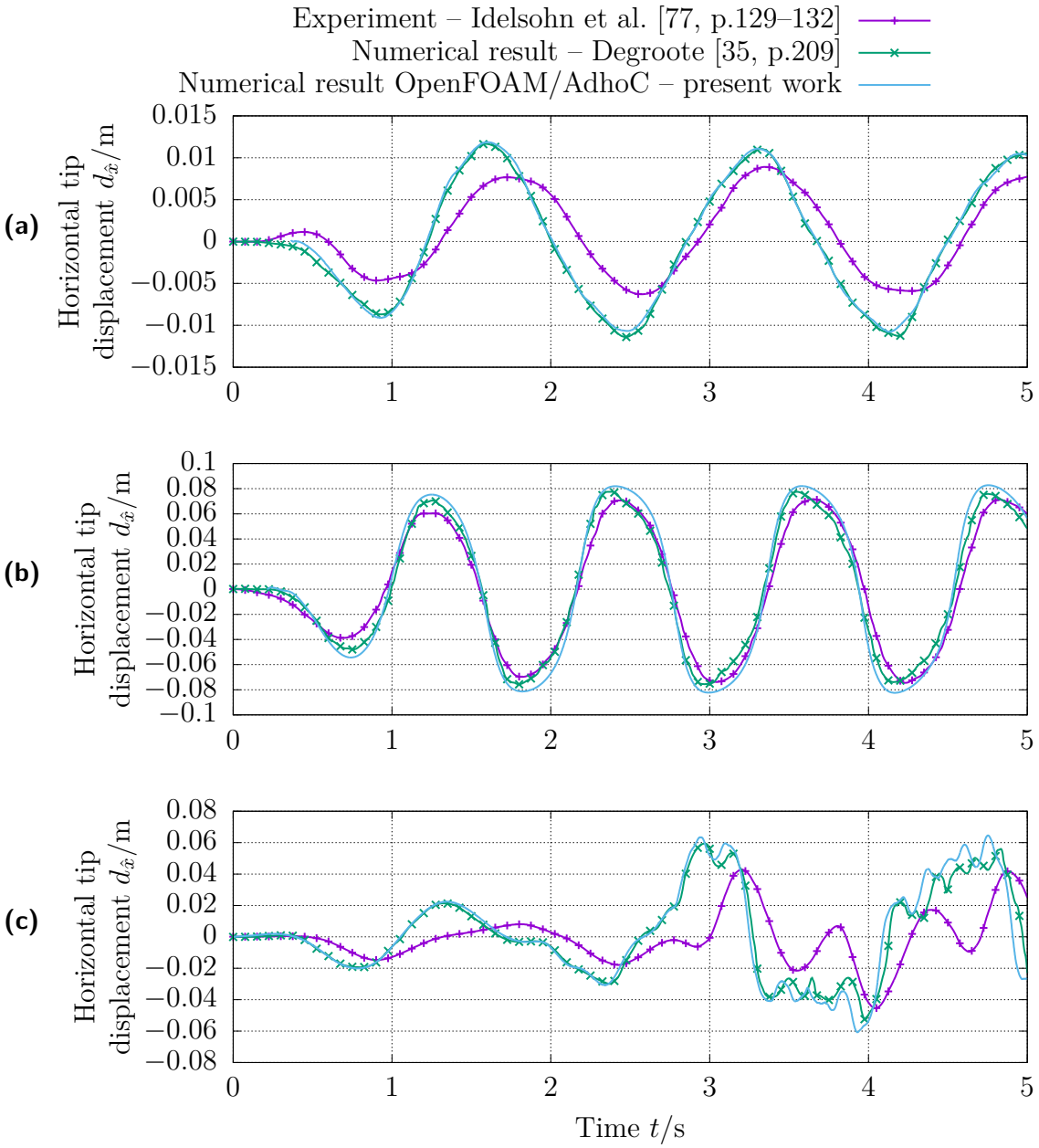
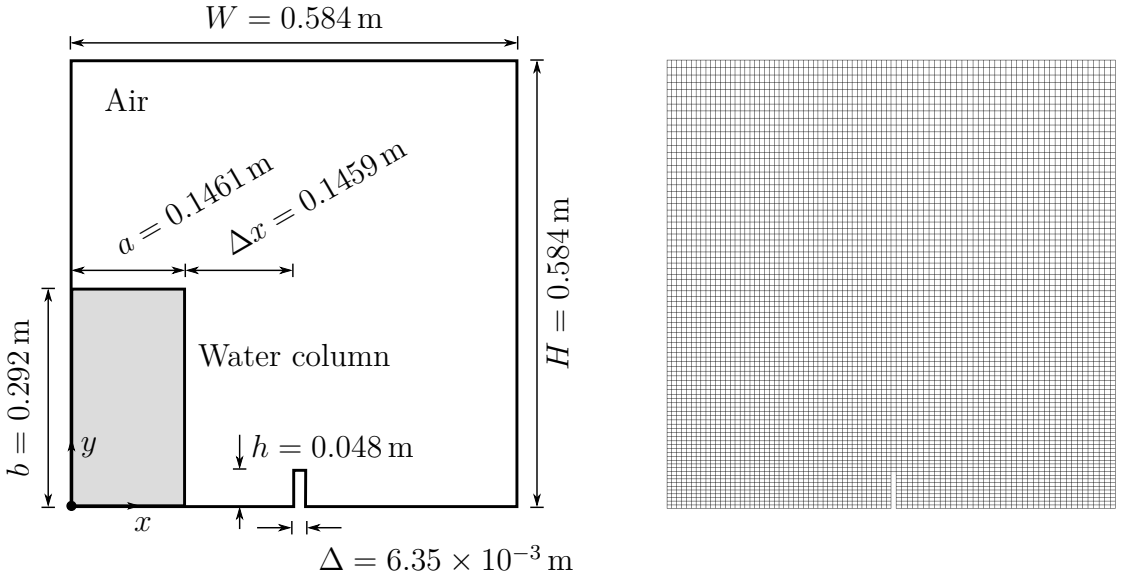


Figure 6.45: Flap tip displacement $d_{\hat{x}}$ in the rotating reference frame for the (a) standing flap in shallow oil, (b) standing flap in deep oil, and (c) hanging flap above shallow water.



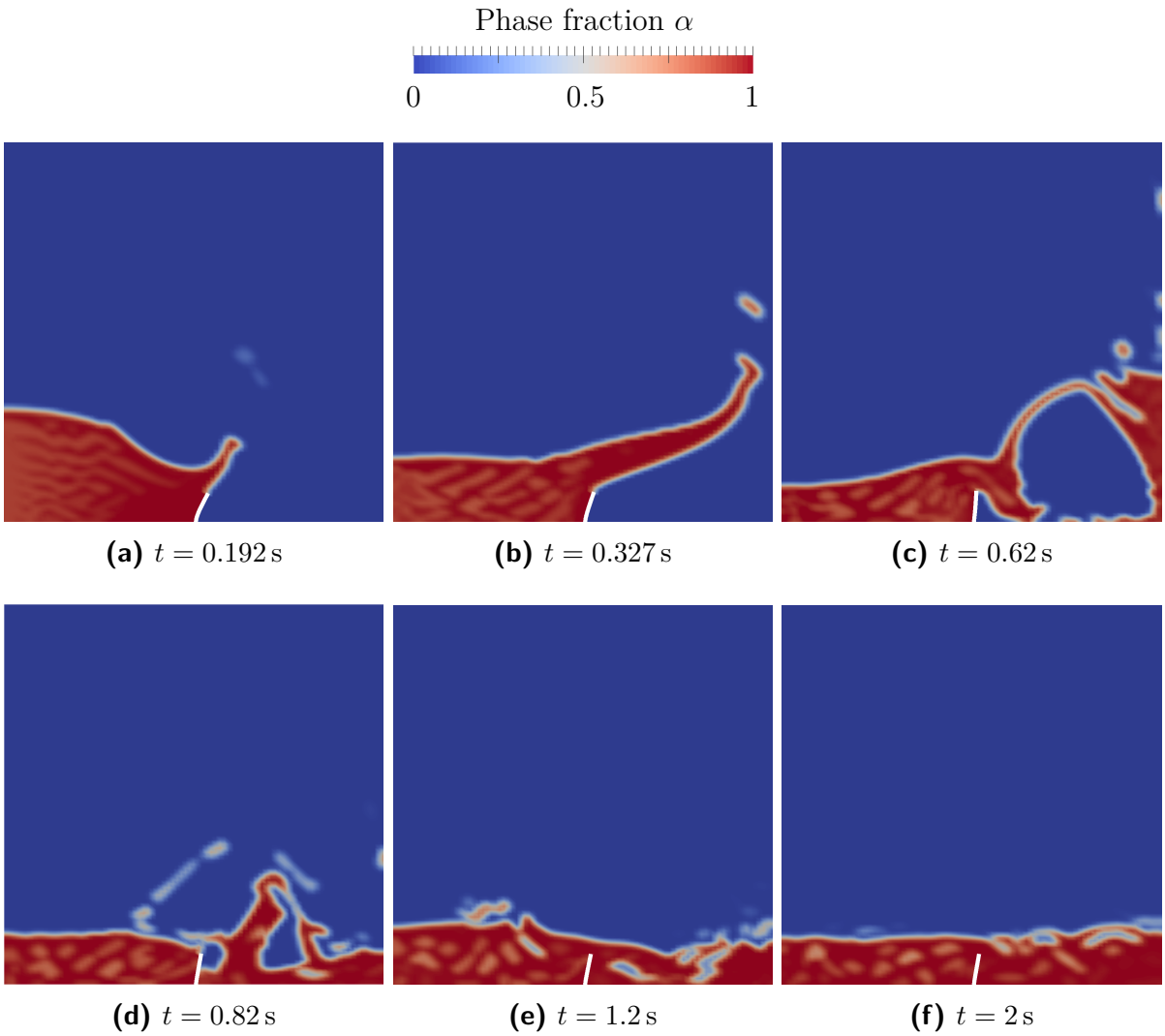
(a) Geometry **(b)** Fluid mesh

Figure 6.46: (a) Geometry and (b) fluid mesh for the dam break problem.

6.12 Dam Break

So far, only elastic deformations of the structure have been considered. In many practical applications, plastic effects play an important role as well. A fluid impacting a structure may, for instance, lead to high loads that exceed the elastic limit of the structure, causing plastic deformations. Due to the use of a partitioned solution approach, a solver supporting the analysis of plasticity problems can easily be applied to solve the structural subproblem. In this example, adopted from [123], we study the plastic deformation of a deformable obstacle subjected to an impact load caused by a breaking dam. According to Figure 6.46, the computational domain has a width $W = 0.584\,\mathrm{m}$ and a height $H = 0.584\,\mathrm{m}$. At the left end of the domain, a water column of width $a = 0.1461\,\mathrm{m}$ and height $b = 0.292\,\mathrm{m}$ is dropped at the beginning of the simulation. The obstacle has a thickness $\Delta = 6.35 \times 10^{-3}\,\mathrm{m}$ and a height $h = 0.048\,\mathrm{m}$. For the water phase, a density $\rho_\mathrm{w} = 10^3\,\mathrm{kg/m^3}$ and a kinematic viscosity $\nu_\mathrm{w} = 10^{-6}\,\mathrm{m^2/s}$ are assumed, while the air phase exhibits a density $\rho_\mathrm{a} = 1\,\mathrm{kg/m^3}$ and a kinematic viscosity $\nu_\mathrm{a} = 1.48 \times 10^{-5}\,\mathrm{m^2/s}$. Regarding the prediction of the effects of turbulence, we use the well-established k-ε model. In order to capture the elastic material behavior of the obstacle, we choose a St. Venant-Kirchhoff model with density $\rho_\mathrm{s} = 10^3\,\mathrm{kg/m^3}$, Young's modulus $E = 5 \times 10^5\,\mathrm{N/m^2}$, and Poisson's ratio $\nu_\mathrm{s} = 0.49$. In the plastic regime, the behavior of the structure is described by a bilinear isotropic hardening model with yield stress $\sigma_0 = 2 \times 10^5\,\mathrm{N/m^2}$ and tangent modulus $E_\mathrm{t} = 5 \times 10^5\,\mathrm{N/m^2}$. No-slip conditions are prescribed at the left, lower, and right boundary of the domain including the surface of the obstacle. At the upper domain boundary, a fixed uniform atmospheric pressure $p = 0\,\mathrm{Pa}$ is imposed.

The fluid problem is discretized by an FV mesh comprising 7,472 cells, solved using the VOF technique available in the interDyMFoam multiphase solver available as part of the open-source CFD package OpenFOAM [123]. For the spatial discretization of the obstacle, we utilize an FE mesh of 576 biquadratic, eight-noded plane-stress PLANE183 elements available in the commercial solver ANSYS [3]. As usual, we use the second-order accurate



(a) $t = 0.192\,\mathrm{s}$ **(b)** $t = 0.327\,\mathrm{s}$ **(c)** $t = 0.62\,\mathrm{s}$

(d) $t = 0.82\,\mathrm{s}$ **(e)** $t = 1.2\,\mathrm{s}$ **(f)** $t = 2\,\mathrm{s}$

Figure 6.47: Evolution of the phase fraction α over time t.

Euler backward scheme as a time stepping procedure for the fluid, while we integrate the structural problem with the Newmark scheme, where $\beta = 0.2525$ and $\gamma = 0.5050$. Starting from an initial time step size $\Delta t = 10^{-3}\,\mathrm{s}$, the time step size is adaptively controlled via the Courant number as a means to save computational costs.

Figure 6.47 depicts the evolution of the phase fraction α over time. The highest deflection is observed at the moment the fluid hits the obstacle, which is also confirmed by the record of the obstacle's tip displacement over time, as shown in Figure 6.48. As illustrated in Figure 6.49, the high impact load leads to a permanent plastic deformation of the structure at the clamping. Due to the fluid flowing around the obstacle, the high pressure difference is alleviated, and the obstacle elastically flips back in negative x-direction. Despite the vanishing pressure difference, the obstacle does not reach its initial state, however, because of the permanent plastic deformation.

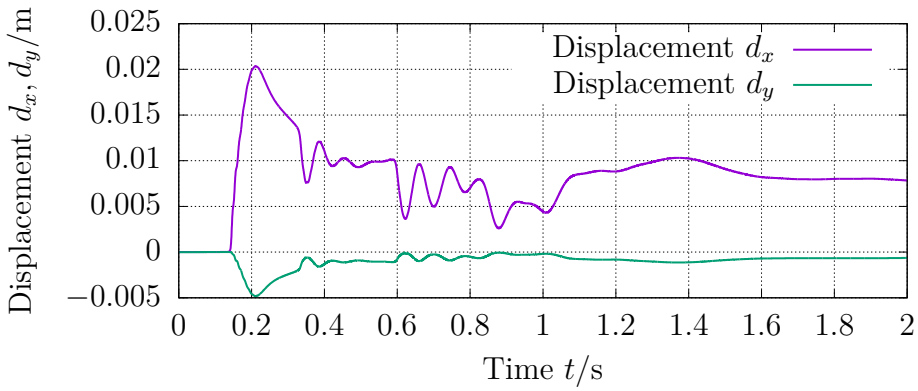


Figure 6.48: Components d_x, d_y of the obstacle's tip displacement over time t.

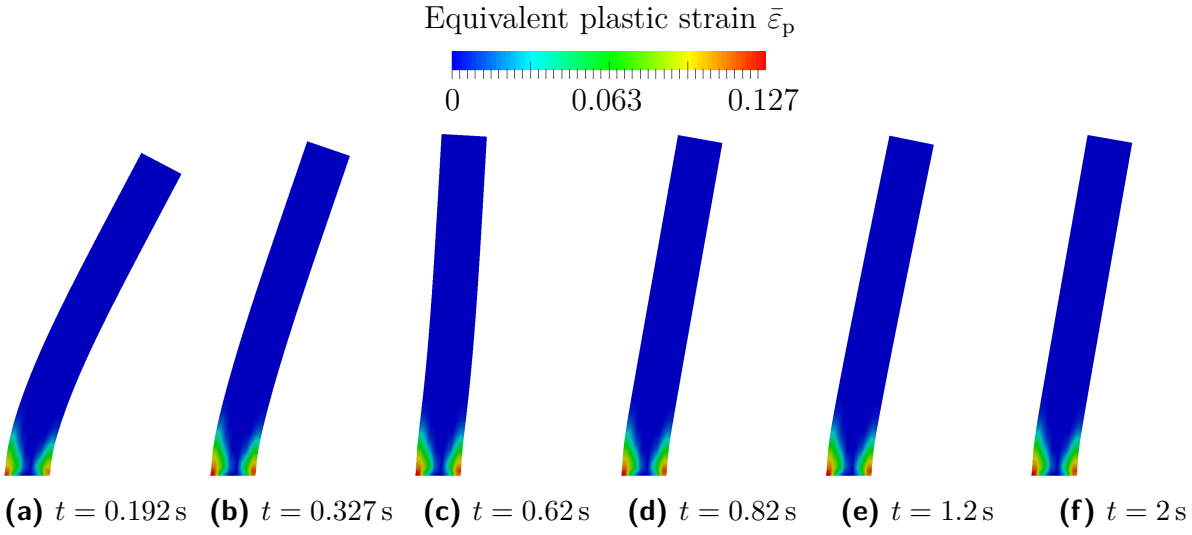


Figure 6.49: Deformed obstacle colored by equivalent plastic strain $\bar{\varepsilon}_\mathrm{p}$ at different instants of time t.

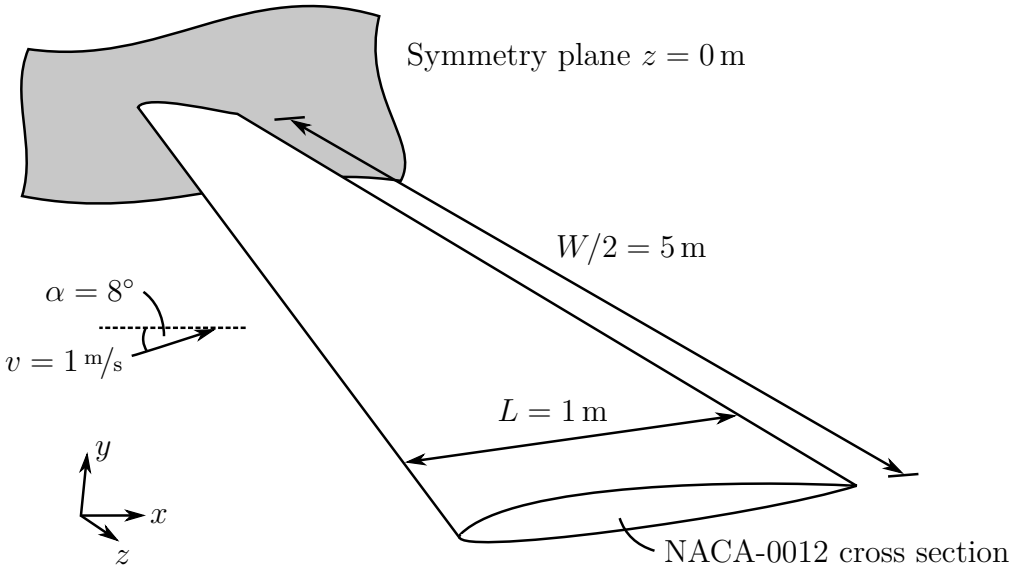


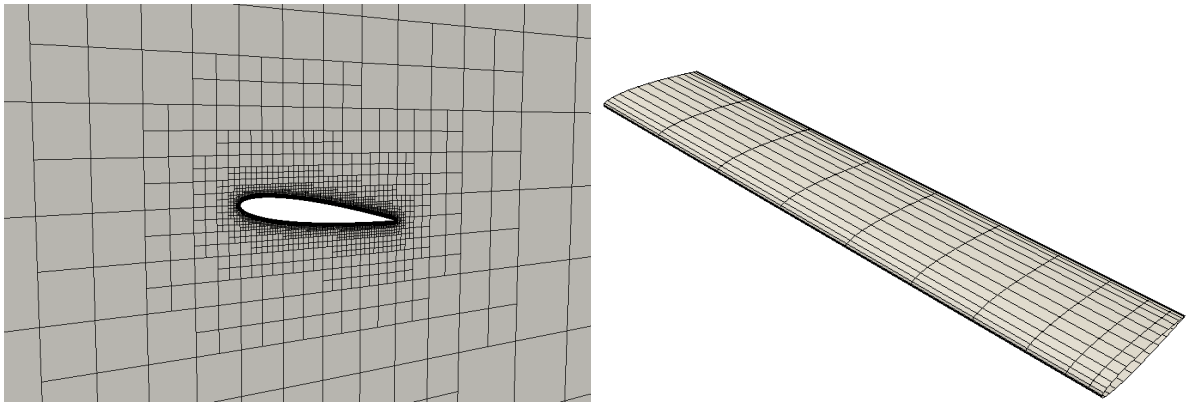
Figure 6.50: Hydrofoil in steady-state flow.

6.13 Hydrofoil in Steady-State Flow

From the above numerical examples, it became clear that the FVM is well suited for various kinds of FSI problems. However, the FVM can quickly become costly and prohibitively expensive for a fully implicit FSI analysis. Moreover, large mesh deformations can lead to solver difficulties because the FV mesh needs to follow the structural deformation at the FSI interface, which may lead to badly-shaped or self-penetrating cells. In contrast, the assumption of an incompressible, inviscid, and irrotational flow – and the use of the BEM as the method of choice to solve potential flow problems – usually results in much less computational effort. In addition, the boundary element mesh can easily track even large deformations of the fluid-structure interface without experiencing any notable mesh quality issues. These facts indicate that the BEM is in fact a very interesting alternative to the FVM, and it appears to be particularly attractive for maritime applications, where the assumption of a potential flow is often justified.

In order to explore the capabilities of the BEM in the context of FSI, we study the steady-state deflection of a NACA0012 cross section profile in this example. Exhibiting a chord length $L = 1\,\mathrm{m}$ and a width $W = 10\,\mathrm{m}$, the profile is fully clamped in the middle. It is subjected to a water flow of density $\rho_\mathrm{f} = 998\,\mathrm{kg/m^3}$ and kinematic viscosity $\nu_\mathrm{f} = 10^{-6}\,\mathrm{m^2/s}$ at constant velocity $v = 1\,\mathrm{m/s}$ under an angle of attack $\alpha = 8°$. Manufactured from steel, the profile exhibits a Young's modulus $E = 2.1 \times 10^{11}\,\mathrm{N/m^2}$ and Poisson's ratio $\nu_\mathrm{s} = 0.3$. As sketched in Figure 6.50, only half of the hydrofoil is modeled, and symmetry boundary conditions for the fluid problem are applied at $z = 0\,\mathrm{m}$ in order to save computational costs.

In the FSI analysis, the flow problem is solved using either the FVM and the pimpleDyMFoam solver distributed along with the open-source CFD package OpenFOAM [123] or the BEM available in the in-house first-order panel code *pan*MARE [12]. In the RANS calculation, we utilize a relatively fine mesh consisting of 1,311,723 cells, strongly graded towards the boundary layer around the hydrofoil. The well-established k-ω-SST model serves to predict the effects of turbulence. For the potential flow problem, only the surface



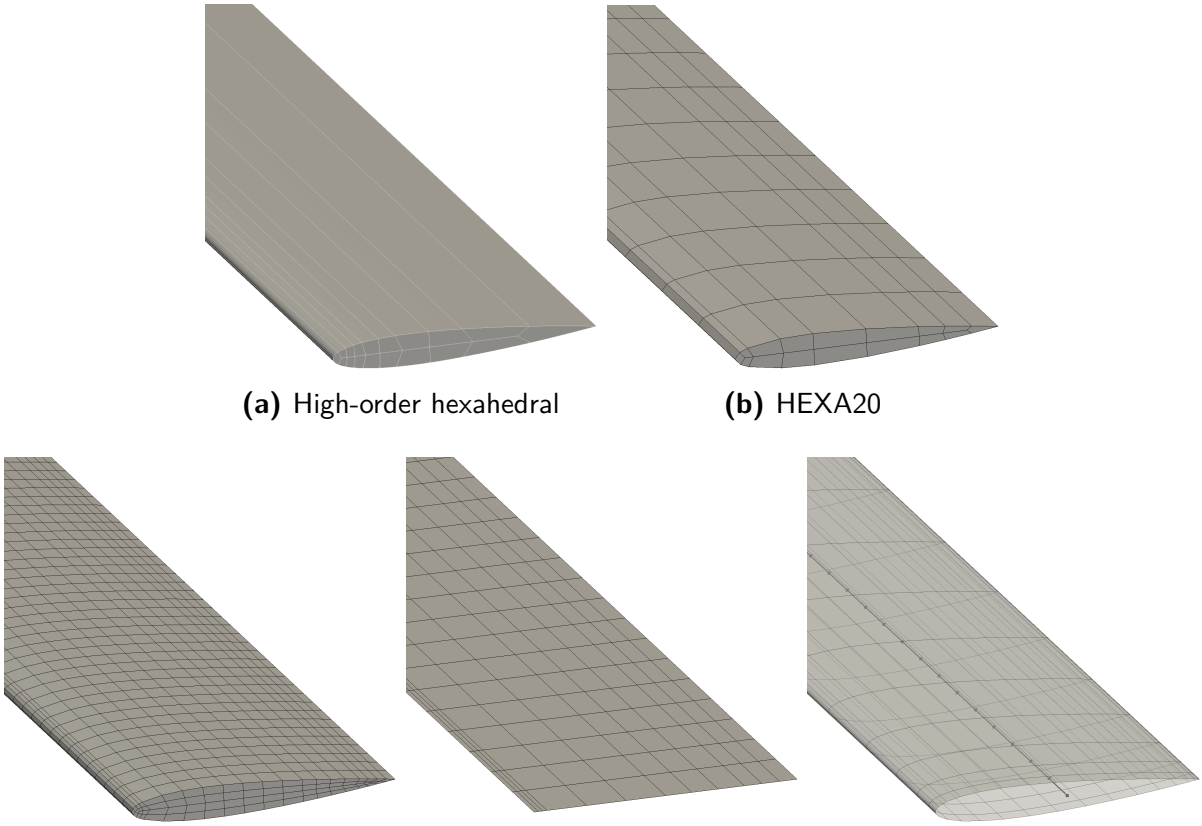
(a) FV mesh **(b)** Boundary element mesh

Figure 6.51: Different fluid discretizations around the hydrofoil.

of the hydrofoil needs to be discretized. Preliminary mesh studies revealed that 30 panels in chordwise and 10 panels in spanwise direction (summing up to a total of 1,600 panels) provide a reasonably accurate solution of the flow problem. An identical discretization is therefore also used in the FSI analysis, where comparably small deflections and, hence, only minor changes in the global flow behavior are to be expected. Figure 6.51 gives an impression of the FVM and BEM mesh for the solution of the flow problem.

For the numerical treatment of the structural subproblem, we choose the FEM as usual, but employ different discretizations comprising either 18 high-order hexahedral elements with polynomial order $p_x = p_y = 4$ and $p_z = 12$, 160 triquadratic twenty-noded hexahedral HEXA20 elements with reduced integration, 2,880 trilinear eight-noded hexahedral SOLID185 elements with enhanced strain formulation, 204 bilinear four-noded SHELL181 elements based on Reissner-Mindlin plate theory, or 20 linear two-noded BEAM188 elements based on Timoshenko beam theory. The high-order hexahedral elements are part of the in-house p-FEM code AdhoC [41, 40]. The open-source FEM software Code_Aster [43] provides the HEXA20 elements, while the SOLID185, SHELL181, and BEAM188 elements are available in the commercial FEM software suite ANSYS [3]. Figure 6.52 gives an overview of the different structural discretizations for the hydrofoil. Table 6.5 lists the computed tip deflections of the hydrofoil in steady state conditions. Apparently, the different considered numerical methods for the solution of the flow problem as well as the different structural discretizations are able to capture the problem reasonably well, as only very little deviation between the results can be observed. It should be noted, however, that a remarkably different amount of computational effort was invested to accomplish these solutions. Clearly, the FVM combined with a structural solid model poses the most expensive solution strategy, whereas the BEM and the structural beam model is considerably cheaper.

In order to demonstrate the capability of the BEM/FEM coupling procedure to also capture large deflections of the hydrofoil, the stiffness of the structure is significantly reduced by scaling the Young's modulus by a factor $s = 10^{-3}$ such that $E^* = 2.1 \times 10^8\,{}^{\mathrm{N}}/_{\mathrm{m}^2}$. It is obvious that the FVM experiences difficulties dealing with larger mesh deformations – as was to be expected in this case. In the BEM, however, only the surface of the body needs to be discretized such that the boundary element mesh can inherently follow the deformation of the body. Apart from the reduced computational expenditure to be



(a) High-order hexahedral **(b)** HEXA20

(c) SOLID185 **(d)** SHELL181 **(e)** BEAM188

Figure 6.52: Different structural discretizations for the hydrofoil.

Table 6.5: Steady-state tip deflection of the hydrofoil.

Fluid solver	Structural solver	Structural element type	Steady-state deflection $d_y/10^{-3}\,\mathrm{m}$
*pan*MARE	AdhoC	High-order hexahedral	1.6160
*pan*MARE	Code_Aster	HEXA20	1.6373
*pan*MARE	ANSYS	SOLID185	1.6111
*pan*MARE	ANSYS	SHELL181	1.6096
*pan*MARE	ANSYS	BEAM188	1.6182
OpenFOAM	ANSYS	SOLID185	1.6064

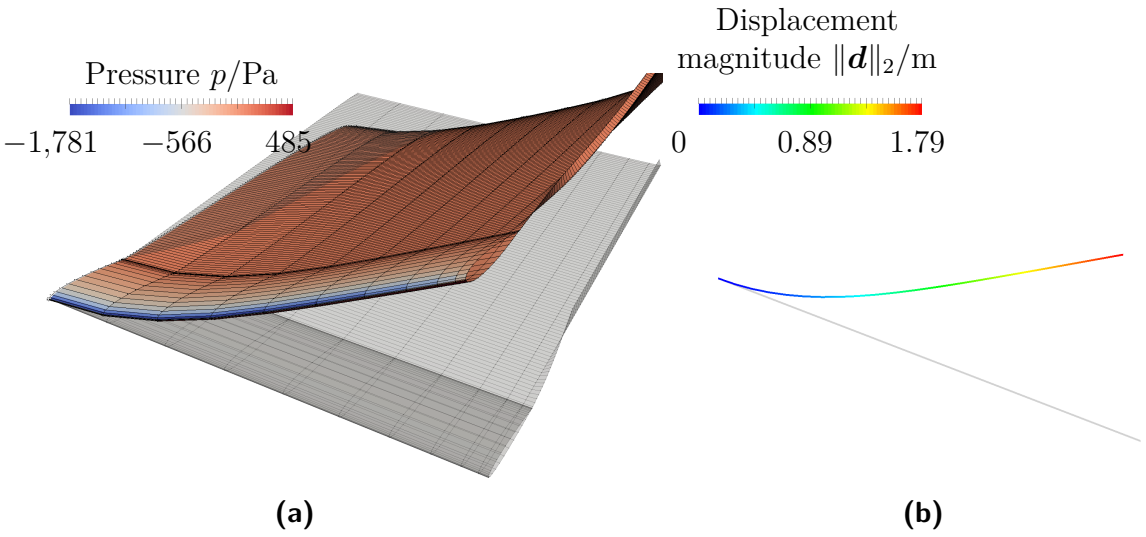


Figure 6.53: (a) Pressure p and (b) displacement magnitude $\|\boldsymbol{d}\|_2$ for the soft hydrofoil with a reduced Young's modulus $E^* = 2.1 \times 10^8\,\mathrm{N/m^2}$ in steady-state conditions.

invested in the solution of the flow field, this mesh deformation capability can be seen as one of the most notable advantages of the BEM in an FSI context. Except for the reduced Young's modulus E^*, all other parameters are kept as before. Figure 6.53 depcits the result of the simulation carried out using the BEM for the fluid problem and the beam model for the representation of the hydrofoil.

6.14 Ship Propeller

Due to the unsteady inflow caused by the wake of the ship, marine propellers operate under highly unsteady loading conditions. Clearly, the varying pressure distribution on the propeller has a significant impact on the resulting torque and thrust and, hence, on its performance. Furthermore, the pressure fluctuations may induce vibrations in the propulsion system, which propagate further into the hull structure [118, p. 191]. In the design stage, the hydrodynamically induced forces are usually approximated using simplified empirical formulas. However, these formulas do not provide a detailed insight into the transient nature of the FSI occurring between the elastic propeller and the surrounding flow field. For the derivation of a both reliable and efficient design, a realistic and accurate prediction of the propeller deflection and the transient flow field is essential. In this work, the partitioned solution approach is employed to deliver these results and to provide a profound understanding of the transient physical phenomena occurring in propeller operation.

In the following, we consider a fully submerged model-scale KCS P1356 propeller of diameter $D = 0.25\,\mathrm{m}$ equipped with five blades, as depicted in Figure 6.54, subjected to a uniform inflow. Manufactured from UNS C36000 free-cutting brass, the propeller exhibits a density $\rho_\mathrm{s} = 8.49 \times 10^3\,\mathrm{kg/m^3}$, Young's modulus $E = 9.7 \times 10^{12}\,\mathrm{N/m^2}$, and Poisson's ratio $\nu_\mathrm{s} = 0.31$. It rotates at constant speed $n = 9.5\,\mathrm{1/s}$ about the x-axis. In the present study, the deflection of the hub is neglected, and only the blades are considered to be deformable. In addition, the influence of gravity is assumed to have minor influence on the results and is hence omitted here. Since the propeller operates under periodic conditions, it suffices to discretize a single blade only and to apply periodic boundary conditions in the flow

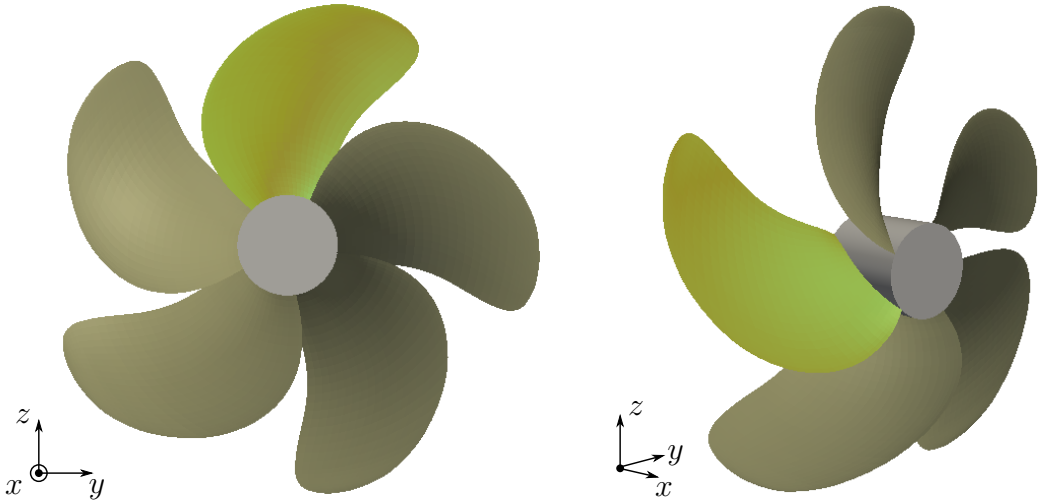


Figure 6.54: (a) Front and (b) side view of the model-scale KCS P1356 propeller.

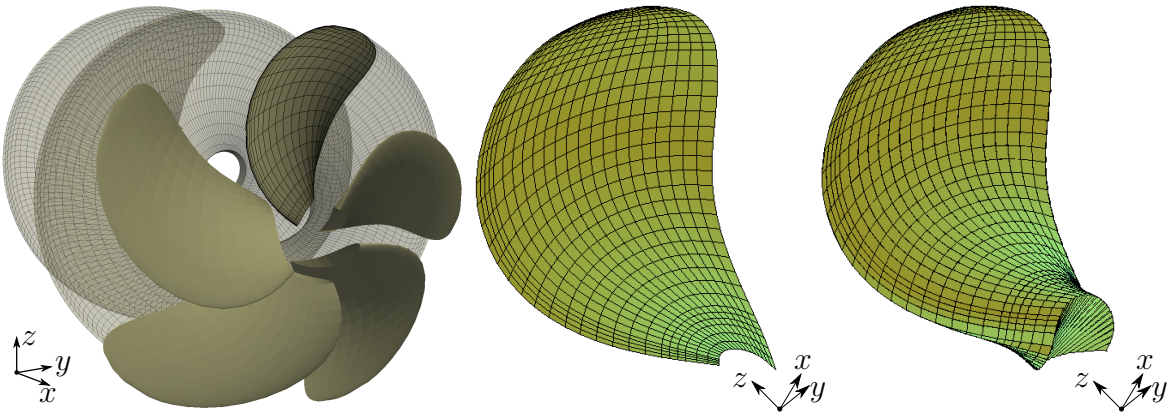
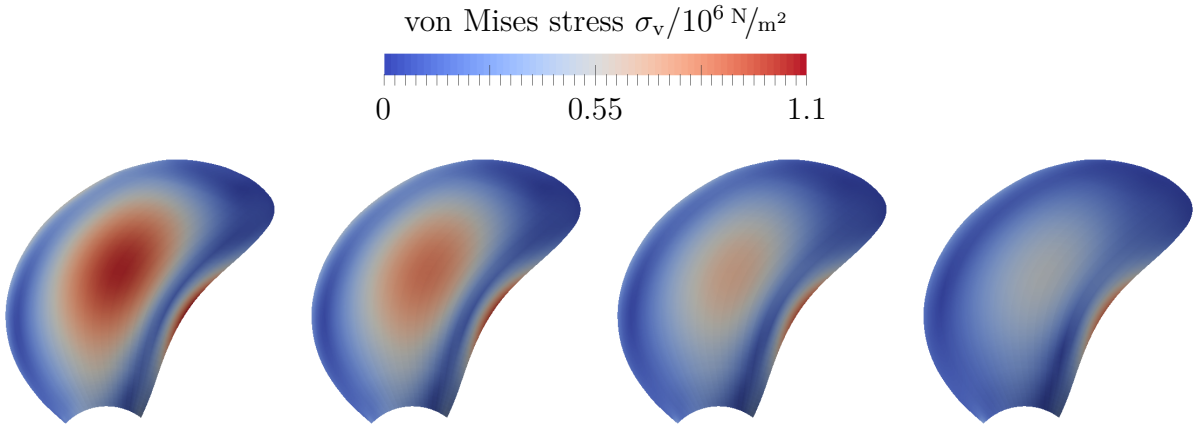


Figure 6.55: (a) Fluid and (b) structural discretization of the ship propeller blade.

field. For the numerical treatment of the fluid problem, again the BEM implemented in the software package *pan*MARE [12] is chosen. Consequently, only the surface of the propeller and the wake need to be discretized, which offers the substantial advantages of being computationally cheap as compared to finite volume methods. Further, this allows to handle arbitrarily large mesh motion without difficulty. As depicted in Figure 6.55a, we use 20 panels in chordwise direction and 18 panels in spanwise direction for the propeller. In the wake region, we resolve two rotations. In total, the boundary element mesh consists of 740 body and 6,480 wake panels. For the structural part, we use the commercial FE software ANSYS [3]. In order to save computational cost, we discretize the blade mid-plane by means of four-noded SHELL181 elements based on the Reissner-Mindlin theory, with three translational and three rotational degrees of freedom per node. As illustrated in Figure 6.55b, the thickness normal to the mid-plane is prescribed at the nodes and interpolated linearly to evaluate the thickness at the element integration points. In total, the shell discretization comprises 770 elements and 4,899 degrees of freedom. In the structural model, we prescribe the translational and rotational motion of the pro-



(a) $J = 0.5$ (b) $J = 0.6$ (c) $J = 0.7$ (d) $J = 0.8$

Figure 6.56: Von Mises stress σ_v for different advance ratios J.

peller at the connection between the blade root and the hub. From the angular velocity $\omega = 2\pi n$, the current rotation angle of a node initially located at (x_0, y_0, z_0) is calculated as $\varphi_t = \varphi_0 + \omega t = \arctan(z_0/y_0) + \omega t$ and the current position at time t becomes (x_t, y_t, z_t), where $x_t = x_0 + vt$, $y_t = r\cos\varphi_t$, and $z_t = r\sin\varphi_t$ with the distance $r = \sqrt{y_0^2 + z_0^2}$ from the rotation axis. Different advance ratios $J \in \{0.5, 0.6, 0.7, 0.8\}$ are considered, resulting in different translational velocities $v = JnD$. Due to the large translations and rotations, a geometrically nonlinear analysis is to be performed. For the description of the material behavior, it suffices to choose the St. Venant-Kirchhoff model, as small strains are to be expected for the comparably stiff material.

Regarding the time integration procedures, the backward Euler scheme is used for the fluid part, whereas the Newmark scheme with parameters $\beta = 0.2525$ and $\gamma = 0.5050$ is employed for the structural problem.

In order to judge whether the fluid and the structural field are equilibrated to sufficient accuracy within a time increment, it is checked whether $\|\mathbf{r}_{j+1}^k\|_2 < \varepsilon_{\text{abs}} = 10^{-4}$ or $\|\mathbf{r}_{j+1}^k\|_2 / \|\mathbf{r}_{j+1}^0\|_2 < \varepsilon_{\text{rel}} = 5 \times 10^{-3}$. A total of three propeller revolutions are computed for all sets of simulations using a constant angle increment $\Delta\varphi = 2°$, which corresponds to a time step size $\Delta t \approx 5.85 \times 10^{-4}\,\mathrm{s}$.

Figure 6.56 illustrates the von Mises stress σ_v for different advance ratios J. As to be expected, the von Mises stress decreases for increasing advance ratios J. Moreover, the highest stress occurs in the center of the blade and in the middle of the trailing edge. Figure 6.57 depicts the openwater diagram with the thrust coefficient $k_t = {}^{t}/_{\rho n^2 D^4}$, the torque coefficient $k_q = {}^{q}/_{\rho n^2 D^5}$, and the efficiency $\eta_0 = {}^{Jk_t}/_{2\pi k_q}$ for the studied propeller, where t and q denote thrust and torque, respectively.

In the next step, the hydrodynamic coefficients of the propeller will be determined. Rotating at constant angular speed and subjected to a constant uniform inflow as before, the propeller is subjected to an additional superimposed harmonic swaying motion $w(t) := w_0 \sin(\omega_0 t)$ such that $v(t) = v_0 + w(t)$, so as to determine the hydrodynamic mass and damping coefficients. Herein, $w_0 = 0.3011\,\mathrm{m/s}$ denotes the swaying amplitude, and $\omega_0 = \omega/5$ is the angular frequency of the oscillation. In order to capture at least two full periods of the oscillatory motion, a simulation is carried out for 15 propeller revolutions. The time step size remains unchanged at $\Delta t = 5.85 \times 10^{-4}\,\mathrm{s}$ corresponding to an angle increment

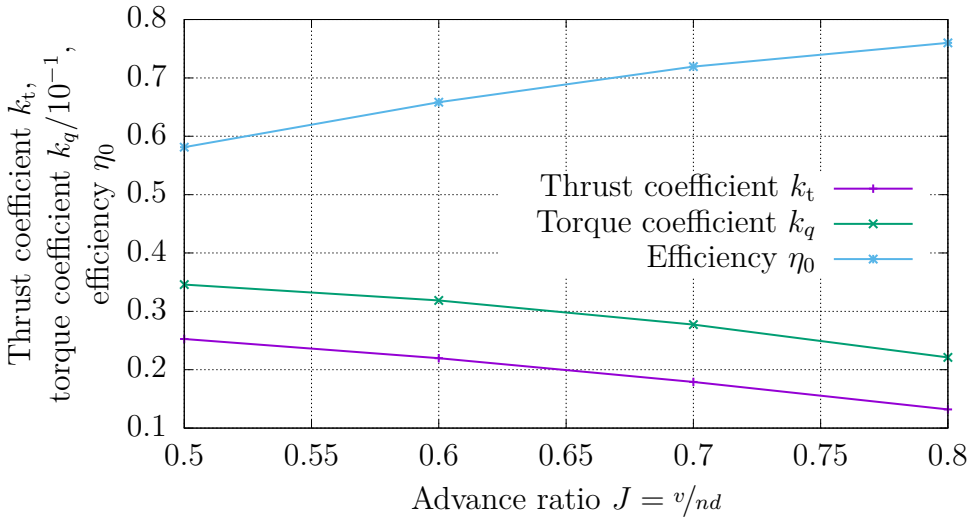


Figure 6.57: Openwater diagram for the KCS P1356 ship propeller obtained from an FSI analysis.

$\Delta\varphi = 2°$.

In order to determine the hydrodynamic coefficients, let us first consider a function $f(w, \dot{w})$ depending on the superimposed swaying velocity w and acceleration $\dot{w}$. Carrying out a Taylor series expansion leads us to

$$f(w,\dot{w}) \approx f_0 + \Delta w f_{\mathrm{w}} + \Delta\dot{w} f_{\dot{w}} + \frac{1}{2}\left(\Delta w^2 f_{2w} + 2\Delta w \Delta\dot{w} f_{w\dot{w}} + \Delta\dot{w}^2 f_{2\dot{w}}\right) \tag{6.8}$$

with the increments

$$\Delta w = w - w_0 = w_0(\sin(\omega_0 t) - 1)\ , \quad \Delta\dot{w} = \dot{w} - \dot{w}_0 = w_0\omega_0\cos(\omega_0 t) \tag{6.9}$$

as an expression for the loads induced by the additional swaying motion. Dropping higher order derivatives, the Taylor series becomes

$$\begin{aligned} f(w,\dot{w}) &\approx f_0 + \Delta w f_{\mathrm{w}} + \Delta\dot{w} f_{\dot{w}} + \frac{1}{2}\left(\Delta w^2 f_{2\mathrm{w}} + 2\Delta w\Delta\dot{w} f_{\mathrm{w}\dot{w}}\right) \\ &= \left(f_0 - w_0 f_{\mathrm{w}} + \frac{1}{2}w_0^2 f_{2\mathrm{w}}\right) + \sin(\omega_0 t)\left(w_0 f_{\mathrm{w}} - w_0^2 f_{2\mathrm{w}}\right) + \sin^2(\omega_0 t)\frac{1}{2}w_0^2 f_{2\mathrm{w}} \\ &\quad + \cos(\omega_0 t)\left(w_0\omega_0 f_{\dot{\mathrm{w}}} - w_0^2\omega_0 f_{\mathrm{w}\dot{w}}\right) + \cos(\omega_0 t)\sin(\omega_0 t) w_0^2\omega_0 f_{\mathrm{w}\dot{w}} \\ &= \left(f_0 - w_0 f_{\mathrm{w}} + \frac{3}{4}w_0^2 f_{2\mathrm{w}}\right) + \cos(\omega_0 t)\left(w_0\omega_0 f_{\dot{\mathrm{w}}} - w_0^2\omega_0 f_{\mathrm{w}\dot{w}}\right) \\ &\quad + \sin(\omega_0 t)\left(w_0 f_{\mathrm{w}} - w_0^2 f_{2\mathrm{w}}\right) + \cos(2\omega_0 t)\left(-\frac{1}{4}w_0^2 f_{2\mathrm{w}}\right) \\ &\quad + \sin(2\omega_0 t)\left(\frac{1}{2}w_0^2\omega_0 f_{\mathrm{w}\dot{w}}\right)\ . \end{aligned} \tag{6.10}$$

In the above approximation, the coefficients f_0, f_{w}, $f_{\dot{\mathrm{w}}}$, $f_{2\mathrm{w}}$, and $f_{\mathrm{w}\dot{w}}$ represent the hydrodynamic coefficients. Having obtained the force $F(t)$ in x-direction and the moment $M(t)$ about the x-axis at the propeller hub from the simulation, the second-order Fourier

Table 6.6: Hydrodynamic (a) force and (b) moment coefficients for different advance ratios J.

(a) Force

Advance ratio J	Hydrodynamic coefficients			
	$f_{\mathrm{w}}/10^{-1}$	$f_{\dot{\mathrm{w}}}/10^{-1}$	$f_{2\mathrm{w}}/10^{-1}$	$f_{\mathrm{w}\dot{\mathrm{w}}}/10^{-1}$
0.5	−4.184	−1.104	−5.010	2.718
0.6	−4.848	−0.932	−4.110	1.911
0.7	−5.387	−0.778	−3.506	1.412
0.8	−5.846	−0.646	−3.133	1.096

(b) Moment

Advance ratio J	Hydrodynamic coefficients			
	$f_{\mathrm{w}}/10^{-2}$	$f_{\dot{\mathrm{w}}}/10^{-2}$	$f_{2\mathrm{w}}/10^{-1}$	$f_{\mathrm{w}\dot{\mathrm{w}}}/10^{-2}$
0.5	4.090	1.426	1.047	−2.101
0.6	5.620	1.240	1.108	−1.428
0.7	7.202	1.044	1.225	−1.133
0.8	8.834	1.081	1.356	−0.486

expansion

$$f(t) \approx a_0 + \sum_{k=1}^{2} \left(a_k \cos(\omega_k t) + b_n \sin(\omega_k t)\right) \tag{6.11}$$

is fitted to $F(t)$ and $M(t)$ to determine the coefficients a_0, a_1, b_1, a_2, and b_2. By equating the coefficients of the Fourier expansion (6.11) and the Taylor series expansion (6.10), the hydrodynamic coefficients are then computed from

$$f_0 = a_0 + b_1 - a_2 \,, \quad f_{\mathrm{w}} = \frac{b_1 - 4a_2}{w_0} \,, \quad f_{\dot{\mathrm{w}}} = \frac{a_1 + 2b_2}{w_0\omega_0} \,, \quad f_{2\mathrm{w}} = \frac{-4a_2}{w_0^2} \,, \quad f_{\mathrm{w}\dot{\mathrm{w}}} = \frac{2b_2}{w_0^2\omega_0} \,. \tag{6.12}$$

It is convenient to transform the dimensioned hydrodynamic coefficient into dimensionless quantities. Regarding the force coefficients, the equations

$$f_{\mathrm{w}}^{'} = \frac{1}{\rho n D^3} f_{\mathrm{w}} \,, \qquad f_{\dot{\mathrm{w}}}^{'} = \frac{1}{\rho D^3} f_{\dot{\mathrm{w}}} \,, \qquad f_{2\mathrm{w}}^{'} = \frac{1}{\rho D^2} f_{2\mathrm{w}} \,, \qquad f_{\mathrm{w}\dot{\mathrm{w}}}^{'} = \frac{n}{\rho D^2} f_{\mathrm{w}\dot{\mathrm{w}}} \tag{6.13}$$

are employed to nondimensionalize the hydrodynamic coefficients, whereas the relations

$$f_{\mathrm{w}}^{'} = \frac{1}{\rho n D^4} f_{\mathrm{w}} \,, \qquad f_{\dot{\mathrm{w}}}^{'} = \frac{1}{\rho D^4} f_{\dot{\mathrm{w}}} \,, \qquad f_{2\mathrm{w}}^{'} = \frac{1}{\rho D^3} f_{2\mathrm{w}} \,, \qquad f_{\mathrm{w}\dot{\mathrm{w}}}^{'} = \frac{n}{\rho D^3} f_{\mathrm{w}\dot{\mathrm{w}}} \tag{6.14}$$

are applied for the calculation of the moment coefficients. Table 6.6 lists the dimensionless hydrodynamic coefficients for the considered KCS P1356 propeller obtained by the aforementioned procedure for different advance ratios J.

6.15 Wind Turbine Rotor

In the previous example, it was demonstrated that structural elements may offer substantial advantages over continuum elements if slender or thin-walled structures are considered. In

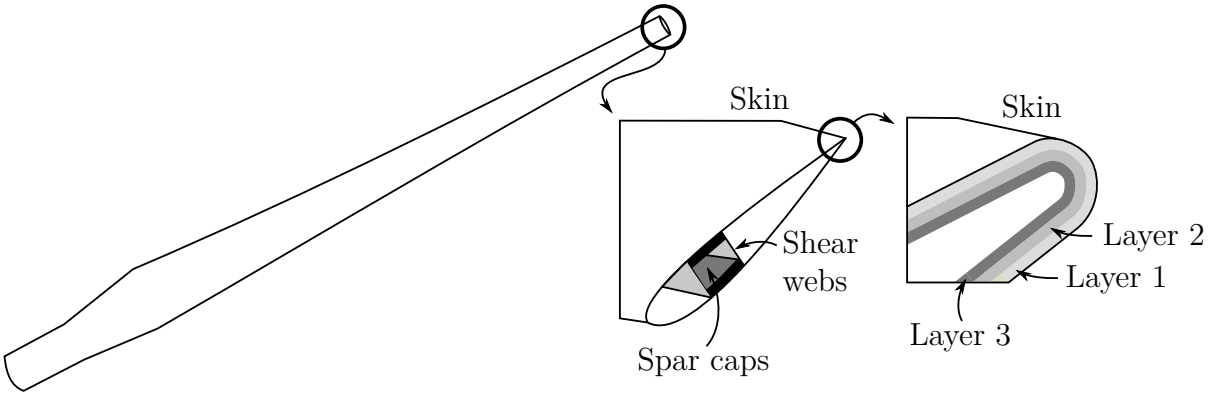


Figure 6.58: Typical wind turbine blade, reproduced from [4, p. 352].

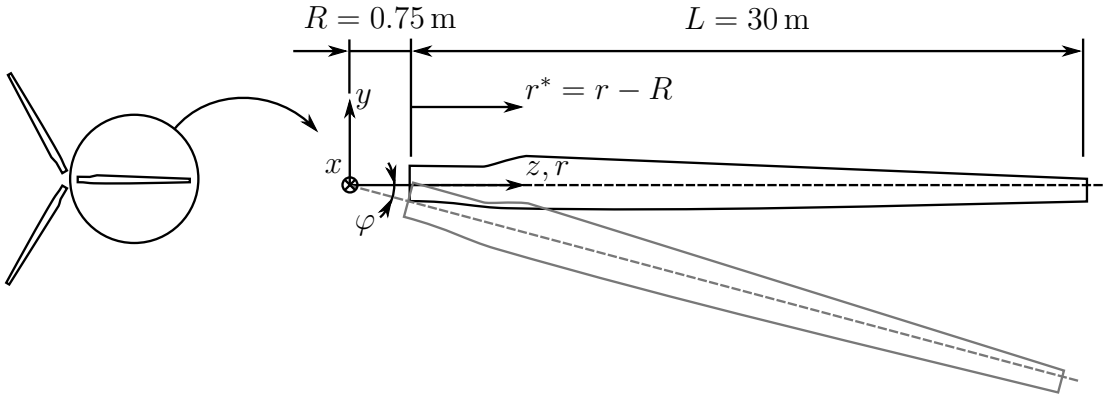


Figure 6.59: Front view of a blade of the three-bladed wind turbine rotor. The airflow of density moves in positive x-direction and the rotor turns in clockwise direction as seen from the direction of the wind.

these cases, structural elements usually require much less degrees of freedom to reach the same accuracy as continuum elements, which is why they have great potential to reduce the computational costs associated to the solution of the structural subproblem.

In the following, we will consider another example in which it is favorable to use structural elements. This time, however, beam elements instead of shell elements will be used.

In order to combine high structural strength with a light weight, wind turbine blades are usually manufactured from highly advanced composite materials arranged in several layers, as pictured in Figure 6.58. From the inside, the blade is equipped with shear webs and spar caps to stiffen the structure. During operation under the action of aerodynamic loads, the structure is usually allowed to deflect by a considerable amount. Hence, a detailed investigation of the FSI is of significant importance to obtain an accurate estimate of the internal stresses as well as to predict the effect of the blade deflection on the surrounding airflow and, thus, on the efficiency of the entire wind turbine.

Our numerical study is based on a structural setup outlined in the ANSYS Technology Demonstration Guide [4, p. 351–362]. As illustrated in Figure 6.59, the three-bladed rotor turns at a constant frequency $f = 13.6\,{}^1\!/_{\mathrm{min}} \approx 0.2267\,{}^1\!/_{\mathrm{s}}$ about the x-axis. For the hub radius, we assume $R = 0.75\,\mathrm{m}$, and each blade has a length $L = 30\,\mathrm{m}$. Figure 6.60 shows the cross section of a blade at different radii $r^* = r - R$, and Table 6.7 summarizes the material parameters for the three skin layers, the spar caps, and the shear webs. For

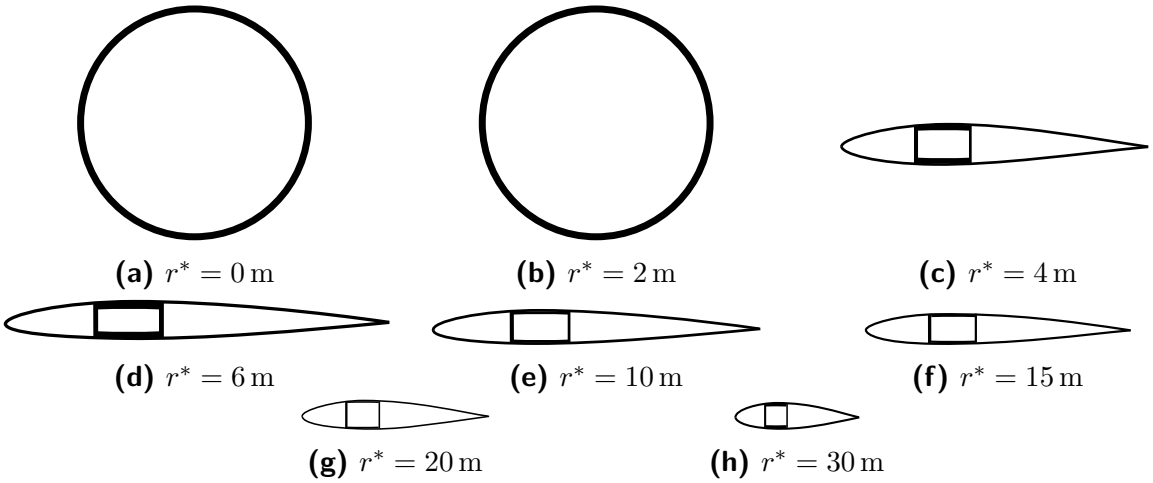


Figure 6.60: Blade cross section at different radii $r^* = r - R$.

Table 6.7: Parameters for the different materials of a wind turbine blade.

Material constants	Skin layer 1	Skin layer 2	Skin layer 3	Spar caps	Shear webs
Density ρ / $^{\mathrm{kg}}/_{\mathrm{m}^3}$	1.90×10^3	1.75×10^3	2.00×10^3	1.80×10^3	1.85×10^3
Young's modulus E_x / $^{\mathrm{N}}/_{\mathrm{m}^2}$	1.39×10^{10}	1.20×10^{10}	1.48×10^{10}	1.66×10^{10}	1.59×10^{10}
Young's modulus E_y / $^{\mathrm{N}}/_{\mathrm{m}^2}$	1.39×10^{10}	1.20×10^{10}	1.48×10^{10}	1.66×10^{10}	1.59×10^{10}
Young's modulus E_z / $^{\mathrm{N}}/_{\mathrm{m}^2}$	1.05×10^{10}	9.50×10^9	1.10×10^{10}	1.21×10^{10}	1.15×10^{10}
Poisson's ratio ν_{xy}	0.43	0.38	0.48	0.40	0.52
Poisson's ratio ν_{yz}	0.15	0.18	0.13	0.15	0.10
Poisson's ratio ν_{xz}	0.15	0.18	0.13	0.15	0.10
Shear modulus G_{xy} / $^{\mathrm{N}}/_{\mathrm{m}^2}$	1.09×10^{10}	9.30×10^9	1.10×10^{10}	1.24×10^{10}	1.19×10^{10}
Shear modulus G_{yz} / $^{\mathrm{N}}/_{\mathrm{m}^2}$	5.48×10^9	6.03×10^9	5.03×10^9	4.08×10^9	4.48×10^9
Shear modulus G_{xz} / $^{\mathrm{N}}/_{\mathrm{m}^2}$	5.48×10^9	6.03×10^9	5.03×10^9	4.08×10^9	4.48×10^9

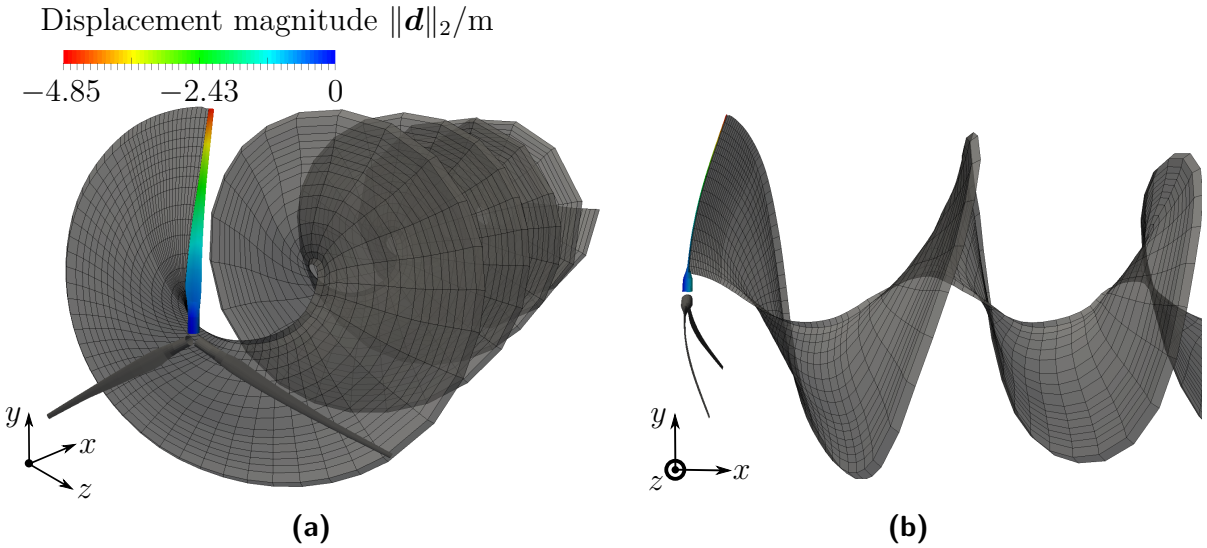


Figure 6.61: (a) Oblique and (b) side view of the wind turbine rotor in steady operation.

the airflow, we assume a density $\rho = 1.2\,\mathrm{kg/m^3}$, kinematic viscosity $\nu = 1.48 \times 10^5\,\mathrm{m^2/s}$, and constant speed $v = 11\,\mathrm{m/s}$ in positive x-direction.

The fluid field is discretized by 400 first-order body and 2,125 wake panels, adding up to 2,525 panels in total, and is solved using the in-house BEM code *pan*MARE [12]. It suffices to discretize only a single blade of the rotor and to apply periodic boundary conditions to account for the other two blades of the entire rotor. For the discretization of the structure, we use a beam model incorporating the varying blade cross section from Figure 6.60 and consisting of 9 linear two-noded BEAM188 elements, based on Timoshenko beam theory, with three translational and three rotational degrees of freedom per node. In order to take the rotation of the blade about the x-axis into account, the displacement and rotation at the blade root are prescribed according to

$$\begin{aligned} d_y &= -R\sin(\omega(t+\Delta t)) \\ d_z &= -R(1-\cos(\omega(t+\Delta t))) \\ \varphi &= \omega t\ . \end{aligned} \tag{6.15}$$

Regarding the temporal discretization, the Euler backward scheme is employed for the fluid problem, whereas the Newmark scheme with $\beta = 0.2525$ and $\gamma = 0.5050$ is chosen for the structure. A time step size $\Delta t \approx 2.45 \times 10^{-2}\,\mathrm{s}$ corresponding to an angle increment $\Delta\varphi = 2°$ is applied for both subproblems. It is kept constant throughout the simulation, which covers three full rotations of the rotor.

Figure 6.61 gives an impression of the deflected wind turbine rotor in steady operation. Figure 6.62 illustrates the displacement d_y at the blade tip at a radius $r^* = L$.

Our numerical investigations reveal that the blade deflection is quite significant. Consequently, it leads to a notable change in the surrounding airflow, which perfectly underpins the necessity to conduct a strongly-coupled FSI analysis for this application.

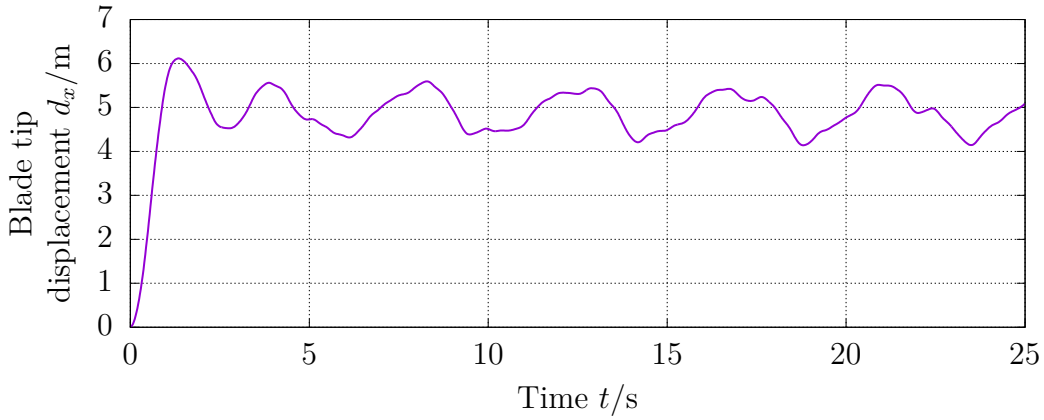


Figure 6.62: Displacement d_y at the blade tip at $r^* = L$.

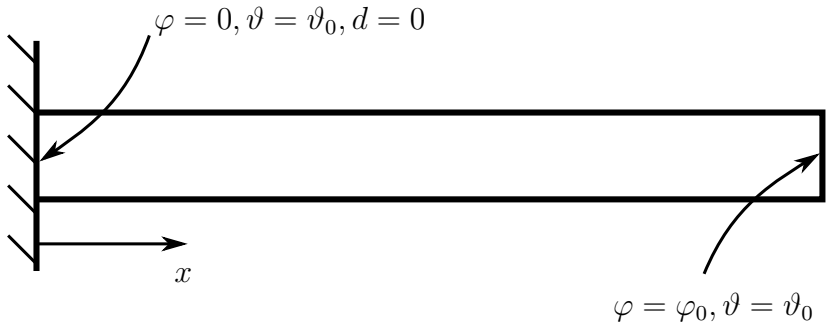


Figure 6.63: Geometry and boundary conditions for the electro-thermo-mechanically coupled rod.

6.16 Electro-Thermo-Mechanically Coupled Rod

So far, only the interaction of a structural and a fluid field has been considered. In order to demonstrate that also other multifield problems can be conveniently solved using a partitioned strategy, an electro-thermo-mechanically coupled problem is considered in this example, involving the interaction of three different fields.

Adopting the problem setting from [45, p. 1088–1090, 96, p. 1784–1786], a rod of length $L = 0.2\,\mathrm{m}$ and cross section $A = 2.5 \times 10^{-5}\,\mathrm{m}^2$ as depicted in Figure 6.63 is considered. For the material parameters of the rod, we choose the values listed in Table 6.8. Geometrically

Table 6.8: Material parameters for the electro-thermo-mechanically coupled rod.

Material parameter	Value
Electrical conductivity $\lambda_\varphi(\vartheta = \vartheta_0)/\mathrm{A/Vm}$	1.2×10^4
Specific heat capacity $c_p/\mathrm{J/kgK}$	460
Thermal conductivity $\lambda_\vartheta/\mathrm{N/sK}$	45
Thermal expansion coefficient $\alpha_\vartheta/1/\mathrm{K}$	1.2×10^{-5}
Young's modulus $E/\mathrm{N/m^2}$	2.07×10^{11}
Poisson's ratio ν	0.3

linear deflection and a linear-elastic material behavior are assumed. Hence, the governing set of partial differential equations reads

$$(\lambda_\varphi \varphi'(x,t))' = 0 \,, \tag{6.16}$$

$$\rho c_p \dot{\vartheta}(x,t) = (\lambda_\vartheta \vartheta'(x,t))' + R_\varphi \,, \tag{6.17}$$

$$EA\,(d'(x,t) - \alpha_\vartheta \vartheta(x,t))' = 0 \,. \tag{6.18}$$

In this particular case, we neglect the thermoelastic coupling effect, meaning that the heat or cooling effects due to the elastic strain rate are considered to be significantly smaller than the Joule heating term R_φ. The boundary conditions are set to

$$\begin{aligned} \varphi = 0,\ \vartheta = \vartheta_0,\ d = 0 \quad &\text{at } x = 0 \,, \\ \varphi = \varphi_0,\ \vartheta = \vartheta_0 \quad &\text{at } x = L \,. \end{aligned} \tag{6.19}$$

For the reference electric potential and the reference temperature, we choose $\varphi_0 = 5\,\mathrm{V}$ and $\vartheta_0 = 293.15\,\mathrm{K}$, respectively. Following [95], the boundary value problem (6.16)-(6.19) can be solved analytically. For the electric potential, we obtain the time-independent linear function

$$\varphi(x,t) = \varphi(x) = \varphi_0 x / L \,. \tag{6.20}$$

Hence, the Joule heating term takes the constant value

$$R_\varphi = \lambda_\varphi \left(\frac{\partial \varphi}{\partial x} \right)^2 = \lambda_\varphi \varphi_0^2 / L^2 \,. \tag{6.21}$$

Due to the fact that the source term is constant, we find that the infinite sum

$$\vartheta(x,t) = \vartheta_0 + \frac{4 R_\varphi}{\pi \rho c_p} \sum_{n=1}^{\infty} \frac{\sin \frac{n\pi}{L} x}{\beta^2 n} \left(1 - \exp\left(-\beta t^2 \right) \right) \,, \quad n = 1, 3, 5, 7, \dots \,, \tag{6.22}$$

solves Equation (6.17). Herein, $\beta = n\pi\gamma/L$ and $\gamma = \sqrt{\lambda_\vartheta / (\rho c_p)}$. Inserting the temperature distribution (6.22) into (6.18) and integrating along the x-coordinate leads us to

$$d(x,t) = \frac{4 \alpha_\vartheta L R_\varphi}{\pi^2 \rho c_p} \sum_{n=1}^{\infty} \frac{1 - \cos \frac{n\pi}{L} x}{\beta^2 n^2} \left(1 - \exp\left(-\beta^2 t \right) \right) \tag{6.23}$$

as the solution for the mechanical field.

Having derived the analytical reference solution, let us apply a partitioned solution approach to the problem, treat the electric, thermal, and mechanical field separately – and, within a time increment, exchange the relevant field quantities between the fields. All fields are discretized by 10 linear finite elements and solved using the commercial FEM software ANSYS. We use bilinear uniaxial LINK68 elements to discretize the electric field, while we employ bilinear uniaxial LINK33 elements for the thermal field and bilinear uniaxial LINK180 elements for the structural problem. In each time increment, we first solve the electric field and transfer the resulting internal heat generation R_φ to the thermal field, where it is applied as a source term. Subsequently, the thermal field is solved to obtain the temperature ϑ, which is then passed over to the mechanical field. Here, the increasing temperature leads to a thermal expansion of the structure. A linear interpolation scheme

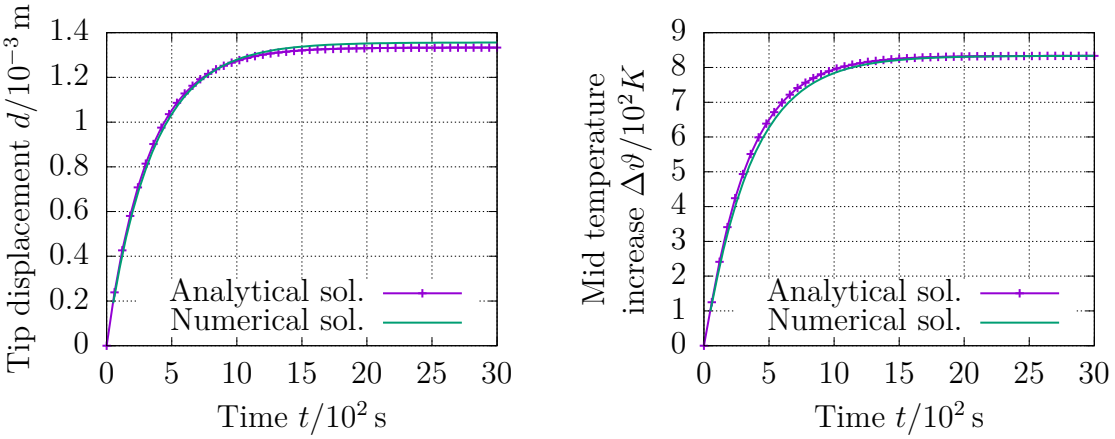


Figure 6.64: Displacement d at the end and temperature increase $\Delta\vartheta$ in the middle for the electro-thermo-mechanically coupled rod.

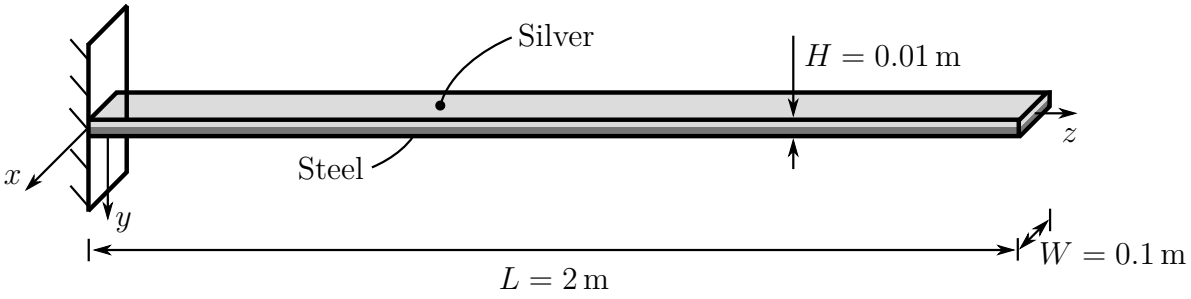


Figure 6.65: Geometry of the bimetallic beam.

is applied to ensure an accurate data transfer from one field to another. For the time step size, we choose $\Delta t = 50\,\mathrm{s}$. The trapezoidal rule is used for the time integration of the electric and thermal field, while the undamped Newmark scheme with the parameters $\beta = 0.25$ and $\gamma = 0.5$ is applied for the structural problem. Since the present example represents a weakly-coupled problem, it suffices to follow an explicit coupling procedure and to exchange the field quantities only once per time increment. Consequently, the implicit iteration within a time increment is omitted and a convergence acceleration scheme does not need to be applied. To allow for a comparison of the numerical results to the analytical solution, we evaluate the displacement d at the end and the temperature increase $\Delta\vartheta$ in the middle of the rod, as depicted in Figure 6.64. Evidently, the numerical solutions are in excellent agreement with the analytical reference results.

6.17 Bimetallic Beam

Due to the fact that only small deflections and weakly-coupled fields have been considered in the previous example, the application of an implicit solution procedure was not required. Intending to increase the difficulty of the problem, let us consider a bimetallic beam consisting of two different layer materials – steel and silver – as depicted in Figure 6.65. The problem is adopted from [174, pp. 343–345, 45, pp. 1090–1094, 44, pp. 145–151], where the solution was likewise accomplished using a partitioned solution strategy. The beam exhibits a length $L = 2\,\mathrm{m}$, a width $W = 0.1\,\mathrm{m}$, and a height $H = 0.01\,\mathrm{m}$. Its material

Table 6.9: Material parameters for steel and silver.

Material parameter	Steel	Silver
Density $\rho/^{\mathrm{kg}}/_{\mathrm{m}^3}$	7.8×10^3	1.05×10^4
Bulk modulus $K/^{\mathrm{N}}/_{\mathrm{m}^2}$	1.642×10^{11}	1.061×10^{11}
Shear modulus $G/^{\mathrm{N}}/_{\mathrm{m}^2}$	8.02×10^{10}	3.03×10^{10}
Heat capacity $c_p/^{\mathrm{J}}/_{\mathrm{kgK}}$	4.6×10^2	2.3×10^2
Thermal conductivity $\lambda_\vartheta/^{\mathrm{N}}/_{\mathrm{sK}}$	0.45×10^2	4.30×10^2
Thermal expansion coefficient $\alpha_\vartheta/^1/_{\mathrm{K}}$	1.55×10^{-5}	1.95×10^{-5}
Emissivity ε	0.8	0.1
Electrical conductivity $\lambda_\varphi(\vartheta = \vartheta_0)/^{\mathrm{A}}/_{\mathrm{Vm}}$	1.2×10^7	6.2×10^7
Linear temperature coefficient $\alpha_\varphi/^1/_{\mathrm{K}}$	5.6×10^{-3}	3.8×10^{-3}

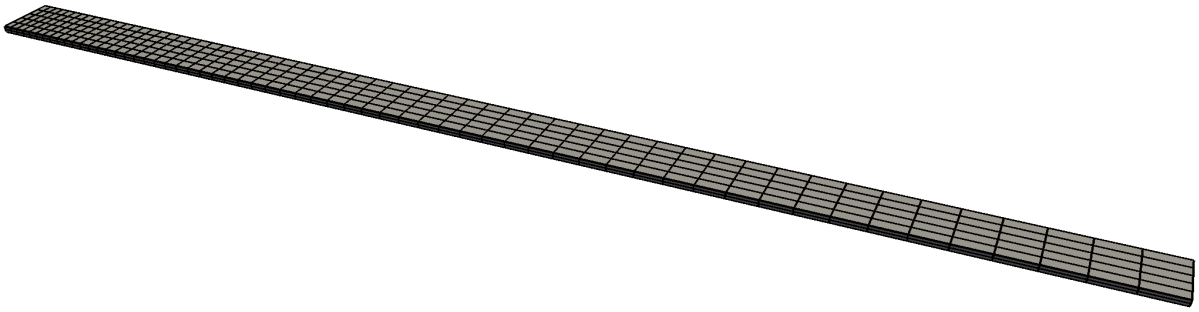


Figure 6.66: Computational mesh used for the discretization of the electric, thermal, and structural field of the bimetallic beam.

parameters are listed in Table 6.9. In order to introduce a feedback effect of the thermal field on the electric field, a temperature-dependent electrical conductivity

$$\lambda_\varphi(\vartheta) = \frac{\lambda_{\varphi,0}}{1 + \alpha_\varphi(\vartheta - \vartheta_0)} \tag{6.24}$$

is assumed, where $\lambda_{\varphi,0} = \lambda_\varphi(\vartheta = \vartheta_0)$ denotes the electrical conductivity at the reference temperature ϑ_0 and α_φ represents the linear temperature coefficient. Essential boundary conditions for the electric and the structural field are chosen as follows:

$$\begin{aligned} \varphi = 0\,, \quad u_x = u_y = u_z = 0 \quad &\text{at } z = 0\,, \\ \varphi = \varphi_{\max}\left(1 - \exp\left(-\dot{\varphi}_0/\varphi_{\max} t\right)\right) \quad &\text{at } z = L\,, \end{aligned} \tag{6.25}$$

where $\varphi_{\max} = 3\,\mathrm{V}$ and $\varphi_0 = 2.25\,\mathrm{V}$. On the free surfaces of the beam, the radiative heat transfer is approximated according to $\bar{q} = \bar{q}_{\mathrm{r}} = \varepsilon\sigma_{\mathrm{sb}}(\vartheta^4 - \vartheta_\infty^4)$, where $\vartheta_\infty = \vartheta_0 = 273.15\,\mathrm{K}$ represents the reference temperature. At the clamped end of the beam, an adiabatic boundary condition $q = 0$ is assumed.

All fields are discretized by an FE mesh consisting of $5 \times 2 \times 60$ triquadratic solid elements strongly refined towards the clamped end of the beam, as illustrated in Figure 6.66, to accurately resolve the expected strong curvature in that region and solved using the commercial FEM software ANSYS [3]. For the electric field, 20-noded SOLID231 elements are used, 20-noded SOLID90 elements are chosen for the thermal field, and 20-noded

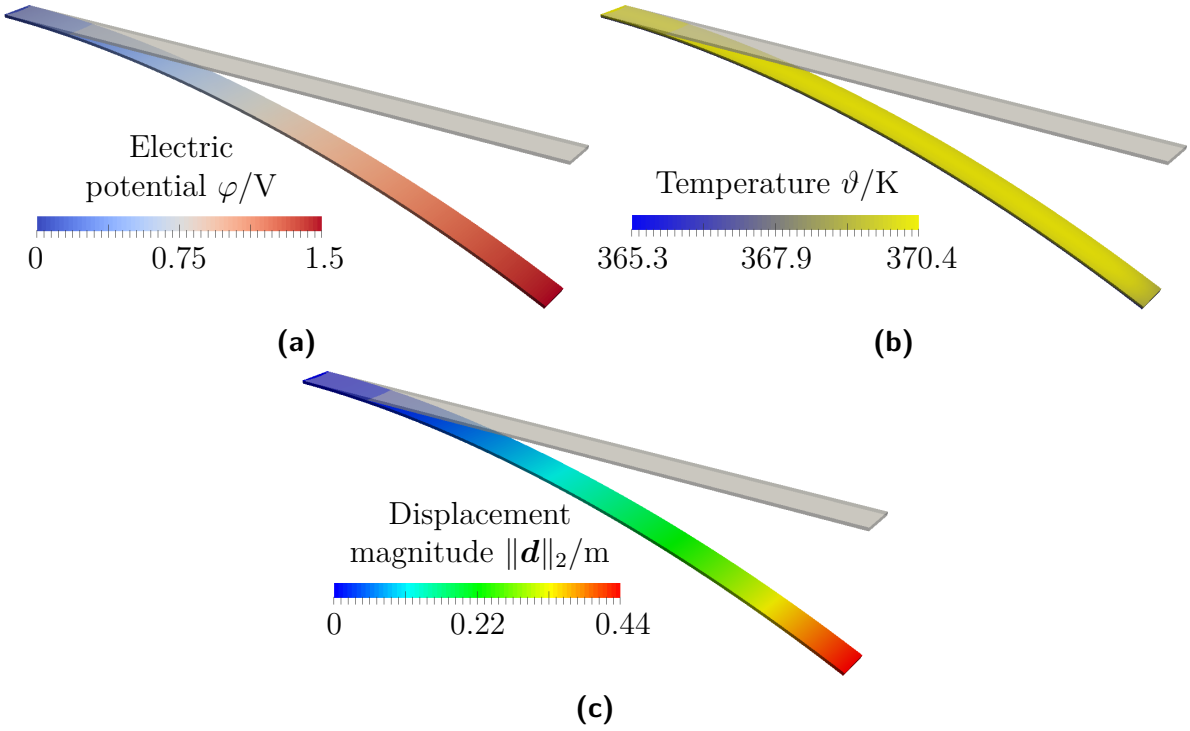


Figure 6.67: (a) Electric potential φ, (b) temperature ϑ and (c) displacement magnitude $\|\boldsymbol{d}\|_2$ at time $t = T$.

SOLID186 elements are employed for the structural problem. The electric and thermal field are solved using the trapezoidal rule, while the Newmark method with the parameters $\beta = 0.25$ and $\gamma = 0.5$ is applied for the structural field. In all cases, an identical time step size $\Delta t = 0.5\,\mathrm{s}$ is used. It is kept constant throughout the simulation over a time interval $T = 100\,\mathrm{s}$.

In the partitioned solution strategy, we first solve the electric field to obtain the electric potential, which leads to a Joule heating to be applied to the thermal problem. Due to the internal dissipative heating, the temperature increases and is passed over to the structural problem, where the thermal expansion causes a displacement, which is finally passed back to the electric field. The procedure continues until we reach an absolute convergence criterion $\|\mathbf{r}_{j+1}^k\|_2 < \varepsilon_{\mathrm{abs}} = 10^{-3}$ or a relative criterion $\|\mathbf{r}_{j+1}^k\|_2 / \|\mathbf{r}_{j+1}^0\|_2 < \varepsilon_{\mathrm{rel}} = 10^{-2}$.

Figure 6.67 illustrates the electric potential φ, temperature ϑ, and displacement magnitude $\|\boldsymbol{d}\|_2$ at time $t = T$. Figure 6.68 shows the evolution of the tip temperature increase $\Delta\vartheta$ and displacement d_y. Evidently, the numerical solutions obtained in this work are in good agreement with the results reported by Erbts [44, p. 149].

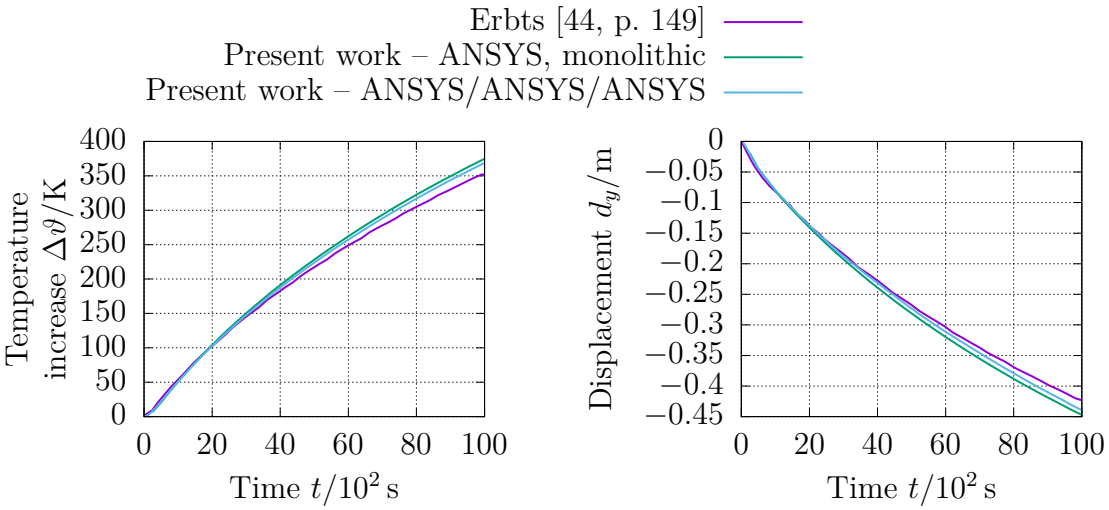


Figure 6.68: Temperature increase $\Delta\vartheta$ and displacement d_y of the tip of the bimetallic beam.

7 Advanced Applications

In this chapter, the partitioned solution approach is applied to more sophisticated problems. As the title of this work suggests, the problems are adopted from the maritime industry, where a major part of applications are governed by FSI in fact. Here, the analysis of the FSI of a floating offshore wind turbine (OWT) and the berthing maneuver of a crew transfer vessel to an OWT are of particular interest. Substantiated by the results obtained in these numerical studies, it is demonstrated that the partitioned solution approach is suitable even for such complex kinds of strongly-coupled problems.

7.1 Floating Offshore Wind Turbine

In recent years, offshore wind energy has become an increasingly attractive alternative to supply environmentally friendly, sustainable energy and replace conventional energy sources. Many countries in the world have already identified the huge potential of wind energy to achieve their ambitious climate targets and to contribute to decelerating global warming by reducing carbon dioxide emissions. During the last decade, floating OWTs have gained much interest as they overcome the limitation of conventional fixed-foundation OWTs to coastal areas, where the water is shallow enough to ground the platforms. Floating platforms, in contrast, may also be installed in higher water depths. Far away from the shore, the wind blows more constantly – representing a more attractive source of energy. Recent studies document that floating OWTs can achieve up to twice the amount of full load hours as compared to land-based wind turbines [63, p. 6], and they can also reach significantly higher energy harvesting rates than near-shore OWTs. In addition, floating OWTs may be produced onshore before being installed offshore quite easily. These factors render floating OWTs technically and economically attractive.

It is obvious that floating OWTs also come with great challenges from a technical point of view. There are hydrodynamic forces that act on the platform permanently, inducing a motion of the platform as well as of the tower structure and the turbine mounted on top. Due to the significant interaction between waves and current, airflow, and the floating structure, it is essential to treat the problem as a strongly-coupled FSI problem to assess the performance of the OWT under different operating conditions and to obtain an accurate estimate of the acting hydro- and aerodynamic forces in order to verify the structural integrity and identify potentials for an optimization of the structural design.

In the following, we present an innovative concept for a floating OWT developed in the scope of the project “Hydrodynamic and structural optimization of a floating platform for offshore wind turbines” (HyStOH) [33], financed by the Federal Ministry of Economic Affairs and Energy (BMWi) under grant number 03SX409C. As illustrated in Figure 7.1, the downwind turbine is mounted on a symmetric floating platform equipped with four floaters that provide the required hydrostatic lift. The platform is moored at the aft such that the aerodynamic forces acting on the rotor and the lateral surfaces of the profiled tower

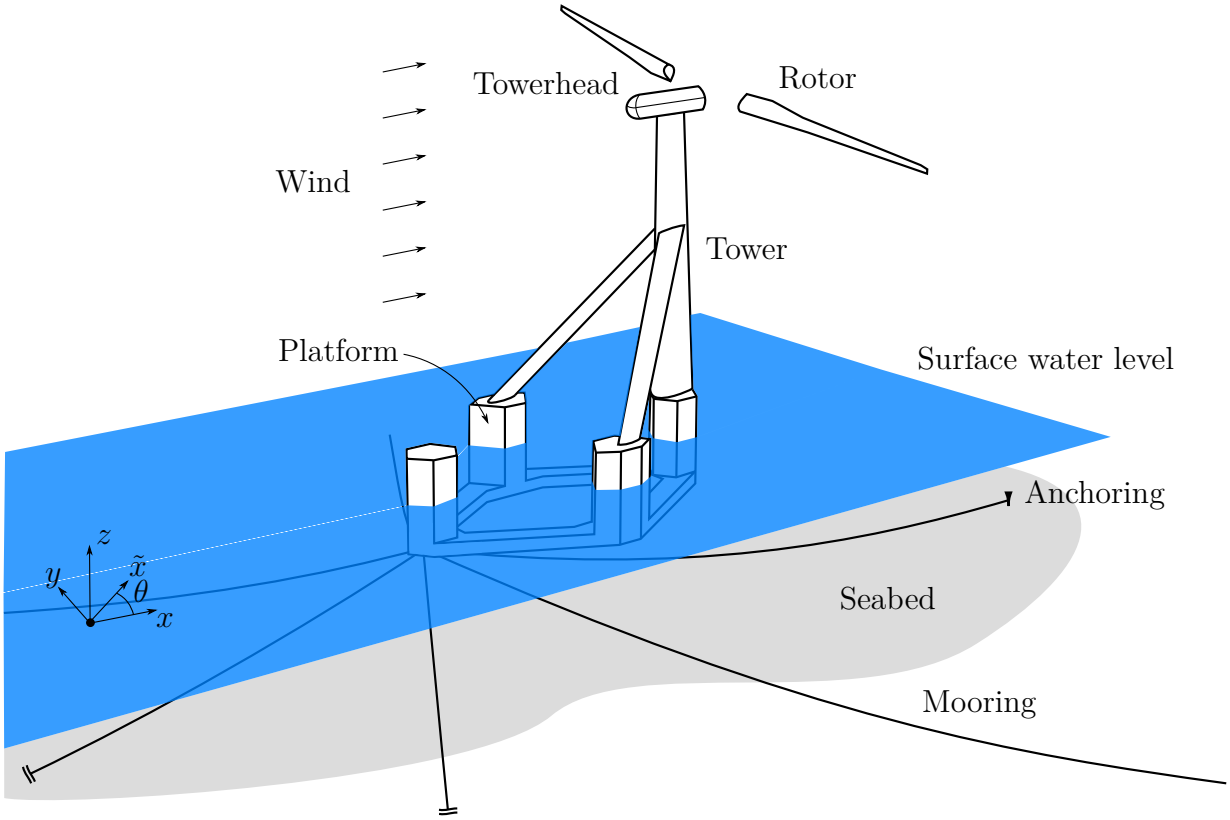


Figure 7.1: Conceptual design of a weather-vaning floating OWT.

result in a self-adjustment of the wind turbine according to the current wind direction. In order to reduce manufacturing costs, the design of the wind turbine is intentionally kept simple. Thus, main components such as the platform and the tower can be produced on shipyards quite inexpensively. A high ratio of generated power to manufacturing costs is intended to boost the competitiveness of the proposed concept and help to open up new resources of wind energy offshore.

For the numerical studies conducted in this work, a constant density $\rho_\mathrm{a} = 1.2\,\mathrm{kg/m^3}$ and kinematic viscosity $\nu_\mathrm{a} = 1.48 \times 10^{-5}\,\mathrm{m^2/s}$ are chosen for the air, which flows at a rated speed $v_\mathrm{a} = 12.3\,\mathrm{m/s}$ in the direction of the global x-axis. For the sea water, a density $\rho_\mathrm{w} = 1.026 \times 10^3\,\mathrm{kg/m^3}$ and a kinematic viscosity $\nu_\mathrm{w} = 10^{-6}\,\mathrm{m^2/s}$ are assumed. Not necessarily identical to the direction of the airflow, the current direction $\tilde{x}$ may enclose a misalignment angle θ with the global x-axis. In what follows, two different sea states will be considered. For the first sea state, we assume regular waves based on Airy wave theory with an elevation $\zeta = 3\,\mathrm{m}$ and period $T = 8\,\mathrm{s}$ corresponding to an angular frequency $\omega = 2\pi/T \approx 0.79\,\mathrm{rad/s}$. In the second sea state, we choose a JONSWAP spectrum with a significant wave height $H_\mathrm{s} = 3.44\,\mathrm{m}$ and peak period $T_\mathrm{p} = 8.96\,\mathrm{s}$, corresponding to an angular velocity $\omega_\mathrm{p} \approx 0.70\,\mathrm{rad/s}$. Proposed by Hasselmann et al. [64] as an enhancement to the Pierson-Moskowitz spectrum developed by Pierson et al. [129], this spectrum is based on the equation [103, p. 184–186]

$$S(\omega) = \frac{\alpha g^2}{\omega^5} \exp\left(-\frac{5}{4}\left(\frac{\omega_\mathrm{p}}{\omega}\right)^4\right) \gamma^r \tag{7.1}$$

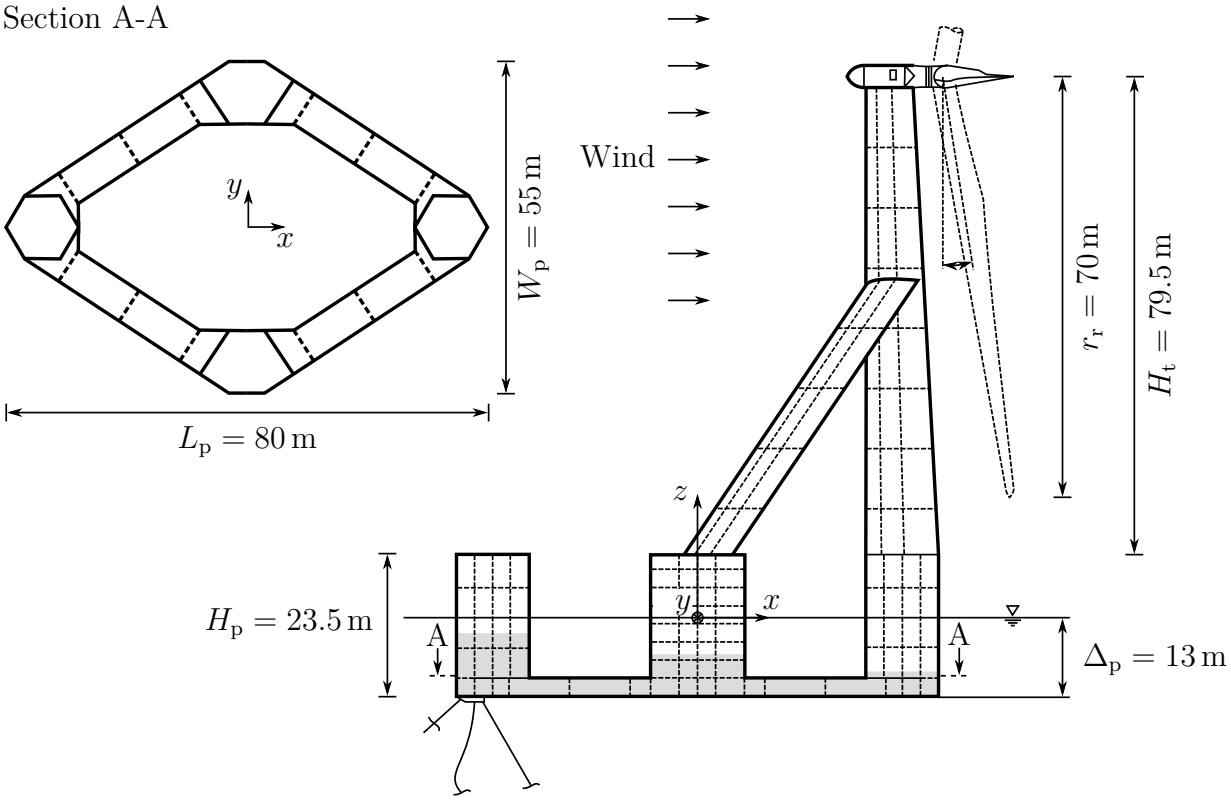


Figure 7.2: Geometry of the floating OWT. The dimensions of the floating platform are detailed in Section A-A.

for the spectral density, where

$$\begin{aligned} &\alpha \approx \frac{5}{16}\frac{H_\mathrm{s}^2\omega_\mathrm{p}^4}{g^2(1+\eta)}\,, \qquad && g = 9.81\,\mathrm{m/s^2}\,, && \eta \approx (\gamma-1)/6\,, \\ &\gamma = 3.3 && r = \exp\left(-\frac{(\omega-\omega_\mathrm{p})^2}{2\sigma^2\omega_\mathrm{p}^2}\right)\,, && \sigma = \begin{cases} 0.07 & \text{if } \omega \le \omega_\mathrm{p} \\ 0.09 & \text{if } \omega > \omega_\mathrm{p} \end{cases}\,. \end{aligned} \tag{7.2}$$

For the preliminary study conducted here, the misalignment angle is taken as $\theta = 0°$.

Figure 7.2 gives the geometry and main dimensions of the OWT. Symmetric with respect to the x- and y-axis, the floating platform is equipped with four hexagonal floaters – at the fore and aft, and at port and starboard – to supply the required hydrostatic lift and floating stability. The floaters are connected by horizontal girders, which simultaneously act as heave plates to dampen the motion induced by the hydrodynamic forces. The overall length of the platform amounts to $L_\mathrm{p} = 80\,\mathrm{m}$ to compensate the high forces acting downwind, which lead to high pitch moments about the y-axis. With a width of $W_\mathrm{p} = 55\,\mathrm{m}$, the platform is still able to pass the locks on the way to the final installation site. The floaters have a total height $H_\mathrm{p} = 23.5\,\mathrm{m}$, while the horizontal girders exhibit a height $h_\mathrm{p} = 3\,\mathrm{m}$. Internally, longitudinal and transverse bulkheads partition the platform hull into several compartments to reinforce the structure and carry the ballast water. To equilibrate the full construction under load-free and calm-water conditions, the floater at the aft is flooded with a larger amount of water than the ones at the sides and at the fore. In ballasted conditions, the draft of the platform amounts to $\Delta_\mathrm{p} = 13\,\mathrm{m}$. The platform

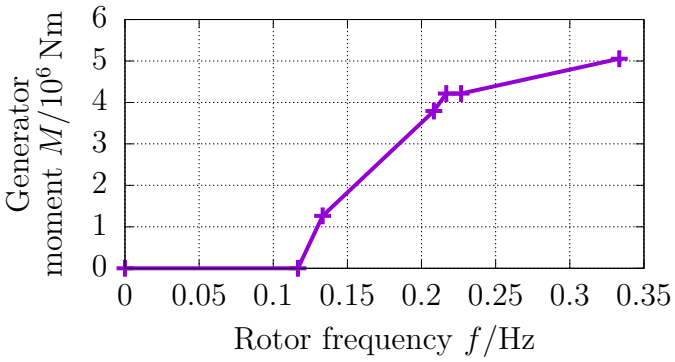


Figure 7.3: Generator moment M versus rotor frequency f.

mass amounts to $m_\mathrm{p} = 1.81 \times 10^6\,\mathrm{kg}$, and the moments of inertia about the center of gravity (COG) at $C_\mathrm{p}(0, 0, -3.65)\,\mathrm{m}$ are $\Theta_{\mathrm{p},x} = 6.03 \times 10^8\,\mathrm{kg\,m^2}$, $\Theta_{\mathrm{p},y} = 9.91 \times 10^8\,\mathrm{kg\,m^2}$, and $\Theta_{\mathrm{p},z} = 1.39 \times 10^9\,\mathrm{kg\,m^2}$. For the ballast water, a mass $m_\mathrm{b} = 5.30 \times 10^6\,\mathrm{kg}$ and moments of inertia $\Theta_{\mathrm{b},x} = 1.36 \times 10^9\,\mathrm{kg\,m^2}$, $\Theta_{\mathrm{b},y} = 2.51 \times 10^9\,\mathrm{kg\,m^2}$, and $\Theta_{\mathrm{b},z} = 3.82 \times 10^9\,\mathrm{kg\,m^2}$ about the COG at $C_\mathrm{b}(-3.16, 0, -10.1)\,\mathrm{m}$ are assumed.

Attached to the aft of the platform, the mooring consists of $n_\mathrm{m} = 6$ radially symmetric lines. All mooring properties used in this study are taken from [138, p. 26]. Each line has a diameter $D_\mathrm{m} = 7.66 \times 10^{-2}\,\mathrm{m}$ and a specific mass $\bar{m}_\mathrm{m} = 113.35\,\mathrm{kg/m}$. The material is assumed to be linear elastic with a Young's modulus $E_\mathrm{m} = 1.64 \times 10^{11}\,\mathrm{N/m^2}$ and Poisson's ratio $\nu_\mathrm{m} = 0.3$. A mass-proportional damping factor $a_\mathrm{m} = 0.02$ reflects the damping properties of the mooring system. Each with a length $L_\mathrm{m} = 200\,\mathrm{m}$, the mooring lines are anchored on the seabed at a water depth $H_\mathrm{w} = 50\,\mathrm{m}$ in a radius $r_\mathrm{m} = 192.62\,\mathrm{m}$ around the platform attachment point. The overall mooring mass amounts to $m_\mathrm{m} = \bar{m}_\mathrm{m} n_\mathrm{m} L_\mathrm{m} \approx 1.30 \times 10^5\,\mathrm{kg}$. The hydrostatic lift is accounted for by a linearly varying fluid pressure $p(z) = -\rho_\mathrm{w} g z$ and an added mass coefficient $\zeta = 1$ includes the effect of increased inertia of the submerged structure.

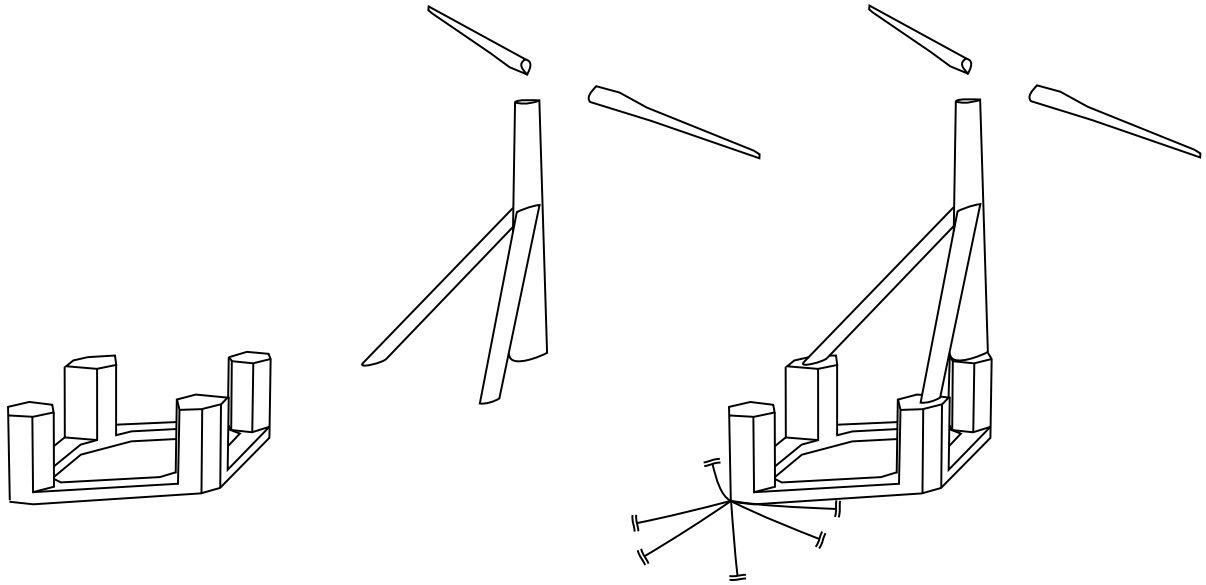
The tower exhibits a NACA0035 cross section with chord length $L_\mathrm{t}^{(1)} = 12\,\mathrm{m}$ narrowed down to $L_\mathrm{t}^{(2)} = 8\,\mathrm{m}$ towards the towerhead. The tower acts similar to an airfoil, and the large lateral surfaces assist in turning the OWT towards the wind direction. It has a mass $m_\mathrm{t} = 3.07 \times 10^5\,\mathrm{kg}$ and the moments of inertia about the COG at $C_\mathrm{t}(24.6, 0, 39.8)\,\mathrm{m}$ are $\Theta_{\mathrm{t},x} = 1.47 \times 10^8\,\mathrm{kg\,m^2}$, $\Theta_{\mathrm{t},y} = 1.50 \times 10^8\,\mathrm{kg\,m^2}$, and $\Theta_{\mathrm{t},z} = 6.48 \times 10^7\,\mathrm{kg\,m^2}$.

The towerhead has a mass $m_\mathrm{h} = 2.22 \times 10^5\,\mathrm{kg}$ and is located at a height $H_\mathrm{h} = 90\,\mathrm{m}$ above the design water line. The COG is located at $C_\mathrm{h}(38.82, 0, 90)\,\mathrm{m}$, and the moments of inertia in the body coordinate system amount to $\Theta_{\mathrm{h},x} = 1.01 \times 10^9\,\mathrm{kg\,m^2}$, $\Theta_{\mathrm{h},y} = 1.19 \times 10^9\,\mathrm{kg\,m^2}$, and $\Theta_{\mathrm{h},z} = 1.78 \times 10^8\,\mathrm{kg\,m^2}$.

The rotor features two blades with a cone angle $\beta_\mathrm{r} = 9°$ and zero tilt angle. It has a radius $r_\mathrm{r} = 70\,\mathrm{m}$ and turns in clockwise direction as seen from the direction of the airflow. At $t = 0\,\mathrm{s}$, the nominal rotor frequency is $f = 0.23\,\mathrm{1/s}$, corresponding to an angular frequency $\omega = 2\pi f \approx 1.42\,\mathrm{rad/s}$. At rated conditions, the generator delivers a nominal moment $M = 4.21 \times 10^6\,\mathrm{Nm}$ and a power $P = 6 \times 10^6\,\mathrm{W}$. As graphed in Figure 7.3, the generator moment M varies with the rotor frequency f and acts in counter-clockwise direction as seen from the direction of the airflow. The rotor has a mass $m_\mathrm{r} = 1.18 \times 10^5\,\mathrm{kg}$, and the moments of inertia about the COG at $C_\mathrm{r}(42.16, 0, 90)\,\mathrm{m}$ are $\Theta_{\mathrm{r},x} = 4.11 \times 10^7\,\mathrm{kg\,m^2}$, $\Theta_{\mathrm{r},y} = 1.03 \times 10^6\,\mathrm{kg\,m^2}$, and $\Theta_{\mathrm{r},z} = 4.21 \times 10^7\,\mathrm{kg\,m^2}$.

Table 7.1: Mass properties of the individual components of the floating OWT.

Component	COG/m			Mass m/kg	Principal moments of inertia/kg m^2		
	x	y	z		Θ_x	Θ_y	Θ_z
Platform	0	0	−3.65	1.81×10^6	6.03×10^8	9.91×10^8	1.39×10^9
Ballast water	−3.16	0	−10.1	5.30×10^6	1.36×10^9	2.51×10^9	3.82×10^9
Tower	24.6	0	39.8	3.07×10^5	1.47×10^8	1.50×10^8	6.48×10^7
Towerhead	38.82	0	90	2.22×10^5	1.01×10^9	1.19×10^9	1.78×10^8
Rotor	42.16	0	90	1.18×10^5	4.11×10^7	1.03×10^6	4.21×10^7



(1) Water flow around platform

(2) Airflow around tower and rotor

(3) Structural motion

Figure 7.4: Partitioning of the coupled FSI problem into separate subproblems.

Table 7.1 once again summarizes the mass properties of the individual components of the floating OWT.

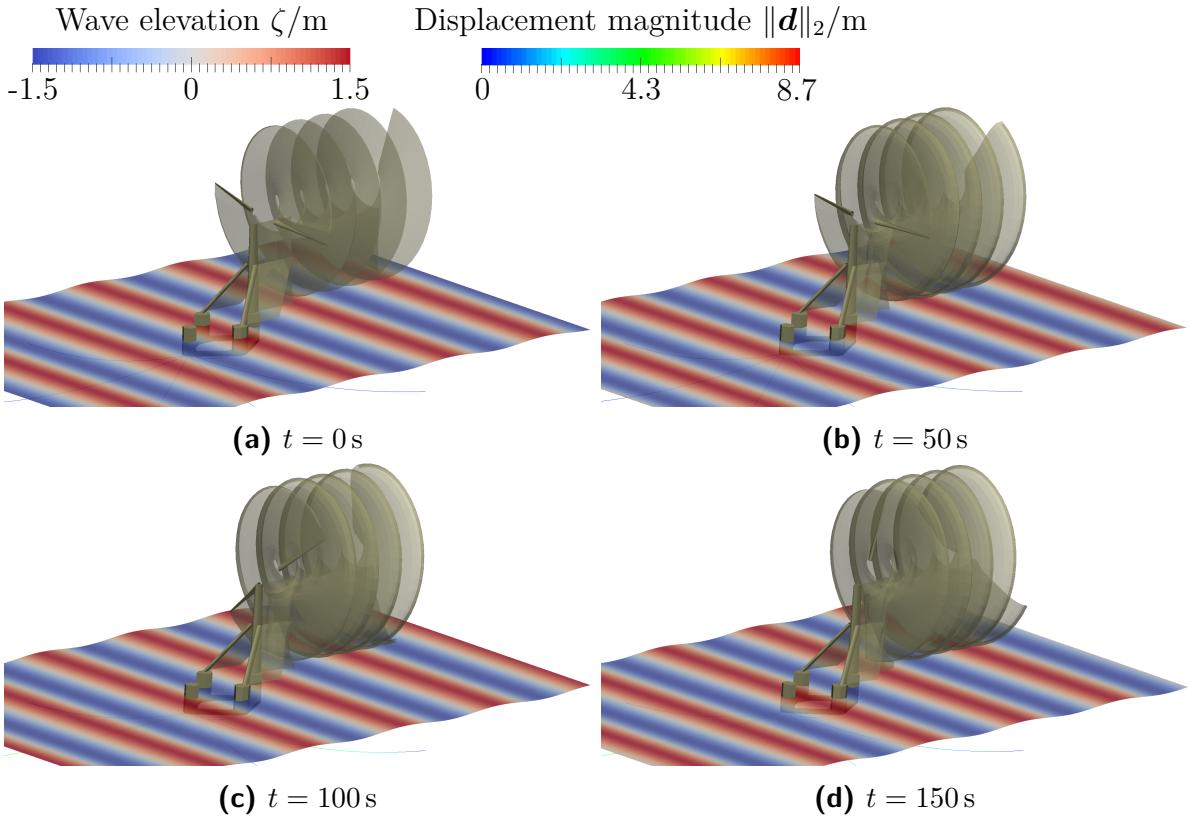
For the numerical analysis of the FSI of the floating OWT, we follow a partitioned solution approach and split the coupled problem into three separate subproblems, as illustrated in Figure 7.4. In subproblem (1), we compute the flow field around the floating platform. In order to save computational cost, we again resort to the assumption of potential flow and solve this fluid problem using the BEM implemented in *pan*MARE [12]. Due to the use of the BEM, only the surface of the platform needs to be discretized. Here, a surface mesh consisting of 1,400 first-order panels is employed. Since the motion behavior of the platform is dominated by the hydrodynamic forces rather than the aerodynamic forces acting above the water line, the influence of the airflow on the platform is neglected. As a second fluid problem, the airflow around the tower and the rotor needs to be computed. A potential flow is likewise assumed for this second flow problem. For its numerical treatment, we choose the boundary element code *pan*MARE [12] together with a surface mesh comprising 3,960 body and 4,900 wake panels. In the third and last subproblem, we use the commercial software package ANSYS [3] to compute the motion

response of the structure to the acting hydro- and aerodynamic forces. In the preliminary study conducted in this work, we primarily focus on the motion behavior of the entire OWT and, hence, assume the floating platform, the tower, and the rotor to be rigid. Since the elasticity of the mooring can be expected to have a significant influence on the motion of the OWT, this part is modeled as elastic. Other components can be modeled as elastic as well, if their deformation is considered to have a notable impact on the flow fields or if their internal stress state is of interest. Based on the assumptions above, the total mass $m_\mathrm{c} = m_\mathrm{p} + m_\mathrm{b} + m_\mathrm{t} + m_\mathrm{h} = 7.63 \times 10^6\,\mathrm{kg}$ of the platform, the ballast water, the tower, and the towerhead can be concentrated in a common COG at $C_\mathrm{c}(0, 0, -3.65)\,\mathrm{m}$. By using an MPC184 joint element, the rotor is constrained to rotate about the local $\hat{x}_\mathrm{r}$-axis of the rotor coordinate system initially located at $C_\mathrm{r}(42.16, 0, 90)\,\mathrm{m}$ with all its axes parallel to the global coordinate system. The rotor mass is represented by a MASS21 element with rotary inertia and concentrated at the origin of the local rotor coordinate system. Using an MPC184 beam element, the rotor coordinate system is connected to the common COG C_c of the platform, ballast water, tower, and towerhead. Hence, the platform, ballast water, tower, towerhead, and rotor represent a multibody system. In order to compute the forces and moments acting on the aforementioned parts of the floating OWT, a surface discretization coinciding with the panel discretizations for the two fluid problems is introduced to integrate the traction from the fluid problems. It is emphasized that the surface discretization in the structural solver does not introduce any additional degrees of freedom, but serves for the sole purpose of integration only. For the discretization of each of the mooring lines, we use 14 two-noded geometrically nonlinear BEAM188 elements based on Timoshenko beam theory, with three translational and three rotational degrees of freedom per node. The translational degrees of freedom of the mooring lines at the anchoring point are completely fixed, while the mooring lines are constrained to follow the motion of the multibody system at the point of attachment to the floating platform.

A time period $T = 150\,\mathrm{s}$ is simulated, split into comparably large, equally-sized time increments $\Delta t = 0.1\,\mathrm{s}$. For both fluid subproblems, the backward Euler scheme is chosen to march in time, whereas the slightly damped Newmark scheme with parameters $\beta = 0.2525$ and $\gamma = 0.5050$ is selected for the structural subproblem. In the coupled solution procedure, the water and the airflow are solved first and the computed traction at the water/platform, air/tower, and air/rotor interface is passed over to the structural solver. By integrating the traction over the discretized surface of the platform, tower, and rotor, and by summing the resulting point forces and associated moments around the corresponding COGs, we obtain a total force and moment, which enter the rigid body equations. After solving for the rigid body displacement with regard to the applied constraints, the displacement at the fluid-structure interfaces is passed over to the fluid solvers. This procedure is continued until either the absolute convergence criterion $\|\mathbf{r}_{j+1}^k\|_2 < \varepsilon_\mathrm{abs} = 10^{-2}$ or the relative criterion $\|\mathbf{r}_{j+1}^k\|_2 / \|\mathbf{r}_{j+1}^0\|_2 < \varepsilon_\mathrm{rel} = 10^{-6}$ is met. As usual, we apply a second-order polynomial predictor and the quasi-Newton least squares method to stabilize and accelerate the solution procedure.

For the regular waves, the partitioned solution procedure requires 6.57 iterations on average to reach the convergence criterion. Figure 7.5 depicts snapshots of the simulation results for the regular sea state at different instants of time t. Figure 7.6 graphs the motion of the common COG of platform, ballast water, tower, and towerhead, as well as the translational motion and acceleration of the towerhead versus time.

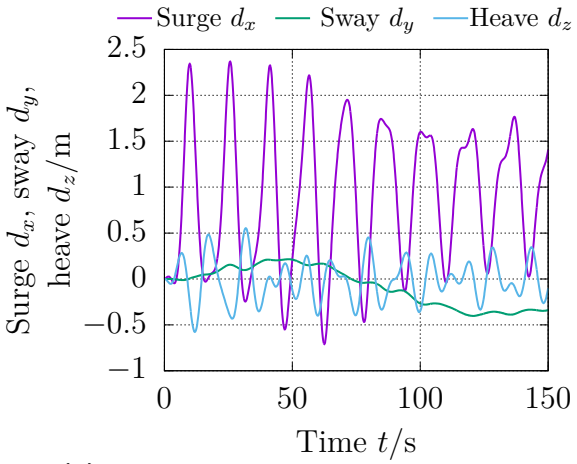
In case of the JONSWAP spectrum, the solution is acquired after 2.57 iterations on



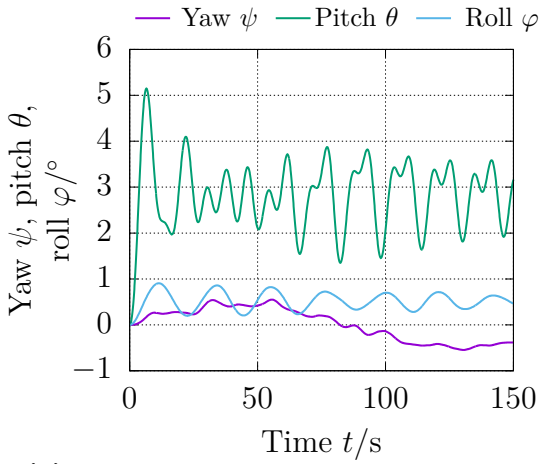
(a) $t = 0\,\mathrm{s}$ **(b)** $t = 50\,\mathrm{s}$

(c) $t = 100\,\mathrm{s}$ **(d)** $t = 150\,\mathrm{s}$

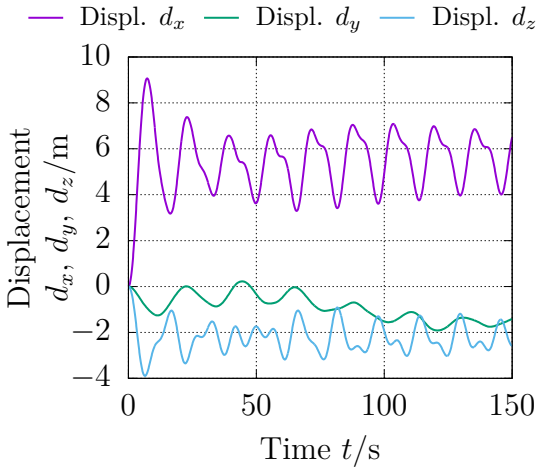
Figure 7.5: Floating OWT in regular waves of height $H_\mathrm{s} = 3\,\mathrm{m}$ and period $T = 8\,\mathrm{s}$ at different instants of time t.



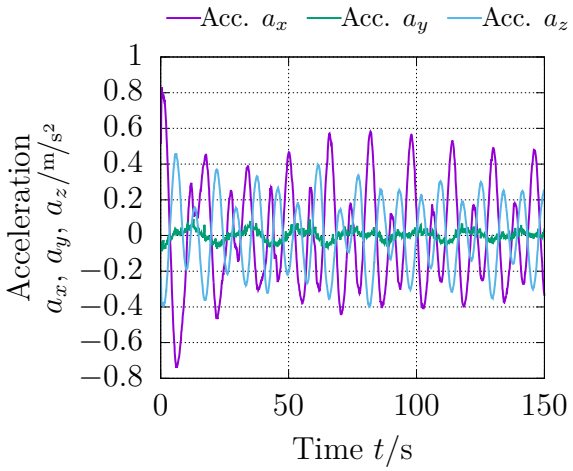
(a) Surge d_x, sway d_y, and heave d_z of the common COG C_c of platform, ballast water, tower, and towerhead



(b) Yaw ψ, pitch θ, and roll φ of the common COG C_c of platform, ballast water, tower, and towerhead



(c) Displacement d_x, d_y, and d_z of the towerhead



(d) Acceleration a_x, a_y, and a_z of the towerhead

Figure 7.6: Motion behavior of the floating wind turbine in regular waves of height $H = 3\,\mathrm{m}$ and period $T = 8\,\mathrm{s}$ over time t.

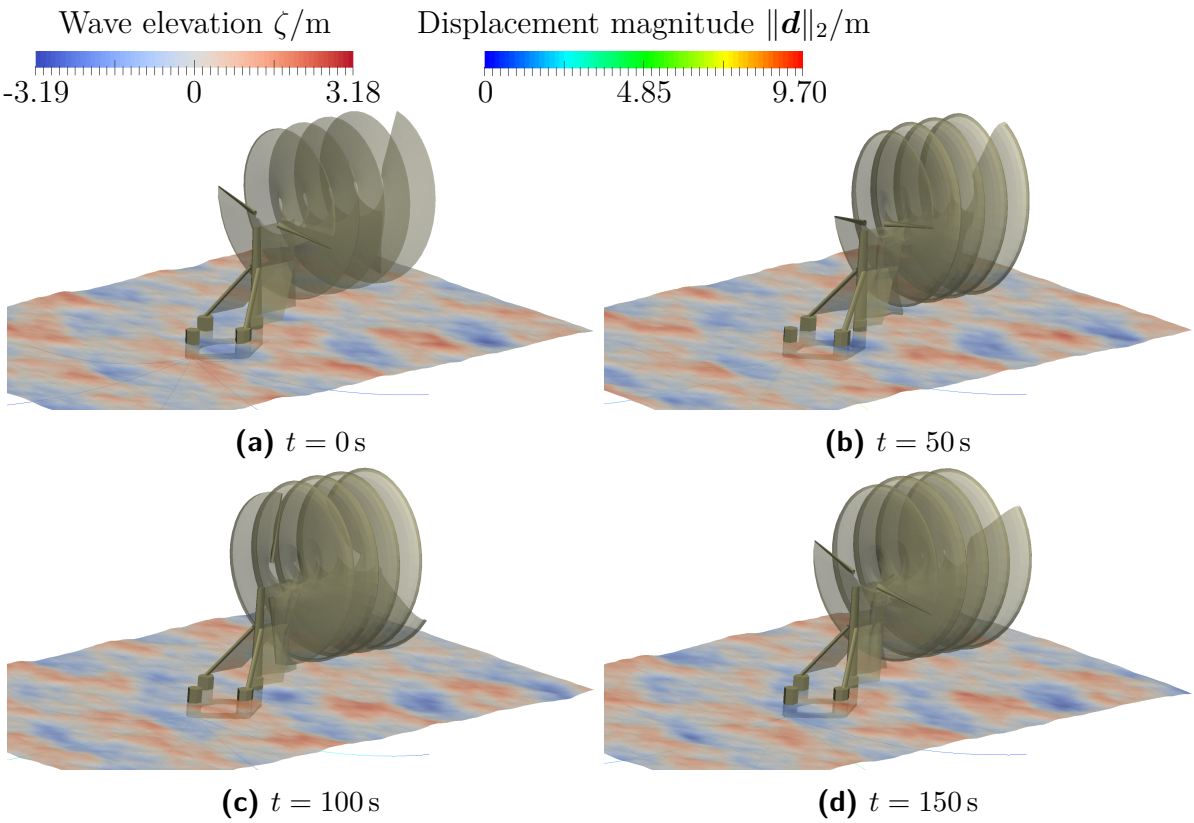


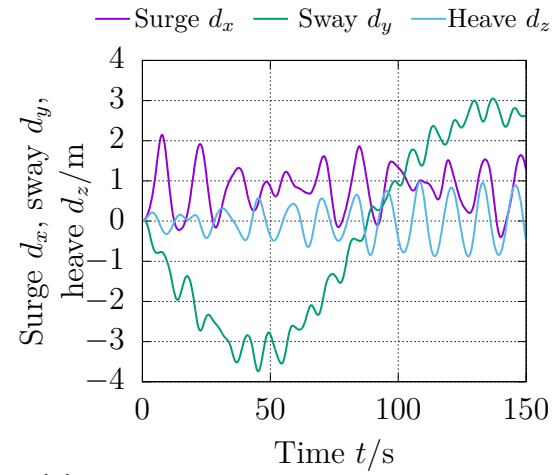
Figure 7.7: Floating OWT in JONSWAP spectrum of significant wave height $H_\mathrm{s} = 3.44\,\mathrm{m}$ and peak period $T = 8.96\,\mathrm{s}$ at different instants of time t.

average. Figure 7.7 shows several snapshots of the numerical results at different instants of time. The towerhead displacement and acceleration for this case are given in Figure 7.8.

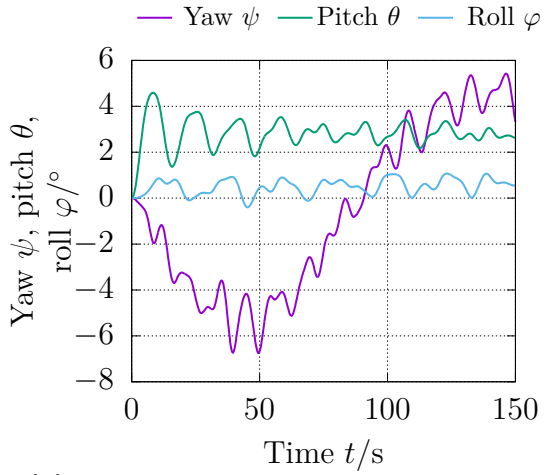
In both considered sea states, the platform motion and the towerhead displacement and acceleration stay within moderate bounds, which is why they can be expected not to have much influence on the power generation of the floating OWT. Yet, it is necessary to carry out further studies regarding the self-adjustment capability in case of a change of the wind direction. In addition, a detailed analysis of the internal stresses, especially in critical regions such as the tower or at the tower/platform connection is required to assess the structural integrity of the floating OWT also under extreme loading conditions, which have not been considered yet. For sophisticated technical applications such as the studied floating OWT, an experimental validation of the numerical results is indispensable and, thus, strongly recommended as a future research task.

7.2 Berthing Maneuver of Crew Transfer Vessel

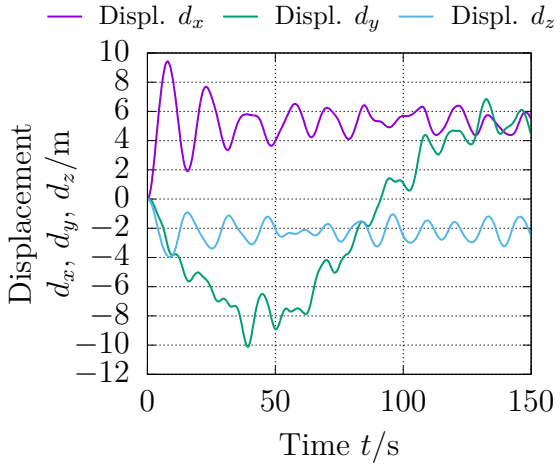
Regular maintenance and, if necessary, occasional repair are one of the key factors for a safe and reliable operation of OWTs. Due to their comparably low operational cost and their ability to carry bulky technical equipment, crew transfer vessels are still the preferred means of transportation to bring the servicing staff, tools, and spare parts from the shore to the OWT. In order to ensure that the service personnel can safely disembark from the vessel, it is essential to limit the relative motion between the ship and the boat



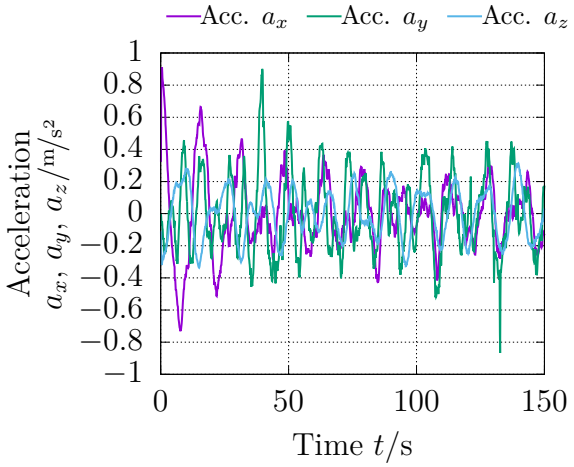
(a) Surge d_x, sway d_y, and heave d_z of the common COG C_c of platform, ballast water, tower, and towerhead



(b) Yaw ψ, pitch θ, and roll φ of the common COG C_c of platform, ballast water, tower, and towerhead



(c) Displacement d_x, d_y, and d_z of the towerhead



(d) Acceleration a_x, a_y, and a_z of the towerhead

Figure 7.8: Motion behavior of the floating wind turbine in JONSWAP spectrum of significant wave height $H_s = 3.44\,\mathrm{m}$ and peak period $T_p = 8.96\,\mathrm{s}$ over time t.

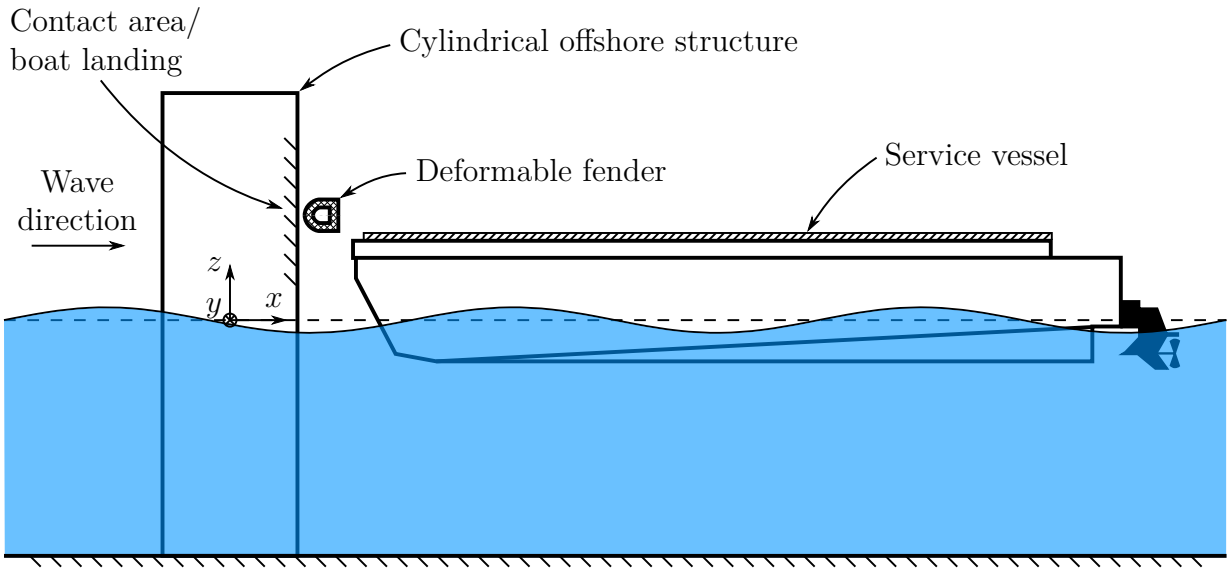


Figure 7.9: Berthing maneuver of a crew transfer vessel to an offshore structure.

landing. For this purpose, the service vessel is equipped with a fender at its bow, which is pressed towards the boat landing to establish frictional contact. The resulting vertical friction force counteracts the hydrodynamic forces acting on the ship, and it holds the bow in position such that the technicians can safely pass over to the OWT. Obviously, the landing maneuver represents a strongly-coupled FSI problem, governed by the ship motion at sea and by the presence of contact between the fender and the offshore structure. Following [97], where the problem was first presented, and building on the partitioned solution approach, this coupled process will be analyzed numerically in the following. It is believed that the findings from the simulation can help to identify the limiting conditions under which the berthing operation can still be carried out safely – and to generate a valuable data basis for the development of future transfer concepts that will help to increase the safety of the service personnel, to improve the accessibility of the offshore structure, and, last but not least, to reduce operational cost.

First, let us discuss the problem setting as shown in Figure 7.9. During the berthing operation, an elastic fender that is attached to the bow of the vessel is pushed towards the boat landing in order to create a friction force acting in vertical direction, holding the bow at rest so that the servicing staff can safely disembark from the ship to the offshore structure.

For the numerical analysis of the problem, the small deformations of the ship hull are neglected, and the service vessel is treated as rigid so as to save computational cost. Apparently, due to the mutual interaction of the rigid body and the surrounding flow field, the ship motion in the seaway represents a strongly-coupled problem. Apart from the vessel motion, the influence of the offshore foundation is taken into account for the computation of the flow field as well. In this case, the problem is a weakly-coupled one: the structure affects the wave field, but the deformations caused by the hydrodynamic forces acting on the structure are negligible. Last but not least, we have to account for the contact between the vessel's fender and the boat landing. If the acting forces exceed the limiting frictional resistance of the contact pairing, the contact partners may also slip or separate. During contact, the fender, which is usually manufactured from a rubber- or foam-like material,

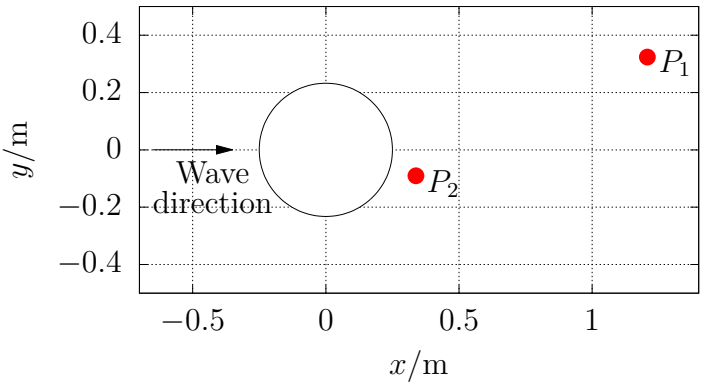


Figure 7.10: Cylinder in regular waves and position of the evaluation points $P_1(1.21, 0.33)$ and $P_2(0.34, -0.096)$.

undergoes large deformations, and there are geometrical and material nonlinearities.

In the following, the subproblems constituting the entire berthing maneuver are first analyzed individually before integrating them in a partitioned solution procedure to compute the coupled problem.

7.2.1 Cylinder in Waves

Due to the fact that most of the OWTs currently in operation are installed on a monopile, this is the foundation type chosen for this study. In a first step, the influence of a cylindrical structure on regular waves of different frequency is investigated numerically. For the sake of comparison to experimental measurements, the study is conducted at model scale. As depicted in Figure 7.10, we consider a cylinder of diameter $D = 0.5\,\mathrm{m}$ founded in a water depth $H = 2.3\,\mathrm{m}$. The wave height is chosen as $h = 0.1\,\mathrm{m}$, and the wave angular frequencies are $\omega_1 = 3\,{}^{\mathrm{rad}}/_{\mathrm{s}}$ and $\omega_2 = 5\,{}^{\mathrm{rad}}/_{\mathrm{s}}$, respectively. Based on the dispersion relation

$$\omega = \sqrt{gk\tanh(kH)}\,, \tag{7.3}$$

the wave numbers are computed as $k_1 \approx 0.94\,{}^{1}/_{\mathrm{m}}$ and $k_2 \approx 2.55\,{}^{1}/_{\mathrm{m}}$. From the equation $\lambda = 2\pi/k$, the wave lengths are calculated as $\lambda_1 \approx 6.67\,\mathrm{m}$ and $\lambda_2 \approx 2.47\,\mathrm{m}$. The waves travel in positive x-direction from left to right.

In order to analyze the wave field around the cylinder numerically, the software package *pan*MARE [12] is employed. Since the wave potential superposed by the diffraction potential of the cylinder is available in analytical form, the wave field around the cylinder can be directly evaluated at different points in space and time. Table 7.2 and 7.3 compare the wave field at different phase angles $\varphi \in \{0, \pi/2, \pi, 3\pi/2\}$ obtained with the BEM, based on the results of a laminar RANS calculation and on experimental measurements. For the RANS solution, we use the FVM, which is available in the open source CFD software package OpenFOAM [123]. Figure 7.11 graphs the wave elevation ζ versus time t for ω_1, evaluated at point P_1, and for the angular frequency ω_2, evaluated at point P_2. Evidently, we observe a fair agreement between the comparably simple (yet effective) BEM, the significantly more sophisticated and hence more expensive RANS calculation, and the experimental data.

Table 7.2: Comparison of the wave elevation ζ in the vicinity of the cylinder for an angular frequency $\omega_1 = 3\,^{\mathrm{rad}}/_{\mathrm{s}}$ and different phase angles φ.

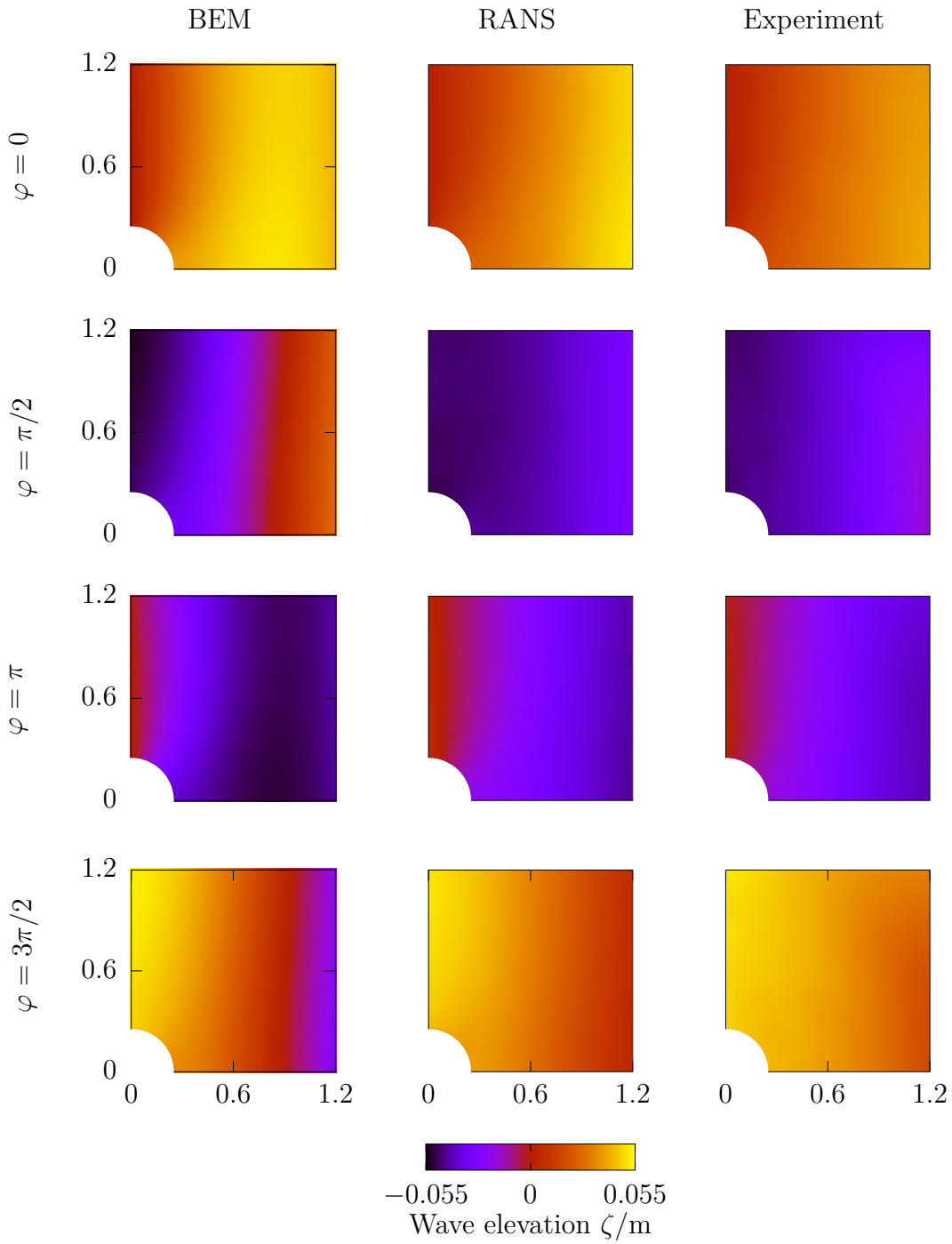
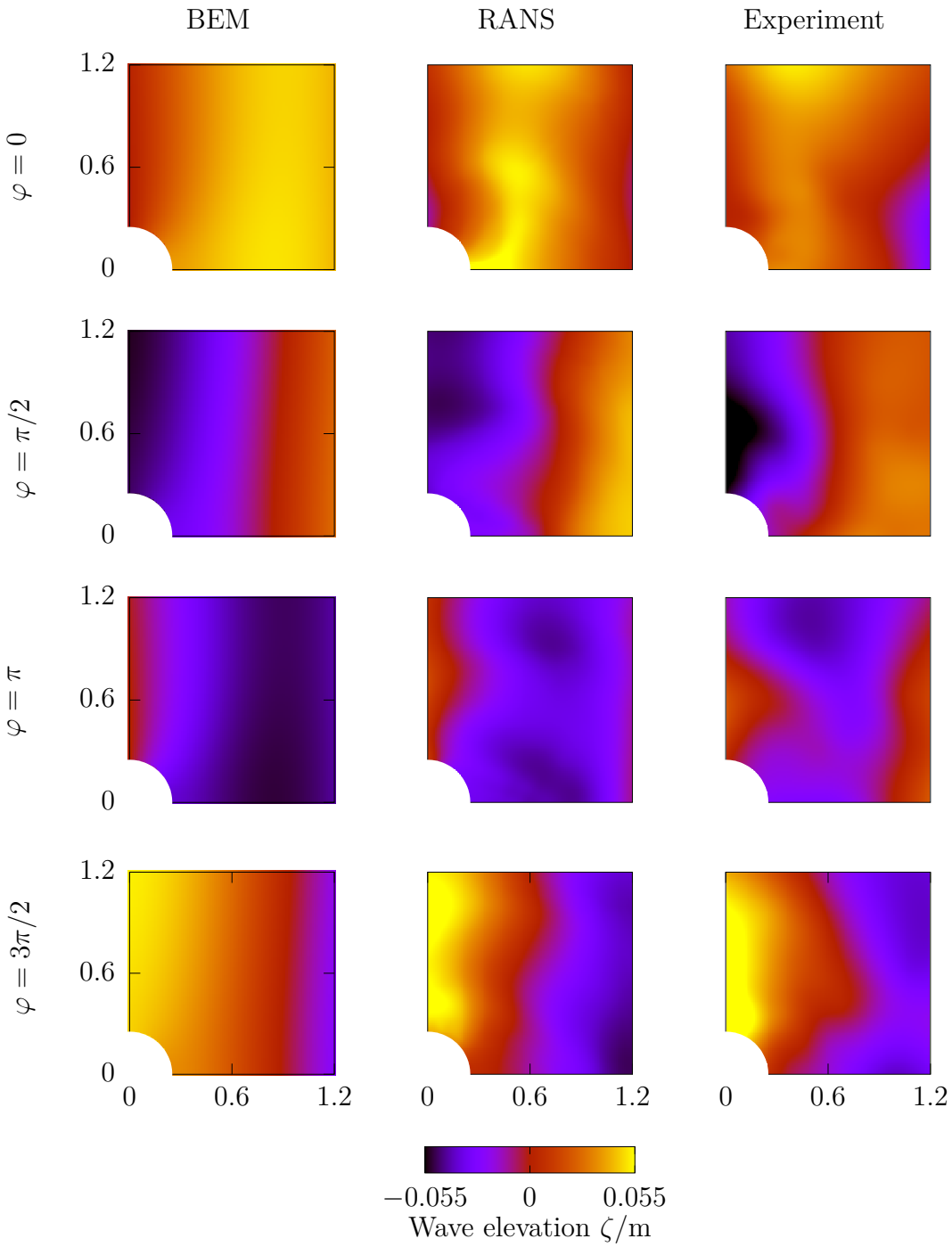


Table 7.3: Comparison of the wave elevation ζ in the vicinity of the cylinder for an angular frequency $\omega_2 = 5\,{}^{\mathrm{rad}}/_{\mathrm{s}}$ and different phase angles φ.



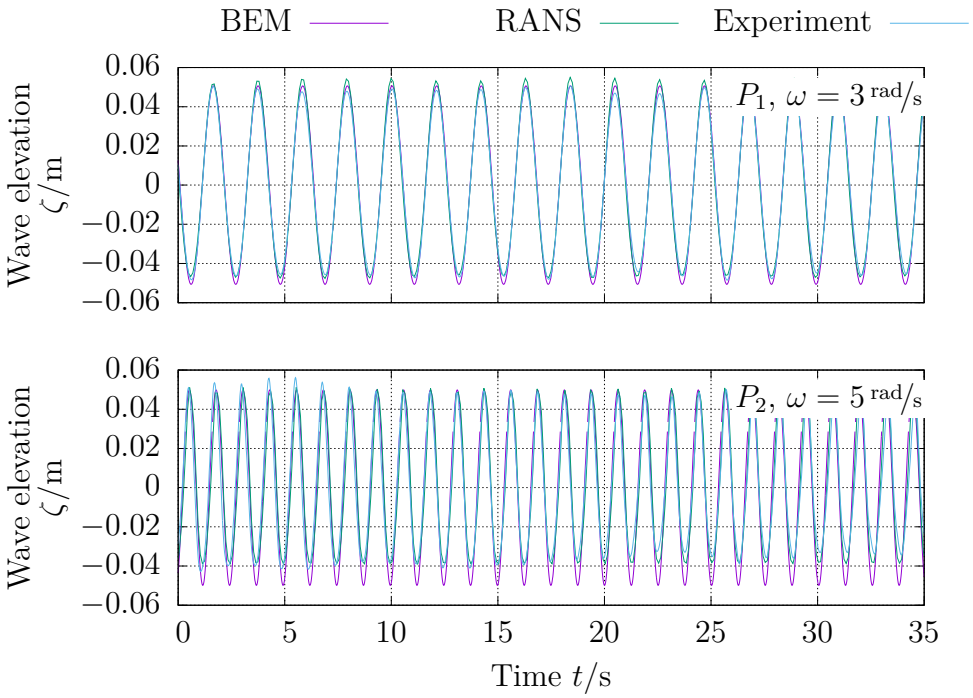


Figure 7.11: Comparison of the wave elevation ζ for an angular frequency $\omega_1 = 3\,\mathrm{rad/s}$, evaluated at point P_1 (upper graph), and for an angular frequency $\omega_2 = 5\,\mathrm{rad/s}$, evaluated at point P_2 (lower graph).

7.2.2 Crew Transfer Vessel in Waves

In the next step, we focus on the numerical analysis of the hydrodynamic behavior of the catamaran vessel in regular waves. As already mentioned in the introductory paragraph, the comparably small deflections of the ship hull are neglected, and the vessel is considered as a rigid body. Also here, the analysis is conducted at model scale to allow for a comparison to experimental measurements. The dimensions of the catamaran are given in Figure 7.12. It has a mass $m = 79.2\,\mathrm{kg}$ and the moments of inertia with respect to the COG are $\Theta_{\hat{x}} = 8.82\,\mathrm{kg\,m^2}$, $\Theta_{\hat{y}} = 31.49\,\mathrm{kg\,m^2}$, and $\Theta_{\hat{z}} = 36.66\,\mathrm{kg\,m^2}$. Degrees of freedom other than heave and pitch are constrained to zero to resemble the setting in the experiments.

For the numerical treatment of the problem, we follow a partitioned solution approach.

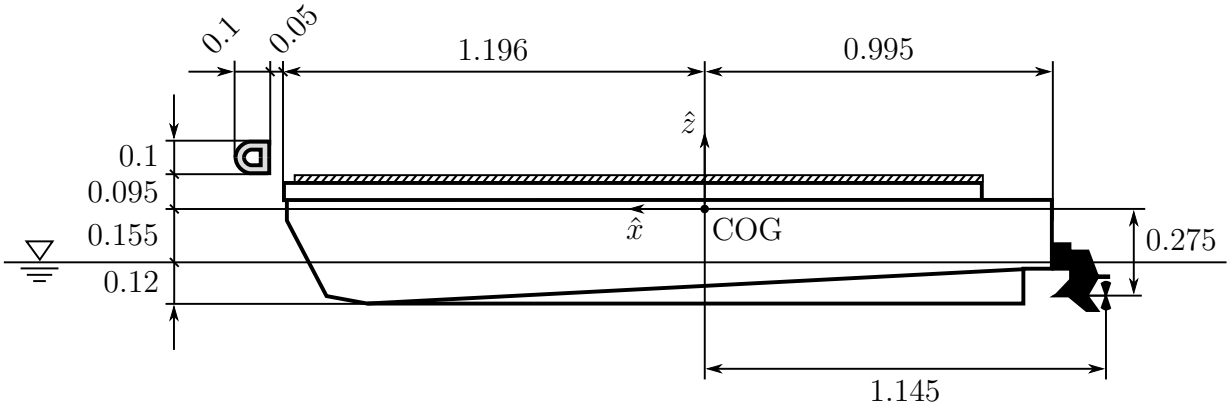


Figure 7.12: Dimensions of the catamaran service vessel. All dimensions are given in meter.

The fluid and the rigid body problem are computed as separate subproblems, and the relevant coupling quantities – that is, the rigid body displacement and the fluid traction at the fluid-structure interface – are exchanged iteratively within a time increment until both subfields are equilibrated with each other to sufficient accuracy. For the fluid field, again the BEM and the software *pan*MARE [12] are employed. In this case, however, a surface discretization is required, as a closed-form expression for the diffraction potential of the moving catamaran cannot be derived. Therefore, the surface of the catamaran is discretized by a boundary element mesh consisting of 1,508 first-order panels. Regarding the temporal discretization, the explicit Euler scheme with a time step size $\Delta t = 10^{-2}\pi/\omega$, corresponding to 200 time steps per wave period, is chosen for both the fluid problem and the rigid body.

Figure 7.13 graphs the response amplitude operator (RAO) of the catamaran. Heave and

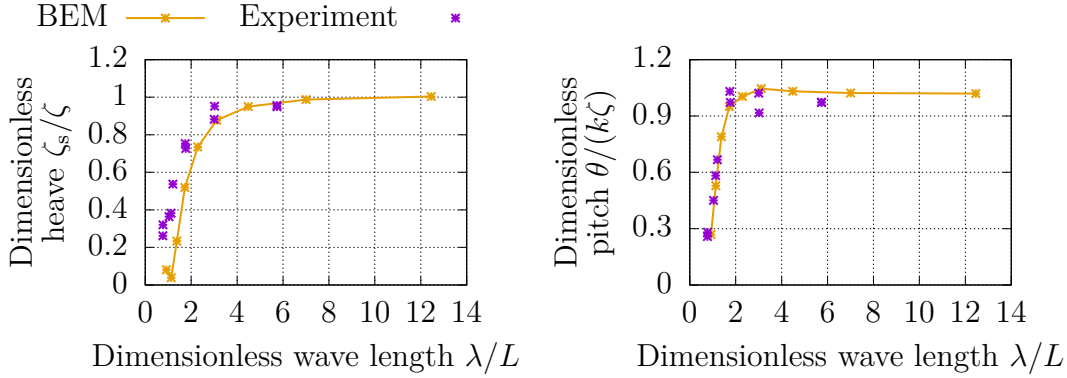


Figure 7.13: Comparison of heave and pitch motion for different wave lengths λ.

pitch motion are underestimated for $\lambda/L < 2$, whereas the motion is slightly overestimated for larger wave lengths. This discrepancy can be attributed to the fact that the free surface boundary condition is not included in the formulation. The difference in the pitch and heave error is associated to the peculiarities of the numerical differencing scheme, where differences between quantities towards the fore and the aft are evaluated, which tends to cancel the pitch motion error, while the heave motion error accumulates. Yet, the deviations are considered acceptable for the purpose of the current study.

7.2.3 Crew Transfer Vessel in Waves Including the Influence of Monopile

Following the analysis of the motion of the catamaran vessel in regular waves and free water conditions, the scope of the next study is the investigation of the influence of a cylindrical offshore structure on the hydrodynamic behavior of the catamaran. As before, the analysis is again carried out at model scale. For this purpose, the catamaran is placed in front of the cylindrical structure of diameter $D = 0.5\,\mathrm{m}$, as already considered in Section 7.2.1, heading in negative x-direction. This setting reflects a typical scenario under real-life operating conditions. For two different angular frequencies $\omega_1 = 3.509\,^{\mathrm{rad}}/_{\mathrm{s}}$ and $\omega_2 = 4.965\,^{\mathrm{rad}}/_{\mathrm{s}}$, the motion of the catamaran is compared to the case without monopile. All spatial and temporal discretization parameters are chosen as in the previous study.

Figure 7.14 and 7.15 depict the results, which confirm the expectation that the influence of the cylindrical structure diminishes towards decreasing angular frequencies or increasing

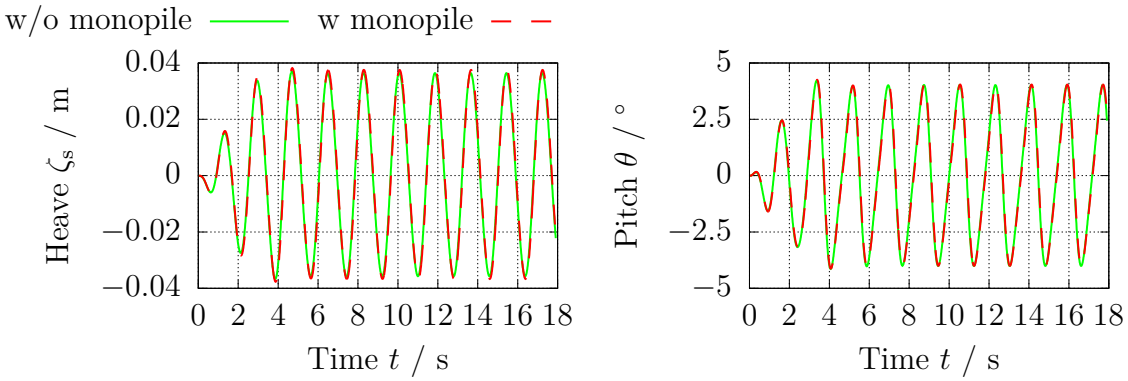


Figure 7.14: Heave and pitch motion for a wave frequency $\omega_1 = 3.509\,\mathrm{rad/s}$.

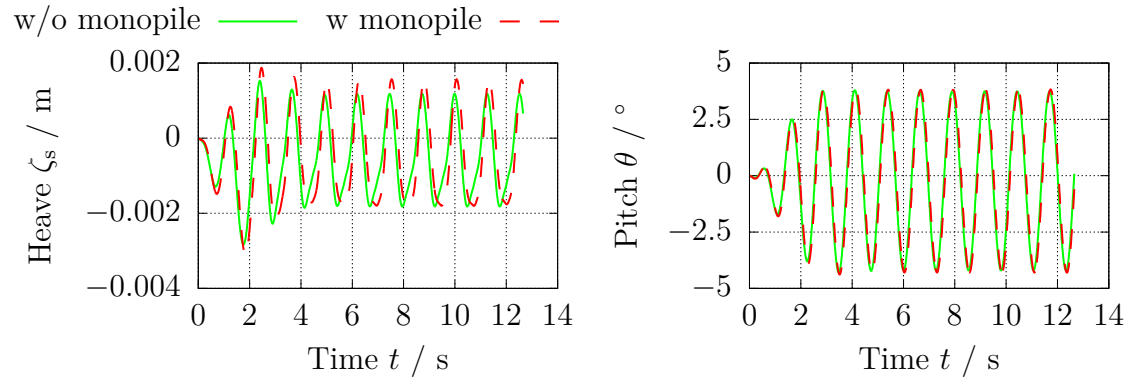


Figure 7.15: Heave and pitch motion for a wave frequency $\omega_2 = 4.965\,\mathrm{rad/s}$.

wave lengths. For the smaller angular frequency ω_1 corresponding to the larger wave length, hardly any influence of the monopile structure on the catamaran motion can be spotted.

7.2.4 Contact Problem between Fender and Monopile

During the landing maneuver, the fender plays an essential role in generating the vertical friction force, which is responsible for keeping the ship's bow in position, allowing the service personnel to disembark safely from the ship to the OWT. In the present work, we consider two D-profile fenders with different wall thicknesses, consisting of rubber materials of different stiffness. Figure 7.16 sketches the geometries of the fenders. Due to the higher wall thickness and the stiffer material, the first fender exhibits a significantly higher stiffness than the second one.

In order to capture the material behavior of the fenders, we choose either a Neo-Hooke or Mooney-Rivlin material as introduced in Section 3.1.3. Several material samples are subjected to uniaxial tension and compression tests to generate the required input for a subsequent parameter identification carried out to determine the unknown coefficients in these hyperelastic material models. For the stiff fender material, for instance, Figure 7.17 indicates that the hyperelastic material models are capable of resembling the material behavior of the rubber material over the whole strain range. Note that in this case, the Neo-Hooke model and the Mooney-Rivlin model coincide, as the parameter identification produces a parameter $C_{01} = 0$ for the Mooney-Rivlin model. Despite the fact that the

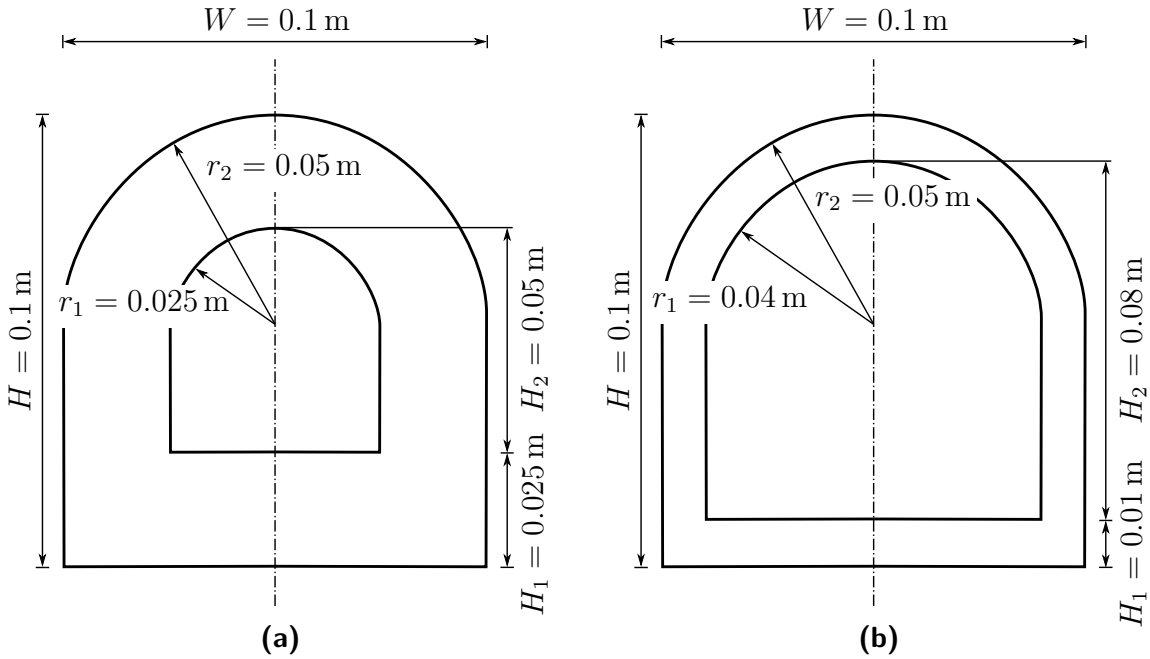


Figure 7.16: Geometry of the (a) hard and (b) soft fender.

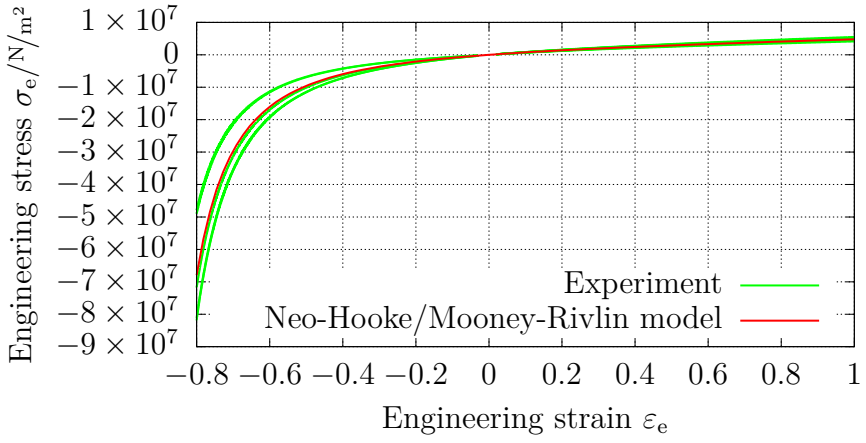


Figure 7.17: Least squares fit of the Neo-Hooke model (3.53) and the Mooney-Rivlin model (3.54) for uniaxial tension and compression tests for the stiff fender material.

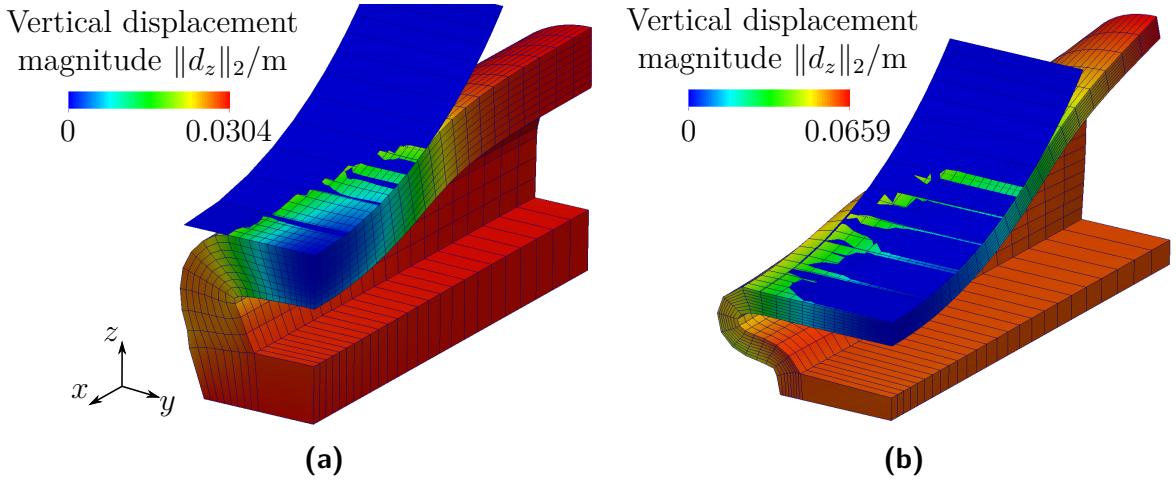


Figure 7.18: Deformed (a) hard and (b) soft fender under compression loading.

stress-strain relationship is captured very accurately for the present case, the parameters obtained from the least squares fit to the uniaxial tests should be expected to change if fully three-dimensional loading conditions are considered. In order to account for this fact, the entire fenders are subjected to a quasi-static displacement-controlled compression test. At the maximum compression level, the reaction force at the bottom of the fender is measured and fed to an optimization procedure based on a three-dimensional FE model, which takes the material parameters obtained from the uniaxial tests as initial values. The effect of friction is included here as well. The applied friction coefficients $\mu = 1.05$ for the stiff and $\mu = 1.46$ for the soft fender material are obtained from separate friction tests conducted on individual material samples. It should be noted, however, that these coefficients may depend on velocity or temperature, for instance. In operating conditions, the friction coefficient may also vary with the local properties of the contact surface, which is often corroded or covered by maritime vegetation. Nevertheless, the coefficients of friction are assumed to be constant due the lack of more detailed information. The stiff and soft fender are discretized by 4,025 and 5,025 underintegrated trilinear hexahedral SOLID185 elements, available in ANSYS [3]. For capturing the contact, we use an augmented Lagrangian method for constrained minimization. In accordance with the experimental setup, a quasi-static analysis is undertaken, and we apply a fixed displacement $\bar{d}_z = 0.03\,\mathrm{m}$ (stiff fender) or $\bar{d}_z = 0.06\,\mathrm{m}$ (soft fender) at the bottom, ramped up linearly over 100 load steps. Figure 7.18 illustrates the strongly deformed fenders subjected to the displacement-controlled compression loading. Table 7.4 holds the parameters for the material models obtained by means of the parameter identification procedure, which will also be used in the following.

In view of the partitioned analysis of the entire berthing maneuver in the next section, where the fluid and the structural problem are solved iteratively multiple times per time increment, it seems reasonable to replace the sophisticated FE model by a reduced structural model, which takes only a fraction to compute. As sketched in Figure 7.19, the simplified model consists of two nonlinear springs based on the Neo-Hooke or Mooney-Rivlin material model, the parameters of which are listed in Table 7.5. Due to the fact that the geometrical stiffness of the reduced structural model differs significantly from the full FE model, the material parameters are of course completely different. With regard

Table 7.4: Material parameters for the FE fender model.

Fender	Material model	Material parameters
A	Neo-Hooke	$C_{10} = 1.51 \times 10^6$
	Mooney-Rivlin	$C_{10} = 1.46 \times 10^6$, $C_{01} = 3.25 \times 10^4$
B	Neo-Hooke	$C_{10} = 2.20 \times 10^5$
	Mooney-Rivlin	$C_{10} = 2.01 \times 10^4$, $C_{01} = 1.87 \times 10^5$

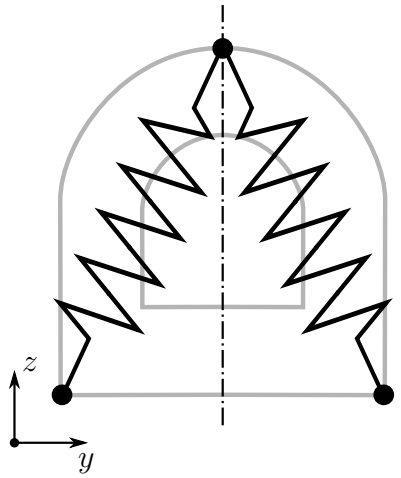


Figure 7.19: Reduced structural fender model.

Table 7.5: Material parameters for reduced structural fender model.

Fender	Material model	Material parameters
A	Neo-Hooke	$C_{10} = 4.04 \times 10^3$
	Mooney-Rivlin	$C_{10} = 3.88 \times 10^3$, $C_{01} = 1.24 \times 10^2$
B	Neo-Hooke	$C_{10} = 1.90 \times 10^2$
	Mooney-Rivlin	$C_{10} = 3.92 \times 10^1$, $C_{01} = 2.02 \times 10^1$

to the nonlinear displacement constraint originating from the presence of the contact, a node-to-segment contact model is chosen.

7.2.5 Berthing Maneuver of Crew Transfer Vessel

Having analyzed all the subproblems arising in the berthing maneuver individually, we next proceed to the analysis of the entire coupled problem. Following a partitioned solution approach, the subproblem simulations are integrated into a three-field simulation, where, within a time increment, the displacement from the rigid body is first applied to the elastic fender, which solves for the reaction force at the attachment point to the ship's bow. The reaction force is then exerted on the rigid body, and a new position and orientation of the rigid body are calculated. In another nested iteration, the rigid body displacement is passed over the fluid solver, and the resulting traction is in turn again applied on the rigid body. Once this embedded procedure is converged, another outer implicit iteration is performed. The whole procedure is continued until all involved fields are equilibrated with each other.

In the following, we consider two different sea states. In the first case, the catamaran vessel already investigated in Section 7.2.2 and 7.2.3 is subjected to regular waves with angular frequency $\omega_1 = 4\,{}^{\mathrm{rad}}/_{\mathrm{s}}$ and amplitude $\zeta_1 = 0.03\,\mathrm{m}$. The propulsion system is represented by a single constant follower force $F_1 = 127\,\mathrm{N}$ applied at the stern of the ship. Here, the stiff fender serves to establish the contact to the cylindrical offshore structure. For the second case, we change the angular frequency to $\omega_2 = 4.5\,{}^{\mathrm{rad}}/_{\mathrm{s}}$ and choose a slightly

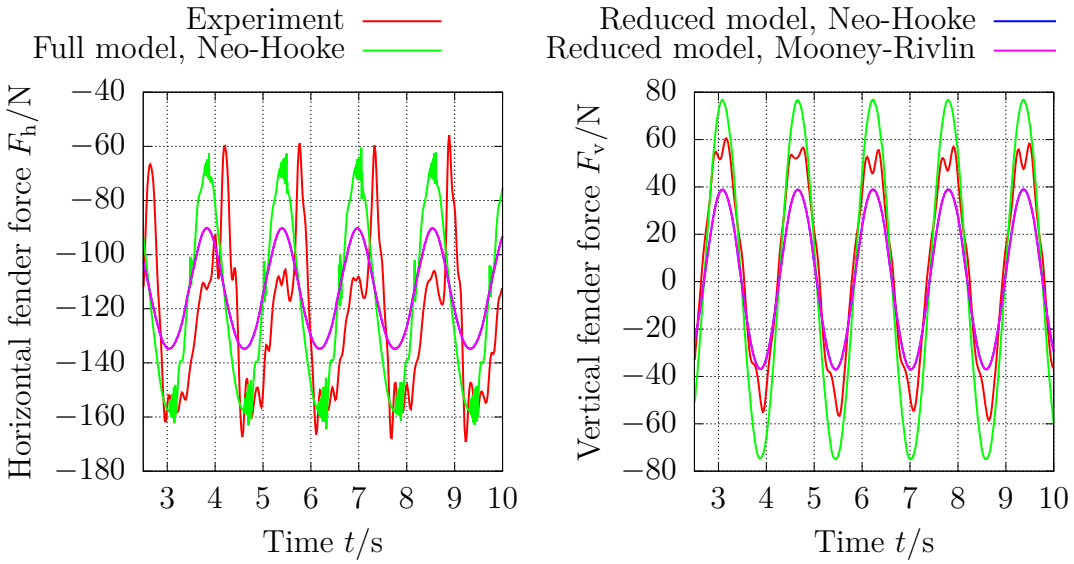


Figure 7.20: Horizontal and vertical fender force F_h and F_v for an angular frequency $\omega_1 = 4\,\mathrm{rad/s}$, wave height $\zeta_1 = 0.03\,\mathrm{m}$, and thrust $P_1 = 127\,\mathrm{N}$.

higher wave amplitude $\zeta_2 = 0.04\,\mathrm{m}$. The propulsion force is reduced to $F_2 = 61\,\mathrm{N}$, and the stiff fender is replaced by the soft fender.

Since the discretization schemes and spatial resolutions for the individual subproblems have been found to perform well, identical settings are also applied in the analysis of the coupled problem. For the fender involving the contact problem, we use the Newmark scheme with the parameters $\beta = 0.2525$ and $\gamma = 0.5050$. In order to investigate the capability of the reduced structural fender model resembling the behavior of the full continuum model and eventually the real fender, each of the simulations is carried out using both fender models. For each of the subproblems, we apply an identical time step size $\Delta t = 10^{-2}\pi/\omega$, corresponding to 200 time increments per wave period.

To begin with, let us discuss the results for the first scenario. Figure 7.20 graphs the horizontal and vertical fender reaction force F_h and F_v over time t. As to be expected, all fender models are able to recover the frequency of the force components measured in the experiments. Notable deviations can, however, be observed regarding the force amplitude and the variation of the force over time. Although the full fender model captures the amplitude of the horizontal force component F_h quite accurately, it does not resemble the particular dynamic behavior of this force component. In contrast, the full model produces a higher vertical force F_v than measured in the experiments, but its curve shape is a better match than that for the horizontal force component. In general, the results produced by the full fender model based on the Neo-Hooke material model are in acceptable agreement to the experimental measurements. Force frequency and amplitude, which are certainly of most interest, match reasonably well. Regarding the reduced structural model, it is interesting to note that the Neo-Hooke and the Mooney-Rivlin model yield almost identical results. For the horizontal force component, a significant deviation in amplitude to the experiments is undeniable. Yet, the vertical force component, although still underestimated, exhibits a much better accordance already. Still, the results obtained using the reduced structural model seem to indicate that a simplified model, possibly enhanced by further spring or damper elements, is, in general, capable of resembling the

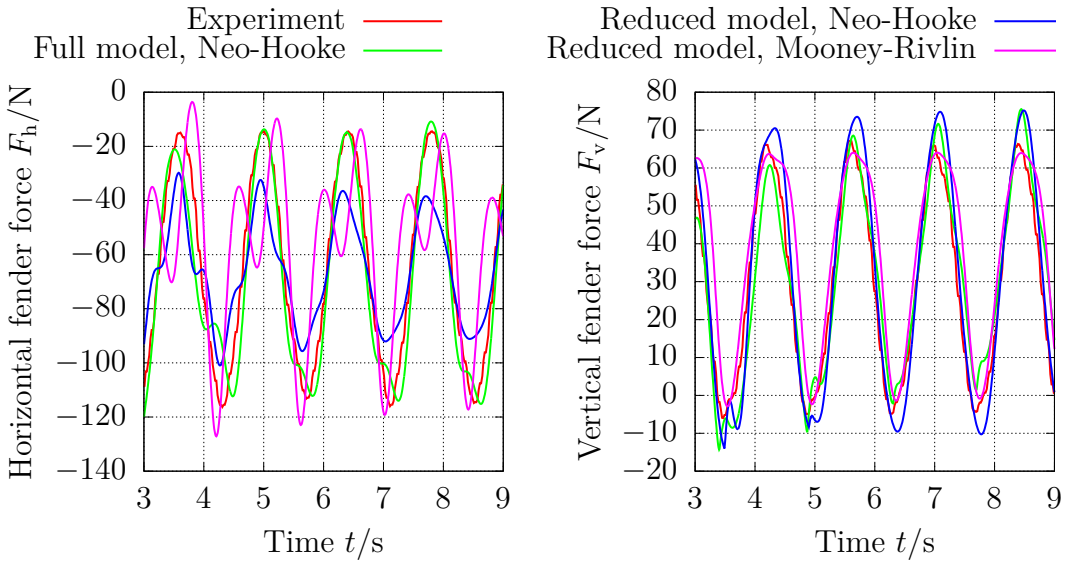


Figure 7.21: Horizontal and vertical fender force F_h and F_v for an angular frequency $\omega_2 = 4.5\,^{\mathrm{rad}}/_{\mathrm{s}}$, wave height $\zeta_2 = 0.04\,\mathrm{m}$ and thrust $P_2 = 61\,\mathrm{N}$.

structural fender response.

Figure 7.21 graphs the horizontal and vertical reaction force F_h and F_v for the second sea state. Apparently, the reaction forces obtained by means of the full fender model incorporating the Neo-Hooke material model are in very good agreement with the experimental measurements. The reduced structural model based on the Neo-Hooke model again underestimates the horizontal force component, whereas the vertical force component is in good accordance with the experiments. The horizontal reaction force produced by the reduced structural model shows a deviating dynamic behavior as observed in the experiments, but exhibits an almost identical amplitude. Similar to the other models, the vertical force component is also recovered by this model.

Finally, Figure 7.22 depicts a snapshot of the deformed fenders in the different sea states at a particular instant of time. Particularly large deformations can be observed especially for the soft fender.

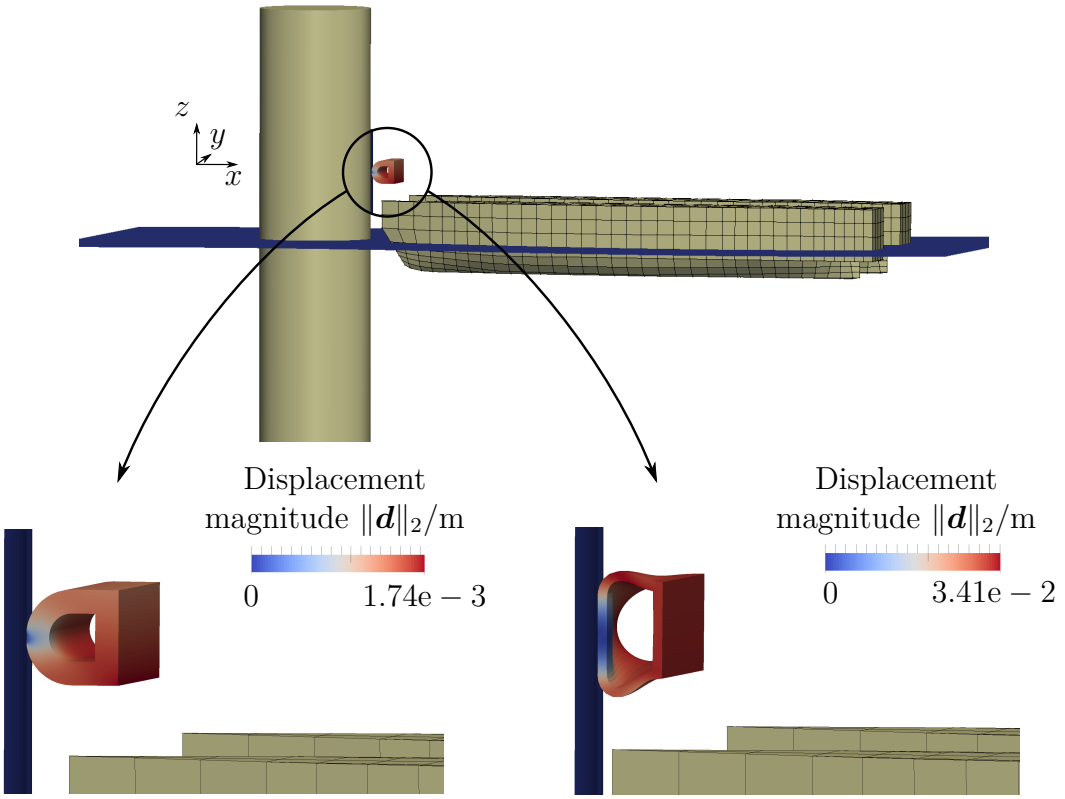


Figure 7.22: Deformed fenders during the berthing operation. The left figure shows the stiff fender at time $t = 6.3\,\mathrm{s}$, whereas the right figure represents the soft fender at time $t = 5.82\,\mathrm{s}$.

8 Conclusions and Outlook

In the present thesis, the partitioned solution approach was successfully applied to solve highly complex strongly-coupled multifield problems. In particular, it was demonstrated that the proposed solution procedure is very well suited for the numerical treatment of strongly-coupled FSI problems arising in the context of maritime applications. Major advantages of the partitioned approach include the possibility to use different spatial and temporal discretizations as well as different dedicated and existing solvers for each of the subproblems, which renders the concept not only very flexible but also efficient.

Recapitulating the main contents of this work, we first reviewed the governing equations of the fluid and the structural subproblem, as well as the common spatial and temporal discretization schemes for their numerical treatment.

Following this, we presented a generic partitioned procedure for the solution of weakly- and strongly-coupled multifield problems. Based upon the idea of dividing the entire computational domain into several separate subdomains, a partitioned solution scheme allows to use different numerical schemes for the discretization of space and time in each of the subdomains. Moreover, it enables the reuse of specialized and efficient solvers that are already available. Subsequently, the essential building blocks of the partitioned solution procedure were discussed in detail. In order to generate a reasonable initial solution at the beginning of a time increment, numerous predictor schemes were proposed. For the sake of comparison, all predictor schemes were applied to a simple benchmark problem based on a three-degree-of-freedom system. It was shown that predictors such as Taylor series-based or adaptive predictors, which take the physics behind the problem under consideration into account, have great potential to generate an initial solution close to the final converged solution in that time increment. This can be expected to stabilize and accelerate the coupled solution process. Regarding the interpolation of the relevant field quantities between the possibly non-conforming subproblem discretizations, various mesh-independent and mesh-based interpolation techniques were investigated. Further, the interpolation schemes were thoroughly tested for several benchmark problems and compared to each other with respect to accuracy and performance. It was demonstrated that mesh-based interpolation schemes usually provide the highest interpolation accuracy – and that they also function efficiently if appropriate search strategies are used to associate the given query points to the corresponding mesh entities. In order to ensure the stability of the solution process and to reduce the required number of implicit iterations within a time increment until convergence is achieved, the use of a convergence acceleration scheme is advisable. A broad range of different convergence acceleration schemes based on vector sequence acceleration or quasi-Newton methods were discussed. In a comparison of these schemes for a simple benchmark problem, the line extrapolation method, the Broyden method, and, in particular, the quasi-Newton least squares procedure were found to perform particularly well. Although it is difficult to generalize the findings, the presented numerical study provides a useful hint on which schemes may prove useful and efficient also for other multifield problems.

In the following section, we introduced the dedicated C++ software library *comana* developed at the Institute for Ship Structural Design and Analysis at the Hamburg University of Technology. Based on a master/slave communication concept, the software permits to easily integrate an arbitrary number of subproblem solvers in a coupled solution strategy. With the proposed communication concept, all subproblem solvers can be addressed in the same manner, independent of whether they operate in serial or shared- or distributed-memory parallelized mode. This way, not only FSI but also other multifield problems can be solved effectively – and specialized subfield solvers that are already available can, quickly and with minimally invasive modifications, be prepared for the integration into a partitioned solution strategy. For the implementation of a customized coupling algorithm in a dedicated C++ program, several modular and well-tested algorithmic building blocks are provided along with *comana*. Each component of the coupling algorithm can thus be carefully selected and used in a coupling strategy suiting the particular problem under consideration.

In order to demonstrate the effectiveness and efficiency of the proposed generic partitioned solution procedure, we computed numerous benchmarks involving FSI, but also other multifield problems such as electro-thermo-mechanically coupled problems. Being in reasonable agreement with the reference solutions available from the literature or obtained by analytical or other numerical methods, the numerical results for the considered benchmark problems confirmed that the partitioned solution strategy is a suitable means for the solution of general strongly-coupled multifield problems.

Following the numerical benchmarks, the partitioned solution procedure was also employed to investigate more advanced applications from the maritime industry. In a first scenario, the FSI of a floating OWT was analyzed in order to examine the aero- and hydrodynamic behavior of the structure under different environmental conditions. The motion behavior of the floating OWT, investigated in a previous study, may now serve to estimate the inertia forces acting on the tower structure and the platform.

As a second application from maritime technology, the berthing maneuver of a crew transfer vessel to an OWT was studied. Once again drawing on a partitioned solution approach, this sophisticated operation was successfully analyzed numerically. We considered different sea states, their influence on the catamaran motion, as well as the contact forces – and the results can now be used to assess the safety of the service personnel during disembarkation or to provide useful data for the development of future transfer concepts.

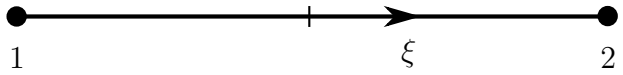
Summarizing the above, the methodological and numerical studies undertaken in the present thesis underline that the partitioned solution approach is a suitable means to solve sophisticated strongly-coupled multifield problems. Also, this solution strategy is clearly quite advanced regarding stability and performance. In future research, a particular focus should therefore be placed on the enhancement of the subproblem models. In view of the fact that each of the subproblems needs to be solved several times per time increment, the computational effort spent on the subfield solution has a large impact on the overall computational time required for the coupled problem. Nowadays, multifield problems involving sophisticated subfield problems are still challenging to solve in acceptable time, and they may in fact occasionally even prove prohibitively expensive to solve for some real-world industrial applications. Apart from the use of parallelized subproblem solvers, which has already become feasible with the proposed master/slave communication concept implemented in *comana*, the simultaneous solution of the subproblems as opposed to a sequential or staggered solution may be a promising measure to reduce the time required

to compute the coupled problem. However, parallel coupling schemes are prone to stability issues or to poor convergence rates, and further research must be invested to achieve considerable savings in computational cost as compared to the sequential schemes.

In order to verify the numerical results for the problems studied in this work, it would be necessary to carry out extensive spatial and temporal convergence studies, which were mostly omitted due to the high computational effort involved. In addition, a validation of the numerical findings by experiments is essential to assess the applicability of the numerical models for the analysis of the considered technical systems and processes, and to underpin the advantages of numerical simulations for a fast and reliable investigation of physical phenomena. This would serve to reduce the need for expensive experiments and to accelerate product or process development cycles.

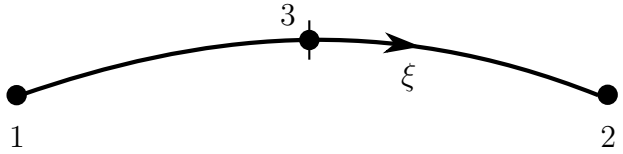
A Shape Functions

A.1 Linear line element



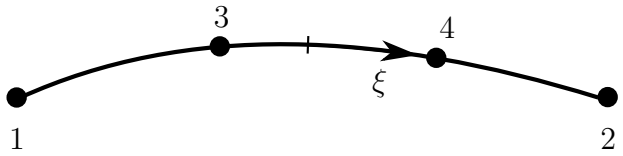
$$N_1(\xi) = \frac{1}{2}(1-\xi) \qquad N_2(\xi) = \frac{1}{2}(1+\xi)$$

A.2 Quadratic line element



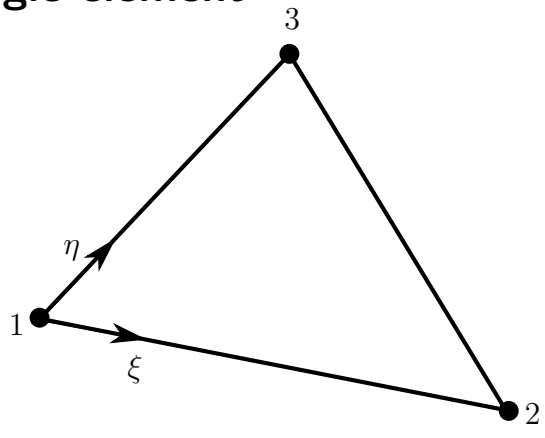
$$N_1(\xi) = \frac{1}{2}\left(-\xi+\xi^2\right) \qquad N_2(\xi) = \frac{1}{2}\left(\xi+\xi^2\right) \qquad N_3(\xi) = 1-\xi^2$$

A.3 Cubic line element



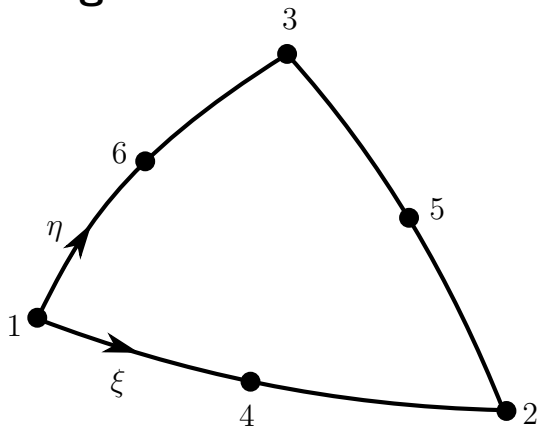
$$N_1(\xi) = \frac{1}{16}\left(-1+\xi+9\xi^2-9\xi^3\right) \qquad N_2(\xi) = \frac{1}{16}\left(-1-\xi+9\xi^2+9\xi^3\right)$$
$$N_3(\xi) = \frac{1}{16}\left(9-27\xi-9\xi^2+27\xi^3\right) \qquad N_4(\xi) = \frac{1}{16}\left(9+27\xi-9\xi^2-27\xi^3\right)$$

A.4 Linear triangle element



$$N_1(\xi,\eta) = 1-\xi-\eta \qquad N_2(\xi,\eta) = \xi \qquad N_3(\xi,\eta) = \eta$$

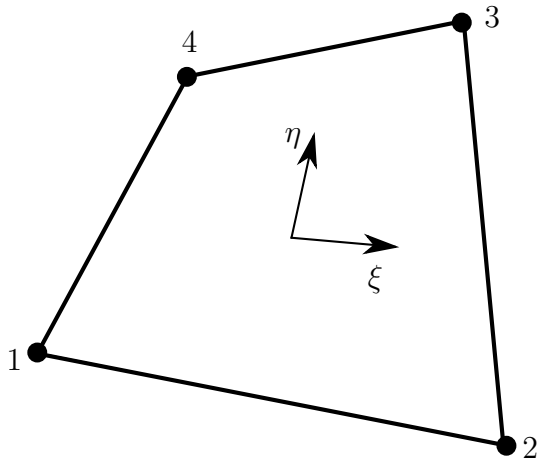
A.5 Quadratic triangle element



$$c_0 = 1-\xi-\eta$$

$$\begin{aligned} N_1(\xi,\eta) &= -c_0(1-2c_0) & N_2(\xi,\eta) &= -\xi(1-2\xi) \\ N_3(\xi,\eta) &= -\eta(1-2\eta) & N_4(\xi,\eta) &= 4\xi c_0 \\ N_5(\xi,\eta) &= 4\xi\eta & N_6(\xi,\eta) &= 4\eta c_0 \end{aligned}$$

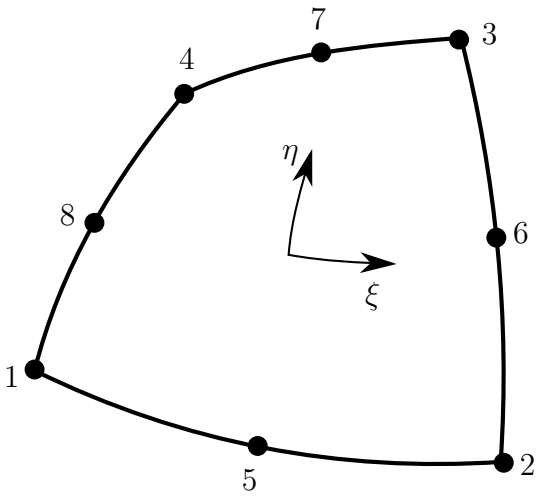
A.6 Linear quadrilateral element



$$\begin{aligned} \xi_- &= 1-\xi & \xi_+ &= 1+\xi \\ \eta_- &= 1-\eta & \eta_+ &= 1+\eta \end{aligned}$$

$$N_1(\xi,\eta) = \frac{1}{4}\xi_-\eta_- \qquad N_2(\xi,\eta) = \frac{1}{4}\xi_+\eta_-$$
$$N_3(\xi,\eta) = \frac{1}{4}\xi_+\eta_+ \qquad N_4(\xi,\eta) = \frac{1}{4}\xi_-\eta_+$$

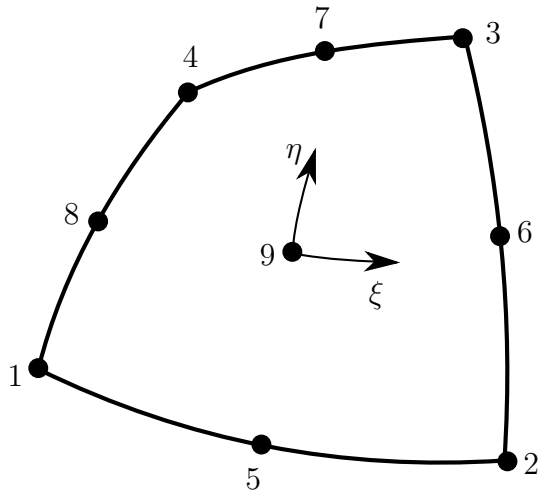
A.7 Serendipity quadrilateral element



$$\xi_- = 1-\xi \qquad \xi_+ = 1+\xi \qquad \tilde{\xi} = 1-\xi^2$$
$$\eta_- = 1-\eta \qquad \eta_+ = 1+\eta \qquad \tilde{\eta} = 1-\eta^2$$

$$N_1(\xi,\eta) = \frac{1}{4}\xi_-\eta_-(-1-\xi-\eta) \qquad N_2(\xi,\eta) = \frac{1}{4}\xi_+\eta_-(-1+\xi-\eta)$$
$$N_3(\xi,\eta) = \frac{1}{4}\xi_+\eta_+(-1+\xi+\eta) \qquad N_4(\xi,\eta) = \frac{1}{4}\xi_-\eta_+(-1-\xi+\eta)$$
$$N_5(\xi,\eta) = \frac{1}{2}\tilde{\xi}\eta_- \qquad N_6(\xi,\eta) = \frac{1}{2}\tilde{\eta}\xi_+$$
$$N_7(\xi,\eta) = \frac{1}{2}\tilde{\xi}\eta_+ \qquad N_8(\xi,\eta) = \frac{1}{2}\tilde{\eta}\xi_-$$

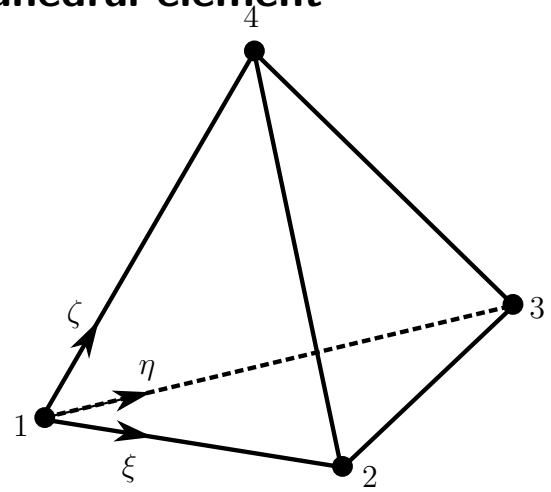
A.8 Quadratic quadrilateral element



$$\xi_- = \frac{1}{2}\xi(\xi-1) \qquad \xi_\pm = -(\xi+1)(\xi-1) \qquad \xi_+ = \frac{1}{2}\xi(\xi+1)$$
$$\eta_- = \frac{1}{2}\eta(\eta-1) \qquad \eta_\pm = -(\eta+1)(\eta-1) \qquad \eta_+ = \frac{1}{2}\eta(\eta+1)$$
$$\zeta_- = \frac{1}{2}\zeta(\eta-1) \qquad \zeta_\pm = -(\zeta+1)(\zeta-1) \qquad \zeta_+ = \frac{1}{2}\zeta(\zeta+1)$$

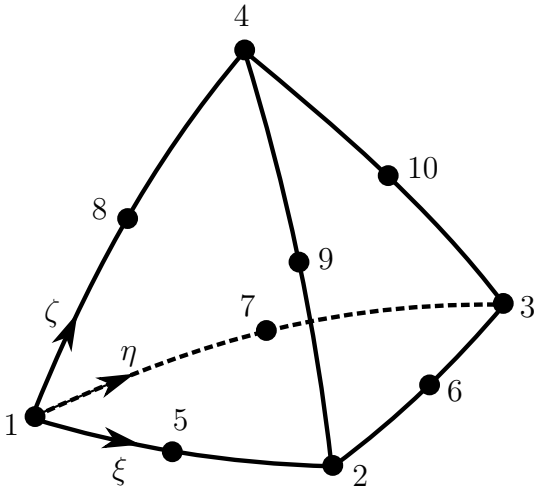
$$N_1(\xi,\eta) = \xi_-\eta_- \qquad N_2(\xi,\eta) = \xi_+\eta_-$$
$$N_3(\xi,\eta) = \xi_+\eta_+ \qquad N_4(\xi,\eta) = \xi_-\eta_+$$
$$N_5(\xi,\eta) = \xi_\pm\eta_- \qquad N_6(\xi,\eta) = \xi_+\eta_\pm$$
$$N_7(\xi,\eta) = \xi_\pm\eta_+ \qquad N_8(\xi,\eta) = \xi_-\eta_\pm$$
$$N_9(\xi,\eta) = \xi_\pm\eta_\pm$$

A.9 Linear tetrahedral element



$$N_1(\xi,\eta,\zeta) = 1-\xi-\eta-\zeta \qquad N_2(\xi,\eta,\zeta) = \xi$$
$$N_3(\xi,\eta,\zeta) = \eta \qquad N_4(\xi,\eta,\zeta) = \zeta$$

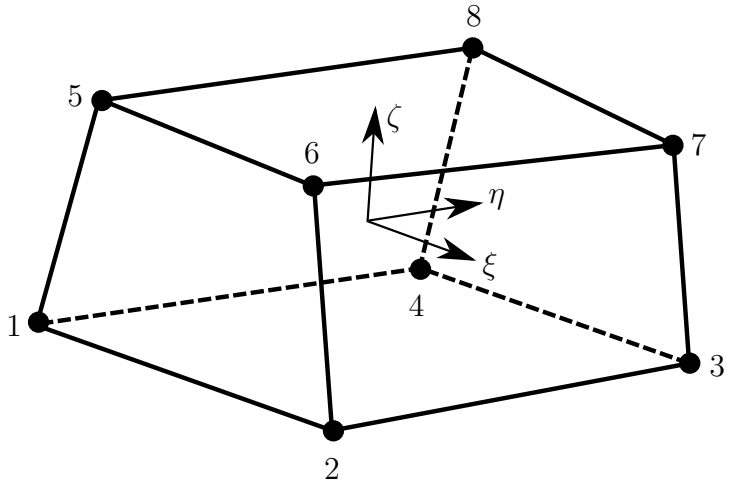
A.10 Quadratic tetrahedral element



$$c_0 = 1 - \xi - \eta - \zeta$$

$$\begin{aligned}
N_1(\xi,\eta,\zeta) &= (2c_0 - 1)c_0 & N_2(\xi,\eta,\zeta) &= (2\xi - 1)\xi \\
N_3(\xi,\eta,\zeta) &= (2\eta - 1)\eta & N_4(\xi,\eta,\zeta) &= (2\zeta - 1)\zeta \\
N_5(\xi,\eta,\zeta) &= 4\xi c_0 & N_6(\xi,\eta,\zeta) &= 4\xi\eta \\
N_7(\xi,\eta,\zeta) &= 4\eta c_0 & N_8(\xi,\eta,\zeta) &= 4\xi\zeta \\
N_9(\xi,\eta,\zeta) &= 4\eta\zeta & N_{10}(\xi,\eta,\zeta) &= 4\zeta c_0
\end{aligned}$$

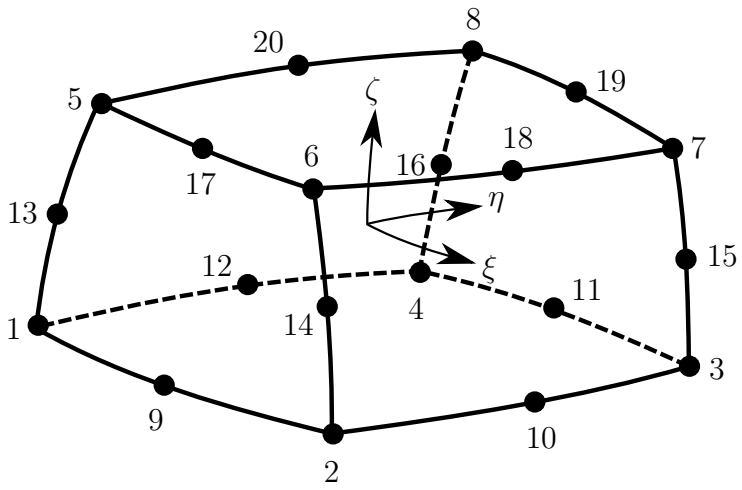
A.11 Linear hexahedral element



$$\begin{aligned}
\xi_- &= 1 - \xi & \xi_+ &= 1 + \xi \\
\eta_- &= 1 - \eta & \eta_+ &= 1 + \eta \\
\zeta_- &= 1 - \zeta & \zeta_+ &= 1 + \zeta
\end{aligned}$$

$$N_1(\xi,\eta,\zeta) = \frac{1}{8}\xi_-\eta_-\zeta_- \qquad N_2(\xi,\eta,\zeta) = \frac{1}{8}\xi_+\eta_-\zeta_-$$
$$N_3(\xi,\eta,\zeta) = \frac{1}{8}\xi_+\eta_+\zeta_- \qquad N_4(\xi,\eta,\zeta) = \frac{1}{8}\xi_-\eta_+\zeta_-$$
$$N_5(\xi,\eta,\zeta) = \frac{1}{8}\xi_-\eta_-\zeta_+ \qquad N_6(\xi,\eta,\zeta) = \frac{1}{8}\xi_+\eta_-\zeta_+$$
$$N_7(\xi,\eta,\zeta) = \frac{1}{8}\xi_+\eta_+\zeta_+ \qquad N_8(\xi,\eta,\zeta) = \frac{1}{8}\xi_-\eta_+\zeta_+$$

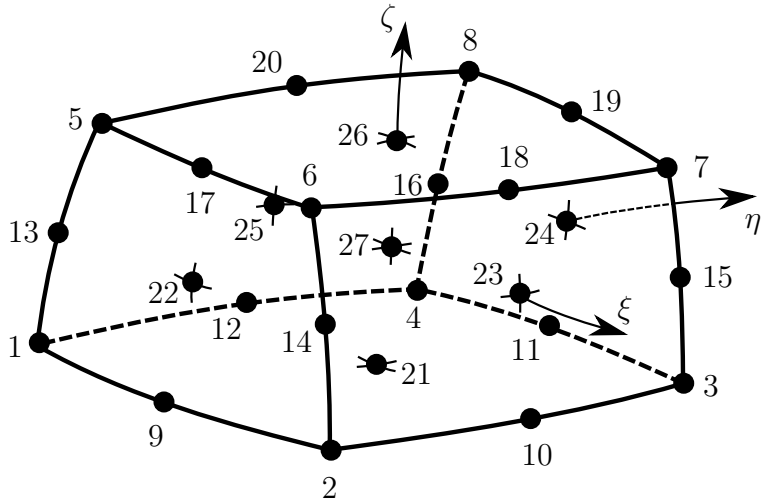
A.12 Serendipity hexahedral element



$$\xi_- = 1-\xi \qquad \xi_+ = 1+\xi \qquad \tilde{\xi} = 1-\xi^2$$
$$\eta_- = 1-\eta \qquad \eta_+ = 1+\eta \qquad \tilde{\eta} = 1-\eta^2$$
$$\zeta_- = 1-\zeta \qquad \zeta_+ = 1+\zeta \qquad \tilde{\zeta} = 1-\zeta^2$$

$$
\begin{aligned}
N_1(\xi,\eta,\zeta) &= \frac{1}{8}\xi_-\eta_-\zeta_-(-\xi-\eta-\zeta-2) & N_2(\xi,\eta,\zeta) &= \frac{1}{8}\eta_+\eta_-\zeta_-(\ \ \xi-\eta-\zeta-2)\\
N_3(\xi,\eta,\zeta) &= \frac{1}{8}\eta_+\eta_+\zeta_-(\ \ \xi+\eta-\zeta-2) & N_4(\xi,\eta,\zeta) &= \frac{1}{8}\xi_-\eta_+\zeta_-(-\xi+\eta-\zeta-2)\\
N_5(\xi,\eta,\zeta) &= \frac{1}{8}\xi_-\eta_-\zeta_+(-\xi-\eta+\zeta-2) & N_6(\xi,\eta,\zeta) &= \frac{1}{8}\eta_+\eta_-\zeta_+(\ \ \xi-\eta+\zeta-2)\\
N_7(\xi,\eta,\zeta) &= \frac{1}{8}\eta_+\eta_+\zeta_+(\ \ \xi+\eta+\zeta-2) & N_8(\xi,\eta,\zeta) &= \frac{1}{8}\xi_-\eta_+\zeta_+(-\xi+\eta+\zeta-2)\\
N_9(\xi,\eta,\zeta) &= \frac{1}{4}\tilde{\xi}\eta_-\zeta_- & N_{10}(\xi,\eta,\zeta) &= \frac{1}{4}\eta_+\tilde{\eta}\zeta_-\\
N_{11}(\xi,\eta,\zeta) &= \frac{1}{4}\tilde{\xi}\eta_+\zeta_- & N_{12}(\xi,\eta,\zeta) &= \frac{1}{4}\xi_-\tilde{\eta}\zeta_-\\
N_{13}(\xi,\eta,\zeta) &= \frac{1}{4}\xi_-\eta_-\tilde{\zeta} & N_{14}(\xi,\eta,\zeta) &= \frac{1}{4}\eta_+\eta_-\tilde{\zeta}\\
N_{15}(\xi,\eta,\zeta) &= \frac{1}{4}\eta_+\eta_+\tilde{\zeta} & N_{16}(\xi,\eta,\zeta) &= \frac{1}{4}\xi_-\eta_+\tilde{\zeta}\\
N_{17}(\xi,\eta,\zeta) &= \frac{1}{4}\tilde{\xi}\eta_-\zeta_+ & N_{18}(\xi,\eta,\zeta) &= \frac{1}{4}\eta_+\tilde{\eta}\zeta_+\\
N_{19}(\xi,\eta,\zeta) &= \frac{1}{4}\tilde{\xi}\eta_+\zeta_+ & N_{20}(\xi,\eta,\zeta) &= \frac{1}{4}\xi_-\tilde{\eta}\zeta_+
\end{aligned}
$$

A.13 Quadratic hexahedral element



$$
\begin{aligned}
\xi_- &= \frac{1}{2}\xi(\xi-1) & \xi_\pm &= -(\xi+1)(\xi-1) & \xi_+ &= \frac{1}{2}\xi(\xi+1)\\
\eta_- &= \frac{1}{2}\eta(\eta-1) & \eta_\pm &= -(\eta+1)(\eta-1) & \eta_+ &= \frac{1}{2}\eta(\eta+1)\\
\zeta_- &= \frac{1}{2}\zeta(\eta-1) & \zeta_\pm &= -(\zeta+1)(\zeta-1) & \zeta_+ &= \frac{1}{2}\zeta(\zeta+1)
\end{aligned}
$$

$$
\begin{array}{lll}
N_1(\xi,\eta,\zeta) = \xi_-\eta_-\zeta_- & N_2(\xi,\eta,\zeta) = \xi_+\eta_-\zeta_- & N_3(\xi,\eta,\zeta) = \xi_+\eta_+\zeta_- \\
N_4(\xi,\eta,\zeta) = \xi_-\eta_+\zeta_- & N_5(\xi,\eta,\zeta) = \xi_-\eta_-\zeta_+ & N_6(\xi,\eta,\zeta) = \xi_+\eta_-\zeta_+ \\
N_7(\xi,\eta,\zeta) = \xi_+\eta_+\zeta_+ & N_8(\xi,\eta,\zeta) = \xi_-\eta_+\zeta_+ & N_9(\xi,\eta,\zeta) = \xi_\pm\eta_-\zeta_- \\
N_{10}(\xi,\eta,\zeta) = \xi_+\eta_\pm\zeta_- & N_{11}(\xi,\eta,\zeta) = \xi_\pm\eta_+\zeta_- & N_{12}(\xi,\eta,\zeta) = \xi_-\eta_\pm\zeta_- \\
N_{13}(\xi,\eta,\zeta) = \xi_-\eta_-\zeta_\pm & N_{14}(\xi,\eta,\zeta) = \xi_+\eta_-\zeta_\pm & N_{15}(\xi,\eta,\zeta) = \xi_+\eta_+\zeta_\pm \\
N_{16}(\xi,\eta,\zeta) = \xi_-\eta_+\zeta_\pm & N_{17}(\xi,\eta,\zeta) = \xi_\pm\eta_-\zeta_+ & N_{18}(\xi,\eta,\zeta) = \xi_+\eta_\pm\zeta_+ \\
N_{19}(\xi,\eta,\zeta) = \xi_\pm\eta_+\zeta_+ & N_{20}(\xi,\eta,\zeta) = \xi_-\eta_\pm\zeta_+ & N_{21}(\xi,\eta,\zeta) = \xi_\pm\eta_\pm\zeta_- \\
N_{22}(\xi,\eta,\zeta) = \xi_\pm\eta_-\zeta_\pm & N_{23}(\xi,\eta,\zeta) = \xi_+\eta_\pm\zeta_\pm & N_{24}(\xi,\eta,\zeta) = \xi_\pm\eta_+\zeta_\pm \\
N_{25}(\xi,\eta,\zeta) = \xi_-\eta_\pm\zeta_\pm & N_{26}(\xi,\eta,\zeta) = \xi_\pm\eta_\pm\zeta_+ & N_{27}(\xi,\eta,\zeta) = \xi_\pm\eta_\pm\zeta_\pm
\end{array}
$$

Bibliography

[1] Y. Abbas and M. Madboulli. "Implementation of the panel method to the solution of flow around aircraft". In: *Proceedings of the 3rd International Workshop on Numerical Modelling in Aerospace Sciences*. Bucharest, Romania, 2015.

[2] A. Aitken. "On Bernoulli's numerical solution of algebraic equations". In: *Proceedings of the Royal Society of Edinburgh*. Vol. 46. 1926, pp. 289–305.

[3] ANSYS, Inc. *ANSYS Academic Research, Release 17.0*. Canonsburg, PA, 2017. URL: `http://ansys.com` (visited on 07/01/2017).

[4] ANSYS, Inc. *ANSYS Mechanical APDL Technology Demonstration Guide, Release 17.0*. Canonsburg, PA, 2017.

[5] D. Balzani et al. "Numerical modeling of fluid-structure interaction in arteries with anisotropic polyconvex hyperelastic and anisotropic viscoelastic material models at finite strains". In: *International Journal for Numerical Methods in Biomedical Engineering* 32.10 (2016).

[6] D. Baraff. *An introduction to physically based modeling: Rigid body simulation I – Unconstrained rigid body dynamics*. Carnegie Mellon University, Pittsburgh, PA, 1997.

[7] M. S. Bartlett. "An inverse matrix adjustment arising in discriminant analysis". In: *The Annals of Mathematical Statistics* 22.1 (1951), pp. 107–111.

[8] Y. Başar and D. Weichert. *Nonlinear continuum mechanics of solids: Fundamental mathematical and physical concepts*. Berlin, Germany et al.: Springer, 2000.

[9] K.-J. Bathe and G. A. Ledezma. "Benchmark problems for incompressible fluid flows with structural interactions". In: *Computers & Structures* 85.11-14 (2007), pp. 628–644.

[10] K.-J. Bathe, C. Nitikitpaiboon, and X. Wang. "A mixed displacement-based finite element formulation for acoustic fluid-structure interaction". In: *Computers & Structures* 56.2 (1995), pp. 225–237.

[11] K.-J. Bathe and H. Zhang. "A mesh adaptivity procedure for CFD and fluid-structure interactions". In: *Computers & Structures* 87.11-12 (2009), pp. 604–617.

[12] M. Bauer and M. Abdel-Maksoud. "A 3d potential based boundary element method for the modelling and simulation of marine propeller flows". In: *7th Vienna Conference on Mathematical Modelling*. Vienna, Austria, 2012.

[13] Y. Bazilevs, M.-C. Hsu, and M.A. Scott. "Isogeometric fluid-structure interaction analysis with emphasis on non-matching discretizations, and with application to wind turbines". In: *Computer Methods in Applied Mechanics and Engineering* 249–252 (2012), pp. 28–41.

[14] Y. Bazilevs et al. “3D simulation of wind turbine rotors at full scale. Part II: Fluid-structure interaction modeling with composite blades”. In: *International Journal for Numerical Methods in Fluids* 65 (2011), pp. 236–253.

[15] David M. Beazley. “SWIG: An easy to use tool for integrating scripting languages with C and C++”. In: *Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop*. Vol. 4. Monterey, CA, 1996, pp. 129–139.

[16] T. Belytschko and J. M. Kennedy. “Computer methods for subassembly simulation”. In: *Journal of Nuclear Engineering and Design* 49 (1978), pp. 17–38.

[17] T. Belytschko, J. M. Kennedy, and D. F. Schoeberle. “Quasi-Eulerian finite element formulation for fluid-structure interaction”. In: *Proceedings of Joint ASME/CSME Pressure Vessels and Piping Conference*. New York, NY, 1978, p. 13.

[18] J. L. Bentley. “Multidimensional binary search trees used for associative searching”. In: *Communications of the ACM* 18.9 (1975), pp. 509–517.

[19] P. Birken et al. “Fast solvers for thermal fluid structure interaction”. In: *Proceedings of the 5th Conference on Computational Methods in Marine Engineering*. Hamburg, Germany, 2013.

[20] J. Blazek. *Computational fluid dynamics: principles and applications*. Amsterdam, Netherlands et al.: Elsevier, 2005.

[21] J. Bonet and R. D. Wood. *Nonlinear continuum mechanics for finite element analysis*. Cambridge, UK et al.: Cambridge University Press, 2008.

[22] C. Brezinski. *Extrapolation methods: theory and practice*. Amsterdam, Netherlands et al.: North-Holland, 1991.

[23] C. Broyden. “A class of methods for solving nonlinear simultaneous equations”. In: *Mathematics of Computation* 19 (1965), pp. 577–593.

[24] M. Buhmann. *Radial basis functions: theory and implementations*. Cambridge, UK: Cambridge University Press, 2003.

[25] H.-J. Bungartz et al. “A plug-and-play coupling approach for parallel multi-field simulations”. In: *Computational Mechanics* 55.6 (2015), pp. 1119–1129.

[26] H.-J. Bungartz et al. “preCICE – A fully parallel library for multi-physics surface coupling”. In: *Computers & Fluids* (2016).

[27] G. F. Carey and S. W. Kim. “Lifting aerofoil calculation using the boundary element method”. In: *International Journal for Numerical Methods in Fluids* 3 (1983), pp. 481–492.

[28] J. R. Cebral and R. Loehner. “Conservative load projection and tracking for fluid-structure problems”. In: *AIAA Journal* 35.4 (1997), pp. 687–692.

[29] C. S. Chen, Y. C. Hon, and R. A. Schaback. *Scientific computing with radial basis functions*. (in preparation). 2012.

[30] J. Chung and G. M. Hulbert. “A time integration algorithm for structural dynamics with improved numerical dissipation: the generalized-α method”. In: *Journal of Applied Mechanics, Transactions of the ASME 1993* 60 (1993), pp. 371–375.

[31] T. J. Chung. *Computational fluid dynamics*. Cambridge, UK: Cambridge University Press, 2010.

[32] J. Cornthwaite. "Pressure Poisson method for the incompressible Navier-Stokes equations using Galerkin finite elements". PhD thesis. Statesboro, GA: Georgia Southern University, 2013.

[33] J. Cruse. "Floating wind turbine promises cost reductions". In: *HANSA International Maritime Journal* 153.9 (2016), p. 173.

[34] R. M. Cummings et al. "Applied computational aerodynamics". In: Cambridge, UK: Cambridge University Press, 2015. Chap. Panel methods, pp. 267–305.

[35] J. Degroote. "Development of algorithms for the partitioned simulation of strongly coupled fluid-structure interaction problems". PhD thesis. Ghent, Belgium: Ghent University, 2010.

[36] J. Degroote, K.-J. Bathe, and J. Vierendeels. "Performance of a new partitioned procedure versus a monolithic procedure in fluid-structure interaction". In: *Computers & Structures* 87 (2009), pp. 793–801.

[37] W. G. Dettmer. "Finite element modeling of fluid flow with moving free surfaces and interfaces including fluid-solid interaction". PhD thesis. Wales, UK: University of Wales, 2004.

[38] J. Donea, P. Fasoli-Stella, and S. Giuliani. "Lagrangian and Eulerian finite element techniques for transient fluid-structure interaction problems". In: *Transactions of the 4th International Conference on Structural Mechanics in Reactor Technology – Volume B: Thermal and Fluid/Structure Dynamics Analysis*. San Francisco, CA: North-Holland Publishing Company, 1977, pp. 1–12.

[39] J. Donea et al. "Arbitrary Lagrangian-Eulerian methods". In: *Encyclopedia of computational mechanics*. Ed. by E. Stein, R. de Borst, and T. J. R. Hughes. Chichester, UK et al.: John Wiley & Sons, 2004, pp. 413–437.

[40] A. Düster, H. Bröker, and E. Rank. "The *p*-version of the finite element method for three-dimensional curved thin walled structures". USenglish. In: *International Journal for Numerical Methods in Engineering* 52 (2001), pp. 673–703.

[41] A. Düster and S. Kollmannsberger. *AdhoC*^{4} *user's guide*. Chair for Computation in Engineering, Technische Universität München, Numerical Structural Analysis with Application in Ship Technology, Hamburg University of Technology. 2010.

[42] H. Edelsbrunner, D. G. Kirkpatrick, and R. Seidel. "On the shape of a set of points in the plane". In: *IEEE Transactions on Information Theory* 29.4 (1983), pp. 551–559.

[43] EDF R&D. *Code_Aster – An open source FEA software, Release 12.1*. Paris, France, 2014. URL: `http://code-aster.org` (visited on 07/01/2017).

[44] P. Erbts. "Partitioned solution strategies for electro-thermo-mechanical problems applied to the field assisted sintering technology". PhD thesis. Hamburg, Germany: Hamburg University of Technology, 2016.

[45] P. Erbts, S. Hartmann, and A. Düster. "A partitioned solution approach for electro-thermo-mechanical problems". In: *Archive of Applied Mechanics* 85.8 (2015), pp. 1075–1101.

[46] C. Ericson. *Real-time collision detection*. Ed. by T. Cox. San Francisco, CA: Elsevier, 2005.

[47] C. Farhat, P. Geuzaine, and G. Brown. "Application of a three-field nonlinear fluid-structure formulation to the prediction of the aeroelastic parameters of an F-16 fighter". In: *Computers & Fluids* 32 (2003), pp. 3–29.

[48] M. A. Fernández and M. Moubachir. "A Newton method using exact Jacobians for solving fluid-structure coupling". In: *Computers & Structures* 83 (2005), pp. 127–142.

[49] J. H. Ferziger and M. Perić. *Computational methods for fluid dynamics*. Berlin, Germany et al.: Springer, 2002.

[50] P. J. Flory. "Thermodynamic relations for high elastic materials". In: *Transaction of the Faraday Society* 57 (1961), pp. 829–838.

[51] L. Formaggia et al. "On the coupling of 3D and 1D navier-stokes equations for flow problems in compliant vessels". In: *Computer Methods in Applied Mechanics and Engineering* 191 (2001), pp. 561–582.

[52] C. Förster, W. A. Wall, and E. Ramm. "On the geometric conservation law in transient flow calculations on deforming domains". In: *International Journal for Numerical Methods in Fluids* 50 (2006), pp. 1369–1379.

[53] R. M. Franck and R. B. Lazarus. "Mixed Eulerian-Lagrangian method". In: *Methods in computational physics*. Ed. by B. Alder, S. Fernbach, and M. Rotenberg. Vol. 3. New York, NY: Academic Press, 1964, pp. 47–67.

[54] Free Software Foundation, Inc. *Using the GNU compiler collection: for GCC version 5.4.0*. Boston, MA, 2015. URL: `https://gcc.gnu.org` (visited on 07/01/2017).

[55] G. Gaël and J. Benoît. *Eigen v3*. 2010. URL: `http://eigen.tuxfamily.org` (visited on 07/01/2017).

[56] T. G. Gallinger. "Effiziente Algorithmen zur partitionierten Lösung stark gekoppelter Probleme der Fluid-Struktur-Wechselwirkung". PhD thesis. Munich, Germany: Technische Universität München, 2010.

[57] B. Gatzhammer, M. Mehl, and T. Neckel. "A coupling environment for partitioned multiphysics simulations applied to fluid-structure interaction scenarios". In: *Proceedings of the International Conference on Computational Science*. Amsterdam, Netherlands, 2010, pp. 681–689.

[58] S. Geller. "Ein explizites Modell für die Fluid-Struktur-Interaktion basierend auf LBM und p-FEM". PhD thesis. Braunschweig, Germany: Technische Universität Braunschweig, 2010.

[59] M. Glück et al. "Computation of fluid-structure interaction on lightweight structures". In: *Journal of Wind Engineering and Industrial Aerodynamics* 89 (2001), pp. 1351–1368.

[60] P. R. Graves-Morris. "Extrapolation methods for vector sequences". In: *Numerische Mathematik* 61.1 (1992), pp. 475–487.

[61] B. Gschaider et al. *foam-extend, Release 3.1*. 2014. URL: `https://sourceforge.net/projects/foam-extend` (visited on 07/01/2017).

[62] R. Haeltermann et al. “Improving the performance of the partitioned QN-ILS procedure for fluid-structure interaction problems: Filtering”. In: *Computers & Structures* 171 (2016), pp. 9–17.

[63] B. Hahn et al. “Die Grenzen des Wachstums sind noch nicht erreicht”. In: *Windindustrie in Deutschland* (June 2015).

[64] K. Hasselmann et al. “Measurements of wind-wave growth and swell decay during the Joint North Sea Wave Project (JONSWAP)”. In: *Ergänzungsheft zur Deutschen Hydrographischen Zeitschrift* Reihe A(8).12 (1973), p. 95.

[65] U. Heißerer. “High-order finite elements for material and geometric nonlinear finite strain problems”. PhD thesis. Munich, Germany: Technische Universität München, 2008.

[66] U. Heißerer, A. Düster, and E. Rank. “Follower loads for axisymmetric high order finite elements”. In: *Proceedings of Applied Mathematics and Mechanics*. Vol. 5. 2005, pp. 405–406.

[67] M. A. Heroux et al. “An overview of the Trilinos project”. In: *ACM Transactions on Mathematical Software* 31.3 (2005), pp. 397–423.

[68] H. M. Hilber, T. J. R. Hughes, and R. L. Taylor. “Improved numerical dissipation for time integration algorithms in structural dynamics”. In: *Earthquake Engineering & Structural Dynamics* 5 (1977), pp. 283–292.

[69] C. W. Hirt, A. A. Amsden, and J. L. Cook. “An arbitrary Lagrangian-Eulerian computing method for all flow speeds”. In: *Journal of Computational Physics* 14 (1974), pp. 227–253.

[70] G. A. Holzapfel. *Nonlinear solid mechanics: A continuum approach for engineering*. Chichester, UK et al.: John Wiley & Sons, 2000.

[71] J. Hron and S. Turek. “Proposal for numerical benchmarking of fluid-structure interaction between elastic object and laminar incompressible flow”. In: *Fluid-structure interaction, modelling, simulation and optimisation*. Ed. by H. J. Bungartz and M. Schäfer. Vol. 53. Berlin, Germany et al.: Springer, 2006, pp. 146–170.

[72] M.-C. Hsu and Y. Bazilevs. “Fluid-structure interaction modeling of wind turbines: simulating the full machine”. In: *Computational Mechanics* 50 (2012), pp. 821–833.

[73] B. Hübner, E. Walhorn, and D. Dinkler. “A monolithic approach to fluid-structure interaction using space-time finite elements”. In: *Computer Methods in Applied Mechanics and Engineering* 193 (2004), pp. 2087–2104.

[74] T. J. R. Hughes, W. K. Liu, and T. K. Zimmermann. “Lagrangian-Eulerian finite element formulation for incompressible viscous flows”. In: *Computer Methods in Applied Mechanics and Engineering* 29.3 (1981), pp. 329–349.

[75] K. Hutter and Y. Wang. *Fluid and thermodynamics – Volume 1: Basic fluid mechanics*. Cham, Switzerland: Springer International Publishing, 2016.

[76] R. A. Ibrahim. *Liquid sloshing dynamics: Theory and applications*. Cambridge, UK et al.: Cambridge University Press, 2005.

[77] S. R. Idelsohn et al. “Interaction between an elastic structure and free-surface flows: experimental versus numerical comparisons using the PFEM”. In: *Computational Mechanics* 43 (2008), pp. 125–132.

[78] K. Iguchi. “Convergence property of Aitken’s Δ^2-process and the applicable acceleration process”. In: *Journal of Information Processing* 7 (1984), pp. 22–30.

[79] International Organization for Standardization. *International standard ISO/IEC 14882:2003 Information technology – Programming languages – C++*. Standard. Geneva, Switzerland, 2003.

[80] International Organization for Standardization. *International standard ISO/IEC 14882:2011 Information technology – Programming languages – C++*. Standard. Geneva, Switzerland, 2011.

[81] International Organization for Standardization. *International standard ISO/IEC 14882:2014 Information technology – Programming languages – C++*. Standard. Geneva, Switzerland, 2014.

[82] International Organization for Standardization. *International standard working draft N4659 Information technology – Programming languages – C++*. Working Draft. Geneva, Switzerland, 2017.

[83] B. Irons and R. Tuck. “A version of the Aitken accelerator for computer implementation”. In: *International Journal for Numerical Methods in Engineering* 1 (1969), pp. 275–277.

[84] R. I. Issa. “Solution of the implicitly discretised fluid flow equations by operator-splitting”. In: *Journal of Computational Physics* 62 (1985), pp. 40–65.

[85] H. Jasak and Ž. Tuković. “Automatic mesh motion for the unstructured finite volume method”. In: *Transactions of FAMENA* 30.2 (2006), pp. 1–20.

[86] A. Jennings. “Accelerating the convergence of matrix iterative processes”. In: *Journal of the Institute of Mathematics and its Applications* 8 (1971), pp. 99–110.

[87] W. Johnson. *Rotorcraft aeromechanics*. Cambridge, UK: Cambridge University Press, 2013.

[88] W. Joppich and M. Kürschner. “MpCCI – A tool for the simulation of coupled applications”. In: *Concurrency and Computation: Practice and Experience* 18.2 (2006), pp. 183–192.

[89] N. Josuttis. *The C++ standard library: A tutorial and reference*. Upper Saddle River, NJ et al.: Addison-Wesley, 2012.

[90] D. Kamensky et al. “A variational immersed boundary framework for fluid-structure interaction: Isogeometric implementation and application to bioprosthetic heart valves”. In: *Computers Methods in Applied Mechanics and Engineering* (2014).

[91] J. Katz and A. Plotkin. *Low-speed aerodynamics*. Cambridge, UK: Cambridge University Press, 1991.

[92] C. T. Kelley. *Iterative methods for linear and nonlinear equations*. Raleigh, NC: SIAM, 1995.

[93] E. M. Knobbe. “Mesh movement governed by entropy production”. In: *Proceedings of the 13th International Meshing Roundtable*. Williamsburg, VA, 2004, pp. 265–276.

[94] S. Kollmannsberger. “ALE-type and fixed grid fluid-structure interaction involving the p-version of the Finite Element Method”. PhD thesis. München: Technische Universität München, 2010.

[95] A. Komech and A. Komech. *Principles of partial differential equations*. New York, NY: Springer, 2009.

[96] M. König, L. Radtke, and A. Düster. “A flexible C++ framework for the partitioned solution of strongly coupled multifield problems”. In: *Computers & Mathematics with Applications* 72.7 (2016), pp. 1764–1789.

[97] M. König et al. “Numerical investigation of the landing manoeuvre of a crew transfer vessel to an offshore wind turbine”. In: *Ships and Offshore Structures* 12.1 (2017), pp. 115–133.

[98] A. Konyukhov and K. Schweizerhof. *Computational contact mechanics: Geometrically exact theory for arbitrary shaped bodies*. Karlsruhe, Germany: Springer, 2013.

[99] R. Krause and E. Rank. “A fast algorithm for point-location in a finite element mesh”. In: *Computing* 57.1 (1996), pp. 49–62.

[100] D. Kuhl and M. A. Crisfield. “Energy-conserving and decaying algorithms in non-linear structural dynamics”. In: *International Journal for Numerical Methods in Engineering* 45 (1999), pp. 569–599.

[101] H. Lee et al. “Hydro-elastic analysis of marine propellers based on a BEM-FEM coupled FSI algorithm”. In: *International Journal of Naval Architecture and Ocean Engineering* 6 (2014), pp. 562–577.

[102] T. Lee. “Modelling time-dependent partial differential equations using a moving mesh approach based on conservation”. PhD thesis. Reading, UK: University of Reading, 2011.

[103] E. M. Lewandowski. *The dynamics of marine craft: maneuvering and seakeeping*. Singapore, Singapore: World Scientific Publishing, 2004.

[104] G. Link et al. “A 2D finite-element scheme for fluid-solid-acoustic interactions and its application to human phonation”. In: *Computer Methods in Applied Mechanics and Engineering* 198.41-44 (2009), pp. 3321–3334.

[105] C. Lothode et al. “Fluid-structure interaction analysis of an hydrofoil”. In: *Proceedings of the 5th International Conference on Computational Methods in Marine Enginerring*. Hamburg, Germany, 2013.

[106] A. J. MacLeod. “Acceleration of vector sequences by multidimensional Δ^2 methods”. In: *Communications in Applied Numerical Methods* 1 (1986), pp. 3–20.

[107] L. E. Malvern. *Introduction to the mechanics of a continuous medium*. Englewood Cliffs, NJ: Prentice-Hall, 1969.

[108] N. Maman and C. Farhat. “Matching fluid and structure meshes for aeroelastic computations: A parallel approach”. In: *Computers & Structures* 54.4 (1995), pp. 779–785.

[109] B. Markert. "Weak or strong – On coupled problems in continuum mechanics". Habilitation thesis. Stuttgart, Germany: University of Stuttgart, 2010.

[110] Bernd Markert. "A survey of selected coupled multifield problems in computational mechanics". In: *Journal of Coupled Systems and Multiscale Dynamics* 1.1 (2013), pp. 22–48.

[111] M. Mehl et al. "Parallel coupling numerics for partitioned fluid-structure interaction simulations". In: *Computers and Mathematics with Applications* (2015).

[112] S. Meyers. *Effective modern C++: 42 specific ways to improve your use of C++11 and C++14*. Beijing, China et al.: O'Reilly, 2015.

[113] S. Minami and S. Yoshimura. "Performance evaluation of nonlinear algorithms with line-search for partitioned coupling techniques for fluid-structure interactions". In: *International Journal for Numerical Methods in Fluids* 64 (2010), pp. 1129–1147.

[114] D. Mira et al. "Heat transfer effects on a fully premixed methane impinging flame". In: *Flow, Turbulence and Combustion* 97.1 (2016), pp. 339–361.

[115] D. P. Mok and W. A. Wall. "Partitioned analysis schemes for the transient interaction of incompressible flows and nonlinear flexible structures". In: *Trends in Computational Structural Mechanics* (2001). Ed. by W. A. Wall, K. U. Bletzinger, and K. Schweizerhof, pp. 689–698.

[116] M. Mooney. "A theory of large elastic deformation". In: *Journal of Applied Physics* 11 (1940), pp. 582–592.

[117] MSC Software Corporation. *MSC Marc 2016*. Santa Ana, CA, 2016. URL: `http://mscsoftware.com` (visited on 07/01/2017).

[118] J. Neugebauer, M. Abdel-Maksoud, and M. Braun. "Fluid-structure interaction of propellers". In: *Proceedings of the IUTAM Symposium on Fluid-Structure Interaction in Ocean Engineering*. Hamburg, Germany, 2007, pp. 191–204.

[119] N. M. Newmark. "A numerical method for structural dynamics". In: *Journal of Engineering Mechanics* 85 (1959), pp. 67–94.

[120] F. Nobile. "Numerical approximation of fluid-structure interaction problems with application to haemodynamics". PhD thesis. Laussane, Switzerland: École Polytechnique Fédérale de Laussane, 2001.

[121] W. Noh. "CEL: A time-dependent two-space-dimensional coupled Eulerian-Lagrangian code". In: *Methods in computational physics*. Ed. by B. Adler, S. Fernbach, and M. Trotenberg. Vol. 3. New York, NY: Academic Press, 1964.

[122] R. W. Ogden. *Non-linear elastic deformations*. Chichester, UK: Dover Publications, 1984.

[123] OpenCFD (ESI Group). *OpenFOAM – The open source CFD toolbox, Release 2.3.1*. Bracknell, UK, 2014. URL: `http://openfoam.com` (visited on 07/01/2017).

[124] T. van Opstal, E. van Brummelen, and G. van Zwieten. "A finite-element/boundary-element method for three-dimensional, large-displacement fluid-structure-interaction". In: *Computer Methods in Applied Mechanics and Engineering* 284 (2015), pp. 637–663.

[125] N. Osada. “Acceleration methods for vector sequences”. In: *Journal of Computational and Applied Mathematics* 38 (1991), pp. 361–371.

[126] B. Overland. *C++ for the impatient*. Upper Saddle River, NJ et al.: Addison-Wesley, 2013.

[127] N. Parolini and M. Lombardi. “Unsteady FSI simulation of downwind sails”. In: *Proceedings of the 5th International Conference on Computational Methods in Marine Engineering*. Hamburg, Germany, 2013.

[128] S. V. Patankar. *Numerical heat transfer and fluid flow*. Ed. by W. J. Minkowycz and E. M. Sparrow. New York, NY et al.: Hemisphere Publishing Corporation, 1980.

[129] W. J. J. Pierson and L. A. Moskowitz. “Proposed spectral form for fully developed wind seas based on the similarity theory of S. A. Kitaigorodskii”. In: *Journal of Geophysical Research* 69 (1964), pp. 5181–5190.

[130] S. Piperno. “Explicit/implicit fluid-structure staggered procedures with a structural predictor and fluid subcycling for 2D inviscid aeroelastic simulations”. In: *International Journal for Numerical Methods in Fluids* 25 (1997), pp. 1207–1226.

[131] S. Piperno and C. Farhat. “Partitioned procedures for the transient solution of coupled aroelastic problems – Part II: Energy transfer analysis and three dimensional applications”. In: *Computer Methods in Applied Mechanics and Engineering* 190 (2001), pp. 3147–3170.

[132] S. Piperno, C. Farhat, and B. Larrouturou. “Partitioned procedures for the transient solution of coupled aroelastic problems – Part I: Model problem, theory and two-dimensional application”. In: *Computer Methods in Applied Mechanics and Engineering* 124 (1995), pp. 79–112.

[133] A. Quarteroni. *Modeling the heart and the circulatory system*. Cham, Switzerland: Springer, 2015.

[134] L. Radtke, M. König, and A. Düster. “The influence of geometric imperfections in cardiovascular FSI simulations”. In: *Computers & Mathematics with Applications* 74 (2017), pp. 1675–1689.

[135] L. Radtke et al. “Convergence acceleration for partitioned simulations of the fluid-structure interaction in arteries”. In: *Computational Mechanics* (2016), pp. 1–20.

[136] J. N. Reddy and D. K. Gartling. *The finite element method in heat transfer and fluid dynamics*. Boca Raton, FL et al.: CRC Press, 2010.

[137] R.S. Rivlin. “Large elastic deformations isotropic materials. IV. Further developments of the general theory”. In: *Philosophical Transaction of the Royal Society of London* 241.835 (1948), pp. 379–397.

[138] A. Robertson et al. *Definition of the semisubmersible floating system for phase II of OC4*. Tech. rep. National Renewable Energy Laboratory, 2014.

[139] C. Runge. “Über empirische Funktionen und die Interpolation zwischen äquidistanten Ordinaten”. In: *Zeitschrift für Mathematik und Physik* 46 (1901), pp. 224–243.

[140] S. Sathe et al. “Fluid-structure interaction modeling of complex parachute designs with the space-time finite element techniques”. In: *Computers & Fluids* 36.1 (2007), pp. 127–135.

[141] F. Schäfer et al. “Fluid-structure-acoustic interaction of the flow past a thin flexible structure”. In: *AIAA Journal* 48.4 (2010), pp. 738–748.

[142] M. Schäfer. *Computational engineering – Introduction to numerical methods*. Berlin, Germany et al.: Springer, 2006.

[143] S. Schulte. “Modulare und hierarchische Simulation gekoppelter Probleme”. PhD thesis. Munich, Germany: Technische Universität München, 1998.

[144] D. Shepard. “A two-dimensional interpolation function for irregularly-spaced data”. In: *Proceedings of the 23rd ACM National Conference*. Las Vegas, NV, 1968, pp. 517–524.

[145] S. Sicklinger. “Stabilized co-simulation of coupled problems including fields and signals”. PhD thesis. Munich, Germany: Technische Universität München, 2014.

[146] S. Sicklinger and T. Wang. *Enhanced multiphysics interface research engine (EMPIRE)*. Munich, Germany, 2016. URL: `http://empire-multiphysics.com` (visited on 07/01/2017).

[147] J.C. Simo and L. Vu-Quoc. “A three-dimensional finite-strain rod model. Part II: Computational aspects”. In: *Computer Methods in Applied Mechanics and Engineering* 58 (1986), pp. 79–116.

[148] S. Slattery, P. Wilson, and R. Pawlowski. “The data transfer kit: A geometric rendezvous-based tool for multiphysics data transfer”. In: *International Conference on Mathematics & Computational Methods Applied to Nuclear Science & Engineering*. 2013, pp. 5–9.

[149] D. R. Smith. *An introduction to continuum mechanics – After Truesdell and Noll*. Dordrecht, Netherlands: Springer Science+Business Media, 1993.

[150] K. R. Stein et al. *Parallel computation of parachute fluid-structure interactions*. UMSI Research Report 97:54. Minneapolis, MN: University of Minnesota, 1997.

[151] K. R. Stein et al. “Simulation of parachute descent and maneuvers”. In: *Proceedings of the 5th International Conference on Computation of Shell and Spatial Structures*. Salzburg, Austria, 2005.

[152] J. Steindorf. “Partitionierte Verfahren für Probleme der Fluid-Struktur-Wechselwirkung”. PhD thesis. Braunschweig, Germany: Technische Universität Braunschweig, 2002.

[153] B. Stroustrup. *A tour of C++*. Upper Saddle River, NJ et al.: Addison-Wesley, 2014.

[154] B. Stroustrup. *C++11 – The new ISO C++ standard*. 2016. URL: `http://www.stroustrup.com/C++11FAQ.html` (visited on 04/19/2017).

[155] B. Stroustrup. *The C++ programming language*. Upper Saddle River, NJ et al.: Addison-Wesley, 2013.

[156] T. Tang. “Moving mesh methods for computational fluid dynamics”. In: *Contemporary Mathematics* 383 (2005), pp. 141–174.

[157] M. S. Tarafder, G. M. Khalil, and M. R. Islam. "Analysis of potential flow around two-dimensional hydrofoil by source based lower and higher order panel methods". In: *Journal of the Institution of Engineers* 71.2 (2009), pp. 13–21.

[158] M. S. Tarafder, G. K. Saha, and S. T. Mehedi. "Analysis of potential flow around 3-ddimensional hydrofoils by combined source and dipole based panel method". In: *Journal of Marine Science and Technology* 18.3 (2010), pp. 376–384.

[159] T. Tezduyar et al. "Modelling of fluid-structure interactions with the space-time finite elements: Arterial fluid mechanics". In: *International Journal for Numerical Methods in Fluids* 54.6-8 (2007), pp. 901–922.

[160] The MathWorks, Inc. *MATLAB, Release R2017a*. Natick, MA, 2017. URL: `https://www.mathworks.com/` (visited on 04/17/2017).

[161] C. Truesdell and W. Noll. *The nonlinear field theories of mechanics*. Berlin, Germany et al.: Springer, 2004.

[162] J. G. Trulio. *Theory and structure of the AFTON codes*. Tech. rep. Albuquerque, NM: Air Force Weapons Laboratory: Kirtland Air Force Base, 1966.

[163] B. W. Uekermann. "Partitioned fluid-structure interaction on massively parallel systems". PhD thesis. Munich, Germany: Technische Universität München, 2016.

[164] P. A. Ullrich and M. A. Taylor. "Arbitrary-order conservative and consistent remapping and a theory of linear maps: Part I". In: *Monthly Weather Review* 143.6 (2015), pp. 2419–2440.

[165] S. Valcke. "The OASIS3 coupler: A European climate modelling community software". In: *Geoscientific Model Development* 6 (2013), pp. 373–388.

[166] J. G. V. Vázquez. "Nonlinear analysis of orthotropic membrane and shell strucures including fluid-structure interaction". PhD thesis. Barcelona, Spain: Universitat Politecnica de Catalunya, 2007.

[167] L. Velho, J. Gomes, and L. H. de Figueiredo. *Implicit objects in computer graphics*. New York, NY: Springer, 2002.

[168] S. Wagner and A. Röttgermann. "A transonic panel method for helicopter flows". In: *Boundary element topics: proceedings of the final conference of the priority research programme "Boundary Element Methods" 1989–1995 of the German Research Foundation*. Ed. by Wolfgang L. Wendland. Berlin et al.: Springer, 1997, pp. 363–393.

[169] W. A. Wall. "Fluid-Struktur-Interaktion mit stabilisierten finiten Elementen". PhD thesis. Stuttgart, Germany: Institut für Baustatik der Universität Stuttgart, 1999.

[170] W. A. Wall and E. Ramm. "Fluid-structure interaction based upon a stabilized (ALE) finite element method". In: *Proceedings of the 4th World Congress on Computational Mechanics*. Ed. by S. Idelsohn, E. Oñate, and E. Dvorkin. Barcelona, Spain, 1998.

[171] W. Wambold, G. Bärwolff, and H. Schwandt. "Moving meshes to fit large deformations based on centroidal Voronoi tessellation (CVT)". In: *Proceedings of the 15th International Conference on Computational Science and Its Applications*. Banff, AB, Canada, 2015. Chap. Moving meshes to fit large deformations based on centroidal Voronoi tessellation (CVT), pp. 313–328.

[172] X. S. Wang. *Fundamentals of fluid-solid interactions: analytical and computational approaches*. Amsterdam, Netherlands et al.: Elsevier, 2008.

[173] H. Wendland. "Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree". In: *Advances in Computational Mathematics* 4.1 (1995), pp. 389–396.

[174] G. Wendt, P. Erbts, and A. Düster. "Partitioned coupling strategies for multi-physically coupled radiative heat transfer problems". In: *Journal of Computational Physics* 300 (2015), pp. 327–351.

[175] J. F. Wendt. *Computational fluid dynamics: An introduction*. Berlin, Germany et al.: Springer, 2009.

[176] P. Wesseling. *Principles of computational fluid dynamics*. Berlin, Germany et al.: Springer, 2001.

[177] D. J. Willis. "An unsteady, accelerated, high order panel method with vortex particle wakes". PhD thesis. Cambridge, MA: Massachusetts Institute of Technology, 2006.

[178] J. Wittenburg. *Dynamics of systems of rigid bodies*. Stuttgart, Germany: Teubner, 1977.

[179] K. Wolf and E. Brakkee. "Coupling fluids and structures codes on MPI". In: *Proceedings of the MPI Developer's Conference*. Notre Dame, IN, 1996.

[180] W. L. Wood, M. Bossak, and O. C. Zienkiewicz. "An alpha modification of Newmark's method". In: *International Journal for Numerical Methods in Engineering* 15 (1981), pp. 1562–1566.

[181] P. Wriggers. *Nonlinear finite element methods*. Berlin, Germany et al.: Springer, 2008.

[182] R. Wüchner. "Computational mechanics of form finding and fluid-structure interaction of membrane structures". PhD thesis. Munich, Germany: Technische Universität München, 2006.

[183] P. Wynn. "On a device for computing the $e_m(S_n)$ transformation". In: *Mathematical Tables and Other Aids to Computation* 10 (1956), pp. 91–96.

[184] T. Yamada and S. Yoshimura. "Line search partitioned approach for fluid-structure interaction analysis of flapping wing". In: *Computer Modeling in Engineering and Sciences* 24.1 (2008), pp. 51–60.

[185] Z. Yosibash et al. "Axisymmetric pressure boundary loading for finite deformation analysis using p-FEM". In: *Computer Methods in Applied Mechanics and Engineering* 196.7 (2007), pp. 1261–1277.

[186] O. C. Zienkiewicz and R. Löhner. "Accelerated relaxation or direct solution? Future prospects for FEM". In: *International Journal for Numerical Methods in Engineering* 21.1 (1985), pp. 1–11.

[187] O. C. Zienkiewicz and R. L. Taylor. *The finite element method for fluid dynamics*. Oxford, UK: Butterworth-Heinemann, 2014.