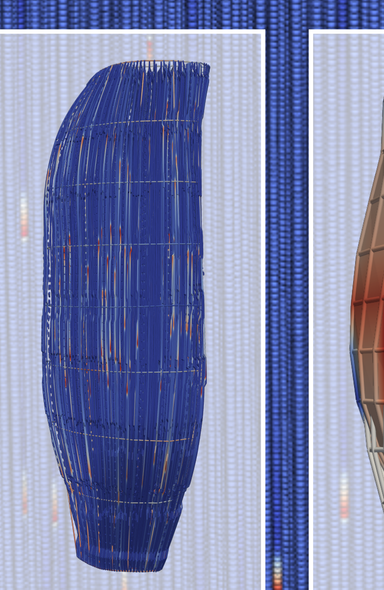
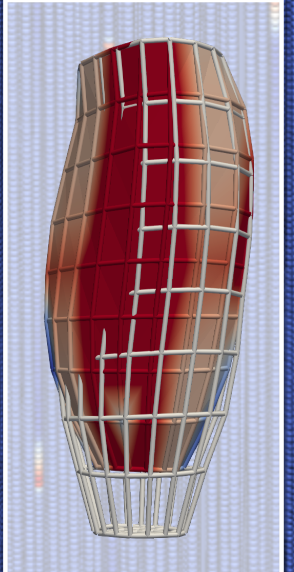
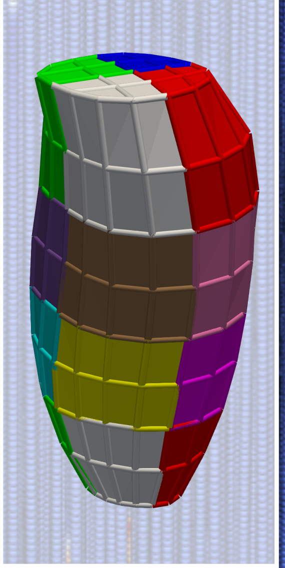
The human neuromuscular system consisting of skeletal muscles and neural circuits is a complex system that is not yet fully understood. Surface electromyography (EMG) can be used to study muscle behavior from the outside.
Computer simulations with detailed biophysical models provide a non-invasive tool to interpret EMG signals and gain new insights into the system.

The numerical solution of such multi-scale models imposes high computational work loads, which restricts their application to short simulation time spans or coarse resolutions.
We tackled this challenge by providing scalable software employing instruction-level and task-level parallelism, suitable numerical methods and efficient data handling. We implemented a comprehensive, state-of-the-art, multi-scale multi-physics model framework that can simulate surface EMG signals and muscle contraction as a result of neuromuscular stimulation.

This book describes the model framework and its numerical discretization, develops new algorithms for mesh generation and parallelization, covers the use and implementation of our software *OpenDiHu*, and evaluates its computational performance in numerous use cases.

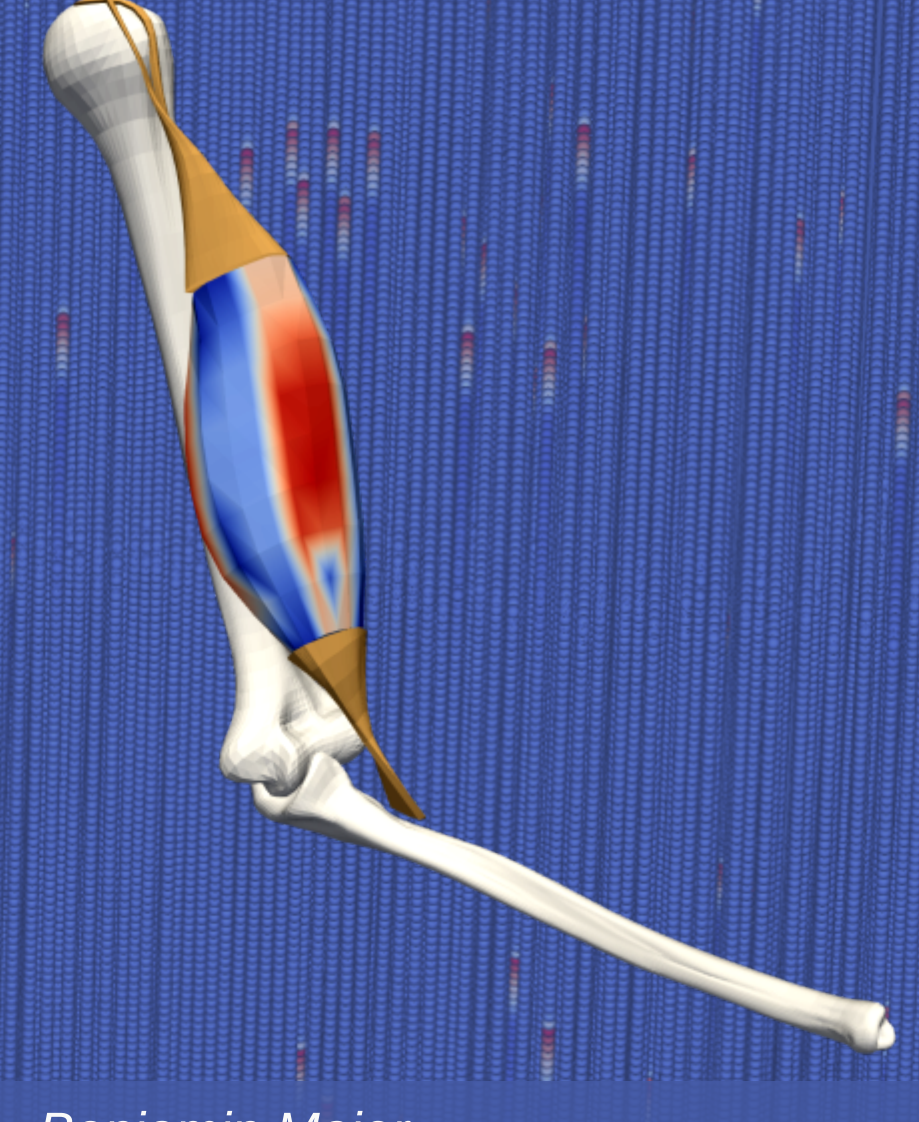






Maier

Scalable Biophysical Simulations of the Neuromuscular System



Benjamin Maier

Scalable Biophysical Simulations of the Neuromuscular System

Scalable Biophysical Simulations of the Neuromuscular System

Vom Stuttgarter Zentrum für Simulationswissenschaften (SC SimTech) und
der Fakultät für Informatik, Elektrotechnik und Informationstechnik
der Universität Stuttgart zur Erlangung der Würde eines Doktors
der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Benjamin Maier

aus Waiblingen

Hauptberichterin: Prof. Dr. Miriam Schulte
Mitberichter: Prof. Dr. Hans-Joachim Bungartz

Tag der mündlichen Prüfung: 22. Juni 2021

Institut für Parallele und Verteilte Systeme der Universität Stuttgart

2021

Benjamin Maier
Simulation of Large Systems
Institute for Parallel and Distributed Systems
University of Stuttgart, Universitätsstr. 38, 70569 Stuttgart, Germany
Contact: maierbn+thesis@gmail.com

We acknowledge the support and are thankful for the fruitful collaboration with the following institutions:

International Research Training Group “Soft Tissue Robotics”
Graduate School Simulation Technology

Group of Simulation of Large Systems,
Group of Scientific Computing,
Group of Usability and Sustainability of Simulation Software,
Institute for Parallel and Distributed Systems (IPVS), University of Stuttgart

Group of Computational Mathematics for Complex Simulation in Science and Engineering,
Institute of Applied Analysis and Numerical Simulation (IANS), University of Stuttgart

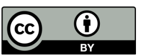
Group of Continuum Biomechanics and Mechanobiology,
Institute for Modelling and Simulation of Biomechanical Systems (IMSB), University of Stuttgart

High Performance Computing Center Stuttgart (HLRS)

Auckland Bioengineering Institute, University of Auckland, New Zealand

D 93 (dissertation)
Submitted to the University of Stuttgart

Copyright © 2021 Benjamin Maier.



This work is licensed under the
Creative Commons Attribution 4.0 International License.

To view a copy of this license, visit https://creativecommons.org/licenses/by/4.0/.

Contents

Abstract/Kurzzusammenfassung vii

Publications ix

1 Introduction 1
1.1 Anatomy and Physiology of the Human Skeletal Muscle 3
1.2 Use-Cases and Requirements for Simulations Used in in-Silico Experiments 5
1.3 Related Work and Software 7
1.4 The Multi-scale Model of the Neuromuscular System 12
1.5 Contributions and Scope of This Work 14

2 Comparative Study: Modeling Upper Arm Movement 19
2.1 Introduction 20
2.2 Experimental Study 24
2.3 Models 27
2.4 Results and Discussion 36
2.5 Conclusion 43

3 Generation of Meshes for the Multi-Scale Models 49
3.1 Overview and Notation of Required Meshes 50
3.2 Related Work 52
3.3 Preprocessing of the Muscle Geometry 58
3.4 Serial Algorithm to Create Muscle and Fiber Meshes 72
3.5 Parallel Algorithm to Create Muscle and Fiber Meshes 98
3.6 Results and Discussion 122
3.7 Conclusion and Future Work 140

4 Muscle Fibers and Motor Units 143
4.1 Introduction 143
4.2 Method 1: Assignment of Motor Units to a Given Set of Fibers 147
4.3 Method 2: Assignment of Motor Units to a Selection of Fibers 153
4.4 Assignment of Different Motor Units for Neighboring Fibers 154
4.5 Results and Discussion 155
4.6 Summary and Conclusion 164

5 Models and Discretization 167
5.1 Electrophysiology Model Equations 169

5.2 Model of Muscle Contraction 179
5.3 Discretization of the Electrophysiology Models 192
5.4 Discretization and Solution Approach for the Solid Mechanics Model . . . 208

6 Usage of the Software OpenDiHu **221**
6.1 Design Goals 221
6.2 Usage of OpenDiHu 225
6.3 Usage of CellML Models 249
6.4 Output File Formats 262

7 Implementation of the Software OpenDiHu **273**
7.1 Data Handling with PETSc 273
7.2 Finite Element Matrices and Boundary Conditions 290
7.3 Parallel Partitioning and Subsampling of Meshes 311
7.4 Parallel Solver for the Fiber Based Electrophysiology Model 323
7.5 Parallel Solver for the Multidomain Electrophysiology Model 333
7.6 Computation of CellML Models 342
7.7 Solid Mechanics Solver 352
7.8 Data Mapping Between Meshes 355

8 Numerical Results and Discussion **367**
8.1 Solution of Poisson and Diffusion Problems 367
8.2 Simulation of Solid Mechanics Models 370
8.3 Simulation of CellML Models 380
8.4 Simulation of Fiber Based Electrophysiology 385
8.5 Simulation of the Multidomain Model 419
8.6 Simulation of Coupled Electrophysiology and Solid Mechanics 429

9 Performance Analysis **447**
9.1 Performance Studies with OpenCMISS Iron 447
9.2 Performance Studies of the Electrophysiology Solver in OpenDiHu 458
9.3 Parallel Strong Scaling and Comparison with OpenCMISS Iron 467
9.4 Performance Measurements on the GPU 474
9.5 Parallel Scaling of the EMG Model Using High Performance Computing . . 480
9.6 Performance Studies of the Solid Mechanics Solver 485
9.7 Numerical Studies 489

10 Conclusion and Future Work **497**
10.1 Summary of this Work 497
10.2 Summary of Main Findings 499
10.3 Summary of Performance Results 501
10.4 Outlook and Future Work 502

Bibliography **505**

Abstract/Kurzzusammenfassung

Abstract

The human neuromuscular system consisting of skeletal muscles and neural circuits is a complex system that is not yet fully understood. Surface electromyography (EMG) can be used to study muscle behavior from the outside. Computer simulations with detailed biophysical models provide a non-invasive tool to interpret EMG signals and gain new insights into the system.

The numerical solution of such multi-scale models imposes high computational work loads, which restricts their application to short simulation time spans or coarse resolutions. We tackled this challenge by providing scalable software employing instruction-level and task-level parallelism, suitable numerical methods and efficient data handling. We implemented a comprehensive, state-of-the-art, multi-scale multi-physics model framework that can simulate surface EMG signals and muscle contraction as a result of neuromuscular stimulation.

This work describes the model framework and its numerical discretization, develops new algorithms for mesh generation and parallelization, covers the use and implementation of our software OpenDiHu, and evaluates its computational performance in numerous use cases.

We obtain a speedup of several hundred compared to a baseline solver from the literature and demonstrate, that our distributed-memory parallelization and the use of High Performance Computing resources enables us to simulate muscular surface EMG of the biceps brachii muscle with realistic muscle fiber counts of several hundred thousands. We find that certain model effects are only visible with such high resolution.

In conclusion, our software contributes to more realistic simulations of the neuromuscular system and provides a tool for applied researchers to complement in vivo experiments with in-silico studies. It can serve as a building block to set up comprehensive models for more organs in the musculoskeletal system.

Kurzzusammenfassung

Beim neuromuskulären System, bestehend aus Skelettmuskeln und Nervenbahnen, handelt es sich um ein komplexes System, welches noch nicht komplett verstanden ist. Das Muskelverhalten kann von außen durch Oberflächen-Elektromyografie (EMG) untersucht werden. Computersimulationen mit detaillierten, biophysikalischen Modellen stellen eine nichtinvasive Methode dar, um EMG-Signale zu interpretieren und neue Erkenntnisse über das System zu erlangen.

Die numerische Lösung solcher Mehrskalenmodelle erfordert eine große Rechenleistung, sodass die Modelle nur für kurze Simulationszeitspannen oder grobe Auflösungen geeignet sind. Wir lösen dieses Problem durch das Bereitstellen skalierbarer Software, welche Parallelität auf Instruktions- und Taskebene ausnutzt, geeignete numerische Methoden einsetzt und eine effiziente Datenverarbeitung sicherstellt. Wir setzen ein umfassendes, dem Stand der Wissenschaft entsprechendes Mehrskalen- und Mehrphysik-Modell um, welches Oberflächen-EMG-Signale simuliert und die Kontraktion eines Muskels als Folge neuromuskulärer Stimulation berechnet.

Diese Arbeit beschreibt das Modell und seine numerische Diskretisierung, entwickelt neue Algorithmen zur Gittererzeugung und Parallelisierung, behandelt die Anwendung und Umsetzung unserer Software OpenDiHu und wertet ihre Berechnungseffizienz in vielen Anwendungsbeispielen aus.

Wir erreichen eine um einen Faktor von mehreren Hundert schnellere Berechnung verglichen mit einem Referenzlöser aus der Literatur. Unsere Parallelisierung für Parallelrechner mit verteiltem Speicher und die Verwendung von Hochleistungsrechnern erlauben es uns, Oberflächen-EMG des Biceps Brachii mit einer realistischen Anzahl an Muskelfasern von mehreren Hunderttausend zu simulieren. Wir stellen fest, dass bestimmte Modelleffekte nur mit solch hoher Auflösung sichtbar werden.

Unsere Software trägt zu realistischeren Simulationen des neuromuskulären Systems bei und stellt ein Werkzeug für die angewandte Wissenschaft zur Verfügung, um In-vivo-Experimente mit In-silico-Studien zu verknüpfen. Sie kann als Baustein zur Erstellung umfassender Modelle für weitere Organe im muskuloskelettalen System dienen.

Publications

The following publications discuss various topics that are covered in this thesis.

Peer-Reviewed Publications

[Bra18] **Bradley**, C. P.; **Emamy**, N.; **Ertl**, T.; **Göddeke**, D.; **Hessenthaler**, A.; **Klotz**, T.; **Krämer**, A.; **Krone**, M.; **Maier**, B.; **Mehl**, M.; **Rau**, T.; **Röhrle**, O.: *Enabling detailed, biophysics-based skeletal muscle models on HPC systems*, Frontiers in Physiology 9.816, 2018, doi:10.3389/fphys.2018.00816

[Mai19] **Maier**, B.; **Emamy**, N.; **Krämer**, A. S.; **Mehl**, M.: *Highly parallel multi-physics simulation of muscular activation and EMG*, COUPLED PROBLEMS 2019, 2019, pp. 610–621, isbn:978-84-949194-5-9, http://hdl.handle.net/2117/190149

[Mai21f] **Maier**, B.; **Stach**, M.; **Mehl**, M.: *Real-time, dynamic simulation of deformable linear objects with friction on a 2d surface*, Mechatronics and Machine Vision in Practice 4, 2021, doi:10.1007/978-3-030-43703-9

[Wal20] **Walter**, J. R.; **Saini**, H.; **Maier**, B.; **Mostashiri**, N.; **Aguayo**, J. L.; **Zarshenas**, H.; **Hinze**, C.; **Shuva**, S.; **Köhler**, J.; **Sahrmann**, A. S.; **Chang**, C.-m.; **Csiszar**, A.; **Galliani**, S.; **Cheng**, L. K.; **Röhrle**, O.: *Comparative study of a biomechanical model-based and black-box approach for subject-specific movement prediction**, 2020 42nd Annual International Conference of the IEEE Engineering in Medicine Biology Society (EMBC), 2020, pp. 4775–4778, doi:10.1109/EMBC44109.2020.9176600

Forthcoming Publications

[Krä21] **Krämer**, A.; **Maier**, B.; **Rau**, T.; **Huber**, F.; **Klotz**, T.; **Ertl**, T.; **Göddeke**, D.; **Mehl**, M.; **Reina**, G.; **Röhrle**, O.: *Multi-physics multi-scale HPC simulations of skeletal muscles (accepted)*, High Performance Computing in Science and Engineering '20, Transactions of the High Performance Computing Center, Stuttgart (HLRS) 2020, ed. by **Nagel**, W.; **Kröner**, D.; **Resch**, M., Springer International Publishing, 2021

[Mai21d] **Maier**, B.; **Göddeke**, D.; **Huber**, F.; **Klotz**, T.; **Röhrle**, O.; **Schulte**, M.: *OpenDiHu - Efficient and Scalable Software for Biophysical Simulations of the Neuromuscular System (in preparation)*, Journal of Computational Physics, 2021

[Mai21e] **Maier**, B.; **Mehl**, M.: *Mesh generation and multi-scale simulation of a contracting muscle-tendon complex (under review)*, Journal of Computational Science, 2021

[Mai22] **Maier**, B.; **Schneider**, D.; **Schulte**, M.; **Uekermann**, B.: *Bridging scales with volume coupling – scalable simulations of muscle contraction and electromyography (under review)*, High Performance Computing in Science and Engineering '21, 2022

Source Code and Data

The most recent version of the OpenDiHu software is made available in the GitHub repository at `https://github.com/maierbn/opendihu`. The released version and the packaged data that are needed to reproduce the results in this work, containing, e.g., mesh files and CellML models, are given below.

[Mai21a] **Maier**, B.: *Input data for OpenDiHu simulations*, version 1.3, Zenodo, 2021, doi:10.5281/zenodo.4705945, https://doi.org/10.5281/zenodo.4705945

[Mai21b] **Maier**, B.: *OpenDiHu*, version 1.3, Zenodo, 2021, doi:10.5281/zenodo.4706049, https://doi.org/10.5281/zenodo.4706049

Chapter 1

Introduction

Tying the shoestrings, running to catch the train, quickly slipping through the closing door, and then lifting a heavy suitcase to the luggage rack over the seat—all actions that are only possible because of the versatility of the musculoskeletal system. Voluntary contractions of skeletal muscles enable humans to perform a variety of tasks: finely controlled and coordinated actions, endurance tasks, fast and vigorous actions, and exercises requiring high forces.

Moreover, skeletal muscles are able to be trained and adapt to requirements, can self-repair, and usually keep their capabilities for an entire lifetime. Understanding this remarkable system that has evolved over millions of years can advance both engineering and healthcare.

From an engineering view, derived biomimetic systems such as powered exoskeletons or robot arms with muscle-like actuators exhibit promising properties such as being lightweight, inexpensive, resilient, damage tolerant, noiseless, and agile and, thus, are potentially emerging field in robotics and medicine [Bar03; Bar04; Mir18].

In the fields of healthcare and medicine, research is interested in obtaining a better understanding of muscular diseases such as muscular dystrophies [Eme02]. Studies show that disabling inherited neuromuscular diseases are prevalent in 1 out of 3500 of the population [Eme91]. However, for most of the neuromuscular disorders no cure is known and treatment focuses on reducing symptoms [Eme02; Hei15]. Developing treatments to neuromuscular disorders is only possible with an extensive understanding of the neuromuscular system. Similarly, for diagnosing the type of disorder from symptoms and clinically available examination tools such as electromyographic recordings, a comprehensive understanding of muscle physiology is needed.

Surface electromyography (sEMG) measures the temporally changing electric potentials on the skin surface that are induced by activation of the muscle fibers [Mer04]. It is

one of the few non-invasive diagnostic tools to gain insights into the functioning of the neuromuscular system. High-density surface EMG (HD-sEMG) involves the signal acquisition by an array of electrodes on the skin surface and, thus, enriches the traditional, monopolar EMG by spatial information about the muscular activity.

Another application, where insights into the neuromuscular system advance technology, bridges the two fields of engineering and healthcare: Exoskeletons for rehabilitation, e.g., of stroke patients, can be controlled by sEMG or HD-sEMG signals from the patients to accurately support the intended movements (e.g., [Leo15; Mul05; And05]).

Despite the need to gain comprehensive insights, experimental in vivo investigations of the neuromuscular system have severe limitations: Boundary conditions, such as contraction velocities, often cannot be accurately controlled, studies are not repeatable because of fatigue effects, material parameters of individual subjects are not known and cannot be measured precisely, and the quantities of interest, such as activation values and active stresses cannot be measured easily. Moreover, experiments are strongly limited to the ethical bounds of natural movements.

A controlled environment for such investigations can be provided by in silico experiments, i.e., using computer simulations. The main advantages of using simulations are unlimited access to all computed quantities, reproducibility, and freedom in the experimental protocol. With appropriate models, predictions can be made even for pathological conditions.

Employing in silico experiments demands a careful formulation and composition of mathematical models, using experimentally found evidence about the functioning of various aspects in the muscular system. Once a model is set up, its execution requires suitable numerical methods and efficient implementation to utilize the available compute hardware in the best way.

This work discusses these numerical methods and their efficient implementation on parallel hardware. The present chapter introduces the fundamentals: Section 1.1 presents the basic anatomy and physiology of a skeletal muscle. Section 1.2 takes a closer look at the application of the in silico laboratory and derives requirements on the simulation technology. Section 1.3 outlines the current state of the art in skeletal muscle simulation. One of the most promising model frameworks that we use is described in more detail in Sec. 1.4 before contributions of this work are summarized in Sec. 1.5.

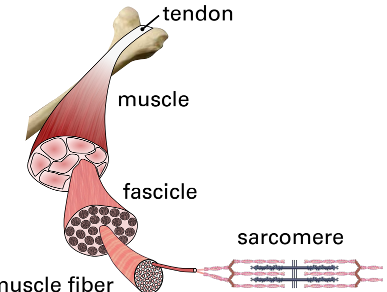


Figure 1.1: Hierarchical structure of a skeletal muscle consisting of fascicles, muscle fibers and sarcomeres.

1.1 Anatomy and Physiology of the Human Skeletal Muscle

Skeletal muscles have a hierarchical structure as shown in Fig. 1.1. On the macroscopic level, fibro-elastic tendons connect the muscle to the skeletal system. The muscle is composed of tens of fascicles with the exact number strongly depending on the muscle (numbers according to [Mac06]). Each fascicle contains between ten and 10,000 muscle fibers, yielding a total of up to one million fibers in a muscle. Each muscle fiber usually runs through the whole length of the muscle. A muscle fiber consists of numerous parallel myofibrils, which each consists of series of sarcomeres, the smallest contractile unit of a muscle. A muscle fiber contains approximately 50,000 sarcomeres and, thus, there are from millions up to billions of sarcomeres in a whole muscle.

The contraction of the muscle is controlled by motor neurons in the spinal cord. The axons of each alpha motor neuron innervate multiple fibers in the muscle. In consequence, all connected fibers are always activated simultaneously. The set of fibers together with their motor neuron form a motor unit (MU).

The neuromuscular junctions where the axons innervate the muscle fibers are mostly located in a band within the mid-belly of the muscle [Chi04]. Upon activation of a muscle fiber, an electric stimulus, the action potential, travels from the neuromuscular junction towards both ends of the muscle. The action potential triggers subcellular processes and leads to force generation in the sarcomeres. The fibers are electrically isolated to each other but mechanically coupled through the fascicles and the extracellular matrix.

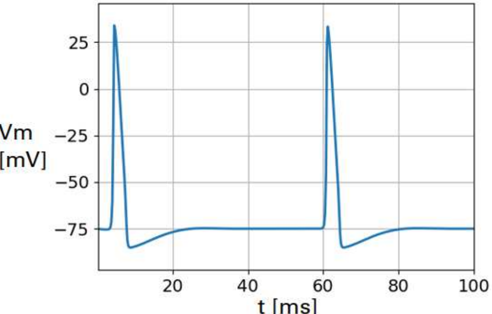


Figure 1.2: Action potentials over time at a fixed point on a muscle fiber, calculated by the monodomain equation with a subcellular model of Hodgkin and Huxley using the software OpenDiHu (More details are given in later sections.).

The propagation of action potentials is governed by ionic currents through ion channels in the fiber membranes and is driven by ion pumps and the activation and deactivation of the ion channels. Figure 1.2 shows the shapes of two subsequent action potentials over time at a fixed point on a muscle fiber. The transmembrane potential V_m initially equals its resting state of −75 mV. After stimulation occurs, the potential rapidly depolarizes to a maximum value of approximately 30 mV, followed by the repolarization and a small overshoot, before returning to the resting potential. After approximately 30 ms, the system is again in equilibrium, and the action potential induced by the next stimulus has the exact same shape.

The MUs are activated according to the size principle, starting with the smallest ones that connect to the least fibers and successively adding larger MUs [Mil73]. The amount of muscle activation is controlled by the number of MUs and the rate-encoded stimulation signals for every MU. This finely controlled level is further modulated by the feedback loops of the neuromuscular system. Sensory organs are located within the muscle sense stretch, contraction velocity, and forces and influence the motor neuron pool. A more elaborate description of the anatomy and physiology of the neuromuscular system can be found in the book of MacIntosh et al. [Mac06].

Considering the origin of EMG signals, all action potentials on the muscle fibers contribute to the electric potential in the muscle volume. While electric conduction is directed inside the muscle fibers, anisotropic conduction occurs in the volume of extracellular space. In addition, electric conduction in adipose tissue above the muscle belly influences the electric potential on the skin surface, which can be measured by EMG.

1.2 Use-Cases and Requirements for Simulations Used in in-Silico Experiments

With a basic understanding of the physiology of the neuromuscular system, we can now define use-cases for in-silico experiments and derive the requirements for models and simulation software.

A simulation should be able to accurately predict the response of the neuromuscular system to different recruitment strategies of the MUs. Also, different organizations of muscle fibers in MUs could be investigated. Similarly, the sensory feedback loop within a single muscle is by far not yet fully understood. Various assumptions could be tested in simulations, and the resulting force and EMG outputs could be compared to experiments. By complementing in vivo and in silico experiments, more comprehensive insights can be generated.

A second use case for simulations of the neuromuscular system lies in the decomposition of EMG recordings. Traditionally, signal processing techniques are used to draw conclusions from EMG data [Mer04; Far10]. Decomposition algorithms exist that identify discharge patterns of individual motor units in HD-sEMG and additively decompose the recording [De 06; Naw10; Hol07b]. Novel, data-based techniques exist that employ deep learning methods [Cla21].

However, these techniques have limitations. The recorded signals are typically weak and noisy because of the layer of body fat between the muscle and the EMG electrodes. Cross-talk from adjacent muscles and destructive interference between signals of spatially close muscle fibers make the decomposition more difficult. Often, only isometric contractions can be considered in experiments since large movements of the muscles with respect to the electrodes would add additional uncertainties to the recorded signal.

Simulations can provide a controlled testing environment for such EMG decomposition algorithms. For data-based methods, simulations are unavoidable to generate training and validation data.

The listed use-cases for in-silico experiments demand detailed, biophysically informed models. Phenomenological descriptions cannot predict unseen scenarios or pathological conditions. The hypotheses to test related to MU organizations, recruitment, or sensory feedback have to reflect in the choice of the model description. A suitable model usually needs to take into account the multi-scale nature of the neuromuscular system. The

geometric structure of muscle fibers embedded in the muscle belly and the layer of adipose tissue have to be part of accurate models.

Multi-scale multi-physics simulations with high resolutions involve high computational loads. The simulated processes on a molecular scale, e.g., in the sarcomere require small timestep widths in the range of microseconds. At the same time, macroscopic quantities such as EMG signals and muscle contraction should be computed, leading to desired overall simulation time spans in the range of seconds. Fine three-dimensional (3D) meshes are needed to achieve high spatial resolutions. To resolve individual muscle fibers, additional one-dimensional (1D) fiber meshes are considered.

To account for fine resolutions and a high number of timesteps in an acceptable runtime, the potential of today's and tomorrow's computer hardware has to be fully exploited. This requires task-level and instruction-level parallelism. For example, the latest processor of the Intel Core X series (the Intel Core i9-10980XE Extreme Edition Processor), which is listed at a customer price below $1000 contains 18 hardware cores, allowing to run 16 tasks in parallel. It supports Intel AVX-512, a technology with which eight double precision floating point operations can be executed per instruction. In a higher price segment, it is, e.g., possible to combine two AMD EPYC 7742 server processors into a shared memory compute node with 128 cores. Distributed memory clusters allow the combination of almost any number of compute nodes to achieve higher total core counts. The supercomputer Hawk at the High Performance Computing Center Stuttgart combines 5632 of the mentioned AMD nodes into an overall cluster of 720 896 cores.

Thus, a requirement to the simulation software is to be able to run on distributed memory computer systems. This requires efficient data management and a domain decomposition approach where the computational domain is partitioned into one subdomain for each process. Highly parallel domain decomposition requires appropriately structured meshes and efficient parallel linear solvers. At the same time, muscle geometries obtained from medical imaging should be used to obtain a realistic setting. In a preprocessing step, the required highly resolved meshes have to be generated from imaging datasets.

A highly resolved simulation model can be used to estimate the accuracy of reduced models that do not include all biophysical processes or have reduced spatial resolutions. The advantage of such reduced models is that they can be solved with lower resources or in shorter runtimes. To assess the error of the reduced resolution, comparisons with results of the full model can be carried out. In this sense, the full model should be able to be used with a realistic number in the order of several 100 000 muscle fibers and hundreds of MUs to allow for a comparison with simulations of smaller numbers of fibers.

Highly resolved simulations are known to exhibit numerical instabilities, poor conditioning, or other causes for divergence in the numerical solvers. Therefore, numerical schemes have to be chosen carefully. At the same time, timestep widths can be increased and runtimes reduced by choosing, e.g., second order timestepping schemes instead of first order schemes.

Another important requirement of the simulation software can be formulated from the user's perspective. Configuring a simulation and exchanging material and numerical parameters should be possible in a convenient way. Simulation results should be accessible in various established file formats, to be examined in dedicated visualization software or used in further post-processing. On the modeling side, comprehensive state-of-the-art models should be implemented while maintaining the possibility to extend given multi-scale models later on as research advances. Standards in the biochemical modeling community should be respected and incorporated, such as the description language CellML [Cue03; Llo04] for subcellular models. To find the most suited numerical solvers, the software should be flexible enough to, e.g., easily exchange timestepping schemes or employ different linear system solvers.

Our contribution is to implement and employ software that fulfills all these requirements. We aim at simulating EMG and muscle contraction with detailed, biophysically informed multi-scale models. The software runs efficiently on the previously described hardware, ranging from workstation computers to supercomputers.

1.3 Related Work and Software

In the following, we give an overview of existing approaches for modeling the neuromuscular system. The overview involves literature and software frameworks and focuses on the multi-scale model that is the basis for the present work. For a recent, comprehensive review on all aspects of neuromuscular modeling, we refer to [Röh19].

1.3.1 Related Work

The lowest computational effort is required when analytically solvable models are used to simulate skeletal muscle forces. The twitch force of a single motor unit can be described by the impulse response of a critically damped, second-order system, for which an analytical

solution exists. For the given superposition of all motor unit action potentials, the transient output force of the muscle is computed [Cis08; Did10].

On the next level of detail, phenomenological Hill-type muscle models, which have to be solved numerically are used to describe muscle forces along a one-dimensional line of action. They are often used for systemic simulations of larger parts of the musculoskeletal system [Zaj89; Del07; Hae14; Bay17]. We use Hill-type models in our case study on predicting forces of the upper arm. However, this type of model is not suited for simulations of EMG and neglects structural properties of the muscle tissue.

While phenomenological models describe a whole muscle by only a few parameters, continuum-based models exist that also take into account structural features and spatial heterogeneity [Joh00; Ble05a; Röh07; Böl08].

A commonly used approach to model muscle contraction in continuum-mechanics is to additively compose the stress tensor of a passive and an active stress term [Ble05b; Joh00; Röh08]. The passive muscle behavior can be parametrized using experimentally found relations, though this is challenging in practice [Böl12; Tak13; Van08; Van06].

Multi-scale models exist that combine formulations of continuum-mechanics with a description of electrophysiology [Röh08; Röh12; Hei13; Her13]. These models couple various physical phenomena that occur on different temporal and spatial scales on cell, tissue and organ levels, such as subcellular ion dynamics in scales of milliseconds and micrometers and mechanical stresses and electric potentials in scales of seconds and multiple centimeters.

Model order reduction techniques and surrogate modeling have been applied to these complex, full models to speed up the computations [Mor17; Val18].

EMG signals of activated muscles can be computed by volume conductor models [Mes13]. Both analytic [Dim98; Far01; Mes06] and numerical methods exist [Low02; Mor15; Mor17; Klo20].

We combine existing multi-scale models of electrophysiology, muscle contraction and generation of EMG with different subcellular models and electrophysiology formulations as well as motor neuron and afferent feedback models to form a novel, comprehensive multi-scale modeling framework for the neuromuscular system. We solve these models using numerically efficient schemes that are implemented in our unified simulation software environment named OpenDiHu.

1.3.2 Related Software

Few software packages exist in the open source world that can be used for comprehensive multi-scale modeling of the neuromuscular system. In the following, CellML, OpenCMISS, Chaste, FEBio, and the generic frameworks OpenFOAM and FEniCS will be briefly evaluated.

A useful and widespread technology for biochemical models is CellML [Cue03; Llo04]. The open standard CellML language allows defining differential-algebraic equations with physical units. It can be used to develop mathematical descriptions of biophysical processes such as subcellular or neuron models. The description language has also been used for broader applications, e.g., for constitutive material laws.

CellML provides an online repository where mathematical models and metadata such as figures and related publications are collected. The models can be downloaded as code in various formats and programming languages. Existing models can be combined into new models in a hierarchical manner. Dedicated modeling environments for CellML models exist. As an example, OpenCOR [Gar15] can be used to edit, simulate and visualize CellML models. Moreover, application programming interfaces (APIs) exist, which provide low-level access to models in CellML format and allow software frameworks to integrate CellML functionality. Among the software frameworks with CellML support are OpenCMISS and Chaste.

OpenCMISS (Continuum Mechanics, Imaging, Signal processing, and System identification) [Bra11] provides a set of open source libraries and applications for modeling and visualization of bioengineering problems. The frameworks allow using CellML models [Nic14]. OpenCMISS Iron, the computational engine, can solve finite element models, discretized also with higher order elements and using Cartesian or curvilinear coordinates. For example, the ventricles of the heart were modeled with a low number of cubic Hermite elements in a prolate spheroidal coordinate frame [Smi04]. Various nested timestepping loops and solvers can be configured to create multi-scale models.

The library is programmed largely in the Fortran-90 standard, wrappers for the Python programming language can be automatically generated. It supports parallel execution on distributed memory systems.

The development of OpenCMISS Iron started in 2005 as a rewrite of the computational modeling tool CMISS, whose history dates back to 1980. It is part of the Physiome Project, an international collaborative open-source effort to provide a public domain

framework for computational physiology [Hun04]. OpenCMISS has been used for multi-scale modeling of the lungs and heart [Smi04], vascular and thermoregulatory system [Lad16; Gha20] and skeletal muscle [Hei13].

The “Cancer, Heart, and Soft Tissue Environment” (Chaste) is an open source C++ library targeted at simulations of physiology and biology in general [Mir13]. The code development is driven by cardiac electrophysiology and cancer growth simulations, but the framework is also capable of solving ordinary and partial differential equations from other fields. This involves solvers for CellML models, which have been used to simulate cellular cardiac electrophysiology [Coo15].

Chaste [Coo15] also uses the approach of first converting a CellML description into C++ code using the tool *PyCml* [Coo06]. Chaste features adaptive timestepping solvers such as the *CVODE* solver from the *SUNDIALS* package [Coh96] and infers analytic Jacobians from the model equations. The CellML support of Chaste targets “automated use” by automatically inferring standard variable names, e.g., for membrane voltage and stimulation current.

While cardiac and skeletal muscle tissue is similar with respect to its multi-scale structure, significant differences exist regarding electrophysiology and recruitment of MUs. On cardiac tissue, propagation of action potentials occurs uniformly on a three-dimensional domain, whereas in the skeletal muscle, a multitude of electrically isolated one-dimensional muscle fibers are recruited independently. Thus, significant development efforts are needed to transform a cardiac simulation into a simulation of skeletal muscles.

In contrast to OpenCMISS Iron, Chaste advertises its test-driven development process to ensure code quality, correctness and reusability [Pit09]. Similar to Iron, the Chaste code runs in parallel on distributed memory systems and uses external numeric libraries for linear system solvers. It also implements a solver for 3D incompressible nonlinear elasticity, which is needed for simulating muscle contraction. However, this solver is not yet parallelized.

A simulation tool specialized in the field of biomechanics is the FEBio project [Maa12; Maa17]. It provides an advanced finite element solver for continuum mechanics of muscle tissue and implements a well-documented library of material models, from basic to advanced and state-of-the-art models. The most recent version includes graphical pre-processing and post-processing tools. Whereas OpenCMISS Iron and Chaste require some knowledge of programming and command line usage, FEBio can be used right away also by application scientists. Unlike OpenCMISS and Chaste, FEBio only runs in parallel on shared memory computers, which makes it unsuited for High Performance Computing.

FEBio contains no electrophysiology models. Prescribing different levels of activation at different locations in a muscle currently is only possible by a workaround of defining separate materials for every finite element. However, FEBio is extensible by user-defined plugins, and multi-scale models would have to be implemented in this way.

More generic simulation frameworks exist that can also be considered for simulations of the musculoskeletal system.

OpenFOAM [Jas07] is a well-known C++ software framework that provides methods for “Field Operation And Manipulation”. It is mainly designed for continuum mechanics problems in the field of computational fluid dynamics and uses the finite volume method. This method can also be used to solve nonlinear solid mechanics problems [Car14].

Another established general framework for solving partial differential equations is FEniCS [Aln15]. It provides a high-level Python interface to directly describe the model in variational form using predefined operators. Then, it derives finite element discretizations, which it is also able to solve in parallel.

Advantages of such generic frameworks are their mature and efficient solvers and infrastructure such as output file formats and their comprehensive documentation and support. Disadvantages are the missing domain-specific functionality. For example, no solvers for CellML models exist in OpenFOAM and FEniCS. For FEniCS, all existing parts of the desired multi-scale model would have to be formulated in the unified form language. This task needs a more in-depth understanding of the framework for the special requirements of the complex model, e.g., the dynamic, incompressible, nonlinear solid mechanics muscle contraction model with active stress contribution, for which mixed finite element formulations and possibly special numerical preconditioners and solvers are needed. This relativizes the advantages of the high-level interface. Furthermore, in generic frameworks, it is more difficult to bring own problem-specific contributions to the core code trunk to make them publicly available and reusable.

Therefore, we select OpenCMISS Iron as the starting point for implementing multi-scale models. We use the multi-scale chemo-electro-mechanical model that was introduced in [Röh12] and initially implemented in the software OpenCMISS for the tibialis anterior muscle [Hei13]. However, this implementation did not fully exploit the parallel capabilities of Iron as it was hard-coded for four processes. We remove this restriction and further improve runtimes of the existing electrophysiology model by implementing second order timestepping schemes.

We evaluate the performance regarding parallel scaling and memory consumption on a supercomputer. While the performance is good for small degrees of parallelism, we see an unavoidable barrier for larger parallelism and High Performance Computing (HPC). This barrier arises due to fundamental design decisions in the memory layout and is difficult to overcome in the existing OpenCMISS Iron code.

Furthermore, some of the functionality we require for our multi-scale framework is not available: Arbitrary data mapping between meshes is needed for a 3D muscle domain with embedded 1D fibers. Output files use a text-based format that is not suited for HPC, established parallel file formats such as defined by VTK or ADIOS are not available. The nonlinear mechanics solver can only solve static problems. At the same time, the implementation assumes generic coordinate frames and element shapes, which are not necessarily needed in our models. Due to the lack of modern programming language features such as object-orientation and polymorphism, it is very difficult to extend the solid mechanics solver to a fully dynamic formulation.

Therefore, we move the existing models to the new code base *OpenDiHu* and expand the multi-scale framework by new model components. This gives us the flexibility to implement all requirements listed in Sec. 1.2 and target towards HPC from the beginning. For compatibility, OpenDiHu can write the same output file format as OpenCMISS. Furthermore, an adapter in OpenDiHu allows to integrate the FEBio mechanics solver with the electrophysiology solver of OpenDiHu.

1.4 The Multi-scale Model of the Neuromuscular System

Next, we briefly describe the existing multi-scale model framework that we base our work on. The chemo-electro-mechanical model was introduced by [Röh12] and described in more detail in [Hei13] and [Hei15]. It has been used to investigate different muscle fibers lengths [Hei14] and later was enhanced by also modeling actin-titin interactions [Hei16]. The work of [Mor15] extended the framework by a description of EMG signals.

This model reflects the structural and functional aspects of skeletal muscle tissue and describes its mechanical and electrophysiological properties. Different biophysical processes are realized by sub-models that are linked together to form the overall multi-scale and multi-physics model.

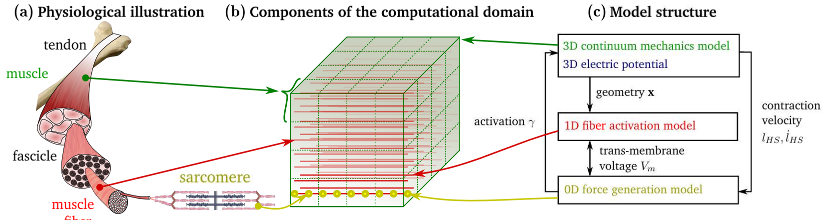


Figure 1.3: Modeling skeletal muscle physiology: From the anatomy (a) over a multi-scale discretization (b) to the multi-physics model (c) of [Röh12].

Figure 1.3 visualizes the model structure. Figure 1.3 (a) depicts the hierarchical skeletal muscle anatomy consisting of muscle, muscle fibers and sarcomeres. Figure 1.3 (b) shows the finite element discretization of these three scales. The muscle is represented by a 3D mesh of hexahedral elements (green). Muscle fibers are modeled as 1D fiber meshes (red) that are embedded in the muscle domain. The nodes of the fiber meshes are locations of 0D sarcomere models (yellow). Figure 1.3 (b) depicts them only for one fiber, however, the nodes of all fibers feature instances of this model. The cube-shaped 3D domain was chosen for the sake of a clear visualization, our simulations use real muscle geometries instead.

Figure 1.3 (c) shows details of the model parts and their exchanged physical quantities. In summary, the model consists of 3D, 1D, and 0D components, which are given in different colors. The green, blue, red, and yellow colors are used throughout this work to indicate these sub-models or the three different spatial scales.

The continuum mechanics model describes muscle contraction and is defined on the 3D mesh. The same mesh is also used for computing the 3D electric potential fields within the volume. The deforming 3D muscle domain defines the geometry, i.e., node positions $\mathbf{x}$ of the embedded 1D fibers meshes. The 1D fiber activation model computes the propagation of action potentials on every fiber. It is strongly coupled via the transmembrane voltage V_m to the 0D force generation model on the sarcomeres, which is also called the subcellular model. It depends on the length ℓ_{HS} of the half-sarcomere and the contraction velocity $\dot{\ell}_{\text{HS}}$. These quantities are computed in the 3D model and mapped to the 0D points. The result of the subcellular model is the activation parameter γ that is homogenized and used as input for the active stress term in the 3D continuum mechanics model.

1.5 Contributions and Scope of This Work

In the following, we give a summarizing preview of the main contributions of this work to the world of existing in-silico models and tools. The contributions include:

(i) **Model extensions.** In addition to the previously existing model components depicted in Fig. 1.3 (c), we add a mesh for adipose tissue to simulate EMG signals on the skin surface. The corresponding model was formulated by [Mor15], however, it has not been implemented together with the other components in a simulation program prior to our work.

Furthermore, we add the multidomain model [Klo20], an alternative homogenized 3D description of electrophysiology that can replace the 1D fiber activation model and the 3D electric potential in Fig. 1.3 (c).

Recruitment of the muscle fibers was previously done in a preprocessing step by simulating motor neuron models such as [Cis08; Neg11]. In OpenDiHu, we explicitly couple models of the motor neuron pool as well as models of sensory organs such as muscle spindles and Golgi tendon organs. This allows us to close the loop of afferent neural feedback.

Another extension is the consideration of tendons together with the contracting muscle. We add separate models and meshes for the tendons that are mechanically coupled to the muscle belly.

The previous quasi-static mechanics formulation in OpenCMISS Iron is also implemented in OpenDiHu and extended to a fully dynamic formulation. Instead of a numerical approximation of the Jacobian matrix in the nonlinear system in Iron, we automatically derive an analytic description in OpenDiHu. This significantly reduces the runtime and allows simulating finer meshes than is possible with OpenCMISS.

Current limitations among the implemented models are convergence difficulties for solid mechanics problems with more than approximately 1000 elements. These are not a result of the implementation but a numerical problem and could be addressed by different nonlinear solver schemes in the future.

Whereas the computation using the fiber based description of electrophysiology is close to its optimum performance and scales near-optimally for any degree of parallelism and number of fibers, the corresponding multidomain implementation exhibits high memory consumption, is less robust with respect to numerical errors and more difficult

to parallelize. This restricts its application to approximately 20 motor units, 128 processes, and timespans of below a second. If scenarios above these limits are required, the fiber based models should be used.

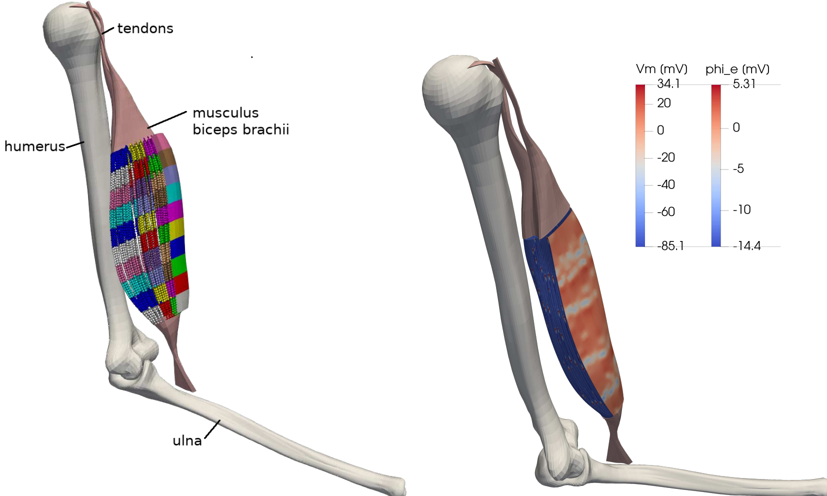
(ii) **Preprocessing Algorithms.** A serial and a parallel algorithm are developed to generate the high-quality 1D and 3D meshes that are required for the simulation from imaging data. The algorithms are applied on the biceps and triceps brachii muscles.

Furthermore, a method is derived to assign fibers to motor units according to physiological properties. Both implementations are made publicly available together with the open-source software OpenDiHu.

(iii) **Improvements in OpenCMISS Iron.** Improvements include the parallelization to an arbitrary number of cores of the existing implementation of the chemo-electro-mechanical model, an algorithmic improvement from quadratic to linear time complexity in the homogenization functionality of the activation parameter and the introduction of configuration files such that different parametrizations can be simulated without recompiling. Further, numerical experiments concerning employed solvers and timestep widths are conducted. The revised choices, e.g., a conjugate gradient scheme instead of the GMRES solver lead to faster computation times.

(iv) **Development of OpenDiHu.** Implementation of an efficient, flexible framework for simulating the full models of surface EMG, muscle contraction as well as subsets of the mathematical multi-scale modeling framework. The software can employ CPUs and GPUs and run on small workstation computers, compute clusters, and supercomputers. As this is the most comprehensive contribution, we refer to the implementation and results chapters, Chapters 7 and 8 for details. Highlights are the simulation of EMG signals with 270 000 muscle fibers on 27 000 cores of the supercomputer Hazel Hen and an overall speedup factor of 200 with respect to the community standard software OpenCMISS Iron.

Figure 1.4 shows two snapshots of a simulation that are characteristic for this work: Figure 1.4a depicts the biceps brachii muscle with a body fat layer. The muscle belly and the fat layer are discretized by nodes and partitioned to multiple processes, indicated by different colors. Figure 1.4b shows a simulation of EMG signals on the skin surface. Solutions of the electrophysiology models can be seen on the fibers and on the surface above the muscle.



(a) The bones of the upper arm with tendons and muscle tissue of the biceps brachii muscle. The colored patches show the domain decomposition of the muscle and of the body fat layer domains.

(b) Simulation of action potential propagation on the muscle fibers (mainly blue, value V_m according to legend) and EMG on the skin surface (value ϕ_e).

Figure 1.4: Preview on setting and simulations of a biceps muscle in this work.

The scope of this work is to efficiently compute the described models and provide an environment to carry out processing and investigations. The models themselves, as well as their parameters, are taken from the literature. It is known that the properties of human organs vary greatly between individuals. For example, the number of muscle fibers in a biceps muscle varies between 172 000 and 419 000 [Mac84]. By parameter fitting, the simulations could be adjusted to represent a particular individual. Within this work, this was done in the initial case study about the upper arm movement for a data-based and a Hill-type based model. However, parameter fitting for the multi-scale model and validation experiments or even preclinical studies on patients with musculoskeletal diseases are beyond the scope of this work. Similarly, our software could be used to design studies that foster the understanding of the neuromuscular system. However, such investigations are also beyond the scope of this thesis.

The developed methods in this work were applied to simulations of the biceps brachii muscle, as this muscle allows straightforward EMG recordings and is well-studied in liter-

ature. Nevertheless, most of the methods and results are also applicable to other muscles. The anatomical match of the used simulation models could be improved by additionally considering the aponeurosis in the biceps muscle or by differentiating between the two muscle heads during motor unit recruitment. However, these model extensions are also not within the scope of the present work. Some notes for future work can be found in Sec. 10.4.

The presented findings and conclusions were partly shaped by discussions with various researchers with expertise from different disciplines. Yet, this doctoral thesis lists essentially own contributions, marks collaborative work in the text, and indicates others' work by citations. Using the pronoun "we", the author refers to the group of the originator, potentially the supervisors, and certainly the interested reader.

The remainder of this work contains the following chapters: Chapter 2 compares two model approaches to simulate upper arm movement. Chapter 3 develops algorithms to generate the meshes that are required in the solution of the multi-scale model. Chapter 4 addresses the assignment of motor units to muscle fibers. Chapter 5 describes all used model equations and their discretization. Chapter 6 introduces the software OpenDiHu and describes its usage. Chapter 7 gives details on the implementation of OpenDiHu. Chapter 8 presents and discusses numerical results. Chapter 9 studies the computational performance of the solvers. Chapter 10 concludes the work and gives an outlook to future work.

Chapter 2

Comparative Study: Modeling Upper Arm Movement

Moving one's upper arms and forearms is an action that is performed unnoticed every day. What seems like a trivial task involves a sophisticated interplay of muscles, tendons, bones and joints. Macroscopic behavior such as mechanical properties of fibers and tissues as well as microscopic mechanisms such as molecular-scale processes inside biological cells and changes in electric potential across muscle fiber membranes contribute to the overall, versatile human musculoskeletal system.

Understanding this system well allows to use observations to make predictions. Using observations and predictions of the musculoskeletal system, we can for example design safe assistive robotic devices. Such robotic devices, in the form of exoskeletons, can potentially support humans in strenuous, unhealthy tasks that pose high loads on the human skeleton. An example is the precise handling of heavy objects that can only be done by humans, which is required in various industries. Moreover, exoskeletons can help to restore muscle function in a rehabilitation therapy.

In order to develop models for such predictions, various choices have to be made. Relevant properties of the muscular system have to be identified. Based on a physiological understanding, essential relationships have to be selected. Phenomenological relations can be incorporated. Mathematical formulations and numerical algorithms have to be found.

Model formulations can be differentiated by how much they are based on biophysical insights compared to raw experimental observations. The following sections present two different approaches that use a relatively high proportion of experimental observations combined with some biophysically justified relations. The two approaches are compared in an experimental study of forearm movement.

The first of the two presented models completely relies on experimental data. The second model adds physiological knowledge at a high level. In the remainder of this thesis after the current chapter, this trend continues: More details of the functioning of the musculoskeletal system get included. More advanced models are introduced that have finer model resolutions.

2.1 Introduction

Actuated orthoses and prosthesis help rehabilitation patients to regain their ability to move arms and legs when muscles have lost their full function. The dysfunction can be a result of muscular or nervous diseases, e.g., after a stroke, or originate from amputation of parts of the limb [Kre02; Zha18].

Rehabilitation or prosthetic devices are firmly attached to the body. Powered actuators at the joints support or replicate the natural movements of the limb. Exoskeletons are similar devices that typically extend to a larger part of the human body. Apart from rehabilitation, they are used as assistive, haptic or teleoperation device. Details can be found in [Per07].

For the actuation to be supportive and helpful, the device has to determine the intended movement of the limb. If muscles are functioning at least residually, EMG signals can be captured from the skin surface. They can be interpreted to determine finely graduated levels of force.

For this purpose, mathematical models are required that, given EMG measurements, predict joint torques for the system of limb segments and muscles. Because of the variety of muscle characteristics among humans, such models have to be patient-specific in order to be safe and effective for the particular individual.

In the following study, two different approaches for formulating such models, A and B, are developed. Instances of these two models are parametrized for a particular healthy subject.

The specific task in this study is to predict movements of a human upper arm. The arm is flexed and extended under varying loads and with varying velocities. Measured EMG signals on the agonist and antagonist muscles are used to predict the torque in the elbow joint. Considering the application of a supportive orthosis or exoskeleton, the predicted torque value is the control input to the actuator at the joint.

Prior to online application of the models, an offline training-phase is carried out, where all required parameter values get identified for the subject. After the two computational models have been trained, we perform validation experiments and compare the output of the models with measurements from the real system.

The first model approach, A, is a non-parametric, data-driven model. It uses the captured information from the training phase to construct a map between input and output values. It is based on Gaussian Process Regression.

The second approach, B, uses biophysically informed models of individual muscles together with the kinematics of the overall system. This approach requires a set of subject-specific parameters which is determined in the training phase. The model is based on the commonly used Hill-type muscle model.

2.1.1 Related Works

Numerous experimental studies of flexion and extension of the upper arm with the aim to predict elbow torques can be found in the literature. The studies presented in the following all include a Hill-type muscle model; such a model is also present in approach B of the present study.

In [Ros99], an exoskeleton across the elbow joint on the forearm is used as a passive measurement device. Experiments with lifting weights are performed and EMG is captured. Two different models are compared with respect to their performance in predicting moments and, thus, their suitability for exoskeleton control. The first model is Hill-based, similar to model B in the study of this work. The second model is data-driven, as is model A in our study. However, the method is different, the authors use a neural network.

The study reveals that the neural network is easier to set up but only works for the space defined by the learning data set. The advantage of the Hill-based model is that it is universal and not task dependent. Further studies using neural networks to estimate muscle activations and elbow torques are presented by [Lin02] and [Son05].

The paper of [Ros01] focuses on an exoskeleton that supports the forearm in lifting heavy weights. A generic Hill-type model is the base for the model predictions. Different control strategies are investigated. A naturally feeling human machine interface is achieved when control input is taken from processed EMG measurements and moment feedback of the external load. This result is promising as it shows that neural control of exoskeletons is possible, even using non-customized models.

The goal and setup of all these studies is similar to the work presented in the following sections. Differences are, apart from different setups, that they use state-less Hill-type models instead of the Hill-based models in our study that are more advanced. Furthermore, they do not use subject specific parametrizations. Both improvements can lead to better predictions of the moments in the elbow.

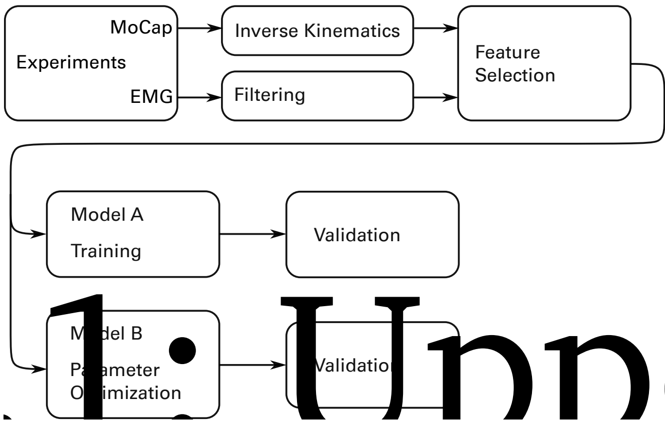
However, several studies predicting joint torques using Hill-type models exist in the literature that optimize model parameters to fit a specific subject. The authors of [Cav05; Cav06] study a scenario where a weight is lifted by the forearm. They use a genetic algorithm to find subject specific parameters to the models. Similar studies are given in [Llo03; Ven05; Pon09; Sar12].

A further study is performed by [Hei03]. The authors include models of activation dynamics, Hill-type muscle contraction and musculoskeletal geometry and restrict the scenario to isometric tasks. They optimize parameters for different subjects and determine the importance of parameters for good model predictions. It is found that the predictive quality of the model decreases with its complexity, but a model with seven parameters still has reasonable validity. In contrast to this study that only predicts static cases, our study also includes muscle dynamics and has more parameters to describe all required muscle properties.

[Fal16] estimate muscle model parameters of the knee joint actuators involving 23 degrees of freedom considering eight flexors and four extensors. Just like our study with model approach B, EMG signals and motion capture data are used to solve an optimization problem to fit the model. A difference is that three-element Hill-type models are used, whereas our study is based on more detailed, four-element Hill-type models but includes a smaller number of muscles.

Hill-based models of the muscle-tendon complex can also be parametrized without using EMG data. [Gar03] estimate characteristic parameters of 26 major muscles around shoulder, elbow and wrist in a two-phase optimization procedure. This approach uses individual experiments to identify different parameters. A method that requires fewer experiments is the ISOFIT method presented by [Wag05]. They use non-linear regression to fit Hill-type model parameters for various muscles from only 6-8 isovelocity contractions.

The authors of [Van14] develop a new method for estimating a subject-specific model of muscles around the knee which achieves higher accuracy than [Gar03] and is robust with respect to noisy data. Two improvements are that they use physiological constraints in the parameter optimization process and a heuristic for the initial guess of the parameters.


MoCap
Experiments
EMG
Inverse Kinematics
Filtering
Feature Selection
Model A
Training
Validation
Model B
Parameter Optimization
Validation

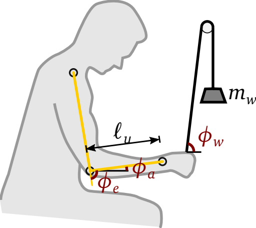


Figure 2.2: Upper Arm Movement Modeling:
Experimental setup for the triceps trials. The subject pulls down a rope over a pulley which is connected to a weight with mass m_w. Angles required for the kinematic formulation are the elbow angle, ϕ_e, the forearm angle, ϕ_a, and the angle of the weight, ϕ_w. The length of the ulna bone is denoted by ℓ_u.

especially filtering of EMG signals and feature selection, derivation and training of both models, A and B, their validation and the overall programming and visualization of the results. The respective fields are presented more detailed in the following, corresponding sections.

2.1.3 Structure of this Chapter

Section 2.2 gives an overview of the experiments, data processing and feature selection, which resulted in the required datasets. In Sec. 2.3, the two models, A and B, are described. Results including the validation of the models and a discussion are given in Sec. 2.4. Conclusions follow in Sec. 2.5.

2.2 Experimental Study

Experiments are required to identify the model parameters for the particular subject. First, the experimental setup is described, then, details on the processing of the measured values are given. Then, the selection of feature points from the experimental data is described.

2.2.1 Experimental Trials

In a series of experiments, eight different actions of flexing and extending the elbow were performed by the subject. Weights of 3 kg and 5 kg were held in the hand during the elbow flexion trials. For the elbow extension trials, a pulley system was installed that redirected the force of the weight such that the downward movement of the forearm acted against the direction of the force. This is shown in Fig. 2.2. A detailed description of the experimental trials can be found in [Wal20].

Time series of position and velocity of the upper arm and the forearm were recorded using a Motion Capture system. It consisted of eight cameras that tracked three markers placed on shoulder, elbow and wrist of the subject.

The elbow torque τ was computed as

$$\tau = m_w\, g\, \ell_u\, \sin(\phi_w) - m_a\, g\, \frac{\ell_u}{2}\, \sin(\phi_a),$$

where $(m_w\, g)$ is the force of the weight, m_a is the mass of forearm and hand, ℓ_u is the length of the ulna bone and ϕ_a and ϕ_w are the angles of the forearm and rope, as visualized in Fig. 2.2.

2.2.2 Data Processing

From the captured data, derived quantities of biceps (B) and triceps (T) muscles were estimated using a geometric model of the upper arm. The geometric model is available in the software OpenSim [Del07] and was customized for the particular subject. The inverse kinematics module of OpenSim was used to estimate the muscle tendon unit lengths, $\ell_{\text{MTU},M}$, contraction velocities, $v_M = \dot{\ell}_M$, and moment arms, r_M, of the two muscles, $M \in \{B, T\}$.

EMG signals were captured by two electrodes on the skin at the biceps and triceps muscles. For both signals, several preprocessing steps were applied to obtain the inputs for the two models, A and B.

The raw signal was filtered with the same procedure as in [Fal16]. First, a fourth-order Butterworth high pass filter with cutoff frequency 30 Hz was applied to reduce non-zero average voltages. Second, the resulting signal was full-wave rectified by taking the absolute value of every measured data point. Third, application of a fourth-order

Butterworth low pass filter with 10 Hz cutoff frequency yielded a smoothed signal. Forth, the resulting filtered EMG signals were normalized to the interval $[0,1]$, such that the value of 1 corresponds to the experimentally determined value of maximum voluntary contraction.

The measured EMG signals on the skin directly correspond to the electric excitation level u in the muscle. Excitation leads to the release of free calcium ions within the sarcomere. Binding of calcium ions to myosin increases the concentration of cross-bridges. This concentration is commonly known as the muscular activation α. The muscular activation directly corresponds to the produced force of the muscle [Bay17].

The concentration of free calcium ions is denoted as γ and can be computed from the excitation u by the following first order differential equation [Hat77]

$$\dot{\gamma} = m\,(u - \gamma).$$

We used the filtered EMG signal u to obtain values for γ. Figure 2.3 shows the raw and filtered EMG signals and the resulting free calcium concentration for a sample of the experimental data.

The activation of the muscle α does not only depend on the free ion concentration γ but also on the current state of muscle contraction. This excitation-contraction coupling has to be described by a dynamic system of ODEs and is included in model B. Therefore, preprocessing is completed with computing the free calcium concentration γ and not the activation α.

2.2.3 Feature Selection

The eight experimental trials were split into $n_{\text{trials}} = 7$ experiments to be used for model identification and one for validation. The total number N of captured values in the training experiments was large, such that not all points could be used for training of models A and B. To reduce the amount of data and, thus, speed up the computation, we selected $n \ll N$ featured values with the assumption that they are representative for the whole data set. For every experimental trial, we choose the same fixed number $n_{\text{per_trial}}$ of data points. Our selection algorithm identifies $n_{\text{per_trial}}$ timesteps, $t_i, i = 1, \dots, n$ such that the summed values of the free calcium concentrations $\gamma_B(t_i) + \gamma_T(t_i)$ for biceps and triceps are evenly distributed along the value range. This leads to $n = n_{\text{trials}} \cdot n_{\text{per_trial}}$ selected data points.

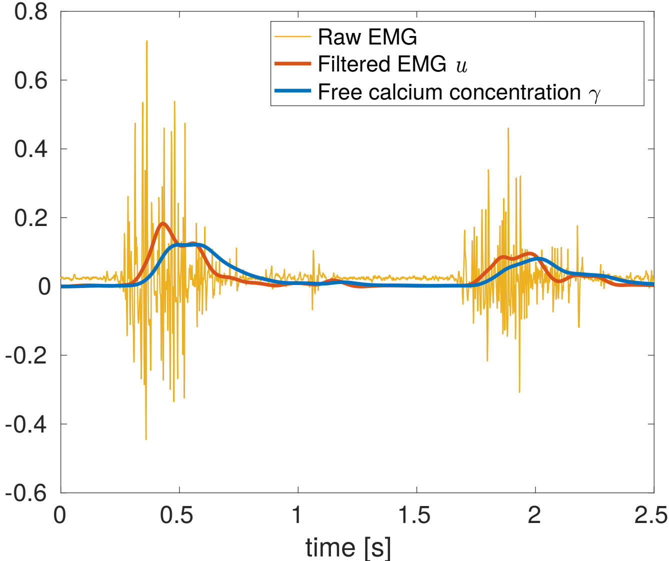


Figure 2.3: From raw EMG data of the biceps (yellow) to the filtered signal u (red) and the free calcium concentration γ (blue). The data are taken from the beginning of the first elbow flexion experiment. It can be seen that the filtering smooths out the initial signal and removes the constant offset. The free calcium ion concentration follows the filtered EMG with a short delay.

The set of training data $\mathcal{D} = \mathcal{X} \times \mathcal{Y}$ consists of the n selected vectors of the experimental values that are the input to the system of muscles, $\mathbf{x}_i \in \mathcal{X}, i = 1, \ldots, n$, together with the observed output values, $y_i \in \mathcal{Y}, i = 1, \ldots, n$. The input vectors contain values for muscle tendon unit lengths, contraction velocities, moment arms and free calcium ion concentrations for biceps and triceps each, $\mathbf{x}_i = (\ell_{\mathrm{MTU},B}, \ell_{\mathrm{MTU},T}, v_B, v_T, r_B, r_T, \gamma_B, \gamma_T)^\top(t_i)$. The output values consist of the elbow torques, $y_i = \tau(t_i)$. This data set, $\mathcal{D}$, serves as training input for both models, A and B.

2.3 Models

The current section describes the two model approaches that can predict elbow torques from experimental input data. Section 2.3.1 introduces the non-parametric, data driven

model A. Section 2.3.2 presents the biophysically based model B. It requires a parameter optimization, which is described in Sec. 2.3.3.

2.3.1 Data-driven Model A

The first modeling approach uses a non-parametric model. Such a model approximates the function f that maps from input to output data points. The function f is learned from the training data set. Regression is used to obtain predictions for new data points. In our case, we use a stochastic model that considers the probability distribution of the model function.

We use the method of *Gaussian Process Regression*. A Gaussian process is a collection of random variables such that the joint distribution of every finite subset of these random variables is multivariate normal (Gaussian). In our example, each input data point in the space of measured values, $\mathbf{x} \in \mathcal{X}$ has an associated random variable $f(\mathbf{x})$ that describes the output of the model for this point.

A Gaussian process, $\mathcal{GP}$, is characterized by a mean function $m(\mathbf{x})$ and a kernel function $k(\mathbf{x}, \mathbf{x}')$ that models the covariance between any pair $(\mathbf{x}, \mathbf{x}') \in \mathcal{X} \times \mathcal{X}$ of points. Different choices of kernel functions are possible and can depend on hyperparameters $\boldsymbol{\psi}$. Describing observed values y by a Gaussian process distribution can be expressed as

$$f(x) \approx y \sim \mathcal{GP}\big(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}', \boldsymbol{\psi})\big).$$

This representation is non-parametric in the sense that no particular parametric form of the function $y = f(\mathbf{x})$ is assumed whose (biophysical) parameters would be determined. Instead, a generic probabilistic model is constructed using the observed function values at measured inputs $\mathbf{x}_i \in \mathcal{X}$.

Gaussian Process Regression is based on *Bayesian Inference* to update a prior belief of the model to a posterior model using information contained in observations of the process. The observed data are the set of measurements $\mathcal{D}$.

The *prior* distribution $p(\mathbf{f} \mid \mathcal{X}, \boldsymbol{\psi})$ for the vector of function values $\mathbf{f}$ is described by the Gaussian process,

$$p(\mathbf{f} \mid \mathcal{X}, \boldsymbol{\psi}) = \mathcal{N}(\mathbf{f} \mid \mathbf{m}, \mathbf{K}),$$

with mean values $\mathbf{m} = (m(\mathbf{x}_i))^\top_{i=1,\dots,n}$ and covariance matrix $\mathbf{K}$ with $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j, \boldsymbol{\psi})$.

The *likelihood* $p(\mathbf{y} \mid f(\mathbf{x}), \boldsymbol{\theta})$ describes the probability of an observation $\mathbf{y}$ given a particular model f. The vector $\boldsymbol{\theta}$ denotes additional parameters of the likelihood.

Using Bayes' rule, the *posterior* distribution $p(\mathbf{f} \mid \mathcal{D})$ of the function values $\mathbf{f}$ can be computed from prior and likelihood as

$$p(\mathbf{f} \mid \mathcal{D}, \boldsymbol{\theta}, \boldsymbol{\psi}) = \frac{p(\mathbf{y} \mid \mathbf{f}, \boldsymbol{\theta})\, p(\mathbf{f} \mid \mathcal{X}, \boldsymbol{\psi})}{p(\mathcal{D} \mid \boldsymbol{\theta}, \boldsymbol{\psi})}.$$

This results in a measure for the uncertainty of the model f at unobserved points $\mathbf{x}_* \notin \mathcal{D}$.

Additionally, the fact that the measured quantities in the experiments are subject to measurement noise can be incorporated into the model. The assumption

$$y = f(\mathbf{x}) + \varepsilon$$

adds a normally distributed random variable of observational noise $\varepsilon \sim \mathcal{N}(0, \sigma_n^2)$ to the formulation. The noise variance $\theta = \sigma_n^2$ is an additional parameter of the likelihood. It is also possible to explicitly model the mean function $m(\mathbf{x})$. By replacing the model $f(\mathbf{x})$ by $g(\mathbf{x}) = f(\mathbf{x}) + \mathbf{h}(\mathbf{x})^\top \boldsymbol{\beta}$, i.e.

$$y = f(\mathbf{x}) + \mathbf{h}(\mathbf{x})^\top \boldsymbol{\beta} + \varepsilon, \qquad \text{with } f(x) \sim \mathcal{GP}\big(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}', \boldsymbol{\psi})\big), \quad \varepsilon \sim \mathcal{N}(0, \sigma_n^2),$$

we allow for a global trend in the data that is formulated in terms of a vector of explicit basis functions $\mathbf{h}(\mathbf{x})$ and corresponding coefficients $\boldsymbol{\beta}$.

The algorithm for Gaussian Process Regression involves estimating the following values from the given data during the training phase. The hyperparameters of the covariance function $\boldsymbol{\psi}$, the noise variance $\boldsymbol{\theta}$, and the coefficients of the fixed basis functions $\boldsymbol{\beta}$ are determined by solving an optimization problem. The computation involves matrix inversions and has a computational complexity $\mathcal{O}(n^3)$, i.e. is cubic in the number of data points. For details, the reader is referred to the literature [Ras05; Kus06].

In our study, training of the Gaussian Process of model A was performed using the ready to use implementation provided by MATLAB. We parametrized the covariance by a squared exponential kernel and used constant basis functions, $\mathbf{h}(x) = 1$. We enabled observational noise, its variance $\boldsymbol{\theta} = \sigma_n^2$ was found by optimization during training of the model.

2.3.2 Biophysical Model B

Extension of the elbow is governed by the triceps brachii muscle. During elbow flexion, three muscles are involved: biceps brachii, brachialis and brachioradialis. For simplicity, only biceps brachii, which contributes most of the moment, is explicitly considered in the current study. The effects of the other two muscle are contained in the biceps brachii model in a lumped manner.

Thus, the biophysical model consists of two Hill-type muscle models, for biceps and triceps, respectively. The muscle models are arranged around a hinge joint for the elbow angle. The muscle forces contribute to the torque at the elbow over their respective moment arms.

Hill-type models describe the macroscopic, dynamic mechanical behavior of an entire muscle along a one-dimensional line of action. The behavior is formulated by phenomenological, mathematical functions that have to be parametrized to fit experimental observations.

Multiple variants of Hill-type models exist that use various configurations of mechanical elements to consider different properties and functionalities of the muscle. The original model was proposed in [Hil38]. It contains a contractile element (CE) and two elastic elements, arranged in series and in parallel to the CE. The authors of [Sie08] compare two different approaches using these three elements. The effect of tension in eccentric contractions is added to the Hill-type model by [Til08]. The authors of [Gun07] add a forth, damping element to account for high-frequency damping of the muscle tissue. In [Mör12], electromechanical delay is investigated with and without the additional damping element.

We employ the four-element Hill-type muscle model that is described by [Hae14]. Its structure is visualized in Fig. 2.4. It consists of four components: the contractile element (CE), the parallel elastic element (PEE), the serial elastic element (SEE), and the serial damping element (SDE). Inputs to the model are the muscular activation $\alpha(t)$, the length $\ell_{\text{MTU}}(t)$ and the contraction velocity $\dot{\ell}_{\text{MTU}}(t)$ of the muscle tendon unit (MTU). The output of the model is the muscle force $f_{\text{MTU}}(t)$. The model contains one internal state variable, the length $\ell_{\text{CE}}(t)$ of the CE. The muscle dynamics determine this internal length and its time derivative, the contraction velocity $\dot{\ell}_{\text{CE}}(t)$ of the CE.

The resulting force of the MTU is given as sum of the forces of the respective parallel

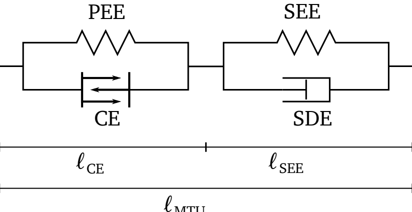


Figure 2.4: Mechanical structure of the Hill-type muscle model. The force generating contractile element (CE) is parallel-connected to the parallel elastic element (PEE) and connected in series to a second parallel-connected structure consisting of the serial elastic element (SEE) and the serial damping element (SDE). The length ℓ_{MTU} of the whole muscle tendon unit is composed of the common length ℓ_{CE} of CE and PEE and the common length ℓ_{SEE} of SEE and SDE. The variable ℓ_{CE} is an internal state of the model.

elements as visualized in Fig. 2.4:

$$F_{MTU} = F_{CE}(\ell_{CE}, \dot{\ell}_{CE}, \alpha) + F_{PEE}(\ell_{CE}) = F_{SEE}(\ell_{CE}, \ell_{MTU}) + F_{SDE}(\ell_{CE}, \dot{\ell}_{CE}, \dot{\ell}_{MTU}, \alpha). \quad (2.1)$$

The force terms of the four elements, F_{CE}, F_{PEE}, F_{SEE} and F_{SDE} are described by analytical functions that use a total of 19 parameters. A description of the detailed equations and parameters can be found in [Hae14]. In the following, an overview over the formulation is given with a focus on the piecewise formulated terms that contribute to the overall muscle model. In the following formulations, underlined variables designate constant parameters that either have to be specified or follow from other given parameters.

The muscle output force F_{MTU} is computed by the second identity of Eq. (2.1), i.e., from the forces F_{SEE} and F_{SDE}. The force F_{SEE} acting in the SEE is formulated as a continuous piecewise function with a constant zero, an exponential and a linear branch:

$$F_{SEE}(\ell_{CE}, \ell_{MTU}) = \begin{cases} 0, & \ell_{SEE} < \underline{\ell_{SEE,0}} \\ \underline{K_{SEE,nl}}(\ell_{SEE} - \underline{\ell_{SEE,0}})^{\underline{\nu_{SEE}}}, & \ell_{SEE} < \underline{\ell_{SEE,nll}} \\ \underline{\Delta F_{SEE,0}} + \underline{K_{SEE,l}}(\ell_{SEE} - \underline{\ell_{SEE,nll}}), & \ell_{SEE} \geq \underline{\ell_{SEE,nll}} \end{cases}, \quad \text{with } \ell_{SEE} = \ell_{MTU} - \ell_{CE}.$$

The damping force F_{SDE} in the SDE is proportional to the lengthening velocity $\dot{\ell}_{SEE} = \dot{\ell}_{MTU} - \dot{\ell}_{CE}$

of this element. It is given by

$$
\begin{aligned}
&F_{\mathrm{SDE}}(\ell_{\mathrm{CE}}, \dot{\ell}_{\mathrm{CE}}, \dot{\ell}_{\mathrm{MTU}}, \alpha) \\
&\quad = \underline{D_{\mathrm{SDE,max}}} \left((1 - \underline{R_{\mathrm{SDE}}}) \frac{F_{\mathrm{PEE}}(\ell_{\mathrm{CE}}) + F_{\mathrm{CE}}(\ell_{\mathrm{CE}}, \dot{\ell}_{\mathrm{CE}}, \alpha)}{\underline{F_{\mathrm{max}}} + \underline{R_{\mathrm{SDE}}}} \right) (\dot{\ell}_{\mathrm{MTU}} - \dot{\ell}_{\mathrm{CE}}).
\end{aligned}
\tag{2.2}
$$

The amount of damping is dependent on the force F_{MTU} of the MTU which appears in the nominator of the fraction in Eq. (2.2) as the sum of the forces F_{PEE} and F_{CE}. Formulas for these two forces are given in the following.

The force F_{PEE} of the PEE is formulated piecewise as a shifted and cut off polynomial function:

$$
F_{\mathrm{PEE}}(\ell_{\mathrm{CE}}) = \begin{cases} 0, & \ell_{\mathrm{CE}} < \underline{\ell_{\mathrm{PEE},0}} \\ \underline{K_{\mathrm{PEE}}}(\ell_{\mathrm{CE}} - \underline{\ell_{\mathrm{PEE},0}})^{\underline{\nu_{\mathrm{PEE}}}}, & \ell_{\mathrm{CE}} \geq \underline{\ell_{\mathrm{PEE},0}} \end{cases}. \tag{2.3}
$$

The force F_{CE} of the CE is the active force produced by the muscle and is given by:

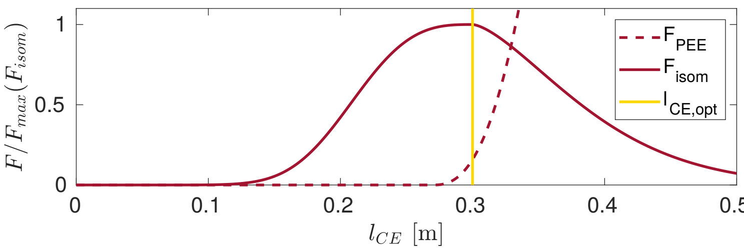
$$
F_{\mathrm{CE}}(\ell_{\mathrm{CE}}, \dot{\ell}_{\mathrm{CE}}, \alpha) = \underline{F_{\mathrm{max}}} \left(\frac{\alpha\, F_{\mathrm{isom}}(\ell_{\mathrm{CE}}) + A_{\mathrm{rel}}(\dot{\ell}_{\mathrm{CE}}, \ell_{\mathrm{CE}}, \alpha)}{1 - \dfrac{\dot{\ell}_{\mathrm{CE}}}{B_{\mathrm{rel}}(\dot{\ell}_{\mathrm{CE}}, \ell_{\mathrm{CE}}, \alpha)\, \ell_{\mathrm{CE,opt}}}} \right) - A_{\mathrm{rel}}(\ell_{\mathrm{CE}}, \ell_{\mathrm{CE}}, \alpha). \tag{2.4}
$$

It can be seen that the active force depends on the activation level α. The formulation of F_{CE} contains the two main characteristic curves for muscle forces, the force-length relation and the force-velocity relation.

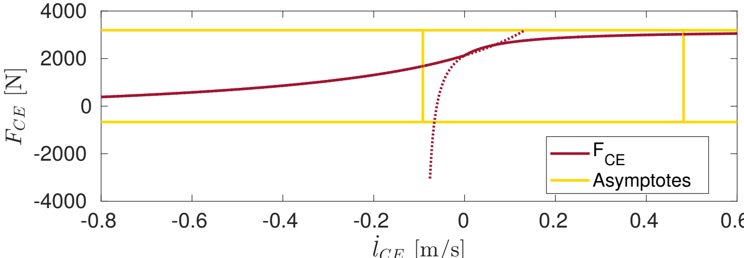
The force-length relation is modeled by the function $F_{\mathrm{isom}}(\ell_{\mathrm{CE}})$ of isometric force, which describes the relative force for the condition $\dot{\ell}_{\mathrm{CE}} = 0$. This function is formulated piecewise for CE lengths ℓ_{CE} smaller and larger than an optimal length $\ell_{\mathrm{CE,opt}}$.

The force-velocity relation follows from the auxiliary functions $A_{\mathrm{rel}}(\dot{\ell}_{\mathrm{CE}}, \ell_{\mathrm{CE}}, \alpha)$ and $B_{\mathrm{rel}}(\dot{\ell}_{\mathrm{CE}}, \ell_{\mathrm{CE}}, \alpha)$. These functions have different forms for concentric ($\dot{\ell}_{\mathrm{CE}} < 0$) and eccentric ($\dot{\ell}_{\mathrm{CE}} \geq 0$) conditions as well as for the two ranges of CE length, $\ell_{\mathrm{CE}} < \ell_{\mathrm{CE,opt}}$ and $\ell_{\mathrm{CE}} \geq \ell_{\mathrm{CE,opt}}$.

Figure 2.5 visualizes the two main characteristic curves of the model. Figure 2.5a shows how the generated force depends on the length of the CE. The active force F_{CE}, given by Eq. (2.4), is visualized by the solid red line. It has its maximum at the optimal length



(a) Force-length curves of the PEE (dashed red line) and isometric force F_{isom} (solid red line) for an isometric condition ($\dot{\ell}_{\text{CE}} = 0$), normalized to the maximum isometric force. The optimal length $\ell_{\text{CE,opt}}$ of the CE is shown as yellow vertical line. The force $F_{\text{PEE}}(\ell_{\text{CE}})$ of the PEE is zero for $\ell_{\text{CE}} < 0.9\,\ell_{\text{CE,opt}}$. The isometric contraction force F_{isom} is formulated piecewise by two branches separated by $\ell_{\text{CE,opt}}$.



(b) Force-velocity curve $F_{\text{CE}}(\dot{\ell}_{\text{CE}})$ of the CE at optimal length $\ell_{\text{CE}} = \ell_{\text{CE,opt}}$, and for activation level $\alpha = 0.5$. The function (red solid line) is formulated piecewise, graphs of the base functions of the two branches continue as red dotted lines. Their limits and singularities are visualized by the yellow horizontal and vertical asymptotes.

Figure 2.5: Force-length and force-velocity relations for the muscle model with generic parameters taken from literature [Hae14].

$\ell_{\text{CE,opt}}$ of the CE. This can be explained by the overlap of actin and myosin filaments in the sarcomere. The overlap is lower when the actin filaments are pulled apart or pushed together. A higher overlap leads to a higher force output.

The dashed line in Fig. 2.5a represents the passive force F_{PEE} of the elastic muscular tissue, formulated in Eq. (2.3). The passive force is essentially generated by the titin proteins in the sarcomere. Only starting from a certain length, the structure exerts reaction forces against lengthening forces to avoid overstretching of the muscle.

Fig. 2.5b shows the force-velocity relation of the Hill-type model. The curve of $F_{\text{CE}}(\dot{\ell}_{\text{CE}})$ is composed of two branches: The concentric branch for shortening contraction with $\dot{\ell}_{\text{CE}} \leq 0$ and the eccentric branch for lengthening contraction with $\dot{\ell}_{\text{CE}} > 0$. It can be seen that the generated force increases monotonically over the lengthening velocity. It approaches a limit for maximum positive and negative velocity. These limits can be adjusted by parameters of the model and are exemplary for how the shape of the curves of a Hill-type model can be parametrized.

In addition to the resulting muscle force F_{MTU}, a formulation for the internal state variable ℓ_{CE} is required. The second identity of (2.1) can be solved for the lengthening velocity $\dot{\ell}_{\text{CE}}$ of the CE to get an evolution equation for the length ℓ_{CE} of the CE. The derivation and the resulting formula can be found in [Hae14].

To describe the activation dynamics, i.e., the evolution of the muscle activation $\alpha \in [0, 1]$, the model of Hatze et al. [Hat77] is used. The activation is computed depending on the free calcium ion concentration γ and the length ℓ_{CE} of the CE by

$$\alpha(\ell_{\text{CE}}, \gamma) = \frac{\underline{a_0} + \big(\rho(\ell_{\text{CE}})\gamma\big)^3}{1 + \big(\rho(\ell_{\text{CE}})\gamma\big)^3}\,.$$

The function ρ is given by

$$\rho(\ell_{\text{CE}}) = \underline{c}\,\eta\,\frac{(\underline{k} - 1)\,\ell_{\text{CE}}}{(\underline{k} - \ell_{\text{CE}}/\ell_{\text{CE,opt}})\,\ell_{\text{CE,opt}}}\,.$$

All used parameter values for the activation dynamics can be found in [Bay17].

In summary, we get the following coupled system of differential-algebraic equations,

where f_{CE} and f_α denote the respective formulas:

$$F_{\text{MTU}} = F_{\text{MTU}}(\ell_{\text{MTU}}, \ell_{\text{CE}}, \dot{\ell}_{\text{CE}}, \alpha), \tag{2.5}$$

$$\dot{\ell}_{\text{CE}} = f_{\text{CE}}(\ell_{\text{CE}}, \ell_{\text{MTU}}, \dot{\ell}_{\text{MTU}}, \alpha), \tag{2.6}$$

$$\alpha = f_\alpha(\gamma, \ell_{\text{CE}}). \tag{2.7}$$

To compute the joint torque in a system of an agonist and antagonist muscle pair, two instances of the presented Hill-type muscle model can be used. In our study considering the upper arm, the torque τ at the elbow is computed by multiplying the predicted forces $F_{\text{MTU},B}$ and $F_{\text{MTU},T}$ of biceps and triceps with the corresponding moment arms $\hat{r}_B$ and $\hat{r}_T$:

$$\tau = F_{\text{MTU},B}(\ell_{\text{MTU},B}, \dot{\ell}_{\text{MTU},B}, \alpha_B) \cdot \hat{r}_B - F_{\text{MTU},T}(\ell_{\text{MTU},T}, \dot{\ell}_{\text{MTU},T}, \alpha_T) \cdot \hat{r}_T. \tag{2.8}$$

2.3.3 Parameter Identification for Model B

The process of model identification finds the parameters that make the model B predict correct values for the specific subject, i.e., minimizes the error in the predicted outcome for the training data set.

The following minimization is performed:

$$\min_{\substack{\boldsymbol{\theta}_M, \ell_{\text{CE},M}(t), \\ \forall M \in \{B,T\}, \forall t \in \mathcal{T}}} \sum_{t \in \mathcal{T}} |\tau(t) - \hat{\tau}(t)|^2 \tag{2.9}$$

$$\text{s.t. } \forall t \in \mathcal{T}: \quad \tau(t) = F_{\text{MTU},B}(t, \ell_{\text{CE},B}, \boldsymbol{\theta}_B) \cdot \hat{r}_B(t) - F_{\text{MTU},T}(t, \ell_{\text{CE},T}, \boldsymbol{\theta}_T) \cdot \hat{r}_T(t), \tag{2.10}$$

$$\dot{\ell}_{\text{CE},M}(t) = \dot{\hat{\ell}}_{\text{MTU},M}(t), \quad M \in \{B, T\}, \tag{2.11}$$

$$\boldsymbol{\theta}_B, \boldsymbol{\theta}_T \in \Theta, \tag{2.12}$$

$$\ell_{\text{CE},M}(t) \in [0, \ell_{\text{MTU},M}(t)], \quad M \in \{B, T\}. \tag{2.13}$$

The optimization variables are the parameters $\boldsymbol{\theta}_B$ and $\boldsymbol{\theta}_T$ for the biceps and triceps Hill-type models and the lengths $\ell_{\text{CE},B}(t)$ and $\ell_{\text{CE},T}(t)$ of the contractile elements for both

models at every point in time. The variables designated as $\hat{\square}$ are the measured quantities from the training experiments. The objective function given in Eq. (2.9) penalizes the difference between computed torque τ and measured torque $\hat{\tau}$ at every timestep $t \in \mathcal{T}$ of the training data.

Equation (2.10) computes the torque values and follows from Eq. (2.8) of the muscle model. For every point in time, the predicted forces $F_{\text{MTU},B}$ and $F_{\text{MTU},T}$ are multiplied with the measured moment arms $\hat{r}_B$ and $\hat{r}_T$.

In Eq. (2.11), the contraction velocities are constrained to the measured values. Because the lengthening velocity $\dot{\ell}_{\text{CE}}$ of the CE is an internal quantity and, thus, cannot be observed in experiments, we assume it to be equal to the lengthening velocity of the whole muscle: $\dot{\ell}_{\text{CE}} \approx \dot{\ell}_{\text{MTU}} = \dot{\ell}_{\text{CE}} + \dot{\ell}_{\text{SEE}}$. This requires the assumption $\dot{\ell}_{\text{SEE}} \approx 0$ which can be justified given the low dynamic nature of the experiments.

By Eq. (2.12), we bound each of the parameters $\boldsymbol{\theta}_B$ and $\boldsymbol{\theta}_T$ to a range between half and twice the generic value from literature. The lengths of the CEs are constrained by Eq. (2.13) to be positive and smaller than the length of the MTU.

All optimization variables are normalized to improve the numerical conditioning of the optimization problem. The parameters $\boldsymbol{\theta}_B$ and $\boldsymbol{\theta}_T$ are normalized with respect to generic values from literature that were taken from [Gun07; Mör12; Hae14]. The initial values are set to one, which corresponds to the generic literature values. The internal states $\ell_{\text{CE},B}$ and $\ell_{\text{CE},T}$, are normalized with respect to the measured MTU lengths $\hat{\ell}_{\text{MTU},B}$ and $\hat{\ell}_{\text{MTU},T}$ and initialized with zero.

We implemented the Hill-type models and the constraints in MATLAB and used the nonlinear programming implementation, `fmincon`, to minimize the given bounded and nonlinear constrained, multivariable function. In our study, the total number of optimization variables is computed by $2 \cdot 19 + 2n = 598$, as each of the parameter vectors $\boldsymbol{\theta}_B, \boldsymbol{\theta}_T$ had 19 entries.

2.4 Results and Discussion

In the following, results of connecting the two model formulations, A and B, to the experimental data are presented. At first, Sec. 2.4.1 gives details on the preprocessed data. The training phase is described in Sec. 2.4.2. Applying the trained models to the validation data is done in Sec. 2.4.3. Then, Sec. 2.4.4 tests a simplified version for model

A. Then, Sec. 2.4.5 shows some insights into the optimized parameter values for model B.

2.4.1 Feature Selection

The experimental data is split into a training and a validation dataset. Figure 2.6 shows the processed data of the training data set. In total, we captured $N = 34\,934$ data points for the seven experimental trials. Out of these, we select $n_{\text{per_trial}} = 40$ feature points in every trial, leading to a total of $n = n_{\text{per_trial}} \cdot n_{\text{trials}} = 280$ points. The selected points are visualized by crosses in the top plot of Fig. 2.6. It can be seen that the algorithm described in Sec. 2.2.3 distributes the feature points equally along the γ axis.

2.4.2 Training of the Models

Training of model A consists of estimating the hyperparameters for the Gaussian Process Regression model from the training data.

For model B, the optimization problem for the biophysical parameters is solved. The resulting parameter values and their relation to the initial values are summarized in Tab. 2.1. It can be seen that none of the final parameter values is limited by the constraints, which would be $-50\,\%$ and $+100\,\%$. However, it was observed that including the constraints helps the optimizer to stay in the valid range of meaningful model parameters and, thus, reach the optimum faster.

After training of the models A and B using the selected points of the training dataset, both models were tested by a *resubstitution prediction*, i.e., predicting output from the training input data. The results are shown in Fig. 2.7a for model A and Fig. 2.7b for model B. As this evaluation only uses the subset of selected experimental values, the data points have no natural ordering. They were sorted for better visibility.

It can be seen that, for both models, the predicted values are a good fit to the measured values. For model A, the predicted 95% confidence interval includes the actually measured values almost everywhere. For model B, the predicted values show a higher variance, especially for high torque values.

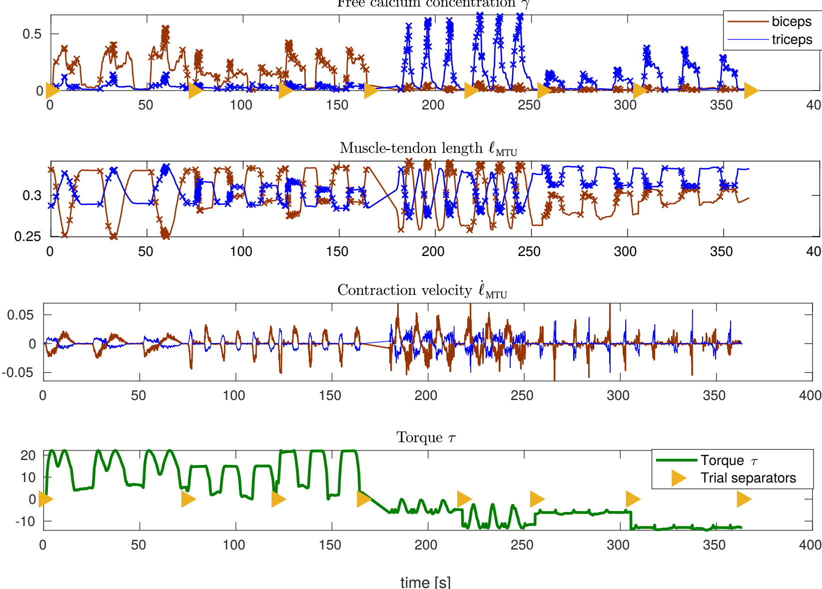


Figure 2.6: Processed experimental values over time that were used for training of both models. The concatenated data of seven trials are shown, which yield an end time of 363.32 s. The individual trials are separated by the yellow triangles on the x-axis. The three upper plots show the values of $\gamma, \ell_{\text{MTU}}$ and $\dot{\ell}_{\text{MTU}}$ for both biceps (brown) and triceps (blue), the bottom plot shows the elbow torque τ. The selected feature points are visualized as crosses in the two top plots. In the upper-most plot, it can be seen that the first three trials, which correspond to elbow flexion, mainly activated the biceps muscle, whereas in the last four trials, corresponding to elbow extension, the triceps is more active.

CE	$F_{\max}$ [N]	$\ell_{\text{CE},opt}$ [m]	ΔW_d []	ΔW_a []	$\nu_{\text{CE},d}$ []
Generic	4260	0.3	0.35	0.35	1.5
Biceps	+11.0 %	+31.7 %	+10.9 %	+91.6 %	+10.9 %
Triceps	−49.0 %	−25.4 %	+10.9 %	+5.1 %	+10.9 %

CE	$\nu_{\text{CE},a}$ []	$A_{\text{rel},0}$ []	$B_{\text{rel},0}$ []	S_{ecc} []	F_{ecc} []
Generic	3.0	0.25	2.25	2	1.5
Biceps	−46.4 %	+14.0 %	+77.5 %	−4.1 %	−30.1 %
Triceps	+95.9 %	−20.3 %	+41.4 %	+22.3 %	+36.8 %

PEE	$L_{\text{PEE},0}$ []	ν_{PEE} []	F_{PEE} []
Generic	0.9	2.5	2.0
Biceps	+10.9 %	+10.9 %	+10.9 %
Triceps	+10.9 %	+10.9 %	+10.9 %

SDE	D_{SDE} []	R_{SDE} []
Generic	0.3	0.01
Biceps	+10.9 %	+8.6 %
Triceps	+10.9 %	−11.0 %

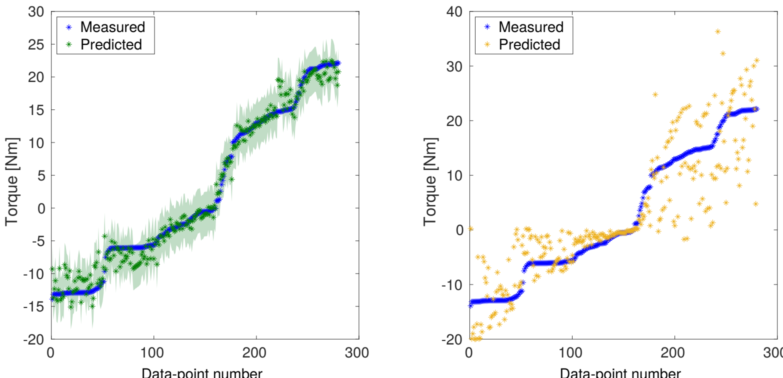
SEE	$\ell_{\text{SEE},0}$ [m]	$\Delta F_{\text{SEE},0}$ []	ΔU_{l} []	ΔU_{nll} []
Generic	0.172	0.0425	0.017	568
Biceps	−25.0 %	−42.4 %	+63.3 %	+59.2 %
Triceps	−10.3 %	+64.75 %	+19.0 %	+23.6 %

Table 2.1: Hill-type muscle model parameters of the four elements: CE, PEE, SDE and SEE, initial values given in literature and relative changes of the optimized values. Further explanations of the parameters and references to literature containing their initial values are given in [Hae14].

2.4.3 Validation

The next evaluation uses the validation dataset and compares the predicted outputs of the models with the actual experimental values. In contrast to the training data, where a small number n of points was selected, we now use all captured values. This involves a total of $54 \cdot 10^3$ data points for a time span of $t = 54\,\text{s}$.

The results are shown in Fig. 2.8. Comparing the green curve for model A with the blue curve for the experimental data, it can be seen that the predicted values match qualitatively for most of the time span. The predicted torque values appear consistently slightly smaller than the real values. Only for the two intervals [11 s, 12 s] and [40 s, 42 s] the predicted value is far off. The 95 % confidence interval that was computed by the



(a) Measured values (blue) and values predicted by model A (dark green), with 95% confidence interval (light green).

(b) Measured values (blue) and values predicted by model B (orange).

Figure 2.7: Resubstitution prediction: Measured and predicted torque values for the training data set. The measured points are ordered and numbered by magnitude, the order of the predicted points matches the order of the measured points.

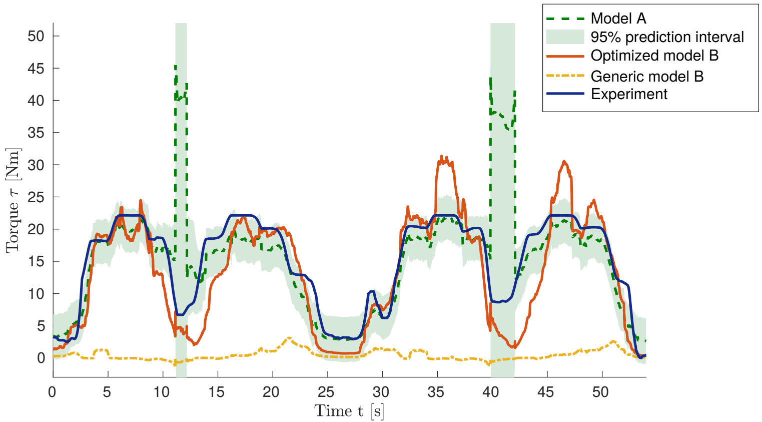


Figure 2.8: Validation: Predicted torque values by model A (green), trained model B (red) and untrained model B (yellow), in comparison to the experimentally measured values (blue), for the validation data set.

Gaussian Process spans a large range for these time intervals which implies that the model prediction is not to be trusted for this area.

The biophysical model approach, model B, was tested in two variants. First, with the generic parameters from literature (yellow curve), second, with the subject-specific, optimized parameters (red curve). It can be seen that the generic model fails to predict the torque values whereas the trained model predicts reasonable values. These values are worse than most of the predictions from model A, but they succeed in giving a qualitative estimate about a low, medium or high torque output.

The match between model outputs τ_i and experimental data $\hat{\tau}_i$ can be quantified using the normalized root-mean-square error (NRMSE). This is a scaled version of the root-mean-square error (RMSE) and can be defined as

$$\text{RMSE} = \sqrt{\sum_{i=1}^{N}(\tau_i - \hat{\tau}_i)^2/N},$$

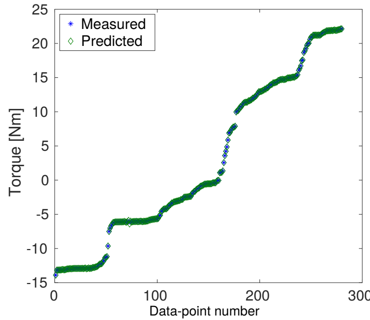
$$\text{NRMSE} = \frac{\text{RMSE}}{\max_i\{\hat{\tau}_i\} - \min_i\{\hat{\tau}_i\}}.$$

The NRMSE for model A is 0.267 which is worse than the value of 0.163 for the trained model B. The generic model B has the worst NRMSE of 0.547.

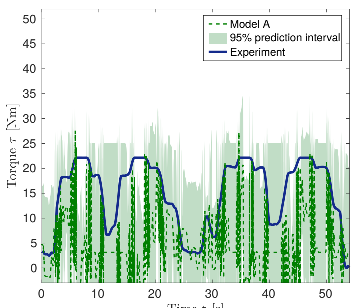
2.4.4 Simplified Model A

An advantage of model approach A is that it forgoes any biophysical description and the associated type of model error. It is a generic approach that does not require expert knowledge about the physiological structure. In the present study, however, some level of expert knowledge and physiological model was required in preprocessing the MoCap data, i.e. solving the inverse kinematics of the observed forearm movements to get the kinematic quantities of muscle lengths, velocities and moment arms.

Since model A performed well in the previous validation study, we tested whether good results can also be achieved without this expert knowledge. Consequently, the next study applies model approach A using only the elbow angle and no muscle lengths, velocities nor moment arms. Thus, the training data consists of input vectors $\mathbf{x}_i = (\phi_e(t_i), \gamma_B(t_i), \gamma_T(t_i))^\top \in \mathcal{X}$. In the following, this model is named "simplified model A" in contrast to the "full model A" that uses the complete set of input variables.



(a) Measured and predicted torque values of the training dataset, ordered and numbered by magnitude. The measured values (blue) and the values predicted by the Gaussian Process (green) lie on each other.



(b) Predicted torque values for the validation trials (green dotted line), 95% confidence interval (light green) and the reference values of the experiment (blue). The plot reveals bad prediction capabilities of the simplified model A.

Figure 2.9: Result for the simplified model A, where, apart from the free calcium ion contractions, only the elbow angles, ϕ_e, are used as training input instead of MTU lengths, velocities and moment arms.

The results are shown in Fig. 2.9. It can be seen that the resubstitution prediction in Fig. 2.9a where the trained model is used to predict the training values shows a perfect fit. In contrast to the full model A, Fig. 2.7a, here, the learned input-output mapping shows no variance. However, the prediction for the validation dataset in Fig. 2.9b shows a high error relative to the experimental data. The curve for the experimental data even lies outside the 95% confidence interval of the prediction at some points.

The simplified model A has a NRMSE value of 0.461. For comparison, the NRMSE values of the full and simplified model A and the generic and optimized model B are summarized in Fig. 2.11.

This evaluation shows that simplified model A gives no useful results where the training input is too scarce. Instead, preprocessing of the measurements using a subject specific geometric model, as done for the full model A, is needed to allow for a useful prediction.

2.4.5 Insights of Model B

An advantage of model approach B is that the trained parameters are physically meaningful and allow insight into the properties of the subject specific model. Furthermore, the quality of the training data can be assessed. Figure 2.10 shows the force-length relation of the biceps muscle model using the generic and the optimized parameter values. It can be seen that the subject-specific model has a smaller slope of the force curve. All points of the training data set are indicated by red crosses on the curves and show the operating range of the muscle in which the model has been trained. It can be seen that the experimental training data are limited to a small range of the muscle length below its optimal CE length $\ell_{\text{CE,opt}}$. In order to improve the quality of the model predictions for this subject, specific additional experimental trials can be designed for model training. They can be designed to fill in values in the missing range of operation, which in this case is for larger muscle extensions.

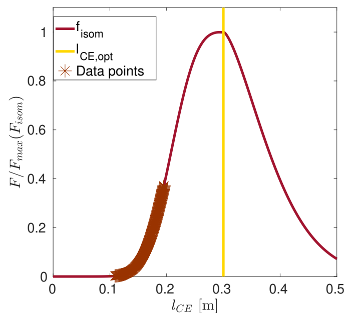
A low computational time of the offline parameter identification and the online evaluation of the two models would be an important measure for their practical applicability. In the present study, the training phase of Model A, i.e., optimization of the quantities for the Gaussian Process Regression using 280 training data points took 2.24 s. The evaluation of Model A for the validation data set containing $54 \cdot 10^3$ points had a duration of 116 ms.

The runtimes for model B were significantly higher. The parameter optimization lasted 25 min 16 s and the evaluation for the validation data set had a duration of 13 s.

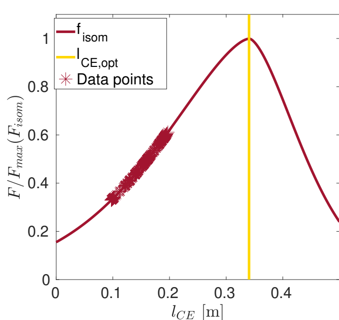
The large differences in runtime between models A and B can be explained by the inefficient implementation of the biophysical model using the MATLAB programming language. During parameter identification, this model needs to be evaluated iteratively in the optimization algorithm. In contrast, the optimization within model A works with an internal implementation of the Gaussian Process which was optimized during development of the particular MATLAB functionality. In general, the evaluation of Gaussian Process Regression has cubic time complexity whereas, for the parameter optimization of model B, iterative solvers with linear time complexity exist.

2.5 Conclusion

In this study, elbow torques during flexion and extension of the upper arm were predicted from motion capture data and EMG measurements. Two models, A and B, were developed.



(a) Model with generic parametrization.



(b) Model with subject-specific parametrization.

Figure 2.10: Isometric force-length relation of the CE for the biceps model, analogue to Fig. 2.5a, but additionally with training data points. The points are placed on the model curve and visualize the predicted relative forces for the lengths of the CE that occurred during the training trials.

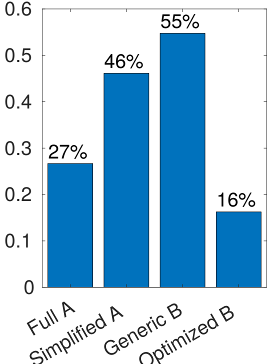


Figure 2.11: Normalized Root Mean Square errors (NRMSE) of the validation trials between the respective models and the measured values. A lower error value means a better fit.

Model A is non-parametric and uses Gaussian Process Regression. Model B is biophysically informed and involves two state-of-the-art Hill-type muscle models for biceps and triceps. Experiments were conducted to generate training and validation data. These training data were used for model parameter identification. Predictions from the two models were compared to real experimental values using the validation data.

Regarding the formulation and implementation, model A requires low effort and no special knowledge about the model, except where experimental motion data is preprocessed for a specific subject. In contrast, model B needs expert knowledge about the biophysical structure and the implementation of all comprised models.

Similar holds for the offline training phase. There are no parameters in model A that have to be tuned manually, which allows a quick start. Conversely, model B requires the appropriate definition of initial values and physiological constraints for the optimization problem. However, this can also be seen as an advantage for model B, as a-priori knowledge can be integrated in such a model.

On the other hand, an advantage of model A is that additional experimental data, e.g., from neighboring muscles or additional sensors, can easily be added to the model. This is not possible with model B, where the model formulation would have to be changed.

Our studies showed that both models were able to predict the levels of torque reasonably. Figure 2.11 showed the best score for model A, followed by model B. It was also seen that the generic parametrization of model B does not yield a useful prediction. The same is true for a simplified version of model A, where the elbow torque was used as training input instead of derived quantities from the motion capture system that required a complex preprocessing step.

Both models provide possibilities to assess the confidence of their predictions. With model A, confidence intervals can be computed directly from the Gaussian Processes. Their usefulness was shown in the validation where regions with large errors also had a large confidence interval. Model B allows insight into force-length and force-velocity characteristics of the two involved muscles. The operating ranges of the muscles during the experiments can be visualized and allow assessing whether the desired model features were covered by the training phase and, thus, will yield a good prediction.

In our study, runtimes were low for model A and high for model B in both offline and online phases. However, this is due to our prototypical implementation of model B. For larger data sizes and a more sophisticated implementation, the reverse effect is expected. The runtime complexity for the training phase is better for model B (linear

in time) compared to model A (cubic in time). For the online phase, costly integration over data points is needed for model A whereas model B directly provides a differential equation of the system that can be solved efficiently.

If EMG is used to control an exoskeleton that supports the movement of the limb, it is known that the measured signals are ahead of the intended movement by a small offset. This is a result of the time delay in the neuromusculoskeletal system. This property gives the assistive exoskeleton a short time to predict the intended movement and thereby allows a seamless integration of the artificial device with human control.

When targeted at such a real-time application, both models could be considered to be integrated into the control. Model A better fits the use case of a device that could be (re-)calibrated by the patient itself. Because of the built-in estimation of prediction quality, compliance and safety could be ensured more easily even for imperfect training. Model B would need a controlled environment such as a specialist's laboratory and careful assistance for the calibration process. After calibration, it would promise a more natural and more responsive experience because of the subject-specific model and possibly smaller compute times.

Where real-time application is not a requirement, biophysically informed models have a high potential to leverage the understanding how the human neuromusculoskeletal system operates for given tasks. In model B of this study, the kinematics and individual muscle dynamics were described close to the current understanding of the system. However, several aspects where not modeled as detailed as possible. The pathway from neural stimulation to excitation and activation of the muscle, the recruitment strategies including different motor units, neural feedback loops as well as effects stemming from the 3D geometry of the muscle were not considered. Therefore, this thesis develops a more detailed, biophysically informed model including these properties in the following chapters.

The presented study reproduced what similar studies in literature have shown: Subject-specific model identification for Hill-based torque prediction models can vastly improve the prediction quality compared to generic models. Our work adds to the common knowledge that this holds also for the four-element Hill-type model that was used for model B. Furthermore, a comparison with Gaussian Process Regression was given, various advantages and disadvantages of these two approaches were identified. Future work can test the two models with more subjects and increase the variety of motion in the training experiments. For example, effects resulting from high contraction velocities or eccentric

contractions could be investigated to evaluate the model's potential in more complex movements.

Chapter 3

Generation of Meshes for the Multi-Scale Models

Multi-scale models of skeletal muscles describe phenomena on different length scales and combine them into a single description. The phenomena are modeled by different sets of equations which need individual discretizations and solvers. For that, various geometrical meshes describing different physical domains are required.

The discretization considered in this work involves three-dimensional (3D) and one-dimensional (1D) meshes. As a whole, muscles and tendons are treated as 3D domains. Muscle fascicles and myofibrils are represented by 1D fibers that are embedded in the 3D domain of the muscle.

The generation of the respective 1D and 3D meshes should be based on biomedical imaging data in order to represent actual human anatomy. The generated meshes should be of good quality such that finding numerical solutions with low error is possible. Good mesh quality usually involves mesh cells with similar lengths and angles. It should also be possible to easily partition the mesh into multiple, equally sized subdomains. This is required for efficient parallel computation. The two requirements of good mesh quality and easy partitioning lead to the decision to employ *hexahedral* elements and a *structured* mesh for the 3D domains.

In this chapter, we present a workflow to construct meshes with the mentioned properties starting from biomedical data. We present novel algorithms to generate the required structured hexahedral meshes. This work contributes an implementation of the algorithms that can be used to construct all meshes needed for our biomechanical simulations.

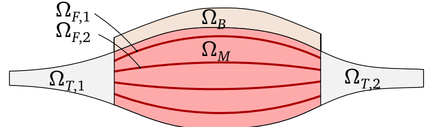


Figure 3.1: Visualization of the computational domains in a simulated muscle: tendons $\Omega_{T,1}, \Omega_{T,2}$, muscle belly Ω_M, body domain Ω_B and fiber domains $\Omega_{F,i}$.

3.1 Overview and Notation of Required Meshes

In the following, we summarize the meshes generated and used in this thesis and introduce their notation used in the following discussions.

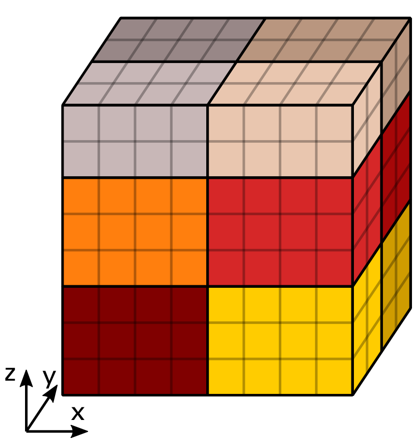
The domain of the muscle belly is denoted by Ω_M. A layer of fat and skin tissue is located on top of the muscle belly. It is denoted as the body domain Ω_B. The muscle belly is attached to tendons on both longitudinal ends. The tendon domains are denoted by $\Omega_{T,1}$ and $\Omega_{T,2}$. These domains are all subspaces of the 3D Euclidean space: $\Omega_M, \Omega_B, \Omega_{T,i} \subset \mathbb{R}^3$.

Additionally, a number n_f of individual muscle fibers $\Omega_{F,i} \subset \mathbb{R}^3$ for $i \in \{0, \dots, n_f\}$ is introduced. Each fiber is a 1D manifold embedded in the 3D domain, i.e., $\Omega_{F,i} \subset \Omega_M$. Figure 3.1 summarizes the notation of the domains.

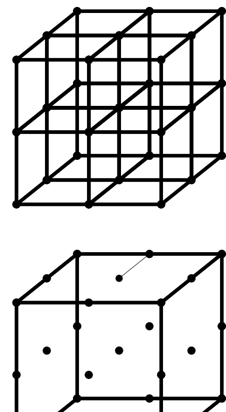
For the application of the finite element method (FEM), we create meshes for each of these domains. Formally, a 3D mesh Ω_{3D} is given by a number of 3D elements $\{U_{3D,i}\}_{i=1,\dots,n}$ with $U_{3D,i} \subset \mathbb{R}^3$ such that their disjoint union approximates the domain, $\dot{\bigcup}_{i=1}^{n} U_{3D,i} \approx \Omega_{3D}$. Similar holds for 1D meshes.

The elements are non-overlapping and can be defined by nodes and edges. In the discretizations used here, no hanging nodes are allowed, i.e., at any node all adjacent elements share the node.

Furthermore, only structured, hexahedral meshes are considered in this chapter. A 3D structured mesh is isomorphic to a 3D Cartesian grid with equidistant elements. This has advantages for programmatically indexing nodes and elements as well as for parallel partitioning of the domain. Figure 3.2a shows an example of a 3D structured mesh that is partitioned into twelve subdomains. The subdomains are constructed by planar cuts through the structure of the mesh. These cuts are typically defined in a way that the resulting subdomains have similar numbers of 3D elements and, thus, every process gets a similar portion of the total computational load.



(a) Parallel decomposition of a 3D mesh with $n_x \times n_y \times n_z = 8 \times 4 \times 8$ elements into $2 \times 2 \times 3 = 12$ subdomains.



(b) Top: a linear 3D mesh with eight elements, bottom: a single quadratic element, which uses the same nodes as the linear 3D mesh at the top.

Figure 3.2: Structured 3D meshes that are used in the simulations: parallel partitioning and construction of quadratic elements.

The number n of 3D elements is the product of the numbers n_i, n_j and n_k of elements in the three coordinate directions x, y and z of the Cartesian grid, i.e., $n = n_i\, n_j\, n_k$. Each element can be indexed by a triple (i, j, k) of indices with the ranges $i \in \{0, \dots, n_i - 1\}$, $j \in \{0, \dots, n_j - 1\}$ and $k \in \{0, \dots, n_k - 1\}$. In the simulation program, typically, consecutive indices ι are used that iterate over all elements $\iota \in \{0, \dots, n-1\}$ and are obtained from the index triples by the mapping $(i, j, k) \mapsto \iota = k\, n_i\, n_j + j\, n_i + i$.

The elements of such a mesh can have different numbers of *degrees of freedom (dof)* depending on the desired spatial order of consistency of the finite element discretization. The number $n_{\text{dofs},d\text{D}}$ of dofs in a d-dimensional element is computed from the number $n_{\text{dofs,1D}}$ of dofs along one coordinate direction of the element as $n_{\text{dofs},d\text{D}} = n_{\text{dofs,1D}}^d$. Consequently, linear elements have two dofs in 1D meshes and eight dofs in 3D meshes. Quadratic elements have three dofs in 1D and 27 dofs in 3D.

The dofs are located at the *nodes* of the elements. In linear and quadratic elements, every node corresponds to a single dof. While the nodes form the "corners" of linear 3D elements, they are also located on the faces and in the interior of quadratic 3D elements.

Figure 3.2b shows a mesh with $2 \times 2 \times 2$ linear 3D elements at the top. The same 27 nodes can be used to define a single quadratic 3D element as shown at the bottom of Fig. 3.2b.

It is sufficient to develop a method for constructing structured 3D meshes with linear elements. Higher order elements can be geometrically constructed by using the nodes of multiple adjacent linear elements. To generate both linear and quadratic elements, we always begin with generating a mesh with even numbers n_i, n_j and n_k of elements in the coordinate directions. Then, linear and quadratic meshes can be extracted from the set of nodes. Similarly, a linear 1D mesh with an even n_i can be easily converted into a quadratic 1D mesh.

The next sections describe workflows and algorithms to construct 3D meshes for the domains $\Omega_M, \Omega_B, \Omega_{T,i}$ and 1D meshes for the fibers $\Omega_{F,i}$ based on anatomical information. Section 3.2 gives an overview over available meshing software tools and existing algorithms in literature. Then, Sec. 3.3 presents a workflow to extract a smooth surface representation from anatomical imaging data. In Sec. 3.4, two serial algorithms are presented to generate 3D meshes and 1D fibers meshes. The next section, Sec. 3.5, extends these serial algorithms formulating a parallel algorithm and shows and discusses results. Finally, Sec. 3.7 gives a summary and concludes this chapter.

3.2 Related Work

Generating volumetric meshes for domains enclosed by a given surface is a task that is frequently needed in computational science. It is a preprocessing step whenever spatially discretized models have to be solved numerically. In consequence, a vast amount of literature has addressed this algorithmic task and various approaches and methods have been proposed. Moreover, numerous software packages that solve this problem exist. Especially tools for Computer Aided Design and Engineering (CAD/CAE) as well as free and commercial preprocessing tools and finite element solver software include functionality to generate meshes from given surfaces.

An example from the biomechanical domain is [Unt13]. The study develops a finite element model of the lower limb of an occupant of a car with the aim to investigate injury scenarios during traffic crashes. The lower extremity geometry was obtained by computer tomography (CT) and magnetic resonance imaging (MRI) scan data of a 50th percentile male volunteer. Different meshes of bones and ligaments were created using

the three tools IA-FEMesh [Gro09], TrueGrid [XYZ20] and Hyper-Mesh [Alt20] which will be outlined in the following.

IA-FEMesh (University of Iowa, Iowa City, USA) is an open source tool to generate hexahedral meshes [Gro09]. It provides an interactive environment where existing geometries can be loaded. In a visualization window, bounding boxes, called blocks, can be positioned such that they contain the whole geometry. A structured grid on the block is then projected onto the surface of the geometry. Multiple blocks can be placed to account for more complex geometries. The resulting surface mesh is improved using Laplacian smoothing which equalizes the edge lengths of the elements. The interior nodes are generated using interpolation. The result is a structured mesh if only one block is used or an unstructured mesh if multiple blocks are used. Further operations to manage mesh density, visually manipulate the meshes and add material properties, load and boundary conditions are available. The model can be exported in a file format for finite element analysis with ABAQUS (Dassault Systèmes, Vélizy-Villacoublay, France) [Das20].

The second tool is *TrueGrid* (XYZScientific Applications, Livermore, USA) [XYZ20]. It is a commercial toolkit to generate hexahedral meshes. The project was started in the early 1990s as the successor to the even older preprocessor software *INGRID*. Similar to *IA-FEMesh*, a projection method and a multi-block technique are used. Some effort has been put into dealing with holes and sewing together dissimilar blocks.

The third tool is *Hyper-Mesh*, the commercial pre-processing and post-processing toolkit of Hyperworks (Altair HyperWorks, Troy, USA) [Alt20]. Altair sells infrastructure and solvers for a multitude of physics and is targeted at a wide range of industries. Being a commercial vendor, information about the internals of their preprocessing software are hardly provided.

More meshing software exists, such as CGALmesh [Jam15] for tetrahedral meshes. The package gives quality guarantees of their generated meshes and includes four mesh optimization algorithms to further improve the mesh quality.

Another application-oriented work dedicated to the use of commercial tools is [Ram18]. A workflow for patient-specific modeling, simulation and analysis of the interaction between a residual lower limb stump and the socket of a prosthesis is presented. Imaging data were taken from magnetic resonance diffusion tensor imaging where also the preferred diffusion direction of water molecules along muscle fibers is captured. The open source tool MedInria (National Institute for Research in Digital Science and Technology (Inria), France) [Vic12] was used to extract muscle fibers. The residual limb data were processed using the commercial 3D image segmentation software Simpleware ScanIP

(Synopsys, Mountain View, USA). Auxiliary tasks were performed using MATLAB (MathWorks, Natick, USA) scripts. The commercial multiphysics solver LS-DYNA (LSTC/Ansys, Canonsburg, USA) was used for the simulations.

Commercial tools usually have the advantage that more development effort was put into them, than is possible for open source codes from the scientific community. This often leads to more robust and user-friendly software. An advantage of open source software is that the used algorithms are disclosed to everyone. They are often well documented or described in a publication. This allows to assess the expected quality of the generated meshes. Conversely, commercial vendors usually have no interest in revealing their internal algorithms.

For our simulation, structured, hexahedral meshes are needed. Several of the described tools are able to generate hexahedral meshes, however the meshes are typically unstructured. For our special need of 1D muscle fibers embedded in a 3D mesh, we develop our own method that is based on the ideas of existing algorithms. In the following, an overview over the algorithmic common knowledge of creating simplex meshes and hexahedral meshes is given as a basis.

Triangulating a 2D domain is the archetype of mesh creation. The triangulation named after B. Delaunay was formulated in 1934 [Del34]. For a given set of points, it maximizes the minimum angle of the triangles and, thus, avoids small angles. Therefore, a guarantee on the quality of the triangulation is given.

In 1995, J. Ruppert presented the Delaunay refinement algorithm [Rup95], which constructs a Delaunay triangulation conforming to prescribed connected points. This algorithm is still commonly used and also part of numerous derived meshing techniques.

In 1997, P. Chew developed an algorithm for meshing a 3D domain with tetrahedra [Che97] and proved that the aspect ratio of the tetrahedra is bounded, i.e., degenerate, "flat" tetrahedra, called slivers, are avoided.

The authors of [All05] propose a variational approach to triangulation where a quadratic energy function is minimized. During minimization both vertex positions and connectivity are optimized. This leads to better quality meshes than by simple Delaunay triangulations.

Hexahedral meshes can be obtained from certain tetrahedral meshes by splitting up each tetrahedron into four hexahedra. This is discussed in [Epp99]. A remaining issue is that the generated meshes from this procedure are highly unstructured and some

hexahedra have poor quality, whereas the goal would be to construct elements that are almost equilateral.

A different approach is to directly generate a hexahedral mesh for the given surface geometry. The survey in [Owe98] identifies four different strategies for generating unstructured hexahedral meshes.

The first one is a *grid-based* approach. It was introduced in [Sch96; Sch97]. The interior of a given solid is filled with a regular and Cartesian grid of as many hexahedral elements as fit into the space. Then, the gaps at the surface are filled with additional elements. This method is robust but can lead to poor quality elements near the surface.

The second approach for generation of hexahedral meshes are *medial surface methods* [Pri95; Pri97]. First, the volume is decomposed into subregions by medial surfaces such that the resulting domains are one of only 13 possible types. Predefined templates are used to fill the domains with hexahedral elements. Then, the continuity between the domains is ensured using linear programming. This approach gives good results for some geometries but has robustness issues when general geometries are considered.

The third approach is called *plastering*. It was first described by [Bla93] and continued by [Sta06; Sta10]. It is a moving-front method where hexahedral elements are placed in layers starting at the boundary and moving towards the interior. Intersection of faces has to be detected when the fronts meet in the interior and rules for connecting to existing faces have to be defined. During this process, complex shaped voids can occur in the interior. When it is no longer possible to fill the voids with hexahedra already placed elements have to be removed. A new method, called unconstrained plastering, starts from an unmeshed volume boundary. The approach has general robustness issues and is not guaranteed to find a solution for arbitrary boundaries.

The forth approach is *whisker weaving*, introduced by [Tau96] and extended by [Led08; Kaw08]. Here, the dual of the hexahedral mesh is considered. The dual consists of the three surfaces per hexahedron that lie in the planes of symmetry. The surfaces of all hexahedra form topological loops. The principle is now to first construct the dual of the mesh, which can be determined from the given boundary surface. Then, the actual hexahedral mesh is created from the dual, using the surfaces as guides where to place the elements. The dual forms topological loops inside the volume. One important criterion for generating good quality meshes is that self-intersections of these loops are resolved in a first step. The approach, used with subsequent smoothing, can produce meshes of good quality. However, no guarantee is given. One problem is that the resulting mesh

depends on the quality of the surface mesh and that the number of nodes can increase significantly during the method.

For the whisker weaving method and for some plastering methods, a quadrilateral mesh of the surface is required. Algorithms for creating high quality quadrangulations of closed surfaces exist [Don05; Kov11; Bes12; Men16].

Other approaches start with 3D volumetric medical imaging data instead of surfaces. In [Zha03; Zha05], adaptive tetrahedral and hexahedral meshes are created from volumetric data using octree subdivision. The method avoids hanging nodes and allows a feature sensitive adaptivity. While adaptive meshing methods can reduce the number of elements in the interior of the volume, a problem is that the worst quality elements are generated at the boundary, the location where the solution in a finite element study usually is most interesting.

Multiple reasons make the previously outlined approaches unsuited regarding the needs for our parallel muscle simulation.

(i) The generated meshes are unstructured. When performing domain decomposition for parallel computing on unstructured meshes, graph-partitioning methods have to be used. Storing an unstructured mesh requires storage of element adjacency information. Partitioned meshes additionally require storage of the adjacent processes. In contrast, structured meshes can be trivially decomposed and stored efficiently. A decomposition can be represented in memory by a very low number of parameters.

(ii) The presented methods are designed for hexahedral meshing of arbitrary volumes. Robustness and mesh quality at the same time remain issues that are not completely solved for most of the algorithms. Often, expensive smoothing steps are needed to increase mesh quality.

(iii) In general, either no assumption can be made about orientation or alignment of hexahedra in the interior, or, for the grid-based approach, the elements at the surface have poor quality. Having a mesh that is consistently aligned with, e.g., the main diffusion direction or the preferential direction of the anisotropic material or the muscle fibers can reduce numerical errors in the finite element solution.

Consequently, a more scenario specific solution is needed that can avoid the mentioned issues. Such solutions can also be found in the literature. An example is [Ble05b], where 3D finite element models for various complex muscle geometries around the hip are generated from magnetic resonance images. Segmentation and surface mesh generation are performed using the old, unmaintained software *Nuages* (Inria, France) [Nat20].

Then, a 3D hexahedral mesh is generated using TrueGrid. A structured template mesh on a unit square is mapped to the horizontal slices of the muscle geometry that resulted from the segmentation. After mesh smoothing, the slices are connected vertically to form a 3D mesh. Fiber directions are described by Bézier curves in a reference volume and mapped to the muscle geometry using the same mapping. The fiber direction then are used in a transversely-isotropic material formulation. Simulations are performed using the finite element solver *Nike3D* (Lawrence Livermore National Lab, Livermore, USA) [Mak91].

We base our work on this study and use a similar mapping from a template mesh to the actual muscle volume. In comparison to [Ble05b], we use an improved mapping based on harmonic maps, which potentially leads to better quality meshes on the slices of the muscle. Instead of the unit circle template mesh, we experiment with different reference meshes and evaluate their quality.

In the study of [Ble05b], fiber directions are defined based on anatomically assumed directions. However, the definition is carried out on the cuboid reference geometry. This means that the authors mentally morph the muscle geometry into the reference geometry in order to define fiber directions, using their expertise. Then, the fiber directions together with the cuboid are transformed back to the actual geometry. This approach simplifies the definition of the fibers. However, defining the fiber direction directly on the muscle geometry can lead to better results. Thus, our approach is to automatically estimate fiber directions and define fibers directly in the muscle domain. At the same time, the 3D mesh and 1D fibers are aligned in our work to allow for better numerical and data structure properties of the discretization.

The definition of fiber directions follows a method proposed in [Cho13]. The directions are assumed to follow a divergence free vector field. Such a field can be created by taking the gradient of the solution of the Laplace equation. Neumann boundary conditions are defined at the attachment points of the muscle tendon complex. The solution of the Laplace equation corresponds to the pressure values of a potential flow. Its gradient corresponds to the velocity and individual fascicles or fibers can be obtain by tracing streamlines through the velocity field. This approach is extended and validated by the studies in [Ino15] and [Han17]. We incorporate this method into our workflow.

3.3 Preprocessing of the Muscle Geometry

The first step towards creating a structured mesh is to obtain a representation of the surface of the muscle. Starting point is a human biomedical imaging data set. In this section, two possible workflows are presented how to extract the muscle and tendon surfaces from imaging data. The two workflows are visualized in Fig. 3.3. The workflow using the branch on the left side in Fig. 3.3 is automized but only works for the particular data set and extracting the biceps muscle. The right branch involves manual steps and is applicable for any muscle geometry.

3.3.1 Data Source

Anatomic images provide the basis for the extraction of muscle geometries. Our used data set originates from the Visible Human Project [Spi96] of the United States National Library of Medicine. The project has published anatomic images derived from a male corps, among other data sets. The data, known as “Visible Human Male”, were published in 1994. Colored images of transversal cross-sections were obtained by cryosectioning. A total of 1871 images with dimensions of 2048 by 1216 pixels and 24 bit color depth visualize the whole human body. Parts of the upper arms are contained in approximately 500 of these images. The size of a pixel is 0.33 mm in transversal direction and 1 mm in axial direction. The size of the complete set of JPEG compressed images is 772 MB. Cropping and selecting the relevant portions of the upper arm extracts a dataset with the size of 35 MB.

An extract of an image of the upper arm is given in Fig. 3.4. The location of biceps and triceps brachii muscles can be identified in the dark red tissue. For the biceps, the two muscle heads are visible, separated by the bright diagonal line from bottom left to top right. For the triceps, at least two of the three heads can be identified. The blue background is colored frozen gelatin that was needed during cryosection to stabilize the arms.

3.3.2 Automatic Surface Extraction

This section outlines the automatic algorithm to obtain the muscle surface from the Visible Human Male data set. The scheme corresponds to the left branch in Fig. 3.3. The algorithm was implemented in a Python script as part of the Bachelor thesis of Kusterer

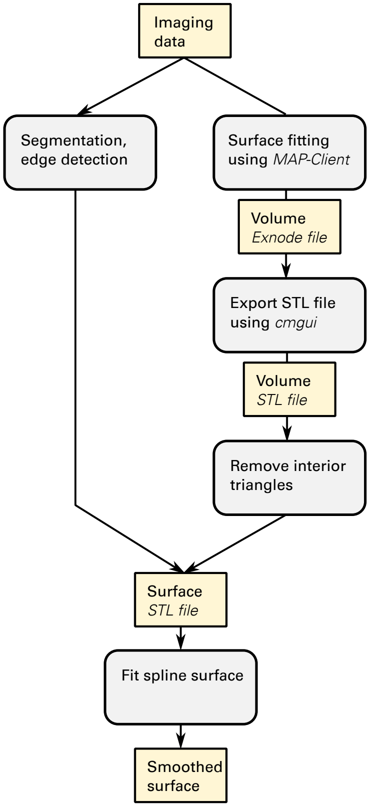


Figure 3.3: Workflow of generating a surface representation of the muscle and tendons from imaging data. Operations and intermediate results are shown as gray and yellow boxes, respectively. Two alternatives are given by the two branches. On the left, the imaging data are automatically processed to directly retrieve points on the surface of the muscle. The right branch achieves the same with three steps of which the first one involves manual adjustments. At the end, a spline surface smooths the collected data from both possibilities to yield the resulting surface representation.

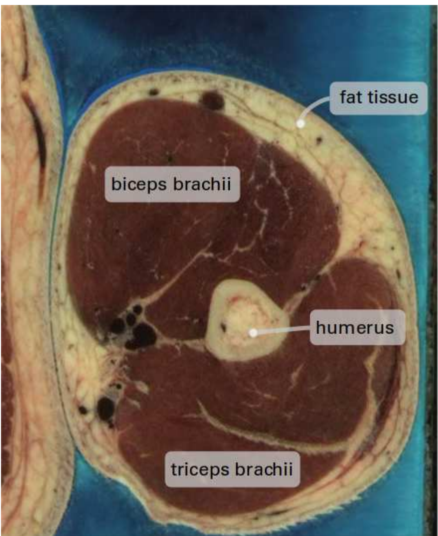


Figure 3.4: Exemplary extract of image number 483 from the Visible Human Male. A transversal slice of the left upper arm is shown as seen from the bottom. The biceps and triceps muscles as well as the humerus bone can be identified.

[Kus19] that was supervised by me. The algorithm is capable of extracting muscle and bone geometries from the mentioned imaging dataset.

At first, the color values in the images are used to segment the pixels into muscle tissue, surrounding tissue and skeletal structure. The algorithm traverses the selected and cropped relevant parts of the images.

For every such part of an image, pixels that match a certain range in the RGB color space are marked and categorized. The categories are muscle tissue and, for demonstration, also bone tissue. The corresponding color ranges are given in Tab. 3.1.

The color based classification does not succeed everywhere as the white shade corresponds not only to bone material but also to fat and other tissue. Therefore, the algorithm removes artifacts located near the outer gelatin from the set of pixels that was categorized as bone.

Exemplary results for image number 483 are given in the left column of Fig. 3.5. It can be seen that the marked regions for muscle and bone have gaps in the interior resulting

	red	green	blue
muscle	$60-100$	$30-75$	$15-60$
bone	$145-255$	$135-205$	$60-160$

Table 3.1: Ranges in the RGB color space to identify pixels of muscle and bone segments. The numbers correspond to 24 bit colors with the range $[0,255]$ for every color channel.

from differently colored tissue inside muscles and bones. On some images, the set of pixels also includes small objects outside the actual muscle and bone regions.

To reduce the gaps and small objects, the morphological operations *closing* and *opening* are applied on the data. These operations consist of *dilation* and *erosion* steps. Both are pixel based operations that traverse the dataset and for every pixel consider a window of 3×3 pixels centered at the current position. Dilation picks the maximum value and erosion the minimum value from this window and assigns it as the pixel's value in a new image. In our case, values of zero and one correspond to non-categorized and categorized pixels, respectively.

Closing consists of dilation followed by erosion and closes small gaps or holes in the marked objects. Opening consists of erosion followed by dilation and removes small artifacts outside the actual bone and muscle areas. It was found effective to perform both dilation and erosion twice in sequence to yield good results containing almost no more holes nor unwanted small objects.

Next, the algorithm determines the contours of all regions with marked pixels. This leads to lines with a width of one pixel that enclose the muscle and bone areas. The right column of Fig. 3.5 shows the results after this step. It can be seen that numerous gaps have been closed by the morphological operations. In some images, as in the considered example, the muscle area gets split into multiple smaller enclosed regions, which is not desired. These images skipped in the processing. However, proper contours of the biceps are found in the majority of images.

In the next step, a single contour for each of muscle and bone is obtained in every image. If there are multiple contours per image, the one that is located closest to the upper right corner of the image is selected for the muscle. If all contours in an image are shorter than 20 pixels, this is an indication for bad segmentation quality and the whole image gets discarded. Because of the discarded images, the resulting surface description has a lower resolution at the respective locations. This is not a problem as the data is subsequently approximated by a smooth spline surface.

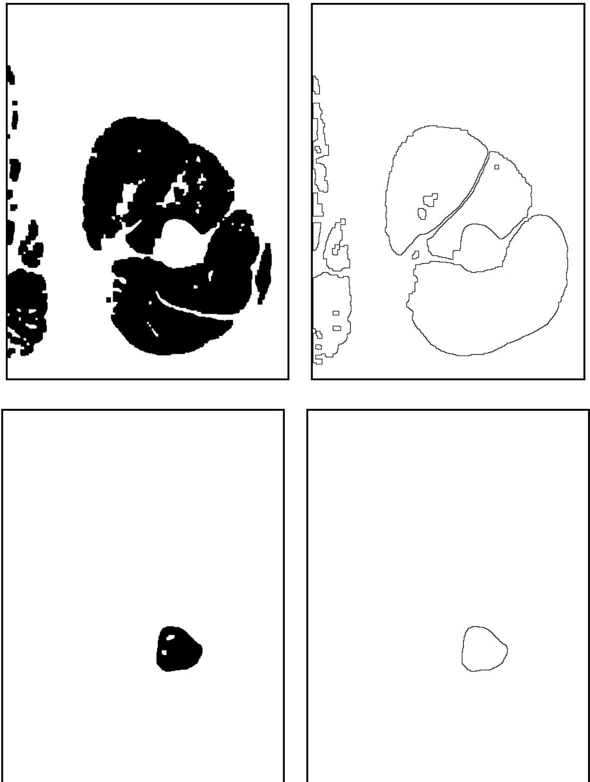
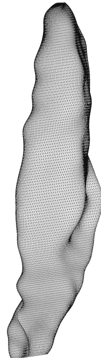
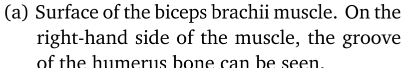
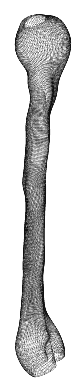


Figure 3.5: Intermediate steps of the algorithm to determine surface geometry of muscles and bones. The left column shows pixels from the image in Fig. 3.4 that were categorized to be muscle tissue (top) and bone material (bottom). The right column shows a later step in the algorithm, where the surface of muscle (top) and bone (bottom) is estimated.





(a) Surface of the biceps brachii muscle. On the right-hand side of the muscle, the groove of the humerus bone can be seen.



(b) Surface of the humerus.

Figure 3.6: Surfaces of biceps and humerus bone obtained by the automatic surface extraction algorithm.

The result is a set of contours for muscle and bone in the cross-sectional planes of the images. Combining these, we get a point cloud in 3D space that approximates the surface of the biceps muscle and the surfaces of the considered bones humerus, ulna and radius. Using these points, a spline surface can be fitted and subsequently triangulated. Resulting surfaces for the biceps and humerus bones are shown in Fig. 3.6.

The runtime for the algorithm applied on a dataset with 495 images and approximately $144 \cdot 10^6$ pixels in total was 121 min. The used hardware was an AMD Ryzen 5 1600 processor with 6 cores, 3.2 GHz and 16 GB RAM, of which a maximum of 2 GB was used. Because processing of the images can be done in parallel, the runtime was reduced to approximately half (62 min) using 2 threads and to a quarter (30 min) using 6 threads.

The advantage of the presented algorithm is that the outcome solely depends on the imaging data and, thus, no modeling error by manual approximation of the geometry occurs. For example, the obtained surfaces of biceps and humerus geometrically fit perfectly into each other. Intermediate steps are stored as black and white images. By editing these between the steps of the algorithm, manual tweaking is possible and can be used to increase the quality of the results.

A disadvantage is that the algorithm relies on color information in the imaging data to differentiate between muscle and other tissue. Because some involved tissue types have

similar colors, this approach can be error-prone. Furthermore, the color ranges need to be determined experimentally. Therefore, the algorithm is not very robust with respect to image noise and needs adjustments when it should be used to extract other muscles. Expert knowledge about the location and shape of human muscles cannot be used easily to improve the results of the algorithm.

An alternative approach is to manually segment the imaging data and construct surfaces with the help of a tool. This approach is described in the following section.

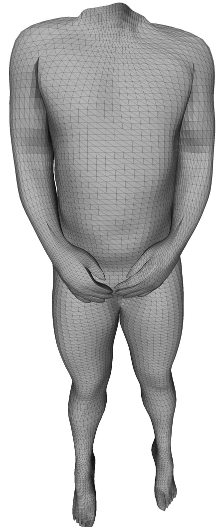
3.3.3 Manually Guided Surface Extraction

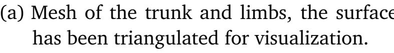
Manually guided segmentation can be done using the *MAP client* of the Musculoskeletal Altas Project (MAP) [Zha14]. This application allows creating and execute a workflow to achieve data processing and simulation tasks. In a graphical window, the user can place and connect various workflow steps. When executing the workflow, each step shows a dialog where the required configuration can be entered or the operations can be performed on a visual representation of the data at this workflow stage.

Possible workflow steps include source and sink operations such as reading image data and writing meshes. Imaging data such as the 2D images from the Visual Human Male can be visualized in a 3D representation. The user can place points in the 3D space to mark boundaries of the visualized muscle and tissue structures. Further workflow steps allow creating meshes of predefined geometrical shapes, such as cubes and cylinders and merge them into a common mesh. These meshes can be fitted to point clouds of user defined points. This is done by a least squares approach minimizing the distances between user created points and the mesh surface. Details can be found in [Fer18].

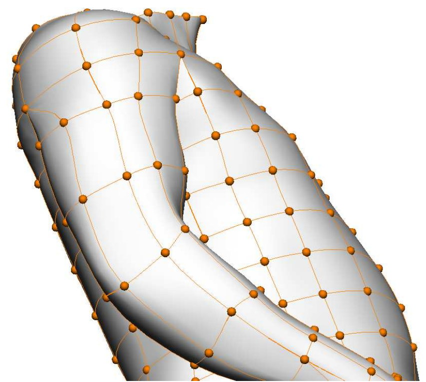
The MAP client has a plugin architecture and allows creating new workflow steps. It imports features from OpenCMISS, especially data processing formats and tools from OpenCMISS Zinc. Meshes can be created with 3D cubic Hermite elements that allow for a high geometric modeling flexibility with a low number of nodes. Such meshes are stored in the OpenCMISS file format of `exnode` and `exelem` files.

As a result, meshes of individual muscles or the whole human organism can be created. Figure 3.7 shows meshes that were created from the cryosectioning data of the Visible Human Male. In Fig. 3.7a, almost the whole body has been extracted. In Fig. 3.7b, the mesh consisting of cubic Hermite elements is visualized. A relatively coarse mesh width suffices to model a smooth surface of the body. When exported in the exfiles format from





(a) Mesh of the trunk and limbs, the surface has been triangulated for visualization.



(b) Detail view of part of the right upper arm and the trunk with orange nodes and edges of a cubic Hermite element mesh.

Figure 3.7: Mesh of the Visible Human Male from the Visible Human Project.

the MAP client, the data can be visualized, e.g., using *cmgui*, the visualization tool of OpenCMISS Zinc.

The mesh width of the meshes obtained using the MAP client was chosen such that the surface fitting yielded good results. The meshes are not necessarily ready for use in a simulation, especially if a high mesh resolution is desired. Apart from the mesh width also the type of elements can be different from what is needed for a finite element simulation. Our goal is to obtain meshes with linear or quadratic Lagrange elements with configurable mesh widths for the specified upper arm muscles, such as the biceps brachii.

Therefore, the next step of the workflow, as visualized by the right branch of Fig. 3.3, is to transform the volume mesh into a surface mesh which then can be used as start for further meshing. The further meshing steps are visualized in Fig. 3.8. The start is the Hermite mesh shown in the left-most image.

The Hermite elements can be triangulated and stored as an STL file using the tool `cmgui`.

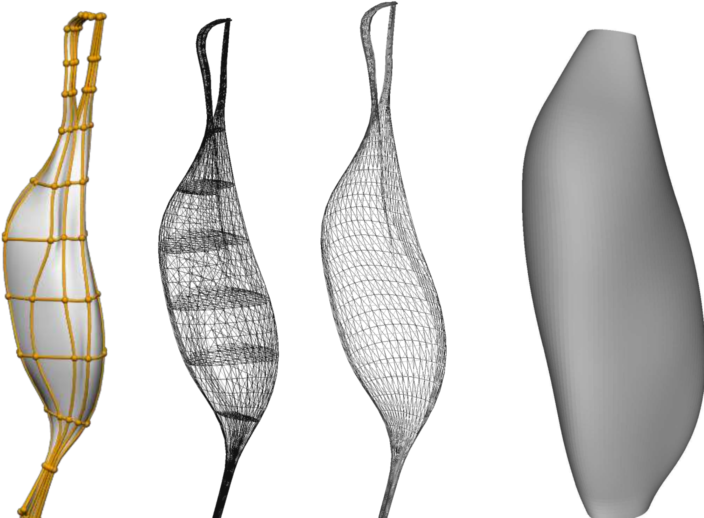


Figure 3.8: Processing the geometry of the biceps brachii muscle. From left to right: mesh with cubic Hermite elements, STL mesh with inside triangles, STL surface mesh where triangles lying inside have been removed, Spline surface of the muscle belly.

This process triangulates the non-planar faces of all Hermite elements. This leads to a dataset with triangles both on the surface and in the inside of the volume, as can be seen in the second image of Fig. 3.8. At this stage, the use of the MAP and OpenCMISS related tools is finished and further processing steps are performed using tools from OpenDiHu that we developed on our own.

A Python script removes the triangles inside the volume. The detection whether a triangle is inside the volume is done by casting four rays from the center of gravity of the respective triangle and determining if the rays intersect any other triangles. The rays have directions $(x, y, z) = (\pm 1, \pm 1, \frac{1}{3})$, where the z axis is oriented along the muscle's longitudinal axis and the x and y axes are oriented in radial direction. The ray-triangle intersection is done using the fast Möller-Trumbore algorithm [Möl97]. For every ray, all triangles are checked. Only if all four rays intersect at least one more triangle, the starting triangle is considered to be inside the volume and subsequently removed from the dataset.

This algorithm has a quadratic time complexity $\mathcal{O}(n^2)$ in the number of triangles n. It

could be improved by organizing the triangles in a spatially adaptive data structure, such as an octree. However, since this preprocessing step has to be performed only once for a given geometry, the runtime is not critical and there is no need for such optimization.

The result of this operation is a triangulated surface, shown in the third image of Fig. 3.3. The next step is to create a Spline surface of the muscle belly, as shown in the right-most image of Fig. 3.3. This is described in the next two sections.

3.3.4 Introduction of Spline Surfaces

After the surface representation of the muscle has been obtained from either the left or the right branch of the preprocessing workflow in Fig. 3.3, the surface is given by a point cloud or a number of triangles. To remedy eventual outliers or unphysiologically sharp edges from the segmentation, a Spline surface is fitted to the data. This leads to a smooth surface representation and later to a better conditioned finite element mesh in the simulation. However, this step is optional. It is also possible to directly use the surface triangulation from Sec. 3.3.3 for the meshing algorithm described in Sec. 3.4.

The surfaces use Nonuniform Rational B-splines (NURBS). A NURBS surface is a generalization of a B-spline surface. From a modeling point of view, B-spline surfaces have three advantageous properties. First, the B-spline surface can be constructed with given smoothness properties. Second, the definition of a particular B-spline surface builds on intuitive geometric information, which simplifies their creation: A control polygon mesh in 3D space is defined. Its convex hull is guaranteed to contain the surface. Third, the geometric parameters of a B-spline surface have only local impact on the shape of the surface. This allows a B-spline surface of a fixed, low polynomial degree to approximate point clouds with any number of points without loosing approximation quality.

A limitation of B-spline surfaces is that circular and spherical shapes cannot be represented. This limitation is overcome by NURBS surfaces. NURBS surfaces are defined as the perspective projection into 3D space of a B-spline surface in 4D space.

The mathematical description is given in this section, following the notation of [Pie12]. The building blocks are the B-spline basis functions of polynomial degree p. Given a knot vector

$$\Xi = (\xi_1, \xi_2, \dots, \xi_k) \in \mathbb{R}^k \quad \text{with } a = \xi_1 \le \xi_2 \le \cdots \le \xi_k = b,$$

the ith B-spline basis function $N_{i,n}$ of degree n is defined recursively starting with the piecewise constant function

$$N_{i,0}(\xi) = \begin{cases} 1 & \text{for } \xi_i \leq \xi < \xi_{i+1}, \\ 0 & \text{else,} \end{cases}$$

and using the following relation to define the functions of higher degree ($n > 0$):

$$N_{i,n}(\xi) = \frac{\xi - \xi_i}{\xi_{i+n} - \xi_i} N_{i,n-1}(\xi) + \frac{\xi_{i+n+1} - \xi}{\xi_{i+n+1} - \xi_{i+1}} N_{i+1,n-1}(\xi), \quad \text{for all } i > 0.$$

Because neighboring entries in the knot vector can be equal, the fraction $0/0$ can occur. In this case, $0/0 := 0$ is defined. Note that by construction of $N_{i,n}$ a zero denominator implies that also the dividend is zero.

A B-spline curve $\mathbf{C} : \mathbb{R} \to \mathbb{R}^d$ of polynomial degree p is defined as

$$\mathbf{C}(u) = \sum_{i=1}^{\ell} N_{i,p}(u) \mathbf{P}_i, \quad u \in [a, b].$$

The coefficients $\mathbf{P}_i \in \mathbb{R}^d, i = 1, \dots, \ell$, of the basis functions $N_{i,p}$ are called *control points* and define the control polygon. The number ℓ of basis functions and control points is determined from the number of knots k in an open knot vector and the polynomial degree p as $\ell = k - p - 1$.

The number of equal entries in series in the knot vector is the *multiplicity* of the respective knot value. Usually *open* knot vectors Ξ are used where the first and the last knot occur with a multiplicity of $p+1$. This makes the first and the last points of the B-spline curve coincide with the control polygon points: $\mathbf{C}(a) = \mathbf{P}_1$ and $\mathbf{C}(b) = \mathbf{P}_\ell$.

The multiplicities of the knots in the knot vector encode information about the smoothness of the B-spline curve. If the knot value $\hat{\xi}$ has a multiplicity of m, the B-spline curve will be $(p - m)$ times continuously differentiable at $\mathbf{C}(\hat{\xi})$.

An exemplary B-spline curve is shown in Fig. 3.9. It uses a *non-uniform* knot vector for polynomial degree $p = 3$, where the differences $\xi_{i+m} - \xi_i$ between neighboring knot values vary. The effect of different multiplicities can be seen. The multiplicity $m = p = 3$ places the point of the curve at the knot on the respective control point, as for $\xi = 49$ in the example. The multiplicity $m = p - 1 = 2$ places the point of the curve at the knot on the control polygon, as in the example at $\xi = 10$. A lower multiplicity $m < p - 1$

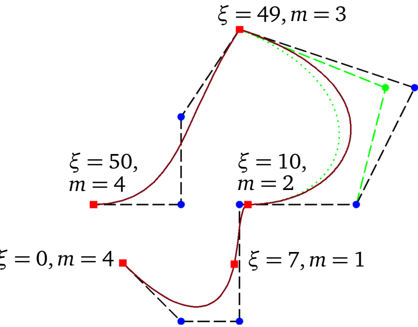


Figure 3.9: Exemplary B-spline curve (red) of degree $p = 3$ for the knot vector $\Xi = (0, 0, 0, 0, 7, 10, 10, 49, 49, 49, 50, 50, 50, 50)$, control points (blue) and control polygon (black). Positions of the curve $\mathbf{C}(\xi_i)$ at the knots ξ_i are indicated by the red squares and the knot value ξ and its multiplicity m is given. The effect of moving one control point is shown in green.

does not yield a higher smoothness and in turn does not force the curve to coincide with the control polygon at the respective knot. It can also be seen that the B-spline curve stays inside the convex hull of the control polygon which is a property of B-spline curves [Pie12].

The effect of moving one of the 10 control points is visualized with green color in Fig. 3.9. The B-spline basis function $N_{i,p}$ has a local support of $S = (\xi_i, \xi_{i+p+1})$. Consequently, only the corresponding part of the curve, $\mathbf{C}(\xi)$ for $\xi \in S$, changes.

A B-spline surface $\mathbf{S} : \mathbb{R}^2 \to \mathbb{R}^d$ is given by the tensor product of two B-spline curves:

$$\mathbf{S}(u, v) = \sum_{i=1}^{\ell^{(1)}} \sum_{j=1}^{\ell^{(2)}} N^{(1)}_{i,p^{(1)}}(u) N^{(2)}_{j,p^{(2)}}(v) \mathbf{P}_{i,j}. \tag{3.1}$$

Here, we have two polynomial degrees $p^{(1)}$ and $p^{(2)}$, the ansatz functions $N^{(1)}_{i,p^{(1)}}$ and $N^{(2)}_{j,p^{(2)}}$ with $\ell^{(1)}$ and $\ell^{(2)}$ ansatz functions per coordinate direction are constructed from the corresponding knot vectors per coordinate direction.

NURBS, B-spline curves and surfaces are formulated using *homogeneous coordinates*. Every point in Cartesian coordinates $(x, y, z) \in \mathbb{R}^3$ has a set of homogeneous coordinates $(\tilde{x}, \tilde{y}, \tilde{z}, w) = (x\,w, y\,w, z\,w, w)$. Thus, the Cartesian coordinates can be obtained from

the homogeneous coordinates by the *perspective division*, i.e., dividing all but the last coordinate by the weight w.

A NURBS surface is given by the same definition as the B-spline surface in Eq. (3.1) except that the control points $\mathbf{P}_{i,j} \in \mathbb{R}^3$ are enriched with scalar weights $w_{i,j}$ and, thus, replaced by $(\mathbf{P}_{i,j}, w_{i,j}) \in \mathbb{R}^4$. The resulting surface $\mathbf{S}$ is given in homogeneous coordinates. Executing the perspective division yields the form:

$$\mathbf{T}(u,v) = \sum_{i=1}^{\ell^{(1)}} \sum_{j=1}^{\ell^{(2)}} R_{i,j}(u,v)\, \mathbf{P}_{i,j},$$

$$\text{with } R_{i,j}(u,v) = \frac{N_{i,p^{(1)}}(u)\, N_{j,p^{(2)}}(v)\, w_{i,j}}{\sum_{r=1}^{\ell^{(1)}} \sum_{s=1}^{\ell^{(2)}} N_{r,p^{(1)}}(u)\, N_{s,p^{(2)}}(v)\, w_{r,s}}.$$

The new rational basis functions $R_{i,j}$ and the possibly non-uniform knot vectors give rise to the name Non-Uniform Rational B-spline surface (NURBS).

3.3.5 Fitting a Spline Surface to the Muscle Geometry

In order to find a NURBS surface for the given triangulated surface of a muscle, at first, the part of the geometry corresponding to the tendons is removed such that the resulting triangles model only the surface of the muscle belly. In our example of the biceps muscle, the resulting belly has a length of 12.8 mm.

Then, twelve cross-sections are extracted from the surface triangles. As a result, we get twelve horizontal circumferential rings. On each ring, 9 equidistant points are determined. The first point is appended after the last point in every ring, such that in total we obtain a grid of 10×12 points.

Then, the least squares surface approximation algorithm by [Pie12] is used to fit a NURBS surface to the points. The implementation of the algorithm is given by the NURBS-Python (geomdl) library. Polynomial degrees of $p^{(1)} = 3$ and $p^{(2)} = 2$ are used where the first dimension corresponds to the cross-sectional direction of the muscle. The knot multiplicity is chosen as $m = 1$ for both coordinate directions. We obtain a two times respective one times continuously differentiable surface in u and v direction. The resulting NURBS surface and the control polygon are visualized in Fig. 3.10. Note that the control polygon is different from the grid of points against which the surface is fitted.

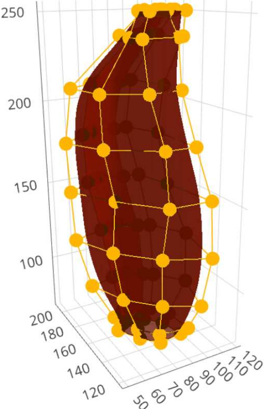
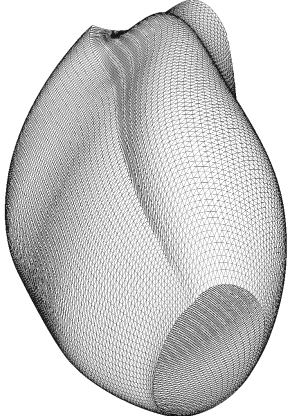


Figure 3.10: Muscle surface description with Splines: Fitted NURBS surface of the biceps muscle (red) and the control polygon (orange).

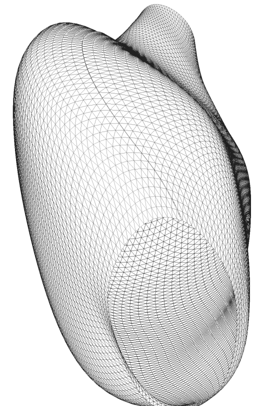
Figure 3.11a shows the result of this approach in more detail. We observe that the surface is non-differentiable and has a kink at the seam line where the first and last points of each ring meet. The reason for this is that the surface fitting algorithm does not pose any conditions on the tangents at the edges of the fitted NURBS surface. Thus, the tangents mismatch.

Since no implementation of a fitting algorithm specifically for a tubular NURBS surface with periodicity in tangential direction is available, we develop a different remedy. We modify the point grid that is used for the surface fitting. The series of 9 equidistant points on each ring is replicated twice and the first point is again added as the last point. This leads to a grid of $(3 \cdot 9 + 1) = 28 \times 12$ points which wrap around the muscle volume in circumferential direction three times. The NURBS surface fitting algorithm is applied on this grid. The resulting NURBS surface also wraps around the muscle three times with the two ends being again not properly fitting to each other. From these three wraps, the middle one is extracted. In the biceps example, this corresponds to restricting the NURBS surface $\mathbf{T}(u, v)$ from $(u, v) \in [0, 1]^2$ to $(u, v) \in [0.4, 0.733] \times [0, 1]$.

The result is depicted in Fig. 3.11b. The tangents now match very well between the two sides of the NURBS surface. Additionally, the comparison with the initial approach in Fig. 3.11a shows that an artificial bulge at the top of the muscle in the perspective of the visualization is removed. The overall shape of the muscle now looks smoother and more



(a) First approach with 10×12 control points. The kink at the seam line along the muscle is clearly visible.



(b) Second, improved approach with 28×12 points. It can be seen that the tangents at the seam line match very well.

Figure 3.11: Muscle surface description with Splines: Fitted NURBS surface of the biceps muscle, triangulated for visualization purposes.

natural. Also, a comparison with the result of the automatic algorithm given in Fig. 3.6a shows that the results of our new approach are smoother.

The generated tubular surface has two holes at the top and bottom which prevent it from being an enclosing surface to the muscle belly volume. The borders of these holes each lie in a plane and, thus, the missing surfaces are treated as being planar during the subsequent creation of the 3D meshes.

For the next step, a triangulation of the tubular surface is created and stored as STL mesh file. We use the respective functionality of the NURBS-Python library that creates a structured triangle mesh using the 2D parametrization of the NURBS surface.

3.4 Serial Algorithm to Create Muscle and Fiber Meshes

Next, a 3D mesh for the muscle volume and 1D meshes for muscle fibers need to be generated from the surface representation described in the previous sections. In this

section, first an algorithm for the 3D mesh is described. Then, a second algorithm that reuses results from the first algorithm is presented which generates one dimensional meshes for muscle fibers. Both algorithms are executed in serial. A derived algorithm that can run in parallel and, thus, can handle larger datasets on a distributed memory hardware is presented in Sec. 3.5.

The steps of a serial algorithm for the generation of a 3D mesh are given in Alg. 1. Input is the set of triangles at the tubular surface of the muscle. The tubular surface is oriented along the z axis. In the following descriptions, the muscle in considered to be oriented upright such that the z axis points in vertical direction towards the top. The borders at the bottom and at the top have a constant z coordinate.

Algorithm 1 Serial algorithm for the generation of 3D meshes.

```
1 procedure Create_3D_mesh
  Input: Triangulated tubular surface
  Output: Structured 3D volume mesh

2   Slice geometry
3   Triangulate 2D slices
4   Compute harmonic maps u,v from the slices to a parameter space
5   Construct regular grid in parameter space and map it to slices
6   Form 3D quadrilateral elements between the 2D slices' meshes
```

The idea of the algorithm is to first create 2D meshes with good quality on cross-sectional slices of the muscle volume and then combine them to get a 3D mesh. The algorithm starts with creating the horizontal 2D slices in lines 2 and 3. The slices get vertically connected at the end of the algorithm in line 6 to create the 3D mesh. This step is visualized in Fig. 3.13d.

Because the goal is to create a hexahedral mesh, the horizontal slices have to consist of quadrilaterals. Decomposing a 2D domain into quadrilaterals is easier for a square or circular shaped domain than for an actual cross-section of the muscle. Therefore, we introduce a separate, square or circular shaped parameter domain for creating the quadrilaterals.

Figure 3.12 outlines the method. We start at the upper left of the figure with a triangulation on the cross-sectional slices of the muscle. A mapping from the muscle slices to the parameter space at the right of the figure is computed. We use harmonic maps to ensure a smooth mapping that results in good mesh quality. This first step corresponds to line 4 in Alg. 1. Different parameter domains such as unit circle and unit square are considered, as shown in Fig. 3.12. It can also be seen at the upper right that the image of the muscle

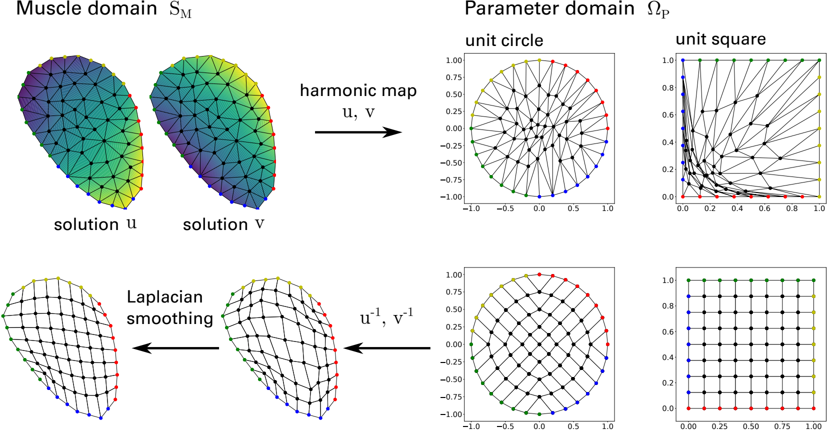


Figure 3.12: Generation of the muscle meshes, overview of the mapping method between the muscle domain (left) and the parameter domain (right) using harmonic maps.

slice triangulation in the parameter domain is better in the unit circle than in the unit square. Therefore, an investigation of different parameter domains and triangulation schemes is necessary.

Next, line 5 of Alg. 1 defines a quadrangulation in the parameter space, shown at the lower right of Fig. 3.12. The quadrilateral elements are mapped back to muscle slices where they are needed for the final 3D mesh. An optional smoothing step at the lower left of the figure further improves the mesh quality. All steps of the algorithm are described in more detail in the following sections.

3.4.1 Slicing of the Geometry

The first step in line 2 of Alg. 1 slices the geometry. This means that horizontal *slices* of the cross-sectional area are extracted from the surface mesh. First, the muscle is divided into equidistant positions $z_i, i = 1, \dots, n$ along the z-axis where the slices are to be extracted. As can be seen in Fig. 3.13a, $n = 13$ z coordinates are selected. Next, for every position z_i, all surface triangles T_j that intersect the plane $Z_i = \{\mathbf{p} = (x, y, z) \mid z = z_i\}$ are considered and the intersection lines $P = T_j \cap Z_i$ are computed. The method of computing plane-triangle intersection is described in the following.

Given is a triangle T with points $\mathbf{p}^1, \mathbf{p}^2, \mathbf{p}^3 \in \mathbb{R}^3$ and a value $\hat{z}$, the result is the set of points $P = T \cap \{\mathbf{p} = (\mathbf{p}_x, \mathbf{p}_y, \mathbf{p}_z) \mid \mathbf{p}_z = \hat{z}\}$ which corresponds to a line segment $\overline{\mathbf{p}^a\mathbf{p}^b}$.

We describe the points in the triangle by two barycentric coordinates ξ_1 and ξ_2 as

$$\begin{aligned} &\mathbf{p}(\xi_1, \xi_2) = (1 - \xi_1 - \xi_2)\,\mathbf{p}^1 + \xi_1\,\mathbf{p}^2 + \xi_2\,\mathbf{p}^3, \\ &\text{with } \xi_1 + \xi_2 \le 1, \quad 0 \le \xi_1, \xi_2 \le 1. \end{aligned} \tag{3.2}$$

$\mathbf{p}_z(\xi_1, \xi_2) = \hat{z}$ defines the equation for the line through the points $\mathbf{p}^a$ and $\mathbf{p}^b$ in barycentric coordinates. The solution is given as

$$\xi_1 = m \cdot \xi_2 + c, \quad m = -\frac{\mathbf{p}_z^3 - \mathbf{p}_z^1}{\mathbf{p}_z^2 - \mathbf{p}_z^1}, \quad c = \frac{\hat{z} - \mathbf{p}_z^1}{\mathbf{p}_z^2 - \mathbf{p}_z^1}, \quad \mathbf{p}_z^2 \neq \mathbf{p}_z^1.$$

For $\mathbf{p}_z^1 = \mathbf{p}_z^2 \neq \mathbf{p}_z^3$ we swap $\mathbf{p}_z^2$ and $\mathbf{p}_z^3$.

Next, the end points of the line segment $\overline{\mathbf{p}^a\mathbf{p}^b}$ are determined. We consider the three sides $\overline{\mathbf{p}^1\mathbf{p}^2}, \overline{\mathbf{p}^2\mathbf{p}^3}$ and $\overline{\mathbf{p}^3\mathbf{p}^1}$ of the triangle and check which of them are intersected by the $z = \hat{z}$ plane by the following three conditions:

1. On the triangle side $\overline{\mathbf{p}^1\mathbf{p}^2}$ the condition $\xi_2 = 0$ holds and the side intersects the plane at $\mathbf{p}(c, 0)$ iff $0 \le c \le 1$.
2. Similarly, on the triangle side $\overline{\mathbf{p}^1\mathbf{p}^3}$ we have the condition $\xi_1 = 0$ and the side intersects the plane at $\mathbf{p}(0, -c/m)$ iff $m \neq 0 \land 0 \le -c/m \le 1$.
3. The third triangle side $\overline{\mathbf{p}^2\mathbf{p}^3}$ is intersected for $\hat{\xi}_1 = (c+m)/(1+m)$ at $\mathbf{p}(\hat{\xi}_1, 1 - \hat{\xi}_1)$ iff $m \neq -1 \land 0 \le \hat{\xi}_1 \le 1$.

If two of these three conditions for intersection of the triangle sides are met, there is an intersecting line segment $\overline{\mathbf{p}^a\mathbf{p}^b}$ with $\mathbf{p}^a \neq \mathbf{p}^b$ and the two intersection points $\mathbf{p}^a$ and $\mathbf{p}^b$ on the triangle sides are determined as stated above. The trivial cases $\mathbf{p}^a = \mathbf{p}^b$ and the case where $\mathbf{p}^a$ and $\mathbf{p}^b$ are equal to two of the triangle corners $\mathbf{p}^1, \mathbf{p}^2$ and $\mathbf{p}^3$ are handled separately in our implementation.

After the presented computations are performed for all planes Z_i and all triangles T_j, we have a number of line segments that form a geometric "ring" for each z plane. The line segments are ordered according to their adjacency and a counter-clockwise orientation with respect to the z axis is ensured. The length of each ring is computed. A number $m = 16$ of equidistant points is selected on each ring.

Because the selected points on the rings are later used as boundary points of the resulting 3D mesh, their position relative to each other on different rings should be in a tidy manner. The positioning should enable straight connection lines in longitudinal direction of the muscle connecting the points on every ring. For illustration, Fig. 3.13a shows such a configuration of properly positioned ring points. Connecting the points from top to bottom is possible with smooth lines rather than zigzag lines. In result, the outer surface of the final mesh in Fig. 3.13d consists of a smooth quadrilateral mesh.

With given rings and number m of equidistant points per ring, only the position of one point per ring is not yet fixed. To close the definition, in the following we formulate a first condition that relates the point positions of two neighboring rings and a second condition for one point at the bottom-most ring.

As mentioned, the first condition should ensure that the point positions on neighboring rings are as similar as possible. This is done by minimizing the distance between the first points on every ring. In the algorithm, the z planes are traversed from bottom to top. The first point $\tilde{\mathbf{p}}_{i,0}$ on a ring at $z = z_i$ is determined from the first point $\tilde{\mathbf{p}}_{i-1,0}$ of the previous ring at $z = z_{i-1}$ as the one with the minimal distance $|\tilde{\mathbf{p}}_{i,0} - \tilde{\mathbf{p}}_{i-1,0}|$. Thus, the searched point $\tilde{\mathbf{p}}_{i,0}$ has the property that the line between $\tilde{\mathbf{p}}_{i,0}$ and $\tilde{\mathbf{p}}_{i-1,0}$ and the tangent of the ring are perpendicular.

Given any point $\mathbf{p}$ on the ring at z_i and the tangent vector $\mathbf{u}$ at this point, we can project the connection vector $\mathbf{v}$ from $\mathbf{p}$ to the start point $\tilde{\mathbf{p}}_{i-1,0}$ of the previous ring, $\mathbf{v} = \tilde{\mathbf{p}}_{i-1,0} - \mathbf{p}$, onto the tangent $\mathbf{u}$. This leads to the plumb foot point $\mathbf{p}_0$ by the computation

$$\mathbf{p}_0 = \mathbf{p} + t\,\mathbf{u} \quad \text{with } t = \frac{\mathbf{v}\cdot\mathbf{u}}{|\mathbf{u}|^2}.$$

Performing this calculation for every line segment u on a ring allows to select the plumb foot $\mathbf{p}_0$ with the smallest distance to the start point $\tilde{\mathbf{p}}_{i-1,0}$ of the previous ring to be the start point $\tilde{\mathbf{p}}_{i,0}$ of the current ring. This is the point that fulfills the first condition.

With this first condition, all points are only fixed relative to each other. The definition of one point, the start point $\tilde{\mathbf{p}}_{0,0}$ of the bottom-most ring, is missing. The second condition fixes this point by a prescribed plane at $x = \hat{x}$ and selects $\tilde{\mathbf{p}}_{0,0}$ such that its x coordinate lies in this plane. From the (usually two) points that meet this condition, the one with lower y coordinate is selected. The actual value of $\hat{x}$ is determined experimentally such that the resulting point positions are visually uniform. Not every value leads to a good result because of the shape of the biceps muscle, especially the groove where the humerus bone is located.

The resulting grid of points on the biceps surface is visualized in Fig. 3.13a. It can be seen that all points of the same ring have the same z coordinate. By connecting neighboring points horizontally and vertically, a regular grid can be formed. This overall grid in this x-z perspective view looks relatively uniform, e.g., compared with the gray surface triangulation mesh of the biceps geometry. The spacing between the points is lower at the top and bottom of the muscle because of the smaller circumference at these locations.

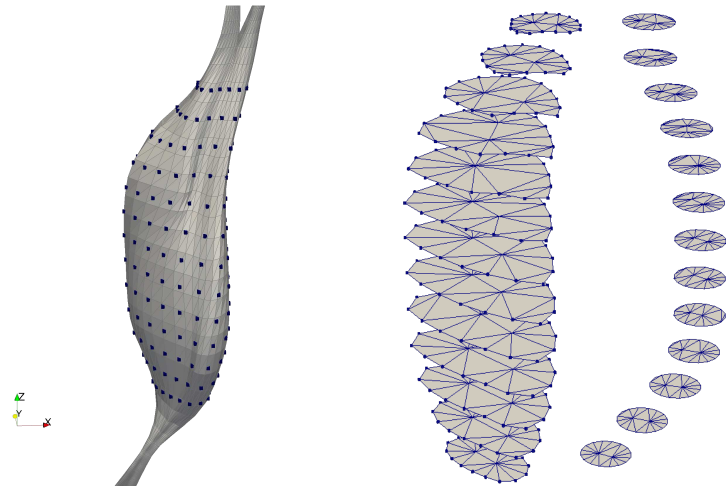
3.4.2 Triangulation of the Slices

The points of each ring enclose a planar, polygonal surface, a *slice* S_M of the muscle. The next step in the algorithm, line 3, is to triangulate the extracted slices, i.e., to construct triangles that decompose the polygons. The result of this step is visualized on the left side in Fig. 3.13b.

We select three different methods to construct this triangulation. The first and second methods are based on Delaunay triangulations. The third method creates a custom triangulation using a simple construction scheme with only one additional point. Figure 3.14 visualizes results of the three methods for one slice.

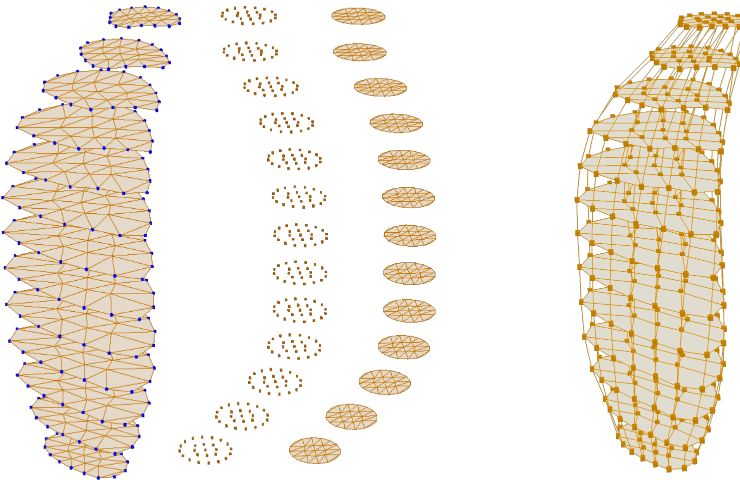
The first method uses the tessellation algorithm from the spatial algorithms and data structures module of the Python package *SciPy*. The Quickhull algorithm [Bar96] is used which triangulates the convex hull of the points. In consequence, the triangulations of concave slices have triangles that lie outside the interior of the slice, which is a disadvantage. An advantage is that the triangulation uses all given points and no new points are added. However, this often results in meshes of lower quality than if adding additional points were allowed. The example in Fig. 3.14a shows such a concave slice. At the bottom of the domain, the triangles are outside the slice and almost degenerate.

The second method uses a Delaunay refinement algorithm described in [She02] and implemented in the *Triangle* software [She96]. A conforming, constrained Delaunay triangulation is created. The triangulation correctly handles convex and concave domains. Conforming means that the triangulation uses the given points at the boundary. Additional points on the boundary as well as in the interior are added. The triangulation is constraint to generate triangles with minimum angles of 20 degrees and a maximum area A that is set to a value depending on the area of the bounding box. In consequence, the generated triangulations of all slices have a guaranteed mesh quality in terms of angles and about the same size and number of triangles.



(a) Extracted boundary points (blue) on the biceps surface mesh (gray). This is the result of line 2 in Alg. 1.

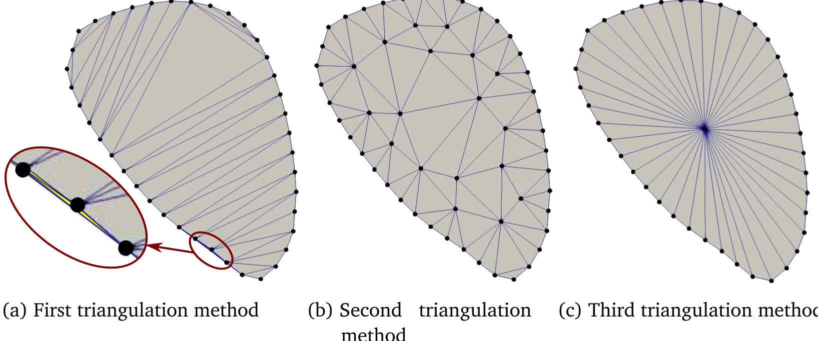
(b) The generated triangulation of the slices (left) and the image $\mathbf{y}(\mathbf{x})$ of the triangulation under the harmonic map (right). This figure shows the result of lines 3 and 4 in Alg. 1.



(c) Grid in parameter space (right) and muscle domain (left), result after line 5 in Alg. 1.

(d) 3D elements formed by connecting the slices in line 6 in Alg. 1.

Figure 3.13: Steps of the serial algorithm for 3D mesh generation, Alg. 1, executed directly on the surface mesh of the biceps muscle (not the B-spline surface).



(a) First triangulation method (b) Second triangulation method (c) Third triangulation method

Figure 3.14: Intermediate step of 3D mesh generation: triangulation of slices, result of different triangulation methods for a slice in the center of the biceps muscle.

Comparing the result of the second method in Fig. 3.14b with the result of the first method in Fig. 3.14a shows the better triangulation quality as the triangles all have larger angles.

The third method places one additional point at the center of gravity of the given points. Triangles are constructed by connecting the center point with two adjacent points on the boundary, for all given points. The resulting triangulation resembles a pie chart. For some extreme concave slices, this method also creates triangles that partly lie outside the slice, but this rarely occurs with muscle cross-sections. The advantage of this approach is its simplicity. Figure 3.14c shows the result for an exemplary slice. In contrast to the first method, the third method creates a valid triangulation despite the concave domain.

3.4.3 Harmonic Maps

Next, harmonic maps are created that allow to smoothly map a given 2D reference mesh onto an actual cross-section of the muscle. The initial application of harmonic maps to meshes used for biomedical simulations is given by [Mar10] and [Mar11]. The authors improve a given, over-sampled surface mesh obtained from classical segmentation. This is done by partitioning the surface into multiple mesh partitions of zero genus (i.e., containing no holes) and transforming them to a reference space using harmonic maps. There, controlled remeshing is carried out before the transformation is reversed.

In our algorithm, harmonic maps are also used for the purpose of generating high quality meshes. In contrast to the literature, the mapping is based on the muscle slices instead of the surface. Also, different parameter domains are investigated.

A function $u : \Omega \to \mathbb{R}$ on a domain $\Omega \in \mathbb{R}^d$ is *harmonic* if it is a solution of the Laplace equation $\Delta u = 0$. From variational calculus, it is known that harmonic functions are extremals of the *Dirichlet energy functional* [Wey40],

$$E[u] = \frac{1}{2} \int_\Omega |\nabla u|^2 \,\mathrm{d}\mathbf{x}.$$

For an intuitive understanding, the map u can be seen as deforming an elastic material that is initially located tension-free in the domain Ω. Then, the Dirichlet energy $E[u]$ describes the total amount of squared stretch or elastic energy resulting from the tension that occurs in the deformed state. A harmonic map minimizes this total tension. Qualitatively, the map deforms neighborhoods of all points in Ω by a similar amount, thus, preserving geometrical structures in Ω, e.g., given by a mesh. The idea of our approach is that applying a harmonic map on a mesh with good quality preserves the mesh quality also in the image under the map.

In Alg. 1, computing the harmonic maps u and v is done in line 4. For a given slice S_M, the functions u and v map from points $\mathbf{x} \in S_M$ to coordinates $u(\mathbf{x}), v(\mathbf{x}) \in \mathbb{R}$ of a parameter domain $\Omega_P \subset \mathbb{R}^2$. The parameter domain is either a unit circle or a unit square.

The vector $\mathbf{y}(\mathbf{x}) := (u(\mathbf{x}), v(\mathbf{x}))^\top$ for $\mathbf{x} \in S_M$ is interpreted as position in Ω_P. The maps are constructed such that the boundary ∂S_M of the slice S_M is mapped to the boundary $\partial\Omega_P$ of the parameter domain Ω_P while preserving the distance between points on the boundary. The mapping $\mathbf{y} : S_M \to \Omega_P$ is bijective and harmonic, i.e., the Laplacians of u and v are zero. More specifically, $u : S_M \to \mathbb{R}$ and $v : S_M \to \mathbb{R}$ are solutions of

$$\Delta u(\mathbf{x}) = 0, \quad \Delta v(\mathbf{x}) = 0 \quad \forall \mathbf{x} \in S_M. \tag{3.3}$$

To derive suitable Dirichlet boundary conditions for these equations, we consider a uniform parametrization $\mathbf{p} : [0,1] \to \partial S_M$ of the boundary ∂S_M of the slice, i.e.,

$$\frac{\partial \ell(t)}{\partial t} = c \in \mathbb{R} \quad \forall t \in [0,1], \quad \text{where } \ell(t) := \int_0^t |\mathbf{p}'(\tau)| \,\mathrm{d}\tau.$$

We require the image of the boundary parametrization in Ω_P to be also uniform, i.e.,

$$\frac{\partial \ell_P(t)}{\partial t} = c_P \in \mathbb{R} \quad \forall t \in [0,1], \quad \text{where } \ell_P(t) := \int_0^t |\mathbf{y}'(\mathbf{p}(\tau))| \, \mathrm{d}\tau.$$

Corresponding boundary points $\mathbf{x}_{\text{M,boundary}} \in S_M$ and $\mathbf{x}_{\text{P,boundary}} = (u_{\text{P,boundary}}, v_{\text{P,boundary}})^\top$ can be defined. This leads to the following Dirichlet boundary conditions that close the definition in Eq. (3.3):

$$u(\mathbf{x}_{\text{M,boundary}}) = u_{\text{P,boundary}}, \quad v(\mathbf{x}_{\text{M,boundary}}) = v_{\text{P,boundary}}. \tag{3.4}$$

Equations (3.3) and (3.4) describe a boundary value problem of ordinary differential equations for u and v. We solve it using the finite element method and the spatial discretization given by the triangulation of the slices. Depending on the method of triangulation, a different number of degrees of freedom is given. For the first method with the Quickhull algorithm, no degree of freedom is present and no system of equations needs to be solved. Then, the mapping is only a FE interpolation of the boundary mapping. For the third method, only one degree of freedom for the center point needs to be computed. The second method has as many degrees of freedom as there are additional points inserted during the Delaunay refinement.

The first step is to compute the prescribed boundary points $\mathbf{x}_{\text{P,boundary}}$ in parameter space. When using the first and third triangulation methods, the boundary points on the slices are equidistant and therefore the same number of points need to be sampled equidistantly on the boundary $\partial\Omega_P$ of the parameter space. If the second triangulation method, which potentially adds additional points is used, the same number of points are created on the boundary $\partial\Omega_P$ of the slice as are given on the slice ∂S_M. The boundary points are created such that the relations of their distances are the same on $\partial\Omega_P$ as for the original points on ∂S_M.

Using the standard procedure of the finite element method for $\Delta u(\mathbf{x}) = 0$ on S_M and $u = f(\mathbf{x})$ on ∂S_M, e.g., as outlined in [Rem10], leads to the weak form with ansatz and test functions ϕ,

$$\int_{S_M} (\nabla u^\top \nabla \phi + \nabla f(\mathbf{x})^\top \nabla \phi) \, \mathrm{d}\mathbf{x} = 0 \quad \forall \phi \in \mathcal{H}_0^1. \tag{3.5}$$

Standard linear hat functions are used on the triangles, such that they provide the inter-

polation property $\phi_i(\mathbf{x}_j) = \delta_{ij}$. Using the barycentric parametrization of triangles with points $\mathbf{p}^1, \mathbf{p}^2$ and $\mathbf{p}^3$ introduced in Eq. (3.2), we define the ansatz functions and get their derivatives within the elements by:

$$\phi_1^{(e)} = (1-\xi_1)(1-\xi_2), \qquad \phi_2^{(e)} = \xi_1(1-\xi_2), \qquad \phi_3^{(e)} = (1-\xi_1)\xi_2,$$

$$\nabla\phi_1^{(e)} = (\xi_2-1, \xi_1-1)^\top, \qquad \nabla\phi_2^{(e)} = (1-\xi_2, -\xi_1)^\top, \qquad \nabla\phi_3^{(e)} = (-\xi_2, 1-\xi_1)^\top.$$

The superscript $\square^{(e)}$ refers to the definition of the functions within elements. The global assembly involves composing the global nodal functions $\phi_i(\mathbf{x})$ for nodes indexed by $i = 1, \ldots, n_{\text{nodes}}$ and using a mapping between the barycentric coordinates $\xi_1, \xi_2 \in [0,1]^2$ inside the elements to the global coordinates $\mathbf{x} \in S_M$. Inserting the discretization

$$u_h(\mathbf{x}) = \sum_{i=1}^{n_{\text{nodes}}} u_i \, \phi_i(\xi_1(\mathbf{x}), \xi_2(\mathbf{x}))$$

into Eq. (3.5) leads to the form

$$\sum_{i=1}^{n_{\text{nodes}}} u_i \int_{S_M} \nabla_{\mathbf{x}}\phi_i^\top \nabla_{\mathbf{x}}\phi_j \, \mathrm{d}\mathbf{x} + \sum_{i=1}^{n_{\text{nodes}}} f_i \int_{S_M} \nabla_{\mathbf{x}}\phi_i^\top \nabla_{\mathbf{x}}\phi_j \, \mathrm{d}\mathbf{x} = 0 \quad \forall j = 1, \ldots, n_{\text{nodes}}. \tag{3.6}$$

The integrations are executed element-wise and over the elemental coordinates ξ_1, ξ_2. The transformation to elemental coordinates involves the computation of the Jacobian $J = \mathrm{d}\mathbf{x}/\mathrm{d}\boldsymbol{\xi}$ of the mapping between element coordinates $\boldsymbol{\xi} = (\xi_1, \xi_2)$ and global coordinates $\mathbf{x}$. From the definition in Eq. (3.2), it follows that

$$J = \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}\boldsymbol{\xi}} = [\mathbf{p}^2 - \mathbf{p}^1, \mathbf{p}^3 - \mathbf{p}^1].$$

The metric tensor for this mapping is given by

$$\mathcal{M} := \left(\frac{\mathrm{d}x}{\mathrm{d}\boldsymbol{\xi}}\right)^T \left(\frac{\mathrm{d}x}{\mathrm{d}\boldsymbol{\xi}}\right).$$

The transformation of the integrals in Eq. (3.6) introduces an additional integration factor $\sqrt{\det \mathcal{M}}$. We get the following matrix equation:

$$M_u \mathbf{u} = -M_f \, \mathbf{f}, \tag{3.7}$$

with the vector $\mathbf{u}$ of nodal solution values, the vector $\mathbf{f}$ of nodal Dirichlet boundary condition values at the boundary and the global stiffness matrices M_u and M_f. The two global stiffness matrices are assembled from the element stiffness matrices $M^{(e)}$ for the degrees of freedom at all nodes respectively at the boundary nodes. The entries of the element stiffness matrices are given by

$$M_{i,j}^{(e)} = \int_0^1 \int_0^{1-\xi_1} \nabla\phi_i^{(e)}(\xi_1,\xi_2)^\top \mathcal{M}^{-1} \nabla\phi_j^{(e)}(\xi_1,\xi_2) \sqrt{\det\mathcal{M}}\, \mathrm{d}\xi_2 \mathrm{d}\xi_1.$$

By solving Eq. (3.7) for $\mathbf{u}$, we get the discretized harmonic map u. The finite element formulation and computation for v is analog and uses the same global stiffness matrices.

Figure 3.15 visualizes the triangulation of S_M and the solutions $u(\mathbf{x})$ and $v(\mathbf{x})$ for a circular parameter domain Ω_P and an exemplary muscle slice in the first two plots. The color range from bright yellow to dark violet corresponds to increasing values of u and v. It can be seen that the u values increase from left to right whereas the v values increase from bottom to top, corresponding to the horizontal and vertical coordinate axes y_1 and y_2 of Ω_P.

Applying the computed harmonic map $\mathbf{y}(\mathbf{x})$ to the triangulation of the slices results in a triangulation of the parameter domain Ω_P. This is shown in the third plot of Fig. 3.15 and in Fig. 3.13b. In both figures, the triangulation of the slices was generated using the second triangulation method with the constrained Delaunay triangulation. On the right side of Fig. 3.13b, the image $\mathbf{y}(\mathbf{x})$ under the harmonic map of the triangulation in the slices is shown on the unit circle parameter domain.

3.4.4 Construction of a Regular Grid in the Parameter Domain

The next step in Alg. 1 is the construction of a 2D structured, regular grid in the parameter domain Ω_P, as stated in line 5. This grid will then be mapped to the slices S_M. Creating a structured grid of quadrilateral elements in a given domain is also called *quadrangulation*.

The parameter domain Ω_P can be selected to be either a unit square or a unit circle. For both choices, two different schemes how to generate a grid with a given number of cells can be selected. Figure 3.16 shows all four possibilities.

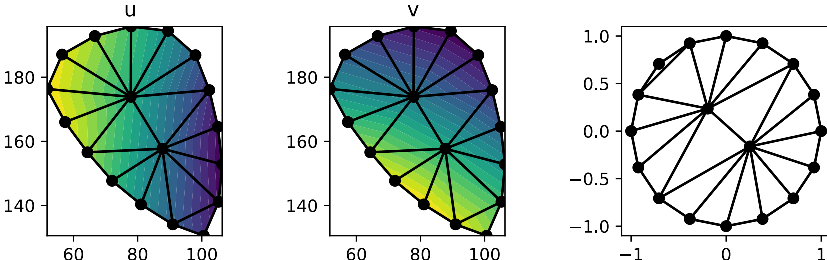


Figure 3.15: Quality improvement of muscle slice meshes as a basis for 3D mesh generation: Initial triangulations and harmonic map for a slice S_M of the biceps muscle. The first two plots show the solutions of u and v on the slice S_M. The third plot shows the image in Ω_P of the triangulation in S_M under the harmonic map.

The first scheme, Fig. 3.16a, uses an equidistant regular grid in a unit square. This is the easiest possibility to generate a quadrangulated reference domain. A possible issue is induced by the corners of the square. The grid will be mapped to a cross-section of the muscle which has no sharp corners. Therefore, the cells of the grid will be distorted at the images of the corners, usually shortening diagonals that point towards the corners and lengthening the other diagonals. This assumption motivates the second scheme in Fig. 3.16b. Here, the elements are already distorted in the described manner, with increasing distortion closer to the corners. The rationale is that the mapped cells in S_M will then be less distorted.

We construct our second quadrangulation scheme of the unit square as follows. The diagonals of the square divide the domain into bottom, top, left and right quarters, which are considered separately. For example, the bottom quarter is the triangle that is formed by the corner points $(0,0), (1,0)$ and the center point $(\frac{1}{2}, \frac{1}{2})$ of the square. In the bottom quarter, the horizontal x coordinate of a point (x, y) in a uniform grid can be described by $x = \frac{1}{2} + \tan(\phi)(\frac{1}{2} - y)$ where ϕ is the angle between a line through (x, y) and the center point $(\frac{1}{2}, \frac{1}{2})$ and the y-axis. The points of the adjusted grid in the quadrangulation scheme are constructed by altering the value of ϕ. On every horizontal series of points in the bottom quarter, ϕ is varied linearly in $[-\pi/4, \pi/4]$ instead of the nonlinear progression according to the actual angle. This leads to the larger spacing between points near the diagonals. All four quarters are treated analogously to produce the shown symmetric pattern.

A different approach is to use a unit circle, which has no corners and therefore might

resemble a muscle cross-section more consistently. The first scheme of the unit circle is given in Fig. 3.16c. It uses the radial and circumferential directions for the two dimensions of the grid. A disadvantage of this scheme is that the quadrilaterals at the center are degenerated to triangles. Additionally, the area of the cells varies significantly and the outer cells have unequal side lengths.

To remedy this problem, we develop the second scheme given in Fig. 3.16d. When traversing from the outer boundary towards the center point and considering the circumferential lines of grid points, the circle morphs into a square as the number of grid points decreases. This approach has the disadvantage that some cells have an inner angle of nearly 180°, especially four elements at the boundary. Apart from that, all elements have similar sized sides and angles. The construction of this scheme is similar to the approach of scheme 2 on the unit square in that the domain is also divided into four quarters. However, different formulas for the point coordinates (x, y) depending on the angle ϕ are used. The detailed construction formulas of all four presented quadrangulation schemes are provided by their implementation in the code. The script `plot_quadrangulation_schemes.py` constructs and visualizes the four schemes with a configurable number of grid points.

Each construction scheme allows to specify the (squared) number of nodes and in consequence the number of cells. The examples in Figures 3.16a, 3.16b, and 3.16d have 11×11 nodes and 10×10 cells. For Fig. 3.16c, the numbers are slightly different. There are $10 \times (11 + 1)$ nodes resulting in 10×11 cells.

Next, the grid in the parameter domain is transferred to the muscle domain by applying the harmonic map $\mathbf{y}(\mathbf{x}) \in S_M$ on every point of the quadrangulation $\mathbf{x} \in \Omega_P$. This is illustrated in Fig. 3.13c for a parameter domain consisting of the unit circle, with quadrangulation scheme 2 and 5×5 nodes. The cells of the grid in Ω_P are shown in the right-most stack of domains. The grid points are visualized left of the grids. The resulting image of the mesh in the slices S_M is shown on the left. For visualization reasons, each quadrilateral has been split into two triangles.

3.4.5 Formation of Three-Dimensional Elements

The result of the previous steps is a number of quadrangulated muscle slices. The grid on every slice has the same number of nodes and elements. The nodes on the boundary of neighboring slices are positioned similarly.

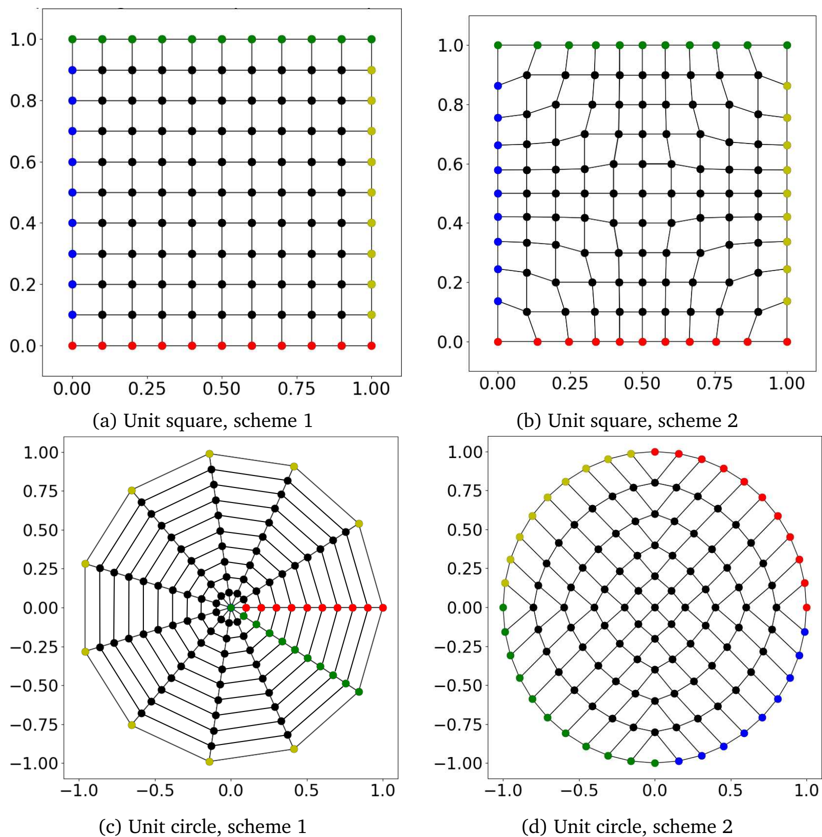


Figure 3.16: Four different quadrangulation schemes of the parameter domain with 11×11 nodes. The boundaries of the grid are colored for better perceptibility. In (a) and (b), the parameter domain is a unit square with a uniform grid (a) and an adjusted grid (b) that tries to reduce the problem of degenerate elements at the corners of the muscle slices. In (c) and (d), quadrangulations on a unit circle parameter domain are shown. (c) shows a rotationally symmetric construction scheme whereas the approach in (d) is similar to a uniform grid.

The final step of Alg. 1 is line 6, the formation of 3D elements. Inserting vertical edges between all corresponding nodes on two neighboring slices creates a set of 3D hexahedral elements and, thus, an overall 3D hexahedral mesh of the muscle volume. This step is visualized in Fig. 3.13d.

3.4.6 Generation of Fiber Meshes

1D fiber meshes are created following the approach of computing a divergence free vector field introduced in [Cho13]. The steps are given in Alg. 2.

The Laplace problem to be solved can be stated as

$$\Delta p(\mathbf{x}) = 0 \quad \text{for } \mathbf{x} \in \Omega_M. \tag{3.8}$$

The vector field is given by the gradient ∇p of a solution p of Eq. (3.8). The quantities can be interpreted as pressure p and (negative) velocity field ∇p of a steady flow. The muscle fibers are given as streamlines or, equivalently, pathlines in this velocity field. Every streamline $\mathbf{x} : [-c_1, c_2] \subset \mathbb{R} \to \Omega_M$ with $c_1, c_2 > 0$ is defined by a seed point $\mathbf{x}_0$ and the property that it is tangent to the velocity field at any point:

$$\mathbf{x}(0) = \mathbf{x}_0, \quad \frac{\partial \mathbf{x}(s)}{\partial s} = \nabla p\big(\mathbf{x}(s)\big).$$

As proposed by [Cho13], Neumann boundary conditions can be specified for the bottom and top surfaces of the muscle volume, $\partial\Omega_{M,\text{bottom}}$ and $\partial\Omega_{M,\text{top}}$:

$$\begin{aligned} \frac{\mathrm{d}p(\mathbf{x})}{\mathrm{d}\mathbf{x}} \cdot \mathbf{n} &= F_\text{in}, && \text{for } \mathbf{x} \in \partial\Omega_{M,\text{bottom}}, \\ \frac{\mathrm{d}p(\mathbf{x})}{\mathrm{d}\mathbf{x}} \cdot \mathbf{n} &= F_\text{out}, && \text{for } \mathbf{x} \in \partial\Omega_{M,\text{top}}. \end{aligned} \tag{3.9}$$

The in and outflow values $F_\text{in} < 0$ and $F_\text{out} > 0$ are balanced such that the total inflow $F_\text{in} \cdot \mu(\partial\Omega_{M,\text{bottom}})$ compensates the total outflow $F_\text{in} \cdot \mu(\partial\Omega_{M,\text{bottom}})$. Here, $\mu(\partial\Omega)$ is the surface area of the respective boundary.

Alternatively, Dirichlet boundary conditions can be specified:

$$\begin{aligned} p(\mathbf{x}) &= 0, && \text{for } \mathbf{x} \in \partial\Omega_{M,\text{bottom}}, \\ p(\mathbf{x}) &= 1, && \text{for } \mathbf{x} \in \partial\Omega_{M,\text{top}}. \end{aligned} \tag{3.10}$$

The specification of Dirichlet boundary conditions has the same effect as Neumann boundary conditions and is easier to define. The in and outflows are still orthogonal to the boundary because the prescribed value of p does not vary in the planar boundary.

The boundary value problem given by Eqs. (3.8) and (3.9) or Eqs. (3.8) and (3.10) is discretized by the finite element method with linear or quadratic ansatz functions and solved by our software OpenDiHu using the 3D mesh generated from Alg. 1. The divergence free gradient field is visualized in Fig. 3.17a. The gradient values are directly given by the finite element discretization. The gradient is elementwise constant for linear ansatz functions and trilinear for quadratic ansatz functions.

The next step in Alg. 2 is line 3, tracing streamlines through the gradient field. Seed points are selected on the 2D cross-section at the vertical center of the 3D muscle domain. The seed points are sampled regularly on the square or circular parameter domain according to the quadrangulation scheme and then mapped to the respective muscle slice. Because the 3D mesh was created using harmonic maps, the resulting spacing between the seed points is very uniform.

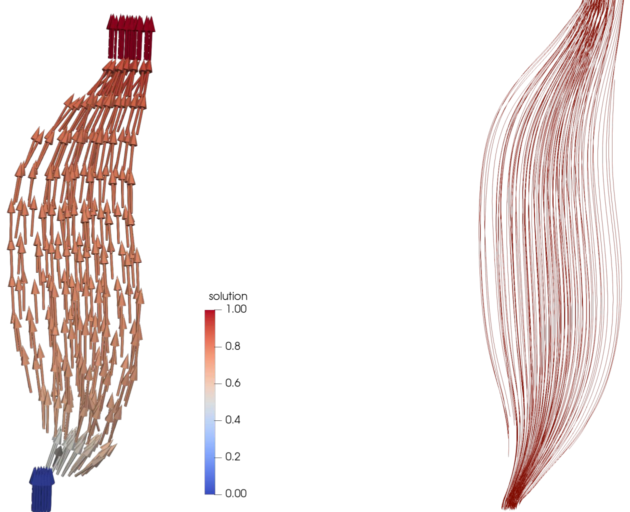
For the tracing of streamlines, the semi-analytical Pollock's method [Pol88] is often used, which was originally developed for fixed 2D finite difference grids. Extensions to irregular 3D grids and for given velocities at nodes instead of fluxes over faces have been formulated [Hæg07]. Other, more accurate algorithms exist [Cor92], including higher order formulations [Jua06].

Because modeling muscle fascicles is only a heuristic approach, the generated streamlines do not have to be exceptionally accurate. Therefore, we use a fully numerical method. The streamlines are generated by explicit Euler integration of the gradient vectors in top and bottom direction. A small spatial step width of $h = 10^{-2}$ is used. Details of the algorithm are given in the next section, Sec. 3.4.7. In line 4 of Alg. 2, all generated streamlines are resampled to obtain the desired widths of the 1D elements. Figure 3.17b visualizes the resulting streamlines in the biceps muscle.

Algorithm 2 Serial algorithm

```
1 procedure Create_1D_meshes
  Input: Structured 3D volume mesh
  Output: 1D fiber meshes

2   Solve Laplacian flow problem
3   Trace streamlines in the gradient field
4   Resample 1D fiber meshes
```



(a) Solution (color coding) and direction vectors of the gradient field for the boundary value problem Eq. (3.8) with Dirichlet boundary conditions Eq. (3.10).

(b) Resulting streamlines that were traced through the gradient field.

Figure 3.17: Setup and solution of the Laplace problem for the biceps geometry that is used to estimate muscle fibers by streamline tracing.

3.4.7 Algorithm for Streamline Tracing

The algorithm for streamline tracing uses an efficient method to traverse the mesh, which makes use of its structuredness. At first, the element $E^{(0)}$ in the mesh that contains the first seed point $\mathbf{p}^0$ needs to be found. By construction of the mesh generation algorithms, this is always the element with the lowest index. However, if the seed point is not found there, the scheme is robust enough to search in all other elements.

Starting from the seed point $\mathbf{p}^0 = \mathbf{p}^{(i)}$ in element $E^{(i)}$, the next point $\mathbf{p}^{(i+1)}$ of a streamline is computed as

$$\mathbf{p}^{(i+1)} = \mathbf{p}^{(i)} + h\,\nabla p(\mathbf{p}^{(i)}).$$

After $\mathbf{p}^{(i+1)}$ has been computed, the mesh element $E^{(i+1)}$ where it is located needs to be identified. This is needed to evaluate the gradient value $\nabla p(\mathbf{p}^{(i+1)})$ at the new point by

interpolation according to the FE representation of p.

At first, the element $E^{(i)}$ of $\mathbf{p}^{(i)}$ is checked whether it also contains $\mathbf{p}^{(i+1)}$. If not, the neighboring element in the direction of the streamline is considered. This neighboring element is chosen among all up to 26 possible neighbors such that the direction from the previous element $E^{(i-1)}$ to the current element $E^{(i)}$ continues.

If this element is also not the right one, all other neighbors of $E^{(i)}$ are subsequently checked, ordered by their plausibility according to the previous streamline direction. If none of the 27 considered elements contains the computed point, a search among all elements of the entire mesh is performed. This case happens only for unsuited choices of the integration width h, i.e., if the streamline tracing skips whole elements.

The end of a streamline is detected when the streamline reaches the final z plane, either at the bottom or top of the muscle volume. To make the algorithm more robust, also the case is considered where the streamline leaves the muscle domain to the side shortly before reaching the end of the muscle. This can happen due to discretization errors for streamlines that start close to the boundary of the muscle. In such a case, the missing rest of the streamline is interpolated from up to four existing neighboring parallel streamlines.

After the end of the streamline is found, tracing of the next streamline starts at the next seed point $\mathbf{p}^0_{\text{next}}$. The element where $\mathbf{p}^0_{\text{next}}$ is located can also be easily determined in the structured mesh.

The presented scheme avoids repeatedly traversing all elements of the mesh by predicting the next elements according to streamline direction and organization of seed points. This is facilitated by the structured mesh, which has well-defined element neighbor relations. At the same time, the scheme is robust enough to also efficiently handle streamlines in other use cases. It can also be reused, e.g., in muscle fiber tracing applications of more complex shaped muscles where the fibers change directions.

3.4.8 Results and Discussion

The presented Algorithms 1 and 2 generate a 3D mesh and 1D muscle fibers from a triangulated surface. Three different triangulation strategies for the slices and four different reference quadrangulations can be chosen. In the following, the different choices are evaluated.

The different triangulation methods for the slices discussed in Sec. 3.4.2 are visualized in the three columns of Fig. 3.18. The top rows show the triangulation of S_M and the harmonic map u as color coded values from violet to yellow. u is the horizontal coordinate on the reference domain. A point with violet color in S_M will be mapped to the left-most point in the parameter space Ω_P. Similarly, a yellow point will be mapped to a point far at the right in Ω_P.

The middle and bottom rows in Fig. 3.18 show the image of the triangulation in Ω_P under the harmonic map, for the unit square and the unit circle, respectively. The mapping between the colored boundary points stems from the Dirichlet boundary conditions in the formulation of the harmonic map. In consequence, the boundary points are by construction equally spaced both in the muscle domain S_M and in the parameter domain Ω_P.

It can be seen that the triangulation appears distorted in the parameter domain. The effect is most significant for the square in Fig. 3.18b and Fig. 3.18c. For the latter, the center point of the muscle domain S_M gets mapped far off the center of the squared parameter domain. This effect does not occur for the circle.

The reason for this lies in the triangulation of S_M together with the boundary shape. In the third method, only the value at the center point is a degree of freedom in the computation of u while the values at the boundary points are fixed. By the triangulation, the value of u varies linearly from the boundary towards the center point. By comparing the colored boundary points in the muscle slice in the top row with the square in the second row, it can be seen that the prescribed values for u are 0 at all blue points, 1 at all yellow points and linearly increasing from 0 to 1 at the green and red points, increasing from left to right. The yellow points of S_M with the prescribed constant value of $u = 1$ are approximately located on a vertical line. The first two derivatives of u in vertical coordinate direction are therefore almost zero, in consequence, the Laplace equation forces the derivatives in horizontal direction to also be approximately zero. Therefore, the solution value at the center point is close to 1. This leads to the mapped center point being close to the right boundary in the square parameter domain. The same happens for the vertical coordinate v of the harmonic map.

For the circle, neighboring boundary points are not located on horizontal or vertical lines and, thus, the Dirichlet boundary conditions for u and v vary along the boundary. Therefore, a better mapping is obtained. The shape of the muscular slice is more similar to the circle than to the square.

Another result that can be seen in Fig. 3.18 is the effect of the failure of the first method to handle concave slices on the harmonic map. As the top left image shows, the first triangulation method produces triangles outside the domain. The triangles are located around the red boundary points. In the square parameter domain, these triangles are degenerated and lie on the bottom boundary. In the circle parameter domain where the respective triangles can be seen at the bottom, they even intersect other triangles. This yields an invalid triangulation.

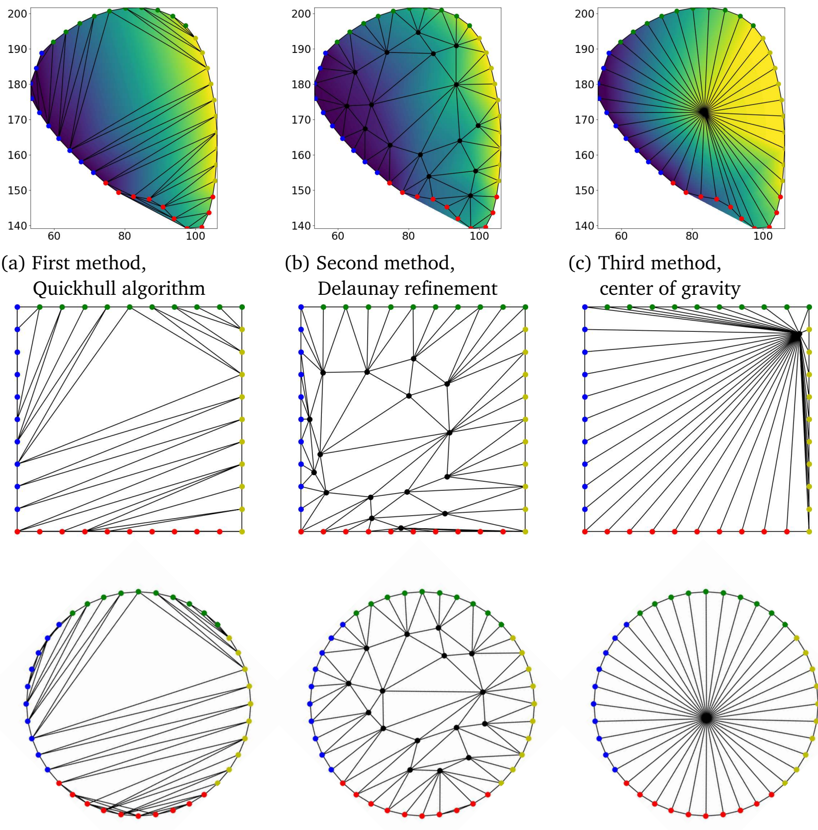
The reasons for degenerate triangles in the square parameter domains are not solely the concave muscle slices. Also, the straight sides of the unit square lead to degenerate triangles whenever three boundary points of the same side form a triangle. In the example in Fig. 3.18, this occurs for the square in column (a). As can be seen in the triangulated slice in the top row, there are three triangles that are entirely made up of blue boundary points. These triangles get mapped onto the left side of the square parameter domain where they have a vanishing surface area.

The same effect also occurs with the second triangulation method in column (b) of Fig. 3.18 where a triangle at the bottom right comprises three red boundary points and, therefore, gets mapped to the bottom side of the square. Because of the guaranteed minimum angle in the second triangulation method, this circumstance occurs less often and only for muscle cross-sections where the boundary makes sharp turns, such as the muscle slice in this example. The third triangulation method is guaranteed to avoid this problem as all triangles are connected with the center point.

In conclusion, the second triangulation method with the unit circle and the third triangulation method with both unit square and unit circle show good behavior for use in our meshing algorithm. Next, their interplay with the quadrangulation of the parameter space needs to be investigated.

In the next step, the algorithm creates a quadrilateral mesh in the parameter domain and computes its image in the muscle domain using the inverse of the harmonic map. The results are shown in Fig. 3.19 for the three different initial triangulation methods (columns) and the four different schemes to create the quadrilateral mesh (rows).

In the images in column (c) and rows (d) and (e), it can be seen that the previously observed effect of a bad mapping for squares and the third triangulation method also leads to a mesh in S_M of poor quality. The result for the two square schemes in column (b) is better but still not satisfactory. Good results with the square reference domain are only observed for the first triangulation method in this example.



(a) First method, Quickhull algorithm

(b) Second method, Delaunay refinement

(c) Third method, center of gravity

Figure 3.18: Initial slice triangulation and harmonic maps in 3D mesh generation: Top row: Different triangulation methods for S_M, the color represents the solution u of the harmonic map. Middle and bottom row: triangulation mapped to the parameter domain Ω_P, for the unit square (middle) and the unit circle (bottom). Each column corresponds to one triangulation method.

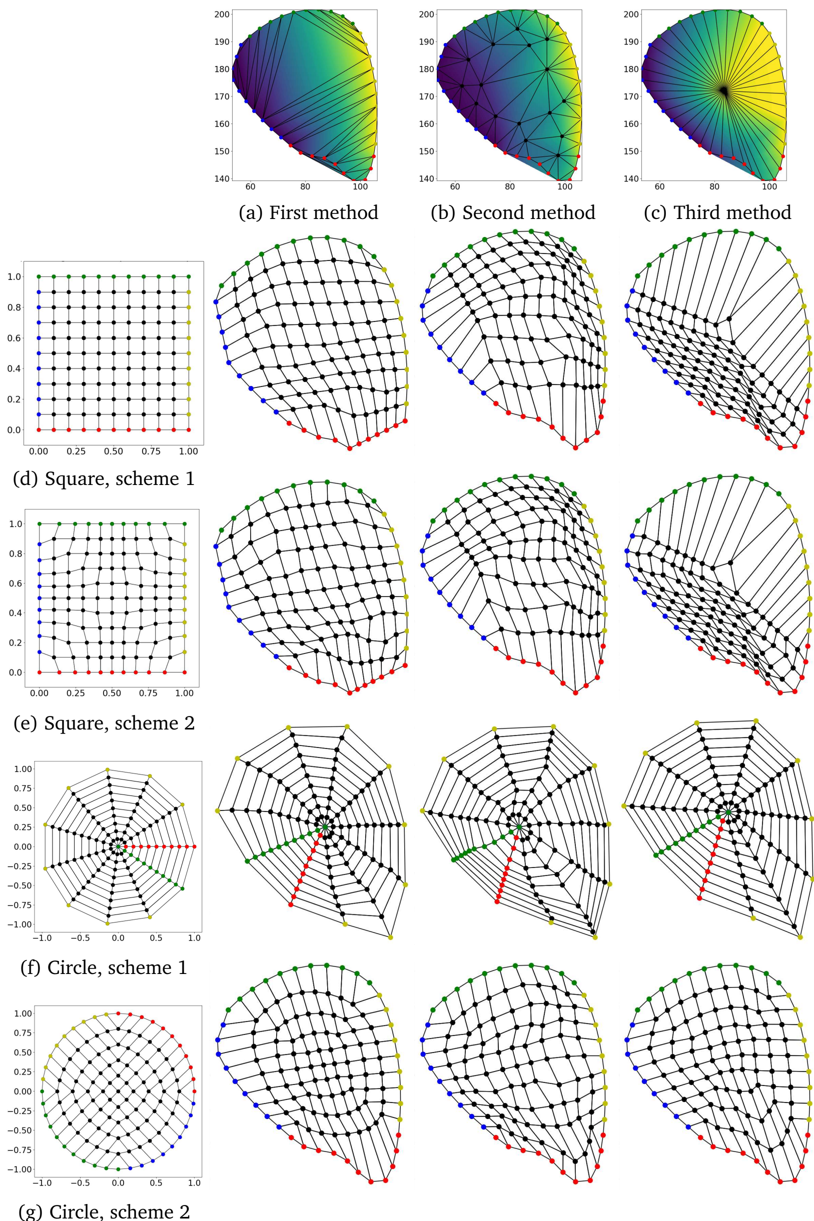
It can be seen that the approximation quality of the boundary of the domain varies. Most of all, the combination of the first triangulation method (column (a)) and the square parameter domain (rows (d) and (e)) reproduces the shape of the slice poorly. The mismatch occurs at the blue and red boundary points. Additionally, the second triangulation method (column (b)) for the squares fails to correctly represent the round boundary at the bottom of the domain. The degenerate triangles in the parameter domain are the cause for this effect. The harmonic map $\mathbf{y} : S_M \to \Omega_P$ is not injective and, therefore, its inverse does not exist. In our implementation, the points on the degenerate triangles in Ω_P are mapped to an arbitrarily selected location inside the corresponding triangles in S_M. Thus, the mapping is correctly inverted at locations of valid triangles and only creates different boundary points in the invalid areas.

Furthermore, it can be seen that an inaccurate representation of the boundary also occurs with the parameter mesh in the unit circle generated by scheme 1. In this case, the reason is the low number of elements at the boundary in the parameter domain quadrangulation.

The two schemes for the circle parameter domain in rows (f) and (g) both generate reasonable meshes for all triangulation methods, despite the different structure of the generated meshes. The best results for both schemes have been obtained with the third triangulation method.

Next, a quantitative comparison of the resulting mesh quality for different parameters of the presented algorithm is carried out. The algorithm was executed for all variants with 43 slices of the biceps muscle, resulting in 43 meshes for every combination of triangulation method and quadrangulation scheme. To assess the quality of the generated meshes, the edge lengths of the elements were collected and normalized to have a mean of 1 in each mesh. The normalization was done to allow for a comparison between meshes with different bounding box sizes. The standard deviation of the normalized lengths was determined in each mesh. The total mean of all standard deviations was computed. This value is a measure for the quality of the mesh. A low value means that, in every slice, the generated mesh has similar edge lengths and, in consequence, the overall mesh has good quality.

Figure 3.20 visualizes the results. Three groups of bars are displayed for the three triangulation methods. For every type of mesh in the parameter space, i.e., unit square (□) or unit circle (○) and scheme 1 or 2, the standard deviation is given by the red bar and the generation runtime of the overall algorithm is given by the yellow bar. The



(a) First method (b) Second method (c) Third method

(d) Square, scheme 1

(e) Square, scheme 2

(f) Circle, scheme 1

(g) Circle, scheme 2

Figure 3.19: Initial triangulations, harmonic maps and final quadrangulations of muscle slices for 3D mesh generation: Meshes in the muscle slice S_M for quadrangulations (rows) and triangulations (columns).

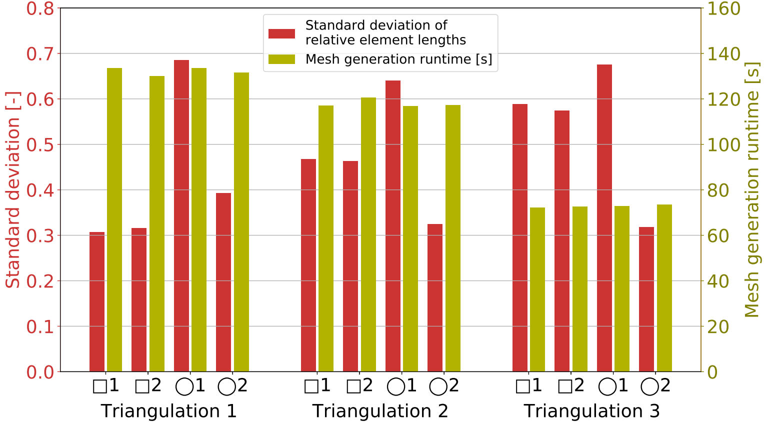


Figure 3.20: 3D mesh generation quality assessment: Mesh quality (red, lower is better) and generation runtime (yellow) for the three different triangulation methods and the different parameter space quadrangulation schemes. □1 and □2 designate the two quadrangulation schemes on the unit square parameter domain, ○1 and ○2 are the schemes on the unit circle parameter domain, as introduced in Fig. 3.16. A low value for the standard deviation of relative element lengths indicates good quality.

corresponding axis labels for standard deviation and duration are given on the left and right of the diagram.

The diagram shows the lowest standard deviation of edges and therefore the best mesh quality for the first triangulation method and the square (□1 and □2), with scheme 1 having a slightly better value than scheme 2. This shows that the modified placement of the nodes in scheme 2 has no positive effect compared to scheme 1. However, from the observations in Fig. 3.18, it is known that the boundaries are not represented correctly. This behavior does not influence the result because of the chosen metric of uniform relative edge lengths. Similarly, good results can be seen for scheme 2 in the circular parameter domain and the second and third triangulations.

Moreover, it can be seen that certain connections between parameter domain and suited triangulation scheme exist. The square parameter domain works best with the first triangulation method. The second scheme for the parameter mesh on the unit circle works best with the triangulation methods 2 and 3.

The first scheme for the parameter mesh on the unit circle (○1) shows bad results for all triangulation methods. This can be explained by looking at the generated meshes in row (f) of Fig. 3.19. By construction, the elements have a bad aspect ratio. This results in the high standard deviation values. However, the generated meshes still look uniform to a certain extent and can be useful in applications where such type of mesh is needed. The score could be improved by adding more nodes in circumferential direction.

The runtime of the algorithm is approximately the same for the different parameter domain meshes. It mainly depends on the triangulation of the slices. The first triangulation using the *SciPy* package takes the most time, followed by the Delaunay refinement. The fastest triangulation is the custom one where only one additional point needs to be placed. In conclusion, when runtime is an issue, the third triangulation should be chosen. It achieves good quality meshes only with the second scheme of the circular parameter domain. This combination also does not suffer from the bad approximation quality of the boundary, as is the case for the unit circle with the first triangulation method.

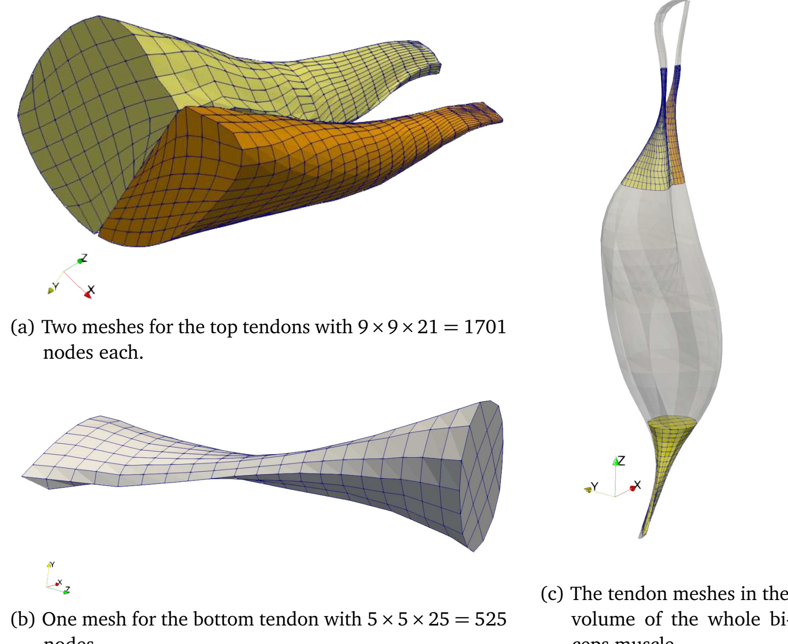
Figure 3.21 shows three structured meshes $\Omega_{T,i}$ for the tendons of the biceps brachii muscle that were created using Alg. 1. The tendon at the bottom of the muscle is represented by a single mesh. At the top, there are two tendons that extend the two muscle heads of the biceps. Because the meshes need to be structured, two tendon meshes are created at the top. It can be seen that the algorithm creates meshes with similar sized elements despite the difficult, wound geometry of the surfaces.

How To Reproduce

The described algorithms are part of the `fiber_tracing` examples. Execute the following commands to get the results in this chapter:

```
cd $OPENDIHU_HOME/examples/fiber_tracing/streamline_tracer/scripts
. run_evaluation.sh
```

Then, the visualizations will be created under `../processed_meshes`. Create Fig. 3.20 with `plot_mesh_quality.py`. How to create the tendon meshes is explained at the end of Sec. 3.6.6.



(a) Two meshes for the top tendons with $9 \times 9 \times 21 = 1701$ nodes each.

(b) One mesh for the bottom tendon with $5 \times 5 \times 25 = 525$ nodes.

(c) The tendon meshes in the volume of the whole biceps muscle.

Figure 3.21: 3D mesh generation results: Tendon meshes that were created using the serial algorithm for mesh creation.

3.5 Parallel Algorithm to Create Muscle and Fiber Meshes

The previously presented algorithm to create 3D and 1D meshes is not parallelized. Thus, the size of the handled meshes is limited by the available memory of the computer. An algorithm that can be used with distributed memory parallelization could, in contrast, benefit from more total memory that is accessible at different compute nodes. Furthermore, the tracing of the streamlines could be performed in parallel which has the potential to reduce runtimes.

In the following, we present an extended algorithm based on the one presented in Sec. 3.4 that can be run in parallel on multiple cores. The extended algorithm employs a partitioning of the 3D volume. Every process only stores data corresponding to its own

partition. This allows to run the algorithm on a distributed memory system, where data transfer between the processes occurs by sending messages using the Message Passing Interface (MPI). It is possible to create meshes with larger sizes than could fit into a single nodes' memory. This enables us to run the algorithm for meshes with very high resolution that can be used for simulations in the field of High Performance Computing. These meshes are partitioned into subdomains for every compute core and can be read from and written to disk concurrently.

3.5.1 Overview of the Parallel Algorithm to Create Muscle and Fiber Meshes

The steps of the algorithm and its input and output are given in Alg. 3. Input and output are the same as for the Alg. 1 presented in Sec. 3.4. The input is a triangulated tubular surface of the muscle that can be obtained as described in Sec. 3.3. A second input, the variable called *boundary_points*, is used only during recursive calls and is not set at the beginning. The output consists of the 3D mesh of the muscle volume Ω_M and embedded 1D fiber meshes $\Omega_{F,i}$.

During execution of the algorithm, the 3D mesh of the muscle is recreated iteratively with increasing resolution and increasing number of subdomains. The algorithm is formulated recursively. At the finest resolution when the recursion terminates, the fiber meshes are finally generated together in all subdomains.

At first, a single process executes all the steps of Alg. 3 from lines 2 to 11. This corresponds to recursion level $\ell = 0$. Then, in line 12, the procedure is called again and in the first recursion executed by eight processes with eight subdomains. On the ℓth recursion level, the number of involved processes and subdomains is 8^ℓ. After a specified maximum recursion depth $\ell_{\max}$ is reached, all involved processes execute the first branch of the **if** statement in line 8 and generate the final 3D and 1D output meshes in line 9.

The steps in Alg. 3 are executed concurrently by the involved processes at the respective levels. Some of the steps only operate on the locally stored data and, thus, are independent of other processes. Other steps involve communication between processes. Whether an instruction effects only the own domain of the process or involves global communication is denoted in parentheses at the beginning of the lines in Alg. 3.

Algorithm 3 Parallel algorithm to create muscle and fiber meshes

```
 1 procedure Create_3D_meshes_parallel
   Input: Triangulated tubular surface
   Input: boundary_points:  4 × 4 points per slice
   Output: Structured 3D volume mesh
   Output: 1D fiber meshes

 2   (own domain) Create_3D_mesh(boundary_points)
 3   (own domain) Fix and smooth 2D meshes
 4   (global)     Solve Laplace problem
 5   (global)     Communicate ghost elements to neighboring subdomains
 6   (own domain) Trace streamlines for new subdomain boundaries
 7   (global)     new_boundary_points ← Construct new subdomains

 8   if recursion ends then
 9     (own domain) Trace streamlines for fiber meshes
10   else
11     (global)     communicate boundary points
12     (global)     Create_3D_meshes_parallel(new_boundary_points)
```

3.5.2 Overview of the Subdomain Refinement

The goal during the recursive calls is to determine smooth boundaries for the new subdomains. Each process splits its own subdomain into eight subdomains and then proceeds to the next recursion level. The subdomain boundaries are determined by tracing streamlines in a divergence-free vector field through the entire muscle volume, similar to the approach in Alg. 2. The divergence-free vector field is computed from the solution of a Laplace problem, which is solved in parallel on the entire mesh of the muscle in every recursion. The mesh width of this mesh gets halved in every recursion, subsequently leading to an increasingly fine mesh.

The subdomain boundaries are always aligned to streamlines in the mesh that was created last. On each recursion level, the existing subdomain boundaries and the new boundaries for the subdomains on the next recursion level are all created anew and, thus, change slightly as the mesh refines.

As the subdomain boundaries in the interior of the volume refine, so do the outer boundaries given by the surface of the muscle. The given triangulation of the surface is sampled again on each recursion level yielding increasingly fine representations.

At the final recursion level $\ell_{\max}$, the muscle is partitioned into $8^{\ell_{\max}}$ subdomains and a respective fine 3D mesh in the muscle volume exists. Then, the algorithm traces the

specified amount of streamlines through the whole mesh to produce 1D meshes for the muscle fibers. By construction of the subdomains, the streamlines enter and leave the subdomains through their top and bottom bounding planes. This allows parallel execution of the final streamline tracing step.

The reason that the algorithm constructs the partitioning iteratively and not once at the beginning using an initial mesh lies in the requirements for parallel streamline tracing. Each subdomain should be able to trace streamlines in longitudinal (z) direction of the muscle without communication to their neighbors in x and y directions. To ensure this property, the partitioning involves a small overlap of neighboring subdomains, i.e., a ghost layer. This ghost layer can consist of a lower number of elements if the mesh is iteratively refined than if the partitioning was created directly on a coarser mesh.

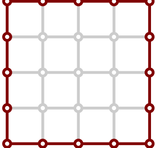
3.5.3 Data Structure of Boundary Points

In the following, Alg. 3 is illustrated in more detail. The execution starts with one process and the only input is the tubular muscle surface. It is given either as triangulation or in parametric form as NURBS surface. The first step is to construct a quadrilateral mesh of this surface. This is done using the procedure explained in Sec. 3.4.1, which creates horizontal slices of the muscle and places equidistant points on the "rings" of the boundaries of these slices. As explained earlier, the points are arranged such that the resulting quadrangulation of the surface has good quality. A result for the biceps muscle is visualized in Fig. 3.13a.

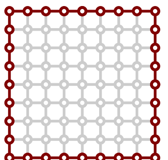
Initially, the parallel algorithm stores the points on these rings in the variable `boundary_points`. If the procedure in Alg. 3 is called recursively, the contents of this variable is passed as an argument from the previous recursion. The set of points in `boundary_points` defines the boundaries of the subdomain of the process where it is stored.

The points on each ring in the x-y-plane are organized such that they enclose a grid of $n_{\text{el,x}} \times n_{\text{el,x}}$ elements, where the number $n_{\text{el,x}}$ of elements per coordinate direction can be specified as parameter. In the following, an example with $n_{\text{el,x}} = 4$ is considered. The grid is shown in Fig. 3.22a, the $4\,n_{\text{el,x}}$ boundary points on the ring are visualized by red color. Note that Fig. 3.22a depicts the ring as a square whereas in reality it has the potentially more irregular shape of the subdomain.

A number $(n_{\text{el,z}}+1)$ of these rings are stacked in z direction to approximate the enclosing surface of the subdomain, where the number $n_{\text{el,z}}$ of elements in z direction is again given



(a) Grid of 4×4 boundary points, which occurs at the beginning of the procedure of Alg. 3.



(b) Grid with 4×8 boundary points, which occurs after a refinement step at beginning of the procedure of Alg. 3.

Figure 3.22: Logical subdomain boundaries (red) and interior grid (gray) before and after the refinement at the beginning of Alg. 3.

by a parameter. Thus, the variable `boundary_points` contains a total of $4\,n_{\text{el,x}}\,(n_{\text{el,z}}+1)$ points. Typical parameter values are $n_{\text{el,x}} = 4$ and $n_{\text{el,z}} = 50$.

As the goal on every recursion level is to construct a mesh with half the mesh width of the mesh on the previous level, the given boundary points are refined to twice the amount by inserting new points at the centers between neighboring points. The refinement happens in all three coordinate directions. For the example with $n_{\text{el,x}} = 4$, the resulting grid with the 4×8 refined boundary points is shown in Fig. 3.22b. In z direction, we get $(2\,n_{\text{el,z}}+1)$ slices with points.

The task in the recursive procedure is now to determine boundaries for eight subdomains. This is achieved by subdividing the given 2D slices into four 2D subdomains each. Additionally, the 3D volume is split at its vertical center. Thus, the upper and lower parts contain four subdomains each. Figure 3.23 visualizes this scheme for the eight subdomains on recursion level $\ell = 1$. The boundary points of the first and the eighth subdomain are shown. The boundary points have already been refined such that every slice in Fig. 3.23 consists of 4×8 points and corresponds to the grid in Fig. 3.22b

3.5.4 Generation and Smoothing of the 3D Mesh

After the `boundary_points` variables has been set, the next step of Alg. 3 is to construct a 3D mesh in the domain. In line 2 of Alg. 3, the harmonic map algorithm Alg. 1 described in Sec. 3.4 is called. Its input consists of the boundary points that define the 2D slices of the volume. This means that Alg. 1 does not need to construct the slice boundary rings from the surface triangulation, instead, the formulation of Alg. 1 can directly start with line 3 to triangulate the slices and then compute the harmonic map. For the harmonic map

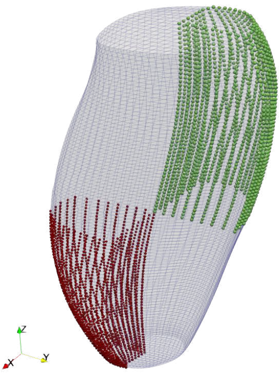


Figure 3.23: Parallel 3D mesh generation: Partitioning of the muscle volume into eight subdomains during the first call to the procedure in Alg. 3. The first (red) and the eighth subdomain (green) are shown.

computation, the second triangulation method is used with a circular reference domain quadrangulated by the second scheme. The result is a set of quadrangulated 2D slices that forms a 3D mesh by vertically connecting the elements of neighbor slices.

Next, line 3 of Alg. 3 improves the mesh quality of the 2D muscle slices S_M from which the 3D mesh is formed. This action consists of two steps. The first step is to ensure that no self-intersecting or degenerate quadrilaterals exist in the slice. The second step applies Laplacian smoothing to improve the mesh quality of the slice.

Theoretically, the first step should not be necessary, as the chosen quadrangulation algorithm always produces valid elements. However, in practice, small or irregularly shaped, concave domains occur and together with rounding and numerical errors in the Laplace problem computations occasionally lead to invalid meshes with self intersecting elements, especially for higher recursion depths in Alg. 3. Executing the first step therefore increases the robustness of the implementation.

The algorithm performs this step by repeatedly iterating over all interior mesh points in every slice S_M and fixing invalid elements. To find invalid elements, for every quadrilateral the four triangles that can be formed from the points of the quadrilateral are considered, as shown in Fig. 3.24. For every triangle with points $\mathbf{p}^0, \mathbf{p}^1$ and $\mathbf{p}^2$, the orientation of the triangle is determined. The orientation is counterclockwise if the oriented triangle

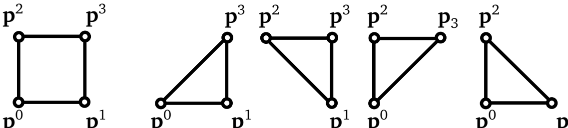
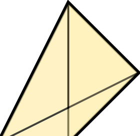
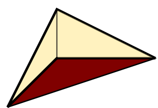


Figure 3.24: Decomposition of quadrilateral elements into triangles as substep of the validity check of muscle slice quadrangulations. A quadrilateral element (left) and the four triangles (right) that can be constructed from its four points. These triangles are needed for the check in Alg. 3 whether the quadrilateral element is valid.



(a) Convex quadrilateral with score $s = 4$ and the contained triangles, which are all oriented counterclockwise.



(b) Concave quadrilateral with score $s = 3$ and the contained triangles. Only the red triangle is oriented clockwise.

Figure 3.25: Check for valid elements in the muscle slice quadrangulations that occurs in Alg. 3: Illustration of the score of valid concave and convex quadrilaterals.

area A_{012} is positive. The oriented triangle area is the determinant of the 3×3 matrix that contains the row vectors $(p_x^i, p_y^i, 1)$ for the triangle points $\mathbf{p}^i = (p_x^i, p_y^i)^\top$ and can be computed by the following formula [Sed11]:

$$A_{012} = (p_x^1 - p_x^0)(p_y^2 - p_y^0) - (p_x^2 - p_x^0)(p_y^1 - p_y^0).$$

If the orientation is counterclockwise, a score value of the triangle is set to one, if it is clockwise, the score is set to zero. The score values of the four triangles are added up to yield a score s for the quadrilateral. Only if this score is $s \geq 3$, the quadrilateral is valid. Figure 3.25 illustrates the cases of valid quadrilateral elements. In a valid, convex element, all four triangles lie inside the element and, thus, the score is $s = 4$. If only one triangle is located outside, the quadrilateral is also valid and concave. In this case the score has the value $s = 3$.

At the current mesh point in the loop over all points that are not at the boundary of the mesh, the four adjacent quadrilaterals are considered. If any of them is invalid, the algorithm tries to improve the situation by deflecting the point by a random, small vector. A maximum of 200 random deflections from the original position with exponentially

increasing deflection vector sizes are tried. After each modification of the point, the scores of the four adjacent quadrilateral elements are evaluated. If the sum of the four element scores increases, the point is kept and the iteration over all interior mesh points starts anew.

Note that this does not necessarily mean that the invalid element was fixed, only its score was improved. If it was not fixed, it will be considered again in the next iteration. For example, a convex element that initially is oriented clockwise instead of counterclockwise has a score of $s = 0$. In the first iteration, one point is deflected such that the quadrilateral intersects itself but has a higher score $s \geq 0$. At least one more iteration is needed until the quadrilateral is oriented correctly. When all elements in the slice S_M are valid, this step is complete.

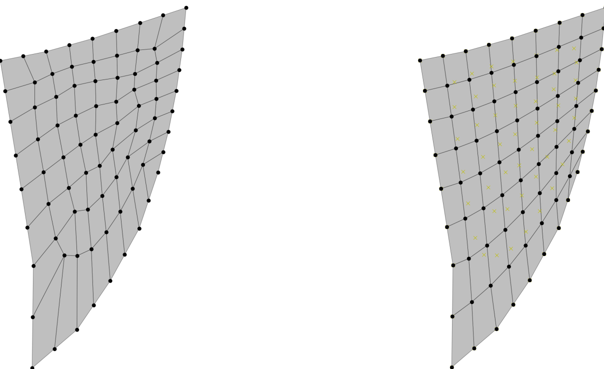
3.5.5 Laplacian Smoothing

The second step is the smoothing step that improves the mesh quality of the 2D slices. 20 iterations of Laplacian smoothing [Fie88] are executed. Laplacian smoothing in our case subsequently visits all interior points of the mesh and sets the location of a point to the center of gravity of its four direct neighbors. Figure 3.26 shows the effect of Laplacian smoothing for a slice in a subdomain on the first recursion level. It can be seen how the smoothing equalizes the element side lengths and angles.

However, this smoothing step can invalidate a mesh by introducing overlapping quadrilaterals. An example for this case is given in Fig. 3.27. The initial mesh in Fig. 3.27a is concave and occurs during recursion level $\ell = 2$. Figure 3.27b shows the result of the smoothing, which contains one invalid element. The smoothing operation placed the fourth point of the element that also contains the three boundary points at the concavity outside the mesh. As a remedy, the smoothing method checks the validity of the adjacent elements before a point is moved. If the move results in an invalid element, the action is not carried out and the traversal continues with the next point instead. Figure 3.27c shows the resulting mesh if this check is enabled. The mesh has slightly different boundaries because the check influenced the behavior already on lower recursion levels.

3.5.6 Solution of the Laplace Problem

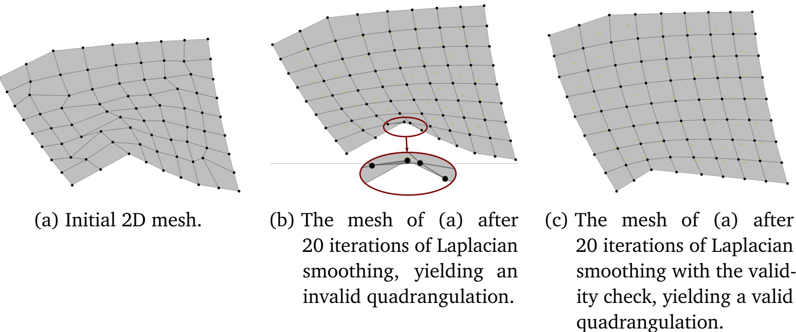
After the 3D mesh has been created and smoothened, the next steps are to solve the Laplace problem in the muscle domain, to trace streamlines in the gradient of the solu-



(a) Initial 2D mesh of a subdomain at the boundary of the biceps muscle.

(b) The mesh of (a) after 20 iterations of Laplacian smoothing.

Figure 3.26: Quality improvement of 2D muscle slice quadrangulation: Effect of Laplacian smoothing of a 2D grid which occurs in line 3 of Alg. 3.



(a) Initial 2D mesh.

(b) The mesh of (a) after 20 iterations of Laplacian smoothing, yielding an invalid quadrangulation.

(c) The mesh of (a) after 20 iterations of Laplacian smoothing with the validity check, yielding a valid quadrangulation.

Figure 3.27: Quality improvement of 2D muscle slice quadrangulation: Effects of Laplacian smoothing on concave domains.

tion vector field and finally to construct the eight subdomains for the recursive calls by subdividing the own domain along the streamlines.

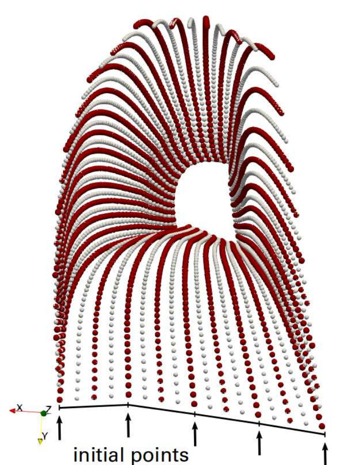
Prior to the solution of the Laplace problem, the 3D mesh gets refined further by increasing the number of elements per coordinate direction by a specified factor $r \in \mathbb{N}$. The rationale is to increase the number of degrees of freedom and, thus, the resolution to get a smaller numerical error in the subsequent Laplace computation. This refinement is in addition to the refinement of the initial boundary points by a factor of two described in Sec. 3.5.3. The mesh with $2\,n_{\text{el,x}} \times 2\,n_{\text{el,x}} \times 2\,n_{\text{el,z}}$ elements gets refined to $2\,r\,n_{\text{el,x}} \times 2\,r\,n_{\text{el,x}} \times 2\,r\,n_{\text{el,z}}$ elements. The new points are found by interpolating in the existing mesh.

For example, the 3D mesh of Fig. 3.22 with $2 \cdot 4 \times 2 \cdot 4 \times 2 \cdot 50$ elements gets refined with the factor $r = 2$ to $16 \times 16 \times 200$ elements. Figure 3.28a shows the refined boundary points in this example in a view in negative z direction towards the bottom of the muscle. The red points are the boundary points of the 4×8 grid, the additional white points are added in between by the refinement with $r = 2$. Because this refinement is carried out by interpolating between the initial points, the new points are located on straight lines between the initial points. This can especially be seen at the lower left of the figure (indicated by arrows and lines) where always five neighboring points lie on a straight line.

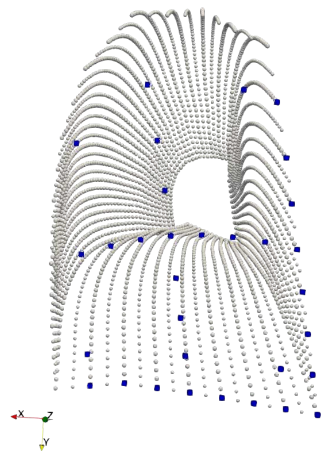
Next, in line 4 of Alg. 3 the Laplace problem gets solved. The same step also occurs in Alg. 2 and is explained in Sec. 3.4.6. The equation is formulated globally and the discretization uses the existing partitioning. Dirichlet boundary conditions of $p(\mathbf{x}) = 0$ and $p(\mathbf{x}) = 1$ are prescribed at the bottom and top of the domain, as shown by the spheres in Fig. 3.29a. Alternatively, Neumann boundary conditions can be used. A parallel GMRES solver is employed to obtain the solution in a few iterations. E.g., for the biceps muscle a linear system at $\ell = 0$ has 4131 degrees of freedom and 26 iterations are needed to obtain a residual norm below 10^{-4}. After the solution $p(\mathbf{x})$ is obtained, the gradient field $\nabla p(\mathbf{x})$ is computed. The solution and the gradient directions are visualized in Fig. 3.29b.

3.5.7 Communication of the Ghost Layer

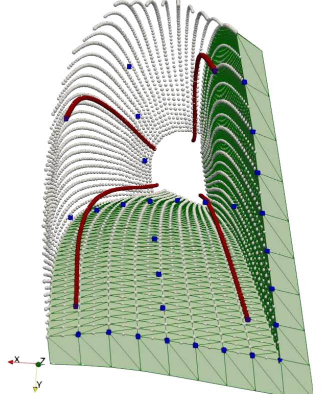
Subsequently, the gradient field $\nabla p(\mathbf{x})$ is used to trace streamlines to determine new boundaries of the subdomain. This involves tracing streamlines that start exactly on the boundary. These streamlines potentially switch between the subdomain owned by the current process and the subdomains of neighboring processes. Streamline tracing



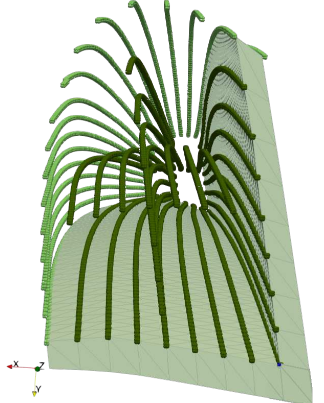
(a) Initial (arrows) and refined boundary points (red) and points after additional refinement by a factor of $r = 2$ (white).



(b) Seed points for the streamlines (blue).

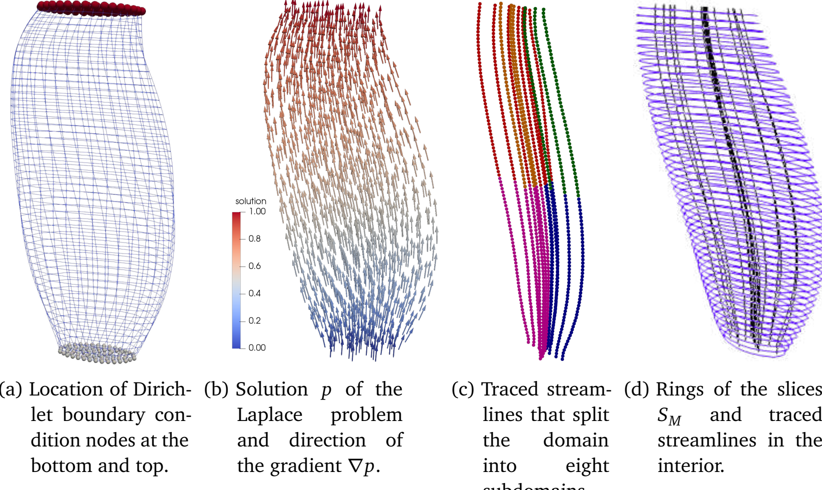


(c) The four boundary streamlines (red) and the layer of ghost elements (green) at the bottom and right of the subdomain.



(d) New boundary points on the outer (light green) and interior boundary (dark green).

Figure 3.28: Parallel generation of 3D meshes: refined boundaries, streamlines and subdomain refinement in the first subdomain for recursion level $\ell = 1$.



(a) Location of Dirichlet boundary condition nodes at the bottom and top.

(b) Solution p of the Laplace problem and direction of the gradient ∇p.

(c) Traced streamlines that split the domain into eight subdomains.

(d) Rings of the slices S_M and traced streamlines in the interior.

Figure 3.29: Parallel 3D mesh generation: Process of subdividing the muscle volume into eight subdomains using the solution of a Laplace problem, which is an important step in the procedure of Alg. 3.

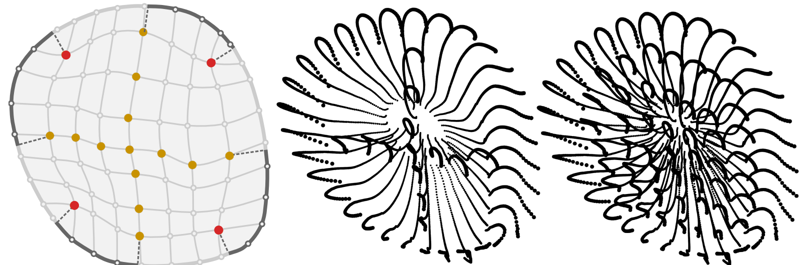
requires the gradient field values of the elements where the streamline passes through. To avoid repeated communications in these cases, a ghost layer of a specified number $n_{\text{ghost_layer_width}}$ of elements is added to the subdomains at all parts of the boundary that touch a neighboring subdomain directly or diagonally adjacent in x and y direction.

The ghost layer is constructed and the node positions and values of p and ∇p associated with the ghost elements are communicated between the neighboring processes after the solution of the Laplace problem. This occurs in line 5 of the algorithm. Figure 3.28c shows $n_{\text{ghost_layer_width}} = 1$ layer of ghost elements on a subdomain at recursion level $\ell = 1$.

3.5.8 Selection of Seed Points for the Streamlines

Next, the seed points from which the streamlines start are determined on the subdomain. All seed points are selected from the set of nodes in the structured mesh of a horizontal 2D slice.

Figure 3.30a visualizes the structured mesh in light gray in the first call to the procedure



(a) The seed points of the streamlines used to determine the subdomain boundaries.

(b) Streamlines and lines on the muscle surface that define the new subdomain boundaries.

(c) All streamlines and lines on the muscle surface that are created if the algorithm is run with one process and $\ell_{max} = 0$.

Figure 3.30: Parallel mesh generation: Seed points and streamlines that occur during the first call to the procedure in Alg. 2, in a view from the top of the muscle.

for recursion level $\ell = 0$ where the domain is not yet partitioned. The selected seed points are shown by the yellow and red points. As can be seen, the seed points consist of the nodes of the 2D mesh at the horizontal and vertical centers in this view which form the *plus sign* shape given by the yellow points. In addition, the four red points near the corners of the structured mesh are selected.

The seed points of the *plus sign* yield the streamlines that subdivide the domain into four parts in x and y direction. With the additional split in z direction, the inner boundaries of the eight subdomain are obtained. The resulting boundaries are given in Fig. 3.30b.

The streamlines of these seed points are also depicted in Fig. 3.29c. The interior boundary points for the eight subdomains that partition the muscle volume at level $\ell = 1$ are shown by different colors. The full subdomain boundaries include also the outer surface of the muscle, which is given by the rings of the muscle slices. Figure 3.29d shows the streamlines in black and the circumferential rings of the muscle slices in blue that were extracted during the call to Alg. 1 in line 2.

At higher recursion levels $\ell > 0$, the boundaries for the new subdomains consist of those at the outer boundary of the muscle defined by the surface representation and those in the interior of the muscle. Similar to the previously considered case at recursion level $\ell = 0$, for $\ell \geq 1$ the boundaries in the interior of the muscle have to be sampled by a set of streamlines. In addition to the streamlines associated with the plus sign shaped

seed points, new streamlines at the boundaries of the current subdomain have to be obtained.

Figure 3.28b shows in blue all seed points that are selected in a subdomain on recursion level $\ell = 1$ in order to create subdomain boundaries for level $\ell = 2$. As can be seen, in addition to the plus sign shape and the four outer seed points two lines of points in approximate x and y directions are selected at the lower and right edges of the image. These are seed points for the new boundaries in the interior of the muscle. Note that the current recursion level $\ell = 1$ also has boundaries at these locations. However, for level $\ell = 2$ these boundaries are recreated by the new streamlines. Tracing of these streamlines potentially uses the ghost layer. The resulting streamlines and the ghost layer for $n_{\text{ghost_layer_width}} = 1$ are shown in Figures 3.28c and 3.28d.

3.5.9 Determination of Subdomain Boundaries on the Outer Muscle Surface

Next, the boundary points on the outer surface of the muscle are determined for the new partitioning. They are obtained by sampling the circumferential rings of the muscle surface with the resolution required in the current recursion level. In our implementation, this can be done either by sampling the original surface triangulation of the muscle or by directly evaluating the parametric form of the NURBS surface that approximates the muscle surface.

At recursion level $\ell = 0$, the entire muscle surface is touched by the new subdomains. Thus, when traversing the circumference of the muscle four new subdomains are encountered. In consequence, every circumferential ring needs to be split into four quarter parts for the four adjacent subdomains. For each of these new subdomains, the quarter part corresponds to two neighboring sides of the subdomain boundary in Fig. 3.22a. Figure 3.30a also visualizes the two neighboring sides per new subdomain as dark and light portions of the outer boundary. To obtain these sides, a splitting point is needed that further splits every quarter part of the circumferential ring into the two sides for the new subdomain. In summary, the ring needs to be split into eight parts that fit to the inner subdomain boundaries.

The eight split points are determined by the eight outer streamline points. In Figure 3.30a, the four outer yellow points of the plus sign and the four red points are

considered. For each split location, the nearest point on the circumferential ring is determined. The employed algorithm for calculating the coordinates of the point on a ring that has the shortest distance to a given, second point is described in Sec. 3.4.1.

After the two sections of the circumferential rings have been determined for all new subdomains, the sections are equidistantly sampled in circumferential direction with $n_{\text{el,x}}$ elements each to create the outer boundary points for the subdomains. Also in longitudinal direction of the muscle, i.e., the z axis, points are sampled on each streamline and on the outer boundary surface to yield the required number of $n_{\text{el,z}}$ points per subdomain. The resulting boundary points obtained during recursion level $\ell = 0$ are shown in Fig. 3.30b.

This method is also similarly required on higher recursion levels $\ell > 0$. Then, however, two cases have to be considered separately. The first case involves splitting the muscle surface boundary on one process into two parts, analogously to the described method at $\ell = 0$. The second case involves two neighboring processes that have to agree on the split point of their common part of the outer surface boundary.

In the example at recursion level $\ell = 1$ in Fig. 3.28c, the red streamlines are used to split the boundary sides at the outer boundary of the global domain. The first case occurs for the upper left red streamline, which is used to bisect the shown white part of the muscle surface.

The second case occurs at the lower left and upper right borders between the shown subdomain and the neighboring subdomains of two other processes. For the case at the upper right, Fig. 3.31 visualizes the following method: First, the point on the outer surface that is closest to the point of the red streamline is determined on both subdomains, visualized by the yellow stars. These points are communicated between the two processes. Each process computes the center point of these two points (orange star) and then finds the closest point to this center point on the boundary. This is done for all rings of the muscle in z direction. In result, both processes have the same line on the surface in longitudinal direction of the muscle that is then used as one edge of the new subdomains.

Thus, the new subdomain boundaries on the outer boundary can be found using the red extra streamlines shown in Fig. 3.28c. For simplicity, the algorithm always computes the four streamlines in every corner of the mesh although all of them are only required for $\ell = 0$. In the shown example for $\ell = 1$, the streamline in the lower right corner is not needed for the sampling of the new boundary points.

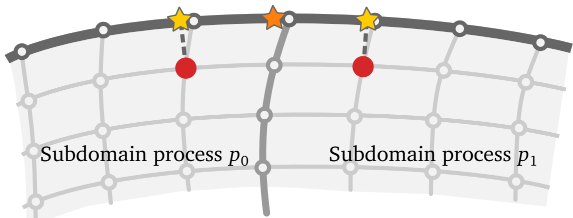


Figure 3.31: Parallel mesh generation: Case of partitioning the outer boundary surface that occurs for recursion level $\ell \geq 1$ in Alg. 2. Shown are the meshes on two subdomains of processes p_0 and p_1 and the location of the streamline at the corner (red points). The orange star is the newly determined border point between the subdomain boundaries.

A summary of the streamlines that are used for the new subdomain boundaries in this example is given in Fig. 3.28d. The sampled boundary points at the muscle surface are shown in light green color. These two sides of the own domain will be split into four sides for the new subdomains. In this example with $n_{\text{el,x}} = 4$, the surface therefore gets sampled at $4 \times 4 = 16$ lines. A comparison with the white lines in Fig. 3.28c shows that the newly sampled points are different from the initially sampled points. While in Fig. 3.28c always five neighboring boundary points are located on a straight line, the points in Fig. 3.28d follow the curved outer boundary better and, thus, refine the boundary representation.

3.5.10 Parallel Algorithm for Streamline Tracing

In line 6 of Alg. 3, streamlines have to be traced through the gradient field $\nabla p(\mathbf{x})$ of the Laplace solution for all the seed points given in Sec. 3.5.8. In the following, more details on the parallel method of streamline tracing is given.

This step is similar to the analog step in Alg. 2. The same method of explicit Euler integration is used. The seed points are located at the horizontal plane at the center in vertical direction of the muscle. From there, streamlines are traced in both directions towards the ends of the muscle following the positive and negative gradient directions. The tracing algorithm uses the efficient scheme of selecting the subsequently traversed elements described in Sec. 3.4.7. The implementation is adjusted in a way to also take into account the layers of ghost elements.

Since the streamlines traverse the entire muscle from the center to the bottom and

top, multiple processes are involved in the computation of every streamline. To describe the scheme, all processes are numbered in z direction from bottom to top by an index $i_z \in \{0, 1, \dots, n_z - 1\}$ where $n_z = 2^\ell$ is the number of processes in z direction on the current recursion level ℓ.

The initial seed points are determined on the processes at the vertical center with index $\lfloor n_z/2 \rfloor$. They are communicated to the processes below with index $\lfloor n_z/2 \rfloor - 1$. These two groups of processes begin with tracing the streamlines through their subdomains starting from the same seed points, the upper processes in upward and the lower processes in downward direction. Then, the end points of the traced streamlines are communicated to the next processes, which continue the tracing. The procedure repeats with further processes until the streamlines reach the bottom and top ends of the overall muscle domain. The time complexity of this approach is $\mathcal{O}(n_z) = \mathcal{O}(\sqrt[3]{n_\text{proc}})$ with the number n_proc of processes.

After the streamlines have been traced, they are sampled at equidistant positions with a distance according to the required distance between the boundary points of the subdomains.

3.5.11 Recursion End: Generation of the Resulting Meshes

In result, one pass of Alg. 3 from lines 2 to 7 creates boundaries for eight new subdomains. Line 8 checks whether the maximum recursion ℓ_max is reached and the recursion ends. If the recursion ends, the final 3D mesh and 1D fiber meshes are constructed in line 9. In this case, the prepared boundary points are not needed for a further subdivision of the domain but are used to construct the final meshes instead.

Every resulting fiber mesh is generated by one streamline. In line 9, additional streamlines are traced starting at the remaining grid points of the 2D slice at the vertical center of the muscle that were not selected as seed points earlier. The parallel method described in Sec. 3.5.10 is used.

As an example, Fig. 3.32 shows all seed points of streamlines for $\ell = 2$ at the beginning of line 9 in a run of Alg. 3 with $n_\text{el,x} = 4$, $\ell_\text{max} = 2$ and 64 processes. The shown points are located at the top of the lowest 16 subdomains in the muscle, i.e., the subdomains of processes 0 to 15. The corresponding streamlines get traced in line 6 to be used for the new subdomain boundary. However, since the recursion ends at $\ell = 2$ the missing seed points of the subdomain grids are subsequently filled in and the remaining streamlines

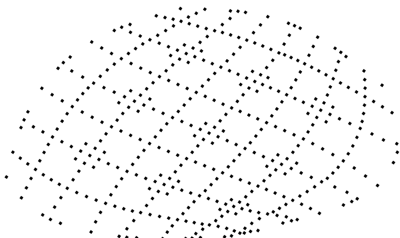


Figure 3.32: Parallel mesh generation: Seed points of the streamlines that are traced on processes 0 to 15 in the procedure of Alg. 3 at recursion level $\ell = 2$.

are traced. From the full grid of $31 \times 31 = 961$ streamlines, 449 or 47 % have already been traced at this point.

In summary, the 2D quadrilateral mesh at the center slice of the muscle defines the location of the resulting muscle fibers. Because the construction of this 2D mesh ensured a good mesh quality with similar element sizes, the distance between the resulting fibers is similar and a spatially homogeneous set of muscle fibers is generated.

To obtain the final 1D fiber meshes $\Omega_{F,i}$, the streamlines are sampled at equidistant z intervals, specified by a parameter Δz. Because the streamlines are directed mainly along the z axis, the constant z interval for the sampling approximately corresponds to the resulting 1D mesh width, i.e., the distance between the points of a fiber. An advantage of this method is that the points of all fibers lie in the same x-y planes. Thus, the total set of points can also be interpreted as a structured 3D mesh of the muscle volume Ω_M. This 3D mesh is aligned with the fiber meshes and planes through the x and y axes. These properties are advantageous for data mapping between the 3D mesh and the 1D fiber meshes and for the numerical solution of models with anisotropic advection processes in the 3D mesh that is oriented according to the direction of the fibers.

At the end, the data is written collectively by all processes into a single file. This is done using the parallel file I/O functionality of MPI. This can be done because the absolute position in the file of every point can be calculated from the index of the point in the structured mesh.

Figure 3.30c gives an example of the resulting streamlines if the recursion ends already after one pass of the procedure at $\ell_{\max} = 0$. For the example with $\ell_{\max} = 1$, the selected seed points and the parts of the resulting streamlines in the considered subdomain are shown in Fig. 3.33a. Here, the dark blue streamlines on the boundary were traced as part

of the refinement actions in line 6. Because the recursion ends for $\ell = 1$, these streamlines are now reused for the final fiber meshes instead of further parallel partitioning. Additionally, the light blue streamlines in the interior were traced to obtain a full grid of fibers for the output of the algorithm.

3.5.12 Continuation on the Next Recursion Level

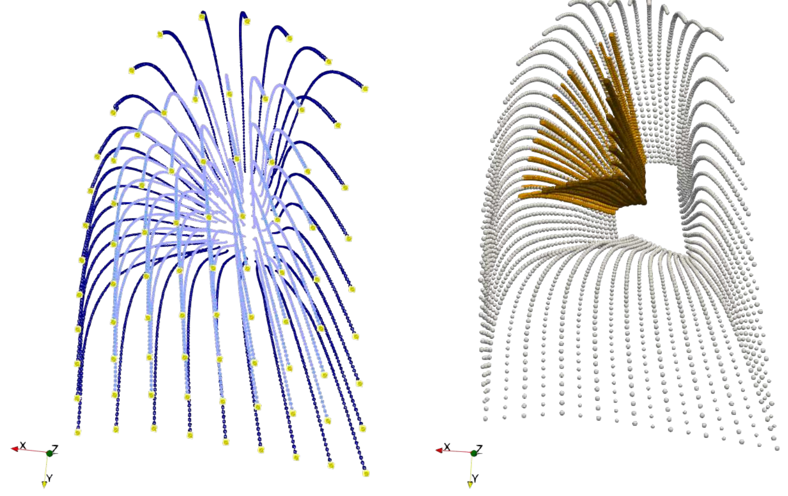
If line 8 of Alg. 3 does not detect the recursion end because the maximum recursion levels is not yet reached, the **`else`** branch in line 10 is chosen. Execution continues with the eight times higher number of processes $8^{\ell+1}$. The processes that executed the previous parts of the algorithm send their determined boundaries of the new subdomains to seven other respective processes in line Sec. 3.5.1. Only the first subdomain remains on the same process. Every process stores the boundary points for its new subdomain in the variable `boundary_points`. In line Sec. 3.5.1, the procedure is called recursively and the next recursion level $(\ell + 1)$ begins.

Figure 3.33b shows the `boundary_points` of the first new subdomain on level $\ell = 2$ for the example on recursion level $\ell = 1$. It consists of the outer boundary (dark yellow lines) and the interior boundary (brown streamlines) and is nearly geometrically similar to the subdomain on level $\ell = 1$.

3.5.13 Repair of Incomplete Streamlines

Practical tests have shown that, for irregular muscle geometries, occasionally some streamlines generated in lines 6 and 9 of Alg. 3 can be incomplete. This means that it was not possible to obtain a streamline that runs through the entire subdomain or the entire muscle domain from top to bottom, instead points are missing for some ranges of z values. This can happen if the streamlines leave the subdomains (because the ghost layer width was chosen too small) or due to numerical errors in irregularly shaped elements mainly on high recursion levels where the system matrix is badly conditioned.

To obtain meaningful results even in these cases, three different repair mechanisms are introduced that interpolate the missing data from valid streamlines. Figure 3.34 visualizes the cases by examples in a setting of four subdomains with grids of 5×5 fibers each. The repair mechanisms #1 to #3a only apply to boundary points. They are executed in line 6 of the algorithm after the local portions of the streamlines have been traced and before the end points of the streamlines are sent to the neighbor processes below and above that



(a) Seed points (yellow), traced interior streamlines (light blue) and boundary points (dark blue), generated if $\ell_{max} = 1$.

(b) Boundary points of the first subdomain on level $\ell = 2$ (dark yellow and brown) embedded in the boundary (white) of level $\ell = 1$, generated if $\ell_{max} > 1$.

Figure 3.33: Generation of 3D and 1D meshes in subdomains: Resulting streamlines after the pass of Alg. 3 for recursion level $\ell = 1$.

continue the streamline tracing. Mechanismn #3*b* and #3*c* repair invalid streamlines in the final result and are executed during line 9 of Alg. 3.

Mechanism #1 checks all streamlines at subdomain boundaries in the interior, which are shared between neighboring processes. If a streamline is incomplete on one process but complete on the neighbor process, the data of the complete side are transferred such that both processes have the same valid points for this streamline. In the example in Fig. 3.34, the valid streamline data are sent from the top left to the top right subdomain.

Mechanism #2 checks streamlines at the outer corners of the subdomains. Incomplete streamlines at these locations are recreated from the given boundary points. Because the set of boundary points is twice as coarse as the required number of sample points at these streamlines, every second point gets interpolated from the top and bottom neighbor points.

Mechanism #3a is concerned with streamlines at interior subdomain boundaries that

could not be fixed by mechanism #1 because the streamlines are incomplete on both sharing processes. In this case, the streamlines are interpolated from the two complete neighboring streamlines that are located next along the boundary as shown in the example in Fig. 3.34. Instead of the factors $\frac{1}{3}$ and $\frac{2}{3}$, the actual relation of distances between the seed points of the streamlines is used. The same interpolation is executed independently on both involved processes. Because the valid streamlines have the same data on both subdomains, the resulting fixed streamlines will also be identical.

Mechanisms #3b and #3c follow the same approach. They are applied to the interior fibers of the final result and can repair any number of incomplete fibers that are located between complete fibers. This case rarely occurs, a cause can be errors in the numerical solution of the Laplace problem. In example #3b in Fig. 3.34, the two invalid streamlines are interpolated from their left and right valid neighbors. In example #3c, no valid right neighbor exists. Instead, the streamlines are interpolated by using valid positions from the upper and lower neighbors.

3.5.14 Post-processing and Output of the Generated Streamlines

After repairing invalid streamlines, the final result of the algorithm is a grid with $(2\,n_{\text{el,x}}\,n_x + 1) \times (2\,n_{\text{el,x}}\,n_x + 1)$ fibers in the x-y plane and a configurable number of points in z direction, where $n_x = 2^{\ell_{\max}}$ is the number of subdomains per coordinate direction on the last recursion level.

If a higher number of fibers is desired than is naturally generated by the parallel algorithm, additional fibers can be created by interpolation in the existing grid of fibers, which is parallel partitioned. The implementation of the presented algorithm in OpenDiHu includes this post-processing functionality as part of the mesh generation program. Alternatively, the step can be applied separately on any binary output file that contains a grid of fibers.

The action of increasing the number of fibers proceeds as follows. The initial grid contains the fibers that were created from the streamlines, called *key fibers*. A specified number m of additional fibers is placed between the key fibers in both x and y coordinate directions. The additional fibers together with the key fibers form a grid of fibers in the muscle cross-sections with an m times finer mesh width. In the grid of key fibers, every portion bounded by 2×2 key fibers contains $(2+m)^2-4$ additional fine fibers. The total number of fibers depending on n_x and m, therefore, is $N = (2\,n_{\text{el,x}}\,n_x\,(1+m)+1)^2$. Due

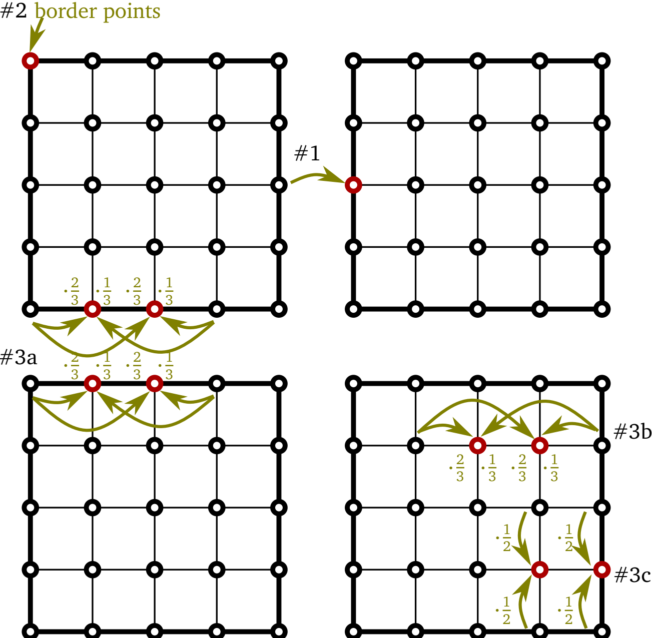


Figure 3.34: Repair of streamlines used during partitioning and fiber approximation in the 3D and 1D mesh generation: Examples of the four repair mechanisms for estimating incomplete streamlines during the parallel algorithm. Invalid streamlines are indicated by red circles, valid streamlines by black circles. The brown arrows show the direction of data transfer.

to construction, this number is always odd. This is a desired property because it yields an even number of elements per coordinate direction and this allows to construct a mesh with quadratic ansatz functions.

The new fibers are computed by barycentric interpolation. The location of every new point $\mathbf{p}$ is calculated from the nearest points $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2$ and $\mathbf{p}_3$ of key fibers in the x-y plane, numbered according to Fig. 3.24, by

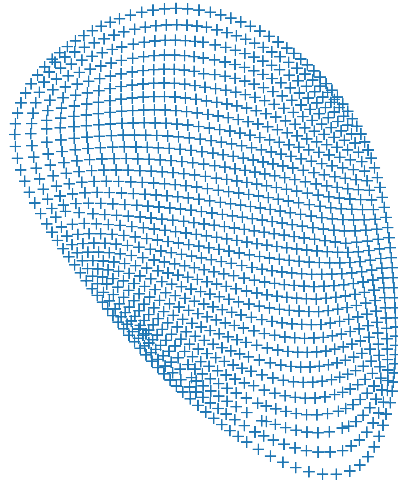
$$\mathbf{p} = (1-\alpha_x)(1-\alpha_y)\,\mathbf{p}_0 + \alpha_x\,(1-\alpha_y)\,\mathbf{p}_1 + (1-\alpha_x)\,\alpha_y\,\mathbf{p}_2 + \alpha_x\,\alpha_y\,\mathbf{p}_3. \tag{3.11}$$

Here, the factors $\alpha_x, \alpha_y \in [0,1]$ are chosen in a way to create the fine grid of fibers:

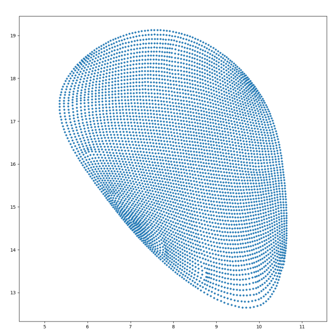
$$\alpha_x = i/(m+1), \quad \alpha_y = j/(m+1) \quad \text{for } i,j = 0,\dots,m, \quad (i,j) \neq (0,0).$$

As a result, we can generate a 3D mesh where the number of points in x and y directions can be adjusted by the parameter m.

An advantage of this algorithm is that each process only has to keep the data of its own subdomain in memory at any time. This allows parallel processing of very large meshes. For small-enough meshes that do not fall under this restriction, the utility script `resample_bin_fibers.py` can be used to create meshes of any resolution from any other mesh using the barycentric interpolation in Eq. (3.11). An example is given in Fig. 3.35, where a mesh of 33×33 fibers is refined by interpolation to a mesh with 71×71 fibers.



(a) Mesh points in a 33×33 grid at the center cross-section of the biceps muscle.



(b) Refined mesh points in a 71×71 grid that were obtained from (a) by barycentric interpolation.

Figure 3.35: Refinement of existing meshes to obtain derived meshes with any number of nodes.

The resulting points are stored in a binary file format. The contents of this output file can either be interpreted as grid points of a 3D mesh or as points of individual 1D fibers. This is an advantage in a multi-scale simulation where both a 3D muscle mesh and multiple embedded 1D fiber meshes occur: First, all mesh information of both Ω_M and $\Omega_{F,i}$ can be given by a single file. And second, the 3D mesh is aligned with the 1D fibers and all 3D mesh points are also 1D mesh points.

The spacing in z direction between points on a fiber is typically chosen as $\Delta z = 0.01\,\text{cm}$. This value was found to ensure a low error in the model for propagation of electric stimuli along the muscle. The value leads to 1481 points per fiber on the belly of the biceps muscle.

Every point coordinate is stored in the output file as double precision value with eight bytes. The file contains a header of 72 bytes with descriptive information such as the number of fibers, some parameter values and a time stamp. The total file size therefore can be calculated by $72 + N \cdot 1481 \cdot 3 \cdot 8$ bytes.

Often, the spatial resolution of the 3D mesh does not need to be as high as those of the fibers. The relation of the 3D and 1D mesh widths as well as the number of 1D meshes should be chosen such that the numerical error of the simulation in both domains is balanced. In case the 3D mesh should be coarser than the output of the algorithm, we can use only a subset of the points contained in the output file. Then, a stride in x, y, and z direction is specified in the settings for the simulation. The corresponding coarse grid of points is extracted and used to construct the 3D mesh that is then used for the simulation.

A remaining issue concerns the mesh quality on the outer boundary. In general, the 3D mesh created by Alg. 3 has good quality because the interior points result from smooth streamlines that were traced through a divergence free vector field. The points at the boundary, however, are either sampled from a triangulation of a tubular surface of the muscle or computed from the NURBS formulation. This surface is derived from imaging data, as described in Sec. 3.3. If the triangulation is used, the quality of the boundary points of the created mesh depends on the quality of the muscle surface and its triangulation. In a case where this quality is poor, only the outer layer of elements of the created 3D mesh is affected. Figure 3.36 shows an example for this effect in a grid of 9×9 fibers. It can be seen that only the fibers at the bottom of the image have an irregularity at their center. Such an irregularity potentially occurs at every z coordinates where a new subdomain begins. The cause is that, at these locations, the points on the rings are slightly shifted relative to each other.

A remedy in such a case is to discard the outer layer of fibers and construct the mesh only from points of the inner streamlines. Accordingly, our implementation of the presented algorithm Alg. 3 always creates two different output files. The first output file contains all fibers, the second contains all except the outer layer of fibers. The second file contains only $N = (2\, n_{\text{el,x}}\, n_x\, (1+m) - 1)^2$ instead of $N = (2\, n_{\text{el,x}}\, n_x\, (1+m) + 1)^2$ fibers.

3.6 Results and Discussion

The following section presents results of the parallel algorithm for mesh generation, Alg. 3. In addition, the effect of various parameters is investigated.

Two types of parameters can be distinguished. Parameters of the first type influences the number of nodes in the resulting mesh. These parameters have to be set such that the desired mesh resolution is achieved. Often, multiple, different parameter combinations

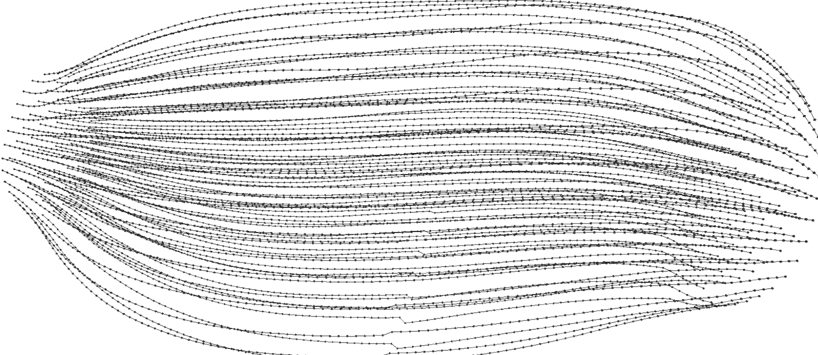


Figure 3.36: Evaluation of the parallel mesh generation algorithm, Alg. 3: Resulting fibers and points on the fibers created with the parallel algorithm, 9×9 fibers with 1481 nodes each. Irregularities in the outer surface can be seen in the center at the bottom of the image.

are possible to achieve a given mesh resolution. Parameters of the second type have no effect on the mesh resolution but on the quality of the mesh. Usually, the parameter combination that gives the highest mesh quality should be chosen.

In the following, Sec. 3.6.1 shows results of the algorithm. Then, Sec. 3.6.2 outlines how parameters of the first type affect the mesh resolution. A specific parameter, the recursion width, is discussed in Sec. 3.6.3. Subsequently, Sec. 3.6.4 evaluates and discusses parameters of the second type, which affect the mesh quality.

3.6.1 Resulting Meshes

At first, results of the whole workflow described in Sec. 3.1 to Sec. 3.5 are presented. The input for the mesh generation algorithm is a geometry representation, which is typically extracted from biomedical imaging. The output of the parallel algorithm, Alg. 3, comprises a 3D mesh with hexahedral elements as well as multiple, embedded 1D fiber meshes.

Figure 3.37 visualizes some results for the biceps and triceps muscles. The parameter values $n_{\text{el,x}} = 4$, $n_{\text{el,z}} = 50$ and $m = 0$ are chosen. If the recursion level is set to $\ell_{\max} = 0$, the algorithm generates meshes with the smallest possible number of fibers, which is a grid of 7×7 fibers. Figure 3.37a shows a grid of 7×7 fibers and the corresponding 3D

mesh that was sampled from the fiber data using every 50th point in z direction of the fiber meshes. It can be seen that the generated fibers traverse all nodes of the generated 3D mesh and, thus, the 3D mesh is aligned with the fiber direction.

Figure 3.37b shows a similar result with 9×9 fibers. Here, the colors correspond to the solution of an electrophysiology simulation. Blue regions indicate that the fiber membranes have an electric potential equal to their resting potential, which indicates no activation. Orange and red colors correspond to activated regions. It can be seen that the activation is present at the same locations on both the fibers and the 3D mesh. In the simulation, this requires data mapping from the fiber meshes to the 3D mesh. Because all nodes of the 3D mesh are located on the fibers, this data transfer becomes trivial.

Figures 3.37c and 3.37d present grids with 13×13 and 67×67 fibers of the biceps muscle, respectively. Results with larger numbers of fibers are not shown here because in such visualizations the fibers become less distinguishable. Figures 3.37e and 3.37f show fibers for the triceps geometry.

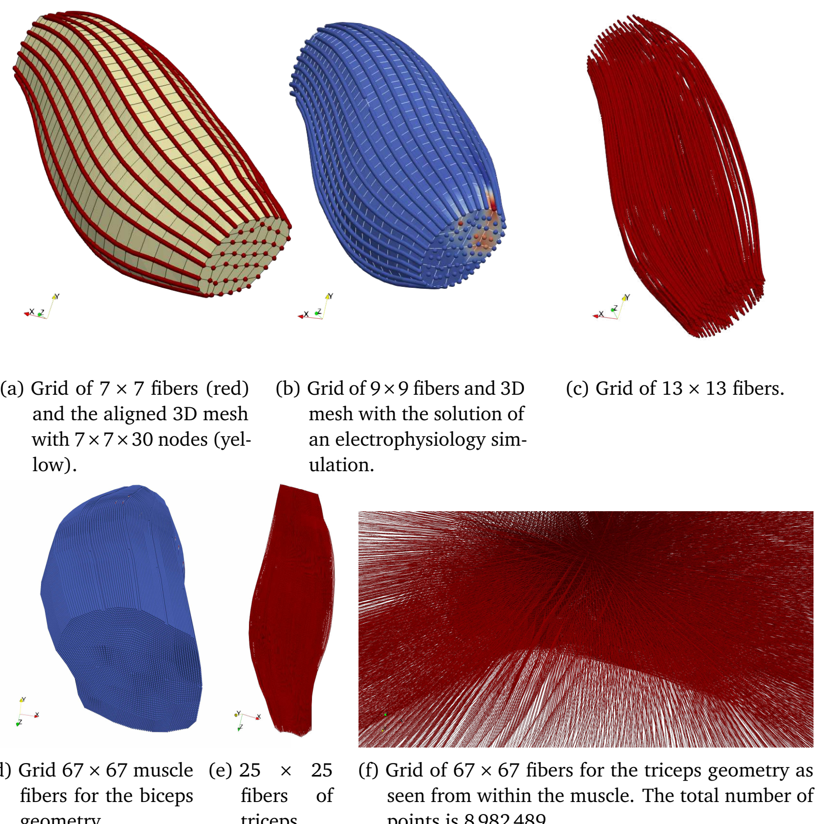
3.6.2 Effect of Mesh Size Parameters

Next, the type of parameters that affect the resulting mesh resolution is discussed. The choices of the maximum recursion level ℓ_{max}, the number $n_{\text{el,x}}$ of elements in x direction of the subdomains and the fine grid parameter m determine the resulting number N of fibers and, thus, the file size of the binary output file. The formulas for these numbers were given in Sec. 3.5.14. Table 3.2 lists exemplary numbers of fibers and file sizes for $n_{\text{el,x}} = 4$ and different values of ℓ_{max} and m. The number n_{proc} of required processes to reach the maximum recursion level is also listed, it depends on ℓ_{max} by $n_{\text{proc}} = 8^{\ell_{\text{max}}}$.

Two different numbers of fibers and corresponding file sizes are listed for every parameter combination. The two variants correspond to the two files that include respectively omit the fibers at the boundary.

The table shows that meshes with different sizes can be constructed by appropriate choices of parameters. A realistic biceps muscle contains about 200 000 to 400 000 muscle fibers [Mac84]. The table shows that constructing a mesh in this range yields a file with a size of ≈10 GiB.[1] A mesh that contains 1 % of the realistic number of fibers can be stored in a file with size of ≈100 MiB.

[1] In this work, file sizes are given using multiples of bytes (B) and the prefixes defined in the ISO/IEC International System of Quantities [80008]. The prefixes are: 1 kibibyte (1 KiB)=2^{10} bytes, 1 mebibyte (1 MiB)=2^{20} bytes, 1 gibibyte (1 GiB)=2^{30} bytes



(a) Grid of 7×7 fibers (red) and the aligned 3D mesh with $7 \times 7 \times 30$ nodes (yellow).

(b) Grid of 9×9 fibers and 3D mesh with the solution of an electrophysiology simulation.

(c) Grid of 13×13 fibers.

(d) Grid 67×67 muscle fibers for the biceps geometry

(e) 25×25 fibers of triceps

(f) Grid of 67×67 fibers for the triceps geometry as seen from within the muscle. The total number of points is 8 982 489.

Figure 3.37: Evaluation of the parallel mesh generation algorithm, Alg. 3: 1D fiber meshes and corresponding 3D meshes. The biceps geometry is used in (a)-(d), the triceps geometry in (e) and (f). The fibers have 1481 nodes each in the biceps muscle and 2001 nodes each in the triceps muscle.

max. level $\ell_{\max}$	fine grid m	# proc. n_{proc}	# fibers	file size
0	0	1	$9 \times 9 = 81$	2.7 MiB
			$7 \times 7 = 49$	1.7 MiB
0/1	1/0	1/8	$17 \times 17 = 289$	9.8 MiB
			$15 \times 15 = 225$	7.6 MiB
0/1/2	3/1/0	1/8/64	$33 \times 33 = 1089$	36.9 MiB
			$31 \times 31 = 961$	32.6 MiB
0	7	1	$65 \times 65 = 4225$	143.2 MiB
			$63 \times 63 = 3969$	134.5 MiB
2	7	64	$257 \times 257 = 66\,049$	2.2 GiB
			$255 \times 255 = 65\,025$	2.2 GiB
2	15	64	$513 \times 513 = 263\,169$	8.7 GiB
			$511 \times 511 = 261\,121$	8.6 GiB

Table 3.2: Parallel 1D and 3D mesh generation: Different parameter choices of $\ell_{\max}$ and m and the resulting number n_{proc} of processes, number of fibers and file size. Some results can be achieved with different parameter combinations, e.g., both $\ell_{\max} = 0, m = 1$ and $\ell_{\max} = 1, m = 0$ result in 17×17 fibers. These combinations are separated by slashes.

The binary files to store the generated meshes are small compared to ASCII-based file formats as each point coordinate is represented by only eight bytes. For comparison, the ASCII-based *exnode* format defined within the OpenCMISS framework uses 24 characters, i.e., 24 bytes to store one point coordinate. Additionally, a larger memory overhead for the description of the data is needed such that *exnode* files are more than three times larger than the binary files used in OpenDiHu.

The binary file format uses no compression that could further reduce the file size. The reason is that no extra effort should be needed when writing programs that parse these files. Thus, they can easily be handled by codes in different programming languages. For example, within OpenDiHu the file format is understood by various Python scripts and C++ programs.

3.6.3 Effect of the Recursion Width

Some numbers of fibers can be achieved with multiple, different parametrizations that use different recursion widths. Table 3.2 contains such alternatives for $\ell_{\max}$ and m separated by slashes in the second and third row. For example, the three combinations ($\ell_{\max} = 0, m = 3$), ($\ell_{\max} = 1, m = 1$), and ($\ell_{\max} = 2, m = 0$) all lead to a grid of 31×31 fibers

(without boundary layer). However, the spatial location of the fibers in the muscle is not identical for these alternatives, because the intermediate mesh used for the streamline tracing of the fibers is differently resolved. In the case with recursion depth $\ell_{\max} = 0$ and fine grid interpolation parameter $m = 3$, numerous of the resulting fibers are interpolated from a coarse grid whereas in the case with $\ell_{\max} = 2$ and $m = 0$ all fibers are key fibers and are obtained by streamlines tracing through a fine mesh.

Figure 3.38 shows parts of the resulting meshes at the longitudinal center of the muscle for these two cases. In Fig. 3.38a, the mesh obtained with $\ell_{\max} = 0$ consists of a grid of traced key fibers and an interpolated finer grid of fibers. The key fiber grid is given by the corners of the gray checkerboard pattern in the image. It can be seen that the mesh consists of patches with 4×5 or 5×5 fibers that each have equal element lengths and angles. In comparison, the mesh in Fig. 3.38b that was obtained with $\ell_{\max} = 2$ consists only of key fibers. Here, the change in shape going from one element to its neighbors occurs more smoothly than in Fig. 3.38a. This qualitatively implies a higher mesh quality.

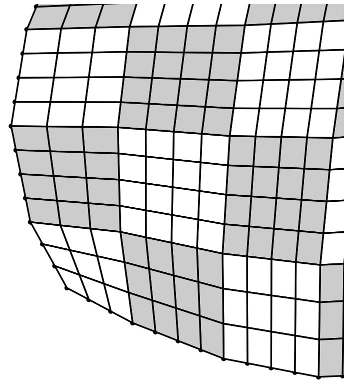
To quantify this effect, we introduce a measure for mesh quality and compare the scores of the three alternatives in the present example. We consider all angles that occur in an element in the x-y plane. The mean value of all angles is obviously $\pi/2$. The variance of all angles can be used as the measure for mesh quality. If the variance is low, this indicates similar elements and, thus, good mesh quality.

For the present example, the variance was computed for the mesh with 31×31 fibers and 1481 nodes per fiber and, thus, 1 332 000 3D elements in total. Figure 3.39 plots the variance for the three cases given in the third row of Tab. 3.2, i.e., parameter combinations $(\ell_{\max} = 0, m = 3)$, $(\ell_{\max} = 1, m = 1)$ and $(\ell_{\max} = 2, m = 0)$. The 3D mesh corresponding to the lowest bar contains the 2D mesh shown in Fig. 3.38a and the 3D mesh corresponding to the upper-most bar contains Fig. 3.38b.

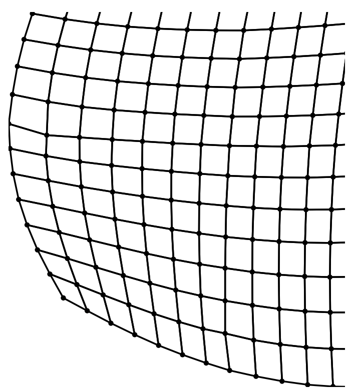
It can be seen that the quality of meshes on higher recursion levels with less interpolation increases as expected. This emphasizes the benefit of the parallel algorithm that uses finer meshes compared to the mesh used during serial execution of the algorithm.

3.6.4 Effect of Mesh Quality Parameters

In addition to the parameters that affect the resulting mesh sizes, $n_{\text{el,x}}$, $\ell_{\max}$ and m, further options exist to tune the behavior of Alg. 3 and in result lead to meshes with different



(a) Result for parameters $\ell_{\max} = 0, m = 3$, i.e., with three interpolated fibers between every two traced fibers.



(b) Result for parameters $\ell_{\max} = 2, m = 0$, i.e., without interpolation.

Figure 3.38: Comparison of generated meshes of the biceps with different maximum recursion levels $\ell_{\max}$. A lower left portion of the full mesh with 31×31 fibers is shown.

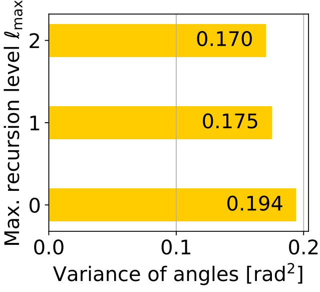


Figure 3.39: Variance of the element angles for meshes with the same number of 31×31 fibers, but created by different recursion levels $\ell_{\max}$. The parameters correspond to the third row of Tab. 3.2. A lower variance means better mesh quality.

quality. These options are described in the following.

The surface that is the input to Sec. 3.5 can be represented either as triangulation or in parametric form as NURBS surface. The triangulation can either be the result of the image segmentation step or it can be obtained by triangulating a NURBS surface. Thus, if the approximation of the geometry by a smooth spline surface is desired it is possible to choose between both options.

One difference is the resulting runtime. To sample a point on the surface using the NURBS representation, the nonlinear equation has to be inverted using a Newton scheme for each point. This is slower than using the triangulation where rings on x-y planes are extracted initially and then equidistantly sampled, as explained in Sec. 3.4.1 and Sec. 3.5.3.

The Laplace problem $\Delta p = 0$ that is solved in every recursion depends on the discretization and mesh resolution on every subdomain. In addition to the number $n_{\text{el,x}}$ of elements in x and y directions, the mesh resolution also follows from the number $n_{\text{el,z}}$ of elements in z direction.

The number of elements in this intermediate mesh is also influenced by the factor $r \in \mathbb{N}$ of the refinement described in Sec. 3.5.6. While $r = 1$ corresponds to no refinement, for $r > 1$ the number of elements is increased by the factor r^3. Note that the output meshes of the algorithm depend on $n_{\text{el,x}}$ but not on $n_{\text{el,z}}$ nor r as they are generated later after the process of streamline tracing.

Furthermore, the finite element discretization of the Laplace problem can either use linear or quadratic ansatz functions, leading to the respective linear or quadratic elements in the mesh. The type of boundary conditions for the Laplace problem in Eq. (3.8) can be selected among the Neumann boundary conditions given by Eq. (3.9) or the Dirichlet boundary conditions given by Eq. (3.10).

After the Laplace problem is solved, the gradient direction $\nabla p(\mathbf{x})$ at a point $\mathbf{x}$ in the domain needed for streamline tracing can be determined by two different methods. Either the gradient vector field is precomputed using finite differences and the nodal values of the solution field p and then evaluated at $\mathbf{x}$. Or the gradient value is directly interpolated at $\mathbf{x}$ in the 3D element using a linear combination of the solution values and derivatives of the ansatz functions of the element.

During parallel streamline tracing, the width $n_{\text{ghost_layer_width}}$ of the ghost layer is important. If it is too small, streamlines leave the domain of the process and have to be repaired, i.e., approximated by neighboring streamlines as described in Sec. 3.5.13. We

found that a value of $n_{\text{ghost_layer_width}} = 2\,r$ is enough and minimizes the number of invalid streamlines leaving the domain. With some parameter combinations, invalid streamlines still occur occasionally. Those result from badly conditioned elements and gradient values with high numerical errors that cannot be fixed by a larger ghost layer. By including the factor r in the ghost layer width, the actual sizes of the ghost layer is always the same independent of the chosen refinement.

To investigate the effect of all options, a parameter study is conducted in the following. We fix the values of $n_{\text{el,x}} = 4$, $n_{\text{el,z}} = 50$, $m = 1$, $\ell_{\text{max}} = 1$, and $n_{\text{ghost_layer_width}} = 2\,r$ and vary all other parameters. The resulting meshes consist of 33×33 fibers and 31×31 fibers if the boundary layer is omitted, as described in Sec. 3.5.14. We compare the mesh quality of the 3D meshes that result from the 31×31 fibers. As before, the variance of the element angles is used to rate the quality of each resulting mesh.

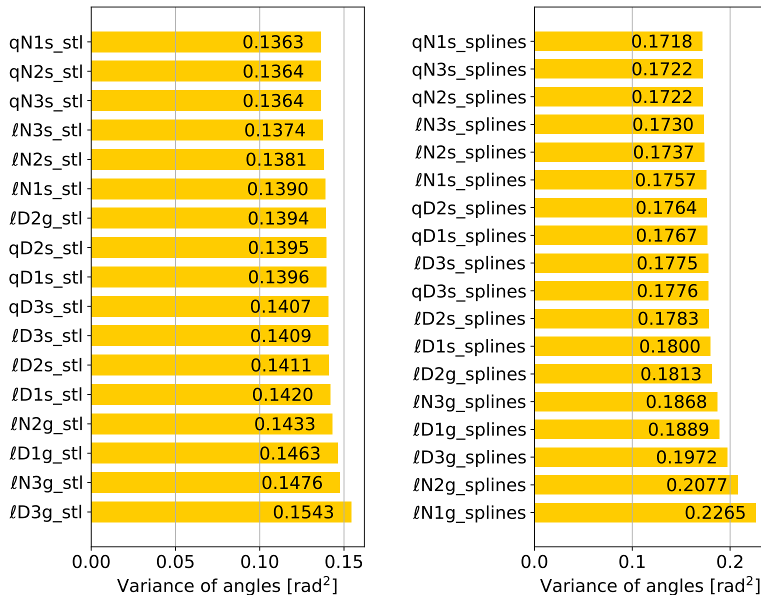
To identify a parameter combination in the study, a scenario name is composed of one character each for the various options, as explained in the following. The linear or quadratic formulation of the Laplace problem is indicated by the characters "ℓ" or "q". Neumann and Dirichlet boundary conditions are indicated by "N" and "D". The refinement level r is specified by the respective integer value. Finally, "g" or "s" indicates whether the precomputed gradient field ("g") is used in streamline tracing or the solution values ("s") and derivatives of the ansatz functions.

For example, the scenario considered in Figures 3.38 and 3.39 can be specified as "ℓD2s", as it uses the linear mesh with Dirichlet boundary conditions, a refinement factor $r = 2$ and the solution values to compute the gradient.

The following study was performed for the biceps geometry in two variants, firstly using the approximated NURBS surface directly and secondly using a triangulation obtained from the NURBS surface. These two variants are indicated by "splines" for the NURBS surface and "stl" for the STL file containing the triangulation.

Figure 3.40 presents the resulting variances of the element angles. The scenarios are sorted according to their mesh quality score, i.e., the variance of their element angles. This means the results are ordered by improving mesh quality from bottom to top.

Two separate plots for the "stl" and "splines" scenarios are shown in Fig. 3.40a and Fig. 3.40b. The two resulting meshes of the best options, "qN1s_stl" and "qN1s_splines" are visualized left and right in Fig. 3.41. In the top plane of the muscle belly, it can be seen that the orientation of the mesh is slightly different. This explains the large difference of the angle variance values between the two scenarios in Fig. 3.40, which are higher for



(a) Scenario using the surface triangulation. (b) Scenario using the spline surface.

Figure 3.40: Parallel mesh generation algorithm: Comparison of the mesh quality that results from different options in the mesh generation algorithm. A lower variance means better mesh quality.

"splines" than for "stl". The scores of parameter combinations should therefore only be compared among the same surface representation. A statement regarding which of the two options is better is not reasonable from this data set.

The results in Fig. 3.40 show that almost all values are close together, which indicates similar good mesh qualities for different parameter combinations. Nevertheless, the ranking reveals some differences between the options. A comparison of the rankings in the columns for "stl" or "splines" shows that some parameter choices consistently resulted in better meshes. A better result was achieved if Neumann boundary conditions were used ("N") compared to Dirichlet boundary conditions ("D"). Similarly, quadratic ansatz functions ("q") performed better than linear ansatz functions ("ℓ"). This is reasonable as quadratic ansatz functions yield a higher spatial consistency in the finite element formulation.

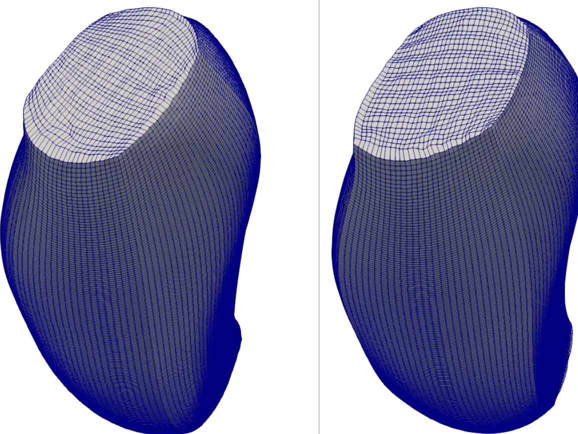


Figure 3.41: Parallel mesh generation algorithm: Results of the scenarios using a surface triangulation (stl, left) and a NURBS surface (splines, right).

A higher refinement factor r of the internal mesh was beneficial for the cases with linear ansatz functions. For quadratic ansatz functions, the effect of r is less clear. The study shows that no refinement ($r = 1$) is often better in this case. The variant without the precomputed gradient field ("s") performed generally better than the variant with gradient field computation ("g").

However, further studies with different recursion depths showed that the effect of some options also depended on the scenario. For a higher recursion depth, Dirichlet boundary conditions turned out to be more robust in the sense that fewer incomplete streamlines occurred.

In summary, the quadratic formulation ("q") and the streamline tracing using solution values ("s") could be shown to be better options than their alternatives. For a maximum recursion level $\ell_{\max} = 1$, the best parameter combination among the tested combination was "qN1s". In a separate study for $\ell_{\max} = 2$, the combination "qD2s" was found to be as robust as "qN1s".

3.6.5 Post-processing of the Meshes

To further improve the mesh quality on every cross-section, we apply two more post-processing steps, one local and one global transformation. As can be seen in the cross-

section of Fig. 3.41, some rows of elements in the generated mesh have zigzag lines, and not all elements are equally sized. Furthermore, there are almost degenerate elements with small interior angles.

Thus, we apply a first, local transformation on the mesh. This operation consists of Laplacian smoothing and randomly deflecting points, where interior element angles are smaller than 20°. If such deflections result in invalid self-intersecting elements, the self-intersection is resolved, which potentially again introduces small interior angles. The total transformation consists of 25 iterations of alternatingly applying the smoothing step and the improvement step of small interior angles.

An exemplary resulting mesh after this transformation is shown in Fig. 3.43a. It can be seen that all lines in the mesh are smooth and almost straight. At the right center of the shown mesh, the effects of the deflection step, which improves small interior angles, can be seen. However, at the left, top and bottom boundary of the mesh, degenerate elements with small interior angles remain. This is especially true for the four mesh points on the boundary that correspond to the corners of the quadrangulation. At these points, elements are present that have two sides that are part of the mesh boundary, forming an interior angle of almost 180°. Three points of these elements are located in an almost straight line. As a consequence, the Laplacian smoothing step moves the fourth point of these elements close to this straight line, which adds another large interior angle and degenerates the element. This effect also occurs for interior elements of the mesh that are close to these points on the boundary.

Figure 3.43b shows the detail of the left boundary of the mesh in Fig. 3.43a, where this effect can be seen. The area of the elements decreases towards the boundary and the elements get more degenerate in this direction.

As a remedy, we perform the second, global transformation step to counteract this tendency. In this step, most of the points in every cross-section of the mesh are translated by a fixed mapping, such that the small elements at the boundary are transformed into elements of better quality.

Each mesh point on a cross-section of the mesh is represented by polar coordinates (r, φ). The radius r is transformed according to the function $r_{\text{new}} = y(r_{\text{old}})$ that is depicted in the lower plot of Fig. 3.42. This piecewise defined function $f \in C^1([0,1] \to [0,1])$ is linear for $x \in [0,s]$ and a polynomial of degree 3 for $x \in [s,1]$. It passes through the yellow point in Fig. 3.42, which in x direction is at the center of the interval $[s,1]$ and in y direction is at fraction α of the shown yellow line. The parameter α controls the extent, to which the mesh is transformed in radial direction. The parameter s specifies

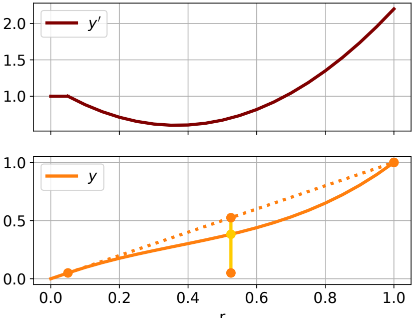


Figure 3.42: Transformation of a mesh to improve the mesh quality. The depicted function f transforms the elements in radial direction.

the size of a region around the center of the mesh where no transformation occurs. We obtain good results by choosing $s = 0.05$ and $\alpha = 0.7$. The resulting function takes the form $f(r) = 1.330\,r^3 - 1.463\,r^2 + 1.136\,r - 0.003$.

The first derivative $f'(x)$ of this function is shown in the upper plot of Fig. 3.42. It quantifies the amount of extension or compression of the mesh elements in radial direction. The right side of the plot corresponds to the outer boundary of the mesh. There, the elements are extended, since $f'(r) > 0$. To compensate this extension, the elements have to be compressed towards the interior of the mesh where $f'(r) < 0$. The range of $r \in [0, s]$ corresponds to the region in the interior of the mesh that is not transformed.

The points of the mesh are transformed in radial direction by adjusting their coordinate r and not transformed in circumferential direction, i.e., the angle φ remains constant. However, the application of the function f on r is additionally modulated by a piecewise sine function depending on φ. The transformation f is only fully applied at the four radii corresponding to the described special points on the boundary, around which the degenerated elements occur. In between these lines, the transformation is reduced and some points of the mesh are not transformed at all.

Figure 3.43b shows the mesh of Fig. 3.43a after this transformation has been applied. Figure 3.43d shows the extract of the mesh from the left boundary that corresponds to the same extract of the original mesh in Fig. 3.43c. It can be seen that the quality of the elements is improved close to the boundary. The area of the rectangles is now approximately equal and no small interior angles occur.

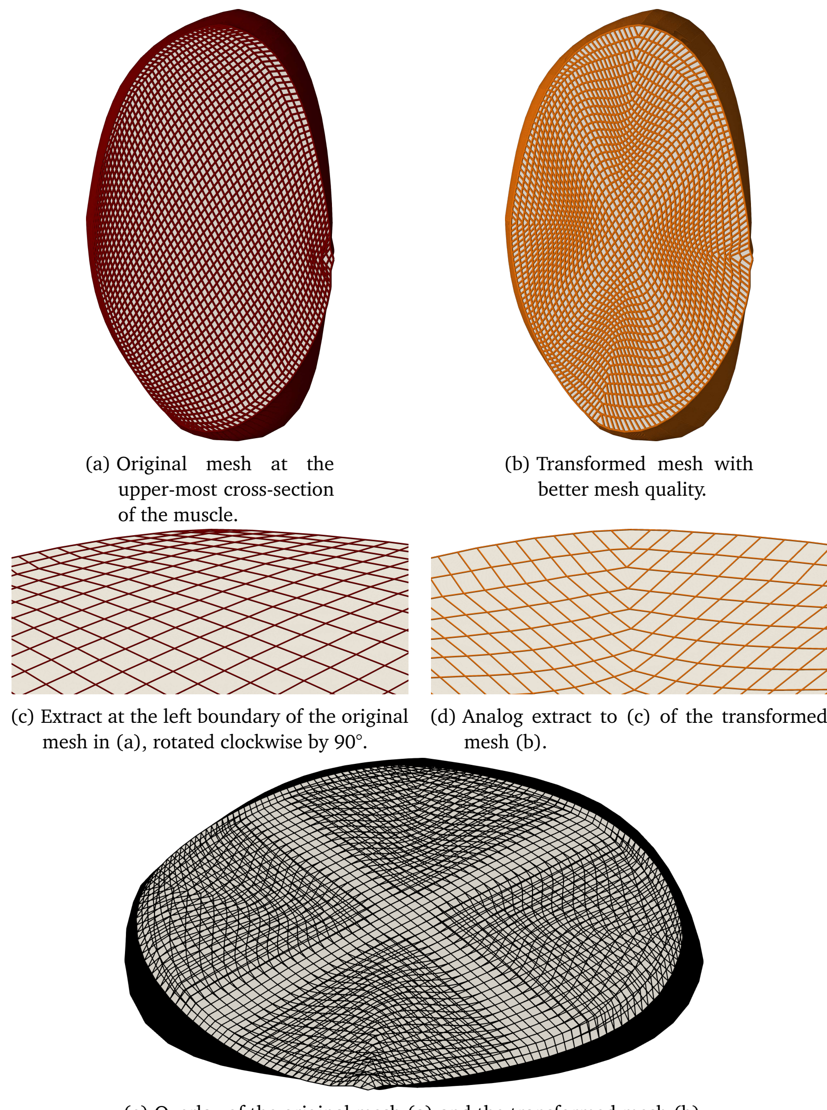
In Fig. 3.43e, the previous and the transformed mesh are overlaid to show the regions that remain unmodified. The unmodified parts form a “cross” shape that touches the boundary at the regions where the mesh quality is also good in the original mesh.

3.6.6 Usage of the Generated Meshes in Simulations

In some biomechanical simulations, also a body fat and skin layer on top of the muscle is considered. In this case, an appropriate mesh is required that is attached to the muscle mesh and seamlessly matches the elements of the muscle mesh. The construction of such a mesh is discussed later in Sec. 7.5.1 together with the parallel partitioning.

A visualization of such a parallel setting is given in Fig. 3.44. Tendons connect the muscle belly of the biceps brachii muscle to the humerus and ulna bones at the top left and bottom, respectively. The muscle mesh consists of fibers that are colored according to a parallel partitioning. In a parallel partitioning, every process contributes calculations only on its associated spatial subdomain to the overall computation. On the right-hand side of the muscle, a layer of adipose tissue is attached to the muscle belly. This layer is needed, if electromyography on the skin surface is simulated.

Figure 3.45 shows another use case of various meshes in a multi-scale simulation model. The muscle is cut open for visualization purposes. The figure depicts numerous muscle fibers in the upper part of the muscle belly. The fibers are colored according to simulation results of the electric membrane potential, which is responsible for the activation of the muscle. The lower part of the muscle shows elements of the 3D mesh. The coloring corresponds to the electric potential, that is measured during intramuscular electromyography in the interior of the muscle or during surface electrophysiology on the outside of the muscle.



(a) Original mesh at the upper-most cross-section of the muscle.

(b) Transformed mesh with better mesh quality.

(c) Extract at the left boundary of the original mesh in (a), rotated clockwise by 90°.

(d) Analog extract to (c) of the transformed mesh (b).

(e) Overlay of the original mesh (a) and the transformed mesh (b), rotated clockwise by 90°. The unmodified regions can be seen.

Figure 3.43: Transformation of a biceps mesh with 47×47 points to improve the mesh quality.

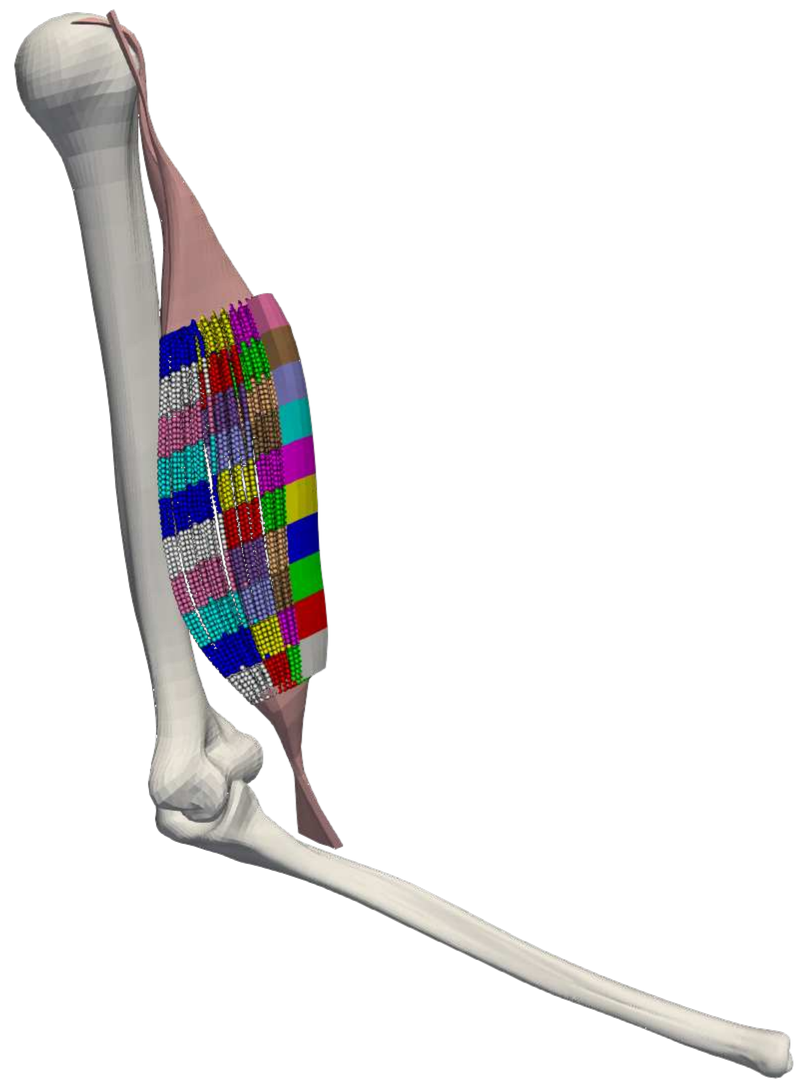


Figure 3.44: Summary visualization of the simulation setup in this work (I): Biceps muscle with tendons and bones, parallel partitioned fibers and a fat layer mesh.

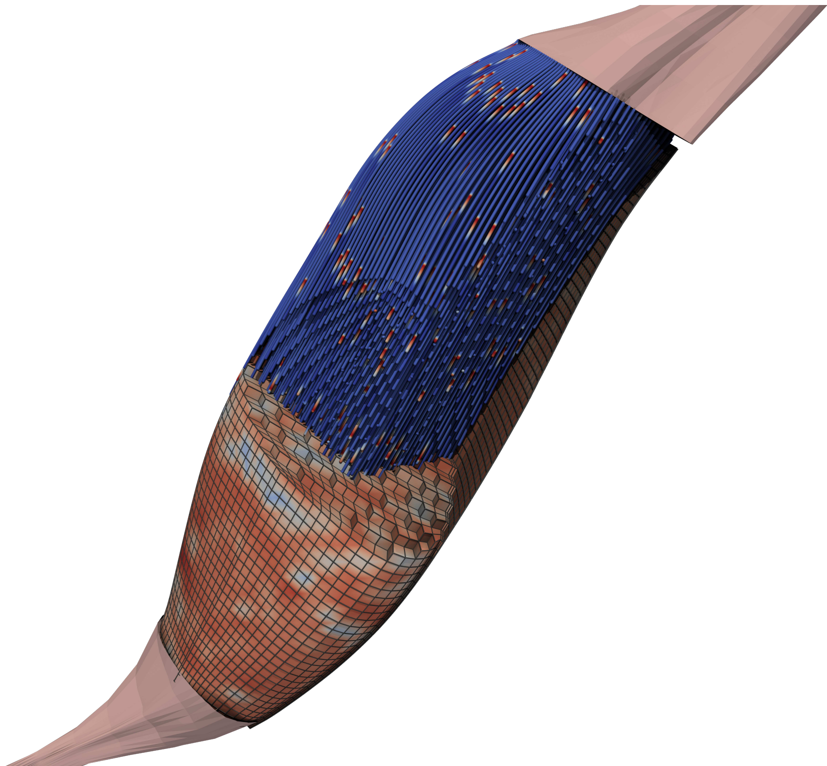


Figure 3.45: Summary visualization of the simulation setup in this work (II): Muscle fibers in the upper part and 3D mesh elements in the lower part of the muscle belly.

How To Reproduce

The parallel algorithm is implemented in the example `parallel_fiber_estimation`. Numerous parameters can be set on the command line. After compilation, run the program as follows to get a description of all available options.

```
cd $OPENDIHU_HOME/examples/fiber_tracing/parallel_fiber_estimation/
  ↪ build_release
./generate ../settings_generate.py --help
```

Running the program without options and `--help` uses sensible default values. A given surface triangulation of the biceps muscle gets used by default. To compute the examples shown in this section, use and adjust the following script that runs the `generate` program and computes the mesh quality:

```
cd $OPENDIHU_HOME/examples/fiber_tracing/parallel_fiber_estimation/
  ↪ build_release
../run.sh
```

Computation of mesh and file sizes as shown in Tab. 3.2 can be done using the *compute_sizes.py* script.

While the previously given commands are good for exploring the algorithms, generation of the meshes used for the simulation involves some more steps. Dedicated scripts exist that perform all steps and call the algorithms with the proper parameters. Starting from the STL file extracted from cmgui, as explained in Sec. 3.3.3, the next steps are:

(i) Scale the points from millimeters (used in the Visible Human dataset) to centimeters (used in the simulation),

(ii) remove the interior triangles,

(iii) translate the mesh such that the bounding box begins at $z = 0$, this is needed for the programs used in the next steps,

(iv) create the spline surface representation as explained in Sec. 3.3.4,

(v) compile and run the OpenDiHu programs to create the binary files of the 3D mesh and the 1D fibers meshes, the algorithm in Sec. 3.5 is used,

(vi) adjust the indexing and undo the translation in (iii),

(vii) refine the created meshes of key fibers by different numbers m of fine grid fibers, in total 10 different mesh sizes are created for differently refined simulations,

(viii) create meshes for the fat layer Ω_B on top of the muscle surface, also in 10 different resolutions.

Two scripts are given for the biceps brachii and triceps brachii muscles. They perform all listed steps and also create intermediate output files that can be used to understand the process. Some steps are automatically skipped if the resulting output file already exists from a previous run. This is especially helpful for the removal of the interior triangles from the initial file which takes nearly a full day.

A third scripts creates three meshes $\Omega_{T,i}$ for the tendons of the biceps muscle, as visualized in Fig. 3.21. At the bottom, a single tendon mesh is created whereas at the top, two separate tendons exist. The script involves numerous rotation and cropping operations of the initial surface, before the algorithm of Sec. 3.4 is executed. The three output files of the tendon meshes have the same file format as the muscle meshes. The files have the extension *.bin* for "binary". The script `examine_bin_fibers.py` can be used to debug the created binary files.

The three scripts can be executed as follows:

```
cd $OPENDIHU_HOME/examples/electrophysiology/meshes
./process_meshes_biceps.sh
./process_meshes_triceps.sh
./process_meshes_tendons.sh
```

The output can be found in the subdirectory `processed_meshes`. For the total output about 68 GiB of drive space is required, however, the resulting meshes have a size of only 19 GiB. A total runtime of more than a day is to be expected.

3.7 Conclusion and Future Work

This chapter presented algorithms for creating muscle meshes that are needed for multi-scale simulations of the musculoskeletal system. For the biceps muscle, 3D meshes for tendons on both ends and the muscle were created. Additionally, 1D fibers meshes were generated that are embedded in the mesh of the muscle. The 3D mesh and the 1D meshes resulting from the parallel algorithm are aligned with each other. This facilitates data

mapping between the meshes and reduces numerical errors. All generated meshes are structured, which allows an efficient parallelization.

First, an overview of available meshing software and known algorithms in the literature was given. Very little software tools were capable of generating structured meshes and none fitted our special needs. Therefore, own algorithms were developed to generate meshes starting from medical imaging data.

A workflow was presented to generate a smooth surface triangulation from imaging data. Our base data was the male dataset from the Visible Human Project. Two alternatives within this workflow were presented, where the first alternative executed automatic image segmentation based on morphological operations and the second alternative used semi-automatic segmentation tools from the Physiome project. Then, smooth NURBS surfaces were fitted to the extracted boundaries of the muscle volumes.

Next, a novel algorithm to create structured meshes from a triangulated muscle surface was presented. The algorithm used harmonic maps on 2D slices in combination with regular grids in a parameter space to achieve good mesh quality. A method of computing streamlines in a divergence-free vector field to estimate muscle fibers, which is established in the literature, was used. It allowed embedding 1D meshes for muscle fibers in the created 3D meshes of the muscle. Numerical experiments tested and evaluated different choices of triangulation and quadrangulation schemes for the 2D cross-section and reference domains in our algorithm.

Next, a parallelized algorithm was introduced that was based on our first, serial algorithm. The algorithm used distributed memory parallelism and provided the same features as the serial algorithm, having the same formats for input and output. The difference was that it constructed a fine, partitioned mesh for streamline tracing that was distributed over all employed processes. Thus, it was possible to create finer meshes using more compute nodes. Differently resolved meshes of the biceps and triceps muscle volumes and muscle fibers were created using this algorithm. The superiority of the parallel algorithm using a higher number of processes compared to the serial execution was explained and demonstrated in a numerical experiment. Several options to fine-tune the algorithm were evaluated. Post-processing methods were described that improved the mesh quality of the resulting meshes.

The presented algorithms and their implementation in OpenDiHu are the basis for further computations within this work. They are used to generating structured hexahedral meshes with good mesh quality. These meshes are required for efficient, parallel finite element simulations of various aspects of the neuromuscular system.

The presented algorithms are specialized for fusiform muscles and require the muscle geometry to be oriented along one coordinate axis (the z axis) in order to generate a structured mesh that comprises planar slices that are normal to that direction. The algorithms can also be applied to any tubular surface geometry of more complex muscles and will construct the corresponding structured 3D mesh. The generated 1D fiber meshes, however, are only valid for muscles, where the approach of streamline tracing through the solution of the Laplacian potential flow problem with boundary conditions at the bottom and top ends of the muscle can be applied. In literature, this approach has been successfully used for various muscles with more complex fiber architectures, such as the tibialis anterior, gluteus maximus and deltoid muscles [Cho13]. However, the locations where boundary conditions were prescribed was not always at the bottom and top ends of the muscle.

If in future work muscles with more complex layouts should be simulated, the approach could be as follows. Depending on the complexity of the outer geometry, first the presented algorithms (either Algorithms 1 and 2 or Alg. 3) can be used to create a structured 3D mesh. Then, a potential flow simulation can be manually setup in OpenDiHu using the 3D mesh and boundary conditions defined at proper locations. Seed points have to be defined and the streamline tracer of OpenDiHu can be used to create fiber meshes. In consequence, the resulting 1D fibers will not be aligned with the 3D mesh. Algorithmically, this poses no problem to the simulations in OpenDiHu as the data mapping functionality can handle arbitrarily positioned meshes. However, the parallel partitioning gets more involved as the combined domain of 1D and 3D meshes has to be partitioned equally for both mesh types.

Chapter 4

Muscle Fibers and Motor Units

The activation of muscle fibers is governed by the functional organization of the fibers in motor units (MU). An MU is the set of fibers that are innervated by the same α-motor neuron, together with the neuron. If a motor neuron fires, all muscle fibers within the MU are activated. The association of the muscle fibers with MUs needs to be specified for electrophysiology simulations that consider activated muscle fibers. This chapter describes algorithms to achieve an MU-fiber association based on biophysical principles.

4.1 Introduction

Given a number of muscle fibers, the goal is to assign each fiber to one MU out of a set of given MUs. A muscle with a fusiform geometry, such as the biceps brachii is considered. Because muscle fibers do not branch or interrupt within the belly of such a muscle, the task can be reduced to the 2D problem on a cross-section of the muscle.

Properties of MUs have been subject to various investigations in literature. The number of MUs in a human muscle can be estimated by anatomical and physiological methods [Mac06]. Anatomical methods include counting large-diameter fibers in postmortem tissue. The morphological studies of [Fei55] revealed high variations between different muscles. For example, the brachioradialis muscle has an estimated number of 333 MUs with 410 muscle fibers on average whereas the external rectus muscles in the eye have 2970 MUs with an average of only 9 muscle fibers.

Physiological methods involve comparing the electrical and mechanical responses of artificially activated muscles, e.g., as in [Mil73; Tho90]. Typically, a high number of MUs with a smaller force or electric response is observed and a smaller number of MUs with a higher response. The review of [Eno01] collects available experimental results and

concludes an exponential distribution of the number of fibers per MU over all the MUs in a muscle.

The spatial arrangement of the fibers of an MU can be revealed by a histochemical method [Bra69]. It was found that the fibers of an MU appear at random positions but are grouped in a subregion of the muscular cross-section. The size of the subregion varies among muscles and fiber types and can be as large as one quarter of the cross-section, as in the tibialis anterior of the rat [Eds68]. Although the fibers of an MU are located in proximity they usually do not touch each other, i.e., there are always fibers of other MUs in between the fibers associated with an MU.

In our algorithm for assigning fibers to MUs we incorporate the following properties that are founded on biophysical experiments.

(a) The number of fibers per MU is exponentially distributed.

(b) The fibers of an MU are spatially distributed around a center point of the MU territory.

(c) The MU center points are reasonably separated from each other. However, the MU territories intermingle.

(d) The spatial extents of the MU territories are proportional to the number of fibers of the MUs.

(e) The exact locations of the fibers are random, but the overall density of fibers in the muscle is approximately constant.

(f) Neighboring fibers are not innervated by the same motor neuron and therefore belong to different MUs.

Further physiological properties of fibers such as their fast or slow-twitch type as well as the distribution of electrical and mechanical properties are not subject to the fiber assignment algorithm. They are considered during configuration of the simulations of electrophysiology or muscular contraction.

4.1.1 Related Works

Simulations involving individually resolved muscle fibers are scarce in the literature. Therefore, not much previous work exists regarding methods to algorithmically assign fibers to MUs. The chemo-electro-mechanical skeletal muscle framework of [Hei13] uses

a method introduced in [Röh12] where center points of MU territories are positioned normally distributed around two distinct weighting centers for fast- and slow-twitch fibers. The algorithm randomly selects from certain sets of fibers and assigns them to MUs with exponentially increasing MU sizes. The method is applied to determine up to 50 MUs in the tibialis anterior (TA) muscle.

This algorithm fulfills the previously formulated properties (a)-(e). Among those, the fulfillment of (c) is not guaranteed but may be given by the random nature of the algorithm. An assumed issue regarding property (b) is that no predictions can be made about the fiber locations of the larger MUs. The larger MUs get assigned to previously unassigned fibers in a late stage of the algorithm, after most of the fibers have been selected for smaller MUs. The largest MU simply gets associated with all remaining fibers that were not selected for other MUs. In the worst case, these fibers can accumulate at multiple different regions, e.g., at boundaries of the muscle which is not physiological.

Instead of the 3D setting in [Röh12] that was needed for the complex anatomy of the TA muscle, we restrict our problem to a 2D cross-section of a fusiform muscle such as the biceps brachii. In comparison, our method creates MU territories of equal quality for all MUs and additionally fulfills property (e). Slow- and fast-twitch fibers are not treated differently by our algorithm, their properties are considered later in the simulation settings.

3D Simulations of skeletal muscle exist that treat MU association as homogenized property in the muscle volume. The approach in [Sai18] assigns volume fractions of MUs to every spatial point of a 3D FEM discretization grid. MU center points are selected randomly ensuring a minimal distance. Prescribed volume fractions are sequentially assigned for each MU. The degree of intermingling can be adjusted by a parameter. It is shown that the algorithm highly depends on the order in which the MUs are traversed. The algorithm fulfills the properties (b)-(e), fulfilling (a) is possible by using appropriate parameter values.

In comparison, our method is targeted at MU assignment to individual fibers. However, distributing volume fractions is also the first step in our method. The volume fractions are interpreted as probabilities of the fibers being assigned to the respective MU. Therefore, our method can also be used as generator for homogenized formulations. A difference is that our method does not depend on a traversing order of MUs and ensures the exponential distribution of MU sizes.

4.1.2 Two Alternative Premises for Motor Unit Assignment

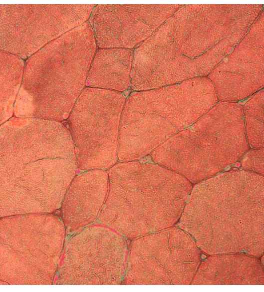
We identify two different sets of requirements which lead to two different methods. Both methods fulfill the properties (a)-(f) listed in Sec. 4.1. The first premise is to assign motor units to a given number of fibers such that every fiber is associated with one MU. The second, alternative premise is to assign motor units only to a portion of the given set of fibers, discarding unassigned ones and thereby reducing the resulting set of fibers.

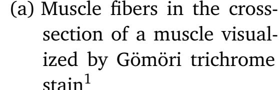
With the first setup, simulations of muscles that are gradually activated by motor neurons are possible. Activating the full muscle corresponds to activating all MUs and, in consequence, all muscle fibers. In reality, it is hardly possible to voluntarily activate all fibers in a muscle. This approach is also chosen in the presented literature, [Röh12] and [Sai18].

The second setup corresponds to modeling only a part of the muscle. The discarded fibers can be seen as belonging to other MUs that are not part of the simulation. When running highly parallelized simulations containing a large number of fibers, the missing fibers can introduce load imbalances, if they are computationally treated equally to the fibers with MUs. Even if no extra computational time is spent for the discarded fibers, a parallel domain decomposition becomes more involved than with the first setup where all fibers in a grid are present.

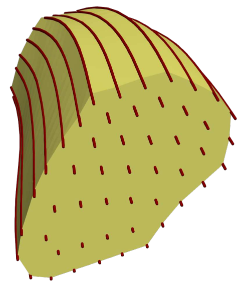
An advantage of the second setup is that the MU assignment to the fibers is generally easier. It also has its analog in volume fraction methods, where scalar fields of factors $f_k : \Omega \subset \mathbb{R}^3 \to [0,1]$ representing multiple MU territories, $k = 1, \dots, n_{\text{MU}}$, can be easily defined. With this setup it is possible, for example, to perform analogous EMG simulations with the Multidomain model of [Klo20] and the fiber based model of [Mor15].

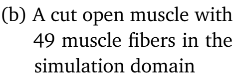
In the remainder of this chapter, Sec. 4.2 presents method 1, which fulfills the first premise where all fibers are associated to the MUs. Next, Sec. 4.3 introduces method 2, which only associates some fibers to MUs. Methods 1 and 2 fulfill the properties (a)-(e). Two derived methods 1a and 2a are subsequently constructed in order to also fulfill property (f). They are presented in Sec. 4.4. Results and a discussion is given in Sec. 4.5 before the chapter ends with a conclusion in Sec. 4.6.



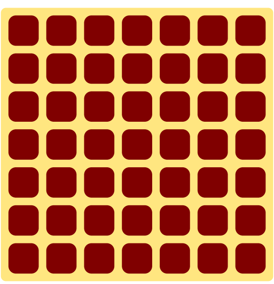


(a) Muscle fibers in the cross-section of a muscle visualized by Gömöri trichrome stain[1]





(b) A cut open muscle with 49 muscle fibers in the simulation domain



(c) A quadratic grid of 7×7 fibers, used for the methods to assign MUs to fibers.

Figure 4.1: Representation of muscle fibers for the methods to associate fibers with MUs: From the real muscle to a quadratic grid.

4.2 Method 1: Assignment of Motor Units to a Given Set of Fibers

In our methods to assign MUs to muscle fibers, the considered set of muscle fibers is organized in a regular grid. Figure 4.1a shows a part of the cross-section of a skeletal muscle, the domains of individual fibers are visible. Figure 4.1b visualizes the representation of muscle fibers in our simulations. In this figure, a relatively low number of 49 fibers was modeled. The fibers are approximated by 1D lines with uniform spacing in radial direction. For the algorithms to assign MUs to fibers, the muscle cross-section is considered as a logical 2D grid with a quadratic number of $n \times n$ fibers. Such a grid is visualized in Fig. 4.1c.

The first method for the assignment of MUs to a given set of fibers associates the $n \times n$ fibers to a set of n_{MU} motor units. First, for each fiber (i, j) in the grid, the probabilities $p(i, j, k_{\text{MU}})$ to be assigned to MU k_{MU} are computed. This computation involves the solution of an optimization problem. Second, the sampling step assigns the actual MU indices to the fibers.

[1]Image copyright © 25/12/2010 Michael Bonert (`https://commons.wikimedia.org/wiki/User:Nephron`), CC BY-SA 3.0 (`https://creativecommons.org/licenses/by-sa/3.0/legalcode`). The picture shows mitochondrial myopathy, it was cropped and the color was adjusted to make the "ragged red fibers" less prominent.

4.2.1 Stochastic Formulation of Motor Unit Assignment

In order to fulfill the formulated properties, the following three conditions are enforced on the probabilities.

(i) The probabilities at every fiber have to be valid, i.e., positive and sum up to 1 for all MUs.

(ii) The portions fibers associated to MUs have to approximately follow an exponential progression q, with MU 1 containing the least and MU n_{MU} containing the most fibers.

(iii) For any given MU, the spatial arrangement of its fibers in the cross-sectional plane is described by a radial kernel function $\hat{p}$. The fiber density of the MU increases when moving closer to the center point of the MU. This condition approximates the fact that the fibers of an MU are located in proximity, forming the MU territory.

The exponential progression q in condition (ii) is defined as follows,

$$q(k_{\text{MU}}) = b^{k_{\text{MU}}} / \sum_{\ell=1}^{n_{\text{MU}}} b^{\ell}. \tag{4.1}$$

The basis b is a parameter and should be set to a value greater than one, e.g., $b = 1.2$. [Eno01] formulate the function to be proportional to $\exp(\log(R)/n_{\text{MU}} \cdot k_{\text{MU}})$ where R is the constant ratio between the sizes of the largest and smallest MUs. Our form is equivalent with $b = R^{1/n_{\text{MU}}}$.

The value of q in Eq. (4.1) is always positive. The division by the scaling factor ensures that the probabilities for all MUs sum up to one. Thus, condition (i) is fulfilled. The construction with the exponential function fulfills condition (ii).

For condition (iii), center positions $\mathbf{x}_{k_{\text{MU}}}, k_{\text{MU}} = 1, \dots, n_{\text{MU}}$ of the MU territories are defined. The center positions are quasi-randomly selected inside the inner 80% of the $n \times n$ grid of fibers. A band at the boundary with width of 10% is not considered because the MU center points should not be at the border of any MU territory but rather at their center.

The used quasi-random sequence is the following Weyl low-discrepancy sequence

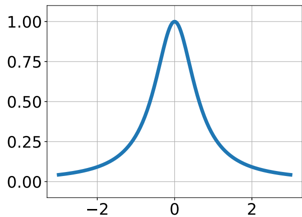


Figure 4.2: Function $1/(1+a\,|x|^2)$, similar to $\hat{p}$ of Eq. (4.3), for $\sigma = 1$.

[Wey16]:

$$\begin{aligned} x_0 &= 0.5, & y_0 &= 0.5, \\ x_i &= x_0 + (i \cdot \alpha_1) \mod 1.0, & y_i &= y_0 + (i \cdot \alpha_2) \mod 1.0, \\ \text{with } \alpha_1 &= 0.5545497, & \alpha_2 &= 0.308517. \end{aligned} \tag{4.2}$$

It is known that the sequences x_i and y_i are equidistributed in $[0,1)$ for any irrational α_1 and α_2 [Wey16]. The chosen values lead to a sequence of 2D points $(x,y) \in [0,1)^2$ with low discrepancy and a good coverage of the domain for any number of sequence elements. Accordingly, the MU territory center points are defined as

$$\mathbf{x}_{k_{\text{MU}}} = \big((0.1 + 0.8\,x_{k_{\text{MU}}})\,n,\, (0.1 + 0.8\,y_{k_{\text{MU}}})\,n\big)^\top$$

The radial kernel function $\hat{p}$ that describes the spatial probability distribution for a given MU k_{MU} according to condition (iii) is defined as follows,

$$\hat{p}(i,j,k_{\text{MU}}) = \frac{1}{1 + a\,|\mathbf{x}_{k_{\text{MU}}} - \mathbf{x}_{i,j}|^2}, \quad \text{with } a = \frac{\pi^2}{4\sigma^4}. \tag{4.3}$$

The coordinates i and j specify the grid point $\mathbf{x}_{i,j} = (i,j)^\top$ of the fiber. The factor a is computed from the given standard deviation σ of the spatial distribution of the MU territory around the center point. A lower value of σ leads to smaller and “sharper” MU territories, for higher values of σ, the MU territories intermingle more with each other. Figure 4.2 shows the graph of the function for $\sigma = 1$.

This kernel function was chosen because it can be computed efficiently with a low number of basic operations unlike, e.g., a Gaussian kernel function which requires costly evaluation of an exponential function.

If the kernel function $\hat{p}$ in Eq. (4.3) is used to describe the probability of fiber (i, j) to be in MU k_{MU}, then condition (iii) is fulfilled but conditions (i) and (ii) will not automatically be fulfilled. Instead of $\hat{p}$, a derived term $p(i, j, k_{\text{MU}})$ is introduced in the following that satisfies all requirements.

To ensure condition (ii), additional scalar factors $\lambda_k, k = 1 \dots n_{\text{MU}}$ for the MUs are introduced that yield the required exponential distribution. To ensure condition (i), the term is normalized by a respective division. The resulting formulation is given as follows:

$$p(i, j, k_{\text{MU}}; \{\lambda_k\}_{1 \dots n_{\text{MU}}}) = \frac{\hat{p}(i, j, k_{\text{MU}}) \cdot \lambda_{k_{\text{MU}}}}{\sum\limits_{\ell_{\text{MU}}=1}^{n_{\text{MU}}} \hat{p}(i, j, \ell_{\text{MU}}) \cdot \lambda_{\ell_{\text{MU}}}}. \tag{4.4}$$

Now, the factors $\{\lambda_k\}_{1 \dots n_{\text{MU}}}$ have to be determined accordingly. Setting $\lambda_{k_{\text{MU}}} = q(k_{\text{MU}})$ would not yield the required exponential distribution of probabilities, because the MU center points $\mathbf{x}_{k_{\text{MU}}}$ have varying distances between each other. Therefore, the accumulated total probability of all fibers to be associated to a particular MU is different for each MU. This is the case even before scaling with any factors $\{\lambda_k\}$.

Instead, the values of the factors have to be determined by solving a global optimization problem. The objective function to be minimized is given by

$$F(\{\lambda_k\}_{1 \dots n_{\text{MU}}}) = \sum_{k_{\text{MU}}=1}^{n_{\text{MU}}} \left(q(k_{\text{MU}}) - \sum_{i=1}^{n} \sum_{j=1}^{n} p(i, j, k_{\text{MU}}; \{\lambda_k\}_{1 \dots n_{\text{MU}}})/n^2 \right)^2. \tag{4.5}$$

It sums up the quadratic error for every MU between the desired, exponentially distributed probability $q(k_{\text{MU}})$ per fiber and the achieved probability per fiber under the current set of the scaling factors λ_k. The achieved probability is computed by a sum over all fibers (i, j) and the formulated radial probability density function p divided by n^2 to get the value per fiber. After solving the optimization problem and plugging the factors $\{\lambda_k\}$ into Eq. (4.4) we get every probability for a fiber to be in an MU by Eq. (4.4). The optimization problem is described in more detail in the following section.

4.2.2 Algorithm to Solve the Optimization Problem

The optimization problem to be solved in order to compute the scaling factors in Eq. (4.4) can be stated as:

$$\text{“Find} \quad \{\lambda_k\}_{1\dots n_{\text{MU}}} \text{ with } \lambda_k > 0 \quad \text{s.t.} \quad F(\{\lambda_k\}_{1\dots n_{\text{MU}}}) \quad \text{is minimal”.} \tag{4.6}$$

The objective function F was given in Eq. (4.5). The solution is obtained by a Quasi-Newton method, more specifically the limited-memory version of the *Broyden-Fletcher-Goldfarb-Shanno* algorithm with box constraints by the authors of [Byr95]. Their Fortran implementation is made accessible in Python by the *SciPy Optimize* package.

With increasing number n^2 of fibers and increasing number n_{MU} of MUs, the evaluation duration for the objective function and the number of optimization parameters increases. For numbers about $n^2 > 1000$ and $n_{\text{MU}} > 25$, the solution times become unfeasible.

As a remedy we develop an algorithm to split the large optimization problem into multiple smaller ones which reduces the total runtime. The set of factors $\{\lambda_k\}_{1\dots n_{\text{MU}}}$ is partitioned into chunks, i.e., subsets of given size $n_{\text{per_chunk}}$ leading to a total of $n_{\text{chunks}} = \lceil n_{\text{MU}}/n_{\text{per_chunk}} \rceil$ chunks. Remainder chunks towards the end potentially get one set element less. The factors for chunk number c are selected in a strided manner as $\{\lambda_k\}$ with indices $k = c, c + n_{\text{chunks}}, c + 2\,n_{\text{chunks}}, \dots$. For example, for $n_{\text{MU}} = 13$ and $n_{\text{per_chunk}} = 4$ we get chunks of sizes $4, 3, 3, 3$ and subsequently solve for $\{\lambda_k\}$ with $k \in \{1, 5, 9, 13\}$, $\{2, 6, 10\}$, $\{3, 7, 11\}$, $\{4, 8, 12\}$.

A number of n_{chunks} optimization problems is solved subsequently where the optimization parameters are each time given by the next chunk. During this loop, more and more scalar factors are determined. Initially, all scalar factors λ_k are set to one. After each solved optimization problem, the respective λ_k values are updated with the values of the found minimizer. The number $n_{\text{factors_up_to_chunk}}$ of already solved scalar factors up to the current iteration starts with zero and increases by $n_{\text{per_chunk}}$ after each iteration, finally arriving at n_{MU} after the last iteration.

These smaller optimization problems have a similar formulation as the overall problem with different values for some variables. The formulation of the optimization problem involves Eqs. (4.1) and (4.3) to (4.5). The number n_{MU} of motor units in Eqs. (4.4) and (4.5) is replaced by the number $(n_{\text{factors_up_to_chunk}} + n_{\text{per_chunk}})$ of factors that will have been solved after the current iteration.

As each of the small optimization schemes only solves for $n_{\text{per_chunk}}$ factors, the argument of the objective in Eq. (4.5) contains only the factors of the current chunk. The other $n_{\text{factors_up_to_chunk}}$ factors in the definition that are not given by the argument of the objective are set to the solutions obtained in previous iterations.

The indexing of the MU center positions $\mathbf{x}_{k_{\text{MU}}}$ in Eq. (4.3) is adjusted such that only the MUs up to and including the current chunk are considered. The indexing in the exponential progression formulated by Eq. (4.1), however, stays the same, here the argument k_{MU} of the function q refers to the full set of MUs.

In summary, the iteratively considered settings contain an increasing number of MUs. The number of optimization parameters and, thus, new MUs is kept constant while the number of summands in the objective function increases. In consequence, the evaluation of the objective function gets more expensive with increasing iteration number while the size of the optimization problem stays constant. The latter has more influence on the optimizer duration. By increasing the chunk size $n_{\text{per_chunk}}$, the size of the small optimization problems can be decreased to any value. This makes the presented algorithm applicable for any large number n_{MU} of MUs.

The result in this approach is not exactly the same as if one big optimization problem including all scalar factors at once would be solved. However, the error is small because of the interleaved MUs indices that are considered in every iteration. Because the resulting MU distribution finally gets drawn from the computed random probability distribution the error is hardly noticable in the result.

4.2.3 Sampling Motor Unit Indices from the Given Probabilities

The next step is to assign actual MU numbers to every fiber. Drawing samples Y with the given probabilities $p(i,j,k_{\text{MU}})$ is done using inverse transform sampling. The inverse cumulative distribution function (CDF) $F_p^{-1}(k_{\text{MU}})$ is applied onto a random value X drawn from a continuous uniform distribution $\mathcal{U}$:

$$Y = F_p^{-1}(X) \quad \text{with } X \sim \mathcal{U}\big(0, F_p(n_{\text{MU}})\big), \quad F_p(k_{\text{MU}}) = \sum_{\ell_{\text{MU}}=1}^{k_{\text{MU}}} p(i,j,\ell_{\text{MU}}).$$

Computing the inverse of the CDF is computationally cheap as the probabilities are discrete and, thus, a loop over the values of the CDF suffices.

To reduce outliers during the random sampling where some MUs get exceptionally little fibers (such as none) or exceptionally many fibers assigned, the sampling procedure is performed five times. Each time, a histogram with bins for the MUs is computed and provides the number of fibers per MUs. For all MUs k that have zero fibers assigned, the one fiber where the probability for the respective MU k is highest is determined. This is usually the fiber closest to the center point $\mathbf{x}_k$ of the MU k. The MU assignment of this fiber is changed to k, such that the MU k is no longer empty but is associated to one fiber.

In each of the five iterations, the squared error between the sampled MU sizes and the expected sizes according to the probabilities, given by $q(k_{\text{MU}}) \cdot n^2$ is computed. The MU assignment of the iteration that yielded the smallest error is used for the final result of method 1.

4.3 Method 2: Assignment of Motor Units to a Selection of Fibers

The second method proceeds similar to the first method in that at first the probability for a specific MU is defined for any fiber (i, j) in the $n \times n$ grid. Then the actual MU assignments are sampled from the probability distributions. The difference to the first method is that any fiber is also allowed to not be assigned to any MU. This makes the definition of the probabilities easier and no optimization is required.

The three conditions defined in Sec. 4.2.1 are also imposed for the second method. The definition of the MU center positions $\mathbf{x}_{k_{MU}}$ follows the same low-discrepancy series. Also, the radial kernel function Eq. (4.3) can be reused to describe the spatial distribution of probability for a given MU. Instead of Eq. (4.4), the probability is formulated directly as the product of the kernel function $\hat{p}$ and the exponential progression q:

$$\tilde{p}(i, j, k_{\text{MU}}) = \hat{p}(i, j, k_{\text{MU}}) \cdot q(k_{\text{MU}}).$$

To ensure that the function is within the bounds of a probability, $p \leq 1$, the result is divided by the maximum occuring value,

$$p(i, j, k_{\text{MU}}) = \frac{\tilde{p}(i, j, k_{\text{MU}})}{\max\limits_{\substack{\bar{i},\bar{j}=1,\dots,n \\ \bar{k}_{\text{MU}}=1,\dots,n_{\text{MU}}}} \tilde{p}(\bar{i}, \bar{j}, \bar{k}_{\text{MU}})}.$$

In the sampling step, for every fiber (i,j) the probabilities $p(i,j,k_{\text{MU}}), k_{\text{MU}} = 1,\dots,n_{\text{MU}}$ for the MUs and the remaining probability $\bar{p} = 1 - \sum_{k_{\text{MU}}} p(i,j,k_{\text{MU}})$ are computed. Then, the MU index is randomly drawn from the set of numbers $\{1,\dots,n_{\text{MU}}\}$ and a "null" event that corresponds to no assigned MU, according to the computed probabilities p and $\bar{p}$.

As a result, we get MUs that satisfy all three conditions formulated in Sec. 4.2.1, including the approximate exponential progression of the MU sizes. However, the resulting number of fibers with assigned MU is an outcome of the algorithm and cannot be prescribed.

4.4 Assignment of Different Motor Units for Neighboring Fibers

As mentioned in Sec. 4.1, one observation in staining experiments was that the fibers of an MU typically do not touch each other, i.e., neighboring fibers always belong to different MUs. However, the presented methods 1 and 2 assign neighboring fibers with a high probability to the same MU. To create a grid with an MU assignment that avoids this behavior, the idea is to interleave four smaller grids where the MU assignments were obtained independently of each other but with the same parameters. In the following, this method is named "method 1a" and "2a" depending on whether the partial grids were handled with method 1 or 2.

First, either method 1 or method 2 are applied four times to smaller grids of fibers, the *partial grids*, as visualized in Fig. 4.3. The partial grids contain $n_{\text{part}} \times n_{\text{part}} = n/2 \times n/2$ fibers. In each partial grid, MU assignments with $n_{\text{MU,part}} = n_{\text{MU}}/4$ MUs are created. The basis b is changed to $b_{\text{part}} = b^4$ and the standard deviation of the kernel function is changed to $\sigma_{\text{part}} = \sigma/2$. In result, we get the same exponential distribution of number of fibers per MU on every partial grid.

The four smaller grids are then merged according to the scheme shown in Fig. 4.3. Fibers of the first partial grid directly touch only fibers of the third and fourth grids and touch fibers of the second partial grid diagonally. By using this scheme, neighboring fibers in any of the partial grids are always separated by fibers of other grids.

The MU indices that are assigned in the partial grids are mapped to the resulting, large grid also in an interleaved manner. MU k of the ℓth grid is mapped to the resulting MU $(4(k-1)+\ell)$. For illustration, the MUs 1,2,3 of the first grid are mapped to MUs $1,5,9$,

Figure 4.3: Repeating scheme for interleaving the four partial grids. The partial grids are indicated by the numbers and have different colors. The pattern is highlighted at the bottom left of the figure.

MUs 1,2,3 of the second grid are mapped to MUs $2, 6, 10$, etc. Since the MU sizes in the partial grids follow the defined exponential progression, this also holds for the final MUs.

The location of the MU center points is determined by contiguous elements of the same Weyl sequence given in Eq. (4.2) for all partial grids. First, all MU center points for the first partial grid are assigned, then for the second, third and forth. By this construction, all MU center points are distributed with similar spacing between each other and the placement of similar sized MUs close to each other is avoided.

4.5 Results and Discussion

In the following, results of the methods described in Sections 4.2 to 4.4 with different parameters are presented. Figure 4.4 shows the resulting assignment of MUs to fibers for methods 1 and 2. The number of fibers per coordinate direction is $n = 13$, a number of $n_{\text{MU}} = 10$ MUs is considered and two different values for the kernel function parameter σ are used.

In Fig. 4.4a, method 1 is used with basis $b = 1.2$ and a kernel function with standard deviation of a tenth of the grid, $\sigma = n/10$. Each square represents one fiber, their colors refer to the MU index as indicated by the legend. Colored crosses visualize the center points $\mathbf{x}_{k_{\text{MU}}}$ of the respective MUs.

It can be seen that the MU territories, i.e., the regions of the fibers of an MU are located around the center points of the MUs. Because of the random sampling, the fibers of an MU are not all located closely together but spread over a larger area. Especially for MU 8, depicted by light orange color, some fibers are located further away from the center point, which is approximately at the center of the grid. On the other hand, for MU 10 most of the red marked fibers are located close to the center point of the MU, which can be found in the center of the lower third of the grid.

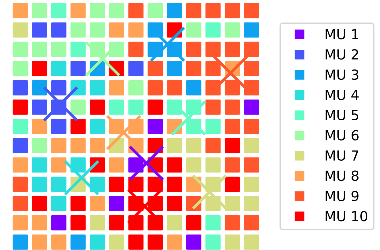
The histogram for the setting considered in Fig. 4.4a is shown in Fig. 4.5. It can be seen that the number of fibers per motor unit approximately follows the prescribed exponential function with basis $b = 1.2$. The observed deviation is due to the random sampling.

Figure 4.6 shows the values of the probability function of a specific MU for all fibers, formulated in Eq. (4.4). Figures 4.6a and 4.6b correspond to the scenario considered in Fig. 4.4a. The comparison shows that the probability is the highest around the center of MU 3 and MU 9, respectively. When moving away from the MU centers, the probability follows approximately the shape of the radial kernel function in Eq. (4.3). An image of the radial kernel function in higher resolution is given by Fig. 4.6c, where the probability function is depicted for MU 41 in a scenario with 50 MUs in a grid of 37×37 fibers.

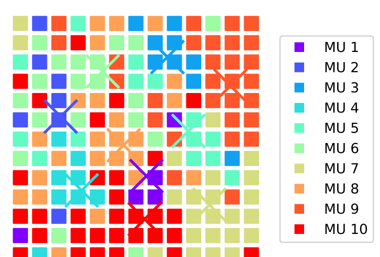
It can be observed, however, that the probability distribution in Figures 4.6a and 4.6b does not entirely follow the kernel function. The effects of the scaling factors $\{\lambda_k\}$ in Eq. (4.4) are visible, e.g., at the top-most and right-most fibers. There, the probability increases again compared to the interior of the grid. The purpose of the scaling factors is to enforce the exponential distribution of MU sizes.

This effect is illustrated more clearly in Fig. 4.7. It shows the value of $p(i, j, k_{\text{MU}})$ for the top right fiber in the grid, $(i, j) = (13, 13)$, for all values of k_{MU}. The blue curve indicates the probability that results from the kernel functions, only according to the distance of the top right fiber to the respective MU centers. In other words, the scaling factors $\{\lambda_k\}_{1\dots n_{\text{MU}}}$ are removed or equivalently set to one. By inspecting again Fig. 4.4a, it can be seen that the MU centers of MUs 9, 3 and 5 are—in this order—closest to the top right fiber whereas MU 4 and 10 are the furthest away. Consequently, the blue curve in Fig. 4.7 has peaks at 9, 3 and 5 and low values for 4 and 10.

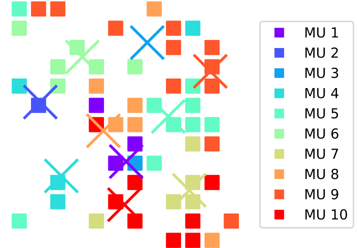
When incorporating the scaling factors $\{\lambda_k\}_{1\dots n_{\text{MU}}}$ that were found by the optimization problem in Eq. (4.6), the probabilities change to the orange curve. It can be seen that the probability for the fiber to be in MU 9 increases. MU 9 which is expected to have a rather high number of fibers according to the exponential progression. It gets more fibers from the top right corner. The areas left to and below the center of MU 9 are at the same time



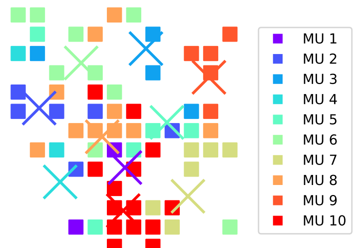
(a) Result for method 1 with $\sigma = n/10 = 1.3$



(b) Result for method 1 with $\sigma = n/100 = 0.13$



(c) Result for method 2 with $\sigma = n/10 = 1.3$, only 54 out of 169 fibers, i.e., 32 % have an assigned motor unit.



(d) Result for method 2 with $\sigma = n/100 = 0.13$, only 63 out of 169 fibers, i.e., 37 % have an assigned motor unit.

Figure 4.4: Resulting MU assignments to a grid of $n \times n = 13 \times 13 = 169$ fibers. Each MU is represented by a color, the MU center points $\mathbf{x}_{k_{\text{MU}}}$ are the same for all scenarios and are shown by the colored crosses.

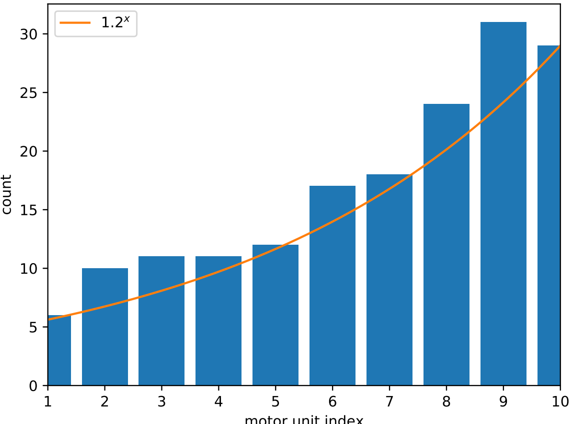
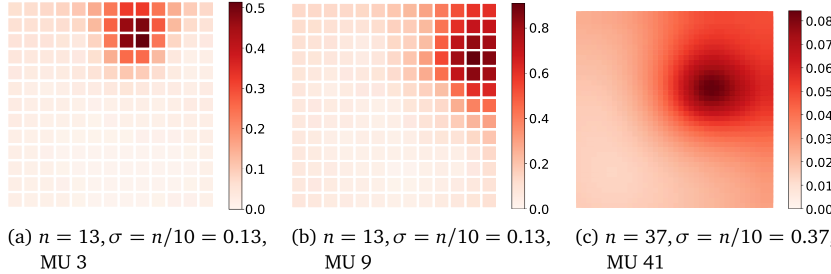


Figure 4.5: Histogram of the number of fibers per MU in Fig. 4.4a. The orange line corresponds to the ideal exponential distribution $y = c \cdot 1.2^x$.



(a) $n = 13, \sigma = n/10 = 0.13$, MU 3

(b) $n = 13, \sigma = n/10 = 0.13$, MU 9

(c) $n = 37, \sigma = n/10 = 0.37$, MU 41

Figure 4.6: Probability at every fiber to be in a given MU, for different grid sizes and number of MUs.

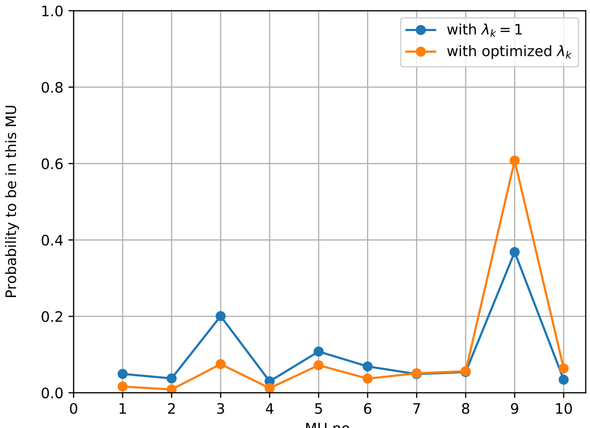


Figure 4.7: Probability of the top right fiber in the 13×13 grid to be in a given MU, without considering the scaling factors $\{\lambda_k\}_{1 \dots n_{\text{MU}}}$ (blue) and including the scaling factors (orange).

close to the centers of MU 3 and 5 and therefore can also be occupied by fibers of MUs 3 and 5. Thus, the optimization performs an exchange where MU 9 forgoes the bottom and left fibers and, conversely, obtains portions of the fibers in the top right area from MUs 3 and 5. Consequently, the probability in Fig. 4.7 decreases for MUs 3 and 5. The shape of the final probability distribution for MU 9 in Fig. 4.6b is the kernel function stretched to the top and right. By looking at Fig. 4.4a, it can be seen that, by chance, the top right fiber indeed gets assigned to MU 9.

The influence of the kernel width σ is demonstrated by comparing Fig. 4.4a with Fig. 4.4b. In Fig. 4.4b the value of σ is only a tenth of the value in Fig. 4.4a. All other parameters are the same such that a similar exponential distribution of MU sizes is obtained. It can be seen that the MU territories are less interleaved and have clearer borders. For example, the territory of MU 7 at the bottom left of the domain has a cohesive shape in Fig. 4.4b whereas the respective fibers are more scattered in Fig. 4.4a.

In comparison, the results for method 2 with the same two values of σ are shown in Figures 4.4c and 4.4d. All other parameters are kept the same. It can be seen how method 2 only associates some fibers with MUs. For the larger standard deviation σ in Fig. 4.4c, only 32 % of the fibers get assigned to a MU, for the smaller value of σ, the fraction is

slightly higher with 37 %. Similar to method 1, the effect of more cohesive MU territories for smaller σ values can also be observed in the results of method 2.

Next, the two methods 1 and 2 are investigated for a higher number of $n_{\text{MU}} = 100$ motor units and a grid of $n \times n = 67 \times 67 = 4489$ fibers.

Figure 4.8a shows the MU center points $\mathbf{x}_{k_{\text{MU}}}$. The color corresponds to the MU index and follows the same rainbow color scheme as in Fig. 4.4. Since the construction scheme is the deterministic Weyl sequence in Eq. (4.2), the first 10 MU center positions are the same as for the scenario with $n_{\text{MU}} = 10$. It can be seen that the MU centers have similar distances throughout the grid and that, in general, MUs located next to each other have different colors and therefore are differently sized.

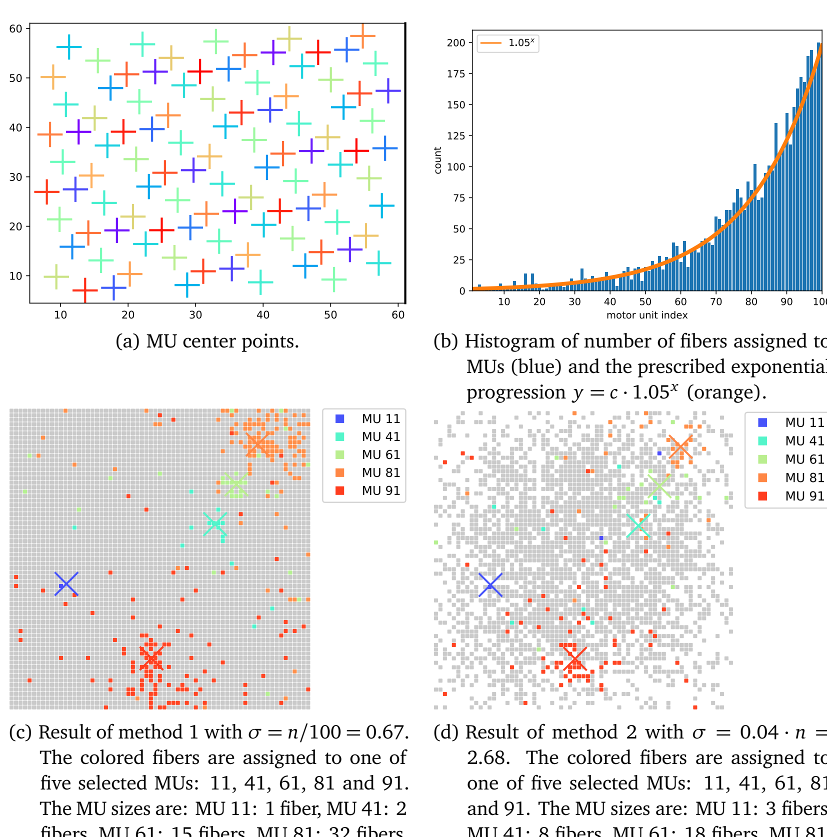
Figure 4.8b shows the histogram of the MUs, i.e., the number of fibers per MU. Following [Eno01] the prescribed basis for the exponential progression was reduced because of the higher number of fibers. It was set to $b = 1.05$. It can be seen that the resulting MU size distribution closely matches the prescribed function. Because the ratio of fibers to MUs (4489/100) is higher than in the previous setting (169/10), the deviation of the realized MU sizes from the prescribed curve appears smaller than in Fig. 4.5.

Figure 4.8c shows the result for method 1. The width of the radial kernel function was chosen as $\sigma = n/100 = 0.67$. Only the fibers of five selected MUs and their center points are visualized for better clarity.

The algorithm for the 4489 fibers and 100 MUs was performed with a chunk size of $n_{\text{per_chunk}} = 10$, yielding a total number of $n_{\text{chunks}} = 10$ chunks. The runtime was 45 min 53.5 sec on a single core of an Intel Core i5-6300U CPU with base frequency of 2.40GHz and 19.5 GiB of RAM.

Figure 4.8d shows the result for method 2. All resulting fibers that were associated to an MU are shown as gray or colored squares, leaving white spaces for unassigned fibers. Again, only the fibers of five selected MUs are colored. The kernel parameter was set to $\sigma = 0.04 \cdot n = 2.68$ which resulted in 2328 of 4489 fibers or 52 % of the fibers being assigned an MU. When the parameter is instead set to $\sigma = n/100 = 0.67$ as in the study with 10 MUs before, the result assigns only 136 fibers or 3 %. This shows that method 2 is very sensitive to the choice of the standard deviation parameter σ.

The comparison with Fig. 4.8c shows that the resulting MU territories are more dispersed than for method 1. This can be explained with the higher value of σ. Obtaining “sharper” MU territories would require a smaller σ, however, this results in less fibers being assigned to MUs.



(a) MU center points.

(b) Histogram of number of fibers assigned to MUs (blue) and the prescribed exponential progression $y = c \cdot 1.05^x$ (orange).

(c) Result of method 1 with $\sigma = n/100 = 0.67$. The colored fibers are assigned to one of five selected MUs: 11, 41, 61, 81 and 91. The MU sizes are: MU 11: 1 fiber, MU 41: 2 fibers, MU 61: 15 fibers, MU 81: 32 fibers, MU 91: 72 fibers

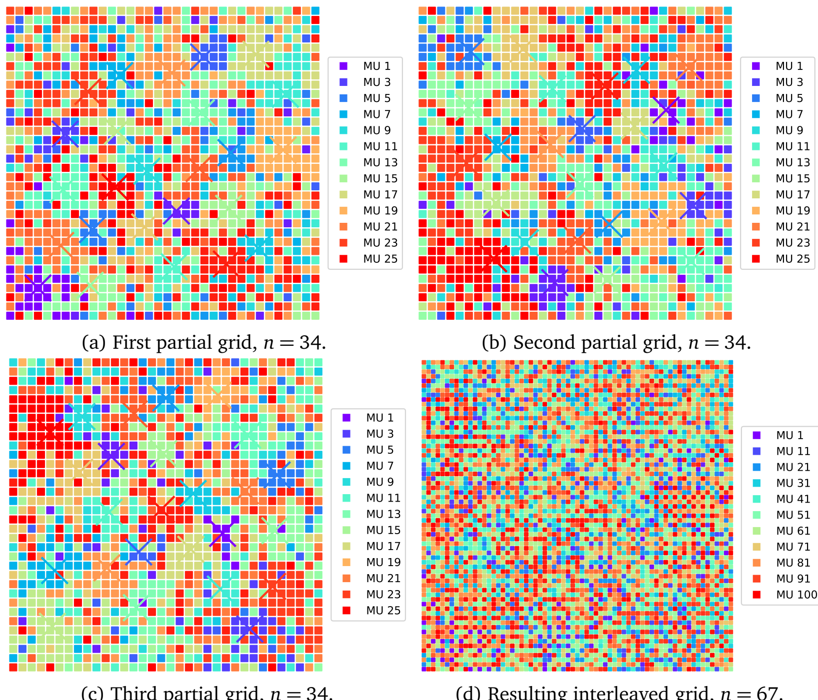
(d) Result of method 2 with $\sigma = 0.04 \cdot n = 2.68$. The colored fibers are assigned to one of five selected MUs: 11, 41, 61, 81 and 91. The MU sizes are: MU 11: 3 fibers, MU 41: 8 fibers, MU 61: 18 fibers, MU 81: 34 fibers, MU 91: 75 fibers

Figure 4.8: Results of the presented algorithm to assign MUs to fibers, using a grid of 67×67 fibers and 100 MUs.

Furthermore, Fig. 4.8d shows that the fiber density decreases towards the outer border of the domain. In reality, staining studies on skeletal muscles do not find this effect.

An advantage of method 2 over method 1 is that we do not have to solve any optimization problem. In consequence, the algorithm for the scenario in Figure 4.8d was completed in 8 sec on the same hardware as before.

Next, the extension of methods 1 and 2, called 1a and 2a, are investigated that ensure that neighboring fibers are not associated to the same MU. Figure 4.9 shows results for



(a) First partial grid, $n = 34$.

(b) Second partial grid, $n = 34$.

(c) Third partial grid, $n = 34$.

(d) Resulting interleaved grid, $n = 67$.

Figure 4.9: Assignment of motor units to fibers using method 1a. Shown are the first three partial grids (a)-(c) to be interleaved and the result (d), parameters $n = 67, n_{\text{MU}} = 100, \sigma = n/10 = 6.7, b = 1.05$.

method 1a for $n_{\text{MU}} = 100$ MUs. In Figures 4.9a to 4.9c, three of the four partial grids with $n = 34$ are shown. Because parameters are the same for those smaller grids, the generated MU assignments look similar for all partial grids, except for different MU center positions. In Fig. 4.9d, the resulting grid with $n = 67$ is shown that is obtained by interleaving the four partial grids. In this MU association, all neighboring fibers belong to different MUs. The resulting distribution of MU sizes is shown in Fig. 4.10. It can be seen that the algorithm for method 1a achieves the approximate, prescribed exponential progression.

An advantage of method 1a is also that the runtime decreases compared to method 1. The result in Fig. 4.9d could be computed in 5 min 52.4 sec with $n_{\text{per_chunk}} = 10$ or in 4 min 53.0 sec with $n_{\text{per_chunk}} = 5$, compared to the 45 min 53.5 sec of method 1.

Method 2a cannot be reasonably used with the same parameters as method 1a. If it is

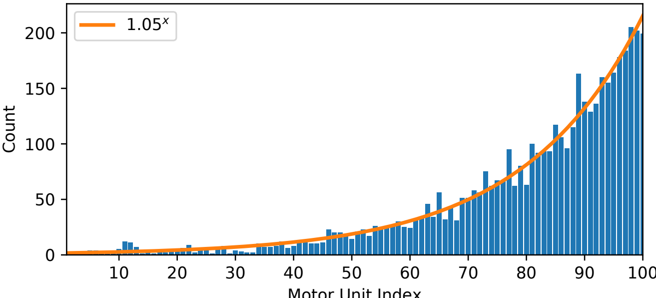


Figure 4.10: Histogram of number of fibers assigned to MUs for method 1a, for the scenario that is shown in Fig. 4.9.

used to generate fibers assigned to 100 MUs, a grid of 67×67 leads to the majority of MUs having only 1 fiber. Therefore, a larger grid is needed. Figure 4.11 shows the result for $n = 251$. The result assigns 13 618 of the $n^2 = 63\,001$ initial fibers to MUs, i.e. 22 %. The number of fibers per MU varies between 41 and 244. As can be seen, fibers of the same MU are separated by either a fiber of a different MU or by an unassigned fiber, i.e., a hole in the grid.

How To Reproduce

Run the script `generate_fiber_distribution.py` without arguments to get usage information. The script contains the implementation for all three presented methods. For example, to run method 1a to get the result of Fig. 4.9, use:

```
generate_fiber_distribution.py
  ↪ MU_fibre_distribution_combined_67x67_100 100 3 1 67 1.05 100 10
```

Then, existing fiber distribution files can be visualized by the following script:

```
$OPENDIHU_HOME/examples/electrophysiology/input/
  ↪ plot_fibre_distribution_2d.py
  ↪ MU_fibre_distribution_combined_67x67_100.txt 67
```

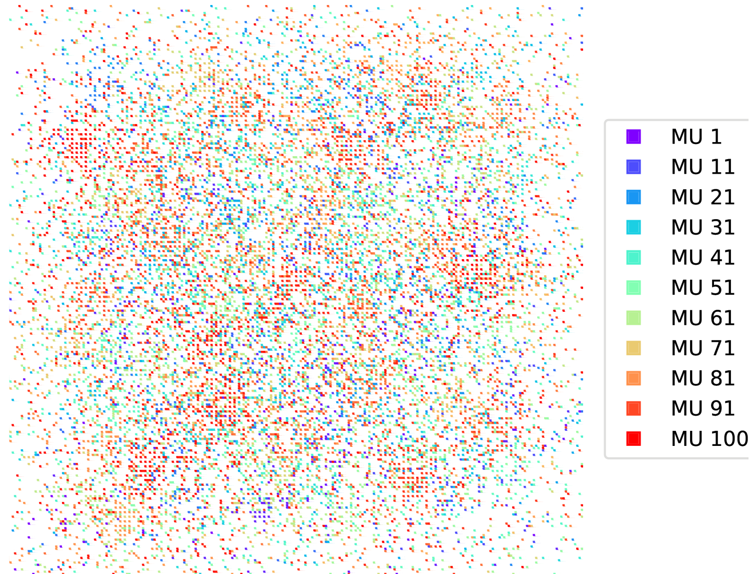


Figure 4.11: Assignment of motor units to fibers using method 2a and parameters $n = 251, n_{\text{MU}} = 100, \sigma = n/100 = 2.51, b = 1.05$.

4.6 Summary and Conclusion

In the beginning of this chapter, two methods 1 and 2 for associating MUs with fibers in a given 2D grid were presented. The methods were constructed based on biophysical properties of MU distribution. The fibers were located in intermingling MU territories that were each centered at different MU center points. The density of fibers belonging to an MU decreased with higher distance from the center and was described by a radial kernel function. The number of fibers assigned to the motor units approximately followed an exponential progression where the first MU contained the lowest number of fibers and the last MU contained the largest amount. Whereas method 1 assigned MUs to all available fibers, method 2 only assigned MUs to some fibers yielding a lower number of fibers in the result. Evaluation of the literature showed that no comparable method with these properties existed previously.

Next, the methods 1a and 2a were introduced that built upon methods 1 and 2. They ensured that neighboring fibers were assigned to different MUs, another behavior that

was known from anatomical studies.

Steps of the algorithms and their final satisfaction of the design criteria were demonstrated with various visualizations. Results were shown for different parameter values. The influence of the kernel width on the “sharpness” or intermingledness of MU territories was pointed out.

It was found that methods 2 and 2a typically produced results where only 20-50% of the fibers get an MU assigned. This ratio highly depended on the problem size and the kernel function width and no direct predictions about the number of resulting fibers was possible. Since the kernel parameter at the same time also influenced the sharpness of the MU territories, adjusting parameters to the desired outcome was an issue for these methods. Reasonable results were only achieved for a higher number of initial fibers. In contrast, methods 1 and 1a robustly produced exponentially distributed MU assignments for all parameter values.

In method 2a for large grid sizes the fibers were dispersed over the grid and “holes”, i.e., unassigned fibers, were present throughout the domain. The fiber density decreased towards the boundaries of the domain. No experimental evidence exists that this behavior occurs in reality. It might also be unfavorable when EMG simulations are performed where the boundary layers contribute most to the measured EMG signal on the muscle surface. In contrary, method 1a did not show this behavior.

The runtime for 4489 fibers and 100 MUs was over 45 min for method 1 and below 10 sec for method 2. The large difference could be explained with the optimization problem that needed to be solved for method 1. To handle large runtimes for a high number of MUs, an algorithm was presented that splits the optimization problem in smaller chunks that could be solved faster.

With the use of the extended methods 1a and 1b, runtime decreased. For method 1a the runtime was under 6 min. These runtimes are all considered acceptable since the task occurs only once during preprocessing.

In conclusion, the developed method 1a proved to be robust for all tested parameter combinations and fulfilled all considered biophysical properties of MU distributions. The exponential distribution of MU sizes and the sharpness of MU territories are adjustable through parameters. If the condition that neighboring fibers belong to different MUs is not desired, method 1 can be used instead.

The presented methods are implemented and made available as Open Source software within OpenDiHu. The program stores the resulting MU assignments in a plain text file

format that is compatible with both OpenCMISS Iron and OpenDiHu. Thus, it can and will be used in simulations of the multi-scale chemo-electromechanical model.

Chapter 5

Models and Discretization

In this chapter, the mathematical description of the multi-scale model and its discretization is presented. We use the multi-scale chemo-electro-mechanical model that was introduced in literature [Röh12; Hei13; Hei15; Mor15]. Additional models known from literature are incorporated that were previously only simulated in isolation: The multidomain description for electrophysiology [Klo20], a model of neural stimulation [Cis08] and sensory organ models such as the muscle spindle model of Mileusnic et al. [Mil06b]. Similarly, models of Golgi tendon organs can be added [Mil06a].

Figure 5.1 shows an overview of the components of the implemented multi-scale model. A pool of motor neurons drives the stimulation of the muscular system in Fig. 5.1 (a). The axons of each motor neuron innervate the muscle fibers corresponding to the same MU and transmit rate-encoded stimulation signals.

In the muscle tissue, action potentials propagate starting at the neuromuscular junctions and subsequently reach the whole length of the muscle. In our multi-scale model, two different formulations are available to describe this phenomenon. The multidomain

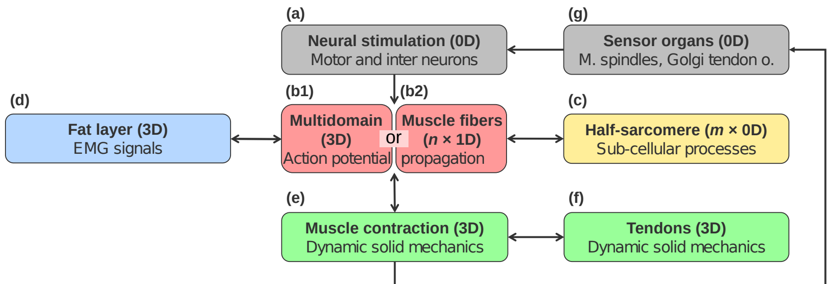


Figure 5.1: Interacting components of the multi-scale model.

description (Fig. 5.1 (b1)) models the MUs from a homogenized 3D perspective. The description with muscle fibers (Fig. 5.1 (b2)) models action potential propagation explicitly with n 1D muscle fibers.

Both of these descriptions of electrophysiology involve a subcellular model (Fig. 5.1 (c)). This model describes the ionic processes involving the fiber membranes and taking place within one half of a sarcomere as the smallest unit to generate muscle forces. A large number m of instances of this model has to be computed.

In addition to the physiology of the muscle, a layer of body fat and skin on top of the muscle belly can be added to the model. This 3D fat layer (Fig. 5.1 (d)) is used to simulate EMG recordings on the skin surface. The model for the fat layer is unidirectionally coupled with the muscle fiber model (Fig. 5.1 (b2)) or bidirectionally coupled with the multidomain model (Fig. 5.1 (b1)). Using the multidomain model, it is, thus, possible to simulate external stimulation by electrodes on the skin, which is subject to research in neuroprosthetics.

The activated muscle generates force by subcellular processes on a molecular scale. They are computed on the cellular level by the half-sarcomere model (c). On the macroscopic scale, stresses lead to strains and contraction of the muscle. This effect is described by the muscle contraction model on a 3D domain (Fig. 5.1 (e)). The description is coupled with the electrophysiology models (b1),(b2) by the geometry of the contracting muscle and fibers. It is coupled with the subcellular model by the generated active stresses of the half-sarcomere. Displacements and stresses can be computed for the muscle belly itself, but also for the connected body fat layer and for elastic tendons (Fig. 5.1 (f)). Depending on the research questions, the contraction model is either formulated quasi-static or fully dynamic taking into account inertia effects.

Sensory organs such as muscle spindles and Golgi tendon organs sense fiber stretch and contraction velocity (Fig. 5.1 (g)). They are connected with the motor neuron pool by layers of interneurons and modulate the stimulation in Fig. 5.1 (a).

In this chapter, Sec. 5.1 presents mathematical descriptions of the electrophysiology model components in the multi-scale framework and Sec. 5.2 derives the solid mechanics models. Then, Sections 5.3 and 5.4 address the spatial and temporal discretizations of the electrophysiology and mechanics descriptions, respectively.

5.1 Electrophysiology Model Equations

In the following, more details and mathematical descriptions are given for the outlined models. The section begins with the 0D half-sarcomere model in Sec. 5.1.1, followed by the bidomain and monodomain models in Sections 5.1.2 and 5.1.3, which constitute the muscle fiber based model of electrophysiology. Section 5.1.4 continues with the multidomain model. Electric conduction in the body fat layer is described in Sec. 5.1.5. An overview of the continuum mechanics model used for muscle contraction is given in Sec. 5.2.

5.1.1 Subcellular Model

Propagation of electric stimuli along muscle fibers involves activation and deactivation of ion channels and ion pumps in the fiber membrane (the sarcolemma) and in the transverse tubules. Functioning of these processes on the subcellular scale have first been suggested in 1952 by Hodgkin and Huxley after their studies of the squid giant axons [Hod52a; Hod52b]. To date, their mathematical model still serves as the basis for electrophysiology models and some of their predictions, e.g., on gating currents that occur during opening of channels, were experimentally confirmed later.

The fiber membrane separates intra- and extracellular space and can be locally described by an electric circuit. The membrane voltage $V_m = \phi_i - \phi_e$ is the difference between the intra and extracellular potentials ϕ_i and ϕ_e. The membrane stores charges Q, quantifiable by its electric capacitance C_m:

$$Q = C_m \cdot V_m. \tag{5.1}$$

A change in the transmembrane potential, e.g., induced by an action potential leads to a change in Q, which is accounted for by an electric current I over the membrane. This can be formally obtained by the derivative of Eq. (5.1) with respect to time:

$$\frac{\mathrm{d}Q}{\mathrm{d}t} = C_m \cdot \frac{\mathrm{d}V_m}{\mathrm{d}t}. \tag{5.2}$$

The current $I = \mathrm{d}Q/\mathrm{d}t$ is realized by ions passing through the membrane. Significant ions in this process are sodium (Na^+) and potassium ions (K^+). Considering a particular point on the fiber, these ions diffuse through ion-specific channels in the membrane. The

diffusion is driven by an interplay of the ion concentration gradient and the electric field that is caused by action potentials.

Without any electric field imposed by action potentials, the equilibrium state of the diffusion process for sodium and potassium ions is given by their Nernst potentials E_{Na^+} and E_{K^+}. These voltage levels depend on logarithmic relations between extra- and intracellular concentrations scaled by constants describing the thermal energy and the number of electrons. In thermodynamic equilibrium, the membrane voltage is equal to the Nernst potential E_i of the involved ions i. At a higher membrane voltage V_m, the remainder $(V_m - E_i)$ is the part of the electric field that drives the ion fluxes and electric currents through the membrane. The currents depend on the conductivity g_i of the membrane for ion i.

Apart from sodium and potassium ions, the diffusion of less frequent ions and ionic pumps can be lumped by a leakage current I_L that is modeled by a channel with constant conductivity $\bar{g}_L$. With this, the total ionic membrane current I_{ion} is formulated as

$$I_{\mathrm{ion}}(V_m) = I_{\mathrm{Na}^+} + I_{\mathrm{K}^+} + I_L \tag{5.3a}$$

$$= g_{\mathrm{Na}^+}(V_m - E_{\mathrm{Na}^+}) + g_{\mathrm{K}^+}(V_m - E_{\mathrm{K}^+}) + \bar{g}_L(V_m - E_L). \tag{5.3b}$$

The conductivities g_{Na^+} and g_{K^+} for the sodium and potassium channels depend on the transmembrane voltage V_m and its history.

In addition to the ionic current I_{ion}, an externally driven current I_{ext} can be modeled that occurs as a result of neural stimulation at the neuromuscular junctions. Substituting the current $I = \mathrm{d}Q/\mathrm{d}t$ in Eq. (5.2), we get the following differential equation for the membrane voltage V_m:

$$C_m \cdot \frac{\mathrm{d}V_m}{\mathrm{d}t} = -I_{\mathrm{ion}}(V_m) + \frac{I_{\mathrm{ext}}}{A}. \tag{5.4}$$

The negative sign of the ionic current I_{ion} is in accordance with the definition of the membrane voltage as $V_m = \phi_i - \phi_e$. The external current I_{ext} is divided by the surface area A of the stimulating electrode or neuromuscular junction, as the description considers an infinitesimal area on the membrane.

Hodgkin and Huxley suggested that ion channels can be activated and deactivated. This molecular process requires independent "gating" particles to move to a new position in order for a channel to be activated. For the potassium channel, four of these independent events have to occur, each modeled by a probability n. The resulting probability for the

channel to open is, thus, n^4. For the sodium channel, three such events are assumed for activation and another one for the deactivation of the channel, described by the probabilities m and h, respectively. The values of the probabilities change over time and modulate the conductivities of the ion channels:

$$g_{\mathrm{Na}^+} = \bar{g}_{\mathrm{Na}^+} \cdot m(t)^3 \cdot h(t), \qquad g_{\mathrm{K}^+} = \bar{g}_{\mathrm{K}^+} \cdot n(t)^4.$$

Here, $\bar{g}_{\mathrm{Na}^+}$ and $\bar{g}_{\mathrm{K}^+}$ are channel specific constants. The gating variables n, m and h can be interpreted as probabilities for the events to occur or as the amount of occurred events related to all available gating particles. The evolution of the activation probability n is modeled by the following ordinary differential equation (ODE):

$$\frac{\mathrm{d}n}{\mathrm{d}t} = \alpha_n(V_m) \cdot (1-n) + \beta_n(V_m) \cdot n,$$

analogously for h and n. The transition rates between activation probability n and deactivation probability $(1-n)$ are nonlinearly dependent on the membrane voltage V_m.

For a constant V_m, this ODE has an analytical solution

$$n(t) = n_\infty\big(1 - \exp(1 - \frac{1}{\tau_n}\, t)\big), \tag{5.5}$$

which for $t \to \infty$ converges to the equilibrium value $n_\infty := \alpha_n\, \tau_n$ as shown in Fig. 5.2. The time constant $\tau_n := 1/(\alpha_n + \beta_n)$ indicates how fast the solution approaches the equilibrium, e.g., when starting from $n(0) = 0$, half of the value of the equilibrium is reached after $t_{1/2} = \log(2)\,\tau_n$. The smaller τ_n, the stiffer is the ODE, which needs to be considered in the choice of a suitable numerical solution scheme.

Because the transmembrane voltage V_m changes over time, the ODEs for n, m and h have to be solved numerically. Then, the dependent ionic current I_{ion} can be calculated. Thus, the model is a system of differential-algebraic equations (DAE).

The internal states in this model can be combined into a state vector $\mathbf{y} = (n, m, h)^\top$. The combined right-hand side for all states is formulated as a vector-valued function $G(V_m, \mathbf{y})$. In summary, the system of DAEs for the subcellular model on a subcellular domain Ω_s can be written in the following form:

$$\frac{\partial \mathbf{y}}{\partial t} = G(V_m, \mathbf{y}), \qquad I_{\mathrm{ion}} = I_{\mathrm{ion}}(V_m, \mathbf{y}) \quad \text{on } \Omega_s. \tag{5.6}$$

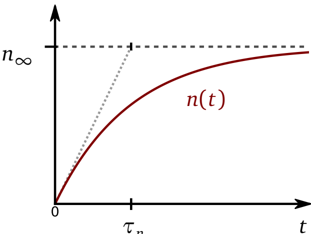


Figure 5.2: Subcellular model: Graph of the analytic solution (red) of the ordinary differential equation that is part of the activation model of ion channels for constant transmembrane voltage and initial condition $n(0) = 0$, given in Eq. (5.5). The variables n_∞ and τ_n can be interpreted as the equilibrium value and a characteristic timescale, respectively.

For an exemplary solution that shows how the membrane potential changes over time, see Fig. 1.2.

The system of equations in Eq. (5.6) together with Eq. (5.4) describe the subcellular processes on a single point $\Omega_s \subset \Omega_f$ on a muscle fiber Ω_f. It does not model the interaction between neighboring points that leads to propagation of action potentials. To account for action potential propagation, ionic currents I_{ion} on multiple points are coupled within the multidomain or fiber models that are formulated in the multi-scale framework. This is described in the following sections, Sections 5.1.2 to 5.1.4. Using these models, the system of ODEs in Eq. (5.6) has to be solved for multiple subcellular points Ω_s^i in the muscle domain.

After Hodgkin and Huxley proposed this model in 1952, more detailed models were formulated that take into account more ion channels, ion pumps and more advanced biochemical processes within the cell. One particular model is the one proposed by Shorten et al. [Sho07], which adds the full pathway from activation to excitation-contraction coupling in the sarcomere. It has a state vector of $\mathbf{y} \in \mathbb{R}^{56}$ and is used to compute active stresses for simulations of muscle contraction. It can also be written in the form given in Eq. (5.6). Apart from I_{ion}, another value $\gamma = H(\mathbf{y}, \dot{\lambda}_f)$ is computed by an additional equation from the vector of states $\mathbf{y}$ and the fiber contraction velocity $\dot{\lambda}_f$, which is given to the model as a parameter. The value γ is a lumped activation parameter in the range $\gamma \in [0, 1]$ that describes the amount of active stress generated in the sarcomere and can be linked to the continuum mechanics model of muscle contraction.

5.1.2 Bidomain Model

A description of electrophysiology on a general 3D muscle tissue is given by the bidomain model formulated by [Tun78; Pes79]. The bidomain model considers the intra (index i) and extracellular spaces (index e) in a homogenized setting, such that the two domains coexist at every spatial point $\mathbf{x} \in \Omega \subset \mathbb{R}^3$. Similar to the setting of the subcellular model, the two domains in the bidomain model have locally varying electric potential fields ϕ_i and ϕ_e that yield a locally varying transmembrane voltage $V_m = \phi_i - \phi_e$. Electric conduction within the two domains is governed by conductivity tensors $\boldsymbol{\sigma}_i$ and $\boldsymbol{\sigma}_e$.

Assuming static conditions, a spatially varying electric potential ϕ induces the electric field $E = -\operatorname{grad} \phi$. According to Ohm's law, the resulting current density j is given by

$$j = \boldsymbol{\sigma}\, E = -\boldsymbol{\sigma} \operatorname{grad} \phi \quad \text{in } \Omega. \tag{5.7}$$

This holds for both intra and extracellular domain, yielding expressions for j_i and j_e.

The intracellular and the extracellular domain are electrochemically coupled. Thus, one assumption is that currents are preserved and a change in current density on one domain corresponds to the opposite change in current density in the other domain. This is expressed by the divergence of the current densities, which in one domain equals to the negated value in the other domain:

$$\operatorname{div}(j_i) = -\operatorname{div}(j_e) \quad \text{in } \Omega. \tag{5.8}$$

This change in current density directly corresponds to a current flow over the membrane:

$$\operatorname{div}(j_i) = A_m\, I_m \quad \text{in } \Omega.$$

Here, the factor A_m describes the membrane area to domain volume relationship. It is needed to convert the units between current per volume and current per area. The membrane current I_m is given by the subcellular model of Hodgkin and Huxley in Eq. (5.4). Neglecting the external current I_{ext} in Eq. (5.4) and using the formulation of the intracellular current density j_i in Eq. (5.7), we get:

$$\operatorname{div}\big(\boldsymbol{\sigma}_i \operatorname{grad}(\phi_i)\big) = A_m \Big(C_m \frac{\partial V_m}{\partial t} + I_{\text{ion}}(V_m)\Big) \quad \text{in } \Omega.$$

The ionic current I_{ion} can be computed by Eq. (5.3b). By plugging Eq. (5.7) also into

Eq. (5.8) and rewriting the equations in terms of the extracellular potential ϕ_e and the transmembrane voltage $V_m = \phi_i - \phi_e$, we get the bidomain equations:

$$\operatorname{div}\big((\boldsymbol{\sigma}_i + \boldsymbol{\sigma}_e)\operatorname{grad}(\phi_e)\big) + \operatorname{div}\big(\boldsymbol{\sigma}_i \operatorname{grad}(V_m)\big) = 0, \tag{5.9a}$$

$$\operatorname{div}\big(\boldsymbol{\sigma}_i \operatorname{grad}(V_m)\big) + \operatorname{div}\big(\boldsymbol{\sigma}_i \operatorname{grad}(\phi_e)\big) = A_m\Big(C_m \frac{\partial V_m}{\partial t} + I_{\text{ion}}(V_m)\Big). \tag{5.9b}$$

With appropriate boundary conditions, these equations are often used to model cardiac electrophysiology. They also serve as a basis for the fiber models in our multi-scale setting, which will be described in the next section.

5.1.3 Monodomain Model

One approach to modeling skeletal muscle electrophysiology is to explicitly resolve muscle fibers and compute propagating action potentials on these spatial domains. Propagation of action potentials can be described by the monodomain equation, which is a specialization of the bidomain equations for a one-dimensional intracellular space.

We assume a muscle domain $\Omega_M \subset \mathbb{R}^3$ with a number of embedded 1D manifolds $\Omega_f^j \subset \mathbb{R}^3$ for $j = 1, \dots, n$ that represent muscle fibers. The domain Ω_M represents the extracellular space and each fiber domain Ω_f^j represents a separate intracellular space. It is further assumed that electric conduction in the extracellular space is directed equally to the embedded fibers. This can be stated as

$$\boldsymbol{\sigma}_i = k \cdot \boldsymbol{\sigma}_e. \tag{5.10}$$

The intracellular conductivity tensor $\boldsymbol{\sigma}_i$ (here prolonged from the scalar value σ_i on a fiber with tangent $\mathbf{a} \in \mathbb{R}^3$ to the 3D domain by $\boldsymbol{\sigma}_i = \sigma_i\,\mathbf{a} \otimes \mathbf{a}$) and the extracellular conductivity $\boldsymbol{\sigma}_e$ are multiples of each other with a scaling factor $k \in \mathbb{R}$.

Plugging Eq. (5.10) into the first bidomain equation, Eq. (5.9a), and restricting the domain to a 1D fiber Ω_f^j allows to combine the terms related to ϕ_e:

$$\operatorname{div}\big(\sigma_i \operatorname{grad}(\phi_e)\big) = -\frac{k}{k+1}\operatorname{div}\big(\sigma_i \operatorname{grad}(V_m)\big) \quad \text{on } \Omega_f^j.$$

Using the second bidomain equation, Eq. (5.9b), we get the expression

$$\operatorname{div}\left(\sigma_{\text{eff}} \operatorname{grad}(V_m)\right) = A_m\left(C_m \frac{\partial V_m}{\partial t} + I_{\text{ion}}(V_m, \mathbf{y})\right) \quad \text{on } \Omega_f^j.$$

The effective conductivity σ_{eff} combines the intra and extracellular conductivities, σ_i and σ_e, analog to a parallel circuit:

$$\sigma_{\text{eff}} := \sigma_i \parallel \sigma_e = \frac{\sigma_i \sigma_e}{\sigma_i + \sigma_e}.$$

Rearranging the terms yields the classical form of the monodomain equation:

$$\frac{\partial V_m}{\partial t} = \frac{1}{A_m C_m}\left(\sigma_{\text{eff}} \frac{\partial^2 V_m}{\partial x^2} - A_m I_{\text{ion}}(V_m, \mathbf{y})\right) \qquad \text{for } x \in \Omega_f^j. \tag{5.11}$$

The multi-scale framework uses multiple instances of the monodomain equation Eq. (5.11) together with the first bidomain equation Eq. (5.9a) to model electrophysiology in the fibers and the extracellular domain [Mor15]. In addition to the fiber domains Ω_f^j, two instances of the muscle domain Ω_M are needed for the bidomain equation, one for the intracellular and one for the extracellular space. The transmembrane potential V_m is unidirectionally coupled from the fiber meshes to the intracellular space of the first bidomain equation. The extracellular potential ϕ_e corresponds to the signals that are measured during intramuscular EMG recording.

Within the multi-scale framework, it is also possible to couple a model for electric conduction in an additional layer of body fat tissue. This is subsequently described in Sec. 5.1.5. Then, electric current fluxes between the muscle and body fat domains have to be modeled.

If no such additions should be made to the model, the following Neumann boundary conditions are used to close the description:

$$\frac{\partial V_m}{\partial x} = 0 \qquad \text{on } \partial\Omega_f^j, \tag{5.12a}$$

$$\left(\boldsymbol{\sigma}_i \operatorname{grad}(V_m)\right) \cdot \mathbf{n}_m = -\left(\boldsymbol{\sigma}_i \operatorname{grad}(\phi_e)\right) \cdot \mathbf{n}_m \qquad \text{on } \partial\Omega_M, \tag{5.12b}$$

$$\left(\boldsymbol{\sigma}_e \operatorname{grad}(\phi_e)\right) \cdot \mathbf{n}_m = 0 \qquad \text{on } \partial\Omega_M, \tag{5.12c}$$

with the outward normal vector $\mathbf{n}_m$. Equation (5.12a) defines homogeneous Neumann boundary conditions for the monodomain equation Eq. (5.11) at the two ends of each 1D

muscle fiber domain. The boundary conditions on $\partial\Omega_M$ are related to the bidomain equations given in Eqs. (5.9a) and (5.9b). Equation (5.12b) is equivalent to a homogeneous Neumann boundary condition on the intracellular current density j_i (cf. Eq. (5.7)) and is expressed in terms of the transmembrane voltage V_m and the extracellular potential ϕ_e. Another homogeneous Neumann boundary condition on ϕ_e as given by Eq. (5.12c) is required.

5.1.4 Multidomain Model

The multidomain model is an alternative approach to the description based on the monodomain and bidomain equations described in Sections 5.1.2 and 5.1.3. It was proposed in [Klo20] and describes the same physics. However, the muscle fibers are homogenized and all equations are formulated using a single 3D muscle domain Ω_M.

The multidomain equations generalize the two bidomain equations and allow taking into account multiple MUs by defining a separate intracellular space for each MU. Thus, at every spatial point $\mathbf{x} \in \Omega_M$ one extracellular and N_{MU} intracellular domains or compartments coexist, where N_{MU} is the number of MUs. As before, the extracellular domain has the electric potential ϕ_e and conductivity tensor $\boldsymbol{\sigma}_e$. For each compartment $k = 1, \dots, N_{\text{MU}}$, a separate electric potential ϕ_i^k, transmembrane voltage $V_m^k = \phi_i^k - \phi_e$, conductivity tensor $\boldsymbol{\sigma}_i^k$, surface to volume ratio of the membrane A_m^k and membrane capacitance C_m^k are defined.

Analog to the fibers of a MU that exhibit different densities at different locations in the muscle, each compartment occupies different locations within the domain to a different extent. This is described by the relative occupancy factor $f_r^k : \Omega_M \to [0,1]$ for MU k. The factors have different values in the domain according to the presence of the MU at the respective location. At every point, their sum is one, $\sum_{k=1}^{N_{\text{MU}}} f_r^k = 1$, if all MUs should be considered or less than one if the effect of remainder MUs that will not be activated in the simulation scenario is neglected.

The first multidomain equation is similar to the first bidomain equation Eq. (5.9a) and balances the current flow between the extracellular space and the weighted sum of all intracellular spaces:

$$\operatorname{div}\big(\boldsymbol{\sigma}_e \operatorname{grad}(\phi_e)\big) + \sum_{k=1}^{N_{\text{MU}}} f_r^k \operatorname{div}\big(\boldsymbol{\sigma}_i^k \operatorname{grad}(V_m^k + \phi_e)\big) = 0. \tag{5.13}$$

By defining a total intracellular conductivity tensor $\boldsymbol{\sigma}_i = \sum_{k=1}^{N_{\text{MU}}} f_r^k \, \boldsymbol{\sigma}_i^k$, Eq. (5.13) can be restated as

$$\operatorname{div}\big((\boldsymbol{\sigma}_e + \boldsymbol{\sigma}_i)\operatorname{grad}(\phi_e)\big) + \sum_{k=1}^{N_{\text{MU}}} f_r^k \operatorname{div}\big(\boldsymbol{\sigma}_i^k \operatorname{grad}(V_m^k)\big) = 0. \tag{5.14}$$

The second multidomain equation equals the second bidomain equation Eq. (5.9b). It describes the current over the membrane and holds for every compartment:

$$\operatorname{div}(\boldsymbol{\sigma}_i^k \operatorname{grad}(V_m^k + \phi_e)) = A_m^k \left(C_m^k \frac{\partial V_m^k}{\partial t} + I_{\text{ion}}(V_m^k) \right) \qquad \forall k \in \{1, \dots, N_{\text{MU}}\}.$$

It is convenient to rearrange it for $\partial V_m^k / \partial t$:

$$\frac{\partial V_m^k}{\partial t} = \frac{1}{A_m^k \, C_m^k} \left(\operatorname{div}\big(\boldsymbol{\sigma}_i^k \operatorname{grad}(V_m^k + \phi_e)\big) - A_m^k I_{\text{ion}}(V_m^k) \right) \quad \forall k \in \{1, \dots, N_{\text{MU}}\}. \tag{5.15}$$

The current I_{ion} over the membrane is again computed by the subcellular model given by Eq. (5.3b).

The resulting system of Eqs. (5.14) and (5.15) constitutes the first and second multidomain equations and can be used to compute muscle electrophysiology. The boundary conditions are defined analogously to Eqs. (5.12b) and (5.12c):

$$\big(\boldsymbol{\sigma}_i^k \operatorname{grad}(V_m^k)\big) \cdot \mathbf{n}_m = -\big(\boldsymbol{\sigma}_i^k \operatorname{grad}(\phi_e)\big) \cdot \mathbf{n}_m \qquad \text{on } \partial\Omega_M \qquad \forall k \in \{1, \dots, N_{\text{MU}}\}, \tag{5.16a}$$

$$\big(\boldsymbol{\sigma}_e \operatorname{grad}(\phi_e)\big) \cdot \mathbf{n}_m = 0, \qquad \text{on } \partial\Omega_M \tag{5.16b}$$

where $\mathbf{n}_m$ is the outward normal vector on $\partial\Omega_M$.

5.1.5 Electric Conduction in the Body Domain

Surface EMG signals are the result of electric conduction in the electrically active muscle tissue as well as in surrounding inactive tissue such as adipose tissue and skin or connective tissue such as tendons and ligaments. This surrounding tissue is summarized by the body domain Ω_B, which partly shares its boundary with the muscle domain Ω_M.

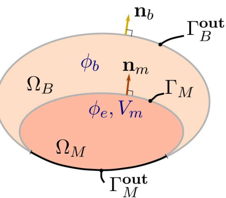


Figure 5.3: Computational domains for the simulation of surface EMG. The body domain Ω_B and the muscle domain Ω_M share a part of their boundary, Γ_M, which has a normal vector $\mathbf{n}_m$. The outer boundary is composed of Γ_B^{out} and Γ_M^{out} and has the outward normal vector $\mathbf{n}_b$.

Figure 5.3 visualizes these domains and defines their names: The domains Ω_M and Ω_B have outward normals $\mathbf{n}_m$ and $\mathbf{n}_b$, the outer boundary is composed of Γ_B^{out} and Γ_M^{out} and the variables ϕ_e, V_m and ϕ_b are defined as shown within the domains Ω_M and Ω_B.

The work of [Mor15] proposes an isotropic conductivity $\boldsymbol{\sigma}_b$ and a harmonic electric potential ϕ_b in the body domain Ω_B:

$$\operatorname{div}\big(\boldsymbol{\sigma}_b \operatorname{grad}(\phi_b)\big) = 0 \qquad \text{on } \Omega_B. \tag{5.17}$$

The electric potentials ϕ_e and ϕ_b of the neighboring domains Ω_M and Ω_B as well as the current densities are continuous on the shared boundary Γ_M. This is described by the following two coupling conditions:

$$\phi_e = \phi_b \qquad \text{on } \Gamma_M, \tag{5.18a}$$

$$\big(\boldsymbol{\sigma}_e \operatorname{grad}(\phi_e)\big) \cdot \mathbf{n}_m = \big(\boldsymbol{\sigma}_b \operatorname{grad}(\phi_b)\big) \cdot \mathbf{n}_m \qquad \text{on } \Gamma_M. \tag{5.18b}$$

On the outer boundary Γ_B^{out}, homogeneous Neumann boundary conditions are assumed:

$$\big(\boldsymbol{\sigma}_b \operatorname{grad}(\phi_b)\big) \cdot \mathbf{n}_b = 0 \qquad \text{on } \Gamma_B^{\text{out}}. \tag{5.19}$$

The description of the body domain has to be combined either with the fiber based description in Sec. 5.1.3 or the multi-domain description in Sec. 5.1.4. In the literature, this combination was mathematically described for the fiber based model in [Mor15] and for the multidomain model in [Klo20]. Correspondingly, additional boundary conditions either given by Eq. (5.12) or Eq. (5.16) are assumed: For the fiber based description,

which uses the bidomain equation for volume conduction, the boundary conditions are:

$$\big(\boldsymbol{\sigma}_i \,\mathrm{grad}\,(V_m)\big) \cdot \mathbf{n}_m = -\big(\boldsymbol{\sigma}_i \,\mathrm{grad}\,(\phi_e)\big) \cdot \mathbf{n}_m \qquad \text{on } \partial\Omega_M = \Gamma_M \cup \Gamma_M^{\text{out}}, \tag{5.20a}$$

$$\big(\boldsymbol{\sigma}_e \,\mathrm{grad}\,(\phi_e)\big) \cdot \mathbf{n}_m = 0 \qquad \text{on } \partial\Gamma_M^{\text{out}}. \tag{5.20b}$$

For the multidomain description with fat layer, the boundary conditions are:

$$\big(\boldsymbol{\sigma}_i^k \,\mathrm{grad}\,(V_m^k)\big) \cdot \mathbf{n}_m = -\big(\boldsymbol{\sigma}_i^k \,\mathrm{grad}\,(\phi_e)\big) \cdot \mathbf{n}_m \qquad \text{on } \partial\Omega_M = \Gamma_M \cup \Gamma_M^{\text{out}}, \tag{5.21a}$$

$$\big(\boldsymbol{\sigma}_e \,\mathrm{grad}\,(\phi_e)\big) \cdot \mathbf{n}_m = 0 \qquad \text{on } \partial\Gamma_M^{\text{out}}. \tag{5.21b}$$

The first condition in Eq. (5.21a) is enforced for all compartments $k = 1, \dots, N_{\text{MU}}$.

5.2 Model of Muscle Contraction

Muscle contraction is described on the organ level by a solid mechanics model. The goal is to describe the deformation of the tissue caused by the internal forces that are generated by sarcomeres and as a response to outer constraints such as applied forces, the attachment to tendons and inertia effects.

Different modeling approaches exist to describe the mechanical muscle behavior. Dynamic *finite elasticity* methods for large strains exist that use hyperelastic materials, both compressible and incompressible. Further, *linear elasticity* descriptions with linearizations at various levels are used in appropriate applications where small strains can be assumed. The whole range from simplifying linearized models to accurate nonlinear approaches can be found in the literature, sometimes with varying conventions and symbols. In this section, we introduce consistent notation and formulate the model equations for both approaches. The discretization and solution is discussed later in Sec. 5.4.

The derivation largely follows the book of Holzapfel [Hol00] and the discretization follows the work of Zienkiewicz, Taylor et al. [Zie77; Zie05]. Further details can be found also in the book of Marsden and Hughes [Mar94].

5.2.1 Geometric Description

We begin with the geometric description of the material body and define the basic quantities that are subsequently used to describe the physics. We consider the 3D muscle domain $\Omega_0 = \Omega_M \subset \mathbb{R}^3$ in reference configuration at time $t = 0$ that deforms into a current configuration Ω_t at time t. The material points are given by $\mathbf{X} \in \Omega_0$. The corresponding points $\mathbf{x} \in \Omega_t$ in the current configuration are defined by the function $\mathbf{x} = \boldsymbol{\varphi}_t(\mathbf{X})$.

In the following, capital letters refer to quantities in material or Lagrangian description, i.e., defined in the reference configuration and small letters refer to quantities in spatial or Eulerian description, i.e., defined in the current configuration.

The relation of point coordinates in the current configuration with respect to the reference configuration can also be described by the displacements field $\mathbf{U}$:

$$\mathbf{x}(\mathbf{X}) = \mathbf{X} + \mathbf{U}(\mathbf{X}).$$

The symbol $\mathbf{u}$ with $\mathbf{u}(\mathbf{x}(\mathbf{X})) = \mathbf{U}(\mathbf{X})$ denotes the displacements formulated in current configuration. The current velocity $\mathbf{v}$ is the time derivative of the displacements, $\mathbf{v} := \dot{\mathbf{u}}$.

The deformation gradient $\mathbf{F}$ is the second order tensor that is obtained by differentiating the function $\boldsymbol{\varphi}_t$. It is given using the unit vectors $\mathbf{e}_i$ and components F_{aA}:

$$\mathbf{F} = F_{aA}\,\mathbf{e}_a \otimes \mathbf{e}_A, \qquad F_{aA} = \frac{\partial x_a}{\partial X_A}.$$

Capital and small indices refer to reference and current configuration, respectively. The deformation gradient can also be expressed using the displacement field $\mathbf{U}$:

$$\mathbf{F} = \mathbf{I} + \nabla \mathbf{U}. \tag{5.22}$$

Here and in the following, the gradient symbol ∇ refers to differentiation with respect to material coordinates $\mathbf{X}$. We assume Cartesian coordinates.

The determinant of the deformation gradient is $J := \det \mathbf{F} > 0$. It is positive for any physically valid transformation. The deformation gradient is used to map geometric

quantities from the reference to the current configuration:

$$\mathbf{t} = \mathbf{F}\,\mathbf{T}, \qquad \text{(tangent map)} \tag{5.23a}$$

$$\mathbf{a} = \operatorname{cof}(\mathbf{F})\,\mathbf{A}, \qquad \text{(normal map)} \tag{5.23b}$$

$$v = J\,V. \qquad \text{(volume map)} \tag{5.23c}$$

As given in Eq. (5.23a) and visualized in Fig. 5.4, the tensor $\mathbf{F}$ maps material tangents $\mathbf{T}$ in Ω_0 to the corresponding spatial line elements $\mathbf{t}$ in Ω_t. Accordingly, the spatial stretch at a point $\mathbf{x} \in \Omega_t$ in a certain direction is given by $\lambda = \sqrt{\boldsymbol{\lambda}^\top \boldsymbol{\lambda}}$ with $\boldsymbol{\lambda} = \mathbf{F}\mathbf{M}$, where $\mathbf{M}$ is a material line element with unit length pointing in the respective Lagrangian direction.

In Eq. (5.23b), the cofactor of $\mathbf{F}$ given by $\operatorname{cof}(\mathbf{F}) = J\,\mathbf{F}^{-\top}$ maps normals $\mathbf{A}$ and surface areas $|\mathbf{A}|$ from Ω_0 to the corresponding values $\mathbf{a}$ and $|\mathbf{a}|$ in Ω_t. Nanson's formula, $\mathrm{d}\mathbf{a} = \operatorname{cof}(\mathbf{F})\,\mathrm{d}\mathbf{A}$, is used to transform surface integrals from Eulerian to Lagrangian description. Note that tangents at a point $\mathbf{X}$ live in the tangent space $T_{\mathbf{X}}\Omega_0$ and normals live in the co-tangent space $T^*_{\mathbf{X}}\Omega_0$.

Equation (5.23c) describes the volume map from Ω_0 to Ω_t, which simply scales the reference volume V by the determinant J to obtain the volume v in the current configuration.

Furthermore, the deformation gradient $\mathbf{F}$ is used to define the right Cauchy Green tensor $\mathbf{C} = \mathbf{F}^\top\mathbf{F}$, which maps from tangent to co-tangent space in reference configuration, and subsequently the Green-Lagrange strain tensor:

$$\mathbf{E} = \frac{1}{2}(\mathbf{C} - \mathbf{I}).$$

This strain measure can be interpreted as comparing the current Lagrangian metric $\mathbf{C}$, a measure for the symmetric part of the current deformation, with the reference metric which is the identity. Using Eq. (5.22), the Green-Lagrange strain tensor can be formulated in terms of derivatives of the displacements:

$$\mathbf{E} = \frac{1}{2}\Big((\nabla\mathbf{U})^\top + \nabla\mathbf{U} + \nabla\mathbf{U}^\top\nabla\mathbf{U}\Big). \tag{5.24}$$

In case of small displacements, a simplification is to not distinguish between reference and current configuration. The strain expression given in Eq. (5.24) can be linearized by neglecting products of the derivatives and using the spatial displacements $\mathbf{u}$ instead of $\mathbf{U}$.

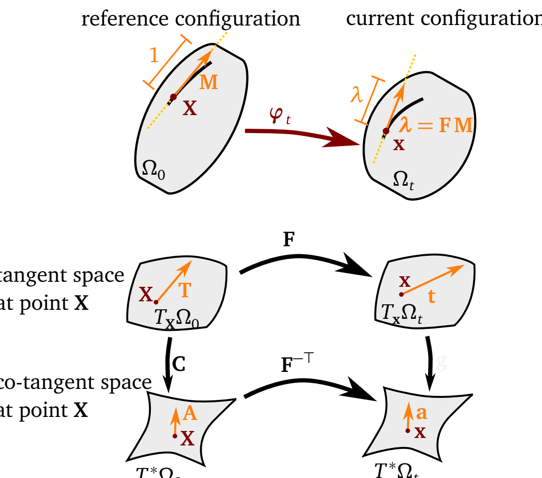


Figure 5.4: Vector spaces and variables used in the geometric description of the solid mechanics model. The left side shows the reference configuration with tangent and co-tangent space of point $\mathbf{X}$. The right side shows tangent and co-tangent space for the current domain and a point $\mathbf{x}$. The spatial stretch λ is defined by mapping a material element $\mathbf{M}$ to the current configuration. The maps φ_t, $\mathbf{F}$ and $\mathbf{F}^{-\top}$ map tangents $\mathbf{T}, \mathbf{t}$ and normals $\mathbf{A}, \mathbf{a}$ between the configurations.

As a result, the linearized strain tensor ε is given by:

$$\boldsymbol{\varepsilon} = \frac{1}{2}\left((\nabla \mathbf{u})^\top + \nabla \mathbf{u}\right). \tag{5.25}$$

It can be used together with linear material models to derive a completely linear model.

5.2.2 Stress Metrics

Continuum mechanical models establish equations for the unknown displacement function $\mathbf{u}$ and its evolution in time via relations between stresses and strains. In the following, we introduce the required stress metrics.

The Cauchy stress tensor $\boldsymbol{\sigma}$ results from Euler's cut principle: we consider the mechanical action on an arbitrary, virtual cut out of the body in current configuration. The contact forces on the cut surface at a point $\mathbf{x}$ are described by the traction force $\mathbf{t}$. The traction vector acts on the current configuration and is a function of the position $\mathbf{x} \in \Omega_t$ and the

local orientation of the cut given by the normal vector $\mathbf{n}$. Cauchy's theorem states that this relation is linear and can be described by the second order Cauchy stress tensor $\boldsymbol{\sigma}$:

$$\mathbf{t} = \boldsymbol{\sigma} \cdot \mathbf{n}. \tag{5.26}$$

Thus, the Cauchy stress describes the "true stress" of contact forces per deformed surface area. Both slots of the second order tensor are associated with the current configuration. More specifically, $\boldsymbol{\sigma}$ is contravariant and maps from a normal $\mathbf{n}$ in co-tangent space $T^*_{\mathbf{x}}\Omega_t$ to the traction $\mathbf{t}$ in tangent space $T_{\mathbf{x}}\Omega_t$.

While the physical description is natural in this Eulerian setting, the numerical treatment is more convenient in the Lagrangian setting, where we can integrate over a non-deforming domain. Moreover, a two-point setting, where surface areas are measured in the undeformed configuration and traction forces are measured in the deformed configuration, is often useful in engineering. This is the natural setting, e.g., in tension tests. Therefore, other stress measures involving the reference configuration are defined.

Using the mappings presented in Eq. (5.23), all quantities can be transformed between both configurations. The physical derivation can be carried out equivalently in a Lagrangian or Eulerian setting and switching between them is possible at any point in the derivation. For this purpose, two operations are defined: the pull-back $\varphi^*(\mathbf{a}) = \mathbf{F}^\top \mathbf{a}\,\mathbf{F}$ and push-forward operations $\varphi_*(\mathbf{A}) = \mathbf{F}^{-\top}\mathbf{A}\,\mathbf{F}^{-1}$, which bring tensors from Eulerian to Lagrangian description and vice-versa.

The first Piola-Kirchhoff stress tensor $\mathbf{P}$ measures contact forces in the current configuration with regard to the area of the reference configuration and relates to the Cauchy stress as $\mathbf{P} = \boldsymbol{\sigma}\,\mathrm{cof}(\mathbf{F})$. The second Piola-Kirchhoff tensor $\mathbf{S}$ is a fully Lagrangian field given as the pull-back of the Cauchy stress scaled by J:

$$\mathbf{S} = \varphi^*(J\,\boldsymbol{\sigma}) = J\,\mathbf{F}^{-1}\boldsymbol{\sigma}\,\mathbf{F}^{-\top}.$$

Figure 5.5 summarizes the geometric maps by black arrows and the stress measures by red arrows. The Lagrangian setting defines the right Cauchy-Green tensor $\mathbf{C}$ and the second Piola-Kirchhoff stress tensor $\mathbf{S}$. We use these quantities in the derivation of the discretized equations, because the Lagrangian formulation is natural for this task and allows integrating over the non-deforming domain Ω_0.

The Cauchy stress $\boldsymbol{\sigma}$ is completely defined in the Eulerian setting. It is used to formulate physical balance principles.

Figure 5.5: Stress tensors and geometric maps that can be used together in a solid mechanics formulation. The right Cauchy-Green tensor **C** and the second Piola-Kirchhoff stress **S** are dual Eulerian tensors and map between tangent space $T_{\mathbf{X}}\Omega_0$ and co-tangent space $T^*_{\mathbf{X}}\Omega_0$ in the reference domain. The deformation gradient **F** and the first Piola-Kirchhoff stress **P** are dual two-point tensors mapping from the reference to the current configuration. The Cauchy stress $\boldsymbol{\sigma}$ is defined entirely in the Eulerian setting.

5.2.3 Overview of the Physical Relations

The previously introduced quantities are linked together by various relations, which are summarized in the diagram in Fig. 5.6. The goal is to find the relationship between given forces (top left in Fig. 5.6) and the resulting deformation of the body described by the displacements (top right in Fig. 5.6). Prescribed external traction forces **T** and external or inertial body forces **B** act on the body and result in stresses **S** satisfying the *equilibrium* relation. A *material law* connects stresses **S** and strains **E**. The *kinematics* of the body determine the relationship between displacements **u** and strains **E**. Geometric Dirichlet boundary conditions prescribe displacements and Neumann boundary conditions such as traction forces contribute to the stress field.

Whereas the equilibrium relation is linear, the material and kinematic descriptions can both be chosen to be linear or nonlinear. In cases of small strains, geometric and material linearity can be assumed. We derive two such formulations: a linear formulation where all relations are linear and a nonlinear formulation with nonlinear material and kinematic relations.

5.2.4 Assumptions and Model Equations

The foundation of continuum mechanics usually builds on three balance principles: conservation of mass, of momentum and of angular momentum. In the following, these principles are presented in their Eulerian forms.

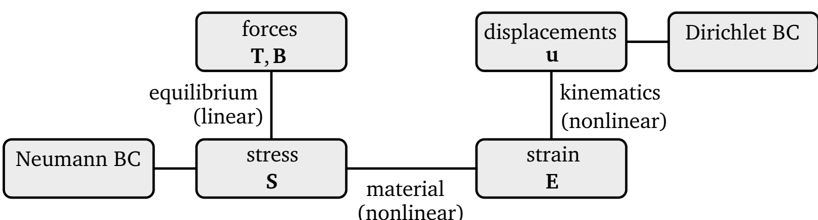


Figure 5.6: The three relations between various quantities that compose the solid mechanics model: Equilibrium links traction and body forces $\mathbf{T}$ and $\mathbf{B}$ to the stresses $\mathbf{S}$. A material model connects them to strains $\mathbf{E}$. The kinematic relations yield the resulting displacement field $\mathbf{u}$. Note that all quantities in this diagram are given in Lagrangian formulation.

First, we assume *conservation of mass* in terms of the densities $\rho_0(\mathbf{X})$ and $\rho(\mathbf{x})$ in reference and current configurations:

$$\int_{V_0} \rho_0 \, \mathrm{d}V = \int_{V_t} \rho \, \mathrm{d}v.$$

The equation holds for all corresponding subdomains $V_0 \subset \Omega_0$ and $V_t \subset \Omega_t$. With the intermediate step of deducing $\mathrm{d}/\mathrm{d}t \int_{\Omega_t} \rho \, \mathrm{d}v = 0$, we get the following differential equation:

$$\dot{\rho}(\mathbf{v}, t) + \rho(\mathbf{x}, t) \operatorname{div}\big(\mathbf{v}(\mathbf{x}, t)\big) = 0. \tag{5.27}$$

As muscle tissue largely consists of water, it is typically assumed to be an incompressible domain. This is equivalent to a constant density, $\dot{\rho} = 0$, and, thus, Eq. (5.27) reduces to

$$\operatorname{div}(\mathbf{v}(\mathbf{x}, t)) = 0. \tag{5.28}$$

The second assumption is the *balance of momentum*, which is expressed as

$$\frac{\mathrm{d}}{\mathrm{d}t} \int_{V_t} \rho \, \mathbf{v} \, \mathrm{d}v = \int_{V_t} \rho \, \mathbf{b} \, \mathrm{d}v + \int_{\partial V_t} \mathbf{t} \, \mathrm{d}a. \tag{5.29}$$

Here, $\mathbf{b}$ describes a body force and $\mathbf{t}$ describes a traction force that acts on the surface of the domain V_t. Using the Cauchy theorem Eq. (5.26), it can be replaced by the Cauchy stress

$\boldsymbol{\sigma}$. The corresponding differential form is given by the following differential equation:

$$\rho\,\dot{\mathbf{v}}(\mathbf{x},t) = \rho\,\mathbf{b}(\mathbf{x},t) + \operatorname{div}\boldsymbol{\sigma}(\mathbf{x},t). \tag{5.30}$$

It relates external forces to the internal stress field and describes the *equilibrium* relation in Fig. 5.6 in Eulerian form. For the discretization, a Lagrangian form is typically used.

For hyperelastic materials, which we consider in the muscle model, the equilibrium relation can also be formulated in terms of the *Hellinger-Reissner energy functional* $\Pi_L(\mathbf{u},p)$, which describes the potential energy of the system depending on the displacement and pressure functions $\mathbf{u}$ and p. Analog to the local form in Eq. (5.30), it contains terms for the external loads and for the internal response of the body. The functional is additively composed of external and internal potential energy:

$$\Pi_L(\mathbf{u},p) = \Pi_{\text{ext}}(\mathbf{u}) + \Pi_{\text{int}}(\mathbf{u},p). \tag{5.31}$$

The external energy functional is formulated by

$$\Pi_{\text{ext}}(\mathbf{u}) = -\int_{\Omega_0} \mathbf{B}\,\mathbf{u}\,\mathrm{d}V - \int_{\partial\Omega_0^t} \bar{\mathbf{T}}\,\mathbf{u}\,\mathrm{d}S, \tag{5.32}$$

with body force $\mathbf{B}$ in reference configuration and prescribed surface traction $\bar{\mathbf{T}}$ on the traction boundary $\partial\Omega_0^t$. The body force term $\mathbf{B}$ also includes the inertial forces of mass density times acceleration, $\rho\,\dot{\mathbf{v}}$, in case of a dynamic model. The internal energy functional $\Pi_{\text{int}}(\mathbf{u},p)$ describes the strain energy of the system depending on the displacement field $\mathbf{u}$ and the hydrostatic pressure p. The term is defined in Sec. 5.4.2.

The *principle of stationary potential energy* demands that the potential energy functional Π_L is stationary. Variational calculus and differentiation of Eq. (5.31) lead to the local Eulerian description given in Eqs. (5.29) and (5.30).

The third assumption is the *balance of angular momentum* and can be formulated using the 3D cross-product:

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_{V_t} \mathbf{x}\times(\rho\,\mathbf{v})\,\mathrm{d}v = \int_{V_t} \mathbf{x}\times(\rho\,\mathbf{b})\,\mathrm{d}v + \int_{\partial V_t} \mathbf{x}\times\mathbf{t}\,\mathrm{d}a.$$

This can be shown to be equivalent to the symmetry of the Cauchy stress tensor, $\boldsymbol{\sigma} = \boldsymbol{\sigma}^\top$.

A further assumption in the multi-scale muscle framework is to only consider isothermal conditions. An activated muscle performs work and energy is added to the system by

metabolism. Further, the muscle is not thermodynamically isolated. The system is not closed regarding conversion and transfer of energy and, thus, the balance of energy cannot be modeled easily.

Regarding the required relations to obtain the deformation of the body from external loads given in Fig. 5.6, the *equilibrium* relation is given by Eq. (5.30) and the nonlinear *kinematic* relation is given by Eq. (5.24). The *material* relation has yet to be defined. The mathematical description is closed by defining a constitutive relation between stresses and strains in the next sections.

Section 5.2.5 defines a linear material model that can be used together with linearized kinematics to formulate a fully linear model. Section 5.2.7 presents the nonlinear material model to proceed with the fully nonlinear description.

5.2.5 Linear Material Model

For a linear constitutive relation between strain and stress, the linearized strain tensor $\boldsymbol{\varepsilon}$, defined in Eq. (5.25) is used together with the Eulerian Cauchy stress $\boldsymbol{\sigma}$. The generic linear material model is *Hooke*'s law, given by

$$\boldsymbol{\sigma} = \mathbb{C} : \boldsymbol{\varepsilon} \tag{5.33}$$

with the fourth order material tensor

$$\mathbb{C}_{abcd} = K\,\delta_{ab}\,\delta_{cd} + \mu\,(\delta_{ac}\,\delta_{bd} + \delta_{ad}\,\delta_{bc} - \frac{2}{3}\,\delta_{ab}\,\delta_{cd}).$$

The bulk modulus K is a measure for the (in-)compressibility and the shear modulus μ specifies the elastic shear stiffness. δ_{ab} is the Kronecker delta. The material tensor $\mathbb{C}$ exhibits the following major and minor symmetries:

$$\mathbb{C}_{abcd} = \mathbb{C}_{cdab}, \qquad \text{(major symmetries)} \tag{5.34a}$$

$$\mathbb{C}_{abcd} = \mathbb{C}_{bacd} = \mathbb{C}_{abdc} = \mathbb{C}_{badc}, \qquad \text{(minor symmetries)} \tag{5.34b}$$

effectively reducing the number of independent entries from 81 to 21 for 3D domains.

To incorporate force generation in the muscle, the stress can be additively composed

of the passive stress $\boldsymbol{\sigma}$ and an additional active stress term $\boldsymbol{\sigma}_{\text{active}}$:

$$\boldsymbol{\sigma}^{\text{total}} = \boldsymbol{\sigma} + \boldsymbol{\sigma}^{\text{active}}. \tag{5.35}$$

5.2.6 Nonlinear Material Modeling

Next, we present the derivation of a nonlinear model that does not make any linearization assumptions of small strains as in the previous section. We begin with the description of the material law, which links strains and stresses.

The scalar strain energy function Ψ describes the elastic energy of the material depending on the deformation. The definition of Ψ suffices to describe the behavior of a hyperelastic material. The strain energy function links the right Cauchy Green tensor $\mathbf{C}$ to the second Piola-Kirchhoff stress tensor $\mathbf{S}$ by the relation

$$\mathbf{S} = 2\frac{\partial \Psi(\mathbf{C})}{\partial \mathbf{C}}. \tag{5.36}$$

The *principle of material objectivity* requires that material properties are invariant under a change of observer. As a result, the *representation theorem for isotropic materials* states that the stress tensor can be represented using three strain invariants I_1, I_2 and I_3. For a transversely isotropic material, two invariants I_4 and I_5 that depend on the anisotropy direction $\mathbf{a}_0$ (corresponding to a fiber direction) are added. Consequently, we can formulate the strain energy function $\Psi = \Psi(I_1, I_2, I_3, I_4, I_5)$ in terms of these invariants. The principle strain invariants I_1 to I_3 of the right Cauchy-Green tensor $\mathbf{C}$ and the additional anisotropic invariants I_4 and I_5 are defined as:

$$I_1(\mathbf{C}) = \operatorname{tr}(\mathbf{C}), \qquad I_2(\mathbf{C}) = \frac{1}{2}\big(\operatorname{tr}(\mathbf{C})^2 - \operatorname{tr}(\mathbf{C}^2)\big), \qquad I_3(\mathbf{C}) = \det(\mathbf{C}) = J^2,$$

$$I_4(\mathbf{C}, \mathbf{a}_0) = \mathbf{a}_0 \cdot \mathbf{C}\,\mathbf{a}_0, \qquad I_5(\mathbf{C}, \mathbf{a}_0) = \mathbf{a}_0 \cdot \mathbf{C}^2\,\mathbf{a}_0.$$

The fiber stretch is related to the fourth invariant by $\lambda_f = \sqrt{I_4}$. Note that requiring incompressibility is equivalent to enforcing $J = 1$, and, in this case, we get $I_3(\mathbf{C}) = 1$.

It is convenient to use a decoupled description, where the deformation gradient $\mathbf{F}$ and the right Cauchy-Green tensor $\mathbf{C}$ are multiplicatively decomposed into volume-changing

(volumetric) and volume-preserving (isochoric) parts:

$$\mathbf{F} = (J^{1/3}\mathbf{I})\,\bar{\mathbf{F}}, \qquad \mathbf{C} = (J^{2/3}\mathbf{I})\,\bar{\mathbf{C}}.$$

Here, the volumetric parts are the identity tensors scaled by a power of the determinant J of the deformation gradient. The isochoric or distortional parts $\bar{\mathbf{F}}$ and $\bar{\mathbf{C}}$ are given by

$$\bar{\mathbf{F}} = J^{-1/3}\,\mathbf{F}, \qquad \bar{\mathbf{C}} = J^{-2/3}\,\mathbf{C}. \tag{5.37}$$

The reduced invariants $\bar{I}_1$ to $\bar{I}_5$ of the reduced right Cauchy-Green tensor $\bar{\mathbf{C}}$ are defined accordingly. Similarly, the strain energy function has a decoupled representation with volumetric part Ψ_{vol} and isochoric part Ψ_{iso}:

$$\Psi = \Psi_{\text{vol}}(J) + \Psi_{\text{iso}}(\bar{\mathbf{C}}) = \Psi_{\text{vol}}(J) + \Psi_{\text{iso}}(\bar{I}_1, \bar{I}_2, \bar{I}_4, \bar{I}_5). \tag{5.38}$$

Using the decoupled form, any incompressible material can be modeled with the *penalty method* as follows. The material behavior is given by the isochoric strain energy $\Psi_{\text{iso}}(\bar{\mathbf{C}})$, e.g., by employing the Mooney-Rivlin model in Eq. (5.41). The volumetric part is defined as

$$\Psi_{\text{vol}}(J) = \kappa\, G(J) \qquad \text{with } G(J) = \frac{1}{2}(J-1)^2,$$

with the incompressibility parameter κ and the penalty function $G(J)$. This function is strictly convex and approaches zero as J approaches 1. For large values of κ, the behavior is nearly incompressible. A disadvantage of this method is, that the resulting system becomes singular for $J \to 1$.

A better approach in this regard is to use a mixed formulation, where incompressibility is enforced exactly using a Lagrange multiplier. This approach is also implemented in OpenDiHu and is the preferred method for incompressible materials.

In OpenDiHu, the strain energy function of a new material can be given using the following four terms:

$$\Psi = \Psi_{\text{vol}}(J) + \Psi_{\text{iso}}(\bar{I}_1, \bar{I}_2, \bar{I}_4, \bar{I}_5) + \Psi_1(I_1, I_2, I_3) + \Psi_2(\mathbf{C}, \mathbf{a}_0). \tag{5.39}$$

The decoupled form is available with Ψ_{vol} and Ψ_{iso}, the coupled form for isotropic materials can be used via Ψ_1. The term Ψ_2 gives the most flexibility, as the constitutive model can be directly formulated using the right Cauchy-Green tensor $\mathbf{C}$ and the fiber direction

$\mathbf{a}_0$. The unused terms among $\Psi_{\text{vol}}, \Psi_{\text{iso}}, \Psi_1$ and Ψ_2 can be defined as constant zero. The incompressibility constraint using Lagrange multipliers can be switched on or off such that both incompressible and compressible materials can be computed.

5.2.7 The Nonlinear Material Model for Muscle Contraction

In the muscle contraction model of [Hei13], the strain energy function is additively composed of two passive terms, one isotropic, one anisotropic, and one additional active term:

$$\Psi(\mathbf{C}) = \Psi_{\text{isotropic}}(I_1, I_2) + \Psi_{\text{anisotropic}}(\lambda_f) + \Psi_{\text{active}}(\gamma). \tag{5.40}$$

The isotropic term $\Psi_{\text{isotropic}}$ is formulated in terms of the strain invariants $I_1 = \text{tr}(\mathbf{C})$ and $I_2 = \big(\text{tr}(\mathbf{C})^2 - \text{tr}(\mathbf{C}^2)\big)/2$. The anisotropic term $\Psi_{\text{anisotropic}}$ depends on the fiber stretch λ_f. The active term Ψ_{active} yields the active stress that results from muscular activation, which is described by the activation parameter γ.

The passive behavior of muscle tissue is modeled by a transversely isotropic Mooney-Rivlin material. The isotropic part is given by the Mooney-Rivlin formulation:

$$\Psi_{\text{isotropic}}(I_1, I_2) = c_1 (I_1 - 3) + c_2 (I_2 - 3). \tag{5.41}$$

The values of the two material parameters c_1 and c_2 can be determined by compression tests and are summarized in the work of [Hei13].

The anisotropic behavior depends only on the fiber stretch λ_f. The formulation in [Hei13] uses two material parameters b and d and the following function:

$$\Psi_{\text{anisotropic}}(\lambda_f) = \frac{b}{d} (\lambda_f^d - 1) - b \log(\lambda_f).$$

The active contribution is directly formulated in terms of the second Piola-Kirchhoff stress $\mathbf{S}$. The relation between the active stress $\mathbf{S}_{\text{active}}$ and the active contribution Ψ_{active} of the strain energy function as well as the definition of $\mathbf{S}_{\text{active}}$ is given as follows:

$$\mathbf{S}_{\text{active}} = \frac{1}{\lambda_f} \frac{\partial \Psi_{\text{active}}}{\partial \lambda_f} \mathbf{A} \otimes \mathbf{A} = \frac{1}{\lambda_f} \cdot S_{\text{max,active}} \cdot f_\ell(\lambda_f) \cdot \bar{\gamma}\, \mathbf{A} \otimes \mathbf{A}. \tag{5.42}$$

Here, the resulting active stress tensor $\mathbf{S}_{\text{active}}$ is the second order tensor oriented according to the material fiber direction $\mathbf{A} : \Omega_0 \to \mathbb{R}^3$ and given by the dyadic product $\mathbf{A} \otimes \mathbf{A} =$

$A_i A_j \mathbf{e}_i \otimes \mathbf{e}_j$, scaled by the maximum active stress parameter $S_{\text{max,active}}$, a function f_ℓ that models the force-length relation, and the 3D homogenized value $\bar{\gamma}$ of the activation parameter $\gamma \in [0, 1]$ following from the half-sarcomere model.

In the deforming body fat layer, the active stress contribution is disregarded. For simulating tendons, different material models can be used such as the model proposed by Carniel et al. [Car17], which describes microstructural interactions between collagen fibers and their matrix in addition to the elastic response of the fibers themselves. To alter the material model, the definition of Ψ can simply be changed while all other equations of the solid mechanics model remain intact.

5.2.8 Summary of the Solid Mechanics Model Equations

In summary, the model of solid mechanics for muscle contraction is solved for the unknown displacements $\mathbf{u}$ and additionally the velocities $\mathbf{v}$ if a dynamic formulation is considered.

The model equations follow from the following balance principles:

$$\operatorname{div}(\mathbf{v}) = 0, \qquad \text{(incompressibility)} \qquad (5.43a)$$

$$\rho\,\dot{\mathbf{v}} = \rho\,\mathbf{b} + \operatorname{div}\boldsymbol{\sigma}, \qquad \text{(balance of linear momentum)} \qquad (5.43b)$$

$$\boldsymbol{\sigma} = \boldsymbol{\sigma}^\top, \qquad \text{(balance of angular momentum)} \qquad (5.43c)$$

with the constant density ρ, external body forces $\mathbf{b}$ and the Cauchy stress tensor $\boldsymbol{\sigma}$.

Additionally, geometric relations between displacements $\mathbf{u}$ and strains $\mathbf{E}$ or $\boldsymbol{\varepsilon}$ are assumed, either fully nonlinear in Eq. (5.24) or with corresponding linearization assumptions in Eq. (5.25). Furthermore, a material model is given that relates strains and stresses. A linear model is described in Sec. 5.2.5. The framework for nonlinear hyperelastic models uses a strain energy function Ψ as described in Sec. 5.2.6. A particular nonlinear material model for muscle contraction from the literature is described in Sec. 5.2.7.

The description of the multi-scale model [Röh12; Hei13] assumes quasi-static conditions, which means that the velocities are set to zero, $\mathbf{v} = \mathbf{0}$, and inertial terms are neglected. As a consequence, the incompressibility constraint in Eq. (5.43a) has to be formulated differently and the balance of momentum in Eq. (5.43b) reduces to $\rho\,\mathbf{b} + \operatorname{div}\boldsymbol{\sigma} = 0$.

Our implementation extends the model to the fully dynamic formulation given in Equations (5.43a) to (5.43c).

Initial conditions for the displacements $\mathbf{u}$ and velocities $\mathbf{v}$ define the initial pose of the muscle tissue:

$$\mathbf{u}(\mathbf{x},0) = \mathbf{u}_0(\mathbf{x}), \qquad \mathbf{v}(\mathbf{x},0) = \mathbf{v}_0(\mathbf{x}) \qquad \text{for } \mathbf{x} \in \Omega_M.$$

Dirichlet boundary conditions for $\mathbf{u}$ and $\mathbf{v}$ can fix certain parts of the muscle, e.g., at the attachment points of the tendons:

$$\mathbf{u}(\mathbf{x},t) = \bar{\mathbf{u}}(t), \qquad \mathbf{v}(\mathbf{x},t) = \bar{\mathbf{v}}(t) \qquad \text{for } \mathbf{x} \in \partial\Omega_{\text{Dirichlet}}.$$

Additionally, Neumann boundary conditions can be used to prescribe traction forces on the surface.

The derivation of the finite element formulation and the resulting numerical scheme to obtain the solution functions $\mathbf{u}$ and $\mathbf{v}$ are discussed in Sec. 5.4.

5.3 Discretization of the Electrophysiology Models

The partial and ordinary differential equations described in the last section contain spatial and temporal derivatives that have to be discretized to be solved numerically. For temporal derivatives, we use timestepping schemes, for spatial derivatives, we employ the finite element method.

In this section, we describe the discretization of the subcellular and electrophysiology models that were presented in the last section. A description of the discretization of the solid mechanics model follows in Sec. 5.4.

We begin with the discretization in time in Sec. 5.3.1, followed by the spatial discretization for the monodomain (Sections 5.3.2 and 5.3.3) and multidomain models (Sections 5.3.4 and 5.3.5).

5.3.1 Discretization of the Monodomain Model

Electrophysiology models typically consist of a reaction-diffusion equation. The diffusion term describes the electric conduction in the tissue and the reaction term includes the

subcellular model. In our model, the monodomain equation Eq. (5.11) used in the fiber based description and the second multidomain equation Eq. (5.15) are equations of this type.

This type of partial differential equation is often solved using operator splitting schemes. A first order operator splitting scheme is Godunov splitting [Gui03]. It was used for the solution of the chemo-electro-mechanical model in [Röh12]. In addition to Godunov splitting, we also employ the second order accurate Strang splitting scheme [Str68].

In the following, the application of these two schemes is illustrated for the monodomain equation Eq. (5.11). The right-hand sides of the diffusion and reaction terms are denoted in short as $\mathcal{L}_1$ and $\mathcal{L}_2$:

$$\mathcal{L}_1(V_m) := \frac{1}{A_m\,C_m}\,\sigma_{\text{eff}}\frac{\partial^2 V_m}{\partial x^2}, \qquad \mathcal{L}_2(V_m) := -\frac{1}{C_m}\,I_{\text{ion}}(V_m, \mathbf{y}).$$

Then, the monodomain equation takes the form:

$$\frac{\partial V_m}{\partial t} = \mathcal{L}_1(V_m) + \mathcal{L}_2(V_m). \tag{5.44}$$

A timestepping scheme is constructed that starts with a given initial value $V_m^{(0)}$ and computes solution values $V_m^{(i)}$ at discrete points in time $t^{(i)} = i \cdot dt$ with a fixed timestep width dt. Godunov splitting proceeds by alternatingly performing steps in the two directions of the right-hand sides $\mathcal{L}_1$ and $\mathcal{L}_2$. In the first substep per iteration, an intermediate value V_m^* is calculated, which is used as starting point for the second substep. Each of the substeps are performed using independent timestepping scheme, e.g., the explicit Euler scheme:

$$V_m^* = V_m^{(i)} + dt\,\mathcal{L}_1(V_m^{(i)}, t^{(i)}), \tag{5.45a}$$

$$V_m^{(i+1)} = V_m^* + dt\,\mathcal{L}_2(V_m^*, t^{(i)}) \tag{5.45b}$$

Strang splitting uses a similar approach with three substeps per timestep and two

intermediate values V_m^* and V_m^{**}:

$$V_m^* = V_m^{(i)} + \frac{dt}{2}\,\mathcal{L}_1(V_m^{(i)}, t^{(i)}), \tag{5.46a}$$

$$V_m^{**} = V_m^* + dt\,\mathcal{L}_2(V_m^*, t^{(i)}), \tag{5.46b}$$

$$V_m^{(i+1)} = V_m^{**} + \frac{dt}{2}\,\mathcal{L}_1(V_m^{**}, t_{(i)} + \frac{1}{2}\,dt). \tag{5.46c}$$

Note that each substep can either be executed as a single timestep of the chosen method as in Eqs. (5.45) and (5.46) or divided into several steps with timestep widths dt_{0D} (for the 0D subcellular model represented by $\mathcal{L}_1$) and dt_{1D} (for the diffusion equation represented by $\mathcal{L}_2$).

Figure 5.7 visualizes both splitting schemes applied to the monodomain equation. The yellow arrows correspond to the solution of the 0D subcellular model. The red arrows correspond to the solution of the 1D diffusion equation. The timestep width of one splitting step is $dt_{\text{splitting}}$. Depending on how the timestep widths are chosen in relation to each other, different numbers of subcycles are used in the solution of the 0D and 1D problems.

Instead of the explicit Euler method in Eqs. (5.45) and (5.46), other timestepping methods can be used for the substeps. We use the following schemes, which are listed as single steps for the generic ODE $\partial V_m/\partial t = \mathcal{L}(V_m, t)$:

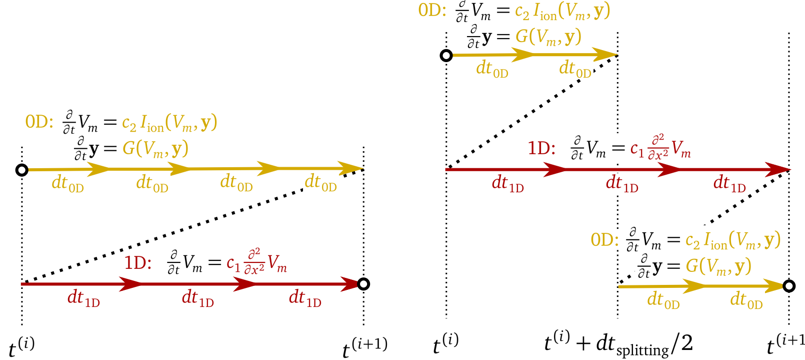
$$V_m^{(i+1)} = V_m^{(i)} + dt\,\mathcal{L}(V_m^{(i)}, t^{(i)}), \tag{5.47a}$$

$$V_m^{(i+1)} = V_m^{(i)} + \frac{dt}{2}\Big(\mathcal{L}(V_m^{(i)}, t^{(i)}) + \mathcal{L}\big(V_m^{(i)} + dt\,\mathcal{L}(V_m^{(i)}, t^{(i)}), t^{(i+1)}\big)\Big), \tag{5.47b}$$

$$V_m^{(i+1)} = V_m^{(i)} + dt\,\mathcal{L}(V_m^{(i+1)}, t^{(i+1)}), \tag{5.47c}$$

$$V_m^{(i+1)} = V_m^{(i)} + \frac{dt}{2}\Big(\theta\,\mathcal{L}(V_m^{(i+1)}, t^{(i+1)}) + (1-\theta)\,\mathcal{L}(V_m^{(i)}, t^{(i)})\Big). \tag{5.47d}$$

Here, Eq. (5.47a) is the first-order accurate explicit Euler scheme, Eq. (5.47b) is the second-order accurate Heun scheme, Eq. (5.47c) is the first order accurate implicit Euler scheme, and Eq. (5.47d) is the Crank-Nicolson scheme [Cra47], which for $\theta = 0$ equals the explicit Euler and for $\theta = 1$ equals the implicit Euler scheme. For $\theta = \frac{1}{2}$, it is second order accurate. An advantage of the implicit schemes in Eqs. (5.47c) and (5.47d) is, that,



(a) The Godunov splitting uses two substeps: 0D, 1D.

(b) The Strang splitting uses three substeps: 0D, 1D, 0D.

Figure 5.7: Godunov and Strang splitting schemes that are used to solve the monodomain equation. The equation is split into a reaction part (0D,yellow) and a diffusion part (1D,red) and these parts are solved alternatingly. The visualizations show one splitting timestep starting at the left circle and completing at the right circle.

for our considered diffusion problems, they are unconditionally stable. A disadvantage is, that a linear equation has to be solved in every timestep.

A second order accurate timestepping scheme yields a faster decrease of the numerical error with decreasing step size and, thus, in many cases allows a larger step size than a first order scheme. To obtain a second order scheme for the monodomain equation, we use Strang splitting (Eq. (5.46)) with the Crank-Nicolson scheme (Eq. (5.47d)) for the diffusion term $\mathcal{L}_1$ and Heun's method (Eq. (5.47b)) for the reaction term $\mathcal{L}_2$. In the subcellular model, the system of ODEs with state vector $\mathbf{y}$ given in Eq. (5.6) is solved with Heun's method along with the equation in terms of V_m.

Next, the spatial derivatives in the diffusion part $\mathcal{L}_2$ of the split equation have to be discretized. Then, both the multidomain and the fiber based models can be solved using the splitting scheme.

5.3.2 Discretization of the Diffusion and Laplace Equations

For the spatial discretization, we first derive the finite element formulation for a generic parabolic diffusion equation in a domain $\Omega \subset \mathbb{R}^d$ of arbitrary dimensionality d. Then,

specialization to 1D yields the formulation for the monodomain equation. Considering a 3D domain, the formulation is an important building block for the discretization of the multidomain model. This is shown in more detail in a later section, Sec. 5.3.4

We consider the following diffusion problem in the variable $u : \Omega \times [0, t_{\text{end}}] \to \mathbb{R}$ with Neumann boundary conditions on a part of the boundary $\Gamma_f \subset \partial\Omega$ with normal vector $\mathbf{n}$:

$$\frac{\partial u}{\partial t} = \operatorname{div}(\boldsymbol{\sigma} \operatorname{grad} u), \quad (\boldsymbol{\sigma} \operatorname{grad} u) \cdot \mathbf{n} = f \quad \text{on } \Gamma_f, \quad (\boldsymbol{\sigma} \operatorname{grad} u) \cdot \mathbf{n} = 0 \quad \text{on } \partial\Omega \backslash \Gamma_f.$$

We discretize the temporal derivative using the Crank-Nicolson scheme as in Eq. (5.47d). Following the procedure of the Galerkin finite element formulation with the Hilbert space $H_0^1(\Omega)$ of test functions ϕ that are zero on the boundary, we arrive at the following weak form:

$$\begin{aligned} &\int_\Omega \big(\theta\, \nabla \cdot (\boldsymbol{\sigma} \nabla \mathbf{u}^{(i+1)}) + (1-\theta)\, \nabla \cdot (\boldsymbol{\sigma} \nabla u^{(i)})\big)\, \phi \, \mathrm{d}\mathbf{x} \\ &\qquad = \frac{1}{dt} \int_\Omega (u^{(i+1)} - u^{(i)})\, \phi \, \mathrm{d}\mathbf{x}, \qquad \forall \phi \in H_0^1(\Omega). \end{aligned}$$

For brevity, we express divergence and gradient using the nabla operator.

To discretize the weak form in space, we choose a function space $V_h = \operatorname{span}\{\varphi_j \mid j = 1, \dots, N\}$ to represent the solution as $u = \sum_{j=1}^N u_j \phi_j$. Applying the divergence theorem, we obtain:

$$\begin{aligned} &\sum_{j=1}^N \big(\theta\, u_j^{(i+1)} + (1-\theta)\, u_j^{(i)}\big)\Big(-\int_\Omega \boldsymbol{\sigma}\, \nabla\varphi_j \cdot \nabla\varphi_k \, \mathrm{d}\mathbf{x} + \int_{\partial\Omega} \big(\boldsymbol{\sigma}\, \nabla\varphi_j \cdot \mathbf{n}\big)\, \varphi_k \, \mathrm{d}\mathbf{x}\Big) \\ &\quad = \frac{1}{dt} \sum_{j=1}^N \big(u_j^{(i+1)} - u_j^{(i)}\big) \int_\Omega \varphi_j\, \varphi_k \, \mathrm{d}\mathbf{x}, \qquad \forall k = 1, \dots, N. \end{aligned} \tag{5.48}$$

This iteration step can be written in matrix notation in terms of the vectors of unknowns $\mathbf{u}^{(i)} = (u_0^{(i)}, \dots, u_N^{(i)})^\top$ at timestep i:

$$\mathbf{A}\, \mathbf{u}^{(i+1)} = \mathbf{b}(\mathbf{u}^{(i)}).$$

The system matrix $\mathbf{A}$ and the right-hand side $\mathbf{b}$ are given by:

$$\mathbf{A} = \theta\, (\mathbf{K}_\sigma + \mathbf{B}_\sigma) - \frac{1}{dt} \mathbf{M}, \qquad \mathbf{b} = \Big((\theta - 1)\, (\mathbf{K}_\sigma + \mathbf{B}_\sigma) - \frac{1}{dt} \mathbf{M}\Big)\, \mathbf{u}^{(i)}.$$

The formulation uses the standard stiffness matrix $\mathbf{K}_\sigma$, the matrix $\mathbf{B}_\sigma$ of the boundary

integral and the mass matrix $\mathbf{M}$, whose components are defined as

$$\mathbf{K}_{\sigma,kj} = -\int_\Omega (\boldsymbol{\sigma}\,\nabla\varphi_j)\cdot\nabla\varphi_k\,\mathrm{d}\mathbf{x},\quad \mathbf{B}_{\sigma,kj} = \int_{\Gamma_f} \big((\boldsymbol{\sigma}\,\nabla\varphi_j)\cdot\mathbf{n}\big)\,\varphi_k\,\mathrm{d}\mathbf{x},\quad \mathbf{M}_{kj} = \int_\Omega \varphi_j\,\varphi_k\,\mathrm{d}\mathbf{x}. \tag{5.49}$$

Note that, after applying the divergence theorem, the definition of the stiffness matrix has a minus sign.

Next, we take into account the Neumann boundary condition $\boldsymbol{\sigma}\nabla u\cdot\mathbf{n} = f$ on the boundary Γ_f. The flux f over the boundary is discretized by M separate ansatz functions ψ_j on Γ_f as $f = \sum_{j=1}^M f_j\,\psi_j$. The flux values are summarized in a vector $\mathbf{f} = (f_1,\dots,f_M)^\top$. Plugging this into Eq. (5.48) yields the following equation in matrix notation:

$$\tilde{\mathbf{A}}\,\mathbf{u}^{(i+1)} = \tilde{\mathbf{b}}(\mathbf{u}^{(i)}), \tag{5.50}$$

with the system matrix $\tilde{\mathbf{A}}$ and right-hand side $\tilde{\mathbf{b}}$:

$$\tilde{\mathbf{A}} = \theta\,\mathbf{K}_\sigma - \frac{1}{dt}\mathbf{M},\qquad \tilde{\mathbf{b}} = \Big((\theta-1)\,\mathbf{K}_\sigma - \frac{1}{dt}\mathbf{M}\Big)\mathbf{u}^{(i)} - \mathbf{B}_{\Gamma_f}\big(\theta\,\mathbf{f}^{(i+1)} + (1-\theta)\,\mathbf{f}^{(i)}\big),$$

and the boundary matrix $\mathbf{B}_{\Gamma_f}$ given by:

$$\mathbf{B}_{\Gamma_f,kj} = \int_{\Gamma_f} \psi_j\,\varphi_k\,\mathrm{d}\mathbf{x}. \tag{5.51}$$

Note that incorporating the Neumann boundary conditions in the weak form corresponds to the following exchange of the boundary matrices $\mathbf{B}_\sigma$ and $\mathbf{B}_{\Gamma_f}$:

$$\mathbf{B}_\sigma\,\mathbf{u} = \mathbf{B}_{\Gamma_f}\,\mathbf{f}. \tag{5.52}$$

Equation (5.50) is used to solve the diffusion part of the monodomain equation given in Eq. (5.11) after inserting the corresponding constant prefactors.

When deriving or implementing new models or optimizing solver code, it is often beneficial to study certain effects in isolation. It can help to use a toy problem such as the simple Laplace problem $\Delta u = 0$, possibly with Neumann boundary condition $\partial u/\partial\mathbf{n} = f$. By specializing the formulation in Eq. (5.50) accordingly, we obtain the system

$$(\mathbf{K}_\mathrm{I} + \mathbf{B}_\mathrm{I})\,\mathbf{u} = \mathbf{0}$$

for the case without boundary condition (set $\mathbf{B_I}$ to zero to assume homogeneous Neumann boundaries) or

$$\mathbf{K_I}\,\mathbf{u} = -\mathbf{B}_{\Gamma_f}\,\mathbf{f} \tag{5.53}$$

to include the formulated Neumann boundary condition.

5.3.3 Using Mass Lumping for Implicit Timestepping

Implicit timestepping schemes such as implicit Euler or the Crank-Nicolson scheme for $\theta = \frac{1}{2}$ need to solve a linear equation in every timestep. Assuming homogeneous Neumann boundary conditions for simplicity, the iteration step of the canonical Crank-Nicolson scheme follows from Eq. (5.50):

$$\left(\frac{1}{2}\mathbf{K} - \frac{1}{dt}\mathbf{M}\right)\mathbf{u}^{(i+1)} = \left(-\frac{1}{2}\mathbf{K} - \frac{1}{dt}\mathbf{M}\right)\mathbf{u}^{(i)} \tag{5.54a}$$

$$\Leftrightarrow \quad (\mathbf{I} - \frac{dt}{2}\,\mathbf{M}^{-1}\mathbf{K})\,\mathbf{u}^{(i+1)} = (\mathbf{I} + \frac{dt}{2}\,\mathbf{M}^{-1}\mathbf{K})\,\mathbf{u}^{(i)}. \tag{5.54b}$$

For the implicit Euler method, we obtain:

$$(\mathbf{K} - \frac{\mathbf{M}}{dt})\,\mathbf{u}^{(i+1)} = -\frac{\mathbf{M}}{dt}\mathbf{u}^{(i)} \tag{5.55a}$$

$$\Leftrightarrow \quad (\mathbf{I} - dt\,\mathbf{M}^{-1}\mathbf{K})\,\mathbf{u}^{(i+1)} = \mathbf{u}^{(i)}. \tag{5.55b}$$

Both iteration steps in Eqs. (5.54a) and (5.54b) and in Eqs. (5.55) and (5.55a) are equivalent, as the second equation follows from the first one by left multiplication of $(-dt\mathbf{M}^{-1})$. In the second equations, the matrices to be multiplied are created by a sum of the unity matrix $\mathbf{I}$ and another matrix term that is scaled by the potentially small timestep width dt. For the implicit Euler in Eq. (5.55b), the matrix on the right-hand side even reduces to the identity matrix. This is preferred over the first iteration steps in Eqs. (5.54a) and (5.55a) as it leads to better conditioned matrix-vector multiplications.

The required inversions of the mass matrix cannot be carried out explicitly as the inversion would fill in numerous matrix entries and eliminate the sparse structure. This is not feasible for highly resolved meshes with many degrees of freedom. Instead, *mass lumping* is used, where the mass matrix $\mathbf{M}$ is approximated by a diagonal matrix with diagonal entries equal to the row sums in $\mathbf{M}$ [Hin76]. Thus, multiplication with the

inverse mass matrix corresponds to a rescaling of columns by the inverse lumped diagonal entries.

5.3.4 Discretization of the Multidomain Model

With the prerequisites of temporal discretization in Sec. 5.3.1 and the finite element formulation of a diffusion equation in Sec. 5.3.2, we can now discretize the multidomain model. Since this has not been previously done in literature using the finite element method, the subsequent derivation is more detailed.

The first multidomain equation given in Eq. (5.14) yields the following form after applying the finite element derivation in Eq. (5.50):

$$\left(\mathbf{K}_{\boldsymbol{\sigma}_e+\boldsymbol{\sigma}_i} + \mathbf{B}_{\boldsymbol{\sigma}_e+\boldsymbol{\sigma}_i}\right)\boldsymbol{\phi}_e + \sum_{k=1}^{N_{\text{MU}}} f_r^k\left(\mathbf{K}_{\boldsymbol{\sigma}_i^k} + \mathbf{B}_{\boldsymbol{\sigma}_i^k}\right)\mathbf{V}_m^k = 0. \tag{5.56}$$

Here, $\boldsymbol{\phi}_e$ and $\mathbf{V}_m^k$ are the vectors of degrees of freedom for the extracellular potential ϕ_e and membrane voltage V_m^k of compartment k. The matrices are defined by Eq. (5.49) and do not yet include the boundary conditions. The subscripts of the stiffness matrices $\mathbf{K}$ and boundary integral matrices $\mathbf{B}$ refer to the anisotropy tensors that occur in their definitions.

The diffusion part of the second multidomain equation, Eq. (5.15), discretized with Crank-Nicolson, yields the system

$$\mathbf{A}\begin{pmatrix}\mathbf{V}_m^{k,(i+1)}\\ \boldsymbol{\phi}_e^{(i+1)}\end{pmatrix} = \mathbf{b}, \tag{5.57}$$

with the 1×2 block system matrix $\mathbf{A}$ and right-hand side vector $\mathbf{b}$ given by:

$$\mathbf{A} = \left[\frac{\theta}{A_m^k\, C_m^k}\left(\mathbf{K}_{\boldsymbol{\sigma}_i^k} + \mathbf{B}_{\boldsymbol{\sigma}_i^k}\right) - \frac{1}{dt}\mathbf{M} \qquad \frac{\theta}{A_m^k\, C_m^k}\left(\mathbf{K}_{\boldsymbol{\sigma}_i^k} + \mathbf{B}_{\boldsymbol{\sigma}_i^k}\right)\right], \tag{5.58a}$$

$$\mathbf{b} = \left(\frac{\theta-1}{A_m^k\, C_m^k}\left(\mathbf{K}_{\boldsymbol{\sigma}_i^k} + \mathbf{B}_{\boldsymbol{\sigma}_i^k}\right) - \frac{1}{dt}\mathbf{M}\right)\mathbf{V}_m^{k,(i)} + \frac{\theta-1}{A_m^k\, C_m^k}\left(\mathbf{K}_{\boldsymbol{\sigma}_i^k} + \mathbf{B}_{\boldsymbol{\sigma}_i^k}\right)\boldsymbol{\phi}_e^{(i)}. \tag{5.58b}$$

A separate instance of this equation holds for every compartment k. Again, the integrals over the boundary are still present in the $\mathbf{B}_{\boldsymbol{\sigma}_i^k}$ matrices. To resolve this and to close the

formulation, we have to consider the fluxes over the boundary of all involved unknowns and to replace them using the boundary conditions.

One required boundary conditions to solve the multidomain model without body domain is given in Eq. (5.16a). The boundary condition for compartment k in terms of the intracellular potential ϕ_i^k,

$$(\boldsymbol{\sigma}_i^k \nabla \phi_i^k) \cdot \mathbf{n}_m = 0 \qquad \text{on } \partial\Omega_M, \tag{5.59}$$

is expressed in terms of the unknowns V_m^k and ϕ_e to yield the condition

$$(\boldsymbol{\sigma}_i^k \nabla V_m^k) \cdot \mathbf{n}_m = -(\boldsymbol{\sigma}_i^k \nabla \phi_e) \cdot \mathbf{n}_m =: p^k \qquad \text{on } \partial\Omega_M. \tag{5.60}$$

We define the value of this flux to be equal to a helper variable p^k. A second flux is formulated for the extracellular potential ϕ_e. We assign its value to the helper variable q:

$$(\boldsymbol{\sigma}_e \nabla \phi_e) \cdot \mathbf{n}_m =: q \qquad \text{on } \partial\Omega_M. \tag{5.61}$$

We can now express the flux value $\big((\boldsymbol{\sigma}_e + \boldsymbol{\sigma}_i)\nabla\phi_e\big) \cdot \mathbf{n}_m$, which occurs in the discretized first multidomain equation, Eq. (5.56), in terms of the variables p^k and q. Using Eqs. (5.59) and (5.60) and the relation $\phi_e = \phi_i^k - V_m^k$, we derive:

$$\big((\boldsymbol{\sigma}_e + \boldsymbol{\sigma}_i)\nabla\phi_e\big) \cdot \mathbf{n}_m = (\boldsymbol{\sigma}_e \nabla \phi_e) \cdot \mathbf{n}_m + (\boldsymbol{\sigma}_i \nabla \phi_e) \cdot \mathbf{n}_m = q - \sum_{k=1}^{N_{\text{MU}}} f_r^k p^k. \tag{5.62}$$

We discretize the flux values p^k and q analogously to the Neumann boundary condition flux f in Sec. 5.3.2 and summarize the degrees of freedoms in vectors $\mathbf{p}^k$ and $\mathbf{q}$.

Next, we combine the flux values with the first and second multidomain equation. Plugging the generic relation Eq. (5.52) for boundary integral terms into the discretization of the first multidomain equation, Eq. (5.56), and using the derived flux values in Eqs. (5.60) and (5.62) leads in a first step to the following equation:

$$\mathbf{K}_{\boldsymbol{\sigma}_e+\boldsymbol{\sigma}_i} \boldsymbol{\phi}_e + \mathbf{B}_{\Gamma_M} \Big(\mathbf{q} - \sum_{k=1}^{N_{\text{MU}}} f_r^k \mathbf{p}^k\Big) + \sum_{k=1}^{N_{\text{MU}}} f_r^k \big(\mathbf{K}_{\boldsymbol{\sigma}_i^k} \mathbf{V}_m^k + \mathbf{B}_{\Gamma_M} \mathbf{p}^k\big) = 0.$$

It can be seen that the terms involving $\mathbf{p}^k$ cancel out, such that we get:

$$\mathbf{K}_{\sigma_e+\sigma_i}\,\boldsymbol{\phi}_e + \sum_{k=1}^{N_{\text{MU}}} f_r^k \mathbf{K}_{\sigma_i^k}\,\mathbf{V}_m^k = -\mathbf{B}_{\Gamma_M}\,\mathbf{q}. \tag{5.63}$$

If the multidomain description is used without body fat domain, the boundary condition in Eq. (5.16b) is used and the right-hand side in Eq. (5.63) vanishes. If a body domain is considered, the right-hand side interacts with the body domain model, which is considered in the next section.

Adding boundary conditions to the discretization of the second multidomain equation proceeds using Eqs. (5.57) and (5.58). Carrying out the analog procedure to the first multidomain equation, we plug in Eq. (5.52) to yield the matrix equation

$$\mathbf{A}\begin{pmatrix}\mathbf{V}_m^{k,(i+1)}\\ \boldsymbol{\phi}_e^{(i+1)}\end{pmatrix} = \mathbf{b} \tag{5.64}$$

with system matrix $\mathbf{A}$ and right-hand side vector $\mathbf{b}$ given by

$$\mathbf{A} = \left[\frac{\theta}{A_m^k\,C_m^k}\mathbf{K}_{\sigma_i^k} - \frac{1}{dt}\mathbf{M} \qquad \frac{\theta}{A_m^k\,C_m^k}\mathbf{K}_{\sigma_i^k},\right], \tag{5.65}$$

$$\mathbf{b} = \Big(\frac{\theta-1}{A_m^k\,C_m^k}\mathbf{K}_{\sigma_i^k} - \frac{1}{dt}\mathbf{M}\Big)\mathbf{V}_m^{k,(i)} + \frac{\theta-1}{A_m^k\,C_m^k}\mathbf{K}_{\sigma_i^k}\,\boldsymbol{\phi}_e^{(i)}$$

$$+\frac{\theta-1}{A_m^k\,C_m^k}\mathbf{B}_{\Gamma_M}\,\mathbf{p}^{k,(i)} - \frac{\theta-1}{A_m^k\,C_m^k}\mathbf{B}_{\Gamma_M}\,\mathbf{p}^{k,(i)} - \frac{\theta}{A_m^k\,C_m^k}\mathbf{B}_{\Gamma_M}\,\mathbf{p}^{k,(i+1)} + \frac{\theta}{A_m^k\,C_m^k}\mathbf{B}_{\Gamma_M}\,\mathbf{p}^{k,(i+1)}.$$

Again, the boundary terms involving $\mathbf{p}^k$ vanish to yield the following expression for $\mathbf{b}$:

$$\mathbf{b} = \Big(\frac{\theta-1}{A_m^k\,C_m^k}\mathbf{K}_{\sigma_i^k} - \frac{1}{dt}\mathbf{M}\Big)\mathbf{V}_m^{k,(i)} + \frac{\theta-1}{A_m^k\,C_m^k}\mathbf{K}_{\sigma_i^k}\,\boldsymbol{\phi}_e^{(i)}. \tag{5.66}$$

In summary, Eq. (5.63) with $\mathbf{q} = \mathbf{0}$ coupled with N_{MU} instances of Equations (5.64) to (5.66) comprises the discretization for the multidomain model without body domain. Definitions of the involved stiffness and mass matrices are given in Eq. (5.49).

5.3.5 Discretization of the Multidomain Model for Surface EMG

To discretize the multidomain model with the electric potential ϕ_b in the body domain, we extend the formulation without body domain in Sec. 5.3.4. The body domain adds the electric potential ϕ_b to the vector of unknowns, for which the system has to be solved. As before, we discretize the field using finite element ansatz functions and solve for the vector $\boldsymbol{\phi}_b$ of degrees of freedom.

The model for ϕ_b is the Laplace equation given in Eq. (5.17) with homogeneous Neumann boundary conditions given in Eq. (5.19). According to Eq. (5.53), the discretized equation is given by

$$\mathbf{K}_{\sigma_b}\, \boldsymbol{\phi}_b = 0. \tag{5.67}$$

In addition, the value of the body potential ϕ_b is coupled to the extracellular potential ϕ_e in the muscle domain Ω_M via the coupling conditions on the boundary Γ_M given in Eq. (5.18).

We write the discretized and coupled multidomain equations as a linear system of equations in generic block-matrix form:

$$\left[\begin{array}{c|c|c|c} \mathbf{A}^k_{V_m,V_m} & \mathbf{B}^k_{V_m,\phi_e} & & \\ \mathbf{B}^k_{\phi_e,V_m} & \mathbf{B}_{\phi_e,\phi_e} & & \mathbf{B}_{\Gamma_M} \\ \hline & & \mathbf{C}_{\phi_b,\phi_b} & -\mathbf{B}_{\Gamma_M} \\ \hline & \mathbf{I}_{\Gamma_M,\phi_e} & -\mathbf{I}_{\Gamma_M,\phi_b} & \end{array}\right] \left[\begin{array}{c} \mathbf{V}_m^{k,(i+1)} \\ \hline \boldsymbol{\phi}_e^{(i+1)} \\ \hline \boldsymbol{\phi}_b^{(i+1)} \\ \hline \mathbf{q}^{(i+1)} \end{array}\right] = \left[\begin{array}{c} \mathbf{b}^{k,(i)}_{V_m} \\ \mathbf{0} \\ \hline \mathbf{0} \\ \hline \mathbf{0} \end{array}\right]. \tag{5.68}$$

The vector of unknowns consists of the degrees of freedom in the finite element formulation at the next timestep $(i+1)$ of the transmembrane voltage $\mathbf{V}_m^{k,(i+1)}$, the extracellular potential $\boldsymbol{\phi}_e^{(i+1)}$, the body potential $\boldsymbol{\phi}_b^{(i+1)}$, and additionally the flux $\mathbf{q}^{(i+1)}$ over the shared boundary Γ_M of the muscle and the body domain, which was defined in Eq. (5.61). For illustration purposes, only one compartment, $k=1$, for one MU, $N_{\text{MU}}=1$, is considered.

We refer to parts of the matrix in Eq. (5.68) as block rows and block columns according to the given block-structure.

The first block row in the matrix equation is given by the discretized second multidomain equation. Following Eqs. (5.65) and (5.66), the matrices and the right-hand side

are given by

$$\mathbf{A}^k_{V_m,V_m} = \frac{\theta}{A^k_m\, C^k_m}\mathbf{K}_{\sigma^k_i} - \frac{1}{dt}\mathbf{M}, \qquad \mathbf{B}^k_{V_m,\phi_e} = \frac{\theta}{A^k_m\, C^k_m}\mathbf{K}_{\sigma^k_i},$$

$$\mathbf{b}^{k,(i)}_{V_m} = \Big(\frac{\theta-1}{A^k_m\, C^k_m}\mathbf{K}_{\sigma^k_i} - \frac{1}{dt}\mathbf{M}\Big)\mathbf{V}^{k,(i)}_m + \frac{\theta-1}{A^k_m\, C^k_m}\mathbf{K}_{\sigma^k_i}\,\boldsymbol{\phi}^{(i)}_e.$$

The second block row describes the first multidomain equation that was derived in Eq. (5.63). The flux term $\mathbf{q}$ has been brought to the left-hand side and is incorporated by the boundary matrix $\mathbf{B}_{\Gamma_M}$ defined in Eq. (5.51). The other matrices are formulated as follows:

$$\mathbf{B}^k_{\phi_e,V_m} = f^k_r \mathbf{K}_{\sigma^k_i}, \qquad \mathbf{B}_{\phi_e,\phi_e} = \mathbf{K}_{\sigma_e+\sigma_i}.$$

The third block row is the formulation of the harmonic body potential ϕ_b and the matrix $\mathbf{C}_{\phi_b,\phi_b}$ equals the system matrix $\mathbf{K}_{\sigma_b}$ in Eq. (5.67). The coupling condition on the flux q in Eq. (5.18b) is accounted for by including the boundary matrix $\mathbf{B}_{\Gamma_M}$ in the last column. The minus sign comes from the fact that the outward normal vector on Γ_M as the boundary of Ω_B has the opposite direction to the normal vector on Γ_M that is used for the models in the muscle domain Ω_M. Using the helper variable $\mathbf{q}^{(i+1)}$, the second and third row of Eq. (5.68) are coupled according to the prescribed condition in Eq. (5.18b).

The other coupling condition, Eq. (5.18a), is accounted for by the last block row in Eq. (5.67). The degrees of freedom for the extracellular potential $\boldsymbol{\phi}^{(i+1)}_e$ and the body potential $\boldsymbol{\phi}^{(i+1)}_b$ have equal values on the boundary Γ_M. The matrices $\mathbf{I}_{\Gamma_M,\phi_e}$ and $\mathbf{I}_{\Gamma_M,\phi_b}$ are identity matrices that only have nonzero entries on the diagonal for the boundary degrees of freedom in the meshes of muscle domain and body domain, respectively.

Because the vector $\mathbf{q}^{(i+1)}$ is not an unknown in the system, the respective values in Eq. (5.68) have to be eliminated. As a result, we obtain the following system, which is

formulated for a generic number N_{MU} of MUs:

$$\left[\begin{array}{ccc|c|c} \mathbf{A}^1_{V_m,V_m} & & & \mathbf{B}^1_{V_m,\phi_e} & \\ & \ddots & & \vdots & \\ & & \mathbf{A}^{N_{\text{MU}}}_{\phi_e,V_m} & \mathbf{B}^{N_{\text{MU}}}_{V_m,\phi_e} & \\ \mathbf{B}^1_{\phi_e,V_m} & \cdots & \mathbf{B}^{N_{\text{MU}}}_{\phi_e,V_m} & \mathbf{B}_{\phi_e,\phi_e} & \mathbf{D} \\ \hline & & & \mathbf{E} & \tilde{\mathbf{C}}_{\phi_b,\phi_b} \end{array}\right] \left[\begin{array}{c} \mathbf{V}^{1,(i+1)}_m \\ \vdots \\ \mathbf{V}^{N_{\text{MU}},(i+1)}_m \\ \hline \boldsymbol{\phi}^{(i+1)}_e \\ \hline \tilde{\boldsymbol{\phi}}^{(i+1)}_b \end{array}\right] = \left[\begin{array}{c} \mathbf{b}^{1,(i)}_{V_m} \\ \vdots \\ \mathbf{b}^{N_{\text{MU}},(i)}_{V_m} \\ \mathbf{0} \\ \hline \mathbf{0} \end{array}\right]. \tag{5.69}$$

Formally, the elimination step is carried out by adding the equations of the third block row in Eq. (5.68), that correspond to the boundary degrees of freedom on Γ_M, to the corresponding equations of the same degrees of freedom in the second block row. This eliminates the last block column, which corresponds to $\mathbf{q}^{(i+1)}$. Next, the duplicate boundary degrees of freedom, that appear in both the Ω_M and Ω_B meshes, get unified. The corresponding matrix columns in the third block column are removed. To preserve the entries in the third block row, they are added in the sub matrix of block row three and block column two.

Now considering the updated matrix equation in Eq. (5.69), all sub blocks are equal to Eq. (5.68), except for the former matrix $\mathbf{C}_{\phi_b,\phi_b}$ and the new matrices $\mathbf{D}$ and $\mathbf{E}$. The new matrix $\tilde{\mathbf{C}}_{\phi_b,\phi_b}$ is obtained from $\mathbf{C}_{\phi_b,\phi_b}$ by removing all rows and columns of boundary degrees of freedom. The removed entries are contained in the new matrices $\mathbf{D}$ and $\mathbf{E}$.

The size of the system matrix in Eq. (5.69) equals $a \times a$, where the number a is composed of $N_{\text{MU}} + 1$ times the number of degrees of freedom in the muscle mesh plus the number of degrees of freedom in the fat layer mesh without the boundary degrees of freedom on Γ_M. Accordingly, the vector $\tilde{\boldsymbol{\phi}}^{(i+1)}_b$ is the same as $\boldsymbol{\phi}^{(i+1)}_b$ except that it does not contain the boundary degrees of freedom, which are already included in $\boldsymbol{\phi}^{(i+1)}_e$.

Equation (5.69) describes one iteration of the Crank-Nicolson scheme that is used to solve the multidomain model. This iteration is carried out alternatingly with the subcellular model according to the chosen operator splitting scheme.

The first N_{MU} block rows in Eq. (5.69) contain the second multidomain equation for every MU. The second-to-last block row contains the first multidomain equation and the last block row contains the body fat layer model.

Because of the implicit formulation, electric conduction in the intracellular and extracellular space and the body domain are bidirectionally coupled. Therefore, the model

can be used to simulate the effects of natural activation in the muscle on EMG signals on the skin surface as well as the reverse effect of external stimulation on the surface on the electrophysiology.

5.3.6 Discretization of the Fiber Based Electrophysiology Model

The fiber based electrophysiology model consists of multiple independent 1D fiber domains, where the monodomain equation Eq. (5.11) is solved. The transmembrane voltage V_m is then mapped to a 3D mesh of the muscle domain and unidirectionally coupled to the first bidomain equation Eq. (5.9a). The first bidomain equation is solved for the extracellular potential ϕ_e and possibly the electric potential ϕ_b in the body fat domain, which corresponds to EMG signals on the skin surface.

The temporal discretization of the monodomain equation was described in Sec. 5.3.1. The diffusion term within the operator splitting requires a spatial discretization for which we use the finite element method. This 1D diffusion equation is given as

$$\frac{\partial V_m}{\partial t} = \frac{\sigma_{\text{eff}}}{A_m\, C_m} \frac{\partial^2 V_m}{\partial x^2}. \tag{5.70}$$

It can be solved using a timestepping scheme such as the implicit Euler method to obtain time-discrete values $V_m^{(i)}, i = 1, 2, \dots$ for the transmembrane potential. The discretization leads to the matrix equation given in Eq. (5.50) and to the variants presented in Sec. 5.3.3 if mass lumping is used. In the stiffness and mass matrices, the anisotropic conduction tensor is replaced by the constant scalar prefactor $c := \sigma_{\text{eff}}/(A_m\, C_m)$ of the spatial second derivative in Eq. (5.70).

The first bidomain equation Eq. (5.9a) is a 3D Poisson problem in terms of the unknown extracellular potential ϕ_e. According to Eq. (5.53), the finite element discretization is given by

$$\mathbf{K}_{\sigma_i+\sigma_e} \boldsymbol{\phi}_e^{(i+1)} = -\mathbf{B}_{\Gamma_f}\, \mathbf{f} + \mathbf{rhs},$$

where the right-hand side **rhs** of the Poisson problem is the transmembrane flow and is given by

$$\mathbf{rhs} = -\mathbf{K}_{\sigma_i} \mathbf{V}_{m,3D}^{(i+1)}. \tag{5.71}$$

Here, $\boldsymbol{\phi}_e^{(i+1)}$ and $\mathbf{V}_{m,3D}^{(i+1)}$ are the vectors of degrees of freedom on the 3D mesh for the extracellular potential ϕ_e and the membrane potential V_m at timestep $(i+1)$. With the homogeneous Neumann boundary conditions for V_m and ϕ_e given in Eq. (5.12), the boundary term $\mathbf{B}_{\Gamma_f}$ vanishes.

In summary, the following matrix equations are solved for the fiber based electrophysiology model with n fibers:

$$\begin{bmatrix} \mathbf{A} & & \\ & \ddots & \\ & & \mathbf{A} \end{bmatrix} \begin{bmatrix} \mathbf{V}_m^{1,(i+1)} \\ \vdots \\ \mathbf{V}_m^{n,(i+1)} \end{bmatrix} = \begin{bmatrix} \mathbf{V}_m^{1,(i)} \\ \vdots \\ \mathbf{V}_m^{n,(i)} \end{bmatrix}, \tag{5.72a}$$

$$\mathbf{V}_{m,3D}^{(i+1)} = \mathbf{P} \begin{bmatrix} \mathbf{V}_m^{1,(i+1)} \\ \vdots \\ \mathbf{V}_m^{n,(i+1)} \end{bmatrix}, \tag{5.72b}$$

$$\mathbf{K}_{\sigma_i+\sigma_e} \boldsymbol{\phi}_e^{(i+1)} = -\mathbf{K}_{\sigma_i} \mathbf{V}_{m,3D}^{(i+1)} \tag{5.72c}$$

with the system matrix $\mathbf{A}$ for a single fiber given according to Eq. (5.55b) by

$$\mathbf{A} = \mathbf{I} - dt\, \mathbf{M}_c^{-1} \mathbf{K}_c.$$

Equation (5.72a) solves the diffusion part of the operator splitting in Eq. (5.44). After the values $\mathbf{V}_m^{j,(i+1)}$ for the timestep $(i+1)$ are computed on the 1D fiber meshes, the homogenized vector $\mathbf{V}_{m,3D}^{(i+1)}$ in the 3D mesh of the muscle domain Ω_M is obtained by the prolongation operation $\mathbf{P}$ in Eq. (5.72b). The homogenized vector is used in the right-hand side of the bidomain model in Eq. (5.72c), which computes the discretized extracellular potential $\boldsymbol{\phi}_e^{(i+1)}$.

Equation (5.72c) can be extended by adding a body fat layer Ω_B and the corresponding model for the electric potential $\phi_b^{(i+1)}$. Then, the vector of unknowns contains the degrees of freedom for both $\phi_e^{(i+1)}$ and $\phi_b^{(i+1)}$. The stiffness matrix $\mathbf{K}_{\sigma_i+\sigma_e}$ is obtained by integrating over both meshes in $\Omega_M \cup \Omega_B$. Only in the elements of the finite element mesh for Ω_B, the conduction tensors are redefined as $\boldsymbol{\sigma}_i = \mathbf{0}$ and $\boldsymbol{\sigma}_e = \boldsymbol{\sigma}_b$. This sets the right-hand side of Eq. (5.72c) to zero in Ω_B and the solution ϕ_b in harmonic according to the model in Eq. (5.17). The coupling conditions Eq. (5.18) between ϕ_e and ϕ_b and the outer Neumann boundary conditions Eq. (5.19) for ϕ_b are satisfied automatically by this approach.

The comparison between the discretized multidomain model in Eq. (5.69) with the

discretized fiber based model in Eq. (5.72) reveals several differences. Whereas the multidomain description consists of a single coupled linear system for electric conduction in the intracellular, extracellular and body domains, the formulations are only unidirectionally coupled in the fiber based description. While the multidomain model always computes the EMG signals on the skin surface in every timestep, the corresponding model in the fiber based description can be solved with larger timestep widths, using subcycling for the action potential propagation model.

As can be seen in Eq. (5.72a), the system matrix is decoupled and contains independent problems for every fiber. This is an advantage compared to the multidomain model, where a system describing the whole muscle domain has to be solved. On the downside, separate representations of the transmembrane voltage V_m exist in the fiber based description. The representation in the 3D mesh has to be computed by interpolation from the representation on the fibers. The multidomain description has a single vector of degrees of freedom for V_m with fewer entries than in the fiber-based description.

5.3.7 Summary of Domains and Meshes

Various finite element meshes occur in the formulation of the multi-scale model. If the fiber based description is used, the description requires finite element meshes for the 1D fiber domains Ω_f^j for $j = 1, \dots, n$. Further meshes are needed for the 3D muscle domain Ω_M and for the 3D body domain Ω_B. The meshes for Ω_M and Ω_B share nodes on their common boundary Γ_M. The fiber meshes are embedded in the muscle domain. Their nodes do not necessarily have to coincide with the nodes of the muscle mesh.

The subcellular model is solved at locations Ω_s^i for $i = 1, \dots, m$. These locations are the nodes of the fiber meshes for the fiber based description and the nodes of the muscle mesh for the multidomain description. We therefore have the inclusion $\Omega_s^i \subset \Omega_f^j \subset \Omega_M$.

For the solid mechanics model, the unified 3D domain $\Omega = \Omega_M \cup \Omega_B$ is used. The mesh for the continuum mechanics formulation can be different from the meshes used for the electrophysiology model. In fact, the continuum mechanics mesh has special requirements in order to yield a consistent formulation. Our implementation uses two overlaid meshes of quadratic and linear hexahedral elements for displacements and the hydrostatic pressure.

Often, the required accuracy of the electrophysiology model is higher than for the continuum mechanics model, such that differently resolved meshes can be used. To

facilitate data mapping, the nodes of the mechanics mesh should be chosen as subset of the nodes of the electrophysiology meshes.

5.4 Discretization and Solution Approach for the Solid Mechanics Model

After the formulation of linear and nonlinear models for solid mechanics in Sec. 5.2, this section discusses their discretization and derives finite element formulations for the linearized model and the nonlinear model, both static and dynamic. We also describe the algorithms used to obtain a numerical solution.

The implementation of a solver for generic hyperelastic descriptions is an interdisciplinary endeavor, if parallel execution is exploited and the model is integrated in a multi-scale biomechanics model. Therefore, we give a comprehensive derivation of the formulas used to numerically solve the equations matching the implementation in OpenDiHu, such that the implementation is also comprehensible for readers that are not specialized in the field of continuum mechanics. More details on finite element discretizations for solid mechanics models can be found in the literature [Zie77; Sus87; Zie05].

5.4.1 Discretization of the Linear Model

In this section, we discuss the linearized and static model. Besides the nonlinear model, our software OpenDiHu also implements the linearized description. The linear model exhibits better numerical properties and can be solved faster than the generic model. Thus, it can serve as a toy problem or can be used for mechanical systems, where the linearization assumptions are valid.

By assuming small strains, we can use the linearized kinematic relation in Eq. (5.25) to express the linear strain tensor $\boldsymbol{\varepsilon}$. The material model is Hooke's law formulated in Eq. (5.33). It relates the strain to the Cauchy stress by $\boldsymbol{\sigma} = \mathbb{C} : \boldsymbol{\varepsilon}$.

Using variational calculus, the system response of external forces and infinitesimal, compatible, virtual displacements $\delta\mathbf{u}$ is studied. We start with the *principle of virtual work*, which states that in equilibrium the virtual work δW performed by external forces

along any virtual displacements $\delta\mathbf{u}$ is zero. Equivalently, the external virtual work δW_{ext} is equal to the internal virtual work δW_{int}:

$$\delta W_{\text{int}}(\mathbf{u}, \delta\mathbf{u}) = \delta W_{\text{ext}}(\delta\mathbf{u}) \qquad \forall \delta\mathbf{u} \in H_0^1(\Omega). \tag{5.73}$$

Here, the external virtual work δW_{ext} is given by the product of external forces $\mathbf{t}$ and the virtual displacements $\delta\mathbf{u}$ at the same location. The internal virtual work δW_{int} is the body's response in terms of stresses $\boldsymbol{\sigma}$ and virtual strains $\boldsymbol{\varepsilon}$. In summary, Eq. (5.73) is equivalent to the following equilibrium equation:

$$\int_\Omega \boldsymbol{\sigma}(\mathbf{u}) : \delta\boldsymbol{\varepsilon}\,\mathrm{d}\mathbf{x} = \int_{\partial\Omega} \mathbf{t} : \delta\mathbf{u}\,\mathrm{d}\mathbf{x} \qquad \forall \delta\mathbf{u} \in H_0^1(\Omega). \tag{5.74}$$

The vectors contain the degrees of freedom of a finite element discretization. The operator ":" denotes the component-wise product.

Often, it is easier to write the equations in component form. Indices $a, b, c, \dots$ are used to specify a dimension index in $\{1, \dots, d\}$. The letters $L, M \in \{1, \dots, N\}$ denote indices over degrees of freedom in a mesh with N nodes. The Einstein sum convention is used where repeated indices implicitly indicate summation, except when the indices are in parentheses. Thus, the right-hand side $\mathbf{f}$ of Eq. (5.74) with ansatz functions ϕ^L and the degrees of freedom δu_a^L of $\delta\mathbf{u}$ can be written as:

$$\mathbf{f}_a = \int_{\partial\Omega} t_{(a)}\, \delta u_{(a)}^L\, \phi^L\, \mathrm{d}\mathbf{x}. \tag{5.75}$$

By combining the kinematic relation between displacements $\mathbf{u}$ and linearized strains $\boldsymbol{\varepsilon}$ in Eq. (5.25), the material relation between $\boldsymbol{\varepsilon}$ and the stress $\boldsymbol{\sigma}$ in Eq. (5.33), the equilibrium relation between $\boldsymbol{\sigma}$ and the right-hand side vector $\mathbf{f}$ in Eq. (5.74) and after discretizing displacements and virtual displacements, we get the following linear matrix equation:

$$\mathbf{K}\,\mathbf{u} = \mathbf{f}. \tag{5.76}$$

The stiffness matrix $\mathbf{K}$ has rows and columns for every combination of degree of freedom $L, M \in \{1, \dots, N\}$ and dimension indices $a, b \in \{1, 2, 3\}$. The entries are given by:

$$\mathbf{K}_{LaMb} = \int_\Omega \mathbb{C}_{adbc} \frac{\partial \phi^L(\mathbf{x})}{\partial x_d} \frac{\partial \phi^M(\mathbf{x})}{\partial x_c}\,\mathrm{d}\mathbf{x}.$$

The resulting model in Eq. (5.76) describes the passive behavior of a body under the

linearization assumptions and can be used in an appropriate biomechanical application.

However, for contracting muscle tissue, we also need to incorporate active stresses that are generated at the sarcomeres in the muscle. As described in Eq. (5.35), an active stress term $\boldsymbol{\sigma}^{\text{active}}$ can be considered. Because this active term is prescribed by the activation dynamics and the subcellular model, it has to appear on the right-hand side of the linear model. We add the active stress term $\boldsymbol{\sigma}^{\text{active}}$ to the external virtual work in Eq. (5.73), yielding the equation:

$$\delta W_{\text{int}}(\mathbf{u}, \delta\mathbf{u}) = \mathbf{f} + \int_\Omega \boldsymbol{\sigma}^{\text{active}} : \delta\boldsymbol{\varepsilon}_- \,\mathrm{d}\mathbf{x} \qquad \forall \delta\mathbf{u} \in H_0^1(\Omega). \tag{5.77}$$

The active stress is associated with compression, i.e., negative virtual strains $\delta\boldsymbol{\varepsilon} < 0$. Therefore, we use $\delta\boldsymbol{\varepsilon}_-$ which is defined equal to $\delta\boldsymbol{\varepsilon}$ for $\delta\boldsymbol{\varepsilon} < 0$ and zero otherwise. From Eq. (5.77), we get the same discretized linear system as in Eq. (5.76), but with an additional term $\mathbf{f}^{\text{active}}$ in the right-hand side that contains the discretized prescribed active stress field $\boldsymbol{\sigma}_{ab}^{\text{active}}(\mathbf{x})$:

$$\mathbf{f}_{La}^{\text{active}} = \int_\Omega \boldsymbol{\sigma}_{ab}^{\text{active}}(\mathbf{x}) \,\frac{\partial \phi^L(\mathbf{x})}{\partial x_b} \,\mathrm{d}\mathbf{x}. \tag{5.78}$$

5.4.2 Discretization of the Nonlinear Static Hyperelastic Model

Next, we discuss the discretization of the nonlinear solid mechanics model, which uses the model equations introduced in Sec. 5.2. We begin with the discretization of a static, incompressible problem, where no velocities have to be considered. The discretization is extended to the dynamic model in Sec. 5.4.6.

As described in Sec. 5.2.4, the equilibrium equation can be formulated in terms of the *Hellinger-Reissner energy functional* $\Pi_L(\mathbf{u}, p) = \Pi_{\text{int}}(\mathbf{u}, p) + \Pi_{\text{ext}}(\mathbf{u})$ given in Eq. (5.31). It consists of the external energy functional, given in Eq. (5.32) as

$$\Pi_{\text{ext}}(\mathbf{u}) = -\int_{\Omega_0} \mathbf{B}\,\mathbf{u}\,\mathrm{d}V - \int_{\partial\Omega_0^t} \bar{\mathbf{T}}\,\mathbf{u}\,\mathrm{d}S,$$

with body force $\mathbf{B}$ and surface traction $\bar{\mathbf{T}}$, and of the internal energy functional

$$\Pi_{\text{int}}(\mathbf{u}, p) = \int_{\Omega_0} \Psi_{\text{iso}}\big(\bar{\mathbf{C}}(\mathbf{u})\big)\,\mathrm{d}V + \int_{\Omega_0} p\,\big(J(\mathbf{u}) - 1\big)\,\mathrm{d}V. \tag{5.79}$$

Here, Ψ_{iso} is the isochoric strain-energy density function introduced in Eq. (5.38) in terms of the reduced right Cauchy-Green tensor $\bar{\mathbf{C}}$ defined in Eq. (5.37). The first term in Eq. (5.79) describes the isochoric elastic response of the material, the second term adds the incompressibility constraint $J = 1$ with the Lagrange multiplier p. The value of p is computed as part of the model and can be identified as the hydrostatic pressure. Therefore, the second term is interpreted as the elastic response to compression and is included in the internal energy functional Π_{int}.

According to the *principle of stationary potential energy*, the system is in equilibrium, if the potential energy functional is stationary. This is the case, if the first variation $\delta\Pi_L$ is zero. Using the additive structure of Π_L, we can express the principle of stationarity as

$$D_{\delta\mathbf{u}}\Pi_L(\mathbf{u},p) = D_{\delta\mathbf{u}}\Pi_{\text{int}}(\mathbf{u},p) + D_{\delta\mathbf{u}}\Pi_{\text{ext}}(\mathbf{u}) \overset{!}{=} 0, \qquad \forall \delta\mathbf{u} \tag{5.80a}$$

$$D_{\delta p}\Pi_L(\mathbf{u},p) = D_{\delta p}\Pi_{\text{int}}(\mathbf{u},p) \overset{!}{=} 0 \qquad \forall \delta p. \tag{5.80b}$$

The variations of the internal and external energy functionals are defined as

$$D_{\delta\mathbf{u}}\Pi(\mathbf{u}) = \frac{\mathrm{d}}{\mathrm{d}\varepsilon}\Pi(\mathbf{u}+\varepsilon\delta\mathbf{u})\Big|_{\varepsilon=0}, \qquad D_{\delta p}\Pi(p) = \frac{\mathrm{d}}{\mathrm{d}\varepsilon}\Pi(p+\varepsilon\delta p)\Big|_{\varepsilon=0}. \tag{5.81}$$

They can be identified as the internal and external virtual work,

$$D_{\delta\mathbf{u}}\Pi_{\text{int}}(\mathbf{u},p) = \delta W_{\text{int}}, \qquad D_{\delta\mathbf{u}}\Pi_{\text{ext}}(\mathbf{u}) = -\delta W_{\text{ext}}.$$

Thus, Eq. (5.80a) can be expressed as

$$\delta W_{\text{int}} - \delta W_{\text{ext}} = 0,$$

which is the form of the equilibrium equation that was used in Eq. (5.73) in the derivation of the linearized model in Sec. 5.4.1 . The Euler-Lagrange equations corresponding to the variational problem are the local incompressibility constraint and the partial differential equation of balance of momentum presented in Eqs. (5.43a) and (5.43b).

Executing the derivative in the definitions of the variations in Eq. (5.81) yields the

following terms:

$$D_{\delta\mathbf{u}}\Pi_{\text{int}}(\mathbf{u},p) = \int_{\Omega_0} \mathbf{S}(\mathbf{u},p) : \delta\mathbf{E}(\delta\mathbf{u})\,\mathrm{d}V, \qquad D_{\delta p}\Pi_{\text{int}}(\mathbf{u},p) = \int_{\Omega_0} \big(J(\mathbf{u})-1\big)\delta p\,\mathrm{d}V,$$

$$D_{\delta\mathbf{u}}\Pi_{\text{ext}}(\mathbf{u}) = -\int_{\Omega_0} \mathbf{B}\cdot\delta\mathbf{u}\,\mathrm{d}V - \int_{\partial\Omega_0^t} \bar{\mathbf{T}}\cdot\delta\mathbf{u}\,\mathrm{d}S,$$

where the variational variables $\delta p, \delta\mathbf{u}$ and $\delta\mathbf{E}$ are the virtual pressure, virtual displacements, and virtual strains.

We discretize the solutions of the functional for the displacements $\mathbf{u}(\mathbf{x})$ and pressure $p(\mathbf{x})$ and their variations using different ansatz functions ϕ^L, $L = 1,\dots,N_u$ and ψ^L, $L = 1,\dots,N_p$:

$$u_a = \hat{u}_a^L \phi_{(a)}^L, \qquad \delta u_a = \delta\hat{u}_a^L \phi_{(a)}^L, \qquad p = \hat{p}^L \psi^L, \qquad \delta p = \delta\hat{p}^L \psi^L.$$

Again, Einstein summation over repeated indices, in this case the index L, is used. The displacements function is vector-valued and given by $\mathbf{u}(\mathbf{x}) = (u_1(\mathbf{x}), u_2(\mathbf{x}), u_3(\mathbf{x}))^\top$. The vectors containing the degrees of freedom are denoted by $\hat{\mathbf{u}} = (\hat{u}^L)_{L=1,\dots,N_u}$ and $\hat{\mathbf{p}} = (\hat{p}^L)_{L=1,\dots,N_p}$.

The kinematics equation to compute virtual strains from virtual displacements follows from Eq. (5.24) in Lagrangian description and is given by $\delta\mathbf{E} = \text{sym}(\mathbf{F}^\top\nabla\mathbf{u})$. Its discretized form is given as follows, where the subscript comma $\square_{,A}$ indicates the derivative with respect to the indexed coordinate $\mathbf{X}_A$:

$$\delta E_{AB} = \frac{1}{2}\left(F_{aB}\,\phi_{(a),A}^M + F_{aA}\,\phi_{(a),B}^M\right)\delta\hat{u}_a^M.$$

In summary, the resulting set of discretized nonlinear equations can be formulated as:

$$\delta W_{\text{int}}(\mathbf{u},p) - \delta W_{\text{ext}} = 0 \qquad \forall\,\delta\mathbf{u}, \tag{5.82}$$

$$D_{\delta p}\Pi_L(\mathbf{u}) = 0 \qquad \forall\,\delta p, \tag{5.83}$$

with the following discretized terms:

$$\delta W_{\text{int}}(\hat{\mathbf{u}}, \hat{\mathbf{p}}) = \int_\Omega \frac{1}{2} S_{AB}(\hat{\mathbf{u}}, \hat{\mathbf{p}}) \left(F_{aB}\, \phi^M_{(a),A} + F_{aA}\, \phi^M_{(a),B} \right) \delta \hat{u}^M_a \,\mathrm{d}V, \tag{5.84a}$$

$$\delta W_{\text{ext}} = \int_\Omega B_a \phi^M_{(a)}\, \delta \hat{u}^M_a \,\mathrm{d}V + \int_{\partial\Omega} \bar{T}^L_a\, \phi^L_{(a)}\, \phi^M_{(a)}\, \delta \hat{u}^M_a \,\mathrm{d}S, \tag{5.84b}$$

$$D_{\delta p} \Pi_L(\hat{\mathbf{u}}) = \int_\Omega \left(J(\hat{\mathbf{u}}) - 1 \right) \delta p \,\mathrm{d}V. \tag{5.84c}$$

The nonlinear system of equations in Eqs. (5.82) and (5.84) can now be solved for the unknown vectors $\hat{\mathbf{u}}$ and $\hat{\mathbf{p}}$ of degrees of freedom using a Newton scheme.

5.4.3 Discretization of the Nonlinear Dynamic Hyperelastic Model

We extend the discretization of the static model in the last section for the dynamic model. The vector of unknowns is extended by a velocity function $\mathbf{v} : \Omega_t \to \mathbb{R}^3$. The additional equation $\dot{\mathbf{u}} = \mathbf{v}$ relates the displacements and the velocity.

As noted in the derivation of the equilibrium equation in Sec. 5.2.4, the body force term $\mathbf{B}$ in the external energy functional also includes the inertial forces $\mathbf{B}_{\text{inertial}} = \rho_0\, \dot{\mathbf{v}}$ to describe the dynamic behavior.

The resulting nonlinear system of equations is given as follows:

$$\delta W_{\text{int}}(\mathbf{u}, p) - \delta W_{\text{ext}}(\mathbf{v}) = 0 \qquad \forall\, \delta\mathbf{u}, \tag{5.85a}$$

$$\mathbf{v} = \dot{\mathbf{u}}, \tag{5.85b}$$

$$D_{\delta p} \Pi_L(\mathbf{u}) = 0 \qquad \forall\, \delta p. \tag{5.85c}$$

5.4.4 Computation of the Stress Tensor and the Elasticity Tensor

In the Newton solver, we need to compute the stress tensor $\mathbf{S}$ and its derivative $\mathbb{C}$, called the elasticity tensor, given the current displacement field $\mathbf{u}$. The relations are defined by the material model given by the strain energy function. This section presents the algorithm how to obtain the values of $\mathbf{S}$ and $\mathbb{C}$ from the displacements $\mathbf{u}$. While the

derivation is formulated in terms of the displacement function $\mathbf{u}$, it is also valid for the finite element discretization, i.e., using the vector $\hat{\mathbf{u}}$ of degrees of freedom instead.

Following Eq. (5.36), the second Piola-Kirchhoff stress $\mathbf{S}$ is given by the derivative of the strain energy function Ψ with respect to $\mathbf{C}$. For the representation using the invariants, the chain rule has to be used:

$$\mathbf{S} = 2\,\frac{\partial \Psi(\mathbf{C})}{\partial \mathbf{C}} = \frac{\partial \Psi}{\partial I_a}\,\frac{\partial I_a}{\partial \mathbf{C}}.$$

Using the decoupled form, the resulting stresses are also decoupled as $\mathbf{S} = \mathbf{S}_{\text{vol}} + \mathbf{S}_{\text{iso}}$. The volumetric stress $\mathbf{S}_{\text{vol}}$ describes the elastic response to compression, the isochoric stress $\mathbf{S}_{\text{iso}}$ describes the response to the deviatoric deformation. In the following, all steps to compute these stresses are listed. The rationale is to give a condensed reference of the implemented algorithm in OpenDiHu to facilitate further development. For the derivation of all intermediate steps, we refer to the literature [Hol00].

At first, the reduced stress tensor $\bar{\mathbf{S}}$ that neglects the volumetric change is formulated as:

$$\bar{\mathbf{S}} = 2\frac{\partial \Psi_{\text{iso}}(\bar{I}_1, \bar{I}_2, \bar{I}_4, \bar{I}_5)}{\partial \bar{\mathbf{C}}} = \bar{\gamma}_1\,\mathbf{I} + \bar{\gamma}_2\,\bar{\mathbf{C}} + \bar{\gamma}_4\,\mathbf{a}_0 \otimes \mathbf{a}_0 + \bar{\gamma}_5\,(\mathbf{a}_0 \otimes \bar{\mathbf{C}}\,\mathbf{a}_0 + \mathbf{a}_0\bar{\mathbf{C}} \otimes \mathbf{a}_0).$$

In case of an isotropic material, the terms involving $\mathbf{a}_0$ are not needed. The prefactors are given by derivatives of the strain energy function with respect to the reduced invariants:

$$\bar{\gamma}_1 = 2\left(\frac{\partial \Psi_{\text{iso}}(\bar{I}_1, \bar{I}_2)}{\partial \bar{I}_1} + \bar{I}_1\,\frac{\partial \Psi_{\text{iso}}(\bar{I}_1, \bar{I}_2)}{\partial \bar{I}_2}\right), \quad \bar{\gamma}_2 = -2\frac{\partial \Psi_{\text{iso}}(\bar{I}_1, \bar{I}_2)}{\partial \bar{I}_2}, \quad \bar{\gamma}_4 = 2\frac{\partial \Psi_{\text{iso}}}{\partial \bar{I}_4}$$

$$\bar{\gamma}_5 = 2\frac{\partial \Psi_{\text{iso}}}{\partial \bar{I}_5}$$

Using the fourth order identity tensor $\mathbb{I}$ and the projection tensor $\mathbb{P}$,

$$(\mathbb{I})_{abcd} = \delta_{ac}\,\delta_{bd}, \qquad \mathbb{P} = \mathbb{I} - \frac{1}{3}\,\mathbf{C}^{-1} \otimes \mathbf{C},$$

the stress tensors can finally be computed as

$$\mathbf{S}_{\text{iso}} = J^{-2/3}\mathbb{P} : \bar{\mathbf{S}}, \qquad \mathbf{S}_{\text{vol}} = J\,p\,\mathbf{C}^{-1}, \qquad \mathbf{S} = \mathbf{S}_{\text{iso}} + \mathbf{S}_{\text{vol}}.$$

In the compressible case including the penalty method, the value of p, that is needed for $\mathbf{S}_{\text{vol}}$, is given by the constitutive model as $p = \mathrm{d}\Psi_{\text{vol}}(J)/\mathrm{d}J$. In the incompressible case, p

is the unknown Lagrange multiplier that gets computed as part of the numerical solution. In that case, p has the physical meaning of the hydrostatic pressure.

Using the present algorithm, the stress tensor $\mathbf{S}$ can, thus, be computed from derivatives of the strain energy function Ψ and the right Cauchy Green tensor $\mathbf{C}$, which can be calculated from the displacement field $\mathbf{u}$.

Another important quantity for the numerical solution of the nonlinear system is the fourth order elasticity tensor $\mathbb{C}$, which is defined as

$$\mathbb{C} = 2\frac{\partial \mathbf{S}(\mathbf{C})}{\partial \mathbf{C}} = 4\frac{\partial^2 \Psi(\mathbf{C})}{\partial \mathbf{C} \partial \mathbf{C}}\,.$$

It is the derivative of the stress tensor and is required in the Jacobian matrix of an iteration of the nonlinear Newton solver. Like the material tensor in Eq. (5.34), it shows major and minor symmetries and has 21 independent entries.

Like the stress tensor, the elasticity tensor is also additively composed into a volumetric term $\mathbb{C}_{\text{vol}}$ and an isochoric term $\mathbb{C}_{\text{iso}}$. The volumetric term can be computed by:

$$\mathbb{C}_{\text{vol}} = J\,\tilde{p}\,\mathbf{C}^{-1} \otimes \mathbf{C}^{-1} - 2J\,p\,\mathbf{C}^{-1} \odot \mathbf{C}^{-1}, \quad \left(\mathbf{C}^{-1} \odot \mathbf{C}^{-1}\right)_{abcd} = \frac{1}{2}\left(C^{-1}_{ac}\,C^{-1}_{bd} + C^{-1}_{ad}\,C^{-1}_{bc}\right).$$

The term includes two pressure variables $\tilde{p}$ and p. In the incompressible formulation, both variables equals the Lagrange multiplier p. For the compressible formulation, $\tilde{p}$ is derived as $\tilde{p} = p + J\,\mathrm{d}p/\mathrm{d}J$ and p is computed from the volumetric strain energy function as stated above.

The isochoric term $\mathbb{C}_{\text{iso}}$ of the elasticity tensor follows from the following algorithm listing the quantities to compute:

$$\bar{\delta}_1 = 4\left(\frac{\partial^2 \Psi_{\text{iso}}}{\partial \bar{I}_1\,\partial \bar{I}_1} + 2\,\bar{I}_1\frac{\partial^2 \Psi_{\text{iso}}}{\partial \bar{I}_1\,\partial \bar{I}_2} + \frac{\partial \Psi_{\text{iso}}}{\partial \bar{I}_2} + \bar{I}_1^2\,\frac{\partial^2 \Psi_{\text{iso}}}{\partial \bar{I}_2\,\partial \bar{I}_2}\right),\ \bar{\delta}_2 = -4\left(\frac{\partial^2 \Psi_{\text{iso}}}{\partial \bar{I}_1\,\partial \bar{I}_2} + \bar{I}_1\,\frac{\partial^2 \Psi_{\text{iso}}}{\partial \bar{I}_2\,\partial \bar{I}_2}\right),$$

$$\bar{\delta}_3 = 4\frac{\partial^2 \Psi_{\text{iso}}}{\partial \bar{I}_2\,\partial \bar{I}_2}, \quad \bar{\delta}_4 = -4\frac{\partial \Psi_{\text{iso}}}{\partial \bar{I}_2}, \quad \bar{\delta}_5 = 4\left(\frac{\partial^2 \Psi_{\text{iso}}}{\partial \bar{I}_1\,\partial \bar{I}_4} + \bar{I}_1\frac{\partial^2 \Psi_{\text{iso}}}{\partial \bar{I}_2\,\partial \bar{I}_4}\right),$$

$$\bar{\delta}_6 = -4\frac{\partial^2 \Psi_{\text{iso}}}{\partial \bar{I}_2\,\partial \bar{I}_4}, \quad \bar{\delta}_7 = 4\frac{\partial^2 \Psi_{\text{iso}}}{\partial \bar{I}_4\,\partial \bar{I}_4}, \quad \mathbb{I}_{abcd} = \delta_{ac}\,\delta_{bd}, \quad \bar{\mathbb{I}}_{abcd} = \delta_{ad}\,\delta_{bc}, \quad \mathbb{S} = (\mathbb{I} + \bar{\mathbb{I}})/2,$$

$$\frac{\partial \bar{I}_5}{\partial \bar{\mathbf{C}}} = \mathbf{a}_0 \otimes \bar{\mathbf{C}}\,\mathbf{a}_0 + \mathbf{a}_0\,\bar{\mathbf{C}} \otimes \mathbf{a}_0, \qquad \frac{\partial^2 \bar{I}_5}{\partial \bar{\mathbf{C}} \partial \bar{\mathbf{C}}} = \frac{\partial}{\partial \bar{\mathbf{C}}}\,(\mathbf{a}_0 \otimes \bar{\mathbf{C}}\,\mathbf{a}_0 + \mathbf{a}_0\,\bar{\mathbf{C}} \otimes \mathbf{a}_0),$$

$$\bar{\mathbb{C}} = J^{-4/3}\Big(\bar{\delta}_1\, \mathbf{I}\otimes\mathbf{I} + \bar{\delta}_2\left(\mathbf{I}\otimes\bar{\mathbf{C}} + \bar{\mathbf{C}}\otimes\mathbf{I}\right) + \bar{\delta}_3\bar{\mathbf{C}}\otimes\bar{\mathbf{C}} + \bar{\delta}_4\,\mathbb{S} + \bar{\delta}_5\left(\mathbf{I}\otimes\mathbf{a}_0\otimes\mathbf{a}_0 + \mathbf{a}_0\otimes\mathbf{a}_0\otimes\mathbf{I}\right)$$

$$+ \bar{\delta}_6\left(\bar{\mathbf{C}}\otimes\mathbf{a}_0\otimes\mathbf{a}_0 + \mathbf{a}_0\otimes\mathbf{a}_0\otimes\bar{\mathbf{C}}\right) + \bar{\delta}_7\left(\mathbf{a}_0\otimes\mathbf{a}_0\otimes\mathbf{a}_0\otimes\mathbf{a}_0\right)$$

$$+ \bar{\delta}_8\left(\mathbf{I}\otimes\frac{\partial\bar{I}_5}{\partial\bar{\mathbf{C}}} + \frac{\partial\bar{I}_5}{\partial\bar{\mathbf{C}}}\otimes\mathbf{I}\right) + \bar{\delta}_9\left(\bar{\mathbf{C}}\otimes\frac{\partial\bar{I}_5}{\partial\bar{\mathbf{C}}} + \frac{\partial\bar{I}_5}{\partial\bar{\mathbf{C}}}\otimes\bar{\mathbf{C}}\right) + \bar{\delta}_{10}\left(\frac{\partial\bar{I}_5}{\partial\bar{\mathbf{C}}}\otimes\frac{\partial\bar{I}_5}{\partial\bar{\mathbf{C}}}\right)$$

$$+ \bar{\delta}_{11}\left(\mathbf{a}_0\otimes\mathbf{a}_0\otimes\frac{\partial\bar{I}_5}{\partial\bar{\mathbf{C}}} + \frac{\partial\bar{I}_5}{\partial\bar{\mathbf{C}}}\otimes\mathbf{a}_0\otimes\mathbf{a}_0\right) + \bar{\delta}_{12}\frac{\partial^2\bar{I}_5}{\partial\bar{\mathbf{C}}\partial\bar{\mathbf{C}}}\Big)$$

$$\tilde{\mathbb{P}} = \mathbf{C}^{-1}\odot\mathbf{C}^{-1} - \frac{1}{3}\mathbf{C}^{-1}\otimes\mathbf{C}^{-1}$$

$$\mathbb{C}_{\text{iso}} = \mathbb{P} : \bar{\mathbb{C}} : \mathbb{P}^{\top} + \frac{2}{3}J^{-2/3}\bar{\mathbf{S}} : \mathbf{C}\,\tilde{\mathbb{P}} - \frac{2}{3}\left(\mathbf{C}^{-1}\otimes\mathbf{S}_{\text{iso}} + \mathbf{S}_{\text{iso}}\otimes\mathbf{C}^{-1}\right)$$

Then, $\mathbb{C} = \mathbb{C}_{\text{vol}} + \mathbb{C}_{\text{iso}}$ can be calculated.

5.4.5 Nonlinear Solver for the Solid Mechanics Model

The governing nonlinear system of equations is solved by a Newton scheme. We define the vector of the unknown degrees of freedom as $(\hat{\mathbf{u}}, \hat{p}) =: \mathbf{z}$. Then, the nonlinear equation takes the general form $\mathbf{W}(\mathbf{z}) = 0$. By linearization around a value $\mathbf{z}$, we get

$$\mathbf{W}(\mathbf{z}+\Delta\mathbf{z}) = \mathbf{W}(\mathbf{z}) + \mathbf{J}\,\Delta\mathbf{z} + o(\mathbf{z}+\Delta\mathbf{z}),$$

with the increment $\Delta\mathbf{z} = (\Delta\hat{\mathbf{u}}, \Delta\hat{p})$ and the Jacobian matrix $\mathbf{J} = \partial\mathbf{W}/\partial\mathbf{z}$. Neglecting the sublinear error term $o(z+\Delta z)$, we can start from an initial guess $\mathbf{z}^{(0)}$ and proceed to find the root of $\mathbf{W}$ using the following iterative Newton scheme:

$$\mathbf{J}\,\Delta\mathbf{z}^{(n)} = -\mathbf{W}(\mathbf{z}^{(n)}), \tag{5.86a}$$

$$\mathbf{z}^{(n+1)} = \mathbf{z}^{(n)} + \Delta\mathbf{z}^{(n)}. \tag{5.86b}$$

Equation (5.86a) is a linear system of equations with the system matrix given by $\mathbf{J}$, which has to be solved in every iteration step n. The linear system of equations can be expressed

as follows:

$$\begin{bmatrix} \mathbf{k}_{\delta\mathbf{u},\Delta\mathbf{u}} & \mathbf{k}_{\delta p,\Delta\mathbf{u}}^\top \\ \mathbf{k}_{\delta p,\Delta\mathbf{u}} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \Delta\hat{\mathbf{u}} \\ \Delta\hat{p} \end{bmatrix} = \begin{bmatrix} -\mathbf{R}_{\delta\mathbf{u}} \\ -\mathbf{R}_{\delta p} \end{bmatrix}. \tag{5.87}$$

The definition of the right-hand sides $\mathbf{R}_{\delta\mathbf{u}} = \delta W_{\text{int}} - \delta W_{\text{ext}}$ and $\mathbf{R}_{\delta p} = D_{\delta p}\Pi_L$ is given in Eq. (5.82). The system matrix is composed as follows. The upper left part consists of 3 times 3 blocks of submatrices, each with size $N_u \times N_u$ and the entries given by:

$$\mathbf{k}_{\delta\mathbf{u},\Delta\mathbf{u},(L,a),(M,b)} = \int_\Omega \phi^L_{(a),B} \tilde{k}_{abBD} \phi^M_{(b),D}\,\mathrm{d}V \quad \text{with} \quad \tilde{k}_{abBD} = \delta_{ab}\,S_{BD} + F_{aA}\,F_{bC}\,\mathbb{C}_{ABCD}.$$

Here, S_{BD} and $\mathbb{C}_{ABCD}$ are entries of the second Piola-Kirchhoff stress tensor $\mathbf{S}$ and the elasticity tensor $\mathbb{C}$. The computation of these terms uses the description in Sec. 5.4.4.

The lower left part of the system matrix in Eq. (5.87) is given by 1 times 3 blocks of submatrices, each with size $N_p \times N_u$ and entries given by:

$$\mathbf{k}_{\delta p,\Delta\mathbf{u},L,(M,a)} = \int_\Omega J\,\psi^L\,(F^{-1})_{Ba}\,\phi^M_{(a),B}\,\mathrm{d}V.$$

The upper right part equals the transposed lower left block such that the system matrix is symmetric. Solving the system in Eq. (5.87) in every iteration of the Newton scheme in Eq. (5.86) converges to the solution of the static solid mechanics problem.

5.4.6 Discretization and Solution of the Dynamic Hyperelastic Model

The dynamic model is given by the system of nonlinear equations in Eq. (5.88). In addition to the spatial discretization with finite elements, we need to discretize the temporal derivatives of the displacement field $\mathbf{u}$ and the velocity field $\mathbf{v}$. The time derivatives are discretized to timesteps $t = i \cdot dt$ with an implicit Euler scheme:

$$\dot{\mathbf{u}} \rightsquigarrow \frac{1}{dt}(\mathbf{u}^{(i+1)} - \mathbf{u}^{(i)}), \qquad \dot{\mathbf{v}} \rightsquigarrow \frac{1}{dt}(\mathbf{v}^{(i+1)} - \mathbf{v}^{(i)}).$$

Because of the added inertial body force, the external virtual work now depends on the vector of unknowns. In consequence, we split the external virtual work δW_{ext} into a

dead part $\delta W_{\text{ext,dead}}$ that solely depends on external forces and an inertial part:

$$\delta W_{\text{ext}} = \delta W_{\text{ext,dead}} + \int_\Omega \rho_0 \, \frac{v_{(a)}^{(i+1),L} - v_{(a)}^{(i),L}}{dt} \, \phi_{(a)}^L \, \phi_{(a)}^M \, \delta \hat{u}_a^M \, \mathrm{d}V = 0.$$

In summary, the system of equations to proceed from timestep i to $(i+1)$ is given as:

$$\delta W_{\text{int}}(\mathbf{u}^{(i+1)}, p^{(i+1)}) - \delta W_{\text{ext}}(\mathbf{v}^{(i)}, \mathbf{v}^{(i+1)}) = 0 \qquad \forall\, \delta \mathbf{u}, \tag{5.88a}$$

$$\frac{1}{dt}(\mathbf{u}^{(i+1)} - \mathbf{u}^{(i)}) - \mathbf{v}^{(i+1)} = 0, \tag{5.88b}$$

$$D_{\delta p} \Pi_L(\mathbf{u}^{(i+1)}) = 0 \qquad \forall\, \delta p. \tag{5.88c}$$

Here, Eq. (5.88a) is the principle of virtual work, Eq. (5.88b) relates displacements $\mathbf{u}$ and velocities $\mathbf{v}$ and Eq. (5.88c) is the incompressibility constraint.

The system is again solved using the Newton scheme presented in Sec. 5.4.5. The linear system for each Newton iteration takes the following form:

$$\begin{bmatrix} \mathbf{k}_{\delta \mathbf{u}, \Delta \mathbf{u}} & \mathbf{l}_{\delta \mathbf{u}, \Delta \mathbf{v}} & \mathbf{k}_{\delta p, \Delta \mathbf{u}}^\top \\ \mathbf{l}_{\delta \mathbf{v}, \Delta \mathbf{u}} & \mathbf{l}_{\delta \mathbf{v}, \Delta \mathbf{v}} & \mathbf{0} \\ \mathbf{k}_{\delta p, \Delta \mathbf{u}} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \Delta \hat{\mathbf{u}} \\ \Delta \hat{\mathbf{v}} \\ \Delta \hat{p} \end{bmatrix} = \begin{bmatrix} -\mathbf{R}_{\delta \mathbf{u}} \\ -\mathbf{R}_{\delta \mathbf{v}} \\ -\mathbf{R}_{\delta p} \end{bmatrix}.$$

The entries $\mathbf{k}_{\delta \mathbf{u}, \Delta \mathbf{u}}$ and $\mathbf{k}_{\delta p, \Delta \mathbf{u}}$ are the same as in the static case in Eq. (5.87). The other non-zero entries are given by

$$\mathbf{l}_{\delta \mathbf{u}, \Delta \mathbf{v}, (L,a),(M,b)} = \frac{1}{dt} \, \delta_{ab} \int_\Omega \rho_0 \, \phi_{(b)}^M \, \phi_{(a)}^L \, \mathrm{d}V, \qquad \mathbf{l}_{\delta \mathbf{v}, \Delta \mathbf{u}, (L,a),(M,b)} = \frac{1}{dt} \, \delta_{ab} \, \delta^{LM},$$

$$\mathbf{l}_{\delta \mathbf{v}, \Delta \mathbf{v}, (L,a),(M,b)} = -\delta_{ab} \, \delta^{LM}.$$

Note that in the dynamic problem, the system matrix is unsymmetric. It would be symmetric if the entries $\mathbf{l}_{\delta \mathbf{u}, \Delta \mathbf{v}}$ and $\mathbf{l}_{\delta \mathbf{v}, \Delta \mathbf{u}}^\top$ were the same. This would be the case for a density of one, $\rho_0 = 1$, and if the term $\int_\Omega \phi_b^M \phi_a^L \, \mathrm{d}V$ would be replaced by $\delta_{ab} \delta^{LM}$. The second condition means that a lumped mass matrix would be used where the diagonal entries are set to the row sums of the original matrix.

We discretize the finite element solution in space by *Taylor-Hood* elements. This type of element uses quadratic ansatz functions ϕ for the displacements and velocities and

linear ansatz functions ψ for the Lagrange multiplier or hydrostatic pressure p on a 3D hexahedral mesh. This choice was proven to exhibit no locking [Zie05]. Locking is a phenomenon of degraded convergence of the finite element method for solid mechanics problems and occurs for improper discretization schemes.

For a compressible material, the incompressibility constraint which is the last equation in the systems Eq. (5.82) or Eq. (5.88) is removed. Instead of solving for the pressure p as a Lagrange multiplier, the value is given by the constitutive model as described in Sec. 5.2.6. In consequence, the system matrix of the linear system of equations that is solved in the Newton iterations has a smaller size for compressible materials.

Moreover, the size varies depending on whether the static or the dynamic problem given in Sections 5.4.5 and 5.4.6 is solved. Assuming a linear mesh with N_p degrees of freedom and a quadratic mesh with N_u degrees of freedom, the square system matrix has $3\,N_u$ rows and columns for a static compressible formulation, $3\,N_u + N_p$ for a static incompressible formulation, $6\,N_u$ for a dynamic compressible model, and $6\,N_u + N_p$ for a dynamic incompressible model.

In any case, the mechanics model can be linked to the subcellular model by defining the active stress as given in Eq. (5.42). Since the active stress does not depend directly on the passive behavior, the active stress term can be added as a constant to the passive stress term. This constant also has no influence on the Jacobian matrix $\mathbf{J}$. As the subcellular model depends on the fiber stretch $\lambda_f = \sqrt{I_4}$, there is a feedback loop between the subcellular and the solid mechanics model.

Details on the connection to the subcellular model as well as details on the numerical solution schemes are given in Sec. 7.7.

Chapter 6

Usage of the Software OpenDiHu

OpenDiHu is an open source software framework for static and dynamic multi-physics problems that can be solved with the finite element method. It was developed essentially by the author as part of this work with some code contributions given by Aaron Krämer, Nehzat Emamy and Felix Huber from the *Institute for Parallel and Distributed Systems* and the *Institute of Applied Analysis and Numerical Simulation*. We use it to simulate the multi-scale models presented in the last chapter: biophysical problems describing biomechanics and neurophysiology of the musculoskeletal system.

In this chapter, we introduce basic concepts and present details on the usage of our software. The next chapter Chap. 7 continues with a discussion of internal software aspects and gives details on how the different algorithms and solvers are implemented.

We begin this chapter with an explanation of the design goals in Sec. 6.1. Next, Sec. 6.2 showcases how to set up simulation programs for various scenarios based on examples. Some biochemical models are conveniently formulated and shared in the bioengineering community using the CellML description language [Cue03]. Section 6.3 gives more details on the usage of CellML models in our software. Finally, Sec. 6.4 discusses various output formats and compatible tools for visualization.

6.1 Design Goals

Simulations of complex multi-scale models require the combination of tailored numerical solution schemes. Spatial mesh resolutions and timestep widths should be chosen carefully to avoid instabilities and to allow the completion of useful simulation time spans in feasible runtimes. Numerical solvers on 1D, 2D and 3D meshes have to be coupled and the data should be mapped between these meshes.

The simulations should be parallelized to efficiently exploit today's hardware and reduce the runtime to a minimum. Parallel runs should be performant on small workstations, on larger compute servers and on supercomputers.

Scenarios for different use cases should be possible, ranging from convenient debugging with simple physics and small problem sizes to large runs with highly resolved meshes and comprehensive models. Schemes for input and output of the data should be available for all of these use cases. Established community standards for models, such as the CellML description, and output file formats should be considered. The configuration of models and solver parameters should be flexible, well organized and properly documented to allow an efficient workflow.

In addition to this feature list from a user perspective, further requirements can be formulated from a developer perspective. The program code should be modular, structured and well documented to allow discovery, reuse and extension in the future. The implemented solvers should compute correct results, which should be testable to preserve correctness during code changes.

OpenDiHu aims to fulfill these requirements. The name originates from the Digital Human project that aimed to advance the field of biomechanics by "providing new possibilities to improve the understanding of the neuromuscular system by switching from small-sized cluster model problems to realistic simulations on HPC clusters" [Röh17]. The software framework contributes to this goal.

In the following, we concretize the requirements and formulate design goals to guide the software development. The design goals can be summarized under the keywords *usability*, *performance* and *extensibility*. They span the field of requirements from user-centric properties to developer centric properties.

6.1.1 Usability

Usability is defined in ISO 9241-11 [ISO18] as "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use." We target at users with a basic understanding of biophysics, numerics, programming and command line usage in Linux. The specified goals include—in increasing complexity—to reproduce results of existing studies, analyze the simulation results, adjust parameters of existing simulations to achieve different model behavior,

conduct studies over a set of different parameters, exchange numerical schemes to improve stability or efficiency, combine implemented parts of models to a new multi-physics model, and implement new solvers for completely new physics. The context of use lies in scientific and educational studies.

We base the usage of the framework on command line programs and scripts and do not include a graphical user interface (GUI). A GUI would need to present an abstract, simplified layer of the simulation setup, that reduces the understanding of the actual process. Furthermore, it would be difficult to keep a GUI up to date with all functionality that gets implemented in the software over time. The advantage of a command-line-only-program is, that it can be easily used in automated studies with different parameter combinations. Furthermore, it simplifies usage on remote computers such as compute clusters and supercomputers.

In this context, good usability is ensured by using the Python programming language for the configuration of the simulation. The computational code of every simulation is written in C++ and compiled to a hardware specific executable, which enables good computational performance. The user can configure all parameters using Python scripting. The Python3 interpreter is linked into the C++ program such that the configuration script can be parsed at runtime when the simulation program is executed.

Thus, users can organize the simulation settings using their own variable names. Users can compute derived parameter values within the settings script, they have the flexibility to organize the settings in multiple files, and define own command line parameters for every example. Input data and results of the simulation can be preprocessed and post-processed directly in the Python settings script.

6.1.2 Perfomance

The second design goal is to achieve good performance. OpenDiHu satisfies this goal by supporting parallel execution on the one hand and by providing efficient algorithms on the other hand.

Simulations can be run on distributed and shared memory systems. The computational domains are mainly discretized with structured meshes, that can easily be partitioned into subdomains for multiple processes. For large scale simulations on multiple cores, the input data such as node positions can be specified in a distributed way. Hence, every

process only needs to know its own portion of the whole simulation and the total amount of data can exceed the storage capabilities of a single compute core.

Efficient algorithms involve efficient numerical solvers such as multigrid or conjugate gradient schemes and optimized data handling within the software framework. We use the external library *PETSc* [Bal97] for the parallel storage of vectors and matrices and for linear algebra operations. PETSc provides a large collection of preconditioners, linear and nonlinear solvers that can be chosen at runtime. At the same time, it offers low-level access to the locally stored data, which, e.g., allows us to optimize data transfer between different arrays.

For multi-scale models, good performance can only be achieved, when the data transfer between different solvers is also efficient. Using profiling tools, runtime hotspots in various simulation programs were regularly identified and evaluated. The portions of code, that use most runtime should be the actual computations and not memory allocations or data transfer operations. Using these insights, the framework was efficiently constructed, e.g., by avoiding repeated memory initialization and expensive copy operations whenever possible.

6.1.3 Extensibility

By extensibility, we refer to the possibility to add solvers for new physical processes to the existing framework. This is facilitated when existing components of the framework are documented and can be reused.

On the highest level, existing simulation programs using models in the CellML format can be altered to solve different physics by exchanging the CellML model. For example, model extensions of the active mechanical behavior of the half-sarcomeres can be implemented in the corresponding CellML model and without changing the C++ code.

It is also possible to use the adapters in OpenDiHu for the numerical coupling library *preCICE* [Bun16] to numerically couple software packages to OpenDiHu. The surface coupling adapter allows coupling external mechanics solvers and exchange displacements and traction forces. The volume coupling adapter allows, e.g., to use the electrophysiology solver in OpenDiHu and couple it with external models of the muscle or other organs.

On the next level, which still does not require changing the C++ core, the modular building blocks of model solvers such as timestepping schemes for the solution of ODEs, operator splittings or coupling schemes can be newly combined for different behavior.

Solving other models, for which no solver has been designed, involves adding new code to the software framework. The solvers in OpenDiHu use structures like function spaces consisting of meshes and basis functions, linear system solvers and output writers for data output, which are self-contained and get reused at different locations in the framework. A completely new model, e.g., an electro-magnetic description of electrophysiology would require a dedicated new solver class. Two template classes exist in OpenDiHu that can be copied and adjusted to create such a new solver for either transient or static problems.

Polymorphism concepts of the C++ programming language such as object orientation (OO) and template meta-programming (TMP) allow writing generic algorithmic code that gets specialized for the particular use-cases at runtime (OO) or at compile time (TMP). For example, most of the solvers are independent of the type of mesh they operate on. Similarly, an explicit timestepping scheme has the same definition regardless whether it solves a 0D subcellular model with a high-dimensional state vector or a 3D linear elasticity formulation discretized by finite elements, where the solution is a vector field with three components. These concepts help to reuse existing structures in OpenDiHu.

6.2 Usage of OpenDiHu

We begin with aspects of OpenDiHu that are relevant from a user's perspective. This section outlines the basic organization of the repository in Sec. 6.2.1, the installation procedure in Sec. 6.2.2 and demonstrates the usage of given example scenarios in Sections 6.2.3, 6.2.4, and 6.2.6. Section 6.2.7 summarizes all available solver classes.

6.2.1 Organization of the Repository

The complete OpenDiHu is contained in a single git repository which is hosted on GitHub at `https://github.com/maierbn/opendihu`. In addition, the documentation is hosted on the "Read the Docs" website under `https://opendihu.readthedocs.io` [Mai21c]. This documentation is split into a user and a developer documentation. The user part includes introductory information such as installation instructions, a description of most of the existing simulation scenarios with images of the results, instructions how to build and run them and a complete reference of the settings of all available solvers.

After cloning the git repository, the directory has the contents shown in Fig. 6.1, which will be explained in the following. The software consists of a core library, that provides

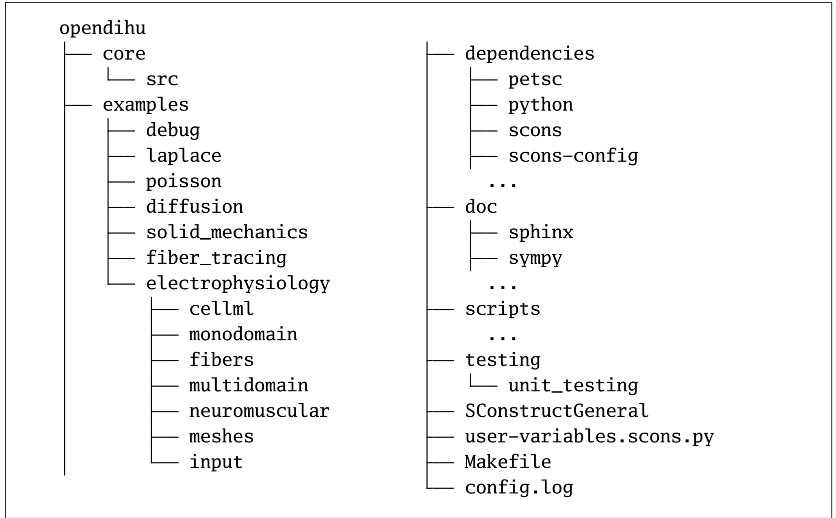


Figure 6.1: Contents of the main opendihu directory.

all functionality such as solvers and data handling. In addition, examples are created, that set up specific simulation scenarios and import the required solvers by linking to the core library.

The subdirectory `core/src` contains all C++ code that is compiled into the core library. This source code consists of approximately 90 000 code lines, 24 000 blank lines and 19 000 comment lines contained in approximately 700 files and structured in a directory tree with approximately 70 total subdirectories.

The `examples` directory contains all simulation scenarios that are packaged with OpenDiHu. Each of the approximately 65 examples demonstrates how to solve a different model, often in several variations with different parameters and numerical schemes. The examples are grouped by the subdirectories shown in Fig. 6.1 in different categories: technical examples for debugging, scenarios for solving the Laplace, Poisson and diffusion equations, various solid mechanics models, fiber tracing examples that can be used to generate meshes as described in Chap. 3, and the electrophysiology models. The electrophysiology examples are further structured as given in Fig. 6.1 with increasing complexity: subcellular CellML model solvers (0D) in the subdirectory `cellml`, solvers for the monodomain equation (1D), i.e., electrophysiology on a single fiber in `monodomain`, models with multiple 1D fibers also coupled with the 3D EMG model or muscle contraction

model in `fibers`, the same but with the multidomain model in `multidomain`, and models of motor neurons coupled with fibers and multidomain models in the `neuromuscular` directory.

The directory `meshes` contains scripts and raw data to generate all meshes needed by the simulations. The directory `input` collects all input files that are used in any example, e.g., cellml models, meshes, and text files that specify MU assignments and firing times. Because this directory contains large files, it is not included in the git repository but hosted on a separate file server.

The `dependencies` directory contains the source files and installations of all external packages, such as PETSc in `petsc`, the Python3 interpreter in `python`, and the SCons related packages in `scons` and `scons-config`. This directory will be automatically filled with more subdirectories during the installation procedure.

The `doc` directory collects various documents and mathematical derivations that help to understand certain solvers. For example, the directory `doc/sphinx` contains the whole online documentation, which is hosted on the "Read The Docs" website [Mai21c] and built using the reStructuredText markup language and the Sphinx generation system. The `doc/sympy` directory contains Python scripts with the derivation of various equations using the symbolic math package SymPy.

Various utility Python scripts are stored in the directory `scripts`. Users should add this directory to the `$PATH` environment variable in their system such that the scripts can be invoked from the command prompt. For example, the `catpy` and `plot` scripts list and visualize Python-based output files of simulations, other scripts can be used to inspect and manipulate binary mesh files.

The directory `testing/unit_testing` contains the code for all unit tests. Furthermore, the files that exist directly under the top level `opendihu` directory are relevant for the build system, which will be explained in the next section.

6.2.2 Installation

The installation procedure involves three steps: First, the dependencies, i.e., all required external packages have to be located. Second, the OpenDiHu core library is compiled and linked. Third and optionally, unit tests are compiled, linked and executed.

The first step consists of finding the location of each dependency, determining the corresponding header and library files that are needed for inclusion and linking, and potentially determining special compiler or linker flags. The step can be configured to fit the individual system setup and use case by taking into account already existing dependencies and enabling or disabling optional packages.

The second step compiles the source code and links it to all dependencies that were collected in the first step. The result is a static library, which contains the functionality of OpenDiHu in executable form. To run a particular simulation, an additional program with a small source code file has to be written, compiled and linked to this library.

For the compilation of unit tests in the third step, a similar action is performed. The step builds and links three unit testing executables that are subsequently run with one, two and six processes and conduct various functional tests of the implementation.

Currently, the following fifteen dependencies are used with OpenDiHu: the standard for shared-memory parallelisation *MPI* [Gab04], the data handling and numerics library *PETSc* [Bal97; Bal15; Bal16], the *Python interpreter* [Van09], a set of *Python packages* including *NumPy* [Har20], *SciPy* [Vir20] and *Matplotlib* [Hun07] among others, a *Base64* compression library [Kis20], the unit testing framework *Googletest* [Goo20], the compile-time differentiation toolbox *SEMT* [Gut04; Gut12], the parallel file I/O library *ADIOS2* [God20], the vectorization toolbox *Vc* [Kre12; Kre15] and its newer version *std-simd* [Hob19], the library for parallel time integration with multigrid *XBraid* [XBr20], the solver and converter for CellML models *OpenCOR* [Gar15], the XML parser *libxml2* [Vei21], the coupling library *preCICE* [Bun16] and the logging library *Easylogging++* [Ser21]. The build system has to install these dependencies and possibly cope with different sets of available versions and prerequisites.

Popular build systems exist that facilitate the three mentioned installation step, e.g., CMake, GNU Autoconf and SCons. CMake uses a three-step process of configuration, generation and building, which requires users to have the corresponding know-how. Autoconf creates a configure script that relies on command line options instead of a configuration file for all settings.

Considering the usability goal for OpenDiHu, we chose the build system SCons as it requires little previous knowledge and can read its settings from a Python based configuration file. SCons performs the three steps of the installation procedure by a single command. Packages that are not yet installed are downloaded and installed automatically, using transparent bash commands. Furthermore, the SCons build system itself is packaged along with our software and, thus, no additional installation steps are required

to begin the build process (apart from checking that some essential packages such as a compiler and an MPI implementation are available).

SCons allows to both specify the installation configuration and extend the functionality by using the Python scripting language. Based on the scons-config package [Hod13], we added functionality to detect and automatically download the dependencies that are required for OpenDiHu. For some dependencies, multiple versions are tried if the first attempt fails, e.g., for Python and PETSc. This adds robustness for different systems and typically allows to set up OpenDiHu on a new Linux computer by only executing a single "`scons`" command.

The top-level files listed in Fig. 6.1 are related to the build system. The file `SConstruct General` contains Python code that defines various flags for the usual build targets: the `release` targets creates optimized and hardware-specific binaries, the `debug` target disables optimization and adds debugging symbols to the executables, other debugging targets produce intermediate outputs after the preprocessing, assembly or optimization stages. All options are documented in the help text of the build system.

The `user-variables.scons.py` file contains the configuration and can be adjusted by the user to enable or disable certain packages or features. The `Makefile` contains convenient shortcuts for longer build command, e.g., the command "`make`" builds the debug and release targets and runs the unit tests. During installation, all text output and progress information is appended to the log file `config.log`. If the installation fails, this file contains all information that is required to track down the respective issue.

After the installation and build step of the core library, individual simulation scenarios can be developed and executed. The place for the code of these simulations is in the `examples` subdirectory, where numerous predefined simulation scenarios are given. In the following, we demonstrate how to use OpenDiHu and, more specifically, we present the structure and configuration of a simulation program by considering three of these examples in increasing complexity: a Laplace problem in Sec. 6.2.3, a simulation of muscle contraction in Sec. 6.2.4 and a simulation of the neuromuscular system in Sec. 6.2.6.

6.2.3 Exemplary Usage: Laplace Problem

Every simulation consists of a single C++ source file that gets compiled to an executable program and a Python file that defines all settings for the simulation. The example considered in the following solves a 2D Laplace problem and is located in the directory `examples`

/laplace/laplace2d. The considered C++ source file is src/laplace_structured.cpp and the corresponding Python settings file is settings_lagrange_quadratic.py. The contents of the two files are listed in Figures 6.2 and 6.3. The directory additionally contains code for other scenarios that have different parameters and discretizations.

After compilation by the "scons" command, an executable is created in the build_release subdirectory of the example. In this directory, the simulation can be run with the following command:

```
./laplace_structured ../settings_lagrange_quadratic.py
```

Here, the first item is the program name. We pass the filename of the settings file as the first command line argument.

Figure 6.2 lists the full C++ source code of this example. Line 2 includes the main header file of opendihu, which makes all functionality available. The rest of the source file contains the definition of the main function. Line 9 defines the context object settings, which uses the command line arguments given by argc and argv. This line invokes the Python interpreter on the Python settings file and stores all parameters in the settings object.

Lines 12 to 17 define an object named equationDiscretized, which is of the FiniteElementMethod class located in the SpatialDiscretization namespace. The new object uses the settings object that was defined before.

The FiniteElementMethod class takes several class template arguments enclosed in angle brackets. The first in line 13 specifies the mesh type, which, in this 2D example, is a structured deformable mesh of dimension two. Furthermore, line 14 specifies quadratic Lagrange basis functions and line 15 specifies Gauss quadrature with three Gauss points per dimension. The argument in line 16 defines the equation that is discretized by this finite element method class, which, in this case, is the static Laplace equation $\Delta \mathbf{u} = 0$.

In line 20, the solver is executed, performs the computation and writes the configured output files. The program finally returns in line 22.

The problem to be solved is parametrized by the settings file settings_lagrange_quadratic.py, which is listed in Fig. 6.3. The code in this file can use all features of the Python scripting language. For example, in line 5, the *NumPy* numerics packages is imported and its sine function is used in lines 11 and 14. The print statement in line 16 is executed in the for loop in line 7 and produces informational output about boundary condition values during execution.

```
1  #include <cstdlib>
2  #include "opendihu.h"
3
4  int main(int argc, char *argv[])
5  {
6    // 2D Laplace equation 0 = du^2/dx^2 + du^2/dy^2
7
8    // initialize and parse settings from input file
9    DihuContext settings(argc, argv);
10
11   // define the tree of solvers (here only one FEM solver)
12   SpatialDiscretization::FiniteElementMethod<
13     Mesh::StructuredDeformableOfDimension<2>,
14     BasisFunction::LagrangeOfOrder<2>,
15     Quadrature::Gauss<3>,
16     Equation::Static::Laplace
17   > equationDiscretized(settings);
18
19   // run the simulation
20   equationDiscretized.run();
21
22   return EXIT_SUCCESS;
23 }
```

Figure 6.2: Example source file of an OpenDiHu solver for the 2D Laplace problem.

The settings file has to define the variable `config` to be a Python dictionary, i.e., an associative container data structure. This dictionary contains the parameter names and values that are required by the solver in the C++ program. In Fig. 6.3, the `config` dictionary is defined in lines 18 to 52. It contains global options such as filenames of log files in lines 19 to 22 followed by specific options for the finite element method object in lines 24 to 50. The exact meaning of all parameters is documented in the online documentation [Mai21c] and also sketched by the comments in the file. Some parameters will be presented in the following.

The parameter set consists of mesh parameters in lines 25 to 28, problem parameters in lines 31 to 34, solver parameters in lines 37 to 44 and output writers in lines 47 to 50. The mesh in this scenario is a cartesian grid on the unit square. The number of elements is specified by the parameter `"nElements"`. The number of elements in x and y-directions is given by the variables `nx` and `ny`, which are defined in line 1 of the settings file.

The parameter `"inputMeshIsGlobal"` is relevant for parallel execution. Its value spec-

ifies, whether all parameters apply to the global problem (`True`) or to a local subdomain (`False`). If the given example is executed by four processes, the mesh will have 10×10 elements, as specified by `nx` and `ny`. However, if `"inputMeshIsGlobal"` is set to `False`, each of the 2×2 subdomains would create a mesh of this size, yielding a total mesh of 20×20 elements. Instead of the Cartesian grid, it is also possible to define the node positions of every element. Then, it is beneficial to only specify the data for the own subdomain on each process for meshes with large numbers of elements and large numbers of processes.

This example problem uses Dirichlet boundary conditions. Values of a sine curve are prescribed at the boundaries $y = 0$ and $y = 1$ of the unit square. Line 31 of the settings file sets the boundary conditions to the variable `bc`. This variable is defined in the loop before the `config` dictionary in lines 6 to 16. The `bc` variable itself is a dictionary that specifies the prescribed values for every degree of freedom.

Lines 38 and 39 specify the employed preconditioner and solver by strings that are given to the solver library PETSc. Thus, all linear solvers available in PETSc can be used. Error tolerances on the residual norm and a maximum number of iterations can be specified. It is also possible to dump the system matrix, right-hand side and solution vectors to a text file or a MATLAB readable file using the options in lines 42 and 43.

Output of the simulation results is configured by specifying a list of output writers in lines 47 to 50. The considered example has the two output writers with formats `"Paraview"` and `"PythonFile"`. The former writes files that can be visualized by the software *ParaView*, the latter outputs files that can be easily parsed with a Python script. Both output writers either generate binary or human-readable files, depending on the `"binary"` option. Binary files have smaller file sizes and are used for large datasets. The human-readable text files make it is easier to debug the output.

After the program has been run, the `out` subdirectory contains the two output files created by the output writers. The Python based file can be visualized using the command “`plot`”, which is also provided by OpenDiHu. Figure 6.4 shows the resulting *Matplotlib* visualization. The figure shows that the Dirichlet boundary conditions for $y = 0$ and $y = 1$ are met and the solution is a harmonic function.

```
1  nx = 10;       ny = nx           # number of elements
2  mx = 2*nx + 1; my = 2*ny + 1     # number of nodes
3
4  # specify boundary conditions
5  import numpy as np
6  bc = {}
7  for i in range(mx):
8    x = i/mx
9
10   # bottom line
11   bc[i] = np.sin(x*np.pi)
12
13   # top line
14   bc[(my-1)*mx + i] = np.sin(x*np.pi)
15
16   print("{}, bc: {}, {}".format(i, bc[i], bc[(my-1)*mx + i]))
17
18 config = {
19   "solverStructureDiagramFile":     "solver_structure.txt",      # diagram file
20   "logFormat":                      "csv",     # "csv" or "json", format of log
21   "scenarioName":                   "laplace", # scenario name for log file
22   "mappingsBetweenMeshesLogFile":   None,      # a log file about mappings
23   "FiniteElementMethod": {
24     # mesh parameters
25     "nElements":                   [nx, ny],   # number of elements in x and y
26     "inputMeshIsGlobal":           True,       # if nElements is a global number
27     "physicalExtent":              [1.0, 1.0], # physical domain size
28     "physicalOffset":              [0, 0],     # physical location of origin
29
30     # problem parameters
31     "dirichletBoundaryConditions": bc,         # Dirichlet BC as dict
32     "dirichletOutputFilename":     None,       # output file for Dirichlet BC
33     "neumannBoundaryConditions":   [],         # Neumann BC
34     "prefactor":                   1,          # constant prefactor c in cΔu
35
36     # linear solver parameters
37     "solverType":                  "gmres",    # linear solver scheme
38     "preconditionerType":          "none",     # preconditioner scheme
39     "relativeTolerance":           1e-15,      # stopping criterion, rel. tol.
40     "absoluteTolerance":           1e-10,      # stopping criterion, abs. tol.
41     "maxIterations":               1e4,        # maximum number of iterations
42     "dumpFormat":                  "default",  # format for data dump
43     "dumpFilename":                None,       # filename for dump
44     "slotName":                    None,       # connector of solver
45
46     # output writers
47     "OutputWriter" : [
48       {"format": "Paraview",   "filename": "out/laplace", "binary": False},
49       {"format": "PythonFile", "filename": "out/laplace", "binary": False},
50     ]
51   }
52 }
```

Figure 6.3: Python settings file of the Laplace example corresponding to the source file in Fig. 6.2.

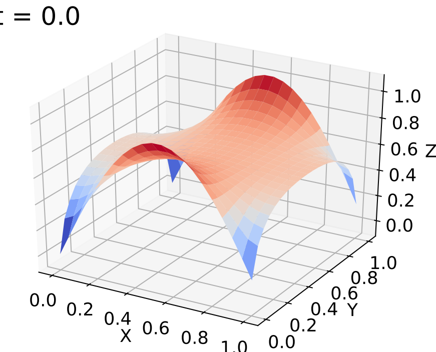


Figure 6.4: Visualization of the solution of the exemplary 2D Laplace problem. The plot was created using the `plot` command on the output of the `PythonFile` output writer.

6.2.4 Exemplary Usage: Multidomain Model With Solid Mechanics

The next example to be studied is a simulation of electrophysiology and muscle contraction. It uses the multidomain model on the muscle and body fat domains and bidirectionally couples the nonlinear solid mechanics model. The example is located in the directory `examples/electrophysiology/multidomain/multidomain_contraction`.

Figure 6.5 shows the source code of the C++ file. The overall structure of the code is the same as in the previous Laplace example: Line 2 includes the OpenDiHu header, the main function consists of the definition of a settings object in line 9, the definition of the solver in lines 13 to 48 and its execution in line 50. The only difference is the definition of the solver, which contains more nested class templates.

The problem is numerically solved by computing the multidomain model, transferring the activation parameter γ from the multidomain mesh to the elasticity mesh, computing the solid mechanics model, and then mapping the deformed geometry back to the multidomain mesh. This compute cycle repeats in every timestep. This coupling between two models is performed by the `Control::Coupling` class defined as the outer-most solver in line 13. It nests the two solvers of the model parts: The first is the class named

```
1  #include <cstdlib>
2  #include "opendihu.h"
3
4  int main(int argc, char *argv[])
5  {
6    // 3D multidomain coupled with contraction
7
8    // initialize everything, handle arguments and parse settings from input file
9    DihuContext settings(argc, argv);
10
11   typedef Mesh::StructuredDeformableOfDimension<3> MeshType;
12
13   Control::Coupling
14   <
15     OperatorSplitting::Strang<
16       Control::MultipleInstances<     // subcellular model
17         TimeSteppingScheme::Heun<
18           CellmlAdapter<
19             57,71,  // nStates,nAlgebraics: 57,71 = Shorten, 4,9 = Hodgkin Huxley
20             FunctionSpace::FunctionSpace<MeshType,BasisFunction::LagrangeOfOrder<1>>
21           >
22         >
23       >,
24       TimeSteppingScheme::MultidomainWithFatSolver<       // multidomain
25         SpatialDiscretization::FiniteElementMethod<       // FEM for initial potential flow
26           MeshType,
27           BasisFunction::LagrangeOfOrder<1>,
28           Quadrature::Gauss<3>,
29           Equation::Static::Laplace
30         >,
31         SpatialDiscretization::FiniteElementMethod<       // anisotropic conduction
32           MeshType,
33           BasisFunction::LagrangeOfOrder<1>,
34           Quadrature::Gauss<5>,
35           Equation::Dynamic::DirectionalDiffusion
36         >,
37         SpatialDiscretization::FiniteElementMethod<       // isotropic conduction in fat layer
38           MeshType,
39           BasisFunction::LagrangeOfOrder<1>,
40           Quadrature::Gauss<5>,
41           Equation::Dynamic::IsotropicDiffusion
42         >
43       >
44     >,
45     MuscleContractionSolver<          // solid mechanics
46       Mesh::CompositeOfDimension<3>
47     >
48   > problem(settings);
49
50   problem.run();
51
52   return EXIT_SUCCESS;
53 }
```

Figure 6.5: Source code of the simulation program that computes the multidomain model coupled with the solid mechanics model.

`OperatorSplitting::Strang` in line 15. It computes the multidomain electrophysiology model. The second class is the `MuscleContractionSolver` in line 45. It calls the solid mechanics solver and incorporates the activation and active stress term.

The multidomain model itself is computed using two coupled solvers. As formulated in Sec. 5.3, a Strang operator splitting is used that alternates between solving the subcellular model and the electric conduction part of the multidomain model. In the code, these two parts are defined in line 16 and line 24. As can be seen, the first part that solves the subcellular model consists of the three nested classes in lines 16 to 18. The inner-most is the `CellmlAdapter`, which loads and executes a DAE model description from a CellML file. Its template arguments are the number of states and number of algebraic variables in line 19 and the type of the function space in line 20 used for spatial discretization. The CellML model is solved by the enclosing Heun timestepping scheme in line 17. Because we need to solve the subcellular model for every compartment $k \in 1, \dots, N_{\text{MU}}$, a `MultipleInstances` class is used in line 16, which encloses the timestepping scheme and applies it on the domains for every compartment.

The second part of the multidomain model is the electric conduction in the intracellular, extracellular and body fat domains. It corresponds to solving the linear system of equations given in Sec. 5.3.5. This is done in OpenDiHu by the `MultidomainWithFatSolver` defined in lines 24 to 43. It can be seen, that it nests three classes of type `FiniteElementMethod`.

The first one in line 25 is used to initially solve a potential flow problem, from which the fiber direction can be estimated. This approach [Cho13] is also used in the fiber generation algorithms described in Sec. 3.4.6. As a result, we get the anisotropy direction in the 3D domain, which is needed to define the anisotropic intracellular conduction tensors $\boldsymbol{\sigma}_i^k$. As the problem to be solved is a Laplace problem, the equation to be discretized by the class is defined accordingly in line 29.

The second and third nested finite element classes are defined in lines 31 and 37. They define the isotropic electric conduction in the muscle domain and the anisotropic electric conduction in the fat domain and are used to set up the stiffness and mass matrices for these subproblems.

Several meshes are involved in the definition of this example. As described in Sec. 5.1.5, the computational domain consists of a muscle domain and a fat domain. Both domains are discretized by a 3D structured mesh, which is given as `MeshType` in line 11. This type is referenced for the subcellular model in line 20 and for the conduction parts in lines 26, 32, and 38. For the muscle contraction solver in line 46, we use a different, “composite”

```
1  config = {
2    "scenarioName": "multidomain_contraction",
3    "Solvers": {
4      "potentialFlowSolver": {...},
5    },
6    "Coupling": {
7      "timeStepWidth": 1e-3,
8      "Term1": {            # multidomain
9        "StrangSplitting": {
10
11         "Term1": {        # subcellular model
12           "MultipleInstances": {
13             "nInstances": variables.n_compartments,
14             "instances": [        # settings for each motor unit
15             {
16               "ranks": list(range(n_ranks)),
17               "Heun": {
18                 "CellML" : {
19                 }
20               }
21             } for compartment_no in range(variables.n_compartments)]
22           },
23         },
24         "Term2": {        # conduction term of multidomain
25           "MultidomainSolver": {
26
27             "PotentialFlow": {
28               "FiniteElementMethod": {
29                 "solverName": "potentialFlowSolver",
30               },
31             },
32
33             "OutputWriter": [
34             ]
35           },
36         }
37       },
38     },
39     "Term2": {        # solid mechanics
40       "MuscleContractionSolver": {
41
42         # the actual solid mechanics solver
43         "DynamicHyperelasticitySolver": {
44          }
45       }
46     }
47   }
48 }
```

Figure 6.6: Excerpt of the settings file for the multidomain and solid mechanics solver.

type of mesh. This type is a combination of the two structured meshes for the body and for the fat domain, as the whole tissue should be considered in the computation of the deformation.

The Python settings script, that corresponds to the C++ source file, is given in Fig. 6.6. Only the main structure of the `config` dictionary is outlined and the details are left out. It can be seen, that the definition in this file has a hierarchical structure. It is the same

tree-like structure as in the C++ source.

The settings for the top-level coupling scheme start in line 6. First, some settings related to the timestepping scheme itself are specified of which one, the timestep width, is shown. Then, the settings of the two nested solvers are listed under `"Term1"` in line 8 and `"Term2"` in line 39. Similarly, also the nested Strang splitting scheme in line 9 defines its two nested solvers under `"Term1"` (line 11) and `"Term2"` (line 24).

Lines 12 to 22 define settings for the `MultipleInstances` class, which holds separate instances of the subcellular solver for all motor units. The number of instances is specified by the `"nInstances"` parameter. A list containing the particular parameters for each instance is given under the keyword `"instances"` in lines 14 to 21. This construct is a Python list comprehension, an inline definition of list entries defined by the for loop in line 21. The nested specifications of parameters of the Heun and the CellML methods, not shown in Fig. 6.6, depend on the iteration index `compartment_no` of this loop. Each of these instance gets computed by a defined set of processes, specified under the parameter `"ranks"` in line 16. In this multidomain example, all processes take part in the computation of all multidomain compartments and, thus, all instances of the `MultipleInstances` class are computed by all processes. The expression in line 16 expands to a list `[0,1,2,...]` indicating all available processes.

Specifications of the parameters for a `FiniteElementMethod` class, similar to the Laplace example considered in Sec. 6.2.3, also appear in the example of this section, once for each of the three occurrences of this class. The excerpt of the settings file in Fig. 6.6 shows one of these specifications, the `PotentialFlow` finite element method, in lines 27 to 31. This `FiniteElementMethod` class shares its mesh and its linear solver with other classes. The mesh and the linear solver both have specific parameters that were listed as blocks in the settings file of the Laplace problem in Fig. 6.3. To avoid duplication of this information and to share linear solvers and meshes, these parameters are not repeated for every class, by which they are required. Instead, parameters for linear solvers and meshes can be specified globally at the beginning of the settings file and referenced at the locations in inner classes, when they are used. In the example settings in Fig. 6.6, this is indicated for the linear solver. Its parameters are defined under the global `"Solvers"` keyword in the beginning and the name `"potentialFlowSolver"` in line 4. These settings are referenced in the finite element method in line 29 using the `"solverName"` keyword. Internally, only one solver object with the related data structures of PETSc is created and reused where ever the solver is referenced by its solver name.

The meshes use an analog approach, in which all meshes can be defined under a global

"Meshes" keyword (not shown in Fig. 6.6) and referenced in the solver objects by their "meshName". This is helpful especially for meshes with many node positions that can be specified once and reused throughout all solvers.

The output of the results is written to files by output writers that are defined as shown in the previous Laplace example in Fig. 6.3. Almost all solver classes allow configuring associated output writers. In Fig. 6.6, such output writer settings are listed in line 33 within the multidomain solver. Additional output writers can be defined in the Heun scheme of the subcellular model and in the MuscleContractionSolver for the solid mechanics models. Each output writer outputs files with the solution variables of the respective solver. Different time intervals can be set for the writers to allow for different output frequencies of large data such as all subcellular model states and of smaller data such as the solid mechanics outputs.

The following exemplary command can be used to run the program for this example:

```
mpirun -n 2 ./multidomain_contraction ../settings_multidomain_contraction.
    ↪ py very_coarse.py --end_time=10
```

Similar to the Laplace example in Sec. 6.2.3, the program ./multidomain_contraction is called with the settings file ../settings_multidomain_contraction.py as its first argument. In addition, a second script, very_coarse.py, is given as second argument. This script gets loaded from within the Python settings script and defines a number of high-level parameters in a separate variables namespace. These parameters are then used in the settings file. For example, line 13 of Fig. 6.6 uses the variable n_compartments, which is defined in the so-called *variables* file very_coarse.py. The file name refers to the coarse discretization that is chosen in the particular scenario.

The rationale of this second script is to summarize important parameter values in a smaller and easier readable file. Whereas the full settings file corresponding to Fig. 6.6 contains approximately 500 lines and a complex nested structure, the variables script only contains about 200 lines, mainly value assignments to parameters and descriptive comments.

Several of these variables files exist in the variables subdirectory of the example. They define different scenarios for the given simulation such as different mesh resolutions or CellML model files. By exchanging the filename in the second argument of the command line, these different scenarios can be easily executed.

The last argument in the command, "--end_time=10", gets also parsed by the Python script. It allows to set the end of the simulation time span to the specified value via the

command line. Other options are available to alter various parameters in this scenario. These command line arguments take precedence over the parameter values that are specified in the Python scripts. This type of command line argument makes it possible to easily conduct parameter studies, e.g., from bash scripts, where the program can be called with different parameter values. The architecture involving a main settings file with all parameters in the hierarchical solver structure, a set of small variables files with dedicated parameter choices and the possibility to override all parameters from the command line is present in most of the advanced examples in OpenDiHu.

Another thing to note is that the given command begins with "`mpirun -n 2`", which instructs MPI to launch the program using two processes. Here, any other number is possible and a corresponding domain decomposition is computed automatically. The parallelism is only bounded by the number of available elements in the meshes.

6.2.5 Data Connections in the Example of a Multidomain Model with Solid Mechanics

The control flow of a simulation program with nested solvers such as the coupled electrophysiology and muscle contraction model studied in the previous section is defined by the tree of solvers in the C++ source file. This structure is reflected in the Python settings file. The corresponding data flow, which connects the solvers, is another important property that has to be specified. To help with this step, the program generates a diagram of its data connections whenever the program stops (either after completing or when interrupted by the shell).

Figure 6.7 shows such a solver structure diagram for the current example. It is given as a text file and visualizes data connections using only Unicode characters. This is advantageous for computers that only provide remote access, such as compute clusters. On the left side, the tree of nested solvers is given. On the right side, lines indicate the corresponding data connections.

Each solver has a fixed number of *data connector slots*. A data connector slot is a scalar field variable or one component of a vector-valued field variable on a certain mesh. In coupling or operator splitting schemes, values can be transferred from a data connector slot of the first solver to a data connector slot of the second solver. This data transfer either reuses the internal data structure, if possible or it involves a copy operation. Either way, after the transfer, the second solver knows the corresponding values of the first solver and can use them in subsequent computations.

```
The following data slot connection were given by the setting "connectedSlots":
  stress ¤ ──> ¤ g_mu
   g_tot ¤ ──> ¤ g_in

The following data slots were connected because the names appeared in both terms:
  lambda ¤ <─> ¤ lambda

Solver structure:
├── Coupling
│     (...)
│  ├── StrangSplitting                        ::::::::::::::
│  │  data slots:                             ::::::::::::::
│  │  [a] solution.wal_environment/vS         ├÷÷÷÷÷÷÷÷÷÷÷÷÷─── vm ¤0  x
│  │  [a] razumova/stress                     :├÷÷÷÷÷÷÷÷÷÷÷÷stress ¤1  x
│  │  [a] (P)razumova/L_S                     ::├÷÷÷÷÷÷÷÷÷÷÷lambda ¤2<─────────┐
│  │  [a] Vm^(i)_0                            :::├÷÷÷÷÷÷÷÷÷÷vm_old ¤3  x        │
│  │  [a] Vm^(i+1)_0                          ::::├÷÷÷÷÷÷÷÷÷vm_new ¤4  x        │
│  │  [a] active_stress_0                     :::::├÷÷÷÷÷÷÷÷─ g_mu ¤5  x        │
│  │  [a] activeStressTotal                   ::::::├÷÷÷÷÷÷÷ g_tot ¤6──────┐    │
│  │                                          ::::::::::::::             │    │
│  │  ├── MultipleInstances  ("Term1")        ::::::::::::::             │    │
│  │  │  ├── Heun                             ::::::::::::::             │    │
│  │  │  │  data slots:                       ::::::::::::::             │    │
│  │  │  │  [a] solution.wal_environment/vS   ├÷÷÷÷÷÷÷÷÷÷÷÷÷─── vm ¤0<─┐  │    │
│  │  │  │  [a] razumova/stress                ├÷÷÷÷÷÷÷÷÷÷÷÷stress ¤1─┐ │  │    │
│  │  │  │  [a] (P)razumova/L_S                 ├÷÷÷÷÷÷÷÷÷÷÷lambda ¤2 x│ │  │    │
│  │  │  │                                       :::::::::::          │ │  │    │
│  │  │  │  └── CellmlAdapter                    :::::::::::          │ │  │    │
│  │  │  └                                       :::::::::::          │ │  │    │
│  │  └                                          :::::::::::          │ │  │    │
│  │                                             :::::::::::          │ │  │    │
│  │  ├── MultidomainSolver  ("Term2")           :::::::::::          │ │  │    │
│  │  │  data slots:                             :::::::::::          │ │  │    │
│  │  │  [a] Vm^(i)_0                            ├÷÷÷÷÷÷÷÷÷÷vm_old ¤0<──┼─┘  │    │
│  │  │  [a] Vm^(i+1)_0                           ├÷÷÷÷÷÷÷÷÷vm_new ¤1───┼───┘    m
│  │  │  [a] active_stress_0                       ├÷÷÷÷÷÷÷÷─ g_mu ¤2<──┘      │    │
│  │  │  [a] activeStressTotal                      ├÷÷÷÷÷÷÷ g_tot ¤3  x        │    │
│  │  │  (...)                                       :::::::            m    │
│  │  └                                              :::::::            │    │
│  └                                                 :::::::            │    │
│                                                    :::::::            │    │
│  ├── MuscleContractionSolver                       :::::::            │    │
│  │   ("Term2")                                     :::::::            │    │
│  │  data slots:                                    :::::::            │    │
│  │  [b] λ                                          ├÷÷÷÷÷÷lambda ¤0<──┼────┘
│  │  [b] λdot                                        ├÷÷÷÷÷─ ldot ¤1  x │
│  │  [b] γ                                            ├÷÷÷÷─ g_in ¤2<───┘
│  │  [b] T (material traction).x                       ├÷÷÷──── T ¤3  x
│  │  [b] u.x                                            ├÷÷────── ¤4  x
│  │  [b] u.y                                             ├÷────── ¤5  x
│  │  [b] u.z                                              ├────── ¤6  x
│  │
│  │  └── DynamicHyperelasticitySolver
│  │        (...)
│  └
└

Connection Types:
  +··+   Internal connection, no copy
  ════   Reuse variable, no copy
  ───>   Copy data in direction of arrow
  ─m──   Mapping between different meshes

Referenced Meshes:
  [a] "3Dmesh", 3D structured deformable, linear Lagrange basis
  [b] "3Dmesh_elasticity_quadratic+3DFatMesh_elasticity_quadratic", 3D quadratic Lagrange basis
  [c] "3DFatMesh", 3D structured deformable, linear Lagrange basis
```

Figure 6.7: Solver structure diagram that shows the data connections of the solvers.

Each field variable has a given, fixed name defined by the solver. The corresponding data connector slot can have a custom name with a maximum length of six character, which is assigned from the Python settings. The diagram shows the field variable names on the left under the “data slots” lists of the solvers. The corresponding data connector slots are marked by the “¤” symbol and a slot number on the right. The custom name of the slot is written before the “¤” symbol.

For example, the `MuscleContractionSolver` listed in line 42 has field variables for the fiber stretch `λ`, contraction velocity `λdot`, muscle activation `γ`, traction in material description `T` and displacements in x, y and z-direction, `u.x`, `u.y` and `u.z`. As can be seen in Fig. 6.7, the first four data slots correspondingly have the names `lambda`, `ldot`, `g_in` and `T`. The fiber stretch `λ` is a quantity that is computed by the solver, and the activation parameter `γ` is a field variable that is an input to the solver and used for the computation of the active stress. However, data connector slots make no distinction between input and output slots, they simply expose the corresponding field variable to be connected to other slots.

The Heun solver in line 22 that solves the subcellular model has three slots: the slot `vm` of the transmembrane voltage, the slot named `stress` of the active stress parameter γ and the slot `lambda`, which is the input of the relative half-sarcomere length of the subcellular model.

The multidomain solver in line 32 has four slots: the slot `vm_old` exposes the field variable for $V_m^{(i)}$, the transmembrane voltage at the previous timestep. After solving the linear system of equations, the field variable $V_m^{(i+1)}$, which is connected to the slot `vm_new`, holds the transmembrane voltage for the next timestep. Another slot used for data input is `g_mu`, which retrieves the muscle activation parameter γ from each compartment. The multidomain solver computes the resulting activation parameter at slot `g_tot` by the weighted sum over the γ values at the intracellular compartments.

In case of the multidomain solver, separate field variables exist for every compartment at the same data connector slot. The solver structure diagram in Fig. 6.7 shows the field variables for slots 0 to 2 ending in “`_0`” for the compartment $k = 0$. Similar field variables exist for $k = 1, \dots, N_{\text{MU}}$, however, those are not shown in the diagram. Similarly, the Heun scheme in line 22 is nested in the `MultipleInstances` scheme in line 21. Here, the field variables that connect to the slots `vm`, `stress` and `lambda` also have different instances for every compartment. To resolve the ambiguity of multiple field variables of the same kind being associated with a single slot, every exposed field variable for data transfer has to be identified by its slot and potentially an array index within this slot.

In case of nested solvers, the parent solver class always exposes data connector slots of its children. For example, the `StrangSplitting` class in line 11 has no own slots, but exposes the slots of its two children. The slots with indices 0 to 2 are the same as the slots of its `"Term1"` in line 21, the slots 3 to 6 are identical to the `"Term2"`, the `MultidomainSolver` in line 32. These connections are indicated in the diagram by the dotted vertical connection lines. Note that the outer-most solver always contains the slots of all nested solvers. In the example in Fig. 6.7, this is the outer `Coupling` scheme. The slot listing has been omitted in the visualization.

The actual connections between the data slots of different solvers are indicated by the arrows on the right-hand side of the slots. Unconnected slots are marked by an “x”. The data transfer behavior is as follows. Each coupling and operator splitting scheme has two nested solvers. The coupling scheme executes the first solver, transfers the data over the connected data slots from the first to the second solver, executes the second solver, and then transfers the data according to the connected slots from the second to the first solver. For the Strang splitting scheme, this data transfer happens twice per timestep, as defined by the splitting algorithm (cf. Fig. 5.7b).

The interaction between the subcellular model and the multidomain model is given by the arrows between the Heun scheme in line 22 and the `MultidomainSolver` in line 32. After the solution of the subcellular model, the transmembrane voltage is transferred from slot 0 (`vm`) of the Heun scheme to slot 0 (`vm_old`) of the multidomain solver. At the same time, the stress is transferred from slot 1 (`stress`) to slot 2 (`g_mu`). After the linear system has been solved, the values for the new timestep are transferred back from slot 1 (`vm_new`) to slot 0 (`vm`).

At the outer `Coupling` scheme, after the electrophysiology model consisting of the subcellular and multidomain model parts have been solved, the active total stress is transferred from slot 6 (`g_tot`) in line 19 to the slot 2 (`g_in`) of the `MuscleContractionSolver`. Note that the starting slot `g_tot` is shared between `StrangSplitting` and `MultidomainSolver`, shown by the dotted vertical lines. Then, the solid mechanics model uses the activation value, computes new displacements and updates the slots `lambda` and `ldot`. The value in `lambda` is transferred back to slot 2 of the `StrangSplitting`, where it is shared with the subcellular model. The value of the slot `ldot`, which is the contraction velocity, is not used here, however, some subcellular CellML models make use of this value. In such a case, the corresponding connection line can be added.

In addition to the `lambda` slot, the `MuscleContractionSolver` updates the muscle geometry with the new deformed configuration. This occurs outside of data connector slots

using defined relationships or mappings between the elasticity and electrophysiology meshes. Reasons for this exception are, first, that a mesh is not owned by a solver class in the same way as other data, e.g., as a solution vector. And second, the geometry information is different from normal field variables. Changing the geometry of a mesh, e.g, invalidates finite element system matrices.

Each field variable is associated with a mesh, which is referenced by `[a]` and `[b]` in front of the field variable names. The referenced meshes are listed at the bottom of the diagram in line 64. The reference `[a]` corresponds to the mesh in the muscle domain used for the subcellular and multidomain models. The reference `[b]` is the composite mesh of both muscle and fat domain used for the solid mechanics problem.

If data connector slots of different meshes are connected, the values get mapped between the slots. This is indicated by an "m" on the connection line. In the presented example, the activation value γ gets mapped from the multidomain mesh `[a]` to the elasticity mesh `[b]` and the fiber stretch value λ gets mapped in the opposite direction.

The different connection types are also listed in the legend in line 58. The dotted connection lines of shared slots between nested solvers refer to internal connections where the slots are reused and no data copy operation is necessary. The solid arrows indicate a copy operation. The legend shows also double connection lines, which indicate that the field variable of two slots can be reused and no copy is required. This type of connection is not present in the current example, but occurs for example in most of the fiber based electrophysiology models. The last connection type is the mapping, indicated by a line with an "m" character.

Which connection type to use is determined by OpenDiHu. In case of matching meshes, the double line connection that reuses the field variable is preferred. However, it is not always possible because changes in a reused field variable also influence the field variable at its original point of use, which may not be desired. In the current example, the reason why the "copy" connections are used between the subcellular and multidomain solvers lies in the number of compartments. The subcellular models holds the data of all compartments in an array-of-vectorized-struct memory layout, such that the order of the compartments' variables in memory is different than the required order for the slot. Thus, the data have to be copied during transfer between connected slots.

The specification of which slots to connect with each other is given in the settings file. Three possibilities how to define slot connections exist: First, the slot numbers of connected slots can be given in the settings of coupling and operator splitting schemes.

Second, the names of connected slots can be specified under the global keyword "connectedSlots". In the given example, this is the case for the slots listed in Fig. 6.7 in lines 1 to 3. Third, slots with the same name are connected automatically. In the considered example, this is the case for the `lambda` slot, which is named identically in the `CellmlAdapter` (line 26) and the `MuscleContractionSolver` (line 45). Slots connected by the third possibility are also listed at the top of the diagram, here in lines 5 and 6.

6.2.6 Exemplary Usage: Neuromuscular System

In the example in the last sections Sections 6.2.4 and 6.2.5, the nested solver structure was a binary tree. However, also scenarios with a more general tree structure exist. The solver tree for a simulation of the neuromuscular system including sensory feedback is shown in Fig. 6.8. The tree corresponds to the example in the directory `examples/electrophysiology/neuromuscular/spindles_multidomain`.

The following solver classes are involved in this example. For executing multiple solvers in series, the top-level `MultipleCoupling` class exists, which calls its nested solvers one by one in every timestep. The `PrescribedValues` class (a) can be configured to set any of its field variables to prescribed values. The values can be set by callback functions in the Python settings that get frequently called by the solver to update the values over time.

A `MuscleContractionSolver` combines either a static or a dynamic hyperelasticity model with the active stress term used in the muscle contraction model. The class in (b) uses the static `HyperelasticitySolver`. The solvers in (a) and (b) compute the static contraction of the muscle under a prescribed constant activation level. Then, another `HyperelasticitySolver` (c) stretches the muscle tissue again by a prescribed external force to yield a prestretched muscle. The actual transient simulation is then performed under the subtree at (d). Again, a `MultipleCoupling` class is used to run all nested solvers in every timestep.

In (e) and (f), two CellML models are solved using a Heun scheme, the first one for the muscle spindles and the second one for the motor neurons. A filter step is applied on the resulting signals and the values are copied to the destination field variables using the `MapDofs` class in (e), (f) and (g). A `MapDofs` class is able to copy degrees of freedom between two field variables, apply custom Python functions on the values and communicate values between processes in parallel execution.

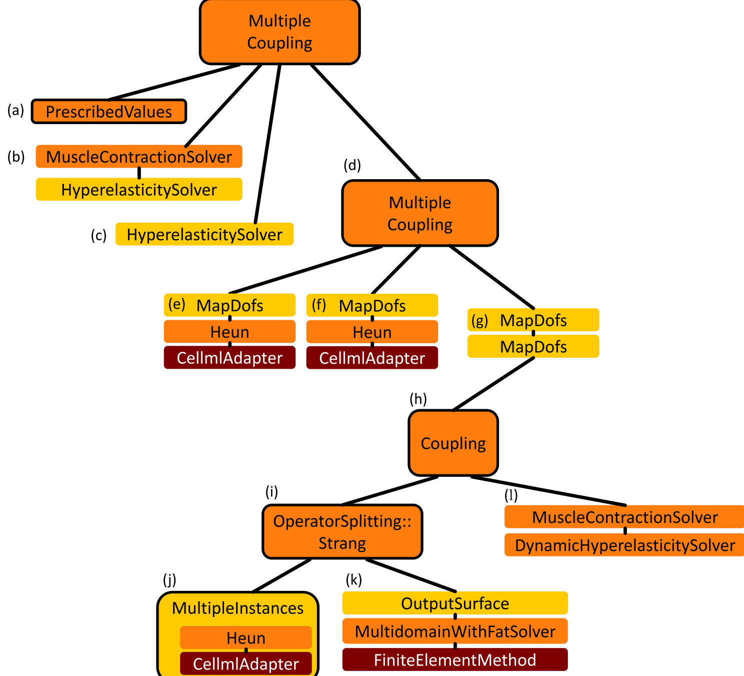


Figure 6.8: Solver tree for a simulation of the neuromuscular system. Classes of type `DiscretizableInTime` are shown as red boxes, `TimeSteppingSchemes` are given by orange boxes.

The subtree under (h) is identical to the example presented in Sec. 6.2.4. It solves the electrophysiology model using the multidomain equations in (j) and a subcellular model in (k), coupled to the solid mechanics model in (l).

The tree in Fig. 6.8 consists of solver classes of different types. The orange boxes indicate timestepping schemes. Internally, these classes derive from a `TimeSteppingScheme` interface class. They have a common set of parameters such as the timestep width and end time. The boxes with dark red background color are classes of type `DiscretizableInTime` . They represent a term or equation that can be nested in a timestepping scheme. There only exist two different classes of this type: The `CellmlAdapter`, which contains a system of DAEs given by a CellML model and the `FiniteElementMethod`, which discretizes the generalized Laplace operator $\operatorname{div}(\boldsymbol{\sigma} \operatorname{grad} u)$.

Besides the classes of the presented example shown in Fig. 6.8, further solver classes are available in OpenDiHu. A comprehensive list of all available solver classes is given in the following section.

6.2.7 Summary of Existing Solver Classes

All the timestepping schemes introduced in Eq. (5.47) are available to solve ODEs given by `DiscretizableInTime` objects: The explicit schemes are the explicit Euler and Heun's method. The available implicit schemes are the implicit Euler and Crank-Nicolson method. Implemented operator splitting schemes are the Godunov and Strang splittings. The implementation of the `Coupling` class is identical to Godunov splitting. As mentioned in Sec. 6.2.6, `DiscretizableInTime` objects are either given by the `CellmlAdapter` or the `FiniteElementMethod`.

Some classes are special solvers for dedicated models: A `StaticBidomainSolver` is used to solve the first bidomain equation Eq. (5.9a). The `MultidomainSolver` and `MultidomainWithFatSolver` classes solve the multidomain models Eqs. (5.14) and (5.15) without and with body fat domain. A class `FastMonodomainSolver` exists that improves the parallel performance of the fiber based electrophysiology solver using the monodomain equation Eq. (5.11).

Solid mechanics models can be computed by a series of specialized solvers. The `QuasiStaticLinearElasticitySolver` class uses a `FiniteElementMethod` object to compute 3D linear elasticity using Hooke's Law with an additional active stress term, as derived in Sec. 5.4.1. The `HyperelasticitySolver` class solves the static hyperelasticity formulation for any material model, as presented in Sec. 5.4.2. The `DynamicHyperelasticitySolver` class inherits from the `HyperelasticitySolver` class and adds functionality to solve the dynamic hyperelasticity formulation shown in Sec. 5.4.6. Both the static and the dynamic hyperelastic solvers do not incorporate the active stress term that is present in the muscle contraction model. This is handled by another class, the `MuscleContractionSolver`. It uses either a `HyperelasticitySolver` or a `DynamicHyperelasticitySolver` object and adds the functionality accordingly.

Instead of solving a model numerically, also precalculated analytic solutions can be used. This can be done using the `PrescribedValues` class, which uses a Python function to set the solution values. Further auxiliary classes exist that are no numerical solver: The `MapDofs` class gives flexibility to transfer certain degrees of freedom between field variables. The `Dummy` class can be used as a placeholder. The `OutputSurface` class extracts

a 2D mesh at the surface of a 3D mesh and writes it to an output file using the normal output writers. This can be used to reduce the amount of data output for finely resolved EMG simulations, where only the values at the surface are of interest.

Moreover, adapters to external software tools are implemented. The class `NonlinearElasticitySolverFebio` allows to use the solver *FEBio* [Maa12; Maa17] for solving a continuum mechanics model and couple it to an electrophysiology model in OpenDiHu. Two adapters to the numerical coupling library preCICE [Bun16], `PreciceAdapter` and `PreciceAdapterVolumeCoupling` exist for surface and volume coupling. They can be configured to implicitly or explicitly couple any field variables to external solvers or to couple two separate instances of OpenDiHu. For more details on the solver classes and their configuration, we refer to the online documentation [Mai21c].

6.2.8 Graphical Helper Program

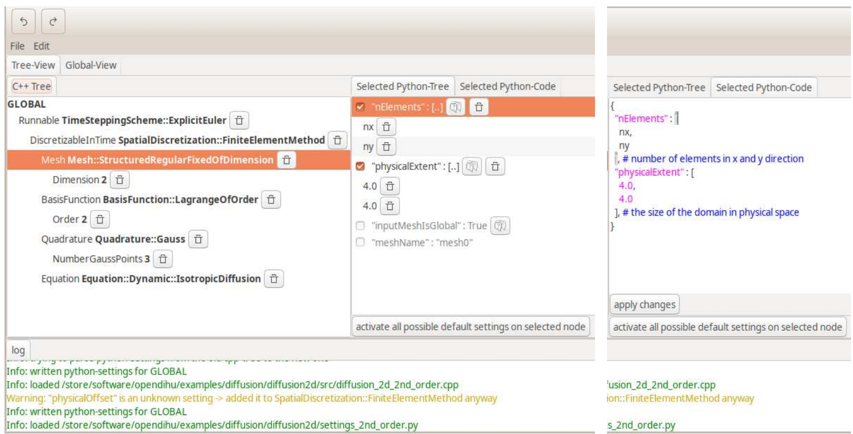
For the creation of new examples from scratch, a helper program with a graphical user interface exists. The program was created by Matthias Tompert in his Bachelor thesis and is included in the OpenDiHu repository under `scripts/gui/gui.py`.

Graphical widgets allow selecting and nesting compatible solver classes for OpenDiHu. The corresponding Python settings are automatically shown with their default values and can be adjusted in the graphical user interface. For every option, explanatory comments are displayed and buttons allows to open the corresponding page of the online documentation in a web browser.

The program also features a horizontally split code editor, which shows both the C++ code and the corresponding Python code for the settings, either for a single node in the solver tree or as a global view of the whole example. After the user completes the adjustments of the solver structure and the settings, the C++ source file and the Python settings can be exported and used with OpenDiHu.

The program is also capable of parsing existing C++ and Python files and, thus, loading an existing example into the graphical representation to be extended by the user. The program is able to parse a large portion of the solvers and options that are available in OpenDiHu. However, some more advanced examples, e.g., where the Python settings contain complex code constructs are not fully supported.

Figure 6.9 shows the user interface after the 2D diffusion example of OpenDiHu has been loaded. The left pane in Fig. 6.9a represents the solver tree as it appears in the C++



(a) User interface with the tree of nested solvers on the left and the Python settings for the selected mesh on the right.

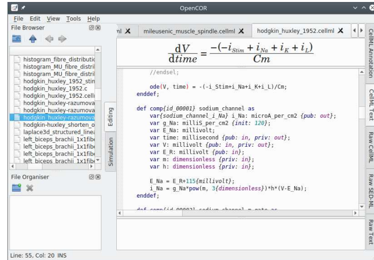
(b) Alternative view of the right pane that shows the Python settings editor.

Figure 6.9: Graphical helper program to create and adjust the C++ and Python codes of OpenDiHu examples.

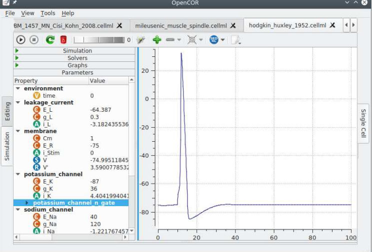
file. The right pane in Fig. 6.9a displays the settings for the selected item in the left pane, which, in this case, is the mesh. Additional options that are not yet present in the Python settings are grayed out and can be added by clicking the checkboxes. Figure 6.9b shows an alternative view in the right pane, which displays the corresponding Python code. The user can also directly edit the settings there.

6.3 Usage of CellML Models

The CellML description language can be used to describe mathematical models of a wide range of physiological processes. Arbitrary systems of differential-algebraic equations (DAE) can be represented. We use it for incorporating and exchanging subcellular models, which describe the electrophysiology on a muscle fiber, and for models of motor neurons or sensory organs. The CellML infrastructure is popular in the bioengineering community. The CellML website of the Physiome project hosts over 600 curated CellML models from different areas. Each model can be downloaded in CellML format or as source code



(a) CellML editor with the ODE for the membrane voltage “V” in the Hodgkin-Huxley cellular model, which corresponds to Eq. (5.3a) inserted into Eq. (5.4).



(b) Visualization of a simulated action potential V_m over time.

Figure 6.10: The CellML modeling environment OpenCOR.

containing the expressions of the equations in various programming languages such as MATLAB, Python and C.

6.3.1 Integration of CellML in OpenDiHu and Comparison with Other Framework

Mathematically, a CellML model describes the functions G and H of the following DAE:

$$\frac{\partial \mathbf{y}(t)}{\partial t} = G\big(t, \mathbf{y}(t), \mathbf{h}(t), \hat{\mathbf{c}}, \hat{\mathbf{p}}(t)\big), \qquad \mathbf{h}(t) = H\big(\mathbf{y}(t), \hat{\mathbf{c}}, \hat{\mathbf{p}}(t)\big). \tag{6.1}$$

Here, $\mathbf{y}$ is the state vector and $\mathbf{h}$ is a vector with additional values that are derived from the state vector. The vectors of constants $\hat{\mathbf{c}}$ and parameters $\hat{\mathbf{p}}$ are prescribed and fixed over time for $\hat{\mathbf{c}}$ or varying over time for $\hat{\mathbf{p}}$.

Various open source tools exist to create or manipulate CellML models and to solve them and visualize the results [Gar08]. A comprehensive list is given on the CellML website [Cel21] and some of them, which are relevant to our work, are outlined in the following.

There exist two application programming interfaces (APIs), the *CellML API* and the newer *libCellML*, which allow direct access to the structures of the CellML model from, e.g., C++ code [Mil10].

OpenCOR [Gar15] provides a modeling environment in a graphical user interface, where models can be edited. Figure 6.10a shows the interface with the editor on the right. Mathematical equations are described in a declarative language and can be rendered to mathematical notation, as seen in the upper part in Fig. 6.10a. OpenCOR automatically transfers the equations to the XML-based MathML syntax and integrates them in the XML-based CellML description. OpenCOR can also be used to solve the system of DAEs using implicit solvers such as backward differentiation formulas. The solver parameters can be adjusted and the solver can be started from the graphical user interface. Figure 6.10b shows the interface that lists all variables with their current values on the left and a visualization of the result, in this case an action potential, on the right.

OpenCOR also provides command line functionality to convert CellML files into C code. This generated C code can evaluate all model equations but not solve the DAE system. Because OpenCOR is robust and well established in the bioengineering modeling community, we decide to use it in OpenDiHu. The installation procedure of OpenDiHu downloads and installs OpenCOR automatically.

During execution of a simulation, our framework parses the C code of CellML models, compiles a shared library and executes the functions, all at runtime. Thus, the CellML model can be directly given as a C file. Otherwise, if the model file is in XML format, it is assumed to be a CellML description and automatically converted to the required C code using the OpenCOR command line interface.

If a CellML model is manually simulated in the OpenCOR graphical user interface with time-varying input signals, these signals have to be hard-coded in the model, e.g., as a piecewise defined function. This is acceptable for getting insight into the models, but counteracts the idea of modular models that can be shared and recombined. As a remedy, we design our framework in a way that simulations of CellML models with configurable time-varying input signals are possible without the need to change the CellML description.

CellML models are limited to single-cell systems of DAEs and are not designed for PDEs that, e.g., involve multiple instances of a DAE system on a given geometry. Thus, the monodomain equation cannot be solved with OpenCOR and a multi-scale software framework is needed for this task. Two such frameworks with CellML support, which were described in the introduction in Sec. 1.3.2, are Chaste and OpenCMISS Iron. In the following, we relate and compare the approaches of CellML integration in OpenDiHu and these existing frameworks.

Symbol	Name			Computed by model?	Initial values can be set?
	OpenCOR	OpenCMISS	OpenDiHu		
$\mathbf{y}$	`state`	`STATES`	`state`	by timestepping	yes
$\partial\mathbf{y}/\partial t$	`rate`	`RATES`	`rate`	yes	no
$\hat{\mathbf{c}}$	`constant`	`CONSTANTS`	`constant`	no	in CellML
$\mathbf{h}$	`algebraic`	`WANTED`	`algebraic`	yes	no
$\hat{\mathbf{p}}$	-	`KNOWN`	`parameter`	no	yes

Table 6.1: The different CellML quantities and their properties and names in various tools.

The variables in the generic DAE in Eq. (6.1) have different names in the different software packages. Table 6.1 compares the symbols and their names in OpenCOR, OpenCMISS in OpenDiHu and summarized how their values are determined.

All three software packages have the concept of `state` and `rate` vectors, where the states $\mathbf{y}$ are the input and the rates $\partial\mathbf{y}/\partial t$ are the output of the CellML formulas. Similarly, the constants $\hat{\mathbf{c}}$ are always a set of predefined values that are fixed during the computations.

The algebraic formulas lead to the values in $\mathbf{h}$, independently of the timestepping scheme. These algebraic values can be considered as the resulting quantities of interest of the model and are typically written to output files or transferred to coupled solvers.

Moreover, OpenCMISS and OpenDiHu define parameters $\hat{\mathbf{p}}$, which influence the behavior of the model. Their values can be changed by a coupled solver or prescribed from the settings. In OpenDiHu, any constant or algebraic variable in a CellML model can be converted into a parameter. All occurrences of the constant or algebraic variable in the CellML description get replaced by the parameter variable. For former algebraic variables, this replacement step overrides the equations that would have defined the algebraic value. Exemplary use cases are to set the external stimulation current I_{ext} in Eq. (5.4) or to set the fiber stretch in a strain-dependent subcellular model.

OpenCMISS uses a similar concept, where some algebraics in the CellML description can be declared as `WANTED` to be read by the framework. Some of the constants can be declared as `KNOWN` such that OpenCMISS sets their values from other computations within OpenCMISS. (Assigning new values to algebraics as in OpenDiHu is not possible.) Because the terms `WANTED` and `KNOWN` can be ambiguous if either seen from within the CellML model or from the framework, we decide to use the terms `algebraics` and `parameters` instead.

The last two columns of Tab. 6.1 summarize the purpose of the different quantities. The CellML description defines formulas for the states, rates and algebraics. Rates and

algebraics are directly calculated by the code that is generated from the CellML model, the vector of states is then computed from the vector of rates by the timestepping scheme. The initial values of the states are either explicitly specified in the OpenDiHu settings, e.g., to allow different values for different instances of a model. Or, if this specification is omitted, the initial values are set according to the specification in the CellML file. The parameter values always have to be specified in the Python settings. By definition, the constants cannot be set from OpenDiHu, but are given in the CellML model. If the value of a constant should be specified from the settings, the variable should instead be configured to be a parameter.

From a computational point of view, a CellML model computes the following function in terms of the introduced variable names:

$$\begin{pmatrix} \texttt{rates} \\ \texttt{algebraics} \end{pmatrix} = \texttt{cellml(states, constants)}. \tag{6.2}$$

In the fiber based electrophysiology model, CellML is needed to formulate the reaction term in the monodomain equation Eq. (5.11), which is repeated here:

$$\frac{\partial V_m}{\partial t} = \frac{\sigma_{\text{eff}}}{A_m\, C_m} \frac{\partial^2 V_m}{\partial x^2} - \frac{1}{C_m} I_{\text{ion}}(V_m, \mathbf{y}). \tag{6.3}$$

The `states` vector in Eq. (6.2) includes both V_m and $\mathbf{y}$ in Eq. (6.3). In consequence, the computed `rates` contain $\partial V_m/\partial t$ and $\partial \mathbf{y}/\partial t$. The right-hand side of Eq. (6.2), i.e., the `cellml` function calculates the term $(-1/C_m \cdot I_{\text{ion}})$, which is the reaction part of the monodomain equation in Eq. (6.3). Thus, the CellML computation can be directly used in the operator splitting approach in Sec. 5.3.1.

For the solution of CellML models, OpenCMISS implements the explicit forward Euler scheme or allows to use the backward differentiation formula (BDF) schemes with adaptive order of convergence of SUNDIALS. Recently, an implementation of the second order explicit Heun scheme was added by Aaron Krämer. Accuracy and runtimes were investigated for Euler, Heun and BDF solvers for the subcellular model within the monodomain equation. Because of the operator splitting scheme, only very small timespans have to be solved by those solvers, which does not redeem the overhead of advanced schemes such as the BDF solver, ultimately yielding the best performance for the Heun solver. Based on these investigations, we choose to implement the forward Euler and Heun schemes for the solution of CellML models in OpenDiHu.

The following differences exist between the approaches to support CellML models in Chaste, OpenCMISS Iron and OpenDiHu: Chaste tries to automatically determine the CellML variable names of standard quantities such as the membrane voltage and the stimulation current. This requires potentially less user intervention when CellML models are exchanged. In OpenDiHu, the step of identifying the CellML variables to be connected to the coupled solvers is done manually to give the user complete control over the setup. It can be achieved in a clear way with the Python settings script. Another difference in OpenDiHu is that all computational code is guaranteed to invoke vector instructions, i.e., following the single-instruction multiple data (SIMD) paradigm. Chaste only relies on the optimization behavior of the Intel compiler, which is not guaranteed to be optimal, e.g., for non-Intel hardware.

The computational core Iron from the OpenCMISS package employs the CellML API and also requires manual connections of CellML variables to the solver code. These variable mappings have to be hard-coded in the main Fortran program (if the Python wrappers are not used) and are compiled into the program. Thus, a CellML model is a fixed part of a compiled simulation program. In contrast, OpenDiHu allows to configure the CellML model at runtime. Another difference in the implementation is that Iron uses a non-optimal memory layout for the state vector, which prohibits vectorization and slows down the solution compared to OpenDiHu.

6.3.2 Mapping of CellML Variables to Slots and Parameters

Preparing the OpenDiHu solver for use with a CellML model consists of the two steps of adjusting the C++ template parameters and setting up the variable mappings in the Python settings. The two C++ template parameters have to be set to the sizes of the state vector $\mathbf{y}$ and the algebraics vector $\mathbf{h}$. The code snipped in Fig. 6.11 belongs to the example program in examples/electrophysiology/cellml/shorten, which solves a single-cell CellML model:

```
1 TimeSteppingScheme::ExplicitEuler<
2   CellmlAdapter<56,71>
3 >
```

Figure 6.11: C++ code snipped that solves a CellML model with an explicit Euler scheme. The two template parameters 56 and 71 correspond to the number of states and algebraics, respectively.

```
mappings = {
  # function in OpenDiHu        name in CellML model     # comment

  ("parameter", 0):             "wal_environment/I_HH", # I_stim (constant)
  ("parameter", 1):             "razumova/L_S",         # λ (constant)

  ("connectorSlot", "vm"):      "wal_environment/vS",   # V_m (state)
  ("connectorSlot", "stress"):"razumova/stress",       # γ (algebraic)
  ("connectorSlot", "lambda"):"razumova/L_S",          # λ (constant)
}

parameters_initial_values = [0.0, 1.0]                  # I_stim=0, λ=1
```

Figure 6.12: Specification of parameters and connector slots in a CellML model. The listed settings define two CellML variables to be parameters and specify three connector slots to transfer values between coupled solvers.

In this case, the model contains 56 states and 71 algebraics. The reason that these numbers have to be fixed at compile-time is that this allows the data structures in the implementation to have a fixed layout and be allocated on the stack instead of the heap, which improves the performance.

If the given numbers are not matching the variables in the CellML file, appropriate warnings or errors are generated, containing the correct C++ code to be copied to the C++ file. If the numbers are too high, the solver still works correctly, however, some memory and computation time is wasted for the excess variables.

The other step is configuring the connections between the CellML computation and input data or coupled solvers. This involves defining a `mappings` parameter. Figure 6.12 shows such a definition for the multidomain example with fat layer and a contraction model, which was presented in Sec. 6.2.4.

The `mappings` define which CellML constants or algebraics are treated as parameters. Lines 4 and 5 make the stimulation current and fiber stretch constants accessible from outside the CellML model by making them parameters. The variables are identified by their names and the model components they are defined in in the CellML model. In this example, the first parameter is the stimulation current `I_HH` within the `wal_environment` model component and the second parameter is the fiber stretch or half-sarcomere length `L_S` in the `razumova` component. The initial values for these parameters are given in line 12, which sets the stimulation current to zero and the fiber stretch to one.

The second information in the `mappings` parameter is which variables from the CellML model are exposed to coupled solvers in OpenDiHu. This happens by defining connector slots that can be connected between the solvers as shown in Fig. 6.7. Three slots are defined in lines 7 to 9 with slot names `"vm"`, `"stress"` and `"lambda"`. The corresponding CellML variables are again specified by their model component name and their own name.

CellML variables of all three different types are connected in the example. The membrane voltage V_m in slot `"vm"` is part of the state vector $\mathbf{y}$. In this example, it is used in a bidirectional coupling with the diffusion solver. The second slot, `"stress"`, connects to the activation parameter γ, which is part of the algebraic vector $\mathbf{h}$. It is an output of the model. The slot `"lambda"` refers to a constant in the CellML description, which has been transformed to a parameter in line 5. It is used as an input and the received values at these slots are moved to the corresponding locations in the CellML formulation.

6.3.3 Consistent Physical Units in CellML Models and the Multi-Scale Framework

The variables in a CellML model describe physical quantities. CellML handles their physical units and computes the appropriate conversions when combining model components within a CellML description. For the integration of a CellML model in external solvers such as OpenDiHu, we have to take care that the units are consistent.

The subcellular models that we use are formulated with the following units for length, time, electric current and capacitance:

$$1\,\text{cm} = 10^{-2}\,\text{m}, \qquad 1\,\text{ms} = 10^{-3}\,\text{s}, \qquad 1\,\mu\text{A} = 10^{-6}\,\text{A}, \qquad 1\,\mu\text{F} = 10^{-6}\,\text{F}.$$

These basic units also fix derived units such as 1 kHz for frequencies and 1 mV for voltages. For example, the membrane capacitance C_m has to be specified in units $1\,\frac{\mu\text{F}}{\text{cm}^2}$ and the stimulation current I_stim in the units $1\,\frac{\mu\text{A}}{\text{cm}^2}$.

With this system of units, values are in a similar scale when computing subcellular models. However, these units are less suitable for organ-scale computations, as the derived mass and density units are $10^{-14}\,\text{kg}$ and $10^{-8}\,\frac{\text{kg}}{\text{m}^3}$ and the derived force and stress units are $10^{-10}\,\text{N}$ and $10^{-6}\,\text{Pa}$. For the dynamic solid mechanics model, where these

quantities play a role, we use the following different system of units:

$$1\,\text{cm} = 10^{-2}\,\text{m}, \qquad 1\,\text{ms} = 10^{-3}\,\text{s}, \qquad 1\,\text{N}.$$

The length and time scales are identical to the subcellular model and allow for consistent coupling. The coupling of active stresses from the subcellular model to the solid mechanics model uses the unit-less activation parameter $\gamma \in [0, 1]$, which is transferred to stress units by multiplication with a maximum active stress value in the solid mechanics model.

Derived units in the solid mechanics system of units are $10^2\,\frac{\text{kg}}{\text{m}^3}$ for the density, $10^4\,\frac{\text{m}}{\text{s}^2}$ for the acceleration and $1\,\frac{\text{N}}{\text{cm}^2} = 10\,\text{kPa}$ for the stress. The values of material parameters and boundary conditions have to be given with respect to these units. The units allow for smaller values in the solid mechanics computation than in the unit system of the subcellular model. Moreover, it is convenient to specify forces directly in terms of 1 N.

6.3.4 Specification of Stimulation Times Using Callback Functions

A muscle fiber is activated by impulse trains that are generated from a motor neuron and stimulate the fiber at its neuromuscular junction. At the synaptic terminal, neurotransmitters are released and open certain ion channels, which results in depolarization of the muscle fiber membrane. This process can either be modeled by adding an external stimulation current I_{stim} through the dedicated ion channels or by directly prescribing the transmembrane voltage V_m to reflect the resulting depolarized state. The first approach is more accurate as it also describes the depolarization process at the stimulated parts of the fiber. The electric "far field" away from the stimulation point, however, is the same for both approaches.

In OpenDiHu, it is possible to configure either approach. Setting the stimulation current is more involved as the actual value of I_{stim} has to be chosen depending on the mesh width. Furthermore, multiple adjacent nodes have to be stimulated such that the electric current that is added to the system balances with the amount that is carried away by the diffusion term. The nonlinear subcellular model fails to compute a valid solution, if too much current is present. With too little current, the membrane potential stays below the activation threshold and no action potential is triggered.

Prescribing the transmembrane voltage to a value above the depolarization threshold at multiple adjacent nodes leads to equivalent action potentials independent of the mesh

width. However, a suitable value for the prescribed voltage also has to be chosen in accordance with the employed subcellular model.

The stimulation current I_{stim} is a CellML parameter and the transmembrane voltage V_m corresponds to a state in the CellML model. The values of both parameters and states can be adjusted during the simulation. This feature is implemented by means of callback functions in the Python settings. A callback is a user defined function that gets called in regular intervals during the simulation, receives various information about the current state of the simulation and can alter some values such as the states vector $\mathbf{y}(t)$ or the parameters vector $\hat{\mathbf{p}}(t)$.

Figure 6.13 defines two such callback functions used in the fiber based electrophysiology model to add electric stimulation to the monodomain model. Either suffices to implement the stimulation. The function `set_specific_parameters` in line 3 and the function `set_specific_states` in line 16 both receive similar information from the simulation as their function arguments: the total number `n_nodes_global` of nodes in the current fiber, the current integer timestep number `time_step_no`, the corresponding floating-point number `current_time` of the current simulation time, and the number `fiber_no` that identifies the current fiber.

The variables `parameters` and `states` are the output of the callback functions that alter the parameter and state values, respectively. Both callbacks determine, whether the current fiber should be stimulated at the current time, in lines 7 and 20. If yes, the parameter or state at the center point of the fiber, computed in lines 12 and 21, gets changed accordingly. In the real scenario, three adjacent points get stimulated instead of a single point.

Because the conversion of transferred data between the Python code and the C++ code costs some runtime, the number of transferred values is reduced to a minimum. Only the parameters and states that should be changed are indicated in the `parameters` and `states` variables in lines 13 and 22. These variables are Python dictionaries, i.e., key-value pairs. The key is a tuple of three items: First, the global coordinates (x, y, z) of the node where the parameter or state change is applied. In case of a 1D fiber mesh, this is only a single coordinate. Second, the dof index on this node. This is different from zero only for Hermite ansatz functions, which have multiple dofs per node. And third, the index of the parameter or state that should be set. Parameter 0 corresponds to the stimulation current as defined in line 4 of Fig. 6.12, and state 0 corresponds to the transmembrane voltage V_m. The new value to set is the value of the key-value pair.

```
1
2  # callback function that can set parameters, i.e. stimulation current
3  def set_specific_parameters(n_nodes_global, time_step_no, current_time,
4                              parameters, fiber_no):
5
6    # determine if fiber gets stimulated at the current time
7    if fiber_gets_stimulated(fiber_no, current_time):
8      stimulation_current = 40.
9    else:
10     stimulation_current = 0.
11
12   innervation_node_global = int(n_nodes_global / 2)
13   parameters[(innervation_node_global),0,0] = stimulation_current
14
15 # callback function that can set states, e.g., prescribe V_m for stimulation
16 def set_specific_states(n_nodes_global, time_step_no, current_time,
17                         states, fiber_no):
18
19   # determine if fiber gets stimulated at the current time
20   if fiber_gets_stimulated(fiber_no, current_time):
21     innervation_node_global = int(n_nodes_global / 2)
22     states[(innervation_node_global),0,0] = 40.0
23
24 config = {
25   (...)
26
27   # callback to adjust parameters
28   "setSpecificParametersFunction":          set_specific_parameters,
29   "setSpecificParametersCallInterval":      1e3,
30   "setSpecificStatesFrequencyJitter":       0,
31
32   # callback to alter values of states
33   "setSpecificStatesFunction":              set_specific_states,
34   "setSpecificStatesCallInterval":          2*int(1/stimulation_frequency/dt_0D),
35
36   "setSpecificStatesCallFrequency":         stimulation_frequency,
37   "setSpecificStatesCallEnableBegin":       0,
38   "setSpecificStatesRepeatAfterFirstCall":  0.01,
39   "setSpecificStatesFrequencyJitter":       [0.1,-0.2,0.0],
40
41   # callback to postprocess the result
42   "handleResultFunction":                   handle_result,
43   "handleResultCallInterval":               1e4,
44
45   "additionalArgument":                     fiber_no,
46 }
```

Figure 6.13: Settings that define neural spike trains activating muscle fibers. The definition of the two callback functions `set_specific_parameters` and `set_specific_states` is demonstrated.

The comparison of the two callbacks functions shows one difference: In the callback for the parameters, the stimulation current is set to zero when there is no stimulation. In the callback for the states, nothing is done during this time. The reason for this is that the state values will be continuously updated from the rates by the timestepping scheme, whereas the parameters keep their values until they are changed from the callback. This has consequences on the times at which the callback functions have to be called from the simulation, which are described in the following.

Invoking the Python interpreter on a callback requires some time. Calling the callback after every small timestep of the simulation is, thus, not performant. We model the stimulation of a fiber by a piecewise constant function with two possible values for on and off. In the approach that sets the stimulation current, the callback `set_specific_parameters` has to be called at the onset and at the end of every stimulation spike. If the approach with the prescribed membrane voltage is used, the callback `set_specific_states` has to be called after every stimulation onset in every subsequent timestep until the stimulation is over.

The requirements for both approaches can be satisfied by defining a small constant interval of timesteps after which the callback functions are invoked. This call interval can be specified in the `config` dictionary of the Python file in Fig. 6.13 , which is shown in excerpts from line 24 onwards. The `config` variable references the callback functions in lines 28 and 33 and the parameters for the call interval in the next lines. Note that this configuration is only shown for demonstration, a real configuration should either specify the states callback or the parameters callback function, not both.

Line 34 in Fig. 6.13 shows how the call interval can be computed to correspond to a given stimulation frequency `stimulation_frequency`, given the timestep width `dt_0D`. The prefactor of two occurs because the callback would be called twice per timestep in the Strang splitting scheme.

Real impulse trains from the motor neuron pool typically follow a base frequency with some added jitter that offsets the exact firing times from the base frequency by a small random time. Furthermore, studies are often designed to start with a completely inactive muscle and switch on certain MUs after specified times. To efficiently account for these two demands, we add another way to specify the times when the `set_specific_states` callback gets invoked. In this second way of specification, the `setSpecificStatesCallInterval` parameter is disabled by setting it to zero. Then, the three options `CallFrequency`, `CallEnableBegin`, `RepeatAfterFirstCall` and `FrequencyJitter` (prefixed by `setSpecificStates`) given in lines 36 to 39 are significant.

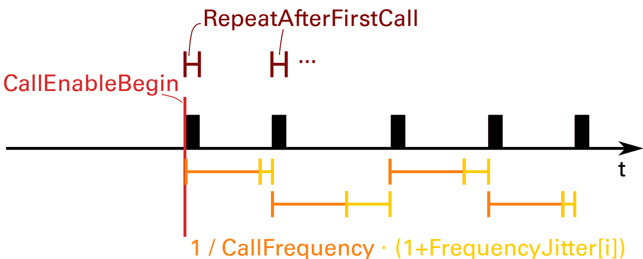


Figure 6.14: Parametrization of stimulation times in electrophysiology simulations. The neuronal impulse train is given by the black spikes. The parameters `CallEnableBegin`, `RepeatAfterFirstCall`, `CallFrequency` and `FrequencyJitter` (in the settings all prefixed by `setSpecificStates`) specify the shape of the spike train.

Their meaning is illustrated in Fig. 6.14. `CallEnableBegin` specifies the time when the callback should be called for the first time. Then, it is called with a frequency that is additively composed of the base frequency given by `CallFrequency` and one entry of the parameter `FrequencyJitter`. This parameter is a ring buffer of relative factors by which the regular time span between subsequent firing events is prolonged. For example, if `FrequencyJitter` contains the list `[0.1,-0.2,0.0]`, the time span T_{01} between the first two firing events is 10 % longer than according to the base frequency f, the next timespan T_{12} is 20 % shorter and the next time span T_{23} exactly equals the inverse base frequency, $T_{23} = 1/f$. Subsequently, the scheme repeats. Typically, this parameter is set to a randomly generated list with a large number of entries. After each onset of a stimulation, the `setSpecificStatesCallInterval` function is called repeatedly in every subsequent timestep for a time span given by `RepeatAfterFirstCall`.

In the fiber based electrophysiology example, every fiber has its own instance of the Python settings, and it is possible to specify different parameter values for different fibers or motor units, e.g., to set a different beginning time of the stimulations. The fibers can be distinguished by the last parameter of the callbacks, which receives the custom value that is defined by the `"additionalArgument"` parameter in line 45. In the given example, the current fiber number is used here, but any other Python variable is possible.

Figure 6.15 shows a scenario, where different parameters are set for different MUs. The figure shows the firing times of fibers grouped to 20 MUs, which are activated in a ramp in the first $t = 19$ s. The base frequency decreases from 23.92 Hz to 7.66 Hz for MUs 1 to 20, which reproduces a scenario in literature [Klo20]. The frequency jitter parameter is

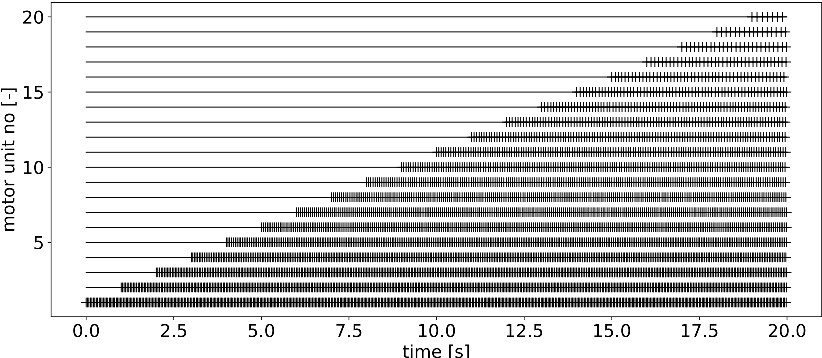


Figure 6.15: Firing times for a scenario with 20 motor units with ramp-like activation and different stimulation frequencies.

a list of 100 randomly chosen values between −10 % and +10 %. The `CallEnableBegin` parameter enables the stimulation of the next MU every second.

Similar to the two presented callbacks, which set parameters and states, a third callback `handle_result` can be defined as given in line 42 of Fig. 6.13. This callback function gets called in a fixed interval specified by `"handleResultCallInterval"`. It receives the complete vectors of states $\mathbf{y}$ and intermediates $\mathbf{h}$ and can be used to perform custom post-processing or to output custom data files from the Python script.

In summary, variables of CellML models can be coupled to other solvers. Their parameters and values can be adjusted from the settings file. Callback functions are used to alter values during the simulation. This flexibility comes at the runtime cost of invoking the Python interpreter, therefore the times when to call the callback functions have to be specified appropriately. Special methods exist to model steady stimulation with frequency jitter, which occurs in typical neural stimulation of muscle fibers.

6.4 Output File Formats

After the simulation program completes, the computed results can be visualized using external tools. As mentioned in the previous sections, output writers are used to generate output files in various formats. The formats of the output writers and additional options

are configured in the Python settings under the parameter `OutputWriter`. The following formats are supported:`"ParaView"`, `"ExFile"`, `"PythonFile"`, `"PythonCallback"` and `"MegaMol"`. The corresponding output can be visualized and post-processed using different tools, which will be presented in the following. We use simulation results of a fiber-based electrophysiology scenario with 49 1D fibers and a 3D muscle mesh to showcase the different output data formats.

6.4.1 Output of VTK Files for the Use with ParaView

The canonical way to visualize simulation results computed by OpenDiHu is to use the software ParaView [Ahr05]. The required output file formats are defined by the Visualization Toolkit (VTK) specification [Sch06]. Depending on the mesh type in OpenDiHu, different file types are generated: *RectilinearGrid* files (with file ending `.vtr`) for the output of "regular fixed" meshes that represent a cartesian grid, *StructuredGrid* files (`.vts`) for the output of "structured deformable" meshes, i.e., structured meshes that can deform over time, *UnstructuredGrid* files (`.vtu`) for the output of unstructured meshes, and *PolyData* files (`.vtp`) containing connected points are used to represent multiple muscle fibers in a single file. ParaView can be used to load and visualize all of these file types.

All of these files are XML based and their payload data can be configured to be either written in ASCII representation or in Base64 encoding. Base64 encoding also translates the raw data into ASCII characters. The data stream is split into pieces of 6 bits, which are each represented by an 8-bit-ASCII character. Thus, the required memory is 4/3 of the raw data. Compared to a full ASCII representation containing the digits of all numerical values, this leads to a significant reduction of file sizes.

The VTK file format specifies parallel file output, where each process writes its local data to a separate file and one additional master file references the pieces in all files. This parallel file output scheme is implemented in OpenDiHu. However, it can lead to an impractically large number of small output files for high degrees of parallelism.

Therefore, we additionally implement a second approach, where non-parallel VTK files, which contain the whole dataset, are written. The same type of output files is generated during serial and parallel execution of the program. Writing the data to such a file is done using the parallel output capabilities of MPI. The respective MPI functions allow to collectively write data to the same file from different processes at different locations in the file. For parallel execution, every process only writes its own local data and no communication of the payload data to a master process is necessary.

Because the byte boundaries in a Base64 encoded data stream coincide with multiples of 8 bits only every three bytes, the processes that write neighboring parts in the output file have to coordinate the bit offsets of their data streams. For this, a small amount of data has to be communicated between these processes. However, the cost of this communication is negligible.

With this improved output scheme, one file is generated per mesh and output timestep of the simulation. The frequency of output timesteps can be configured in the Python settings. It is also possible and useful to combine all 1D fiber meshes into a single output file per timestep to reduce the number of output files.

Different meshes can be written with different frequency. For example, for a simulation of fiber based electrophysiology with EMG signals, it is reasonable to output the comprehensive dataset of all fibers less frequently than the smaller dataset of EMG signals on the 2D skin surface. To associate the output files with the correct times, a timestamp of the current simulation time is added to every file. Furthermore, partitioning information is added, i.e., which part of the mesh is computed by which process.

To synchronize output files of different meshes with different output frequencies in the visualization tool, additional *series files* (with file ending `.series`) are automatically created for every mesh. Such a file references all available output files of a mesh with their simulation times in JSON format. These files can be opened in ParaView to get a time-series of the simulation result.

Using the series files is also convenient, if the simulation is run in a directory, where old simulation results from previous runs exist. Because the series files are updated every timestep and only reference the newly created files, opening these files in ParaView only visualizes newly created simulation output, in contrast to opening a whole directory, which would potentially also load old results.

6.4.2 Visualization With ParaView

ParaView allows various manipulations and types of visualization of the loaded data. Figure 6.16 shows the ParaView window with simulation data of a fiber-based electrophysiology scenario. The loaded data are organized in a tree of datasets with applied filters, which can be seen in the “Pipeline Browser” in the top left. The center view shows a visualization of the muscle fibers and the 3D mesh at simulation time $t = 89.6\,\text{ms}$. An

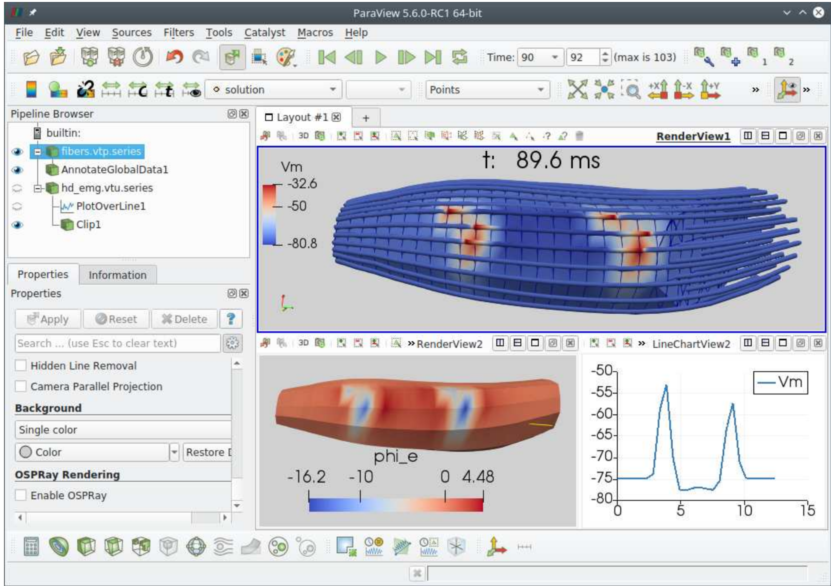


Figure 6.16: Visualization of simulation results with ParaView: ParaView window with a visualization of muscle fibers and a 3D muscle mesh.

animation of the transient data can be shown by using the playback controls in the top bar.

The visualization in the center top view displays the membrane voltage V_m at the fibers and in the 3D mesh, colored by the scheme shown at the left in the view. The 3D mesh is sliced on the right-hand side of the muscle to make the fiber dataset better visible.

The view on the bottom left depicts the extra-cellular potential ϕ_e on the 3D mesh. The view on the bottom right shows a plot of the value of ϕ_e along a horizontal line on the surface of the muscle.

It can be seen that three fibers near the surface are activated, and that the action potentials effect the EMG value given by ϕ_e on the surface.

For larger datasets, a head-less render server of ParaView also can be run in parallel on a remote server and the graphical user interface shown in Fig. 6.16 can be used as the client to interactively control the visualization.

ParaView also supports ray tracing using the OSPRay ray tracing engine. With ray tracing, more advanced lighting and the computation of shadows are possible.

6.4.3 ExFiles and Visualization with CMGUI

Another option in OpenDiHu is to output files in the “ExFile” format. This format originates from the software environment of OpenCMISS. Output of results in simulations with OpenCMISS Iron relies on this type of files. The visualization toolbox of OpenCMISS Zinc is able to create various visualizations of the data given in this format. The program *CMGUI* provides a graphical user interface to visualize the data.

The output consists of corresponding `.exelem` and `.exnode` files containing information at element and node level, respectively. The mesh is assumed to be unstructured and, thus, the information which nodes correspond to a particular element has to be explicitly stored. It is stored in the `.exelem` file. The payload data are contained in the `.exnode` file. The file format supports parallel output to separate files. However, only serial output is supported in OpenDiHu. ExFiles are ASCII-based and, thus, only usable up to a certain problem size.

An advantage of the “ExFile” format is that also higher order elements can be represented. The visualization tools are capable of representing the geometric data accordingly, e.g., it is possible to visualize the correct shape of cubic Hermite 3D hexahedral elements. In contrast, ParaView only visualizes linear elements and linearly interpolates the data between the nodes of an element.

The program CMGUI can be used to visualize the output files of OpenDiHu in ExFile format. In the graphical user interface, the `.exelem` and `.exnode` files can be loaded. Representations of loaded points, lines and elements can be added to the visualization in the scene editor. Various options such as coordinate frames and parameters for shading and tessellation can be set. The visualizations can be colored using predefined appearances or according to the loaded solution values.

For larger datasets, these manual adjustments are tedious. For example, for a dataset with 49 fibers, the user would have to load 49 `.exelem` and 49 `.exnode` files one by one. Instead, the Perl scripting interface of CMGUI can be used. Every command in the GUI corresponds to a Perl command and CMGUI can load and execute those commands from a given Perl script.

OpenDiHu automatically creates such Perl scripts. The generated script for a mesh loads all generated output files into CMGUI, adds a corresponding visualization depending on the mesh dimensionality and opens the required CMGUI windows such that the data are immediately visible. This is an improvement to OpenCMISS Iron, where all steps have to be done manually. By using the generated Perl script, less expert knowledge on the usage of CMGUI is required, and it is also possible to visualize datasets with a large number of fibers.

Figure 6.17 shows two windows of CMGUI. In Fig. 7.14d, the main graphics window can be seen with a visualization of 49 muscle fibers. The membrane potential is visualized by varying colors, and action potentials can be seen on three of the shown fibers. Similar to ParaView, the transient data can be animated by using the controls at the bottom. Figure 6.17b shows the spectrum editor, where the color scheme can be adjusted to the range of the loaded data.

The other Perl script besides the one used in Fig. 6.17 to visualize the muscle fibers addresses the 3D mesh of the muscle. Figure 6.18 shows the graphics windows with the resulting visualizations of this dataset. In Fig. 6.18a, the extracellular potential ϕ_e is visualized on the muscle surface. The visualization contains the colored 3D representation for the mesh and a 1D representation of the mesh consisting of white tubes.

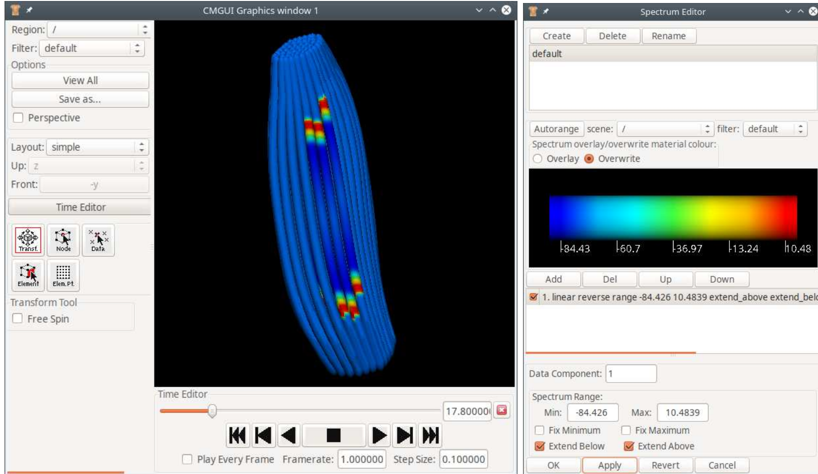
Figure 6.18b demonstrates the feature of visualizing nodal data using glyphs. The V_m values at every node are represented by colored circles with a radius that corresponds to the value. With this representation, it is possible to also show the data inside the muscle volume.

6.4.4 Python Output Files

Another option in OpenDiHu is to output data in a Python-friendly format, which can easily be parsed from within a python script. The data can then be used, e.g., for error analysis or to convert them to other custom formats.

If the format `PythonFile` is specified in the output writer, the data get written to output files. If the format `PythonCallback` is specified, the same data are passed to a callback function and can directly be used in the Python settings script during the running simulation.

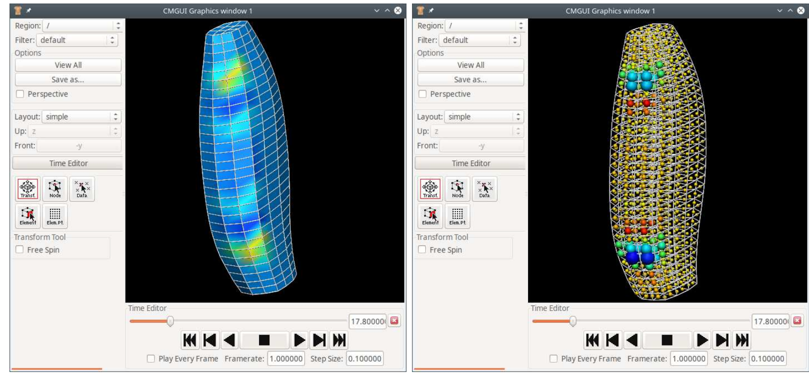
For output, the data are organized in a Python dictionary. The output files either contain the plain Python code of this dictionary or a binary representation obtained by the *pickle*



(a) The main graphics window that displays the visualization and allows to control the current view and the current timestep.

(b) The spectrum editor that can be used to adjust the coloring according to the loaded solution values.

Figure 6.17: Visualization of the results of an electrophysiology simulation with CMGUI involving 49 muscle fibers.



(a) Graphics window with the visualization of a 3D mesh.

(b) Visualization of the same data as in (a), but using sphere glyphs at every node.

Figure 6.18: Visualization of data on a 3D mesh with CMGUI.

package of Python. In parallel execution, every process writes its own file containing the data of the corresponding subdomain. OpenDiHu provides a Python module to parse these output files. The data representation, whether the data are stored in binary or in human-readable format and whether it is composed of multiple files resulting from parallel execution is abstracted and transparent in the call to this module.

The utility program `plot` can be used to quickly visualize the simulation results in such Python output files. It creates plots and animations of 1D and 2D structured meshes and chooses different layouts for the type of data, e.g, a plot over time for single-cell CellML models or an animation with multiple plots for subcellular models with multiple ion channels. This script is useful mainly for 1D and 2D toy problems, such as the Laplace, Poisson and Diffusion problems.

Figure 6.19 shows the output of the `plot` script for one muscle fiber. The top plot visualizes the geometry in 3D space, colored by the membrane potential V_m. The plot below shows the spatial progression of the V_m value along the x-axis. However, for the visualization of 3D data, other options such as ParaView or CMGUI are better suited and should be used instead.

6.4.5 ADIOS output files and MegaMol

Another output format is the binary-pack file format defined by the Adaptable Input Output System library (ADIOS2). This type of output is selected by the OpenDiHu output writers for the `"MegaMol"` format. ADIOS2 provides a framework for high-performance computing data management [God20]. ADIOS2 manages self-describing data that allows rapid metadata extraction also from large data sets.

Output files in this format can be loaded into the visualization software MegaMol by experts. MegaMol has been successfully used together with OpenDiHu to implement in-situ visualization, where OpenDiHu shares the computed simulation data with MegaMol using the ADIOS2 format and triggers updates of the visualization by sending asynchronous messages to MegaMol during the runtime of the simulation. As both OpenDiHu and MegaMol can run in parallel, the partitioned data needs to be merged from all processes only at the stage of rendering the visualization image. For highly parallel runs on supercomputers, the local data that are generated by the OpenDiHu processes on the same compute node can be shared in memory with one instance of MegaMol per compute node. Then, all MegaMol instances collectively render the resulting visualization. This

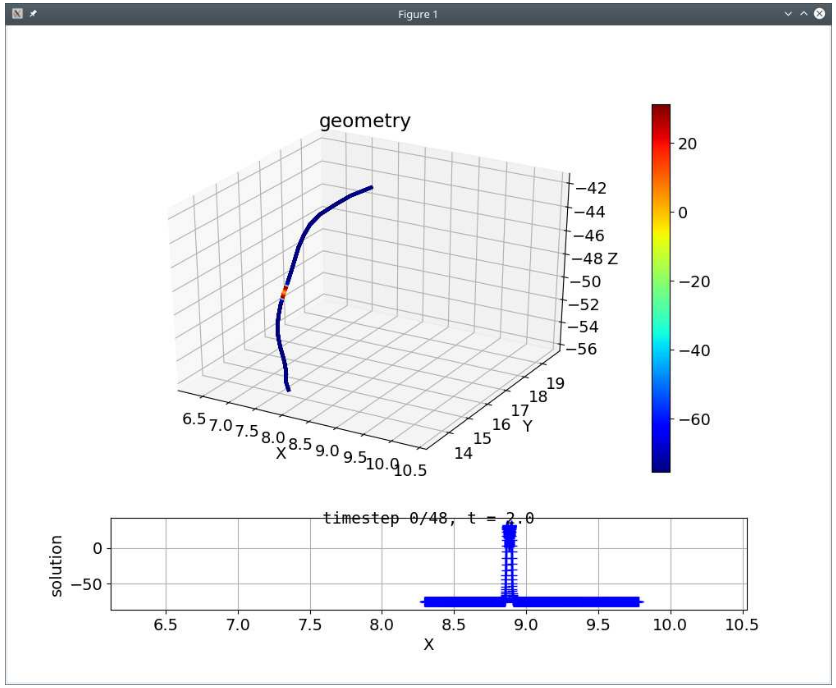


Figure 6.19: Visualization of Python based simulation results using the `plot` utility.

approach bypasses the costly file output operation on the highly distributed file system of a supercomputer.

The generated output files can be inspected using the `bpls` utility. Figure 6.20 shows a description of the 3D dataset extracted from the binary-pack format that was written by a simulation with four processes. Each line corresponds to one variable in the file. The first column specifies the variable type and the second column is the name of the variable. The third column contains structural information with minimum and maximum values for numeric types.

The first three shown variables are of type `string` and contain metadata for the simulation run. The `config` variable contains the Python settings code of the scenario and, thus, accurately describes the settings of the simulation run. The values of the `meta` and `version` variables are fully listed in Fig. 6.20 and contain meta information about the simulation program and the particular run.

```
1   string    config                            (...)
2   string    meta                              "current time: 2021/3/30 19:48:05,↩
3                                                hostname: lapsgs05, n ranks: 4"
4   string    version                           "opendihu 1.2, built ↩
5                                                Mar 27 2021, C++ 201402, GCC 7.5.0"
6   double    localBoundingBox                  10*{4, 6} = -56.3 / 19.7732
7   double    globalBoundingBox                 10*{6} = -56.3 / 19.7732
8   double    global_radius                     10*scalar = 0.1 / 0.1
9   int32_t   nPointsPerCoordinateDirection     10*{3} = 4 / 31
10  int32_t   nodeOffsetOnOwnComputeNode        10*{4} = 0 / 368
11  int32_t   node_count                        10*scalar = 496 / 496
12  int32_t   rankNo                            10*{496} = 0 / 3
13  double    emg                               10*{496} = -12.0536 / 4.89757
14  double    transmembraneFlow                 10*{496} = -125.014 / 226.428
15  double    vm                                10*{496} = -81.3198 / -27.4762
16  double    xyz                               10*{1488} = -56.3 / 19.7732
```

Figure 6.20: Contents of the output file created by ADIOS2.

For the numeric values, the third column specifies the dimension of the stored data. The file contains the simulation output for 10 different timesteps, which can be seen in the third column. For example, the `localBoundingBox` variable in line 6 stores 10 instances of a matrix with dimension 4×6. The four rows of this matrix correspond to the four processes and the columns store the six values of the geometric bounding box of the subdomain on the respective process. This information is required by MegaMol to constrain the volume that has to be rendered on each process. Further structural information is contained in the variables in lines 7 to 12. The remaining variables contain the payload data. The variables `emg`, `transmembraneFlow` and `vm` correspond to ϕ_e, the right-hand side of the first bidomain equation in Eq. (5.71), and V_m, respectively. The variable `xyz` holds the geometry information for all nodes.

How To Reproduce

The visualizations in this section are based on outputs of the following simulation:

```
cd $OPENDIHU_HOME/examples/electrophysiology/fibers/fibers_emg/
  ↪ build_release
./fast_fibers_emg ../settings_fibers_emg.py output_demo.py
```

Output files for ADIOS2, CMGUI, ParaView and Python will be generated in corresponding subdirectories under `out/`. The following commands invoke the respective visualization tool in the corresponding output directory:

```
paraview fibers.vtp.series              # ParaView
cmgui fibers.com                        # CMGUI
cmgui hd_emg.com                        # CMGUI
plot fibers_0000001_MeshFiber_*.py      # Python
bpls hd_emg.bp -la                      # ADIOS2
```

In the graphical user interfaces of CMGUI and ParaView, more settings have to be adjusted to obtain the results shown in Figures 6.16 to 6.18.

The listing shown in Fig. 6.20 was obtained by a simulation with 4 processes. Because the ExFile output writer does not work for parallel execution, the corresponding option has to be disabled in the `output_demo.py` variables file prior to execution:

```
mpirun -n 4 ./fast_fibers_emg ../settings_fibers_emg.py output_demo.
  ↪ py
```

Afterwards, the shown listing can be obtained by `bpls -la hd_emg.bp`.

Chapter 7

Implementation of the Software OpenDiHu

After the usage of OpenDiHu has been described in the last chapter, we now discuss the implementation of the algorithms and solvers that are available in the framework. This chapter begins with the basic data organization in Sec. 7.1 and generic algorithms to set up finite element discretizations and parallel partitionings of a problem in Sections 7.2 and 7.3. Then, details are given on the implementation of particular solvers. Section 7.4 discusses the solvers for the fiber based electrophysiology model, Sec. 7.5 addresses the multidomain solver and Sec. 7.6 presents optimizations for the solver of the subcellular model. The chapter closes in Sec. 7.8 with a discussion of the data mapping required in coupling schemes.

7.1 Data Handling with PETSc

OpenDiHu processes various types of data: geometry data, the discretized solution data, system matrices and vectors in the specification of the mathematical model such as right-hand sides and prescribed values in boundary conditions. All these data need to be organized in accordance with the parallel partitioning. Linear system solvers need to be applied on matrices and vectors to obtain the solution. The result of the simulation has to be invariant under a change of the number of processes that execute the program.

For parallel data handling and solvers of linear and nonlinear systems, the *Portable, Extensible Toolkit for Scientific Computation (PETSc)* [Bal16; Bal15; Bal97] is used. PETSc provides a large collection of solvers and preconditioners that can be selected and configured at runtime. More solvers are accessible through interfaces to external software, such as the *Multifrontal Massively Parallel Sparse Direct Solver (MUMPS)* [Ame01; Ame19] and

the preconditioner library *HYPRE* [Fal02]. PETSc natively supports MPI parallelism and provides parallel data structures for vectors and matrices. Numerous operations on the data are provided including value communication and access, housekeeping, arithmetical operations, and more advanced calculations in the field of linear algebra.

Since MPI is used, processes can be identified by their *rank* r within the used *MPI communicator*. An MPI communicator is a subset of processes that can communicate with each other. The rank of a process is its number in this communicator, i.e., a consecutive number starting with zero.

7.1.1 Organization of Parallel Partitioned Data

Basic building blocks in the implementation of OpenDiHu are *field variables* that represent scalar fields. A scalar field $v : \Omega \to \mathbb{R}$ defined on a domain $\Omega \subset \mathbb{R}^3$ is represented in the program by its finite element discretization. It comprises, on the one hand, the specification of the mesh of Ω, i.e, the node positions, elements and ansatz functions and on the other hand the values of the coefficients of the ansatz functions. The values of the coefficients are called *degrees of freedom (dof)*. Meshes with linear ansatz functions have one dof on every node. In the following, regular Cartesian meshes with linear ansatz functions are considered.

The partitioning of a regular, d-dimensional mesh is constructed as follows. A partitioning in terms of number of processes is given in the form $n_x \times n_y \times n_z = n_{\text{proc}}$, where n_x, n_y and n_z are the number of processes or subdomains in x, y and z direction, respectively. For 2D meshes, n_z is set to one, for 1D meshes, n_y and n_z are set to one. The given mesh is partitioned on the level of elements. In every coordinate direction $i \in \{x, y, z\}$, the number N_i^{el} of elements is equally distributed to the specified number n_i of processes. Every process gets either $\lfloor N_i^{\text{el}}/n_i + 1\rfloor$ or $\lfloor N_i^{\text{el}}/n_i\rfloor$ elements, where the larger number of elements is assigned to the processes with lower ranks. Thus, the subdomains with smaller index in x, y and z direction potentially have one layer of elements more than other subdomains.

For example, in Fig. 7.1, a 1D mesh with $N_x^{\text{el}} = 6$ elements is partitioned into three subdomains with two elements each. Figure 7.2 (a) shows a 2D mesh with $N_x^{\text{el}} \times N_y^{\text{el}} = 5 \times 4$ elements, a partitioning to $n_x \times n_y = 2 \times 3$ processes is given in Fig. 7.2 (b).

The nodes of the mesh are assigned to the same subdomains as their adjacent elements. The assignment of the nodes that lie on the cutting planes between the subdomains

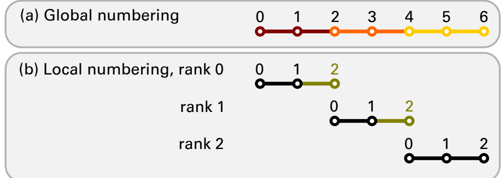


Figure 7.1: Partitioning and local and global numbering of a 1D mesh with $N_x^{\mathrm{el}} = 6$ elements partitioned to $n_x = 3$ processes. Ghost nodes are marked in green in (b)

remains to be specified. These nodes are assigned to the subdomain of the adjacent element in positive x, y and z direction such that each of these nodes is also owned by a single rank. On all other adjacent ranks, the node is stored as so called *ghost* node. In contrast, the other local nodes are called *non-ghost* nodes in the following.

The assignment of nodes to processes leads to the situation, that subdomains with the highest index in x, y and z direction (i.e., the subdomains at the "right", "top", or "back" end of the domain) potentially have one layer of nodes more than other subdomains. This effect is intentionally chosen to be balanced out by our strategy to assign a higher number of elements to subdomains with lower index. Therefore, the total number of nodes and dofs is approximately equally distributed. Moreover, in the limit for $N_x^{\mathrm{el}}, N_y^{\mathrm{el}}, N_z^{\mathrm{el}} \to \infty$, the imbalance vanishes totally.

In the exemplary partitionings in Fig. 7.1 (b) and Fig. 7.2 (b), owned nodes are represented by black circles and numbers, ghost nodes are represented by green circles and numbers. In the 1D example in Fig. 7.1, the three subdomains with two elements each have three, two and two nodes. In the 2D example in Fig. 7.2, the six subdomains have either three (ranks 2 and 3) or six (ranks 0,1,4 and 5) nodes while the number of elements varies between six (rank 0) and two (rank 5). This demonstrates the construction of nearly equally sized subdomains in terms of the number of assigned nodes.

On the partitioned meshes, field variables can be defined to represent the scalar and vector fields in the FEM computations. A field variable in OpenDiHu manages its values using the basic PETSc data type for storing scalar fields: the *Vec*. It represents a vector $\tilde{\mathbf{v}} \in \mathbb{R}^{n_{\mathrm{global}}}$ with n_{global} values. The vector is distributed to n_{proc} processes according to the partitioning of the mesh, such that every value is owned by exactly one process.

In a PETSc `Vec`, every rank r locally stores a distinct portion of $n_{\mathrm{local_without_ghosts}} \leq n_{\mathrm{global}}$ values of the global vector of dofs. Therefore, every dof is *owned* by exactly one rank.

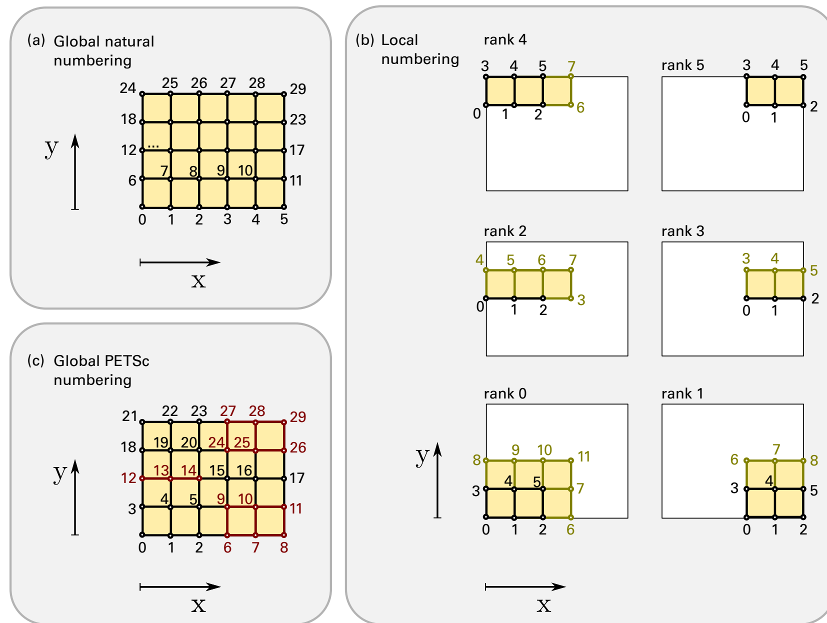
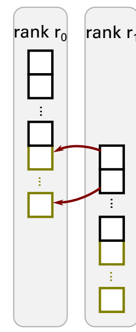


Figure 7.2: Subdomains and numberings of a 2D mesh with $N_x^{\text{el}} \times N_y^{\text{el}} = 5 \times 4$ elements partitioned to $n_x \times n_y = 2 \times 3$ processes. (a)-(c) show the different numberings needed for (a) boundary condition specification, (b) identification of local non-ghost dofs (black) and local ghost dofs (green), and (c) identification of global dofs.

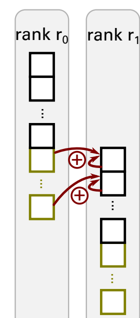
These dofs correspond to the local nodes in the partitioning. Additionally, the process maintains storage for n_{ghosts} ghost dofs that are owned by other ranks. PETSc is able to communicate corresponding values between all ranks where the dof is present either as ghost or non-ghost dof.

In total, the local buffer of a Vec stores $n_{\text{local_with_ghosts}} = n_{\text{local_without_ghosts}} + n_{\text{ghosts}}$ values. The non-ghost dofs are located at array positions $0, \dots, n_{\text{local_without_ghosts}} - 1$, the ghost dofs follow at positions $n_{\text{local_without_ghosts}}, \dots, n_{\text{local_with_ghosts}} - 1$. This array is consecutive in memory. The latter part for the ghost dofs is called the *ghost buffer*.

The local dofs in every subdomain are numbered according to the layout of this buffer. Figure 7.1 (b) shows the local dof numbering on the three ranks. It proceeds through all non-ghost dofs followed by the ghost dofs. A global numbering of all dofs is given in



(a) `startGhostManipulation()`: Communication of values at r_1 to the ghost buffer at r_0.



(b) `finishGhostManipulation()`: Communication and addition of ghost values at r_0 to the non-ghost dofs at r_1.

Figure 7.3: Communication operations for ghost values in an example with two ranks r_0 and r_1. Depicted are the vectors of local storage for non-ghost (black) and ghost values (green). The red arrows indicate the data transfer. The visualized operations are needed, e.g., in the assembly of finite element stiffness and mass matrices and their application on the vector of unknowns.

Figure 7.1 (a). It is needed, if global operations have to be performed with the `Vec`, e.g., computing matrix vector products.

In the following, we outline how the finite element stiffness and mass matrices are assembled in parallel. The algorithm proceeds by iterating over the elements of the mesh. The contributions to the matrix, i.e., the "element matrices", are computed at the dofs of every element. Additional material data, such as values of a diffusion tensor, may be stored at the dofs and are used in these computations.

The step of assembling the global matrix entries adds up the contributions of all elements that are adjacent to every dof. This includes a parallel reduction operation for the ghost dofs, which contribute to matrix entries that are owned by a different subdomain.

PETSc provides specific functionality for the two required communication operations: (i) gathering data into the ghost buffers on every rank, i.e., communicating the values from the owning rank to the ghost buffers at all other ranks, where the respective dofs are ghosts, and (ii) the global reduction of values between ghost and non-ghost dofs, i.e., communicating the values from the ghost buffers back to the one rank, where they are non-ghosts and adding their values to the values present at the respective rank.

The OpenDiHu code wraps the two operations in the methods `startGhostManipulation()` and `finishGhostManipulation()`. After the call to `startGhostManipulation()`, the vector can be accessed using the local dof numbering. Values of the local dofs including ghosts can be retrieved, inserted or added as needed, e.g., during FEM matrix assembly. After a concluding call to `finishGhostManipulation()`, the vector is in a valid global state. Then, global operations such as adding or scaling the whole vector, computing a norm or a matrix vector product can be performed by using the respective PETSc routines. For these operations, the partitioning is transparent, i.e., the calls are the same for serial and parallel execution. Individual entries of the vector can now be accessed using a global numbering. However, every process can still only access the non-ghost dofs owned by its subdomain. The two operations can be interpreted as switching between a local and a global view on the vector object.

One thing to note is that calling `startGhostManipulation()` and `finishGhostManipulation()` directly in sequence changes the values of the vector. The reason is that during the call to `startGhostManipulation()`, the ghost buffers get filled with the ghost values from other subdomains. Then, by `finishGhostManipulation()` the values in every ghost buffer get summed up and added to the value at the corresponding non-ghost dof. Thus, these dof values finally have a multiple of their initial value. This is usually not intended. Thus, between the calls to the two methods either all ghost values have to be set, such as during computation of the stiffness matrix. Or, if the ghost values were only needed for reading instead of updating them, the ghost buffers have to be cleared to zero. For the latter, a helper method `zeroGhostBuffer()` exists. A typical usage is therefore to call `startGhostManipulation()`, then operate on the local dof values including ghosts, and then finish with `zeroGhostBuffer()` and `finishGhostManipulation()`.

7.1.2 Numbering Schemes for Nodes and Degrees of Freedom

PETSc's definition of the local value buffer used by `Vec` objects dictates the local numbering scheme of dofs on meshes of any dimensionality. While, for 1D meshes, the numbering as given in Fig. 7.1 seems natural, for 2D and 3D meshes, a more complex ordering of local dofs is needed.

Three different numbering schemes for nodes and dofs exist within OpenDiHu. They are visualized in Fig. 7.2 for a 2D mesh. The first is the *global natural* numbering scheme, which numbers all $n_{\text{global}} = N_x^{\text{dofs}} \times N_y^{\text{dofs}} \times N_z^{\text{dofs}}$ global dofs in the structured mesh. It starts with zero and iterates through the mesh using the triple of coordinate indices (i, j, k)

for the x, y and z axis with the ranges $i \in \{0,\dots,N_x^{\text{dofs}}-1\}$, $j \in \{0,\dots,N_y^{\text{dofs}}-1\}$ and $k \in \{0,\dots,N_z^{\text{dofs}}-1\}$. The numbering proceeds fastest in x or i direction, then in y or j direction and then in z or k direction. Examples are shown in Fig. 7.2 (a) for a 2D mesh and in Fig. 7.4 for a 3D mesh.

The intention of this first numbering is to facilitate the problem description by the user. If values for a variable in the whole computational domain should be specified, the order of the given value list will be interpreted according to this numbering. Boundary conditions can be given for some dofs by simply specifying the corresponding dof numbers in global natural numbering. The advantage is that this numbering scheme is easily understandable from a users' perspective and independent of the partitioning.

The second numbering scheme is the *local* numbering. An example is given in Fig. 7.2 (b). It specifies the order of dofs in the local PETSc `Vec` and is defined locally on every subdomain for the non-ghost and ghost dofs. At first, all non-ghost dofs are numbered with the order equal to the one in the global natural scheme. Then, all ghost dofs are numbered, again in the order of the global natural scheme. This numbering has the counter-intuitive property of jumps between some neighboring nodes.

The third numbering scheme is called *global PETSc* numbering and is defined by PETSc. It is the numbering used to access global `Vec`s. It is also the ordering of the rows and columns of matrices. The numbering starts with all local non-ghost numbers on rank 0, then proceeds over all non-ghost numbers of rank 1 and continues like this for all remaining ranks. An example for this numbering is given in Fig. 7.2 (c). The portions of local dofs for the different ranks are indicated by the grid of red and black colors. This numbering depends on the partitioning and, thus, on the number of processes. For serial execution it is identical to the global natural numbering.

7.1.3 Parallel Data Structures in OpenDiHu

All operations on scalar and vector fields in the simulation break down to manipulating variables of the `Vec` type provided by PETSc. Because this involves low level operations such as working with different numbering schemes and communicating ghost values, an abstraction layer on a higher level is implemented in OpenDiHu. The data handling classes are visualized in Fig. 7.5 with the data representation in raw memory at the top and increasing abstraction towards the bottom of the figure.

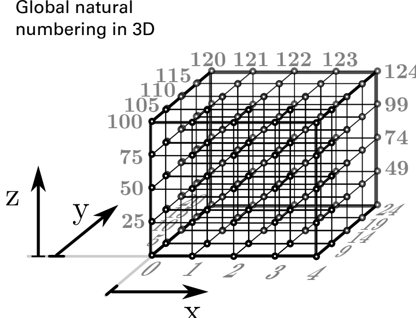


Figure 7.4: Global natural numbering of nodes in a mesh with $4 \times 4 \times 4$ linear elements.

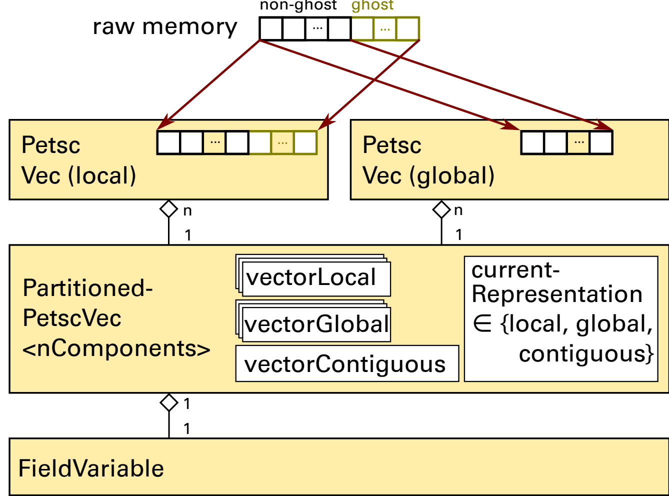


Figure 7.5: Classes in OpenDiHu that represent vectors in parallel execution. The abstraction layer increases from raw memory at the top to the `FieldVariable` at the bottom.

Depending on whether the local or the global PETSc numbering scheme describes the data, two different objects of the `Vec` type are used: one local and one global `Vec`. In Fig. 7.5, these two `Vec`s are represented by the *PETSc Vec (local)* and *PETSc Vec (global)* boxes. At any time, only one of these is in a valid state and allows to manipulate the data. Internally, both PETSc `Vec`s use the same memory to store their data. However, as shown at the top of Fig. 7.5, the memory range of the local `Vec`'s buffer includes the ghost buffer, which is never accessed by the global `Vec`. As mentioned, PETSc functions are available to switch the valid state between the two `Vec`'s, involving communication of ghost values.

The next abstracting class is *`PartitionedPetscVec<nComponents>`*. It represents a discretized vector field $\mathbf{v} : \Omega \to \mathbb{R}^c$ with a given number of components c. The number of components c is a template parameter to the class, which has to be specified at compile time. An example for such vector fields is the *geometry field* with $c = 3$, which is defined for every mesh and specifies the node positions. Another example is the solution variable of the considered problem. For scalar problems such as the Laplace equation, it has $c = 1$ component, for vector-valued problems, e.g., static elasticity, it has $c = 3$ components, namely the displacements in x, y and z direction. For the subcellular model of Shorten [Sho07], the solution variable, i.e., the vector of states has $c = 57$ components.

For each component, a separate pair of local and global `Vec`s is stored in the variables *`vectorLocal`* and *`vectorGlobal`*. The global number of entries in each of the `Vec`s is given by the number n_{global} of dofs in the mesh that discretizes the domain Ω. Thus, the memory layout of such a multi-component vector is struct-of-array (SoA).

Besides `vectorLocal` and `vectorGlobal`, a third variable *`vectorContiguous`* of type `Vec` exists in the class `PartitionedPetscVec`. It contains the concatenated values of all component vectors in `vectorGlobal`. Its size is therefore $c \cdot n_{\text{global}}$ and the layout is again SoA but stored in a single `Vec`.

This representation is chosen when a timestepping scheme operates on a state vector with multiple components. An example is the solution of multiple instances of a subcellular problem. Here, the dofs in the mesh correspond to the individual instances and the components are the state variables of the system of ODEs. Thus, the contiguous vector begins with the values of the first state for all instances, then stores the values of the second state for all instances, etc. If the right-hand side of the system of ODEs is evaluated together for all instances, this memory layout is very efficient as it leads to a cache aware access pattern.

Only one of the three vectors `vectorLocal`, `vectorGlobal` and `vectorContiguous` is valid at any time and can be used to retrieve or update the vector values. A state variable *`currentRepresentation`* in `PartitionedPetscVec<nComponents>` indicates which one that is. The state and the `Vec` variables are encapsulated and hidden in the class, i.e., not directly accessible from outside. Instead, the class provides data access methods and ways to change the internal representation. For example, calls to `startGhostManipulation()` and `finishGhostManipulation()` change the representation from global to local and from local to global, respectively. Thus, it is ensured that only the current valid representation gets accessed at any time.

As noted before, the change between local and global representation does not involve data copying because of the shared physical data structures. When the representation is changed from local to contiguous, the c sets of values of the `vectorLocal` variables have to be copied into the buffer of `vectorContiguous`. This operation is performed by copying memory blocks (`memcpy`) instead of the slower iteration over all values and the value-wise copy. The reverse change from contiguous back to local representation happens analogously. Thus, the change between all representations is fast. Despite occurring often during transient simulations, profiling of simulations has shown negligible runtime for the action of switching between these representations.

The top level class in the value storage hierarchy as shown in Fig. 7.5 is the *`FieldVariable`*, which contains a `PartitionedPetscVec` and adds numerous methods to facilitate access to the data container. Model formulations use this class to manipulate scalar and vector fields. At the same time, the underlying global PETSc `Vec` can still be obtained from a `FieldVariable`. Vector operations such as addition, norms and matrix-vector products are performed using the low-level PETSc functions on the global `Vec` obtained from the `FieldVariable`s.

7.1.4 Discussion of Several Design Decisions

In the following, some of the design decisions in Sections 7.1.1 to 7.1.3 are discussed. In the present code, PETSc functionality is used for value storage and organization of ghost values transfer. The employed PETSc data model naturally corresponds to a 1D mesh. The representation of arbitrary dimensional meshes is added by OpenDiHu and involves the presented local and global PETSc numberings.

PETSc also provides the management of abstract 2D and 3D mesh objects in the `DM` (data management) module. It allows to automatically create a partitioning with local

numberings and data vectors. However, the mesh always has a symmetric ghost node layout, where ghost layers are present on all faces of a subdomain (box stencil) or also at diagonal neighbors (star stencil). This partitioning layout is based on distributing the nodes of the mesh to all processes. It is needed, e.g., for Finite Difference computations. For the finite element method, however, we need an element based partitioning with ghost layers only on one end of the mesh per coordinate direction. Therefore, we do not use this functionality of PETSc and instead implemented the numberings for 2D and 3D meshes on our own.

Another choice was made regarding the data layout in the `PartitionedPetscVec` class. Instead of an interleaved storage of the component values in one long `Vec` in array-of-struct (AoS) memory layout, one separate `Vec` for each component is stored, which corresponds to SoA layout. Thereby, the implementation differs from OpenCMISS Iron, which is also based on PETSc, but uses the AoS approach.

In Iron, not only the values of multiple components, but actually the values of multiple field variable are combined into a single `Vec`. A local numbering is defined that enumerates all components, all dofs, and all field variables. Differences to our code are, that Iron uses unstructured meshes, which additionally are allowed to contain different types of elements in a single mesh. Field variables can be defined with dofs either associated with nodes or with elements. All these possible variations are accounted for by the local numbering. The construction of the numbering is, thus, a complex process. Iron implements it by a loop over all n_{global} dofs of the domain. The same loop is executed in parallel by all n_{proc} processes. The runtime complexity of this approach is $\mathcal{O}(n_{\text{global}})$ regardless of the partitioning. In contrast, OpenDiHu constructs its local numberings separately on each process and only iterates over the $n_{\text{local_with_ghosts}}$ dofs, leading to a runtime complexity of $\mathcal{O}(n_{\text{local_with_ghosts}}) = \mathcal{O}(n_{\text{global}}/n_{\text{proc}})$. In a weak scaling experiment with constant relation $n_{\text{global}}/n_{\text{proc}}$, the approach of Iron yields infinite runtime in the limit for $n_{\text{global}} \to \infty$, whereas the runtime in the approach of OpenDiHu stays constant.

For OpenDiHu, the AoS approach with separate `Vec`s was chosen for three reasons. First, it is more cache efficient than the alternative during the computation of the subcellular model, as explained in Sec. 7.1.3.

Second, the AoS structure is easier, and it allows to treat the components separately, which makes modular code possible. Only a single local dof numbering has to be constructed per mesh, and it can be reused for all components of all field variables.

Third, it is possible to extract one component of a vector-valued field variable and place it into another, scalar field variable without copying. This is used during the solution of

the monodomain equations given in Eq. (5.11). There, the subcellular models have a vector-valued solution variable and the diffusion problem needs a scalar solution variable that consists of the first component of the vector-valued variable of the subcellular model. This first component is the transmembrane voltage V_m. The program needs to switch between these two required vectors in every timestep of the splitting scheme. Only with the chosen representation by multiple `Vec`s, the `Vec` for the particular component can be efficiently exchanged between the two field variables without an expensive copy operation.

Another design decision was to make the number c of components fixed at compile time. Upon construction of a new `FieldVariable`, its number of components needs to be known. Typically, this is the case and does not pose any restriction. The main advantage is that local variables that hold all components for a given dof can be allocated on the stack instead of a much slower dynamic allocation on the heap. For example, in a dynamic solid mechanics problem, the solution `FieldVariable` contains three components each for displacements and velocities plus one component for the pressure, in total $c = 7$ components. The program can use static arrays with seven entries as temporary variables to handle these values in various computations. If the number of components was not fixed at compile time, a costly dynamic allocation of the seven components would be needed wherever values of the `FieldVariable` are retrieved. In addition, with a compile-time fixed c the compiler knows the size of the arrays and can perform automatic optimizations such as vectorization and loop unrolling.

The C++ implementation of `FieldVariable`s and all other constructs that depend on the number of components is generic, as the c value is a template argument. Specializations for particular numbers of components such as for the scalar case $c = 1$ are possible using *template specialization*. This flexibility while using object orientation is an advantage over codes using procedural programming languages such as the Fortran standard used by OpenCMISS Iron. It contributes to the extensibility design goal of OpenDiHu.

7.1.5 Implemented Basis Functions

In the FEM, the number of dofs and nodes per element depends on the chosen ansatz functions or basis functions. OpenDiHu supports linear and quadratic Lagrange as well as cubic Hermite basis functions. Table 7.1 shows these three sets of functions and the resulting node configuration of an element in a 1D, 2D and 3D mesh. Profiling showed that evaluation of the basis functions contributes most to the runtime during calculation

Ansatz functions	Element shapes 1D 2D 3D
$\phi_0(\xi) = 1 - \xi$, $\phi_1(\xi) = \xi$	
$\phi_0(\xi) = (2\,\xi - 1)(\xi - 1)$, $\phi_1(\xi) = 4(\xi - \xi^2)$, $\phi_2(\xi) = 2\,\xi^2 - \xi$	
$\phi_0(\xi) = 2\,\xi^3 - 3\,\xi^2 + 1$, $\phi_1(\xi) = \xi\,(\xi - 1)^2$, $\phi_2(\xi) = \xi^2\,(3 - 2\,\xi)$, $\phi_3(\xi) = \xi^2\,(\xi - 1)$	

Table 7.1: Finite element ansatz functions and resulting element shapes of hexahedral meshes in 1D, 2D and 3D. From top to bottom: Linear Lagrange, quadratic Lagrange and cubic Hermite ansatz functions.

of the stiffness matrix. Therefore, care was taken to choose the formulations of the basis functions among different factorizations that need the least operations. Those are listed in Tab. 7.1.

In the program, every basis function is defined by a class that specifies the constant, static numbers $n_{\text{dofs_per_basis}}$ of dofs per 1D element and $n_{\text{dofs_per_node}}$ of dofs per node. Furthermore, the actual functions and their first derivatives are implemented. All algorithms working with meshes or ansatz functions only use this information given in the basis function class. Therefore, it is easily possible to introduce new nodal ansatz functions as needed, e.g., a cubic Lagrange basis, by accordingly defining a new class.

If any Lagrange basis is used, every node has exactly one dof, i.e., $n_{\text{dofs_per_node}} = 1$. With the 1D Hermite basis, every node has $n_{\text{dofs_per_node}} = 2$ dofs, one that describes the function value and one that defines the derivative at the particular node. For higher dimensional meshes, the bases are constructed by the tensor product approach. For 2D meshes, this results in four and for 3D meshes in eight dofs per node for the Hermite basis. For example, at a node at location $\mathbf{x}$ in a 2D mesh, the first dof describes the value $f(\mathbf{x})$ of a scalar field $f : \Omega \to \mathbb{R}$ and the others relate to the derivatives $\partial_x f(\mathbf{x})$, $\partial_y f(\mathbf{x})$ and $\partial_{xy} f(\mathbf{x})$.

Note that the dof values for derivatives only match the real derivatives of f in meshes

with unity mesh widths. In a general, the derivatives are scaled by the element lengths. In general meshes with varying element sizes, the represented FE solution f is not continuously differentiable at element boundaries, i.e., $f \in \mathcal{C}^0(\Omega, \mathbb{R})$.

For quadratic Lagrange and cubic Hermite basis functions, the numbering schemes presented in Sec. 7.1.2 have to be adjusted, such that, at every node, all dofs are enumerated in sequence before the numbering continues at the next node.

7.1.6 Implemented Types of Meshes

Meshes of different types can be selected independently of the choice of basis functions. Three types are supported. Figure 7.6 visualizes meshes of these types with linear and quadratic elements.

The first type is `Mesh::RegularFixedOfDimension<D>` where `D` $\in \{1,2,3\}$ is a compile-time constant of the dimension. This type describes a rectilinear, regular structured mesh that is defined by a fixed mesh width h in all coordinate directions. This mesh is "fixed", which means that the positions of the nodes cannot change after the mesh object was created. Regular fixed meshes describe a line (1D), a rectangular (2D) or a cuboid domain (3D). This mesh type exists, because such domains are often used in exemplary problems to study certain effects independently of the shape of the domain. A regular fixed mesh can be easily configured by specifying origin point coordinates, mesh widths and number of elements. For this mesh type, matrix assembly in the FEM is simplified and more efficient by using precomputed stencils.

The second mesh type is `Mesh::StructuredDeformableOfDimension<D>`. The structured deformable mesh is a generalization of the regular fixed mesh. The mesh again has a structure of $N_x^{\text{el}} \times N_y^{\text{el}} \times N_z^{\text{el}}$ elements. Contrary to the regular fixed mesh, the nodes can now have arbitrary positions. In the name of this mesh, "deformable" indicates that the node locations can be changed over time. Thus, this mesh type is usable in dynamic solid mechanics problems, where the domain deforms over time. If the user wants to configure a mesh of this type, they either have to provide the same information as for regular fixed meshes—then, a mesh with fixed mesh width will be created—or they provide the positions of all nodes, yielding an arbitrarily shaped domain as shown in Fig. 7.6.

The third mesh type is `Mesh::UnstructuredDeformableOfDimension<D>`. In contrast to the two other types, this mesh is unstructured implying that element adjacency is no longer given implicitly. The example at the lower third of Fig. 7.6 shows capabilities of

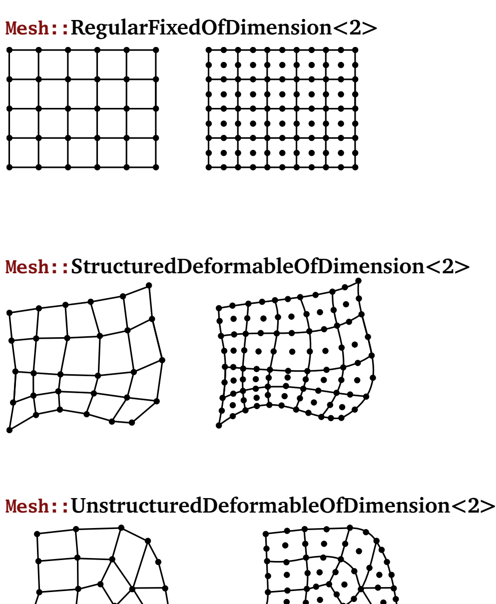


Figure 7.6: The three implemented mesh types in OpenDiHu, each time for 2D linear Lagrange or Hermite ansatz functions (left) and for 2D quadratic Lagrange ansatz functions (right).

this mesh type: The overall shape of the domain is not restricted to resemble a rectangle. Protruding parts like the element at the bottom left are possible. Furthermore, not every node needs to be adjacent to exactly four elements in 2D. The example shows nodes with three and five adjacent elements that allow to properly approximate the round shape of the right side of the domain. The mesh is again "deformable", which means that it can be used for elasticity problems. In order to configure such a mesh, the node positions have to be specified, similar to a structured deformable mesh. Additionally, the elements with links to their corresponding nodes have to be given. OpenDiHu implements a second possibility to specify these meshes. A pair of `exelem` and `exnode` files, which are common in the OpenCMISS community, can be loaded.

A disadvantage of unstructured meshes is that the simple parallel partitioning scheme of subdividing the domain according to element index ranges is not applicable. Instead, the set of elements for every subdomain needs to be computed individually. Typically,

this is done using graph partitioning methods in order to minimize subdomain border lengths while ensuring equal subdomain sizes. Another disadvantage is that information about neighbor elements and neighbor subdomains has to be stored explicitly, while it is given implicitly in structured meshes. For these reasons, unstructured meshes can be used in OpenDiHu only for serial computation. The construction of parallel partitionings is only possible with the other two, structured mesh types.

The choice, which mesh type to use in a simulation, has to be made at compile time. A simulation program can be easily compiled for different meshes by substituting the type in the main C++ source file. By proper abstraction in the code, all implemented algorithms are independent of the used mesh type when run in serial. Some algorithms, e.g., streamline tracing, are specialized for structured meshes to exploit the structure and lead to more efficient code. Unit tests ensure the correct solution of a Laplace problem with all combinations of mesh type, dimensionality and ansatz function.

7.1.7 Composite Meshes

To overcome the limitations of structured meshes regarding possible domain shapes and, at the same time, preserving the advantage of efficient parallel partitioning, *composite* meshes are introduced. These meshes of type `Mesh::CompositeOfDimension<D>` are built using multiple meshes of type `Mesh::StructuredDeformableOfDimension<D>`, called *submeshes* in this context. The structured submeshes are positioned next to each other to form a combined single mesh on the union of the domains of all meshes. Figure 7.7a shows a 2D example where three structured meshes are combined to a composite mesh. As can be seen, the submeshes can have different numbers of elements. The nodes on the borders between touching structured meshes are shared between the individual meshes. Thus, these nodes contain only a single set of dofs like every other node in the mesh.

In the code, composite meshes reuse the implementation of structured meshes by defining different numbering schemes for nodes and dofs over the whole composite domain. The numbering of nodes starts with all nodes of the first submesh, then proceeds over all remaining nodes of the second submesh and so on, until all nodes are numbered. The numbering of dofs is analog. Figure 7.7b shows an example with two quadratic submeshes with four and two elements. The resulting composite mesh has six elements. The node numbers in the first structured mesh are identical to the corresponding nodes in the composite mesh. The numbering continues in the set of remaining nodes of the second structured mesh and the shared nodes on the border between the meshes are

skipped in the numbering, as they already have a number assigned. The shared nodes have the numbers 14, 19 and 24.

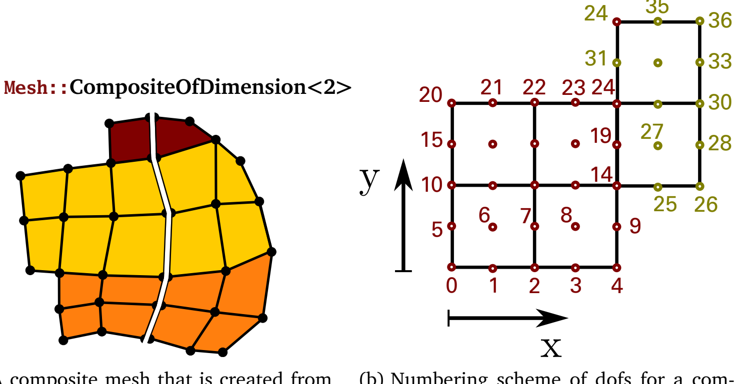
In parallel execution, this scheme is executed first on the non-ghost and then on the ghost nodes of the subdomains of all submeshes. Thus, the local numbering of the composite scheme visits the non-ghost nodes of all subdomains first before iterating over the ghost dofs on all subdomains. Thus, the ghost buffer is consecutive in memory as required by the parallel PETSc `Vec`s.

For the construction of this numbering, the shared nodes of different submeshes, which lie at the same position, have to be determined. The identification of shared nodes occurs according to their position in the physical domain. The distance in every coordinate direction has to be lower than the tolerance of 10^{-5} for a pair of nodes to be considered identical and shared. The shared nodes are determined on every local subdomain of the underlying structured meshes. To correctly number ghost nodes that are shared between submeshes, communication between processes is necessary.

Using the set of shared nodes, mappings in both directions between the local numberings of the submeshes and the local and global PETSc numberings of the composite mesh are constructed. These mappings are used to transfer operations on the composite mesh to operations on the structured submeshes. Thus, every implemented algorithm can transparently work also on composite meshes.

The creation of the numbering schemes requires that neighboring elements on different submeshes are located on the same process. If this was not the case, submeshes would potentially have ghost nodes at their outer border, which does not occur in normal structured meshes and would disallow reusing their implementation. Furthermore, the MPI communicator of the submeshes has to be the same and no subdomain can be empty. This means that a composite mesh has to be partitioned, such that every submesh is subdivided into the same number of partitions involving all processes. If these requirements are fulfilled, the parallel implementation of any algorithm on structured meshes can be reused for composite meshes. Figure 7.7a shows a valid partitioning of the exemplary composite mesh to two subdomains.

To configure composite meshes in the settings, their submeshes have to be specified as usual for structured meshes. Then, a list of all submeshes is given for the composite mesh. In parallel execution, a proper partitioning that fulfills the requirements has to be constructed in the Python script of the settings as well.



(a) A composite mesh that is created from three structured meshes (different colors) and a possible subdivision for parallel partitioning (white vertical line).

(b) Numbering scheme of dofs for a composite mesh, which is created from two quadratic meshes.

Figure 7.7: Examples for composite meshes that combine the advantages of structured and unstructured meshes.

An application of composite meshes is the biceps muscle with a fat and skin layer. Figure 7.8 visualizes the composite mesh. It consists of two structured submeshes for the muscle belly and the body layer on top, as visualized in the top image. The bottom image shows a partitioning to four processes. As can be seen, the domain can be split along the x and z coordinate axes to produce valid partitionings. Using this decomposition strategy, any number of subdomains (limited by the number of elements, though) is possible.

7.2 Finite Element Matrices and Boundary Conditions

Another important mathematical object besides the vector, which has to be represented in finite element simulation programs, is the matrix. Matrices are mainly needed to store the linear system of equations that results from the discretized weak formulation within the FEM. Dirichlet boundary conditions can be enforced by adjusting the system matrix.

In the following sections, the storage of matrices is discussed, an efficient, parallel algorithm to assemble the FEM system matrix is presented and evaluated and a second parallel algorithm for handling Dirichlet boundary conditions is given.

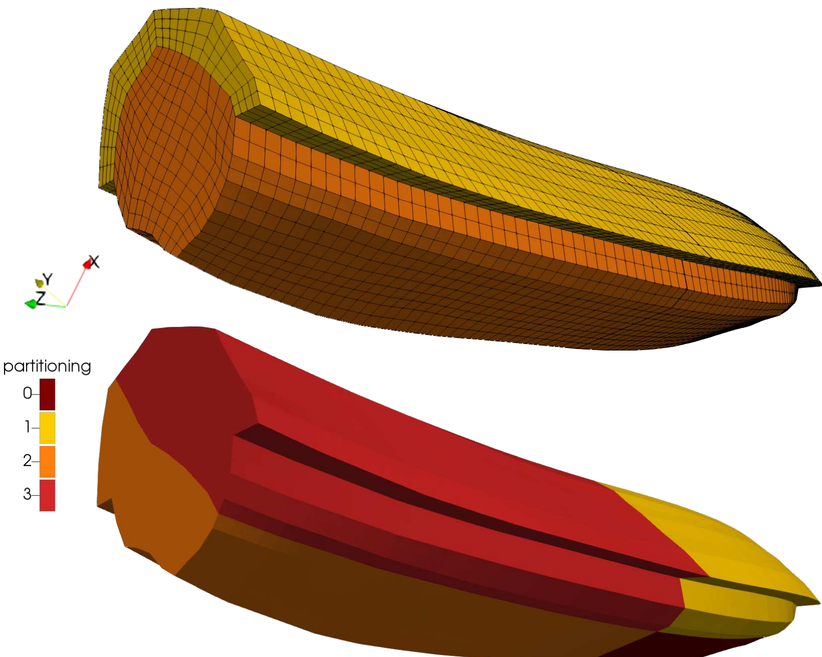


Figure 7.8: Composite mesh of the biceps muscle. Top: the two structured meshes from which the composite mesh is created, bottom: partitioning to four processes.

7.2.1 Storage of Matrices

The storage of matrices is delegated to PETSc, like the storage of vectors. The default sparse matrix format of PETSc, *compressed row storage (CRS)* or "AIJ" in PETSc notion, is used. The representation stores the non-zero locations and their values for every row of the matrix.

The system matrices in the FEM have as many rows and columns as there are global dofs in the system. The typical linear system of equations can be expressed as:

$$\mathbf{K}\,\mathbf{u} = \mathbf{f},$$

with system matrix $\mathbf{K}$ and the parallel vectors $\mathbf{u}$ and $\mathbf{f}$ of the solution and right-hand side, respectively. The partitioning of the rows of the matrix corresponds to the partitioning of

the right-hand side vector **f**. Thus, every rank has the complete information of a subset of lines in this matrix equation.

Every rank stores a submatrix of size $n_{\text{local_without_ghosts}} \times n_{\text{global}}$. In PETSc, this submatrix is composed of two blocks. The *diagonal* block is a square matrix of size $(n_{\text{local_without_ghosts}})^2$ and holds only the columns of the local dofs. The rest of the columns are stored in the *off-diagonal* block which is a non-square matrix in general.

The memory of these two storage blocks needs to be preallocated prior to the assignment of matrix entries. This allows PETSc to allocate the whole data storage in one chunk instead of potential reallocations for every new matrix entry. According to the documentation of PETSc, this can speed up the assembly runtime by a factor of 50 [Bal16]. For the preallocation, the numbers of non-zero entries per row in the two storage blocks need to be estimated. The estimated numbers need to be equal to or greater than the actual number of non-zeros per row.

The stiffness and mass matrices in the FEM have a banded non-zero structure that implies a maximum number of non-zero entries per matrix row. The value can be computed as follows:

$$\begin{aligned} n_{\text{1D_overlaps}} &= (2\,n_{\text{dofs_per_basis}} - 1) \cdot n_{\text{dofs_per_node}}, \\ n_{\text{non-zeros}} &= \left(n_{\text{1D_overlaps}}\right)^d. \end{aligned} \tag{7.1}$$

Here, the number $n_{\text{dofs_per_basis}}$ of dofs per 1D element is 2 and 3 for linear and quadratic Lagrange bases and 4 for cubic Hermite basis functions. The number $n_{\text{dofs_per_node}}$ of dofs per node is 1 for Lagrange basis functions and 2 for Hermite basis functions. The value $n_{\text{1D_overlaps}}$ describes the number of basis functions in a 1D mesh that have overlapping support with a given basis function. By the tensor product approach, the resulting estimate $n_{\text{non-zeros}}$ of non-zero entries per row is computed by exponentiation of $n_{\text{1D_overlaps}}$ with the dimensionality d.

Because no assumption can be made about how the bands of non-zero entries in the matrix are distributed to the diagonal and off-diagonal storage parts, the same value of $n_{\text{non-zeros}}$ is used as estimate to preallocate both the diagonal and the off-diagonal part of the local matrix storage.

In the following, the non-zero structure of an exemplary stiffness matrix is shown. A 3D regular fixed mesh of $4 \times 4 \times 4$ elements with quadratic Lagrange basis functions is considered. The Laplace equation is solved with Dirichlet boundary conditions at the bottom and top planes of the volume. The prescribed values are 1 at the bottom and 2 at

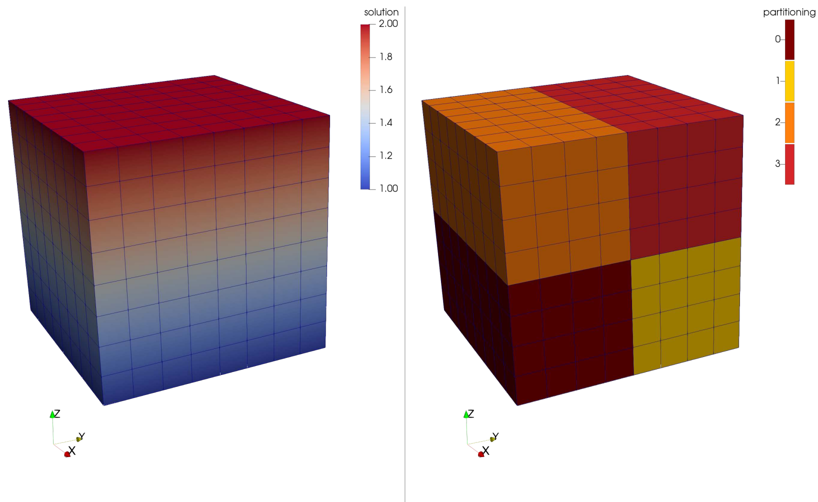


Figure 7.9: Solution of the Laplace equation $\Delta \mathbf{u} = \mathbf{f}$ with prescribed values $\mathbf{u}|_{\text{bottom}} = 1$ and $\mathbf{u}|_{\text{top}} = 2$. The mesh consists of $4 \times 4 \times 4$ quadratic elements and, thus, 9^3 nodes. Left: Solution, right: Partitioning to four processes.

the top. The solution is visualized in the left of Fig. 7.9. The computation is performed with four processes. Figure 7.9 shows the partitioning on the right.

The non-zero estimates computed by Eq. (7.1) are $n_{\text{1D_overlaps}} = 5$ and $n_{\text{non-zeros}} = 125$. The mesh has 4 elements, thus, 9 nodes per coordinate direction, and, therefore, $n_{\text{global}} = 9^3 = 729$ dofs. Figure 7.10 shows the resulting sparsity pattern of the stiffness matrix $\mathbf{K}$. The portions of the four processes are indicated by different colors. The maximum number of non-zeros per row and column is indeed 125, as calculated. Some rows have less non-zero entries. These correspond to dofs that lie on the boundary of the domain. The rows with only one non-zero entry on the diagonal enforce the Dirichlet boundary conditions. The total size of preallocated memory for the diagonal and off-diagonal blocks on all processes is $2\, n_{\text{global}}\, n_{\text{non-zeros}} = 182\,250$. The actual number of non-zero entries is 35 937, which is approximately 20 % of the preallocated values.

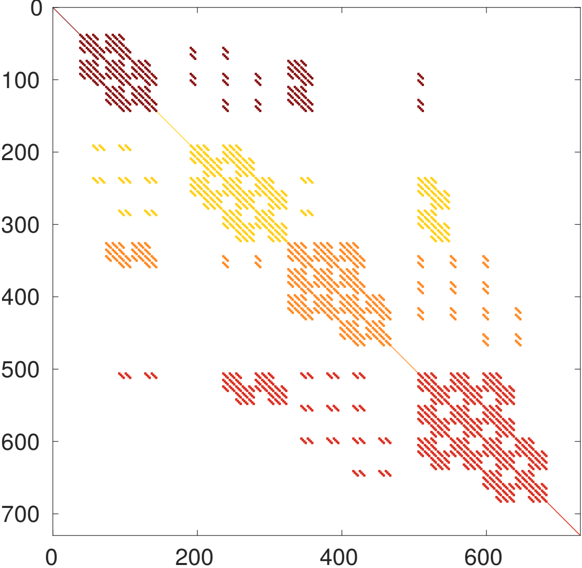


Figure 7.10: Sparsity pattern, i.e., locations of non-zero entries of the 729×729 stiffness matrix $\mathbf{K}$ for the example problem in Fig. 7.9. The rows for the four processes are given by different colors matching the partitioning in Fig. 7.9. The processes have 144, 180, 180 and 225 local dofs.

How To Reproduce

In any example, the system matrix can be written to a MATLAB compatible file by specifying the settings `'dumpFormat': 'matlab', 'dumpFilename': 'out'`. To get the non-zero structure for the example in Fig. 7.10, compile the *laplace3d* example and run the following:

```
cd $OPENDIHU_HOME/examples/laplace/laplace3d/build_release
mpirun -n 4 ./laplace_quadratic ../settings_quadratic_matrix_output.
  ↪ py -ksp_view
```

The flag `-ksp_view` is parsed by PETSc and outputs matrix statistics such as the number of preallocated and actual non-zeros. A file `out_matrix_000.m` is created that can be loaded in MATLAB. Use `spy(stiffnessMatrix)` to plot the non-zero structure.

7.2.2 Assembly of Finite Element Matrices

Next, the algorithm to compute stiffness and mass matrices in parallel for the application of the d-dimensional FEM is discussed. The matrix entries to be computed are given by

$$m_{i,j} = \int_{\Omega} I(\mathbf{x})\,\mathrm{d}\mathbf{x}, \qquad (7.2)$$

where the integrand I is derived from the respective FEM formulation in weak form.

A generic algorithm for the evaluation of this integral and parallel assembly to a global matrix is presented in Alg. 4. Multiple variants of this algorithm, which only differ in their achieved performance, have been implemented for evaluation purposes. They are discussed in Sec. 7.2.3. The listed algorithm in Alg. 4 shows the fastest variant.

Algorithm 4 Finite element matrix assembly

```
 1 procedure Assemble FE system matrix
 2   for elements e = {e1, e2, e3, e4} in all elements do
 3     for sampling point ξ do
 4       Compute Jacobian J_e(ξ)
 5       Evaluate integrand I_{e,i,j}(ξ) = c · I(J_e, ξ)   ⇝for all elements e/dofs (i, j) at once
 6     matrix_entries[i,j] = Quadrature(I_{e,i,j}(ξ))     ⇝for all el. e/dofs (i, j) at once
 7     for dof i = 0,...,n_dofs_per_element − 1 do
 8       for dof j = 0,...,n_dofs_per_element − 1 do
 9         rows    = dofs i of elements e1, e2, e3, e4
10         columns = dofs j of elements e1, e2, e3, e4
11         matrix[rows,columns] = matrix_entries[i,j]
12   Call PETSc final matrix assembly
```

The main loop in line 4.2 iterates over the local elements of the subdomain. The shown implementation iterates over sets of four elements e_1, e_2, e_3 and e_4. A simpler variation of the algorithm is to instead visit every single local element in its own iteration. However, the more efficient variant is the presented one that always considers the set e of four elements at once. Explicit vectorization is employed on all following operations on these four elements, such that the four sequences of calculations for the elements are performed by identical instructions. This adheres to the single-instruction-multiple-data (SIMD) paradigm. The vectorization is explicit since the C++ library *Vc* [Kre12; Kre15] is used. Vc provides zero-overhead C++ types for explicitly data-parallel programming and directly employs the respective vector instructions where these types are used.

To compute the integral in Eq. (7.2), a node based quadrature rule is used. In our code, the quadrature rule has to be chosen at compile time among Gauss, Newton-Cotes

and Clenshaw-Curtis quadrature rules. All three schemes are implemented for different numbers $n_{\text{sampling_points}}$ of sampling points. The loop in lines 4.3—4.5 iterates over the respective sampling points $\xi \in [0,1]^d$ in the element coordinate system. In line 4.4, the Jacobian matrix of the mapping from element to world coordinate frame is computed at the given coordinate ξ for all elements in the set e. The Jacobian is needed in the integrand for the transformation of the integration domain.

In line 4.5, the integrand I is evaluated for all elements in e and also for all pairs (i, j) of local dofs in each of these elements. The indices i and j are in the range $i, j \in \{0, 1, \dots, n_{\text{dofs_per_element}} - 1\}$ with the number $n_{\text{dofs_per_element}}$ of dofs per element. The set of $4\,(n_{\text{dofs_per_element}})^2 \cdot (n_{\text{sampling_points}})^d$ computed values is passed to the implementation of the d-dimensional quadrature rule in line 4.6. The numerical values of the integrals get computed for all considered elements in e and dof pairs (i, j), yielding $4\,(n_{\text{dofs_per_element}})^2$ quadrature problems to be solved at once. This means that the result of the quadrature rule is a linear combination of quadrature weights and vector-valued function evaluations instead of scalar function values.

Next, the two loops in lines 4.7—4.11 assign the computed values stored in the variable `matrix_entries` to the actual matrix. The loops iterate over all dof pairs (i, j) per element. The corresponding rows and columns are determined in lines 4.9 and 4.10 and the respective computed value is assigned in line 4.11. The values are added to the matrix entry indicated by the row and column index. Since all dofs including ghosts are considered on every local domain, the same matrix entry can get contributions on multiple processes.

Thus, the last step in line 4.12 is a PETSc call that communicates and sums all matrix entry contributions to the respective processes where the dof is non-ghost. Additionally, the call frees the residual preallocated memory that was not needed for non-zero entries and finalizes the internal data structure of the CRS storage format.

In the last iteration over local elements of the main loop in line 4.2, the remaining number of elements is potentially less than four. Nevertheless, the computations proceed as normal. The spare entries of the SIMD vectors get computed using dummy values and are discarded at the end.

For the case of vector-valued finite element problems, e.g., linear elasticity with a solution vector of vector-valued displacements, two more inner loops over the components of the vector are inserted. As a result, the presented algorithm can be used to assemble any FEM matrix on any mesh type and for any formulation given by the term I in Eq. (7.2). Examples are stiffness and mass matrices for the Laplace operator with and without

diffusion tensor or stiffness and mass matrices for the linear equations that have to be solved during the solution of nonlinear, dynamic elasticity problems.

Note that the algorithm operates in parallel execution entirely on data stored in the local subdomain and does not need any global information. The loop iterates over local elements. For every element, the indices in the local numbering of the nodes that are adjacent to the element are needed. In structured meshes, this information is determined from the numbers $N_x^{\text{el}} \times N_y^{\text{el}} \times N_z^{\text{el}}$ of local elements in the coordinate directions. In unstructured meshes, these indices are explicitly stored in the elements. To assemble the global matrix, PETSc uses the mapping from local to global numbering, which it can maintain by storing the constant offset in the global numbering on every subdomain. Mappings from global dof or node numbers to local numbers are not needed in this algorithm. In general, storing global information, which would require memory of $\mathcal{O}(n_{\text{global}})$, is avoided in all algorithms to ensure good parallel weak scaling properties.

7.2.3 Performance of the Algorithm for Parallel Matrix Assembly

In the following, the performance of two variations of the algorithm in Alg. 4 will be examined. The first variation is to not use explicit vectorization and, thus, iterate over the elements one by one instead of the groups of four elements in line 4.2.

The second variation is to not accumulate multiple values for the application of the quadrature scheme in line 4.6. Instead, the loop over the sampling points in line 4.3 is made the inner-most loop and placed inside the loop in line 4.8. Then, the quadrature scheme only computes a single value at once. In consequence, this value can directly be stored in the resulting matrix, and the temporary variable `matrix_entries` is not needed. This loop reordering requires the evaluations of the Jacobian and the integrand in lines 4.4 and 4.5 to also be located in the new inner-most loop over the sampling points.

The algorithm with these two variations corresponds to the naive way of implementing matrix assembly because iterating first over elements, then over dof pairs and then performing the quadrature directly mirrors the mathematical description.

Different combinations of these two variations result in four variants of the algorithm. A study was conducted to measure their effects on the runtimes. A simulation with the same settings as in Fig. 7.9 was run except for a larger number of $50 \times 50 \times 50$ elements. This setup lead to a total number of $n_{\text{global}} = 1\,030\,301$ dofs. Gauss quadrature with three sampling points per coordinate direction and, thus, $3^3 = 27$ sampling points in

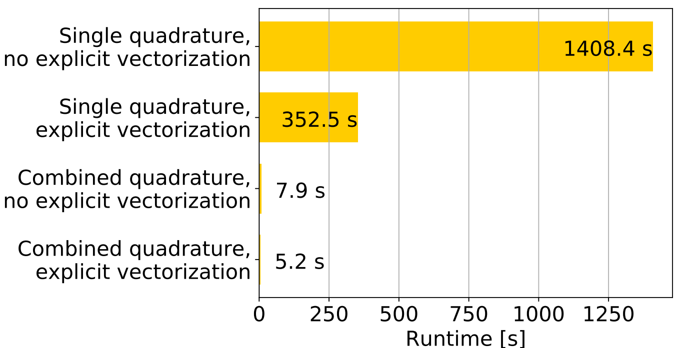


Figure 7.11: Runtimes of different optimizations for the algorithm to assemble the FEM stiffness matrix.

total was used. The program was executed with four processes on an AMD EPYC 7742 processor with base frequency of 2.25 GHz, maximum boost frequency of 3.4 GHz, 2 TB of memory and a memory bandwidth of 204.8 $\frac{\text{GB}}{\text{s}}$ per socket. The runtime for the assembly of the stiffness matrix with dimensions $n_{\text{global}} \times n_{\text{global}}$ was measured for all four variants. Figure 7.11 presents the resulting runtimes.

It can be seen that a large difference in runtime exists between the variants with quadrature of single values compared to the combined quadrature. In the case of no explicit vectorization (first and third bar from the top in Fig. 7.11), the runtime reduces to less than 0.6 %. In the case of explicit vectorization (second and fourth bar from the top in Fig. 7.11), the runtime reduces to less than 1.5 %. The reason for this enormous gain in performance is three-fold. First, the values of the Jacobian can be reused for the same element and sampling point. Second, the combined quadrature for multiple values yields more cache-efficient memory access, because the vector of values is stored consecutively in memory and can be fetched from the cache by less load operations. For the single quadrature, the individual values are fetched at different times from different memory locations. Third, the compiler is able to employ SIMD instructions for the combined quadrature, a process called auto-vectorization.

The performance improvements from the second variation, the use of explicit vectorization by simultaneously computing the entries for four elements at once can be seen by comparing the first and second bars and the third and forth bars in Fig. 7.11. The run-

time reduction of explicit vectorization with single quadrature from 1408.4 s to 352.5 s is exactly by the expected factor of four. This shows that explicit vectorization works as expected, and that no auto-vectorization could be performed by the compiler for the single quadrature. The runtime reduction of explicit vectorization from 7.9 s to 5.2 s during combined quadrature corresponds to a speedup of only approximately 1.5. This shows that combined quadrature without explicit vectorization already allows the compiler to employ some auto-vectorization. However, using the explicit vectorization approach on the level of different elements instead of the level of quadrature values still has a positive effect.

In total, the performance gain from the most naive implementation (top bar in Fig. 7.11) to the most optimized version (bottom bar in Fig. 7.11) equals a speedup of more than 270. Together with the solution of the linear system using an algebraic multigrid preconditioner and a GMRES solver with a residual norm tolerance of 10^{-10}, the total runtime of the program to solve the Laplace problem with over a million degrees of freedom using a modest parallelism of four processes takes 28 s.

How To Reproduce

The results of Fig. 7.11 can be reproduced as follows. The explicit vectorization can be turned on and off with the variable `USE_VECTORIZED_FE_MATRIX_ASSEMBLY` in the configuration of the SCons build system in the file `$OPENDIHU_HOME/user-variables.scons.py` (ca. line 75). Normally, only the variant with combined quadrature is implemented. To test the single quadrature, checkout the git branch `fem_assembly_measurement`. The single quadrature is on by default, to change back to the combined quadrature, edit the following line:

```
vi $OPENDIHU_HOME/core/src/spatial_discretization/
  ↪ finite_element_method/01_stiffness_matrix_integrate.tpp +17
```

For all variants of the algorithm, compile and run the following example:

```
cd $OPENDIHU_HOME/examples/laplace/laplace3d/build_release
mpirun -n 4 ./laplace_quadratic ../settings_quadratic.py
```

The duration of the algorithm for stiffness matrix assembly will be printed.

7.2.4 Assembly of Finite Element Matrices for Regular Meshes

For equidistant meshes of type `Mesh::StructuredRegularFixedOfDimension<D>`, all elements are similar through the uniform grid resolution and, thus, all elements matrices equal the same constant matrix. In consequence, the integral terms in Eq. (7.2) can be precomputed analytically and no numerical quadrature at runtime is needed. This speeds up the determination of the FEM matrices.

We implement matrix assembly using precomputed values for the stiffness and mass matrices of the Laplace operator for linear Lagrange basis functions. For the stiffness matrix of the Laplace operator, the integral term $(-\int_{\Omega^{\text{el}}} \nabla\phi_i \cdot \nabla\phi_j \,\mathrm{d}\xi)$ is calculated analytically. The result is a value for every combination of the dofs i and j in the element. Thus, the contribution of one representative element in the mesh to the values at adjacent dofs is known. To get the matrix entry for a particular dof, the element contributions of all elements that are adjacent to the node need to be summed up. For this process, it is convenient to represent the precomputed values in *stencil notation*.

Table 7.2 shows the stencils for element contributions in the left column and the resulting stencils for the dofs in the right column. In the element contribution stencils, dof i is chosen as the first dof in the local dof numbering. Values are calculated for all choices of dof j in the element and the values are noted in the stencil. The location of dof i is marked by the underlined number. Stencils for all other locations of dof i follow by symmetry.

The node stencils describe the values of the term $(-\int_{\Omega} \nabla\phi_i \cdot \nabla\phi_j \,\mathrm{d}\xi)$ with the integration over the whole domain. The node i is fixed and marked in the stencil notation by the underlined number. Values for all neighboring nodes j are computed and listed in the stencils. For a given node i, the integral over the whole domain Ω is the sum of integrals over all elements Ω^{el} adjacent to node i. These have been computed in the element contribution stencils. As can be seen in Tab. 7.2, the node stencils follow by adding up mirrored variants of the element stencils centered around the underlined node.

The entries in the stiffness matrix are computed from the node stencils by a multiplication with a mesh dependent prefactor. For 1D, 2D and 3D meshes with mesh width h, these prefactors are $1/h$, 1 and h, respectively. Thus, e.g., the 1D stiffness matrix has the entries $-2/h$ on the diagonal, $1/h$ on the secondary diagonals above and below the main diagonal and zero everywhere else.

A similar computation is possible for the mass matrix, where the term $\int \phi_i\, \phi_j \,\mathrm{d}\xi$ can be precalculated. The element and node stencils for the mass matrix are given in Tab. 7.3.

Dim.	Element contribution	Node stencil
1D	$\begin{bmatrix} \underline{-1} & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & \underline{-2} & 1 \end{bmatrix}$
2D	$\frac{1}{6}\begin{bmatrix} 1 & 2 \\ \underline{-4} & 1 \end{bmatrix}$	$\frac{1}{3}\begin{bmatrix} 1 & 1 & 1 \\ 1 & \underline{-8} & 1 \\ 1 & 1 & 1 \end{bmatrix}$
3D	center: $\frac{1}{12}\begin{bmatrix} 0 & 1 \\ \underline{-4} & 0 \end{bmatrix}$ bottom: $\frac{1}{12}\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$	top: $\frac{1}{12}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 0 & 2 \\ 1 & 2 & 1 \end{bmatrix}$ center: $\frac{1}{12}\begin{bmatrix} 2 & 0 & 2 \\ 0 & \underline{-32} & 0 \\ 2 & 0 & 2 \end{bmatrix}$ bottom: same as top

Table 7.2: Stencils of the finite element stiffness matrix of Δu for a regular mesh with mesh width $h = 1$ and linear ansatz functions. The stiffness matrix entries can be computed by multiplication with a mesh width dependent factor.

Dim.	Element contribution	Node stencil
1D	$\frac{1}{6}\begin{bmatrix} \underline{2} & 1 \end{bmatrix}$	$\frac{1}{6}\begin{bmatrix} 1 & \underline{4} & 1 \end{bmatrix}$
2D	$\frac{1}{36}\begin{bmatrix} 2 & 1 \\ \underline{4} & 2 \end{bmatrix}$	$\frac{1}{36}\begin{bmatrix} 1 & 4 & 1 \\ 4 & \underline{16} & 4 \\ 1 & 4 & 1 \end{bmatrix}$
3D	center: $\frac{1}{216}\begin{bmatrix} 4 & 2 \\ \underline{8} & 4 \end{bmatrix}$ bottom: $\frac{1}{216}\begin{bmatrix} 2 & 1 \\ 4 & 2 \end{bmatrix}$	top: $\frac{1}{216}\begin{bmatrix} 1 & 4 & 1 \\ 4 & 16 & 4 \\ 1 & 4 & 1 \end{bmatrix}$ center: $\frac{1}{216}\begin{bmatrix} 4 & 16 & 4 \\ 16 & \underline{64} & 16 \\ 4 & 16 & 4 \end{bmatrix}$ bottom: same as top

Table 7.3: Stencils of the finite element mass matrix for a regular mesh with mesh width $h = 1$ and linear ansatz functions. The mass matrix entries can be computed by multiplication with a mesh width dependent factor.

The precalculated values can only be used for meshes with uniform mesh width and linear Lagrange basis functions. In OpenDiHu, the type of the mesh and basis function is fixed at compile time. The stencil based approach to set the entries of stiffness and mass matrix is implemented as partial template specialization of the template, which otherwise uses the numerical algorithm presented in Sec. 7.2.2. Thus, the stencil based implementation is instantiated automatically by the compiler for regular fixed meshes of all dimensionalities with linear basis functions.

The conditions for the stencil based approach are fulfilled whenever regular fixed meshes and linear bases are used, e.g., for toy problems or studies where the shape of the domain is irrelevant and, e.g., a cuboid cutout of muscle tissue is sufficient. Mathematical models involving a Laplace operator, such as Laplace, Poisson or diffusion problems can benefit from the faster system matrix setup.

Another purpose of the stencil based approach in OpenDiHu besides runtime reduction is to verify the implementation of the numerical integration method of Alg. 4. Because of the regular mesh and linear ansatz functions, the numerical method computes the exact result with proper quadrature schemes and, thus, can be compared to the stencil based approach. Several unit tests ensure that the generated system matrices of the two approaches are equal.

7.2.5 Algorithm for Dirichlet Boundary Conditions

The assembled system matrix needs to be adjusted when Dirichlet boundary conditions are specified. Dirichlet boundary conditions are ensured by replacing equations involving the prescribed dofs by the definition of the boundary conditions. This involves changes in the system matrix and right-hand side of the finite element formulation.

Consider the following matrix equation resulting from a FE discretization with four dofs u_1 to u_4:

$$\begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{pmatrix}.$$

We assume a Dirichlet boundary condition for the last dof, $u_4 = \hat{u}_4$. Enforcing this condition is accomplished by the following adjusted system of equations:

$$\begin{pmatrix} m_{11} & m_{12} & m_{13} & 0 \\ m_{21} & m_{22} & m_{23} & 0 \\ m_{31} & m_{32} & m_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} f_1 - m_{14}\,\hat{u}_4 \\ f_2 - m_{24}\,\hat{u}_4 \\ f_3 - m_{34}\,\hat{u}_4 \\ \hat{u}_4 \end{pmatrix}.$$

The last equation has been replaced by $u_4 = \hat{u}_4$, all summands in the other equations where u_4 occurred have been brought to the right-hand side and u_4 has been substituted by the prescribed value $\hat{u}_4$.

Thus, setting a dof u_i to a prescribed value $\hat{u}$ corresponds to subtracting the column vector of column i of the system matrix multiplied with $\hat{u}$ from the right-hand side, replacing the right-hand side entry at i by $\hat{u}$ and setting row i and column i in the matrix to all zero and the diagonal entry m_{ii} to one.

This method also works for more prescribed values as demonstrated with the additional Dirichlet boundary condition $u_2 = \hat{u}_2$. Executing the scheme results in the following system:

$$\begin{pmatrix} m_{11} & 0 & m_{13} & 0 \\ 0 & 1 & 0 & 0 \\ m_{31} & 0 & m_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} f_1 - m_{12}\,\hat{u}_2 - m_{14}\,\hat{u}_4 \\ \hat{u}_2 \\ f_3 - m_{32}\,\hat{u}_2 - m_{34}\,\hat{u}_4 \\ \hat{u}_4 \end{pmatrix}.$$

During parallel execution, only a distinct subset of rows of the matrix equation is accessible on every rank. However, for the subtractions at the right-hand side, the full vector of prescribed boundary conditions values is needed on every rank. Additionally, the corresponding matrix entries are required. While the needed matrix entries are all stored on the local rank, the vector of prescribed values is partitioned to all ranks and only the local subdomain is accessible. Some of the non-accessible prescribed values correspond to a non-zero matrix entry, though, and are not needed for the subtraction. In consequence, some data transfer between processes is required. In the following, the identification of the values that have to be communicated is illustrated with an example.

Figure 7.12 (a) shows a 2D quadratic mesh with 3×2 elements. The top layer of nodes has prescribed Dirichlet boundary conditions, marked by the red circles. The elements

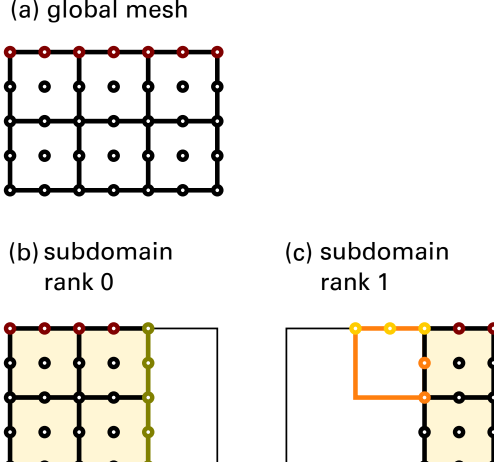


Figure 7.12: Example to illustrate ghost element transfer that is needed for handling Dirichlet boundary conditions in parallel. A mesh with Dirichlet boundary conditions on the red nodes is given in (a). The subdomains for two processes are given in (b) and (c). (c) shows a ghost element in orange that is sent from rank 0 to rank 1.

are partitioned to two processes with subdomains containing four and two elements, as shown in Fig. 7.12 (b) and Fig. 7.12 (c).

On rank 0, the right-most layer of nodes consists of ghost nodes. All other nodes are non-ghost and correspond to the matrix rows and right-hand side entries that have to be manipulated by this rank. In every of these matrix rows i, only entries in columns j that correspond to nodes in the same element as i are non-zero. For these columns j, the prescribed Dirichlet boundary condition values $\hat{u}_j$ need to be known, such that the product of matrix entry m_{ij} and prescribed value $\hat{u}_j$ can be subtracted from the right-hand side at the corresponding row i. This is fulfilled for rank 0 since the required boundary condition values are all part of the subdomain. The top right boundary condition node in the top right element of rank 0's subdomain is stored as ghost value, the other four are non-ghosts.

As can be seen in Fig. 7.12 (c), the subdomain of rank 1 has no ghost nodes. The rank owns three boundary condition nodes in the top layer. It has to perform right-hand side subtractions for the twelve rows of the matrix equation that correspond to the 3×4 other nodes, which have no prescribed boundary condition. For the bottom two

horizontal layers of nodes, the subtraction terms are zero, because the bottom element of the subdomain has no boundary conditions and, thus, these matrix entries are all zero. The upper three horizontal layers of nodes that all belong to the upper element, however, lead to non-zero right-hand side subtraction terms because of the boundary conditions at the top. The terms can be computed for all but the two orange nodes. At the corresponding rows i, the prescribed values $\hat{u}_j$ for all five yellow and red boundary condition nodes j are needed. However, the left two yellow nodes are not stored on the subdomain of rank 1. They have to be communicated from the subdomain of rank 0. As rank 1 has no topology knowledge of rank 0's subdomain, the information that the missing boundary condition nodes are in the same element as the two orange nodes has to be also transferred.

In total, the information indicated in Fig. 7.12 (c) by the orange element with the two orange nodes and the three yellow boundary condition nodes and values have to be communicated from rank 0 to rank 1. This element is called *ghost element*. Rank 0 knows that rank 1 will need this information because it stores the right-most yellow node and the orange nodes in subdomain 1 as ghost nodes in its own subdomain. Therefore, no request from rank 1 is necessary.

In general, every rank constructs ghost elements from own elements that contain both at least one boundary condition node and at least one ghost node without boundary condition. The global indices of all nodes of these two kinds and the corresponding boundary condition values are packaged as ghost element and sent to the rank of the neighboring subdomain. Every process potentially sends multiple ghost elements to multiple neighboring ranks.

Because a rank does not know the number of ghost elements it will receive a-priori, one-sided communication is employed, which was introduced with the MPI 2.0 standard. More specifically, *passive target* communication is used where only the sending rank is explicitly involved in the transfer.

After the data are received, the proper matrix entries can be retrieved from the local matrix storage and the subtraction operations on the right-hand side of the formulation can be performed. The algorithm has to ensure that the same subtraction is not executed multiple times, when the particular pair of nodes is obtained once from the local subdomain and once from a received ghost element. After solving the linear system with the updated stiffness matrix and right-hand side, the dofs on nodes with Dirichlet boundary conditions will have the prescribed values.

Considering the overhead for ensuring Dirichlet boundary conditions, the question may arise whether the partitioning scheme should be designed in a better way to simplify the presented algorithm. However, applying Dirichlet boundary conditions is the only process where the subdomains including ghost nodes, which were created by the parallel partitioning of the mesh, do not provide all required local information. All other algorithms such as matrix assembly successfully operate on the given partitioning. Therefore, designing the ghost information of subdomains differently, e.g., by storing a full ghost layer of elements or nodes around the local subdomains seems not beneficial. In fact, our presented approach is minimal with respect to stored local mesh information. Furthermore, the communicated information for the Dirichlet boundary conditions only involves a few elements depending on the number of boundary conditions. Of these elements, only a subset of nodes is actually communicated.

Another alternative approach would be to store all Dirichlet boundary condition information globally on all processes, such as done in OpenCMISS Iron. This approach is not chosen, because the required total storage would increase linearly with the number of processes. Thus, the possible number of boundary conditions would be limited by available memory.

In summary, the presented algorithm fits our design goals of good performance. It is used in our implementation to enforce Dirichlet boundary conditions for static and dynamic problems. The algorithm is executed after assembling the stiffness matrix. In consequence, for static problems the linear system solver sets the prescribed values at the respective dofs and the boundary conditions are fulfilled. For dynamic problems with constant stiffness matrices, Dirichlet boundary conditions have to be ensured in every timestep. After running the algorithm on the system matrix once, the operation on the right-hand side vector needs to be repeated in every timestep on the updated right-hand side.

7.2.6 Neumann Boundary Conditions

The other supported boundary conditions besides Dirichlet boundary conditions are the Neumann type boundary conditions. In general, they are formulated on a subset $\Gamma_f \subset \partial\Omega$ of the boundary $\partial\Omega$ with outwards normal vector $\mathbf{n}$ as follows:

$$\big(\boldsymbol{\sigma}\,\mathrm{grad}\,u(\mathbf{x})\big)\cdot\mathbf{n} = f(\mathbf{x}) \quad \text{for } \mathbf{x}\in\Gamma_f. \tag{7.3}$$

Here, $\boldsymbol{\sigma}$ is a conductivity tensor, which describes the anisotropy of the problem. For problems with a scalar solution function $u : \Omega \to \mathbb{R}$, the Neumann boundary condition is interpreted as a flux f over the boundary of the quantity described by u. For elasticity problems, the solution function $\mathbf{u} : \Omega \to \mathbb{R}^d$ is vector-valued with $d = 2$ or $d = 3$ and describes the displacement field. Then, the value $\mathbf{f}$ corresponds to a traction force $\mathbf{f}$ per area that acts on the surface Γ_f.

In a finite element formulation, we use Neumann boundary conditions to resolve the boundary integrals that appear after applying the divergence theorem on the weak form. In the derivation in Sec. 5.3.2, these boundary integrals were summarized by the matrix $\mathbf{B}_{\boldsymbol{\sigma}}$. By using the definition of the Neumann boundary condition in Eq. (7.3), we can derive the following equation for the boundary integral:

$$-\sum_{j=1}^{M} u_j \int_{\Gamma_f} (\boldsymbol{\sigma}\, \mathrm{grad}\, \varphi_j \cdot \mathbf{n})\, \varphi_k \, \mathrm{d}\mathbf{x} = -\sum_{j=1}^{M} \int_{\Gamma_f} f_j\, \psi_j\, \varphi_k \, \mathrm{d}\mathbf{x} \tag{7.4a}$$

$$=: \mathrm{rhs}_k, \tag{7.4b}$$

where the dofs u_j and the ansatz functions φ_j for $j = 1, \dots, M$ discretize the solution function $u(\mathbf{x})$, the dofs f_j and the ansatz functions ψ_j discretize the flux $f(\mathbf{x})$ and φ_k is the test function, which is chosen from the same function space as the ansatz functions.

Equation (7.4a) is equivalent to the matrix equation Eq. (5.52), and Eq. (7.4b) defines the final right-hand side vector **rhs** of the linear system of equations to be solved. Analogously, the discretization of the Laplace problem in Eq. (5.53) contains a right-hand side of $\mathbf{rhs} = -\mathbf{B}_{\Gamma_f}\, \mathbf{f}$ with the vector $\mathbf{f}$ of dofs of the discretized flux function f.

For elasticity problems where the solution $\mathbf{u}(\mathbf{x})$ and the traction $\mathbf{f}(\mathbf{x})$ are vector-valued, the definition of the right-hand side, which is equivalent to Eq. (7.4b), can be formulated as

$$\mathrm{rhs}_{aM} := -\int_{\partial\Omega} T_a^L\, \psi^L(\mathbf{x})\, \delta u_M \, \mathrm{d}\mathbf{x}. \tag{7.5}$$

Here, the right-hand side vector **rhs** consists of the given coefficients rhs_{aM}, where $a = 1, \dots, d$ is the index over the dimension and $M = 1, \dots, N$ iterates over the dofs in the discrete function space. T_a^L is the dof of the discretized traction force using ansatz functions ψ^L and summation over the repeated index L. δu_M is the virtual displacement, which is equivalent to the test function in the Galerkin finite element formulation.

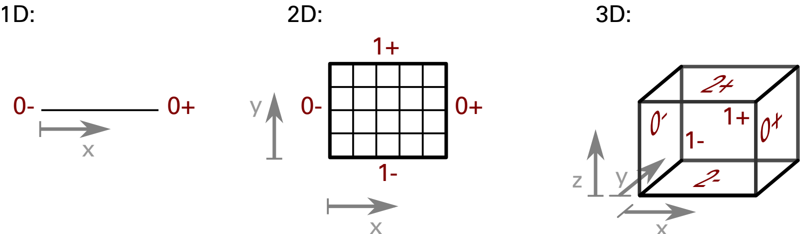


Figure 7.13: Notion of the faces of 1D, 2D and 3D elements, as used in the definition of Neumann boundary conditions.

In OpenDiHu, a class exists that parses Neumann boundary conditions from the Python settings and computes the negative right-hand side vector $-\mathbf{rhs}$, either for the scalar case in Eq. (7.4b) or the vector-valued case in Eq. (7.5).

If no Neumann boundary conditions are specified for parts of the boundary $\partial\Omega$, the right-hand side vector is set to zero for the corresponding dofs. This means that specifying no Neumann boundary conditions is equivalent to specifying homogeneous Neumann boundary conditions, i.e., setting $f \equiv 0$.

Neumann boundary conditions are specified in the Python settings as a list of elements with associated flux or traction values. This is in contrast to the Dirichlet boundary conditions, which are defined per dof or node. In every element with Neumann boundary conditions, the boundary face that is part of Γ_f has to be specified. The face is identified by one of the strings `"0-"`, `"0+"`, `"1-"`, `"1+"`, `"2-"` or `"2+"`, which describe the positive or negative coordinate directions of the element coordinate system, as given in Fig. 7.13.

For elasticity problems, where the function $f(\mathbf{x})$ is interpreted as traction force, two more options can be set. The first option is `"divideNeumannBoundaryConditionValuesByTotalArea"`. If set to `True`, the traction force vector is interpreted as a total force on the whole surface. The value of T in Eq. (7.5) is computed by scaling down the given value by the total surface area. This allows to conveniently specify a total force, which, e.g., acts on the lower end of the muscle. Without this option, the traction force is interpreted as force per area unit.

The second option `"isInReferenceConfiguration"` allows switching between reference and current configuration to specify the traction force. The mapping between the traction $\mathbf{T}$ in reference configuration and the traction $\mathbf{t}$ in current configuration is given

by the inverse deformation gradient **F**:

$$\mathbf{T} = \mathbf{F}^{-1}\,\mathbf{t}. \tag{7.6}$$

Because the implemented model uses the Lagrangian formulation with the right-hand side term defined in Eq. (7.5), the transformation in Eq. (7.6) and subsequently the right-hand side have to be computed in every timestep of a dynamic problem.

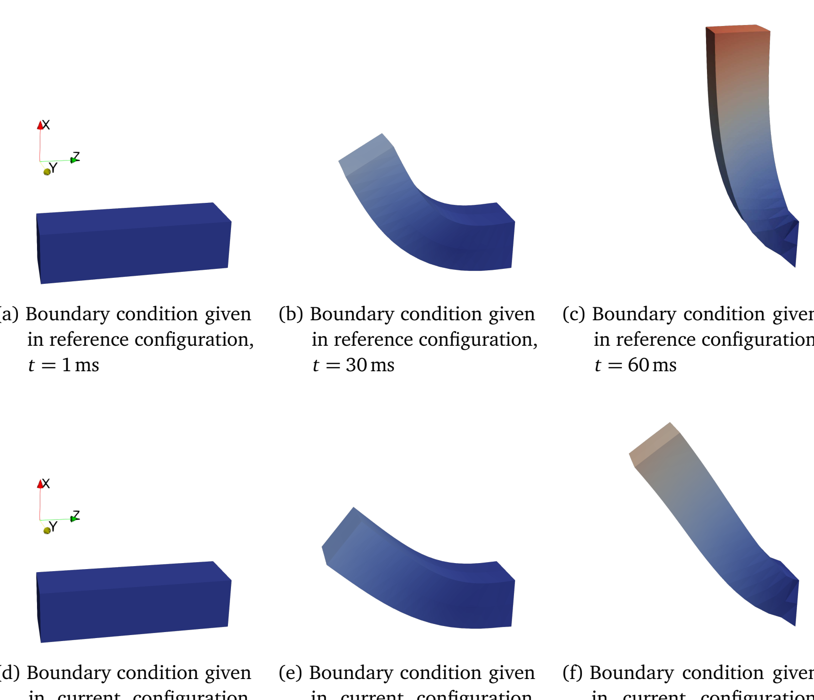
Figure 7.14 illustrates the difference between Neumann boundary conditions that are specified in the reference configuration and the current configuration. A horizontal, cuboid rod is fixed at its right end and a traction force in positive x direction acts on the surface on its other end. A dynamic hyperelasticity model with isotropic Mooney-Rivlin material is solved.

Figures 7.14a to 7.14c show the simulation results, where the traction force is specified in reference configuration, Figures 7.14d to 7.14f show results of the same simulation, but with the traction force specified in the current configuration. It can be seen that the rod bends more to the right, if the traction force is specified in reference configuration. In this case, the force is always acting perpendicular to the rod, whereas, in the other version, the direction of the applied force in the global coordinate system stays constant.

How To Reproduce

Use the following commands to create the results in Fig. 7.14.

```
cd $OPENDIHU_HOME/examples/electrophysiology/fibers/
  ↪ fibers_contraction/with_tendons_precice/meshes
./create_cuboid_meshes.sh     # create the cuboid mesh
cd $OPENDIHU_HOME/examples/electrophysiology/fibers/
  ↪ fibers_contraction/with_tendons_precice/
  ↪ traction_current_or_reference_configuration
mkorn && srr       # build
./muscle_precice settings_current_configuration.py ramp.py
./muscle_precice settings_reference_configuration.py ramp.py
```



(a) Boundary condition given in reference configuration, $t = 1\,\text{ms}$

(b) Boundary condition given in reference configuration, $t = 30\,\text{ms}$

(c) Boundary condition given in reference configuration, $t = 60\,\text{ms}$

(d) Boundary condition given in current configuration, $t = 1\,\text{ms}$

(e) Boundary condition given in current configuration, $t = 30\,\text{ms}$

(f) Boundary condition given in current configuration, $t = 60\,\text{ms}$

Figure 7.14: Influence of whether external traction forces are defined in the current or in the reference configuration. Simulation of a dynamic solid mechanics problem with a Mooney Rivlin material model and a constant traction force in x direction.

7.3 Parallel Partitioning and Subsampling of Meshes

The derivation of increasingly detailed models in the domain of biomechanics has to be complemented by engineering of efficient software that is used to solve these models. Using proper parallelization allows to increase the amount of computational load that is possible to handle. In turn, this allows to simulate more complex models with higher resolution and ultimately enables physiological and pathological insights on a new level.

For detailed multi-scale model solvers, parallelization is a complex task. The paradigm has to be regarded during the whole setup process of the system. Different descriptions for the same physical behavior have to be evaluated with respect to their solvability in parallel. For a given model, suitably parallelizable numerical solution schemes have to be selected. The implementation of individual solvers and their coupling have to take into account the parallel environment. Discretization schemes enabling parallel domain decomposition are required. Their representation on compute hardware with distributed memory has to be taken into account as well as ensuring acceptable conditioning of large scale problems. To ensure fast runtimes, load balancing between compute nodes and parallel scalability are important.

All these fundamental considerations potentially depend on each other and require a comprehensive solution. Thus, it is often difficult to port existing, isolated solver software that was designed for serial or moderately parallel execution to efficiently fit into a highly-parallel, multi-scale solution framework. To not (re-)create this kind of isolated solvers for individual model components, we focus on their parallel design from the ground up in the current and following sections.

In this section, we introduce algorithms for the generation of parallel partitioned meshes, which are fundamental ingredients to all our solvers. The parallel organization of the data and their indexing using various numbering schemes has already been discussed in Sec. 7.1.1 and Sec. 7.1.2, respectively. In the following, we consider the parallel partitioning problem on a higher level and provide algorithms to construct the domain decomposition for various meshes in the multi-scale model discretization. Meshes with different mesh widths are obtained by subsampling a finely resolved initial mesh, which is the outcome of the algorithms described in Chap. 3, and, in practice, is given to a particular simulation program by the respective mesh input file.

Sections 7.3.1 and 7.3.2 set the scene and define our requirements for well-behaved parallelized meshes. Sections 7.3.3 and 7.3.4 give details on the implemented algo-

rithms and Sec. 7.3.5 addresses the configuration for the user. Section 7.3.6 concludes by comparing the resulting partitionings for different parameters.

7.3.1 Specification of the Partitioning

Structured meshes of the types `RegularFixedOfDimension<D>` or `StructuredDeformableOfDimension<D>` are partitioned for parallel execution by distributing the elements to all processes. As mentioned in Sec. 7.1.1, planar cuts in the space of the element indices separate the subdomains. For example, in computations on a structured 3D mesh with $N_x^{\text{el}} \times N_y^{\text{el}} \times N_z^{\text{el}}$ global elements, the process with rank r owns a subdomain with $N_x^{\text{el,local},r} \times N_y^{\text{el,local},r} \times N_z^{\text{el,local},r}$ local elements. The sizes of the local subdomains depend on the specified total number of subdomains n_i in each coordinate direction $i \in \{x, y, z\}$. Given n_i, the number of local elements in every subdomain along the coordinate axis i can be set to either $N_i^{\text{el,local}} = \lfloor N_i^{\text{el}}/n_i + 1 \rfloor$ or $N_i^{\text{el,local}} = \lfloor N_i^{\text{el}}/n_i \rfloor$ to allow for good load balancing.

A prerequisite to construct such a partitioning for n_{proc} processes is to fix the numbers of subdomains $n_x \times n_y \times n_z = n_{\text{proc}}$. In OpenDiHu, the Python settings file can either specify the global numbers N_i^{el} of elements or separate local numbers $N_i^{\text{el,local},r}$ of elements for every rank r. This step involves setting the option `inputMeshIsGlobal` to either `True` or `False` as explained in Sec. 6.2.3.

Specifying the global numbers of elements is often useful for toy problems, when the total element count is small and the actual partitioning is not important. In this case, PETSc is used to determine optimal subdomain sizes for all processes and, subsequently, constructing the partitioning. Because the partitioning is not yet known at the time of parsing of the Python settings, spatial information such as node positions or boundary conditions have to be specified on every rank for the whole domain.

Most of the electrophysiology examples, however, use the specification of local numbers of elements. Thus, every rank only needs to specify the local data of its subdomain, such as node positions and boundary conditions. This is a prerequisite for good parallel weak scaling behavior, as the amount of data processing on each process stays constant when simultaneously increasing problem size and total process counts.

In the electrophysiology examples, the partitioning into $n_x \times n_y \times n_z$ subdomains can be specified by the command line parameter `--n_subdomains n_x n_y n_z`, where `n_x`, `n_y` and `n_z` are replaced by the actual numbers. Their product has to match the process count n_{proc} that is given to MPI to start the program. If this option is not specified, the

values are determined automatically by the following algorithm: For all partitions of the number n_{proc} into three integer factors, a performance value p is computed as follows:

$$p = (n_x - n_{\text{opt}})^2 + (n_y - n_{\text{opt}})^2 + (n_z - n_{\text{opt}})^2.$$

The optimal value is given by $n_{\text{opt}} = n_{\text{proc}}^{1/3}$, which, in general, is not an integer. The partitioning with the lowest value of p is selected among all partitions, as it leads to nearly cuboid subdomains with the best volume-to-surface ratio. An advantage of this method is that it is independent of the mesh size.

7.3.2 Requirements for Partitioning and Sampling of the 3D Mesh Based on 1D Fiber Meshes

Next, we specify desired properties of the parallel partitioned 3D meshes, which are used together with 1D muscle fiber meshes in the discretization of fiber based multi-scale models. Subsequently, we construct an algorithm to generate the accordingly partitioned 3D meshes in parallel by sampling a finer dataset based on 1D fiber meshes.

Simulation scenarios with fiber based electrophysiology use a 3D muscle mesh and embedded 1D fiber meshes, which are generated from the same node positions as described in Sec. 3.5.14. The binary input file contains a structured grid of points, which can be either interpreted as 1D fibers by connecting the points in z-direction or as 3D mesh by additionally connection points in x and y-directions.

Usually, all points in such a file are used to define the 1D fiber meshes and the 3D mesh is constructed from only a subset of the available points. To obtain a 3D mesh with approximately equal mesh widths in all coordinate directions, the point data are sampled by constant strides in x, y and z direction. The stride in fiber direction (z direction) is typically chosen larger than the strides in transverse directions as the distance between the given points is smaller in this direction.

In the following, we discuss the sampling procedure that generates the partitioned 3D mesh from the fiber data in more detail. Given a structured hexahedral fine 3D mesh, numbers of subdomains n_i and sampling stride parameters `sampling_stride_i` for the three coordinate directions $i \in \{x, y, z\}$, we have to determine the nodes that should be part of each subdomain in the resulting coarser hexahedral 3D mesh.

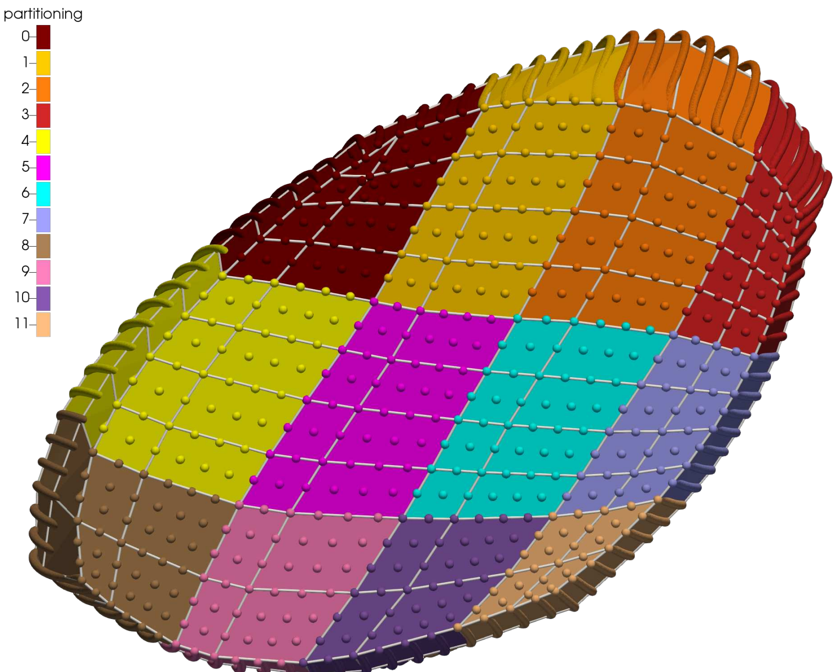


Figure 7.15: Partitioning and subsampling of a fiber mesh to twelve processes. The fiber data indicated by the spheres are sampled with a stride parameter of two to obtain the partitioned quadratic coarse mesh given by the white elements. The subdomains are indicated by different colors. The image shows a perspective view on the top 2D face of the 3D muscle mesh.

For illustration, Fig. 7.15 shows the initial fine mesh visualized by spheres that are arranged in fibers, that run from the shown cross-section to the back. The resulting sampled mesh is given by the white elements and uses a subset of the nodes in the fine mesh. The sampled mesh is partitioned into the colored subdomains. Furthermore, the coarse mesh consists of quadratic elements that are formed from two by two white standard elements, in the cross-section each. Hence, every subdomain contains an even number of the white elements in horizontal and vertical directions.

The requirements for the sampling and partitioning algorithm are as follows:

(i) The resulting coarser 3D mesh should use every kth node, where k is adjustable by the

parameter `sampling_stride_i` in the settings.

(ii) The number of nodes in every subdomain should be approximately equal to allow for a good load balancing in the computation.

(iii) There should be as little "remainder elements" that have a different mesh width than the majority of the elements as possible.

(iv) If a quadratic shape functions are required, e.g., for solid mechanics models, the number of (standard) elements in every subdomain in every coordinate direction has to be even to allow for the generation of quadratic hexahedral elements.

Clearly, not all requirements can be fulfilled exactly for all given input meshes. For combinations of given input mesh sizes and sampling strides that lead to an even number of sampled nodes, requirement (iv) cannot be fulfilled. Exact fulfillment of requirement (ii), i.e., an equal number of nodes in every subdomain is also only possible for suited parameter choices. Therefore, we relax requirement (i) and also occasionally allow different step widths between the selected nodes on the fine grid. Having varying distances between the nodes leads to elements with different mesh widths, which is unfavorable in terms of the numerical conditioning of the problem. Therefore, the number of such elements should be as low as possible, which is also stated by requirement (iii).

To avoid differently sized elements as far as possible, we work with a granularity parameter. This parameter specifies the amount of nodes to summarize and treat as an indivisible unit. For example, a value of `granularity_x=2` specifies that pairs of two neighboring points are in the same element. Then, subdomain boundaries and element boundaries can only occur at every second node.

7.3.3 Algorithm for Partitioning and Sampling of the 3D Mesh Based on 1D Fiber Meshes

Important steps in the algorithm for sampling the fine mesh and constructing the partitioning are, first, to determine the locations of the new subdomains in the original fine grid, second, to determine the number of sampled points in each subdomain and third, to determine which points from the fine grid will be sampled in every subdomain of the coarse grid. The steps have to be carried out independently for all three coordinate directions. Thus, it suffices to only consider the algorithm for the partitioning along one axis. In the following, we present the algorithms of the first two steps for the x-axis.

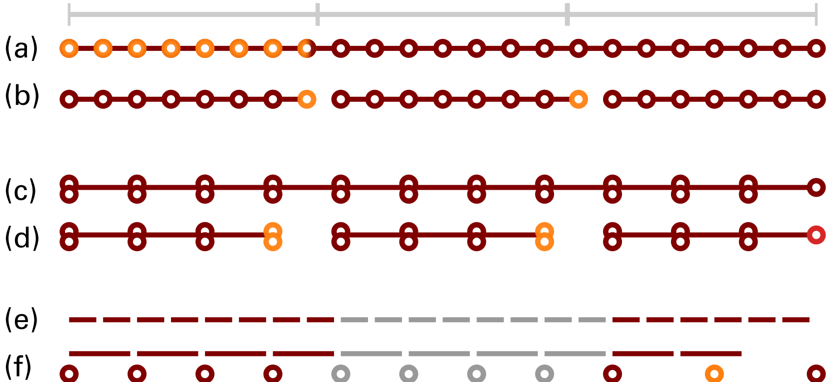


Figure 7.16: Visualization of the steps of the partitioning algorithms given by Algorithms 5 and 6 that yield the partitioning shown in Fig. 7.15.

The algorithm for the first step is given in Alg. 5. Input to the function `n_fibers_in_subdomain_x` is a subdomain coordinate in the range $[0, n_x - 1]$ that identifies the subdomain. The output to be computed is the number of grid points in the fine grid or, equivalently, the number of fibers that are contained in the subdomain. Calling this function for all subdomains defines the partitioning of the fine grid.

Algorithm 5 Computation of subdomain sizes, needed for the construction of a parallel partitioning.

```
 1 procedure n_fibers_in_subdomain_x(subdomain_coordinate_x)
   Input: Index of a subdomain in x-direction
   Output: Number of fibers that are contained in this subdomain

 2   α = ⌊ n_fibers_x / n_x / granularity_x ⌋ * granularity_x

 3   a1 = ⌊(n_fibers_x - n_x * α) / granularity_x ⌋     ⇝subdomains with > α nodes
 4   a2 = n_x - a1                                       ⇝subdomains with α nodes

 5   if subdomain_coordinate_x < a1 then                 ⇝first a1 subdomains
 6     return α + granularity_x
 7   else if subdomain_coordinate_x < n_x - 1 then
 8     return α
 9   else                                                ⇝last subdomain
10     return α + n_fibers_x % granularity_x
```

Figure 7.16 provides a visualization of the algorithmic steps, corresponding to the partitioning in vertical direction of the mesh shown in Fig. 7.15. Figure 7.16 (a) shows

a 1D mesh with `n_fibers_x=23` nodes or fibers. The goal is to partition them to $n_x = 3$ subdomains. According to requirement (ii), the nodes should be distributed equally to the subdomains. Dividing 23 nodes by 3 subdomains yields an average number of $7\frac{2}{3}$ nodes per subdomain, which is indicated by the orange color in Fig. 7.16 (a).

For now, we neglect the granularity parameter and set `granularity_x=1`. Line 2 of the algorithm computes the rounded down value α of the average number fraction. Every subdomain should obtain either α or $(\alpha+1)$ nodes. We specify that the first `a1` subdomains obtain $(\alpha+1)$ nodes and the remaining subdomains obtain α nodes. The amount of nodes that remain after we fill every subdomain with α nodes is the difference between all nodes `n_fibers_x` and $n_x \cdot \alpha$. This difference is equal to `a1` and the formula in line 3 of the algorithm computes the value of `a1` accordingly. The remainder number of subdomains `a2` follows as given in line 4. The visualization in Fig. 7.16 (b) shows that, in the example, `a1=2` subdomains obtain $\alpha + 1 = 8$ nodes and only the last subdomain, i.e., `a2=1`, obtains $\alpha = 7$ nodes.

The rest of Alg. 5 checks whether the given subdomain coordinate `subdomain_coordinate_x` refers to a subdomain with $(\alpha+1)$ or with α nodes by comparing the coordinate with `a1` in line 5. The first branch of the **`if`** statement returns the high number of nodes $(\alpha + 1)$, the other branches return the low number α, as far as the granularity parameter is neglected.

Next, we discuss the algorithm with a granularity value that is different from 1. Assuming a value of, e.g., `granularity_x=2`, always two neighboring nodes are grouped and the algorithm acts on these groups instead of individual nodes. The visualization in Fig. 7.16 (c) shows this grouping. Because the considered example has an odd total number of 23 nodes, only a single nodes remains for the last group.

The number of nodes per subdomains should now be a multiple of the granularity. This is ensured in line 2 of Alg. 5 by dividing by the granularity, rounding down and multiplying again with the granularity. The subdomains obtain either α or `(α + granularity_x)` nodes. The computation of the number `a1` of subdomains with the higher number of nodes in line 3 requires a division by `granularity_x` as every subdomain with the higher number takes `granularity_x` extra nodes. The rounding down in line 3 is needed to obtain an integer value even if the total number of nodes is not a multiple of the granularity.

In the example in Fig. 7.16 (d), the subdomains obtain either $\alpha = 6$ or $\alpha+$ `granularity_x` $= 8$ nodes. In fact, for the last subdomain, only seven nodes remain, as the total number of 23 nodes is not divisible by the granularity of two. In the algorithm, this is accounted

for by the last branch of the **if-else** construct in line 10, where only the remaining nodes are added to the last subdomain.

7.3.4 Algorithm for Sampling Points from a Fine Fiber Mesh

Next, we can sample points from the nodes that were assigned to each subdomain. The sampling process is parametrized by the value of `sampling_stride_x`, which specifies the step width of the nodes from the fine mesh to select for the coarse mesh. Algorithm 6 lists the function that determines the number of sampled points in a given subdomain. Similar to Alg. 5, the input is a 1D subdomain coordinate. The output is the number of sampled points in this subdomain.

Algorithm 6 Algorithm for sampling the fine mesh to obtain the coarser 3D mesh

```
 1 procedure n_sampled_points_in_subdomain_x(subdomain_coordinate_x)
   Input: Index of a subdomain in x-direction
   Output: Number of points in the subdomain for the coarse 3D mesh

 2   n = n_fibers_in_subdomain_x(subdomain_coordinate_x)
 3   if subdomain_coordinate_x == n_x - 1 then
 4     n -= 1

 5   if linear 3D elements then
 6     result = ⌊ n / sampling_stride_x ⌋
 7   else
 8     result = ⌊ n / (sampling_stride_x * 2) ⌋ * 2

 9   if subdomain_coordinate_x == n_x - 1 then
10     result += 1
11   return result
```

First, line 2 of Alg. 6 calls Alg. 5 to obtain the number of fine grid points in the subdomain. The number of elements `n` is equal to the number of points for all except the last 1D subdomain, which has one element less. This can be seen, e.g., in Fig. 7.15, where the first process with rank 0 (dark brown at the upper left) does not own the nodes on its subdomain boundary, whereas the last process with rank 11 (light brown at the lower right) owns all nodes on its subdomain boundary. Thus, lines 3 and 4 of Alg. 6 decrement the value of `n` to yield the correct number of elements.

The corresponding visualization in Fig. 7.16 (e) assumes `granularity_x=2` and shows $n = 8$ elements for both the first and the second subdomain and $n = 6$ elements for the last subdomain.

The resulting number of sampled points is obtained from the number of elements by a division by the sampling stride parameter and rounding down in lines 5 to 8. For the last subdomain, line 10 increments the result by one to account for the additional node on the boundary.

Depending on whether the sampled mesh should contain linear or quadratic elements, the number of elements obtained from the algorithm has no restriction, or it has to be even. This is checked in the `if` statement in line 5. In case of quadratic elements, an even number of elements is enforced by the formula in line 8.

In the considered example, we require quadratic elements and set `sampling_stride_x =2`. The visualization in Fig. 7.16 (f) shows the number of elements as long bars, which equals the `result` variable before line 9 in the algorithm. The resulting number of nodes is given in Fig. 7.16 (f) by the circles below.

The actual selection of the nodes from the fine grid according to the stride parameter and using the determined subdomains and their numbers of contained nodes is a straight-forward task and not part of the algorithms listed here. For quadratic elements in the last subdomain, the potentially different mesh widths are resolved by selecting the second-last node in the middle between the third-last and the last node. In Fig. 7.16 (f), this case occurs in the last subdomain. The orange node is sampled at the middle between the two neighboring dark red nodes. This behavior can also be observed in the corresponding partitioning in Fig. 7.15 for the elements given by white lines in the lowest row. These elements have a larger vertical mesh width of three sampled points than the other elements, which have a vertical mesh width of two sampled points.

7.3.5 User Options for the Algorithms

By adjusting the sampling stride and granularity parameters, it is possible to tune the outcome of the partitioning algorithms. The trade-off between the two requirements given in Sec. 7.3.2 by the numbers (ii) and (iii), i.e., that each subdomain obtains the same number of nodes, and that the least possible number of remainder elements is generated, can also be managed in the settings by enforcing either of the two requirements.

Moreover, we set the granularity parameters to the same value as the sampling parameters by default and additionally ensure for quadratic finite elements that the granularities are a multiple of two. This setting typically yields partitionings with equally sized elements. However, the number of nodes per subdomain is not always optimal.

To allow users to enforce a partitioning, where every rank gets the exact same number of nodes, except for the last subdomains in each coordinate direction, which potentially gets one layer of nodes less, we provide the option `distribute_nodes_equally`, which can be set in the variables files. If this option is set to `True`, the granularity values are internally fixed to one for "linear" meshes and to two for "quadratic" meshes, i.e., discretizations with quadratic finite element ansatz functions.

7.3.6 Results

The different results for the `distribute_nodes_equally` option are demonstrated in Figures 7.17 and 7.18. Figure 7.17 shows the automatic partitioning, where a simulation of fiber based electrophysiology with a grid of 9×9 fibers is executed with eight processes and the stride values `sampling_stride_x` and `sampling_stride_y` are set to two. By default, a linear mesh of 4×4 elements in x and y-directions is created with $2 \times 2 \times 2 = 8$ subdomains, as shown in Fig. 7.17a. Only the first four subdomains can be seen in the visualization, the other four are located behind and hidden in the background.

The distribution of the fibers to the two 1D subdomains along both x and y directions yields four fibers for the first and five fibers for the second 1D subdomain. Thus, the total 3D subdomains of the first four processes contain $16, 20, 20$ and 25 fibers.

Figure 7.17b shows the same scenario, except that the option `distribute_nodes_equally` has been set. The resulting partitioning is different and the fiber distribution is reversed, five and four fibers are assigned to the two 1D subdomains in both x and y directions. As a result, we get $25, 20, 20$ and 16 fibers for the first four 3D subdomains. Note that this is the best balanced partitioning of a structured mesh that is possible for 9×9 fibers. The subdomain sizes are the same as in Fig. 7.17a, except for a different order. However, for larger examples using more processes, the respective partitioning with the `distribute_nodes_equally` option is always optimal, whereas the balance rapidly degrades without this option.

While, in this example, there is no difference between Fig. 7.17a and Fig. 7.17b in terms of load balancing, the 3D mesh quality of the generated partitioning is worse for Fig. 7.17b. As can be seen in Fig. 7.17b, the first and the third subdomain have one layer of elements more in both x and y direction, and these elements have half the mesh width of the normal elements. Additionally, the second and fourth subdomain also contain elements of different mesh widths.

Similar effects can also be studied in the scenario of Fig. 7.18, where the same mesh is partitioned to four processes in z-direction. The number of nodes in z-direction is 1481 and the sampling stride is chosen as `sampling_stride_z=50`. Figures 7.18a and 7.18b show the resulting partitioning without and with the `distribute_nodes_equally` option. Again, the second scenario shows "remainder" elements with smaller mesh widths at the boundaries of every subdomain. The distribution of nodes is $400, 350, 350$ and 381 nodes per subdomain in Fig. 7.18a and $371, 370, 370$ and 370 nodes per subdomain for the scenario in Fig. 7.18b, where the `distribute_nodes_equally` option has been set. The first case has the better 3D mesh quality, whereas only the second case yields the perfect load balancing.

In summary, it is possible to tweak the created partitioning by adjusting the sampling stride and deciding between mesh quality and perfect load balancing. For electrophysiology simulations, which impose high computational load because of the subcellular model, the load balancing aspect is more important and the option `distribute_nodes_equally` should be set to `True`. In simulations with elasticity models, the quality of the 3D meshes is more important and the partitioning for the corresponding meshes should be parametrized with the `distribute_nodes_equally` option set to `False`.

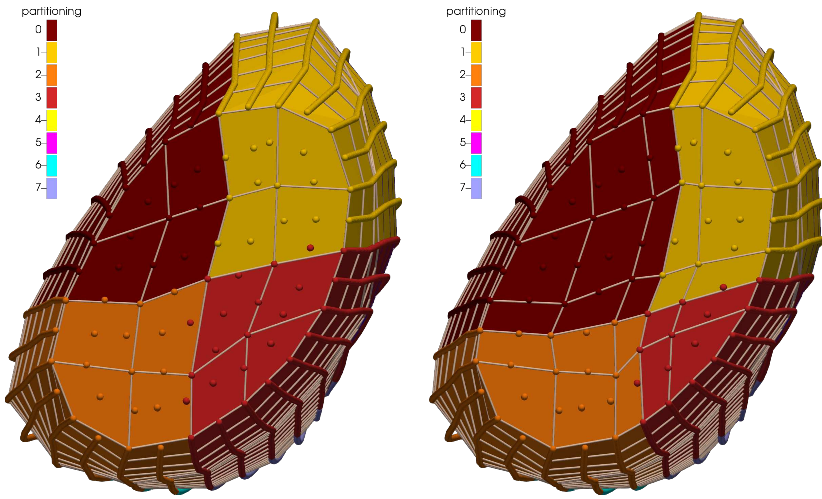
How To Reproduce

The partitioning in Fig. 7.15 is obtained by the following simulation:

```
cd $OPENDIHU_HOME/examples/electrophysiology/fibers/
  ↪ fibers_contraction/no_precice/build_release
mpirun -n 12 ./biceps_contraction ../settings_biceps_contraction.py
  ↪ partitioning_demo.py --n_subdomains 4 3 1
```

The partitionings in Figures 7.17 and 7.18 are created by the following simulations. For Figures 7.17a and 7.18a, edit the variables file `partitioning_demo.py` and set `distribute_nodes_equally = False`. For Figures 7.17b and 7.18b, set `distribute_nodes_equally = True`.

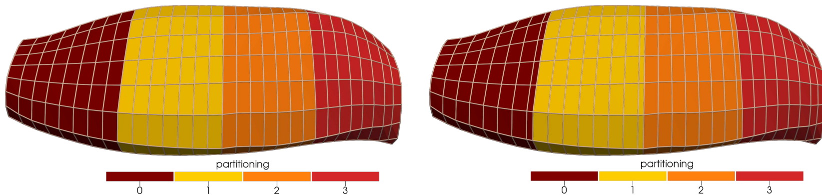
```
cd $OPENDIHU_HOME/examples/electrophysiology/fibers/fibers_emg/
  ↪ build_release
mpirun -n 8 ./fast_fibers_emg ../settings_fibers_emg.py
  ↪ partitioning_demo.py
mpirun -n 4 ./fast_fibers_emg ../settings_fibers_emg.py
  ↪ partitioning_demo.py --n_subdomains 1 1 4
```



(a) Resulting sampled mesh with the option `distribute_nodes_equally=False`.

(b) Resulting sampled mesh with the option `distribute_nodes_equally=True`.

Figure 7.17: Mesh partitions generated by the sampling algorithm with different settings. A fine mesh with 49 fibers is sampled with a stride parameter of two and partitioned to eight processes.



(a) Resulting sampled mesh with option `distribute_nodes_equally=False`. The mesh width is constant, but the partitioning is not perfectly balanced.

(b) Resulting sampled mesh with option `distribute_nodes_equally=True`. The partitioning is perfectly balanced, but the mesh width is not constant.

Figure 7.18: Sampling a mesh along the fiber direction. The original mesh has 1481 nodes and is sampled with a stride value of 50.

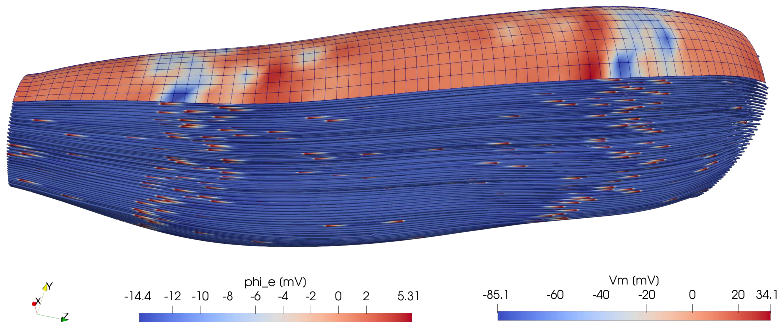


Figure 7.19: Simulation result of the fiber based electrophysiology model with 1369 muscle fibers and a 2D surface mesh on top of the muscle. The fibers are colored according to the transmembrane potential V_m, the surface is colored according to the EMG values given by the extracellular potential ϕ_e.

7.4 Parallel Solver for the Fiber Based Electrophysiology Model

After discussing the general partitioning and sampling of 3D and 1D meshes in the last section, we now focus on the concrete application for the fiber based electrophysiology model. We describe our basic solver and algorithmic improvements that yield lower runtimes.

The fiber based electrophysiology model consists of the action potential propagation model given by the 1D monodomain equation Eq. (5.11) and a 0D subcellular model as described in Sec. 5.1.1. The 0D and 1D problems are solved on the 1D fiber meshes. They are coupled to the 3D bidomain problem given in Eq. (5.9a), which computes the EMG values. In summary, the components (b2),(c) and (d) of the diagram in Fig. 5.1 are involved in this computation. Figure 7.19 shows a simulation result of this model, where the 1D fibers and the surface of the 3D mesh can be seen.

In the following, Sec. 7.4.1 begins with a description of the solver structure and the parallelization. Subsequently, performance improvements considering the parallel execu-

tion of the solver are discussed. Section 7.4.2 presents a variant, where a faster solver is employed for the 1D part of the computation. Section 7.4.3 shows how the computational load can be reduced by only computing activated parts of the muscle.

7.4.1 Parallel Solver Structure

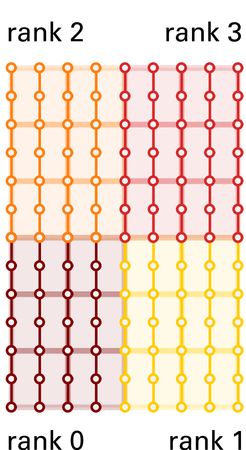
For better visualization, we consider the 2D setting of a mesh and embedded 1D fibers partitioned to 2×2 processes as shown in Fig. 7.20a by different colors. However, all discussions are also valid for the real 3D setting shown in the last section and for arbitrary partitionings to $n_x \times n_y \times n_z$ processes.

Figure 7.20b shows the program structure of the example that solves the fiber based electrophysiology model. The outer class is a `Coupling` that alternates between computing the monodomain equation Eq. (5.11) on the 1D fibers and computing the static bidomain equation Eq. (5.9a) on the 3D domain. The second part, the bidomain solver, is given in Fig. 7.20b by the class `StaticBidomainSolver`, which includes two `FiniteElementMethod` classes. The first class solves the potential flow to obtain the fiber direction for the anisotropic conduction tensor, the second class is used to discretize the spatial derivatives in the bidomain equation.

The first part of the coupling scheme in Fig. 7.20b consists of a `MultipleInstances` class, which encloses the Strang operator splitting. The splitting has two child solvers for the subcellular model and the diffusion or conduction term. The first child consists of another `MultipleInstances` class with a `Heun` scheme and the `CellmlAdapter`. The second child of the Strang splitting also consists of a `MultipleInstances` class and a combination of an `ImplicitEuler` scheme (alternatively a `CrankNicolson` scheme can be used) and a `FiniteElementMethod`.

A `MultipleInstances` class can be used to apply a solver to more than one problem of the same kind. The class allows to specify a number of instances of its nested solver. Each instance can be given a subset of processes that will take part in the computation of the instance. Each process then iterates over all instances, for which it is part of the subset. Thus, the nested solver of a `MultipleInstances` class is called in series for all instances that share a process/MPI rank, and it is called in parallel and independently for 1D model instances that have disjoint subsets of ranks.

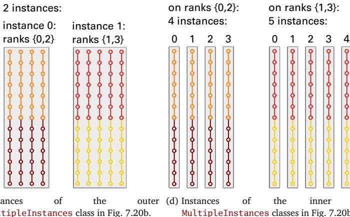
Furthermore, the class provides a common output writer, which collectively writes the data of all instances. This allows, e.g., to create a single output file in every timestep



(a) Visualization of the 3D mesh with embedded 1D fibers, partitioned to four ranks.

Coupling
MultipleInstances (c)
Strang
MultipleInstances (d)
Heun
CellmlAdapter
MultipleInstances (d)
ImplicitEuler
FEM
StaticBidomainSolver
FEM (Bidomain eq.)

(b) Structure of the OpenDiHu example program to solve the fiber based electrophysiology model. The colors match the scheme introduced in the overview chart in Fig. 5.1.



(c) Instances of the outer `MultipleInstances` class in Fig. 7.20b.

(d) Instances of the inner `MultipleInstances` classes in Fig. 7.20b.

Figure 7.20: Visualizations for the discussion of the program structure and partitioning used for fiber based electrophysiology simulations. The circles and lines represent the 1D meshes, their coloring indicates the MPI rank.

containing the data of all fibers. Especially for large scenarios, this is more practical than having as many output files as fibers.

The settings that have to be specified in the Python file for a `MultipleInstances` class comprise the number of instances and a list with the according number of entries, which further configure the instances. Each list entry can be `None` if the rank does not take part in the computation of the corresponding instance. Otherwise, the list entry consists of (i) a specification of all ranks that should collectively compute the corresponding instance and (ii) the settings of the corresponding nested solver.

The own MPI rank of a process is known in the Python settings file. This allows to specify different settings for different ranks in the same file. By omitting the configuration of irrelevant instances and setting their list entry to `None`, the amount of data is reduced and parsing of the script is sped up, especially for large problem sizes.

The settings and corresponding subdomains of the `MultipleInstances` classes that are indicated by (c) and (d) in Fig. 7.20b are shown in Figures 7.20c and 7.20d, respectively. As can be seen in Fig. 7.20c, the outer `MultipleInstances` class separates the subdomains that are not connected by any fibers, such that they can be computed in parallel and independently of each other. In the example of Fig. 7.20a, the subdomains of ranks 0 and 2 can be computed independently of the subdomains of ranks 1 and 3. As a consequence, all processes specify that their `MultipleInstances` class has two instances. At rank 0, the list of instance settings contains the settings of the nested Strang solver with all information of rank 0's subdomain (in the first item) and the value `None`, as rank 0 has no information about fibers outside its subdomain (in the second item). Ranks 1, 2 and 3 specify their subdomain accordingly, as shown in Fig. 7.20c.

During computation, ranks 0 and 2 as well as ranks 1 and 3 enter the `Strang` solver class collectively with a shared MPI communicator. The inner `MultipleInstances` classes employ the 0D subcellular and the 1D electric conduction solver on multiple fibers. As shown in Fig. 7.20d, ranks 0 and 2 specify four instances with the settings of the four shared fibers. At the same time and concurrently, ranks 1 and 3 specify five instances with settings for their five shared fibers.

Note that the multiplicity of the 0D instances on a fiber is not achieved by another `MultipleInstances` class, but the model is solved for all points on the mesh together, using parallelism on the lower, instruction-based level.

These different splits of the geometry allow to compute the electrophysiology model on the fibers in parallel. The partitioning of the domain has to be the same for the 3D mesh

and the embedded fibers to allow value mapping from the fibers to the 3D mesh without communication. The fibers are oriented along the z-direction in the 3D setting. This explains, why the ranks for a particular fiber, e.g., $\{0,2\}$ or $\{1,3\}$ are not direct successors of each other but increasing with a stride equal to the number of subdomains in x and y directions, $n_x \cdot n_y$.

7.4.2 Improved Parallel Solver Scheme using the Thomas Algorithm

The monodomain model, which is solved on each fiber, consists of a reaction-diffusion equation, which is solved using the Strang operator splitting. The diffusion part uses an implicit timestepping scheme, which leads to a linear system of equations to be solved in every timestep. As the finite element method with linear ansatz functions is used for spatial discretization, this linear system has a tridiagonal system matrix.

In the solver tree structure in Fig. 7.20b, this solution step occurs in the solvers under the second inner `MultipleInstances` class. As can be seen in Fig. 7.20d, the dofs of each fiber that are part of this linear system are partitioned to multiple processes. Hence, this linear system is solved using a parallel conjugate-gradient solver of PETSc.

However, there is the possibility to improve the performance by exploiting the tridiagonal matrix structure. The *Thomas algorithm* is the specialization of Gaussian elimination for this matrix type and is known to efficiently solve such a system in linear time complexity. More specifically, it only requires a first downwards sweep through the matrix entries for forward substitution and a second upwards sweep for back substitution to compute the solution. It is stable for diagonally dominant matrices and this condition is met for the governing system matrix.

As the Thomas algorithm is not parallel, we have to gather the matrix data on a single process in order to employ the algorithm. In OpenDiHu, the `FastMonodomainSolver` class is tailored to the parallel solution of fiber based electrophysiology using the Thomas algorithm. Figure 7.21 outlines the steps performed by the `FastMonodomainSolver` class.

During initialization, the `FastMonodomainSolver` class initializes its nested solver tree as normal and the parallel partitioning of the fibers is carried out as described in Sec. 7.4. This is visualized on the left in Fig. 7.21 for four fibers and two ranks. At the beginning of the first timestep, the communication to gather complete fiber data on single processes is carried out. The fiber data are communicated, such that every fiber is completely

accessible at a single processes. The assignment of the fibers to processes occurs in a round-robin fashion, i.e., the first fiber is sent to rank 0, the second to the next rank, etc. As a result, every process has approximately the same number of complete fibers. This is shown in the middle image of Fig. 7.21, where the red colored rank has all values of the first and third fiber and the orange colored rank has all values of the second and fourth fiber.

The processes then each compute the full monodomain model consisting of the Strang splitting with the subcellular model on the nodes of each fiber and the diffusion part using the Thomas algorithm. This is done in a separate serial implementation for the now locally owned fibers, i.e., not using the nested solvers. The solution is obtained for as many subsequent timesteps as were specified in the settings. When the end time of the enclosing coupling scheme is reached, the fiber data are communicated back to the original partitioned fibers, as shown on the right in Fig. 7.21. Then, the coupling scheme continues with the data mapping from the partitioned fibers to the 3D domain and with the `StaticBidomainSolver`. Afterwards, the `FastMonodomainSolver` is called again and performs its computation anew starting with the communication step.

In the C++ file, the `FastMonodomainSolver` class is inserted as a wrapper to the outer `MultipleInstances` class that is indicated by (c) in the solver structure in Fig. 7.20a. In the Python settings, the class does not add an additional nesting level such that the same settings file can be used for programs with and without the `FastMonodomainSolver` class and yields the same simulation results.

In summary, the efficient serial computation of the monodomain model in the `FastMonodomainSolver` is wrapped by communication steps of the partitioned fiber data. The frequency of this communication step is determined by the timestep width of the coupling scheme. The scenario solves the bidomain equation to simulate EMG signals. A typical sampling frequency of EMG capture devices is $f = 2\,\text{kHz}$, which corresponds to a coupling timestep width of $dt_{3\text{D}} = 0.5\,\text{ms}$. The timestep widths $dt_{0\text{D}}$ of the subcellular model and $dt_{1\text{D}}$ of the diffusion term have to be set at maximum to $10^{-3}\,\text{ms}$, yielding 500 timesteps of computations on the fiber between subsequent communication steps. As a result, the communication cost is negligible.

7.4.3 Adaptive Computation of the Subcellular Model

During simulations of the fiber based electrophysiology model, often only a small fraction of the given fibers is activated. The reason is, that, in physiological conditions, the smaller

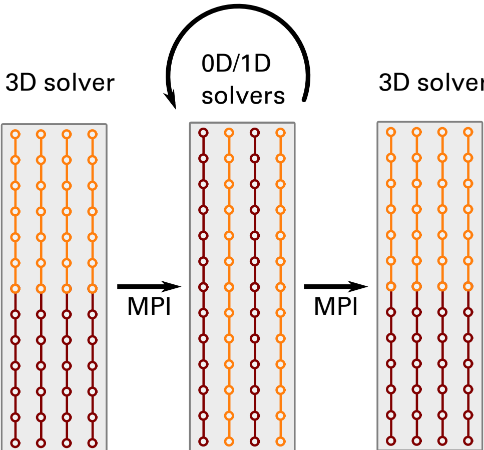


Figure 7.21: Algorithmic steps of the `FastMonodomainSolver` to efficiently solve the 0D/1D problems. After a timestep of the 3D solver (left), the partitioned fibers, visualized by different colors for the MPI ranks, are communicated using MPI, such that every fiber is accessible on a single MPI rank (middle). The 0D/1D solvers compute multiple subsequent timesteps until the next 3D coupling step. Then, the original partitioning is restored by a second communication step and the 3D solver can continue with the next timestep (right).

MUs are activated first and the larger MUs only get activated when the full force of the muscle is required. As the majority of the fibers belongs to larger MUs, a high portion of fibers is less frequently activated, also depending on the scenario. But even if the scenario specifies a tetanic stimulation of all MUs, the larger MUs have lower stimulation frequencies, which again leads to less action potentials on large MUs than on smaller MUs in the same time span.

A naive solver of the monodomain models always computes all 1D electric conduction problems on the fiber meshes and all 0D subcellular models on the nodes of the fiber meshes, regardless of their activation state. In the following, we present a method in OpenDiHu that exploits the infrequent activation events on most of the fibers while obtaining the same solution as the naive solver.

We assume that the subcellular models are initialized in their equilibrium state, where the temporal derivative of the state vector $\mathbf{y}$ vanishes, $\partial\mathbf{y}/\partial t = 0$. The first algorithmic improvement is to only consider those fibers in the solver that have yet been stimulated.

This improves the performance especially for “ramp like” motor recruitment, where more and larger MUs are activated over time. However, after all MUs have been activated at least once, all fibers are computed again and no more performance improvement is obtained.

The second improvement is to only compute instances of the subcellular model at those points, where it is not in equilibrium. To determine, whether an instance of the subcellular model is in equilibrium, we compare the solution before and after one integration step by the Heun method. Only if the relative change of any component of the state vector $\mathbf{y}$ is larger than 10^{-5}, we consider the model to be not in equilibrium.

This check requires to compute the solution of the subcellular model, the avoidance of which is subject of the improved scheme. Therefore, we use the property of the 1D diffusion problem discretized by linear finite elements that the value at one spatial point can only influence its two neighbors in a single timestep. This allows us to avoid checking the equilibrium condition at points that are surrounded by other points in equilibrium. This means that the subcellular model does not have to be solved at most points in equilibrium, which drastically reduces the runtime. The 1D electric conduction problem, however, has to be solved for the whole fiber mesh if at least one point it is not in equilibrium.

In our method, each subcellular point can be in one of the three states “active”, “inactive” and “neighbor is active”. If the subcellular model is not in equilibrium, the point is in the state “active” and has to be solved in the next timestep. If the subcellular model is in equilibrium and does not have to be solved because the solution vector stays constant, the point is in the state “inactive”. The state “neighbor is active” occurs for a previously inactive point, of which at least one neighbor became active and, thus, the check if the point is still in equilibrium has to be performed and the subcellular model has to be solved in the next timestep. After each solution step, the state of a point changes according to the transitions given in Fig. 7.22.

An active point stays active, if the solution has changed in the last numerical integration step. It transitions to inactive, if the solution did not change. The same applies to points in the state “neighbor is active”, which also change to “active” or “inactive” after one timestep. An inactive state cannot be activated by a check on the point itself, as this state implies that no computation and no subsequent equilibrium check are carried out. The only transition for a point A from an inactive state occurs, when a neighbor point B reaches the state “active” (or for external stimulation). Then, point A changes to “neighbor is

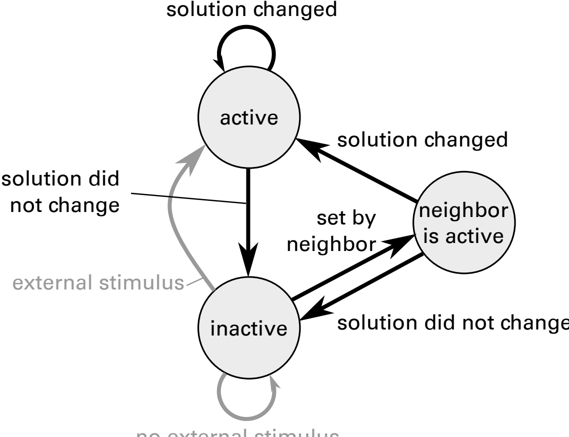


Figure 7.22: Transition diagram for the adaptive computation of the subcellular model. The diagram shows the transition between local states of points on the fibers. Points in inactive state do not perform the computation of the 0D subcellular model.

active". For propagating action potentials along a fiber that is in the "inactive" state, this leads to a propagating front of points in the "neighbor is active" state.

Initially, all states are set to "active". If no stimulation occurs and the subcellular model is in equilibrium, they momentarily change to "inactive". Upon external stimulation, the stimulated points are automatically set to "active" and their neighbors are set to "neighbor is active" such that the effect of the stimulation can be considered in subcellular model computations.

Figure 7.23 shows a simulation, where the effect of both improvements is visible. The Hodgkin-Huxley subcellular model has been solved on a set of 49 fibers. At the displayed time of $t = 28\,\text{ms}$, two MUs have been activated. The value of the membrane potential V_m is visualized by the radius of the fibers. The active or inactive state of the improved scheme is indicated by the colors.

It can be seen that several fibers have gray color which indicates that they have not yet been stimulated and, thus, are not part of the computation. The other fibers have been stimulated either by the first or the second MU. Action potentials at two different distances from the center corresponding to the two MUs can be identified by the bulbous shapes. The red parts of the fibers contain the active points, where the subcellular model

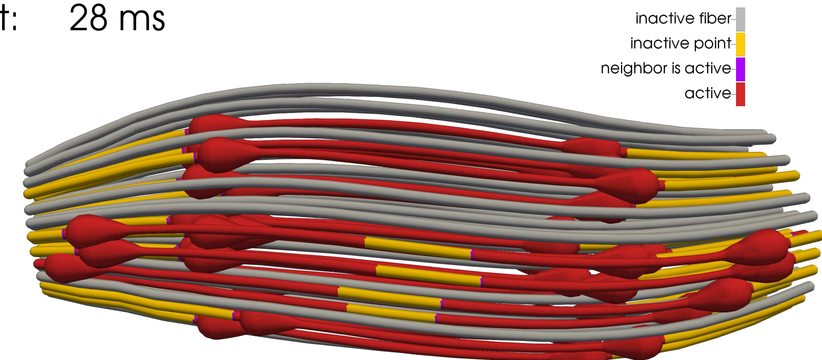


Figure 7.23: Simulation scenario that demonstrates the adaptive computation method of fibers and subcellular points. A simulation of the monodomain equation on a set of 49 fibers with the subcellular model of Hodgkin and Huxley is shown. The transmembrane potential is visualized by the fiber radius. The states of the points used in the algorithm are given by the different colors.

is not in equilibrium. At the yellow regions, the subcellular models are in equilibrium, and no computational work is performed there. The yellow regions are at the outer ends of the fibers that were not yet reached by the action potentials as well as around the center for fibers of the first MU. This demonstrates the repolarisation effect, after which the model reaches its equilibrium state again.

The purple colored points are in the state “neighbor is active” and can be found between active and inactive points. As the algorithm iterates over all points of a fiber from left to right, these purple points only occur at the left boundaries of active regions. At their right boundaries, the initial “neighbor is active” points transition to “active” or “inactive” directly after the computation step within this iteration.

Instead of individual nodes on the fiber mesh, our implementation treats SIMD vectors of four or eight such adjacent nodes (depending on the hardware capabilities) as one point in the algorithm. If one of these nodal instances is not in equilibrium, the whole SIMD vector is considered not in equilibrium and transitions to the “active” state. This coarser granularity of the model instances allows to solve the subcellular problem in chunks according to the SIMD lane width using SIMD instructions.

How To Reproduce

The scenario of Fig. 7.23 can be run as follows:

```
cd $OPENDIHU_HOME/examples/electrophysiology/fibers/fibers_emg/
  ↪ build_release
mpirun -n 4 ./fast_fibers_emg ../settings_fibers_emg.py
  ↪ compute_state_demo.py
```

Instead of four processors, you can use as many as you have to speed up the computation.

7.5 Parallel Solver for the Multidomain Electrophysiology Model

After the details on the parallel partitioning and solvers for the fiber based electrophysiology model have been discussed in Sections 7.3 and 7.4, we now consider the multidomain based model of electrophysiology, which includes electric conduction in the body fat layer. The class for the implicit solver within the operator splitting is the `MultidomainWithFatSolver` class, which has been introduced in Sec. 6.2.4.

The multidomain based electrophysiology model contains the two multidomain equations, Eqs. (5.14) and (5.15), which are solved on the 3D domain. The model leads to a large linear system of equations that is solved in every timestep, described in Sec. 5.3.5. Figure 7.24 visualizes the body fat and muscle domains, on which the multidomain model is solved. The coloring of the muscle domain also gives an example for the occupancy factor f_r^k, which specifies to which extend every point in the domain is occupied by a particular MU.

7.5.1 Construction and Partitioning of the Mesh

The mesh used in this solver is a composite mesh of type `Mesh::CompositeOfDimension<D>`, as introduced in Sec. 7.1.7. Figure 7.25 shows the layout, in particular how the mesh of the body fat domain Ω_B is connected with the mesh of the muscle domain Ω_M. The muscle and body fat meshes have $N_x^{\text{el}} \times N_y^{\text{el}} \times N_z^{\text{el}}$ and $(N_x^{\text{el}} + N_y^{\text{el}}) \times N_{\text{fat}}^{\text{el}} \times N_z^{\text{el}}$ elements, respectively. Only the muscle mesh has been generated from medical imaging data by the pipline given

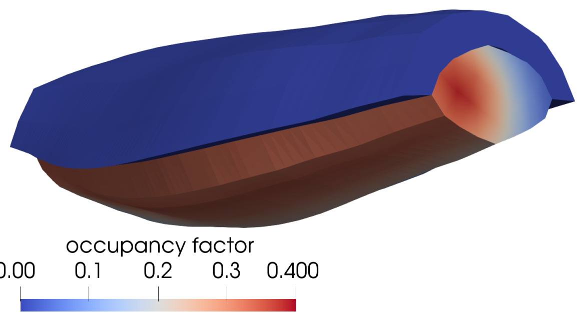


Figure 7.24: Visualization of the domains for the multidomain electrophysiology model: The body fat domain is shown in blue, the muscle domain is colored according to the values of one occupancy factor f_r^k, which specifies the territory of a MU.

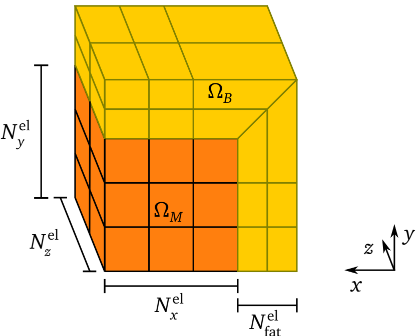


Figure 7.25: Layout of the composite 3D mesh for the multidomain model with fat layer. The orange elements belong to the mesh of the muscle domain Ω_M, the yellow elements are added on top to represent the body fat domain Ω_B.

in Chap. 3. The fat mesh is created on top of the muscle mesh geometry and has to use the same number of elements as the muscle mesh along the muscle surface for compatibility in the composite mesh. Only the physical thickness of the adipose tissue layer and the corresponding number $N_{\text{fat}}^{\text{el}}$ of elements in radial direction have to be specified. (The mesh generation step is implemented in the script `create_fat_layer.py`.)

Figure 7.27 shows such as composite mesh. The muscle mesh is based on a dataset of 13×13 fibers with 1481 nodes per fiber. This fine mesh is sampled as described in Sec. 7.3.1 with stride values of $3, 3$ and 20 in x, y and z directions and `distribute_nodes`

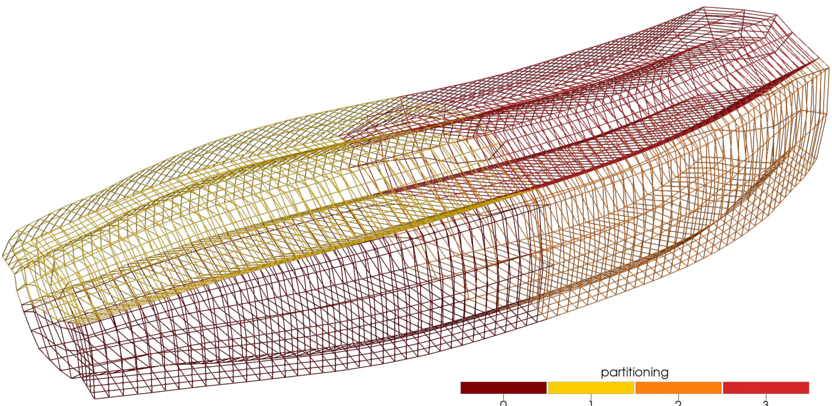


Figure 7.26: Composite mesh of the multidomain example, partitioned into four parallel subdomains.

`_equally=True`. As a result, we get $N_x^{\text{el}} \times N_y^{\text{el}} \times N_z^{\text{el}} = 5 \times 4 \times 75$ elements. The fat mesh consists of a 1 cm adipose tissue layer with $N_{\text{fat}}^{\text{el}} = 4$ elements. The muscle and fat meshes have 2280 and 3800 dofs.

The partitioning of the composite mesh into $n_x \times n_y \times n_z$ subdomains cannot be chosen arbitrarily. The reason is that both the muscle and the body fat mesh have to be partitioned into the same number of subdomains. If, e.g., a partitioning of $n_x = n_y = 2$ is chosen, the cube in Fig. 7.25 gets divided by one horizontal planar cut and one vertical planar cut. This divides the orange muscle mesh into four subdomains as expected. The yellow body fat mesh, however, is only partitioned to three of the four processes as there are no yellow elements below the horizontal cut and left of the vertical cut.

Thus, a valid partitioning can only be created if either n_x or n_y is set to one. Because there is no restriction on n_z, the total mesh can still be partitioned in two dimensions to a product of subdomains, either as $1 \times n_y \times n_z$ or as $n_x \times 1 \times n_z$. The example mesh in Fig. 7.27 is partitioned to $2 \times 1 \times 2$ subdomains as shown by the different colors.

7.5.2 Structure of the System Matrix

Figure 7.27 shows the solver structure of a simulation of the multidomain model. The Strang operator splitting couples the Heun scheme of the 0D subcellular model with the multidomain solver, which is given by the `MultidomainWithFatSolver` class. The

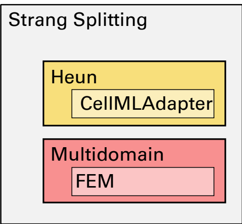


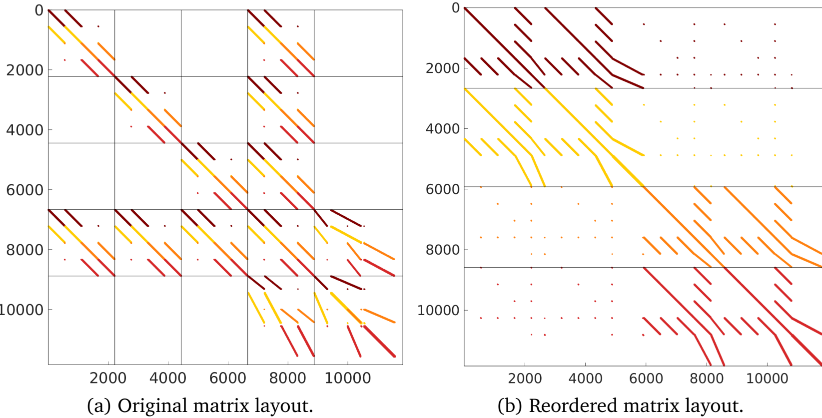
Figure 7.27: Solver structure of the multidomain solver, consisting of the Strang operator splitting, which contains the two nested solvers for the subcellular model (Heun scheme with CellML adapter) and the solver for the diffusion part of the multidomain problem. The colors match the scheme introduced in the overview chart in Fig. 5.1.

`MultidomainWithFatSolver` class uses nested `FiniteElementMethod` classes to describe the anisotropic electric conduction in the muscle domain and the isotropic electric conduction in the fat domain. The system matrix for the system of equations is given in Eq. (5.69) in Sec. 5.3.5. The solver calculates the matrix block entries using the stiffness and mass matrices computed by the nested `FiniteElementMethod` classes, i.e., the unknowns are organized in blocks in the system matrix for each MU.

Figure 7.28a shows the location of non-zeros in the resulting sparse matrix for three MUs. The matrix blocks are indicated by boxes and correspond to the symbolic formulation given in Eq. (5.69). The first three blocks correspond to the electric conduction problems of the 3 MUs given by the second multidomain equations in Eq. (5.15), the fourth row and column of blocks corresponds to the first multidomain equation Eq. (5.14), and the last block corresponds to the electric conduction problem in the fat domain. It can be seen that the dimension of the last block is different, corresponding to the number of dofs in the fat mesh.

In this visualization, it may seem that most of the blocks only have three non-zero entries per row, however, the actual number is higher (the “lines” consist of multiple diagonals of non-zero entries) with a maximum of 27 entries, as the finite element ansatz function of a node in the 3D mesh has overlapping support with the ansatz functions of other nodes in a $3 \times 3 \times 3$ grid. The actual non-zero structure per block is close to the example shown in Fig. 7.10.

The colors in Fig. 7.28a correspond to the four processes, as defined in the partitioned mesh in Fig. 7.27. The entries in every block are all partitioned in the same way to the



(a) Original matrix layout.

(b) Reordered matrix layout.

Figure 7.28: Nonzero structure of the system matrix of the multidomain problem for three MUs. The five blocks in every row and column in (a) correspond to the dofs of the three MUs, the extracellular potential in the muscle domain, and the body potential.

four processes, as given by the partitioning of the nested `FiniteElementMethod` classes. The data structure for this layout is the `MATNEST` type of PETSc.

However, to be able to apply the multitude of PETSc solvers to this linear system, the matrix has to be transferred to the canonical parallel matrix layout of PETSc, which groups all dofs of the subdomains together. As this conversion is not available in PETSc, it is done in OpenDiHu by reordering the dofs and, as a consequence, the matrix entries. The same permutation is applied to the rows and to the columns of the matrix. The result of this operation is shown in Fig. 7.28b. It can be seen that the portions for each process are now consecutive matrix rows. The non-zero structure within each process resembles the global matrix structure of the original matrix.

7.5.3 Properties of a Diagonal Block-Matrix for the Preconditioner

With the reordered matrix, the linear system can now be solved using almost any preconditioner and linear solver of the PETSc framework. For the construction of the preconditioner $\mathcal{P}$ with left preconditioning matrix $P = \mathcal{P}(A)$, we can either use the system matrix A or provide a different matrix A'. The preconditioned linear system $P^{-1}A$ should have a

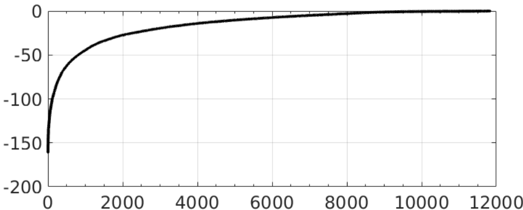


Figure 7.29: Real parts of the eigenvalues sorted by magnitude and corresponding to the example in Fig. 7.27. The non-zero eigenvalue with largest and smallest absolute values are $\lambda_{\text{max}} = -161.2576$ and $\lambda_{\text{min}} = -0.0116$.

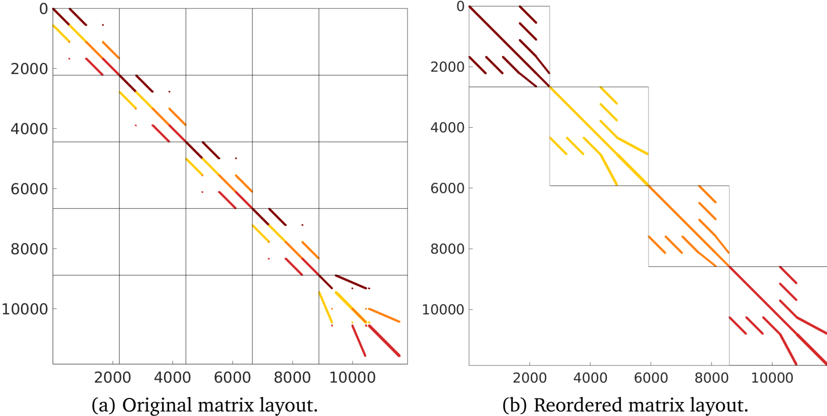
smaller condition number than A and, thus, solving the preconditioned system iteratively should be significantly faster than the original A system.

To compute the condition number of the system matrix A, we determine its spectrum. Figure 7.29 shows the real parts of all eigenvalues of A. The imaginary parts vanish for almost all eigenvalues. The matrix is singular with one zero-eigenvalue. This property corresponds to the fact that the membrane potential in the problem is abitrary with respect to a constant offset. The singular problem can be solved using appropriate iterative solvers.

The real parts of the eigenvalues are all negative, which is in line with the fact that the model consists of a combination of several diffusion problems. The progression in Fig. 7.29 shows a large difference between the largest and the smallest eigenvalues. The condition number of A can be computed by $\text{cond}(A) = |\lambda_{\text{max}}|/|\lambda_{\text{min}}| = 161.2576/0.0116 \approx 1.4 \cdot 10^5$. Thus, the problem is ill-conditioned and can benefit from for preconditioning. The condition number is also dependent on the spatial mesh resolution and increases for larger problem sizes.

We experiment with a preconditioning matrix that only uses the diagonal blocks of the system matrix in reordered matrix layout, as shown in Fig. 7.28b. Figure 7.30b shows the non-zero structure of the resulting matrix and compares it with the non-zero structure of the diagonal blocks of the matrix in original ordering in Fig. 7.30a. As all diagonal blocks are symmetric matrices on both orderings, the resulting block-diagonal matrices A' are also symmetric in contrast to the original matrix A.

Furthermore, it can be seen that the matrices in Fig. 7.30 are different. The reordered layout depends on the parallel partitioning and contains only matrix entries within one



(a) Original matrix layout. (b) Reordered matrix layout.

Figure 7.30: Nonzero structure of the symmetric preconditioner matrix of the multidomain problem. The symmetric matrices are obtained from the full matrices in Fig. 7.28 by removing all blocks outside the main diagonal.

subdomain, i.e., decouples the problems for different subdomains. In contrast, the diagonal blocks of the original system matrix are independent of the partitioning and contain dependencies between dof in different subdomains. We use the reordered diagonal matrix for the preconditioner, as this approach is compatible with the parallel matrix storage in PETSc and allows to use the preconditioners and solvers of PETSc. The decoupled entries on every rank potentially allows for a faster computation in the application of the preconditioner.

7.5.4 Mesh and Matrices for Higher Degrees of Parallelism

To show the effect of a higher degree of parallelism on the matrix structure, we also partition the same mesh as in Fig. 7.27 to 16 processes. The resulting partitioning of the mesh is given in Fig. 7.31. Figure 7.32 shows the non-zero structure of the system matrix and the diagonal matrices for the preconditioner. Figure 7.32a contains the original matrix structure that is permuted to the structure in Fig. 7.32b. The symmetric matrices for the preconditioner are shown in Figures 7.32c and 7.32d. A comparison with Fig. 7.30 shows that the width of the non-zero band decreases for higher parallelizations.

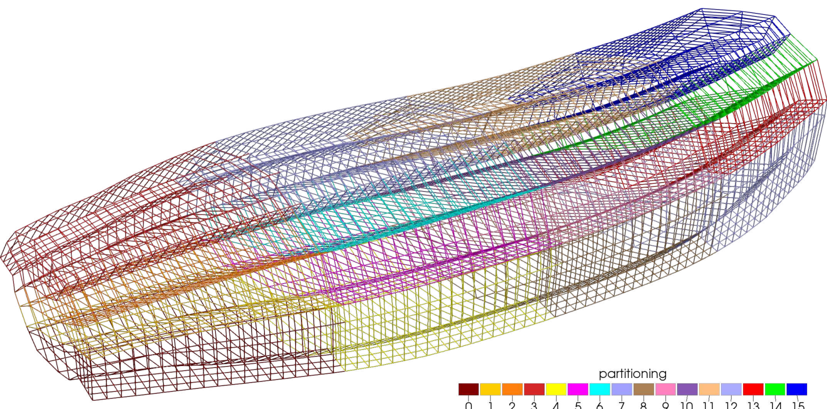


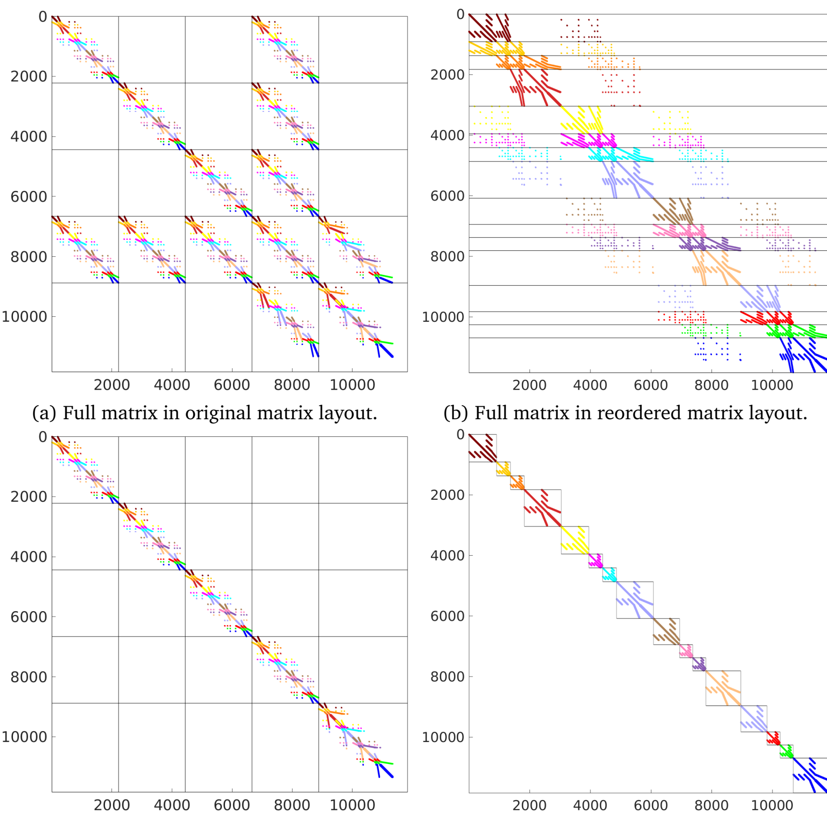
Figure 7.31: Partitioning of the mesh in the multidomain example into 16 subdomains.

How To Reproduce

The following commands run one timestep of the multidomain simulation with fat layer:

```
cd $OPENDIHU_HOME/examples/electrophysiology/multidomain/
  ↪ multidomain_with_fat/build_release
mpirun -n 4 ./multidomain_with_fat ../settings_multidomain_with_fat.
  ↪ py matrix.py
mpirun -n 16 ./multidomain_with_fat ../settings_multidomain_with_fat
  ↪ .py matrix.py
```

To inspect the system matrix, define a directory where the matrix should be stored. This can be done by setting the parameter `config["Solvers"]["multidomainLinearSolver"]["dumpFilename"]`, e.g., to `"out/matrix/m"`. Then, the directory `out/matrix` will contain MATLAB files with the system matrix. To create the plots, open MATLAB, load the system matrix from the respective file and open the script `display_matrix_entries.m`. Adjust the name of the matrix variable in the first code block, the run the desired steps of the Live Script to produce various plots.
The saved file contains the system matrix already in the reordered layout shown in Figures 7.28b and 7.32b. The MATLAB script reverses the permutation that was applied in OpenDiHu to generate the plots of Figures 7.28a and 7.32a.



(a) Full matrix in original matrix layout.

(b) Full matrix in reordered matrix layout.

(c) Block-diagonal submatrix in original matrix layout.

(d) Block-diagonal submatrix in reordered matrix layout.

Figure 7.32: Nonzero structure of the full system matrices in (a) and (b) and the symmetric preconditioner matrices in (c) and (d) of the multidomain problem, partitioned into 16 subdomains. The comparison with Fig. 7.28 reveals smaller relative diagonal band widths for the larger number of subdomains in this example.

7.6 Computation of CellML Models

In the following, we consider the computation of models that are given in CellML description, such as the subcellular model in the multi-scale muscle model.

The subcellular model is a system of DAEs that is solved at every node of the meshes in the discretized muscle. For the fiber based electrophysiology description, instances of the 0D subcellular model are computed on every node of every 1D fiber mesh. The 0D instances are coupled by the monodomain equation on every fiber. For the multidomain description, 0D model instances are solved at every node of the 3D muscle mesh for every compartment.

The subcellular model is provided as a CellML file and can be configured in the Python settings as described in Sec. 6.3. The class in OpenDiHu that computes CellML model instances for all nodes of a given mesh is the `CellMLAdapter`. It computes the expression G of the right-hand side of the ODE system, to obtain the vector of rates $\partial \mathbf{y}/\partial t = G(\mathbf{y})$ and the expression H for the algebraics $\mathbf{h} = H(\mathbf{y})$. The new state vector $\mathbf{y}$ is computed from the previous vector by a timestepping scheme, which uses the computed rates $\partial \mathbf{y}/\partial t$ as right-hand side. In the solver tree, the timestepping solver class has to be the parent node of the `CellMLAdapter`.

CellML models can be obtained as C source files, which can be compiled to shared libraries, loaded and accessed by the solver program. This approach is used in both OpenCMISS and OpenDiHu. The operation of computing multiple instances of a CellML model at once can be done more efficiently than in the naive way of repeatedly executing the model function, as done in OpenCMISS. To exploit the structure of computing multiple model instances together, dedicated C code has to be generated from the CellML model at runtime for a given number of model instances. In the following, we describe our code generation functionality for this purpose.

7.6.1 Data Flow for the Computation of CellML Models

Figure 7.33 shows the information flow for the CellML subsystem in OpenDiHu. On the left, a subcellular model is specified in CellML format in a file `model.cellml`. OpenDiHu uses the command line interface of OpenCOR to generate corresponding C code in the file `model.c`. The C code computes the functions $G(\mathbf{y})$ and $H(\mathbf{y})$ for the right-hand side and algebraics vector, respectively. A parser traverses the generated C source file and stores all instructions in an internal syntax tree data structure. The parser also determines

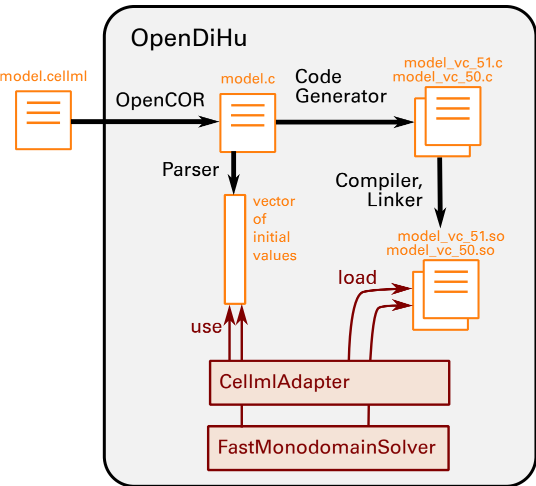


Figure 7.33: Processing of the given CellML model prior to solution. The CellML description is converted to C code using OpenCOR. The parser loads the C source code file and determines the contained initial values. Additionally, it parses the compute instructions into an internal syntax tree. The code generator produces optimized C code that can solve as many instances of the model as needed on every process according to the global partitioning of the domain. The generated C code is compiled, linked to a shared library and accessed from the solver code.

the initial values for the state vector $\mathbf{y}$ from the code that initializes the variables. Next, certain constants and algebraics in the compute instructions are replaced by parameter variables, as configured in the settings.

Then, a code generator outputs new C code that is optimized for a given number of CellML instances according to the number of nodes in the processes' subdomain within the global domain decomposition. This step is executed in parallel by different processes, but only once for every required number of model instances.

For example, if two fibers with 100 elements each are computed by 2×2 processes, the 101 nodes on each fiber are equally distributed to two different processes. As a result, each MPI rank has to compute either 51 or 50 CellML instances. Thus, the code generators on two of the ranks produce source code files for 51 and 50 model instances, named `model_vc_51.c` and `model_vc_50.c` in Fig. 7.33, respectively. After generation,

the source files are compiled and linked to a shared library, resulting in the shared object files `model_vc_51.so` and `model_vc_50.so` in Fig. 7.33.

The generation, compilation and linking steps are performed only by one process per source file. If a source file or shared library with the required name already exists from a previous run, the respective code generation and compilation steps are omitted. In the example, only two processes generate and compile the code. All four processes synchronize after all shared libraries have been generated, before proceeding to execute the computations.

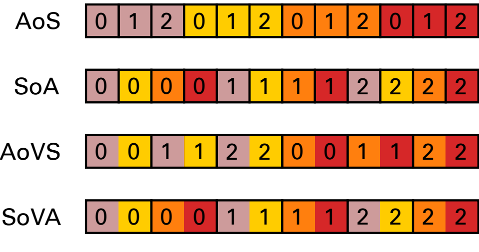
The generated shared libraries contain machine-code to compute $G(\mathbf{y})$ and $H(\mathbf{y})$. They are loaded into the simulation program and executed by the `CellmlAdapter` class with the corresponding values, as indicated in Fig. 7.33. Furthermore, the `CellmlAdapter` uses the previously inferred vector of initial values to initialize the state vector before the first timestep. Also, the `FastMonodomainSolver` class presented in Sec. 7.4.2, which efficiently solves the monodomain equation, makes use of the code generator and the shared libraries to evaluate the operators G and H of the subcellular model.

7.6.2 Optimizations in the Generated Code

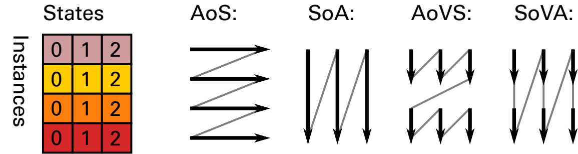
The code generator can be configured to employ various types of optimizations in the generated code. These optimizations can be selected in the settings by the parameter `"optimizationType"`.

The naive way to solve multiple CellML model instances leads to storing the state vectors in an Array-of-Struct (AoS) memory layout. The “struct” containing all components of the state vector for a single CellML model is stored at consecutive locations in memory and multiple structs for all computed instances are lined up next to each other. Figure 7.34a shows the AoS layout in the top row for four model instances given by different colors. Each instance contains the three state variables $0, 1$ and 2.

The transposed memory layout is Struct-of-Array (SoA), where the same state components for all model instances are close in memory. In the example in the second row of Fig. 7.34a, always four states of the same kind are stored contiguously. Figure 7.34b shows the construction schemes for the memory layouts. Comparing the scheme for SoA with AoS, it can be seen that the traversal in the 2D field of values is now vertical instead horizontal.



(a) Data in different memory layouts.



(b) Schemes how to construct the memory layouts. On the left, the entries are organized in a 2D field according to the state index and instance index. On the right, the traversal schemes for the different layouts are shown.

Figure 7.34: Different memory layouts for the CellML model: Array-of-Struct (AoS), Struct-of-Array (SoA), Array-of-Vectorized-Struct (AoVS), and Struct-of-Vectorized-Array (SoVA). The entries for four instances of the CellML model are shown by different colors, where each contains the three state variables 0,1 and 2.

Such a vertical layout is a prerequisite for employing single-instruction-multiple-data (SIMD) parallelism. SIMD instructions perform the same calculations on multiple components of SIMD vectors simultaneously. In the visualization of SoA in Fig. 7.34a, always four operands could be loaded simultaneously from memory to the vector registers in the CPU. Modern processors support the AVX2 instruction set with a SIMD lane width of $\mathcal{W}_T = 4$ double values or the AVX-512 instruction set with $\mathcal{W}_T = 8$ double values.

Figure 7.35 demonstrates the code generation and presents different approaches to efficiently evaluate the operators of a CellML model for multiple instances. Figure 7.35a shows the original code for a single model instance, which can be obtained from the CellML website or exported from a CellML model using OpenCOR. The listing shows the computation of the algebraic variable with index one and the rate with index one. The formulas typically use other states, algebraics and constant variables and consist of basic arithmetic such as additions, multiplications, potentiations to integer exponents

```
1 ALGEBRAIC[1] = ( - 0.100000*(STATES[0]+50.0000))/(exp(- (STATES[0]+50.0000)/10.0
2 RATES[1] =  ALGEBRAIC[1]*(1.00000 - STATES[1]) -  ALGEBRAIC[5]*STATES[1];
3 ...
```

(a) Original C code for one CellML model instance generated by OpenCOR.

```
1 #pragma omp for simd
2 for (int i = 0; i < 1481; i++)
3   algebraics[1481+i] = ( - 0.100000*(states[0+i]+50.0000))/(exp(- (states[0+i]+5
4
5 #pragma omp for simd
6 for (int i = 0; i < 1481; i++)
7   rates[1481+i] =  algebraics[1481+i]*(1.00000 - states[1481+i]) -  algebraics[7
8 ...
```

(b) Generated code for optimization type "simd".

```
1 // fill input vectors of states and parameters
2 for (int stateNo = 0; stateNo < nStates; stateNo++)
3   for (int i = 0; i < nVcVectors; i++)  // Vc vector no
4     for (int k = 0; k < W_T; k++)  // entry no in Vc vector
5       statesVc[i*nStates + stateNo][k] = states[stateNo*nInstances + i*W_T+k];
6    // statesVc[stateNo*nVcVectors + i][k] = states[stateNo*nInstances + i*W_T+k]
7
8 for (int i = 0; i < nVcVectors; i++)
9 {
10   algebraicsVc[i*nAlgebraics + 1] = ( - 0.100000*(statesVc[i*nStates + 0]+50.000
11 //algebraicsVc[371+i] = ( - 0.100000*(statesVc[0+i]+50.0000))/(exponential(- (st
12   ...
13 }
```

(c) Generated code for optimization type "vc".

```
1 #pragma omp parallel for
2 for (int i = 0; i < 1481; i++)
3 {
4   algebraics[1481+i] = ( - 0.100000*(states[0+i]+50.0000))/(exp(- (states[0+i]+5
5   rates[1481+i] =  algebraics[1481+i]*(1.00000 - states[1481+i]) -  algebraics[7
6   ...
7 }
```

(d) Generated code for optimization type "openmp".

Figure 7.35: Output of the CellML code generator in OpenDiHu for 1481 model instances and different optimization types. The model is the subcellular model of Hodgkin and Huxley, and the code shows only two formulas of this model. Furthermore, the lines are truncated.

and exponential functions. Some models such as the subcellular model of Shorten et al. [Sho07] also involve piecewise definitions that include “inline if” branching operations.

Calling the code in Fig. 7.35a for multiple model instances is associated with the AoS memory layout. An improvement is the generated code with optimization type `"simd"` in Fig. 7.35b, which assumes the data to be organized in the SoA memory layout. The code is generated specifically to solved 1481 instances of the model. The array indexing for the `algebraics` and `rates` variables sums the constant offset according to the memory layout and the number of the model instance. For example, for the second algebraic (with former index 1), the offset is 1481 because so many memory locations are filled with values of the first algebraic (with former index 0).

Furthermore, every formula is enclosed in a loop over all 1481 instances of the model. The loops have OpenMP pragmas that instruct the compiler to use SIMD instructions for the loop body, if possible. Because of the consecutive storage, $\mathcal{W}_T$ loop iterations can be combined into a single computation using vector instructions. For the remainder iterations at the end of the loop, the compiler automatically adds different instructions with corresponding smaller SIMD vector lengths.

The approach of using OpenMP pragmas has the advantage that it is independent of the actual hardware capabilities and does not fix the SIMD vector size $\mathcal{W}_T$. If vectorization is disabled at compile-time, sequential CPU code is generated and the same valid solution is computed. A disadvantage is that the performance of the generated code depends on the vectorization ability of the compiler and its detection that the variables have the proper memory layout. For some constructs such as exponential functions or branching instructions, no vectorization is employed and the particular loop falls back to serial code. Such behavior is observed when inspecting the vectorization reports, which are emitted by the compiler.

Thus, we implement another optimization type `"vc"` in the code generator that guarantees usage of vector instructions for all formulas. We use the C++ library *Vc*, which provides a wrapper to hardware-specific vector instructions and abstracts the SIMD lane width [Kre12; Kre15]. Using the data types of this library also allows writing hardware independent code and to achieve performance portability, like with the `"simd"` optimization type. As *Vc* only supports vectorization up to the AVX2 instruction set, we also use the `std::experimental::simd` specification, which is currently considered by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) for inclusion in the C++ standard library [Hob19]. Switching be-

tween these two libraries is transparent in the code and depends on whether the compiler supports C++17.

Similar to the `"simd"` optimization type, the `"vc"` optimization type also uses a memory layout where consecutive memory entries correspond to different instances of the model, and the traversal direction in Fig. 7.34b is vertical for at least $\mathcal{W}_T$ entries. Figure 7.34 shows two such memory layouts for a SIMD vector length of $\mathcal{W}_T = 2$: Array-of-Vectorized-Struct (AoVS) and Struct-of-Vectorized-Array (SoVA). Both are implemented in the code generator.

The SoVA memory layout is very similar to SoA, the only difference is, that, in SoVA, entries are always accessed in multiples of the SIMD vector length $\mathcal{W}_T$. The advantage of SoVA is that the array indices are given by the sum of a constant offset with the loop index, whereas, with the AoVS layout, a multiplication is required for every access.

AoVS resembles more the AoS layout. Its advantage over SoVA is that the complete state vector $\mathbf{y}$ for any model instance is located more locally in memory. As the total computation iterates over model instances, the accessed memory is more coherent than for the same iteration scheme with the AoVS layout. This possibly leads to more cache hits, however, for set-associative caches, the effect is reduced. Due to the complexity of today's cache architectures, only measurements can decide which of the two memory layouts leads to a faster execution. Our measurements show that the SoVA layout leads to 2 % shorter runtimes than the AoVS memory layout and, thus, is the preferred choice.

Figure 7.35c shows the resulting code using the AoVS memory layout. The commented lines 6 and 11 show the corresponding code for the SoVA memory layout. At the beginning of the generated program code, the given data in the `states` variable are copied to the `statesVc` variable in the new memory layout. Nested loops over all states, over the SIMD vectors and over the scalar values within the SIMD vector are used for this operation. For comparison, lines 5 and 6 show the corresponding indexing of the `statesVc` variables for the AoVS and SoVA memory layouts, respectively.

For the evaluation of the model operators, we iterate over the number `nVcVectors` of SIMD vectors instead of the number of model instances as for `"simd"`. In the example with 1481 instances, we have `nVcVectors=`$\lceil 1481/\mathcal{W}_T \rceil$`=371` SIMD vectors for $\mathcal{W}_T = 4$. Accordingly, the offsets for indexing the variables in the SoVA layout are smaller, e.g., in line 11, the offset for indexing the `algebraicsVc` variables is 371 instead of 1481 for the non-vectorized variable in the previously considered `"simd"` code. Comparing the statements for AoVS and SoVA in lines 10 and 11, it can be seen that the AoVS memory layout involves an additional multiplication during the indexing of the array.

In case of branching instructions in the CellML formulas, the Vc library provides an implementation of the “inline if” statement for SIMD vectors, which checks the condition, potentially executes both branches and merges the components from the active branches into the resulting SIMD vector.

Profiling the execution of the "vc" code for different subcellular models shows that about half of the runtime is spent in evaluating the exponential function. Therefore, we use the following approximation:

$$\exp(x) \approx \exp^*(x) = \left(1 + \frac{x}{n}\right)^n. \tag{7.7}$$

The series converges to the exact value for $n \to \infty$. We choose $n = 1024$ and are able to compute the approximate value by only one addition and 11 multiplications using the following formula:

$$\exp^*(x) = \left(1 + \frac{x}{1024}\right)^{2^{10}} = \left(\cdots\left(\left(\left(1 + \frac{x}{1024}\right)^2\right)^2\right)^{\cdot^{\cdot^{\cdot}}}\right)^2.$$

In the subcellular models of Hodgkin and Huxley [Hod52a] and Shorten et al. [Sho07], the values for x are bounded by $|x| < x_{\max} = 12$, and we get a relative error of the approximation of $|(\exp^* - \exp)(x_{\max})/\exp(x_{\max})| < 0.07$. This approximation can be enabled or disabled in the code generation.

Another optimization is implemented for exponentiation a^b. In the considered CellML models, only integer exponents $b \in \mathbb{Z}$ occur. We add a recursive implementation of the power function that requires a logarithmic number of multiplications.

The code generator with the "vc" optimization type is also used by the `FastMonodomainSolver` class described in Sec. 7.4.2. The generated codes for the `FastMonodomainSolver` class additionally contain the Heun scheme to solve the model, integrate code for the stimulation of muscle fibers and export certain algebraic values that were declared as parameters in the settings.

Another possiblity to improve the performance besides instruction-level parallelism is thread-level parallelism. The "openmp" optimization type generates code containing OpenMP pragmas that distribute the computations to multiple OpenMP threads with shared memory. Figure 7.35d shows the generated code for this optimization type. A loop iterates over all model instances and the variables are stored in SoA memory layout. The loop iterations are independent of each other as they correspond to different instances

```
1  #pragma omp target parallel for \
2                     map(to:states,t,parameters) map(from:rates,algebraics)
3  for (int i = 0; i < 1481; i++)
4  {
5    algebraics[1481+i] = ( - 0.100000*(states[0+i]+50.0000))/(exp(- (states[0+i]+5
6    rates[1481+i] =  algebraics[1481+i]*(1.00000 - states[1481+i]) -  algebraics[7
7    ...
8  }
```

Figure 7.36: Generated code for optimization type "gpu" corresponding to the scenario in Fig. 7.35.

of the CellML model. OpenMP distributes the workload to a predefined number of threads that can be specified by environment variables.

7.6.3 Code Generation for GPUs

Besides instruction-level and thread-level parallelism, which were discussed in the last section, accelerator hardware such as GPUs can be considered to reduce the runtime of solving a CellML model. Our code generator features the "gpu" optimization type to generate code that is called on the CPU and then offloads the main computations to a GPU.

We use OpenMP 4.5 to instrument the generated code for device offloading. At the time of writing, only an experimental version of GCC 11 is fully capable of compiling this code. In our studies, the code is compiled for the *nvptx* target, which generates and compiles device-specific CUDA code using the NVIDIA parallel thread execution (PTX) instruction set architecture. We successfully run the computation on various NVIDIA GPUs, including a GeForce RTX 3080. However, the approach is device-agnostic and other accelerator hardware can also be used.

Figure 7.36 shows an excerpt of the generated code. It resembles the code of the "openmp" optimization type, except that the OpenMP pragma in lines 1 and 2 is different. The lines specify the variables to be mapped to and from the target device: The vectors of states and parameters as well as the current simulation time `t` are sent to the GPU and, after computation, the rates and algebraics are transferred back to the CPU.

Using the `CellmlAdapter`, it is, thus, possible to run any CellML model on the GPU. However, for the fiber based electrophysiology model uploading and downloading the

data of all model instances between CPU to GPU in every timestep is clearly not the most efficient way to utilize the GPU. Therefore, we add efficient GPU integration with proper memory management to the `FastMonodomainSolver` class, which is specialized to solve the monodomain equation for multiple fibers. The class allows computing multiple timesteps in series on the GPU between subsequent points of synchronization with the CPU. This synchronization is only required, e.g., for writing output files or coupling to a solid mechanics solver.

The generated GPU source code for the `FastMonodomainSolver` contains the full algorithm for solving multiple timesteps of the electrophysiology model for multiple fibers with a given number of nodes each. The Strang splitting scheme is used, which solves the 0D subcellular part and the 1D electric conduction part in the scheme 0D-1D-0D. The 0D part is solved by the Heun scheme. The 1D part is computed either with the implicit Euler method or the Crank-Nicolson method. The linear system of equations is solved using the linear complexity Thomas algorithm.

The parallelization on the GPU uses a fixed number of thread teams, where all threads in a team execute the same code. For the 0D problem, the iterations of the two nested loops over fibers and model instances per fiber are distributed to all thread teams, such that the iterations are *workshared*. Thus, the 0D subcellular models are computed concurrently for all instances. Between the computations of the 0D and 1D parts, synchronization occurs as the data on all instances on a fiber are accessed in the solution of the 1D problem. The 1D computations are distributed on the fiber level, before the second 0D computation in the Strang splitting is again distributed on the model instance level. Another synchronization occurs after each timestep of the whole Strang splitting.

The data transfer in both directions between CPU and GPU is reduced to a minimum. Initially, all required parameters and initial values have to be transferred to GPU memory. The initial state vector $\mathbf{y}$ is only sent once to the GPU and all model instances of all fibers get initialized to these same values. Further data to be sent includes parameters that describe the stimulation times as presented in Sec. 6.3.4, locations of the neuromuscular junction and the distribution of fibers to motor units. Instead of the callback functions described in Sec. 6.3.4, the stimulation times can be altered by an input file. For details, we refer to the online documentation [Mai21c].

During computation, smaller amounts of data are transferred before and after each set of consecutive timesteps on the GPU. The data to be sent to the GPU before the computations consist of the CellML parameter values and the lengths of all elements in the 1D mesh, which change, if muscle contraction is computed on the CPU. The data to

be transferred back to the CPU after the computations on the GPU consist of a subset of the state vector for every model instance. This subset contains only those components of $\mathbf{y}$ that should be written to an output file on the CPU or are required for coupling to another solver. Thus, the majority of the data stay on the GPU.

7.7 Solid Mechanics Solver

Next, we discuss details on the solver of the solid mechanics models, which is needed for the muscle contraction part of the multi-scale model, described in Sec. 5.2.

Section 7.7.1 gives details on the solver for the linear solid mechanics model. Section 7.7.2 addresses the nonlinear model and describes how the material model is specified. Section 7.7.3 presents the timestepping method for the dynamic problem and describes the implemented measures to improve the convergence.

7.7.1 Solver for the Linear Model

As noted in Sec. 6.2.7, the `QuasiStaticLinearElasticitySolver` class can be used to solve the linearized solid mechanics model described in Sec. 5.2.5 and discretized in Sec. 5.4.1. Within this solver class, the matrix equation Eq. (5.76) is assembled and solved by an object of the `FiniteElementMethod` class, which is the same class that is used to solve Laplace problems.

If the solver is explicitly coupled with an electrophysiology model, we obtain a quasi-static formulation of muscle contraction. The activation parameter $\bar{\gamma}$ on the 3D mesh is transferred from the electrophysiology model to the elasticity model. Then, the linear system of equations of the elasticity model is solved using the new muscle activation values in the right-hand side. The system matrix stays constant in all timesteps. After the new displacements have been computed, the geometries of the 3D mesh and the embedded 1D fiber meshes are updated accordingly.

In this scenario, the active stress tensor $\boldsymbol{\sigma}^{\text{active}}$ in Eq. (5.78) is computed as the product of the activation parameter $\bar{\gamma}$ with a scalar maximum active stress parameter $\sigma_{\text{max,active}}$ and an anisotropy tensor $\mathbf{a}$:

$$\boldsymbol{\sigma}^{\text{active}} = \sigma_{\text{max,active}}\,\bar{\gamma}\,\mathbf{a}. \tag{7.8}$$

The tensor **a** can be specified in the Python settings by a 3×3 matrix and allows to specify the anisotropic active behavior of the muscle tissue. In this specification, the first unit vector $\mathbf{e}_1 = (1,0,0)^\top$ designates the fiber direction, $\mathbf{e}_2$ and $\mathbf{e}_3$ specify the transverse direction. Prior to the computation in Eq. (7.8), the basis of the given matrix is changed, such that $\mathbf{e}_1$ in the old basis maps to the fiber direction in the new basis and the new basis is orthonormal. This change of basis is performed at every point in the muscle with the respective fiber direction. Thus, it is possible to specify transversely isotropic material behavior with contraction in fiber direction.

7.7.2 Specification of Nonlinear Material Models

To compute the nonlinear model, the `HyperelasticitySolver` class is used for the static formulation of a passive material, the `DynamicHyperelasticitySolver` class is used for the dynamic passive behavior, and the `MuscleContractionSolver` is used for either the static or the dynamic model with active stress contribution.

These solver classes can be coupled to the electrophysiology model in the same way as described in Sec. 7.7.1. Similar to Sec. 7.7.1, the `MuscleContractionSolver` adds an active stress term to the formulation according to the formula in Eq. (5.42). The force-length relation $f_\ell(\lambda_f)$ can either be added by the `MuscleContractionSolver` or specified in the CellML description as part of the subcellular model for the activation parameter γ.

To specify the passive material behavior, the strain energy function Ψ has to be defined. This definition has to be available at compile-time and is specified in the C++ code.

Four different terms can be defined to describe the material model in different forms such as the coupled or decoupled representation. The four terms are introduced in Sec. 5.2.6 and given in Eq. (5.39) as follows:

$$\Psi = \Psi_{\text{vol}}(J) + \Psi_{\text{iso}}(\bar{I}_1, \bar{I}_2, \bar{I}_4, \bar{I}_5) + \Psi_1(I_1, I_2, I_3) + \Psi_2(\mathbf{C}, \mathbf{a}_0). \tag{7.9}$$

Formulas for these terms can be specified using C++ expressions with a syntax specified by the *SEMT* library [Gut12; Gut04] (and also described in the online documentation of OpenDiHu [Mai21c]). Mathematical functions such as power and log functions are available, intermediate variables can be defined and reused, and constants for material parameters can be used, whose values can be specified in the Python settings.

The implementation uses the SEMT library to symbolically differentiate the given terms with respect to their function arguments. Thus, all values used in the Newton solver including the Jacobian matrix can be computed automatically. Using this technology, OpenDiHu provides the flexibility to add new material models at compile-time without the need for manual differentiation.

Additionally, three options, which alter the computation and efficiency, have to be set in the C++ description: The first option specifies, whether the material is considered incompressible. If this option is set to true, the solution approach with Lagrange multiplier p is used, otherwise the unknowns only contain the displacements $\mathbf{u}$ and possibly the velocities $\mathbf{v}$. The second option specifies, if the active stress term $\mathbf{S}^{\text{active}}$ should be added to the material. This option is only relevant for the `MuscleContractionSolver` class, disabling it allows computing passive tissue.

The third option determines, if the fiber direction $\mathbf{a}_0$ appears in the description of the material model. Only if this option is enabled, the corresponding invariants I_4 and I_5 are available for the definition of the Ψ_{iso} term in Eq. (7.9). If disabled, all terms in the formulas in Sec. 5.4.4 that involve $\mathbf{a}_0$ are left out of the computation, which speeds up the computations in the solver.

7.7.3 Convergence Improvements for the Nonlinear Solver

The nonlinear equation is solved using the Scalable Nonlinear Equations Solvers (SNES) component of PETSc, which provides Newton-type and quasi-Newton methods for solving systems of nonlinear equations. The method to use and other parameters such as the line-search type can be configured in the Python settings file.

Fast convergence of a Newton-based nonlinear solver is facilitated with a good initial guess for the vector of unknowns. Therefore, we predict the solution functions $\mathbf{u}$ and $\mathbf{v}$ for the next timestep in a dynamic problem using the following computations:

$$\mathbf{u}^{(i+1),\text{predicted}} = \mathbf{u}^{(i)} + dt\,\mathbf{v}^{(i)}, \quad \mathbf{a}^{(i)} = \frac{1}{dt}(\mathbf{v}^{(i)} - \mathbf{v}^{(i-1)}), \quad \mathbf{v}^{(i+1),\text{predicted}} = \mathbf{v}^{(i)} + dt\,\mathbf{a}^{(i)}.$$

The predicted displacements $\mathbf{u}^{(i+1),\text{predicted}}$ for the next timestep $(i+1)$ are estimated by a forward Euler scheme from the displacements $\mathbf{u}^{(i)}$ and velocities $\mathbf{v}^{(i)}$ of the current timestep i. The current acceleration $\mathbf{a}^{(i)}$ is estimated by finite differences from the current and previous velocities, $\mathbf{v}^{(i)}$ and $\mathbf{v}^{(i-1)}$. The predicted velocities $\mathbf{v}^{(i+1),\text{predicted}}$ for the next timestep again use a forward Euler method with the estimated acceleration

values $\mathbf{a}^{(i)}$. Using the initial guess $(\mathbf{u}^{(i+1),\text{predicted}}, \mathbf{v}^{(i+1),\text{predicted}}, p^{(i)})^\top$, the solution vector $(\mathbf{u}^{(i+1)}, \mathbf{v}^{(i+1)}, p^{(i+1)})^\top$ for the next timestep can be obtained by the nonlinear system solver.

Independently of the predictions of initial values from previous timesteps, the convergence of the nonlinear solver within a timestep can be improved by employing load stepping. This approach involves solving $N > 1$ sub problems with increasing load steps. In each step i, the problem is solved with the right-hand side $\mathbf{f}_i = \alpha_i\,\mathbf{f}$, scaled by the load factor $\alpha_i \in [0,1]$. The obtained solution in iteration i is used as the initial guess for the subsequent load step $(i+1)$. Increasing values of α_i are used until the final solution is found for $\alpha_N = 1$. Typical load factors are $(\alpha_i)_{i=1,\dots,N} = (b^{-(N-1)}, b^{-(N-2)}, \dots, b^0)$ for a basis $b > 0$.

The list of load factors can be specified in the settings. If the nonlinear solver diverges or fails because an unphysical negative determinant J of the deformation gradient occurs, the current load factor is automatically reduced and the solution processes is started again, using the last valid solution as initial guess. If the last successful solution was found for load factor α_i and the current load factor α_{i+1} fails, a new load factor $\alpha^*_{i+1} = (\alpha_i + \alpha_{i+1})/2$ is inserted in the list of load factors between α_i and α_{i+1} and the solution of the nonlinear problem with this new factor is attempted.

In case of a poorly conditioned problem, it can happen that no more solution can be found, regardless of how far the load factor gets decreased. If the difference between two load factors falls below a configurable threshold, the nonlinear solution process for the current timestep is aborted.

Practical tests with the dynamic incompressible problem have shown that the convergence sometimes degrades only for a single timestep and returns to normal in the next timestep. Thus, we allow a single timestep i to diverge and, in this case, continue with the next timestep $(i+1)$ using the (diverged) solution with the lowest residual norm from timestep i to predict the initial guess for timestep $(i+1)$.

7.8 Data Mapping Between Meshes

After the implementation of various solvers for specific parts of the multi-domain model has been described in the previous sections, we now focus on the data mapping between different meshes that occurs in the coupling schemes between the execution of the coupled solvers.

Data mapping between meshes is required in scenarios that involve both a finely resolved 3D mesh for the electrophysiology model and a coarse 3D mesh for the solid mechanics model. Moreover, data are mapped between the 3D muscle mesh and the embedded 1D fiber meshes in the fiber based electrophysiology model. In these two cases, the mapping has to be carried out in both directions between the involved meshes. The operation can be characterized as *volume mapping*.

We implement a generic mapping scheme between two meshes of any dimensionality and with any relative orientation with respect to each other. Given is a finite element interpolant on a *source* mesh, defined by the dof values at the nodes. The goal is to set the dof values of the *target* mesh, such that the error between the finite element representations on the common domain of source and target mesh is as low as possible. By using the ansatz functions of the target mesh, the constructed mapping has the same order of accuracy as the target mesh interpolant. For a linear target mesh, the mapping operation is second order accurate, for a quadratic target mesh, the mapping operation is third order accurate.

The considered source and target meshes are possibly partitioned. In order to perform the data mapping directly between two such meshes without communication between the processes, the partitioning of the volumes would have to be identical. However, this is not practical for different meshes and would disallow different orientations of source and target meshes. The only way to allow such a mapping is to consider either the source or the target mesh as a point cloud and construct the mapping between individual points and a mesh.

Considering the case of mapping the activation parameter value γ, which is stored on multiple 1D fibers, to the value $\bar{\gamma}$ on the 3D muscle mesh, it is natural to consider the source mesh as a point cloud. Then, instead of multiple fibers, we have a set of points, where γ is known. This set of source points is mapped to the enclosing 3D target mesh. On every process, the source points have to be located inside the local subdomain of the target mesh.

The reverse mapping in this example is also required: The geometry of the 3D muscle mesh has to be mapped to the fibers points, such that a deformation of the muscle also affects the embedded fibers. This reverse mapping from the target mesh to the source fiber meshes or points is trivial: The source values can be interpolated in the target mesh using the finite element discretization. We construct the mapping from source to target mesh to be the transpose operation to this interpolation. In the following section, we introduce the method with a graphical example.

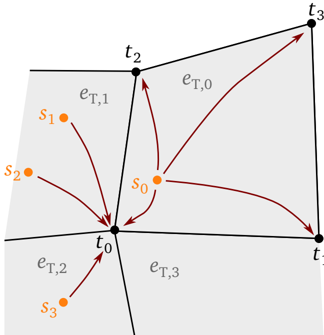


Figure 7.37: Data mapping scheme from source points (orange) to a target mesh (black).

7.8.1 Construction of the Parallel Data Mapping

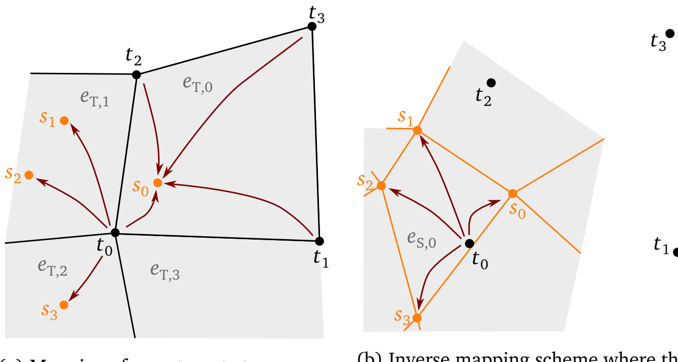
Figure 7.37 shows a scenario, where data mapping is performed from a source mesh to a target mesh. The source mesh is given by the orange points s_0 to s_3. The target mesh is visualized by the black and gray elements $e_{T,0}$ to $e_{T,3}$ and nodes t_0 to t_3.

The value at s_0 contributes to all nodes t_0 to t_3 of the target element $e_{T,0}$ as indicated by the red arrows. The relations between the contributions to t_0, t_1, t_2 and t_3 are determined by the values of the respective target element ansatz functions ϕ_0 to ϕ_3, evaluated at the location of s_0. Similarly, the source points s_1 to s_3 contribute to the nodes of their enclosing target elements $e_{T,1}$ and $e_{T,2}$.

As a consequence, all shown source points s_0 to s_3 influence the value of the target node t_0, as indicated by the red arrows. The value $\hat{t}_0$ at point t_0 is computed using the values $\hat{s}_i$ at the points s_i for $i = 1, \dots, 4$ as follows:

$$\hat{t}_0 = \sum_{i=0}^{4} \alpha_i \, \hat{s}_i, \quad \text{with } \alpha_i = \frac{\phi_{t_0}(\xi_{s_i})}{\sum_{i=0}^{4} \phi_{t_0}(\xi_{s_i})}. \tag{7.10}$$

Here, ϕ_{t_0} is the finite element ansatz function for the node t_0. It is evaluated at the locations ξ_{s_i} of the source points s_i in the respective target elements. The factors α_i specify the fractions, with which the different contributions to $\hat{t}_0$ are scaled. Their construction ensures the property $\sum_{i=1}^{4} \alpha_i = 1$. Note that the number of summands in the sum over the source points can be different from 4 for other target points.



(a) Mapping from target to source points using the reverse scheme of Fig. 7.37.

(b) Inverse mapping scheme where the roles of source and target mesh are swapped.

Figure 7.38: Data mapping from target mesh to source mesh.

The reverse mapping from target nodes to source points uses the same data dependencies between dofs in the source and in the target meshes. Figure 7.38a shows the scheme for the reverse mapping. It is the same as in Fig. 7.37, except that the direction of the arrows has been flipped. As noted earlier, the mapping from the target to the source mesh is simply an interpolation in the target mesh. The value $\hat{s}_0$ is computed from $\hat{t}_0$ to $\hat{t}_3$ using the finite element interpolation formula in element $e_{T,0}$:

$$\hat{s}_0 = \sum_{i=0}^{4} \hat{t}_i \, \phi_{t_i}(\xi_{s_0}).$$

Again, the contribution factors sum up to one, $\sum_{i=0}^{4} \phi_{t_i}(\xi_{s_0}) = 1$.

This mapping scheme has the advantage that it requires no communication between the involved processes to determine the target dofs, to which a source dof contributes to. Considering the example in Fig. 7.37 and assuming that the four target elements are located on four different subdomains, it can be seen that each source point s_i only has to access the target element, where it is contained, to determine the respective element coordinates ξ_{s_i}.

For the computation of the factors α_i in Eq. (7.10) and for the computation of the target dofs, communication is required. This communication step is the same exchange of ghost dof values, which is also needed for the assembly of finite element stiffness and mass matrices. In the implementation, it is available by the respective functionality of PETSc

as described in Sec. 7.1.1. The reverse mapping from target to source meshes, i.e., the interpolation scheme, works without any communication as all required data are local to the processes.

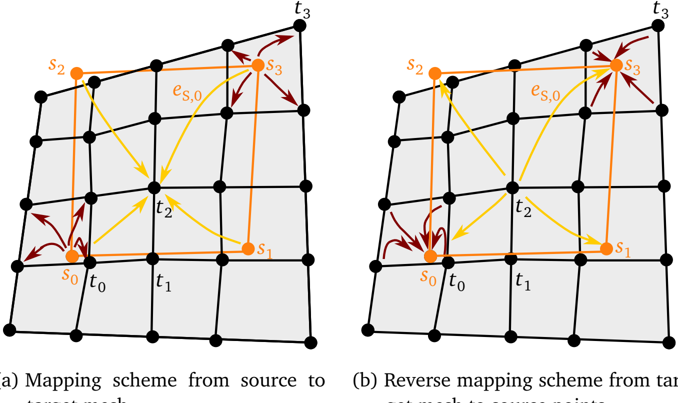
Instead of reversing the source to target mapping as described, it is often also possible to change the roles of source and target mesh and construct a new mapping in this way. This is only possible, if the two meshes have the same dimensionality, as in the considered example visualizations with two 2D meshes. Figure 7.38b shows the presented mapping scheme with the roles of source and target meshes reversed. By comparing with Fig. 7.38a, it can be seen that, in this case, the node t_0 contributes to the same nodes s_0 to s_3 in both approaches. However, the contribution factors are different. In general, the dependent nodes in both meshes are not necessarily the same in the two mapping directions. This means that, in general, the reversed or transposed mapping is not equal to the inverse mapping that is created by interchanging source and target meshes.

For mappings between different dimensionalities, only the approach of reversing the mapping in one direction is possible. For example, mapping from a 1D mesh to a 3D mesh allows no interpolation in the 1D mesh to get the 3D mesh data, as the 1D mesh occupies only a subset of the domain of the 3D mesh. This is a reason for implementing the presented mapping scheme, where the mapping direction can be reversed. Another advantage is that, once the mapping is constructed, both mapping directions are available and the expensive operation of locating the points of one mesh inside the elements of the other mesh has only be performed once.

7.8.2 Special Treatment of Coarse Meshes

While the mapping error in the described scheme converges to zero, when the mesh widths approach zero, an issue occurs, if one of the meshes is significantly coarser than the other. Figure 7.39a depicts the case of a coarse source mesh in orange color that is mapped to a finer target mesh in black and gray colors. According to the presented scheme, the source points s_0 and s_3 contribute to target nodes as visualized by the red arrows. The analog contributions for s_1 and s_2 are not shown in Fig. 7.39a. Some target mesh nodes in this example have large distances to the source points and, as a result, do not get contributions from any source point. For example, this is the case for the target points t_1 and t_2.

To define the value at t_2 depending on the source data, we add new contributions from all nodes of the source element in which t_2 is located. These contributions are visualized



(a) Mapping scheme from source to target mesh.

(b) Reverse mapping scheme from target mesh to source points.

Figure 7.39: Data mapping scheme with additional data dependencies for a coarse source mesh.

by the yellow arrows in Fig. 7.39a. The contributions use the ansatz functions of the source element, and the operation is equivalent to interpolating the value for t_2 in the source mesh:

$$\hat{t}_2 = \sum_{i=0}^{4} \hat{s}_i \, \phi_{s_i}(\xi_{t_2}).$$

The location ξ_{t_2} of t_2 in element coordinates of the source element is required for this computation. Analogously, corresponding contributions are added for the other target nodes that do not yet get any contribution from the source data.

These additional contributions are also present in the reversed mapping scheme from the target to the source mesh. As can be seen in Fig. 7.39b, the value at t_2 contributes to the source nodes s_0 to s_3. At any source node, the number of contributions increases accordingly. The visualization in Fig. 7.39b shows five incoming arrows with contributions for s_0 and s_3. The actual number is higher, since not all target nodes with additional contributions are visualized. At the target nodes, the contribution factors get rescaled, such that they add up to 1 and their relations are preserved.

7.8.3 Computation of Element Coordinates For Mapped Points

During the setup of the mapping between the source mesh and the target mesh, we need to find, for every source point s_i, the target element $e_{T,j}$ that contains s_i. Furthermore, we need to determine the local element coordinates $\mathbf{x}_{s_i} = (\xi_1, \xi_2, \xi_3)^\top$ of the point in this element. To check, if the point is inside a particular element, we compute its coordinates in the element coordinate system. If the coordinates ξ are inside the range of $[0,1]^d$, the point is considered inside this element, and the coordinates are determined.

The source point is given by coordinates $\mathbf{x} = (x_1, x_2, x_3)^\top \in \mathbb{R}^3$ in the world coordinate frame. The point is related to the d-dimensional element coordinate frame $(\xi_1, \dots, \xi_d)$ of its containing target element by the following map:

$$\mathbf{x}(\xi) = \sum_{i=1}^{n_{\text{dofs}}} \phi_i(\xi)\,\mathbf{x}^i. \tag{7.11}$$

Here, ϕ_i for $i = 1, \dots, n_{\text{dofs}}$, are the nodal ansatz functions of the target element. The element has n_{dofs} dofs and is given by its node positions $\mathbf{x}^i$.

The computation of the element coordinates ξ from the world coordinates $\mathbf{x}$ consists of inverting the mapping in Eq. (7.11). In the following, several approaches are presented to perform this inversion for different mesh types.

For meshes of type `StructuredRegularFixedOfDimension<D>`, the inversion can be performed analytically. The mesh is a Cartesian grid with a fixed mesh width h. For quadratic elements, h denotes the side length of an element, not the distance between adjacent nodes. The computation of the element coordinates ξ for the point $\mathbf{x}$ uses the position $\mathbf{x}^1$ of the first node and is given by:

$$\xi = (\mathbf{x} - \mathbf{x}^1)/h.$$

This formula is also used for 1D meshes of any type.

For non-Cartesian 2D meshes with linear ansatz functions, i.e., meshes of type `StructuredDeformableOfDimension<2>`, the inversion of the mapping from element to world coordinate frame in Eq. (7.11) can also be done analytically. We consider this problem in a generic way, where both the point $\mathbf{x}$ and the nodes $\mathbf{x}^1$ to $\mathbf{x}^4$ of the 2D element are embedded in 3D space, $\mathbf{x}, \mathbf{x}^1, \dots, \mathbf{x}^4 \in \mathbb{R}^3$. The computation determines the element coordinates (ξ_1, ξ_2) of the projection of $\mathbf{x}$ onto the plane of the triangle $(\mathbf{x}^1, \mathbf{x}^2, \mathbf{x}^3)$.

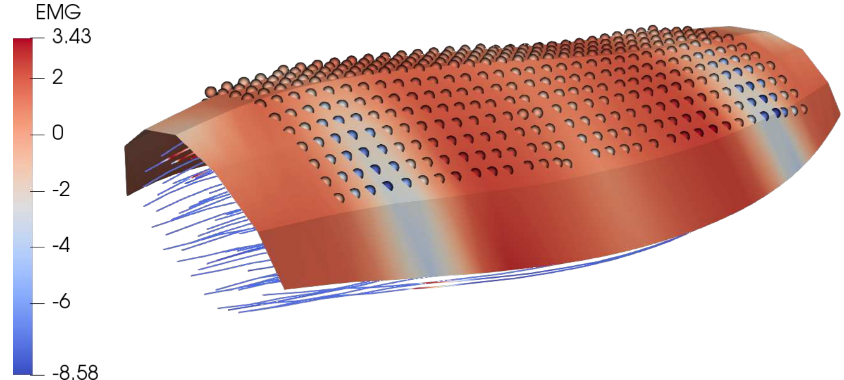


Figure 7.40: Simulation of surface EMG using the fiber based electrophysiology model with body fat layer. Only the top surface of the fat layer mesh is shown. The spheres correspond to the position of surface electrodes that are used to sample the simulation result in a spatial grid.

This functionality is used for specifying electrodes on the skin surface. The electrode positions are specified as a 2D grid in 3D space above the muscle. The mapping automatically projects the points of this grid onto the surface of the 3D mesh. Figure 7.40 shows such a use case. A simulation of surface EMG is shown, the coloring corresponds to the potential ϕ_b in millivolts in the body domain. A grid of electrode points, visualized by spheres, is mapped onto the surface of the muscle mesh and simulates electrode patches that capture high density surface EMG.

After calculating the mentioned projection on the 2D plane, the computation has to invert the map in Eq. (7.11). The ansatz functions ϕ_i are bilinear in the coordinates ξ_1 and ξ_2. This quadratic equation has two solutions for the unknown coordinates $\boldsymbol{\xi}$. The formulas for those solutions have been determined using the symbolic mathematics toolbox *SymPy* [Meu17] and the solution, where the point is inside the element or closer to its center is chosen.

For generic hexahedral 3D meshes, Eq. (7.11) is a cubic equation in $\boldsymbol{\xi}$, and the analytic inversion is not feasible. However, for simplex elements, i.e., tetrahedra given by points $\mathbf{x}^1$ to $\mathbf{x}^4$, it is possible. The ansatz in this case is given by:

$$\mathbf{x} = (1 - \xi_1 - \xi_2 - \xi_3)\,\mathbf{x}^1 + \xi_1\,\mathbf{x}^2 + \xi_2\,\mathbf{x}^3 + \xi_3\,\mathbf{x}^4.$$

This can be reformulated as:

$$\mathbf{x} - \mathbf{x}^1 = (\mathbf{x}^2 - \mathbf{x}^1)\,\xi_1 + (\mathbf{x}^3 - \mathbf{x}^1)\,\xi_2 + (\mathbf{x}^4 - \mathbf{x}^1)\,\xi_3. \tag{7.12}$$

This linear system of three equations can be solved for the three unknowns ξ_1, ξ_2 and ξ_3.

To invert the mapping for hexahedral elements, we proceed as follows. A hexahedral element can be subdivided into five simplex elements. Four outer simplex elements share their faces with parts of the hexahedral's surface. One interior simplex element only touches the hexahedron surface by its edges.

In each of the four outer simplex elements, we define a coordinate system (ξ_1, ξ_2, ξ_3) with the origin located at a corner of the hexahedron. In these elements, the coordinates $\boldsymbol{\xi}$ for the point $\mathbf{x}$ can be computed using the ansatz in Eq. (7.12). The computed coordinate values can be transformed to the hexahedral coordinate system by applying the appropriate mirror operations $\xi \mapsto (1-\xi)$ on some coordinates. Using the average values of the hexahedral coordinates resulting from all four outer simplex elements gives a good approximation for the correct hexahedral element coordinates $\boldsymbol{\xi}$ of the point $\mathbf{x}$.

To obtain the correct element coordinates, these approximate values are used as initial guess in a Newton scheme, which subsequently tries to find the root of $\mathbf{r} = (\boldsymbol{\xi} - \mathbf{x}(\boldsymbol{\xi}))$ and, thus, invert the mapping in Eq. (7.11).

Runtime measurements have shown that the lower number of Newton iterations resulting from the heuristic with the four simplex elements to compute an initial guess outweighs the additional runtime for the heuristic and, in total, leads to a faster computation.

The Newton scheme uses the inverse Jacobian matrix of the mapping in Eq. (7.11). If the residual norm $||\mathbf{r}||_2$ cannot not be brought under the threshold of 10^{-8} in 16 iterations, this indicates that the problem of inverting the Jacobian is badly conditioned and the Jacobian has a large numerical error. In this case, the optimization is restarted using the derivative-free Nelder-Mead algorithm.

Before applying the Newton and Nelder-Mead algorithms, our implementation performs two basic checks that can directly terminate the computation of the coordinates: First, the coordinates of the point $\mathbf{x}$ are compared with the bounding box of all nodes of the element. If the point is outside the bounding box, the element coordinates do not have to be computed, and a different element, which contains the point $\mathbf{x}$, is searched.

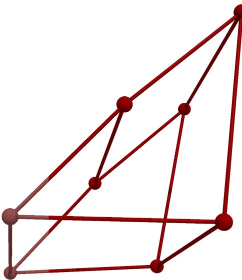


Figure 7.41: Hexahedral element of the muscle mesh with an interior angle of nearly 180°. Such elements lead to poor conditioning of the Jacobian matrix inversion problem and, as a consequence, require numerous Newton iterations in the setup of the data mapping between meshes.

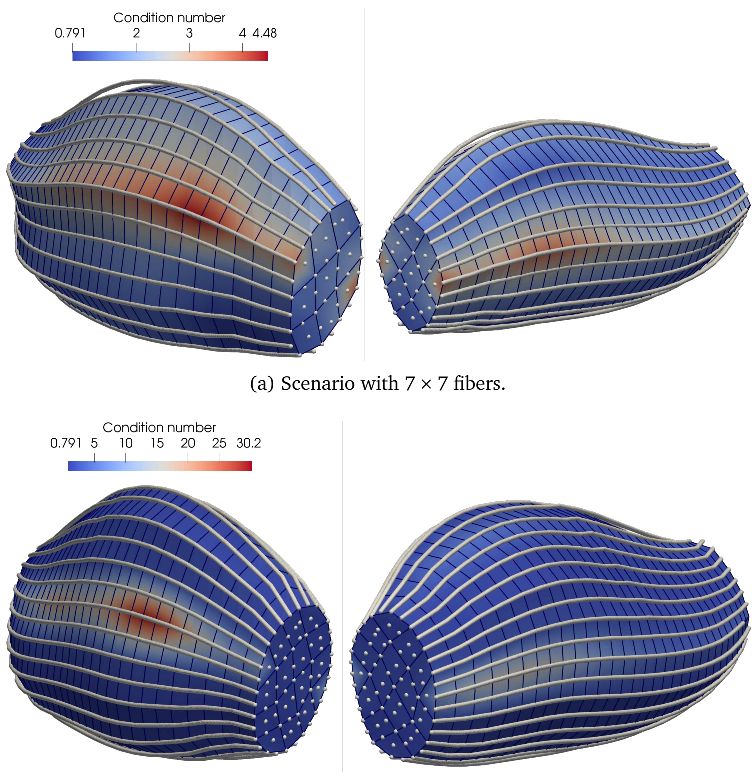
Second, all node positions are checked for equality with the point $\mathbf{x}$. If $\mathbf{x}$ is the same as one of the node positions, the element coordinates are directly known. This case frequently occurs, if one of source and target mesh is a subset of the other.

7.8.4 Conditioning of the Problem and Mapping Tolerances

As mentioned in the last section, the Newton scheme that solves the inverse problem of mapping a point from elemental coordinates to world coordinates uses the inverse Jacobian matrix of the mapping. The inversion of this matrix has a high numerical error, if the condition number of the Jacobian matrix is large. A large condition number can be found for 3D hexahedral elements, where the two element coordinate directions for ξ_1 and ξ_2 are almost linearly dependent. This is the case for elements with an interior angle of nearly 180°. Figure 7.41 shows such an element, which occurs at the outer boundary of the muscle mesh.

If the conditioning is too bad and the computed inverse Jacobian has a large numerical error, the Newton scheme fails to find a solution in the given maximum number of iterations and the Nelder-Mead algorithm is used instead. This algorithm usually succeeds. However, it requires significantly more compute time than the Newton scheme. The case where the Nelder-Mead algorithm is needed, however, only occurs for a small number of elements and only in highly-resolved meshes.

Figure 7.42 shows the condition number of the Jacobian matrix of the mapping from element to world coordinates per element. The condition number is numerically approx-



(a) Scenario with 7 × 7 fibers.

(b) Scenario with 9 × 9 fibers

Figure 7.42: Fiber meshes and corresponding muscle mesh, obtained with a sampling stride of two. The left and right view show both sides of the muscle mesh. The 3D mesh is colored by the condition number of the Jacobian matrix.

imated using the *von Mises* power iteration algorithm to obtain the largest eigenvalue of the Jacobian and its inverse. It can be seen in Fig. 7.42a that the elements with the highest condition number are located along longitudinal lines on the outer surface of the muscle mesh. Two such lines exist on both sides of the muscle. The cross-sectional mesh at the top of the muscle shows that the elements along these lines have large interior angles at the respective positions.

Figure 7.42b shows the same information for a different mesh with 9×9 fibers instead of the subset of 7×7 fibers in Fig. 7.42a. In Fig. 7.42b, the outer surface of the muscle

mesh is smoother and the interior angle of the elements along the respective longitudinal lines is even closer to 180°. Thus, the resulting maximum condition number has a higher value of 30.2 compared to 4.48 in the example of Fig. 7.42a.

Another effect can be seen in the visualization in Fig. 7.42a. The muscle mesh was generated from the fiber data with sampling strides of two in the cross-sectional directions. As a consequence, the nodes of the 3D mesh are part of every second fiber. This results in some outer fibers being located outside the domain of the 3D mesh. Such a case can be seen for the upper-most fiber in Fig. 7.42a.

To also involve such fibers in the computation, we enable data mapping between the 3D mesh and fibers that are outside but close to the 3D mesh. We add a tolerance parameter $\xi_\text{tolerance}$ to the implementation that specifies, how far outside the mesh fibers can be located to still be included in the mapping. On the element level, a point is considered to be part of an element, if its element coordinates (ξ_1, ξ_2, ξ_3) are no further than $\xi_\text{tolerance}$ off the element domain, i.e., for

$$-\xi_\text{tolerance} \le \xi_i \le 1 + \xi_\text{tolerance} \quad \forall i \in \{1,2,3\}.$$

This treats the outside fibers as if they were located inside the 3D mesh. For the fibers in the interior, the threshold leads to potentially multiple neighboring elements claiming ownership of a point. In this case, the element that contains the point without this tolerance value is chosen. By default, the tolerance value is set to $\xi_\text{tolerance} = 0.1$, but it can be adjusted to different values in the Python settings file if needed.

In summary, OpenDiHu can map data between any two overlapping meshes. The inversion of the mapping from elemental to world coordinates is an important task of this problem, which is non-trivial for 3D hexahedral elements and is solved numerically. The combination of fiber meshes with a 3D muscle mesh leads to specific effects such as degraded condition numbers or fibers outside the 3D mesh that have to be considered in the mapping.

How To Reproduce

The visualizations in Fig. 7.42 were obtained using the `electrophysiology/fibers/fibers_emg` example. The condition number of the Jacobian is computed by the `StaticBidomainSolver` if the parameter `"enableJacobianConditionNumber"` is set to `True`.

Chapter 8

Numerical Results and Discussion

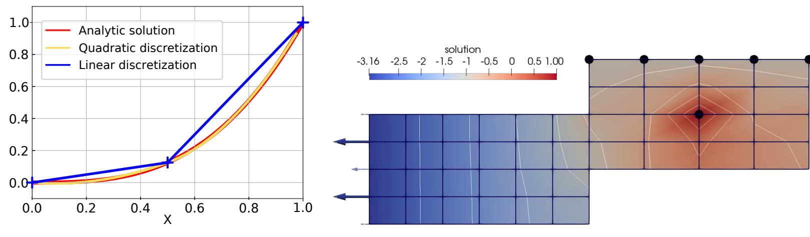
After various numerical models and methods for biophysical simulations of the neuromuscular system were described in the previous chapters, the remainder of this work deals with their application and discusses the newly obtained insights. The current chapter presents numerical results and demonstrates the use of OpenDiHu for all major components of the multi-scale models.

Section 8.1 begins with the simulation of toy problems such as Poisson and diffusion equations, which are used as building blocks for the more advanced simulations. Subsequently, dedicated solvers for the solid mechanics problem, the CellML models, the fiber based electrophysiology and the multidomain model are presented in Sections 8.2 to 8.5. Finally, Sec. 8.6 combines the fiber based electrophysiology model and the multidomain model with the solid mechanics solver to yield a comprehensive multi-physics simulation of muscle contraction.

8.1 Solution of Poisson and Diffusion Problems

Setting up a composite multi-scale simulation, where multiple equations are coupled, requires a profound understanding of the model components. Thus, it can help to first simulate isolated models. We provide simple examples with our software, such as Laplace and Diffusion problems, as prototypes for elliptic and parabolic partial differential equations. The examples with analytic solutions are also used to validate the basic finite element solvers.

In this section, we showcase three of these simple problems. First, we consider the 1D Poisson problem $u''(x) = f$ on $\Omega = [0,1]$ with Dirichlet boundary conditions $u(0) = 0$ and $u(1) = 1$ and right-hand side $f(x) = 6\,x$. The analytic solution is $y(x) = x^3$. Figure 8.1a shows the analytic solution and the results of the finite element computation with linear



(a) Solution of a 1D Poisson problem for linear and quadratic ansatz functions.

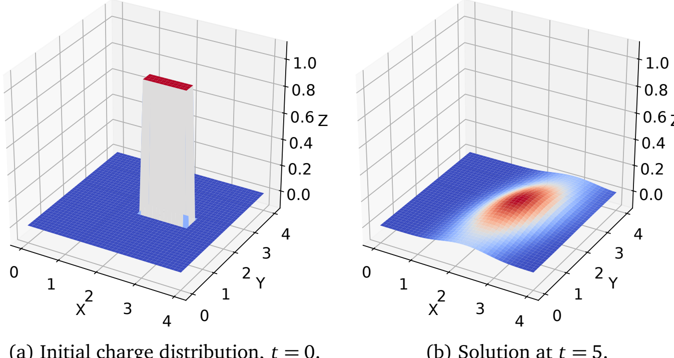
(b) Finite element mesh and solution of a 2D electric conduction problem .

Figure 8.1: Exemplary problems that can be solved with OpenDiHu and are part of the multi-scale problem.

and quadratic ansatz functions for two elements. Both linear and quadratic finite element solutions cannot exactly represent the cubic function, however, yield the best possible approximation. The first bidomain equation given in Eq. (5.9a) is a 3D version of this Poisson problem and is needed in the multi-domain model to simulate EMG signals on the muscle surface.

The second example is a 2D Laplace problem $c(x)\,\Delta u(x) = 0$. The solution is given in Fig. 8.1b and can be interpreted as a static electric potential field. The discretization uses quadratic Lagrange ansatz functions and is composed of two joined rectangular parts, each given by a structured mesh. The conductivity is set as $c = 1$ in the left part and as $c = 2$ in the right part. Dirichlet boundary conditions prescribe the electric potential at the five upper points in the right mesh to $u = -1$ and at the center of the right mesh as $u = 1$. In addition, Neumann boundary conditions $\partial u/\partial \mathbf{n} = -1$ corresponding to an outward electric current are set on the left boundary of the left mesh with the normal vector $\mathbf{n}$ pointing to the left. Figure 8.1b visualizes the values of the degrees of freedom of the right-hand side contribution of the Neumann boundary conditions by the arrows. A 3D Laplace problem is also part of the multi-scale model and describes volume conduction in the adipose tissue domain as formulated in Eq. (5.17).

The third presented example solves the 2D diffusion equation $\partial u/\partial t - \mathrm{div}(\boldsymbol{\sigma}\,\mathrm{grad}\,u) = 0$ with homogeneous Neumann boundary conditions. The equation can be interpreted as a transient electric conduction problem. As shown in Fig. 8.2a, the initial charge distribution is $u = 1$ in a rectangle in the inner of the domain and $u = 0$ everywhere else.



(a) Initial charge distribution, $t = 0$.

(b) Solution at $t = 5$.

Figure 8.2: A 2D electric conduction problem as a demonstrator for the solution of transient problems in OpenDiHu.

The anisotropic diffusion or conductivity tensor $\boldsymbol{\sigma}$ is constant in the domain and set to

$$\boldsymbol{\sigma} = \frac{1}{5}\begin{pmatrix} 1 & 1 \\ 1 & 6 \end{pmatrix}.$$

A regular mesh with 40×40 elements and linear finite element ansatz functions is used. Figure 8.2b shows the solution at time $t = 5$, where the initially discontinuous charge distribution has smoothed out and has expanded mainly in y direction, which is the preferential direction of electric conduction in this example. A 3D version of this equation is part of the multidomain model and given by Eq. (5.14).

How To Reproduce

The three presented simulations can be executed and visualized as follows:

```
cd $OPENDIHU_HOME/examples/poisson/poisson1d_2/build_release
./linear ../settings_1d.py && plot out/*.py
./quadratic ../settings_1d.py && plot out/*.py
./hermite ../settings_1d.py && plot out/*.py

cd $OPENDIHU_HOME/examples/laplace/laplace_composite/build_release/
./laplace_composite_2d ../settings_2d_2.py && paraview
  ↪ paraview_state.pvsm

cd $OPENDIHU_HOME/examples/diffusion/anisotropic_diffusion/
  ↪ build_release
./anisotropic_diffusion2d ../settings2d.py && plot out/*.py
```

8.2 Simulation of Solid Mechanics Models

Next, we demonstrate the solid mechanics solvers, which can be used to compute muscle contraction. In this section, we focus on the passive material behavior. As described in Sections 5.2.5 and 5.2.6, the mechanics equations can be computed in linearized or in nonlinear form within OpenDiHu. In the following, Sec. 8.2.1 applies both model approaches in a simulation of an externally stretched muscle and compares the results. Then, Sec. 8.2.2 validates the implementation of the nonlinear hyperelasticity solver in OpenDiHu. Finally, Sec. 8.2.3 showcases, using the simulation of a tendon, how more complex material models can be computed.

8.2.1 Comparison of Linear and Nonlinear Mechanics Models

We demonstrate the use of linear and nonlinear mechanics models in a simulation of an externally stretched biceps muscle. The muscle belly is fixed at its lower end and an upwards pulling force acts on the upper end, effectively stretching the muscle tissue in vertical direction.

We solve two scenarios with the same geometry and boundary conditions but different material models. The first scenario uses the linearized mechanics model given in Sec. 5.2.5. We use material parameters obtained from porcine in vitro indentor tests in literature [Sch82] and set the bulk modulus to $K = 39\,\text{kPa}$ and the shear modulus to $\mu = 48\,\text{kPa}$.

The second scenario uses the incompressible transversely isotropic hyperelastic muscle material based on the Mooney-Rivlin description without active stress, which is defined in Sec. 5.2.7. The material parameters are set to the values given [Hei16].

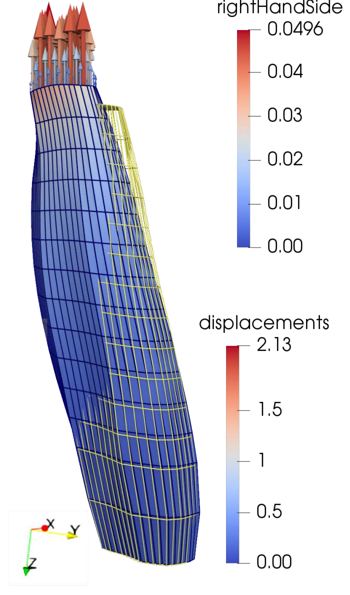
Figure 8.3 shows the geometric setup of the model. We discretize the biceps geometry by a 3D mesh with 252 elements, quadratic finite element ansatz functions, and a total of $13 \times 13 \times 15 = 2535$ nodes. The linearized material model uses this mesh to construct the stiffness matrix and to solve the linear system. For the nonlinear model, an additional coarser linear mesh is constructed, and linear-quadratic Taylor-Hood elements are used for the discretization.

A total force of $(F_x, F_y, F_z) = (0, -0.4\,\text{N}, -3\,\text{N})$ is applied, which points in negative z-direction, i.e., upwards in Fig. 8.3, and slightly in negative y-direction, i.e, to the left in Fig. 8.3. Instead of a single force vector acting on a point, the equivalent constant surface load is applied on the whole top face of the muscle geometry.

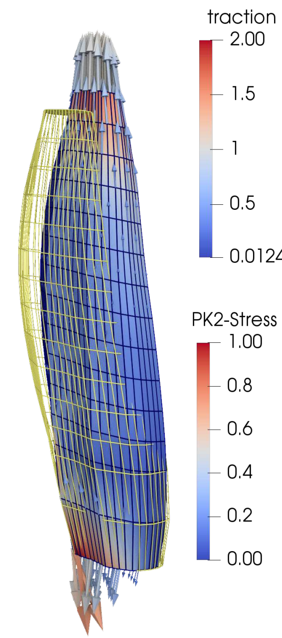
We consider a static problem where no timestepping is required. In the linear model, the resulting displacements are obtained by a GMRES solver, which solves the linear system of equations Eq. (5.76) corresponding to the finite element formulation. The nonlinear model uses increasing load steps as described in Sec. 7.7.3, which are adaptively refined in case the solver diverges at one load step. The scheme solves a system of nonlinear equations for every load step, and the contained linear system is solved by a direct solver.

The results of the linear and nonlinear models are shown in Figures 8.3a and 8.3b. In both images, the identical reference configuration is given by the yellow wireframe and the deformed muscle is given by the solid body with colored mesh. The deformed body in the linear model in Fig. 8.3a is colored according to the resulting vector of unknowns, which contains the displacements. Arrows on the upper end of the geometry indicate the negative right-hand side of the linear system as formulated in Eq. (5.75). The arrows correspond to the applied Neumann boundary conditions in the weak form of the finite element formulation and point in the direction of the applied surface load.

In the visualization of the nonlinear model in Fig. 8.3b, the deformed muscle body is colored according to the second Piola-Kirchhoff (PK2) stress. It can be seen that the stress



(a) Solution of the linear model. The arrows at the top visualize the (negated) right-hand side of the finite element formulation, with the absolute values indicated by the arrow lengths and color. The surface of the muscle mesh is colored according to the values of the displacements.



(b) Solution of the nonlinear model. The arrows specify the traction vectors in current configuration. Their absolute values are indicated by the arrow sizes and the color. The surface of the muscle mesh is colored according to the second Piola-Kirchhoff stress.

Figure 8.3: Solid mechanics solver example: Comparison of linear and nonlinear mechanics models. A biceps muscle is stretched by an applied force. The yellow mesh specifies the identical reference configuration in both scenarios.

is highest at the bottom bearing and at the top end, where the muscle cross-section is smaller. The arrows visualize the traction forces **t** on virtual horizontal cuts. As a result, the arrows that can be seen on top of the muscle geometry correspond to the applied external force, and the arrows at the bottom indicate the forces on the bearing.

A comparison of the two obtained results from the linear and nonlinear models shows a qualitatively different outcome. With the linear model, the muscle bends to the left, whereas, with the nonlinear model, it bends to the right. This effect is a result of the different material behavior. The linear model is isotropic and the deformation follows the direction of the applied force, which points to the upper left. The nonlinear model has an anisotropy and is stiffer in fiber direction. As a consequence, the muscle deforms less in longitudinal direction and therefore moves to the right. Thus, the material models influence the bending direction in this scenario.

In a second example, we compare the muscle stretches that results from different external forces acting in z-direction. We use the same scenario as before and increase the applied force from 0 to 15 N. We measure the displacement of one node in the top face of the muscle, for both the linear model and the nonlinear model. While the stress-strain relations in 1D extension tests can be derived analytically for linear and nonlinear models, our examples considers a real 3D setting where this relation is influenced by the geometry, e.g., by non-parallel fiber directions, as the force is not applied exactly in fiber direction.

Figure 8.4 shows the resulting muscle extensions for different applied external forces for the linear and nonlinear models. It can be seen that the stretch of the muscle increases nonlinearly for the transversely isotropic hyperelastic model, in contrast to the linear progression of the linear model. The slopes of the two curves are qualitatively different, which is a result of the chosen material parameters from different experimental origins. It would be possible to scale the linear model to better match the nonlinear model behavior by simply reducing the value of the bulk modulus accordingly.

The two presented studies show that a linear isotropic material model can give significantly different results than a more accurate nonlinear transversely isotropic model. Therefore, simulations of muscle contraction that target high accuracy should use the according nonlinear models. Nevertheless, both approaches are implemented and can be used with OpenDiHu.

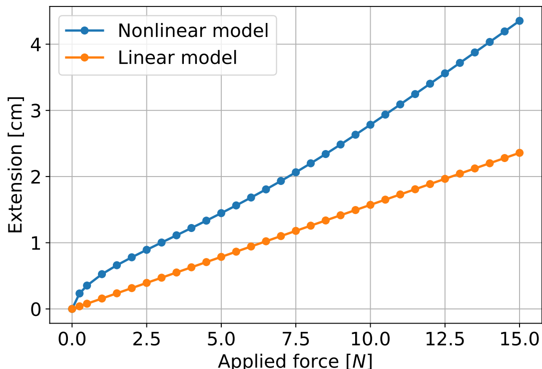


Figure 8.4: Solid mechanics example: Quantitative comparison of the relation between applied force and extension of the muscle for a linear and a nonlinear solid mechanics model.

How To Reproduce

The two simulations for Fig. 8.3 can be run as follows:

```
cd $OPENDIHU_HOME/examples/solid_mechanics/linear_elasticity/muscle/
  ↪ build_release
./linear_elasticity ../settings_linear_elasticity.py
cd $OPENDIHU_HOME/examples/solid_mechanics/mooney_rivlin_transiso/
  ↪ build_release
./3d_hyperelasticity ../settings_3d_muscle.py --njacobi=1
```

The study in Fig. 8.4 can be run and plotted using the scripts in the repository at github.com/dihu-stuttgart/performance in the directory `opendihu/23_linear_nonlinear_mechanics`.

8.2.2 Validation of the Nonlinear Solid Mechanics Solver

Next, we perform tests to validate our implementation of the nonlinear hyperelasticity solvers. We simulate the same scenario with our software and with the nonlinear finite element analysis tool *FEBio* [Maa12]. FEBio is developed at the University of Utah and the Columbia University in the USA. FEBio contains solid mechanics solvers that can be run

from the command line or a graphical user interface model. An extensive model library contains material models also from the domain of biomechanics. The mechanics solver uses the PARDISO linear solver [Ala20], which exploits shared memory parallelism.

An adapter in OpenDiHu exists, which can output the required configuration file for FEBio, run the solver, and parse the computed solution from the text files that are output by FEBio. Thus, we can conduct our validation studies fully in OpenDiHu by using similar Python settings files and the same meshes for the computation in OpenDiHu and the reference solution computed by FEBio.

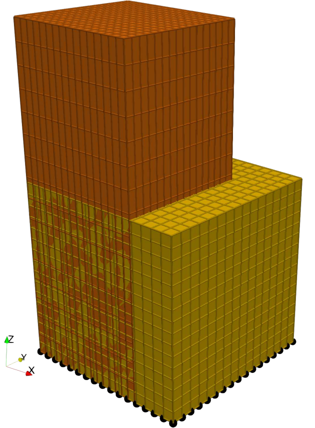
Apart from the present study, the FEBio adapter in OpenDiHu can also be used to solve quasi-static coupled problems with the electrophysiology part solved in OpenDiHu and the mechanics part solved in FEBio. However, test have shown that the interfacing method of generating configuration files and parsing result files in every timestep leads to higher runtimes than directly using the mechanics solver of OpenDiHu.

In our validation studies, we consider a unit cube that is discretized by $8 \times 8 \times 8$ quadratic elements and 4913 degrees of freedom. Figure 8.5 shows the discretized cube in yellow color. Its orientation is given by the coordinate frame in the lower left of Fig. 8.5a. The following Dirichlet boundary conditions are prescribed: All points of the lower face are fixed at $z = 0$. The points of the two edges $(y = 0 \wedge z = 0)$ and $(x = 0 \wedge z = 0)$ are additionally fixed in y and x directions, respectively. The corner at $x = y = z = 0$ is fixed completely. Thus, the cube can freely deform in its bottom plane, but not move nor rotate as a whole.

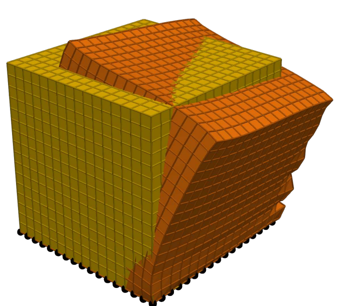
The first study is a tensile test, where a uniform surface load pointing in positive z direction is applied on the top face of the cube. We increase the force from 1 to 50 N. For the largest force, the cube deforms as shown by the orange geometry in Fig. 8.5. Note that the volume is preserved due to the incompressibility constraint in the material model.

We use an incompressible and isotropic Mooney-Rivlin material with parameters $c_1 = c_2 = 1$. The material can be simulated in three different forms in OpenDiHu. In the following, we list all model formulations in OpenDiHu and the reference formulation in FEBio, expressed by the strain energy functions Ψ, Ψ_{iso} and Ψ_{vol} introduced in the modeling chapter in Sec. 5.2.6:

(i) the "fully incompressible", mixed u-p formulation, which ensures incompressibility



(a) Tensile test scenario used in the first validation experiment.



(b) Shear test scenario used in the second validation experiment.

Figure 8.5: Scenarios used for validation of the solid mechanics solver. The reference and the current configuration are given by the yellow and orange meshes, respectively.

using the Lagrange multipliers,

$$\Psi_{\text{iso}}(\bar{I}_1, \bar{I}_2) = c_1\,(\bar{I}_1 - 3) + c_2\,(\bar{I}_2 - 3), \quad J = 1, \tag{8.1}$$

(ii) the “nearly incompressible” formulation in terms of the invariants I_1 to I_3,

$$\Psi(I_1, I_2, I_3) = c_1\,(I_1 - 3) + c_2\,(I_2 - 3) + \kappa\,(\sqrt{I_3} - 1)^2 - d\,\log(\sqrt{I_3}), \tag{8.2a}$$

$$d = 2\,(c_1 + 2\,c_2), \tag{8.2b}$$

(iii) the nearly incompressible formulation given in decoupled form, in terms of the reduced invariants $\bar{I}_1$ and $\bar{I}_2$,

$$\Psi_{\text{iso}}(\bar{I}_1, \bar{I}_2) = c_1\,(\bar{I}_1 - 3) + c_2\,(\bar{I}_2 - 3), \tag{8.3a}$$

$$\Psi_{\text{vol}}(J) = \kappa\,G(J) \quad \text{with } G(J) = \frac{1}{4}\big(J^2 - 1 - 2\,\log(J)\big), \tag{8.3b}$$

(iv) and the one used in FEBio, which also describes a nearly incompressible material in

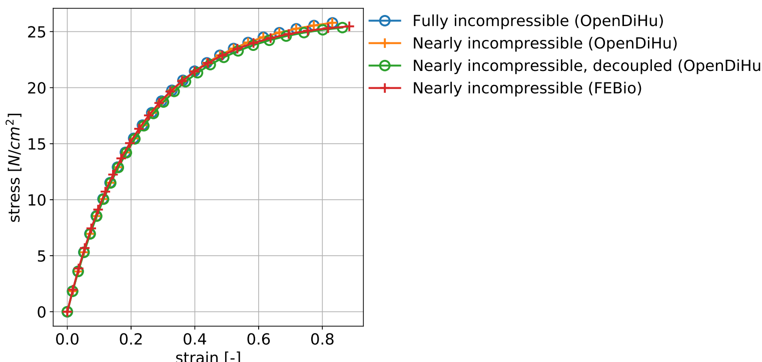


Figure 8.6: Solid mechanics solver validation: Results of the tensile test validation experiment. The stress-strain curve for three different formulations in OpenDiHu and the computation in FEBio match.

decoupled form, but with a different penalty function $G(J)$,

$$\Psi_{\text{iso}}(\bar{I}_1, \bar{I}_2) = c_1\,(\bar{I}_1 - 3) + c_2\,(\bar{I}_2 - 3), \tag{8.4a}$$

$$\Psi_{\text{vol}}(J) = \kappa\, G(J) \quad \text{with } G(J) = \frac{1}{2}\big(\log(J)\big)^2. \tag{8.4b}$$

For the three nearly incompressible descriptions in Equations (8.2) to (8.4), we set the incompressibility parameter to $\kappa = 10^3$.

We compare the resulting normal stress value S_{33} in z-direction of the second Piola-Kirchhoff stress tensor $\mathbf{S}$ for all formulations listed in Equations (8.1) to (8.4). For the tensile test, this stress value is constant throughout the domain. Figure 8.6 shows the computed stresses over the computed strain values. It can be seen that the three formulations in OpenDiHu yield approximately the same results as the reference solution given by FEBio over the whole range of applied forces.

As the previous tensile test only validates stress and strain in one direction, we additionally conduct a numerical shear experiment. A shear force $\mathbf{F} = (0.1\alpha, 0.05\alpha, 0)^\top$ is applied on the top face of the cube and α is again varied between 1 and 50 N. Figure 8.5b shows the deformed configuration for the highest force by the orange colored body.

In this second study, we consider one point in the interior of the domain, which is

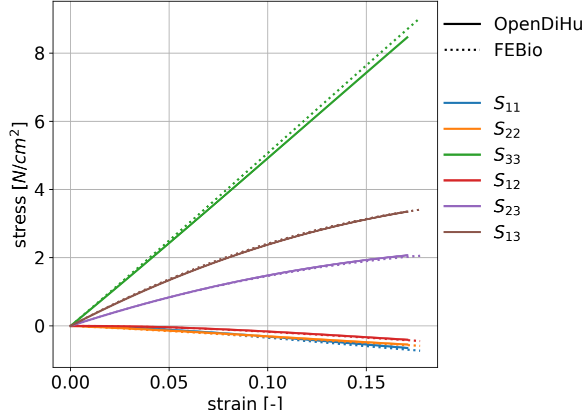


Figure 8.7: Solid mechanics solver validation:Results of the shear test validation experiment. The values of the second Piola-Kirchhoff tensor computed by OpenDiHu (solid lines) and FEBio (dotted lines) closely match.

3 elements below the top face of the mesh. We compare all six distinct entries of the symmetric second Piola-Kirchhoff tensor **S** between the fully incompressible model in OpenDiHu and the nearly incompressible model in FEBio.

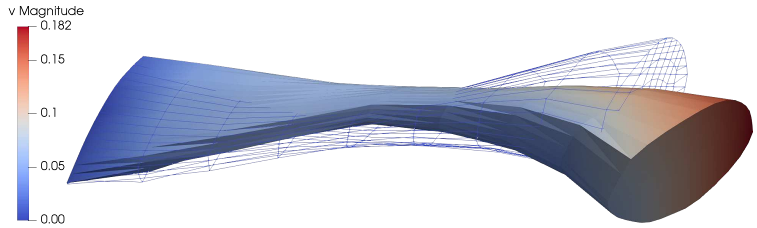
Figure 8.7 shows the computed values in a stress-strain diagram. The solutions of OpenDiHu and FEBio are given by solid and dotted lines, respectively. It can be seen that the curves coincide, which validates the implementation in OpenDiHu.

How To Reproduce

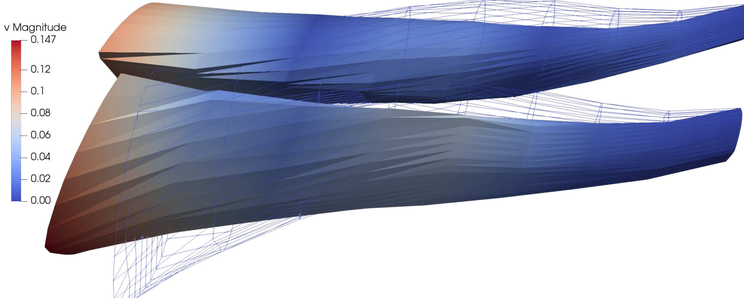
The tensile test validation experiment can be reproduced by the following commands:

```
cd $OPENDIHU_HOME/examples/solid_mechanics/tensile_test/
  ↪ build_release
../run_force.sh
cd $OPENDIHU_HOME/examples/solid_mechanics/tensile_test
./plot_validation.py
```

The shear test can be executed analogously by replacing `tensile_test` by `shear_test` in the given paths.



(a) Dynamic simulation of the lower tendon of a biceps brachii. The attachment to the ulna bone is at the left end. The free right end bends due to the applied surface traction.



(b) Simulation of the two upper tendons of the two biceps heads.

Figure 8.8: Simulation of tendons as a showcase of dynamic simulations with complex material models. The color coding indicates the velocity.

8.2.3 Simulation of a Hyperelastic Tendon Material

Next, we demonstrate the use of a more complex constitutive material model, which represents tendon tissue. The material is formulated in [Car17]. The model describes microstructural interactions between collagen fibers and their matrix. It consists of a transversely isotropic model, which describes the high stiffness in fiber direction, and a coupled model for the compressive response. The model is formulated in terms of a logarithmic strain measure.

Figure 8.8 shows the geometries of the tendons of the biceps brachii and the results of the simulations. The lower tendon in Fig. 8.8a is fixed at its left end and a constant surface traction of 1 N in total pulls to the right. The image shows the initial configuration by the wireframe mesh and the current configuration after $t = 10\,\mathrm{ms}$, colored according to the resulting velocity. Similarly, the upper tendons in Fig. 8.8b are fixed at the right ends and stretched to the left resulting from the applied force at the left end.

In summary, this section demonstrated the capabilities of the solid mechanics solvers in OpenDiHu. Sections 8.2.1 and 8.2.3 simulated extension of the biceps muscle and tendons due to external forces. The comparison of results from a linear and a nonlinear model showed that a linear isotropic material cannot always accurately predict the behavior of muscle tissue and, thus, a nonlinear model is required. The validation experiments in Sec. 8.2.2 demonstrated that OpenDiHu correctly computes deformation and stresses of incompressible materials.

The solid mechanics solvers can also be coupled to solvers of electrophysiology to simulate muscle contraction resulting from the spatially heterogeneous activation and considering the neuronal stimulation dynamics. Moreover, coupled simulations of the muscle and tendons are possible. Such simulations are described in Sec. 8.6.1 and Sec. 8.6.5, respectively.

8.3 Simulation of CellML Models

The subcellular models used in the multi-scale model are given in CellML description and can be solved in OpenDiHu using the `CellmlAdapter` class. In the following, we show simulation results of the most commonly used CellML models in this work.

8.3.1 Simulation of Subcellular Models

First, we consider a single instance of the subcellular model of Shorten et al. [Sho07]. We solve the model with Heun's method with a timestep width of $dt_{0D} = 10^{-5}$. A stimulation current of $I_{stim} = 40\,\frac{\mu A}{cm^2}$ is applied during the time range $[5\,ms, 5.1\,ms]$. Figure 8.9 shows the resulting values of the membrane voltage V_m over time in the upper plot and the temporal evolution of all other variables in the lower plot.

In the upper plot, the depolarization and repolarization of the membrane can be seen upon the stimulation at $t = 5\,ms$, exhibiting the characteristic action potential shape. The membrane voltage V_m reaches its equilibrium value approximately 5 ms after the stimulation. The lower plot in Fig. 8.9 shows longer durations for several other variables to return to the equilibrium state. As a consequence, the generated force or active stress of the sarcomeres, indicated by the thick red line in Fig. 8.9, does not directly decrease after the stimulation is over. For frequent stimulation patterns, the force level would show a smooth progression over time.

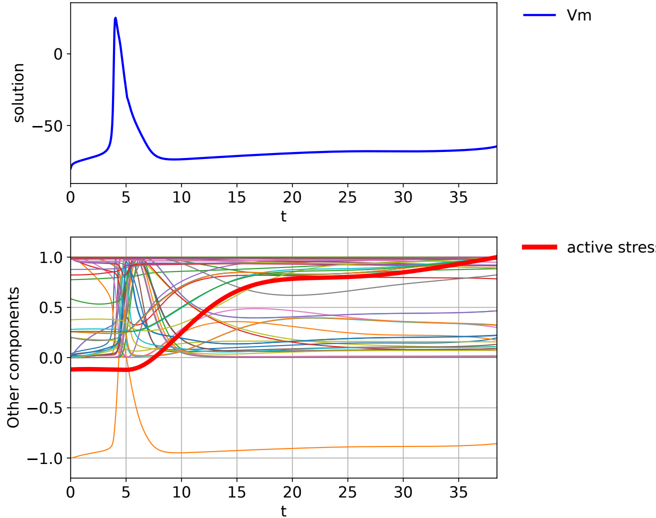


Figure 8.9: Simulation of the Shorten subcellular model over time t in milliseconds. The cell is stimulated at $t = 5$ ms. The upper plot shows the membrane voltage V_m, the lower plot show all other variables, some of which are listed in the legend on the right.

The propagation of the action potentials of the Shorten subcellular model can be simulated with the monodomain equation (Eq. (5.11)). Figure 8.10 shows simulation results on a 1D muscle fiber mesh with length 1 cm, discretized to 100 elements. The muscle fiber is stimulated at its center at $t = 0$. The upper plot in Fig. 8.10 displays the membrane voltage V_m at time $t = 3.5$ ms. Two action potentials, which move towards both ends of the fiber can be identified. The lower plot shows all other variables, normalized to the value range $[-1, 1]$. At the outer ends of the fiber, the variables are still in equilibrium, whereas towards the center, their values change dynamically as the action potentials propagate.

Figure 8.11 shows analog results for a simulation with the subcellular model of Hodgkin and Huxley [Hod52a]. Two action potentials can be seen at time $t = 4.25$ ms on a fiber mesh with 200 nodes and of 2 cm length. The system state is fully described by the values of the four variables that are plotted in Fig. 8.11.

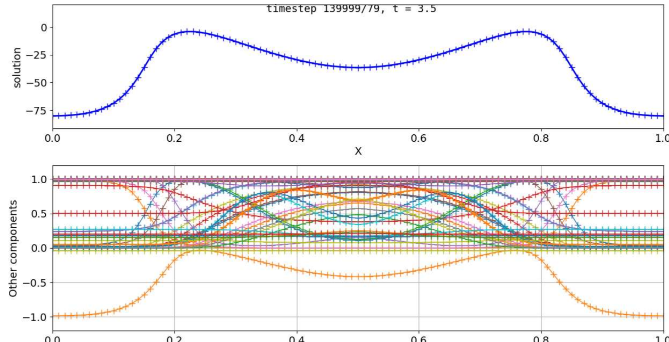


Figure 8.10: Simulation of action potential propagation on a 1D mesh with the Shorten subcellular model. The upper plots shows the membrane voltage V_m at time $t = 3.5$ ms over the fiber along the x axis. The lower plot shows all other variables of the model.

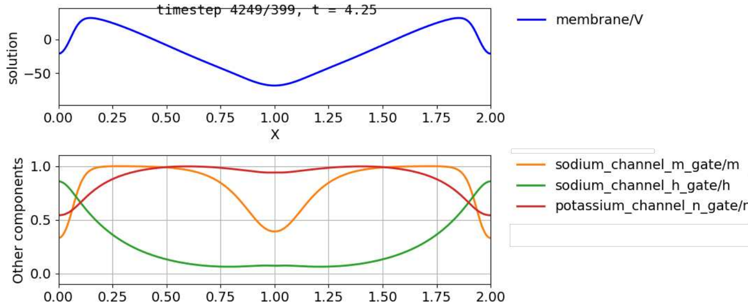


Figure 8.11: Simulation of action potential propagation on a 1D mesh with the Hodgkin-Huxley subcellular model for time $t = 4.25$ ms, analog to Fig. 8.10.

How To Reproduce

The simulation of a single instance of the Shorten model and the plot of Fig. 8.9 can be obtained as follows:

```
cd $OPENDIHU_HOME/examples/electrophysiology/cellml/shorten/
  ↪ build_release
./cellml ../settings_cellml.py
cd out && plot
```

The simulation of the monodomain equation for the Shorten model shown in Fig. 8.10 can be executed and visualized as follows:

```
cd $OPENDIHU_HOME/examples/electrophysiology/monodomain/
  ↪ new_slow_TK_2014_12_08/build_release
./shorten_implicit ../settings_new_slow_TK_2014_12_08.py
cd out && plot
```

8.3.2 Simulation of Motor Neuron Models

If an activation model with a pool of motor neurons is considered in the neuromuscular multi-scale model, the transient behavior of motor neurons has to be simulated as well. In our simulations, we use the motor neuron model of Cisi and Kohn [Cis08]. The drive parameter of the model is set to a constant value of $0.01\,\frac{\text{V}}{\text{s}}$. As a consequence, the motor neuron fires with a frequency that depends on the input drive, which in the presented scenario is approximately 25 Hz.

Figure 8.12 shows the evolution of the different variables of the model over time. Six firing times can be identified.

To connect the motor neuron with the fibers of the MU in the simulation, we stimulate the muscle fibers whenever the V_s value of the motor neuron reaches a certain threshold. It is possible to configure a pool of several motor neurons with different model parameters and different input drive values. Each motor neuron can be connected to a different set of fibers, according to the MU to fiber association. As a result, the MUs get physiologically activated according to the different firing frequencies of the motor neurons.

In summary, we showed simulations of the 0D subcellular models of Shorten et al. and Hodgkin and Huxley, simulations of these models together with the 1D conduction

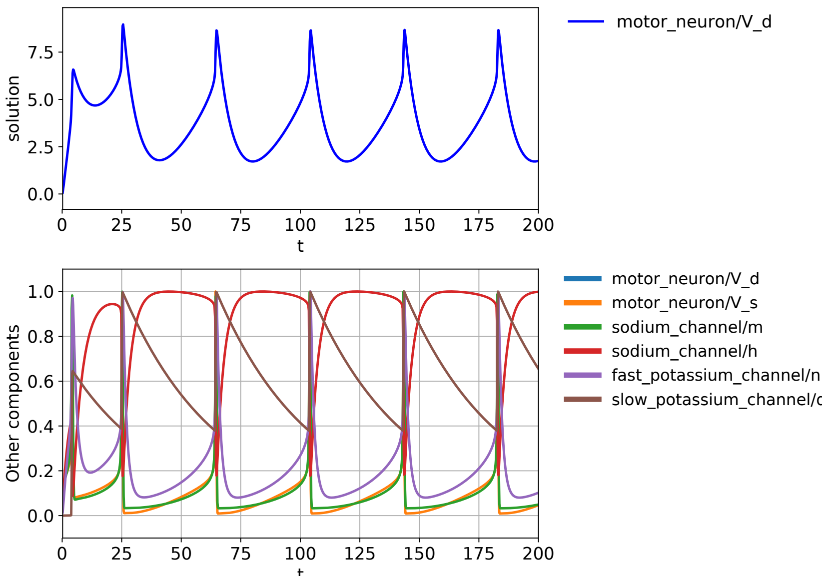


Figure 8.12: Simulation of the motor neuron model of Cisi and Kohn [Cis08] for a constant input drive of 0.01 $\frac{\text{V}}{\text{s}}$. The evolution of all variables of the model is plotted over time t in milliseconds.

model in the monodomain equation, and a simulation of motor neurons. All of these models are given in CellML description and can be further combined with other parts of the multi-scale model, e.g., to simulate surface EMG signals.

How To Reproduce

The simulation and visualization for Fig. 8.12 can be executed with the following commands:

```
cd $OPENDIHU_HOME/examples/electrophysiology/monodomain/
  ↪ motoneuron_cisi_kohn/build_release
./motoneuron_cisi_kohn ../settings_motoneuron_cisi_kohn.py
cd out && plot motoneuron*
```

This simulation also computes the monodomain equation for one muscle fiber that gets activated whenever the motor neuron fires.

8.4 Simulation of Fiber Based Electrophysiology

In this section, we consider surface EMG signals on the upper arm by simulating the activation of the biceps brachii muscle. We use the fiber based multi-scale model consisting of 1D action potential propagation on muscle fibers, potentially involving a 0D subcellular model, and the 3D bidomain model. In Sec. 8.4.1, we introduce the setting of the simulation and present an exemplary scenario to compute EMG signals. Subsequently, we simulate various scenarios to investigate the effects of different model parameters and numerical settings on the resulting EMG signal. Section 8.4.2 considers the effects of single motor units, Sec. 8.4.3 the fat layer, and Sec. 8.4.4 shows effects of the mesh width. Section 8.4.5 presents a way to simulate realistic EMG electrodes and Sec. 8.4.6 deals with the decomposition of EMG signals. In Sec. 8.4.7, we describe our simulations with a phenomenological model for action potential propagation.

8.4.1 Overview of the EMG Simulation

Figure 8.13 shows the setting of the biceps muscle and the tendons, which attach to the skeleton near the shoulder and to the ulna bone in the forearm. For the simulation of EMG, we only consider the muscle belly of the biceps muscle. Figure 8.13 shows muscle fibers inside the muscle, which run in longitudinal direction between the tendons at both ends. The image also visualizes the results of an EMG simulation. The fibers are colored according to the transmembrane potential V_m. On some fibers, action potentials can be seen.

The surface of the muscle is colored by the extracellular electric potential ϕ_e. In a reasonable approximation, the value of ϕ_e corresponds to the measured EMG signals on the skin surface. Additionally, we consider volume conduction in a layer of adipose tissue on top of the muscle in the following section.

For the EMG simulations, we solve the multi-scale model of fiber based electrophysiology. We solve the monodomain equation Eq. (5.11) independently on all 1D muscle fiber meshes. After a fixed number of timesteps, we map the membrane voltage V_m from the 1D meshes to the 3D mesh. Subsequently, we solve the static bidomain equation Eq. (5.9a) on the muscle domain and potentially the body fat domain to obtain the ϕ_e values on the skin surface.

Figure 8.14 shows a close-up view of the active muscle fibers and the resulting EMG signals on the upper surface, which are identical to Fig. 8.13. The scenario considers

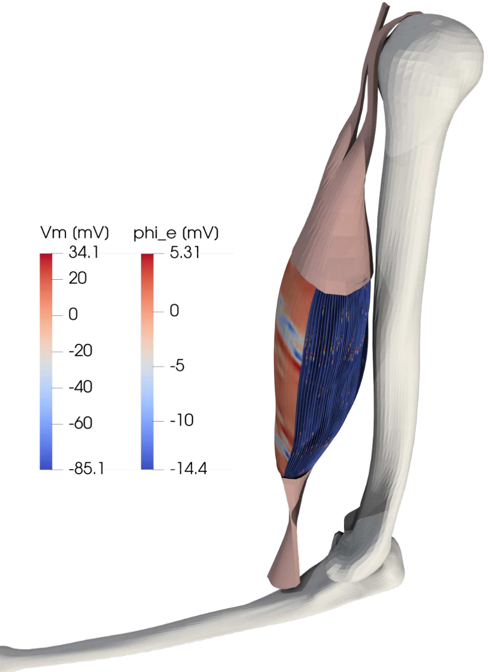


Figure 8.13: Considered setting for simulations of surface EMG for the upper arm, consisting of the biceps brachii muscle, tendons and bones. A simulation result of the membrane voltage V_m on the muscle fibers and the extracellular potential ϕ_e on the surface is shown.

961 muscle fibers, each described by a 1D mesh with 1481 nodes. It can be seen that they are approximately equally spaced as a result of the meshing algorithms described in Chap. 3.

Figure 8.14 also shows the mesh of the muscle surface, which is colored according to the extracellular potential ϕ_e. It can be seen that the values correlate with the activation state of the underlying fibers. At the two blue colored regions at the surface near the left and right end of the muscle, the ϕ_e value is close to its minimum, while the majority of fibers exhibits its maximum positive V_m value. Towards the center of the muscle, the value of ϕ_e increases to its maximum, which reflects the hyperpolarization of the muscle fibers behind the propagating action potentials, i.e., the overshoot of the membrane voltage before it reapproaches the equilibrium level.

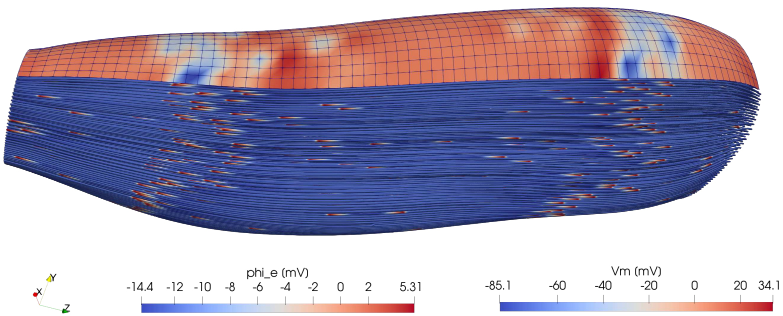


Figure 8.14: Overview of some of the meshes in an electrophysiology simulation: 1369 muscle fibers are located in the muscle belly. A 2D surface mesh on top of the muscle describes the computed EMG values. The visualized simulation is the same as in Fig. 8.13.

The lower left corner in Fig. 8.14 shows the coordinate frame that is used in all simulations. The z axis is approximately oriented in fiber direction, the x and y axes are oriented in transverse direction and describe cross-sectional planes of the muscle.

The scenario uses the subcellular model of Hodgkin and Huxley [Hod52a]. The shown image corresponds to the simulation time of $t = 200\,\mathrm{ms}$. The simulation also considers a body fat mesh layer on top of the muscle, which is not visualized in Fig. 8.14.

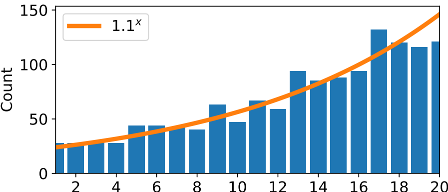
We run the simulation with 128 processes on a two-socket shared-memory node comprising two AMD EPYC 7742 64-core processors with 2.87 GHz clock frequency and 1.96 TiB RAM. The total runtime for a simulation end time of one second is 8 h 38 min.

8.4.2 Effects of Single Motor Units on the Electromyography Signal

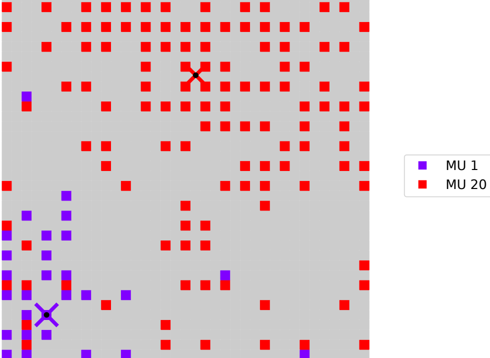
Next, we investigate how the surface EMG signals are influenced by several parameters of the simulation. We begin by studying EMG of only a single activated MU in the muscle.

The first scenario contains 20 MUs that connect to an exponentially increasing number of fibers as shown in Fig. 8.15a. The progression follows the function $y = c\,1.1^x$ for an appropriate constant $c > 0$. The MU assignment is created using method 1a of the algorithm described in Chap. 4, where the MU territories are centered around given points, and neighboring fibers are never part of the same MU.

Figure 8.15b shows the fibers that are assigned to the smallest and to the largest MU, MU 1 and MU 20. For this visualization, the muscle cross-section is mapped to the large gray square and every colored small square corresponds to one fiber. The purple and red crosses indicate the center of the MU territories for MU 1 and 20, respectively. As a consequence, the fibers of MU 1 are mostly located at the bottom left of the cross-section and the fibers of MU 20 are mostly located in the upper right region of the muscle cross-section. The visualization shows that the fibers of the same MU always have some spacing between them, which is due to the construction of the MU assignment algorithm.



(a) Exponential distribution of motor unit sizes. The diagram shows the motor unit numbers with the corresponding sizes or fibers counts of the MUs.



(b) Fibers that belong to motor units 1 and 20. The crosses are the center points around which the MU territories are generated by our algorithm.

Figure 8.15: Fiber based upper arm EMG simulation: Assignment of the 1369 fibers to 20 motor units used in the simulation scenario for fiber based electrophysiology.

We begin with a simulation scenario, where only a single MU is stimulated, and study the effect on the surface EMG. The fibers of the respective MU are stimulated with a frequency $f = 24\,\text{Hz}$ starting at time $t = 0\,\text{ms}$. Each of the $13 \times 13 = 1369$ fibers consists of a mesh with 1481 nodes, the 3D mesh of the muscle contains $19 \times 19 \times 38 = 13\,718$ nodes and the 3D mesh of the fat layer contains $37 \times 5 \times 38 = 7030$ nodes. The domains are partitioned into 27 subdomains associated to 27 MPI ranks. The subcellular model of Hodgkin and Huxley is used, yielding a total number of more than $8.1 \cdot 10^6$ degrees of freedom. The timestep widths are $dt_{0D} = dt_{\text{splitting}} = 2.5 \cdot 10^{-3}\,\text{ms}$, $dt_{1D} = 6.25 \cdot 10^{-4}\,\text{ms}$ and $dt_{3D} = 5 \cdot 10^{-1}\,\text{ms}$, leading to 4 subcycles for the 1D model in each splitting step and 200 splitting steps per solution of the bidomain equation.

We compute the linear systems for the initial potential flow problem to estimate fiber

directions in the 3D domain, Eq. (3.8), and for the bidomain equation Eq. (5.9a), which is solved in every timestep using a conjugate gradient solver. The program uses the `FastMonodomainSolver` class for the electrophysiology model. The Thomas algorithm solves the linear system of the diffusion problem. We use the `"vc"` optimization type and employ the scheme to only compute active fibers and the subcellular problems that are not in equilibrium.

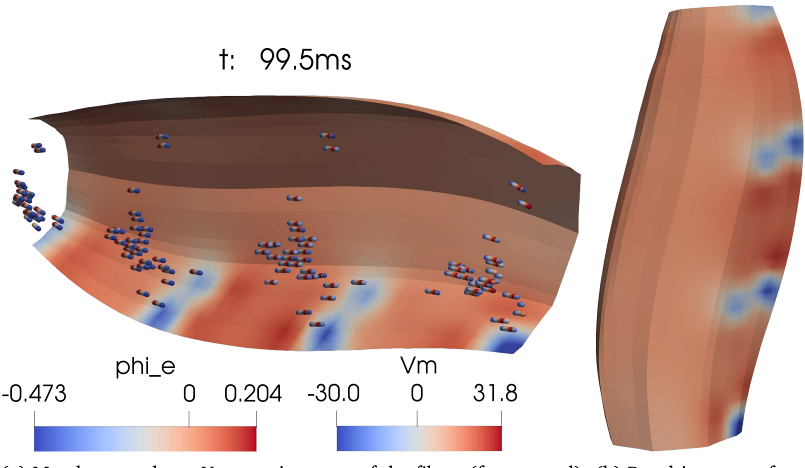
The computation of a simulated time span with $t_{\text{end}} = 100\,\text{ms}$ on an AMD EPYC 7742 64-core processor with 2492 MHz base frequency and 1.96 TB RAM takes approximately 100 s in the scenario that activates only the smallest MU, and 126 s in the scenario that activates only the largest MU.

Figure 8.16 shows the result for the scenario of activating the smallest MU, MU 1. In Fig. 8.16a, the surface is shown in the background and colored according to the extracellular potential ϕ_e, which represents the EMG signal. The muscle volume is not shown. Instead, the active parts of the respective fibers are displayed as tubes in the 3D domain. Their color visualizes the value of the transmembrane voltage V_m. In every of these small tube segments, the rising and declining shape of an action potential can be observed by the color progression from blue over orange to red for the rising part and back to blue for the declining part.

In this scenario, the fibers of MU 1 are stimulated three times within the first 100 ms at 0 ms, 41.6 ms and 83.3 ms. The innervation zone contains the starting points for the propagating stimulus on every fiber. The scenario positions the neuromuscular junctions randomly with a uniform distribution within the central 10 % of every muscle fiber. The activated parts of the fibers visible in Fig. 8.16a correspond to the propagated action potentials of the last two stimulations in this scenario.

By comparing the results in Fig. 8.16a with the fiber distribution in Fig. 8.15b, it can be seen that fibers of MU 1 are located opposite of the outer arm surface, which is at the upper side of the cross-sectional square diagram in Fig. 8.15b. The left side of the diagram in Fig. 8.15b corresponds to the lower part of the skin in Fig. 8.16a. This part of the skin is closer to the activated fibers and, thus, the effect on the surface EMG is highest for this region.

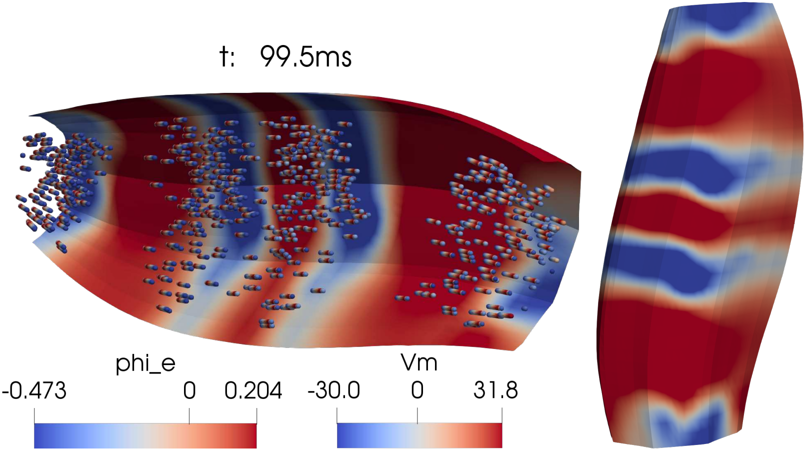
Figure 8.16b shows the skin surface as seen from the inside of the arm in Fig. 8.16a. The active region is located on the right-hand side in this image. It can be seen that the active region on the skin surface, which results from fibers of the activated MU 1, only spans a small portion of the surface.



(a) Membrane voltage V_m at active parts of the fibers (foreground) and EMG signals ϕ_e on the skin surface (background).

(b) Resulting surface EMG.

Figure 8.16: Fiber based EMG simulation for the upper arm (biceps) model: Simulation result at $t = 99.5$ ms where only MU 1 is activated.



(a) Membrane voltage V_m at active parts of the fibers (foreground) and EMG signals ϕ_e on the skin surface (background).

(b) Resulting surface EMG.

Figure 8.17: Fiber based EMG simulation for the upper arm (biceps) model: Simulation result at $t = 99.5$ ms where only motor unit 20 is activated, analog to Fig. 8.16.

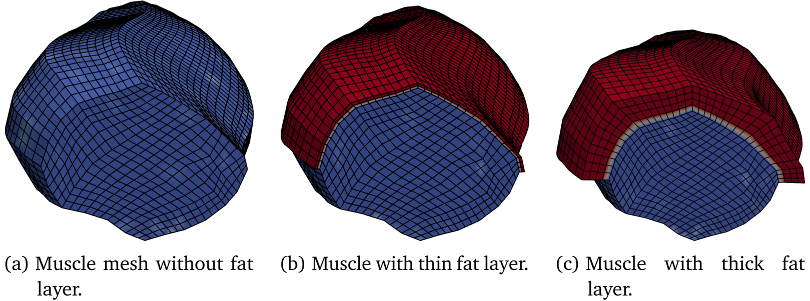
Figure 8.17 shows the analogous scenario that activates MU 20 instead of MU 1. Figure 8.17a shows that, now, more fibers are activated as MU 20 is larger than MU 1. According to the MU layout in Fig. 8.15b, the active fibers are also located closer to the skin surface. This layout results in a stronger EMG signal compared to the previous scenario.

The color coding in the two scenarios in Figures 8.16 and 8.17 is identical, and it can be seen that the absolute value of the extracellular potential ϕ_e is larger in the scenario for MU 20. For the scenario with MU 1 in Fig. 8.16, the value range of the extracellular potential ϕ_e is $[-0.473\,\text{mV}, 0.204\,\text{mV}]$. For the scenario with MU 20 in Fig. 8.17, it is $[-0.834\,\text{mV}, 0.579\,\text{mV}]$, which is more than twice the range.

Figure 8.17b shows the overall EMG signal on the skin surface for MU 20. Compared to the result of MU 1 in Fig. 8.16b, nearly the inverse region is activated. It can, thus, be observed that the EMG signal is highly influenced by the location and size of the MUs. MUs with territories closer to the skin surface have a larger effect on the EMG signals than MUs that are located further away. As seen in Fig. 8.16b, the influence of fibers completely vanishes if the distance is larger than a certain value. The effects of several

close fibers add up, such that large MUs located near the surface have the largest impact on the resulting EMG signal.

8.4.3 Effects of the Fat Layer on the Electromyography Signal



(a) Muscle mesh without fat layer. (b) Muscle with thin fat layer. (c) Muscle with thick fat layer.

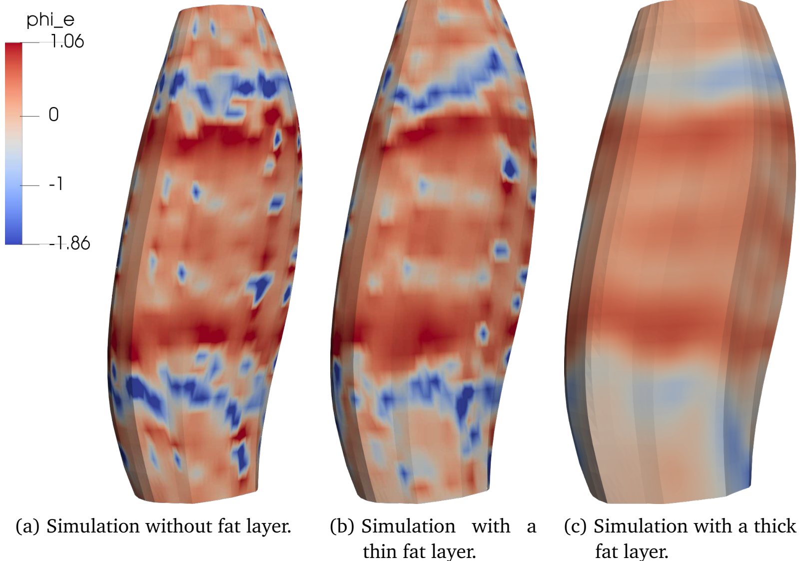
Figure 8.18: Fiber based EMG simulation for the upper arm (biceps) model: Meshes for the muscle domains (blue) and the layer of adipose tissue (red) used in the study to compare different fat layer widths.

In the next study, we investigate the effect of the fat layer on the resulting EMG signals. The same scenario as in the previous section is used, except that the size of the body fat domain is varied and the activated MUs are chosen differently. We consider the domains and meshes shown in Fig. 8.18: Scenario (a) only considers the muscle domain without additional fat layer. Scenario (b) adds a thin fat layer with thickness of 2 mm, discretized by two layers of finite elements. Scenario (c) considers a fat layer with thickness of 1 cm and four layers of elements. The scenario in the previous section also used this thick fat layer.

In this series of experiments, the first 10 MUs are activated with different stimulation frequencies ranging from 7 Hz for the smallest MU to 15.15 Hz for MU 10. The runtime of the simulation for one scenario on the same hardware as in the previous section is approximately 9 min.

Figure 8.19 shows the simulation results at $t = 100\,\text{ms}$ for the three scenarios with different fat layers. The figure uses the same color coding for the extracellular potential ϕ_e in all three scenarios. It can be seen that the volume conduction in the fat layer significantly smooths the resulting EMG signal, especially for the thick fat layer. The

scenarios with no fat layer and the thin fat layer also exhibit a small difference. This effect has implications for experimental studies, where the EMG recordings capture the less resolved spatial information, the more tissue is located between the muscle and the surface electrodes.



(a) Simulation without fat layer.

(b) Simulation with a thin fat layer.

(c) Simulation with a thick fat layer.

Figure 8.19: Fiber based EMG simulation for the upper arm (biceps) model: Simulated surface EMG signals for the different fat layers shown in Fig. 8.18.

How To Reproduce

The simulations in this section use the examples `examples/electrophysiology/fibers/fibers_emg` and `examples/electrophysiology/fibers/fibers_fat_emg` with the variables file `20mus_fat_comparison.py`.

The scenario data that are necessary to run the simulations are given in the repository at github.com/dihu-stuttgart/performance in the directory `opendihu/18_fibers_emg`. The main scripts that runs the simulations for the two sections are the following:

```
./run_single_MUs.sh
./run_compare_fat_layer.sh
```

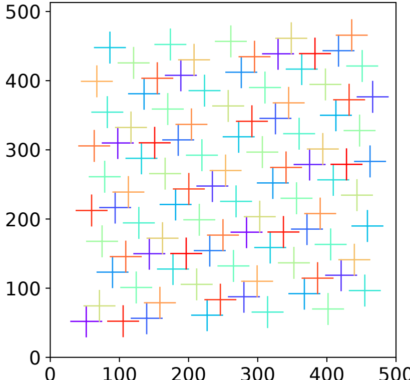


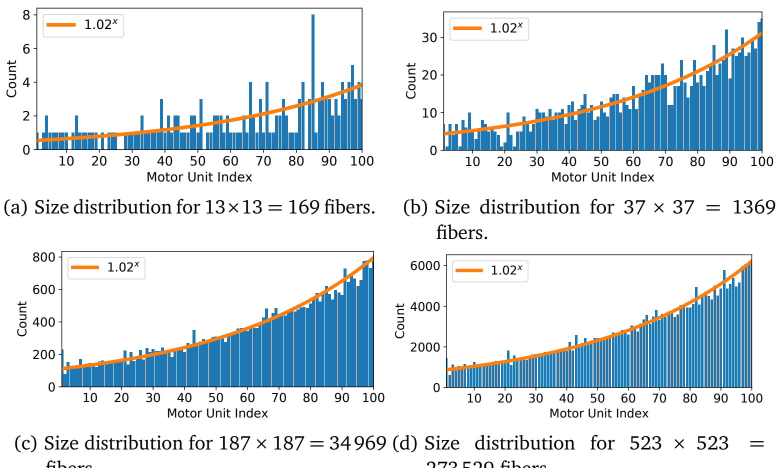
Figure 8.20: Fiber based EMG simulation for the upper arm (biceps) model; study of different mesh widths, MU territory center points. The shown center points of the 100 motor units are used in all different scenarios within the study of different mesh widths.

8.4.4 Effects of the Mesh Width on the Electromyography Signal

One goal of our simulation studies is to evaluate the required mesh width and the necessary number of fibers to obtain accurate simulation results of surface EMG signals. Experimental studies reveal a large variation in the number of muscle fibers in a real biceps brachii muscle. MacDougall et al. estimate in vivo numbers for elite and intermediate bodybuilders and untrained control subjects and find comparable numbers for these groups [Mac84]. They determine $278.5 \pm 60.7 \times 10^3$ muscle fibers for the group untrained subjects. Thus, we simulate scenarios with different 3D mesh resolutions and numbers of fibers up to the realistic number of 273 529. By comparing the obtained simulation results, we can determine if certain effects are only visible for high resolutions.

We consider a scenario with 100 MUs and increase the spatial resolution and the number of processes that execute the computation on the supercomputer Hawk at the High Performance Computing Center Stuttgart. Each compute node consists of two AMD EPYC 7742 processors with 64 cores each, a clock frequency of 2.25 GHz and 256 GB memory per node.

Our simulated scenarios consider between 1369 and 273 529 fibers. The specified number of 100 MUs has to be assigned to these numbers of fibers for each scenario. We use the method 1a of the algorithm described in Chap. 4. The MU territories are centered



(a) Size distribution for $13 \times 13 = 169$ fibers.

(b) Size distribution for $37 \times 37 = 1369$ fibers.

(c) Size distribution for $187 \times 187 = 34\,969$ fibers.

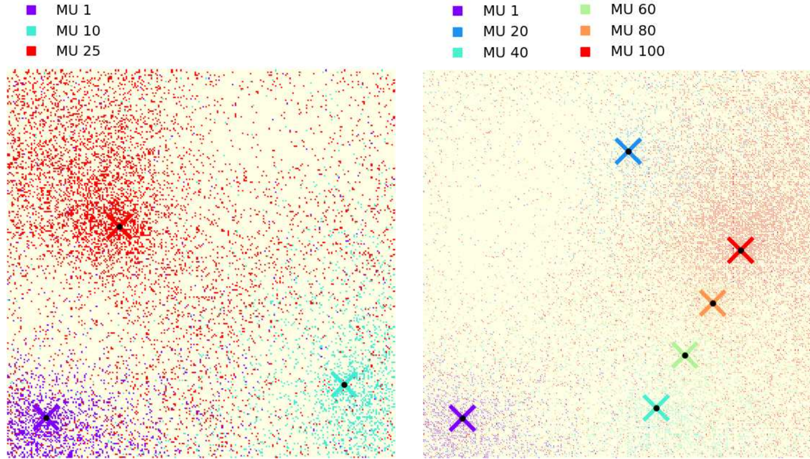
(d) Size distribution for $523 \times 523 = 273\,529$ fibers.

Figure 8.21: Fiber based EMG simulation for the upper arm (biceps) model; Distribution of the sizes of the 100 MUs in the scenarios with different number of fibers.

around quasi-randomly generated center points, as shown in Fig. 8.20. It can be seen that the MU territory center points are homogeneously distributed in space.

For every fiber, the algorithm assigns a MU with a close center point with higher probability than a MU whose center is located further away. The total number of fibers per MU is progressing exponentially for the MUs from 1 to 100. The progression is described by an exponential function with basis 1.02. Figure 8.21 shows the MU size distributions for four scenarios with increasing numbers of fibers from 169 to 273 529. For 169 fibers in Fig. 8.21a, not all 100 MUs get associated with a fiber. Further, it can be seen that the error of the actual size distribution to the exponential function decreases with increasing number of fibers. For the largest scenario in Fig. 8.21d, the MU sizes range from 602 to 6097 fibers.

The number of approximately $3 \cdot 10^6$ fibers in the largest scenario matches the realistic number in a biceps muscle [Mac84]. The number of MUs can be higher in reality, approximately by a factor of 5 [Fei55; Mac06]. Thus, the modeled MUs in this scenario can be seen as a combination of multiple real MUs. Especially the smallest MUs, which in reality can consist of only some dozens of fibers, are lumped by the first few MUs in our scenario. We restrict the number of MUs to 100 to be able to simulate the same problem also with smaller resolutions, e.g., with only 169 fibers.



(a) Partial problem with a quarter of the whole set of fibers and 25 MUs, which occurs in the algorithm 1a described in Chap. 4. Only three MUs are shown. The MU territory centers are indicated by crosses.

(b) The resulting assignments of MUs to fibers after four parts similar to (a) have been combined, here only shown for six MUs.

Figure 8.22: Fiber based EMG simulation for the upper arm (biceps) model; association of MUs to the fibers. The square domain corresponds to a cross-section in the muscle, every colored point is one fiber, and the color corresponds to the MU.

As described in Chap. 4, the MU assignment algorithm ensures that neighboring fibers are part of different MUs, by splitting the assignment problem for the given set of fibers into four smaller problems and then interleaving the results of the four parts. Figure 8.22a shows the first of these four parts, where 25 MUs are associated to a subset of the fibers for the largest scenario with 273 529 fibers. It can be seen that the three visualized MUs are largely clustered around their MU territory centers.

The final association of fibers and MUs is given in Fig. 8.22b. Six selected MUs are shown, of which the first, MU 1, corresponds to the first MU in Fig. 8.22a. The figure shows that the fibers, especially the ones of the larger MUs, are distributed far across the muscle. Comparing the smallest MU, MU 1, with the largest MU, MU 100, gives an impression of the MU size differences in this scenario.

The numerical parameters of the simulations are the same as in the last section. The

#fibers	3D stride		2D surface mesh	3D dofs (k=1000)	0D dofs	#proc.	#comp. nodes
	x, y	z					
$37^2 = 1369$	2	8	19 × 186	67 k	8109 k	144	3
$67^2 = 4489$	2	4	34 × 371	428 k	26 592 k	448	7
$109^2 = 11\,881$	2	3	55 × 495	1497 k	70 383 k	1152	18
$187^2 = 34\,969$	2	2	94 × 741	6547 k	207 156 k	3600	57
$277^2 = 76\,729$	2	1	139 × 1481	28 614 k	454 542 k	7744	121
$523^2 = 273\,529$	2	1	262 × 1481	101 661 k	1620 M	26 912	421

Table 8.1: Fiber based EMG simulation for the upper arm (biceps) model; Parameters of spatial discretization and parallel partitioning. The 3D stride refers to the stride with which the 3D mesh is generated from the 0D points. The 2D surface is the output of the EMG and corresponds to one face of the 3D mesh.

scenario is computed for a simulation time span of 1 s. The MUs are activated in a ramp every 2 ms such that all MUs are active after 200 ms. The fiber radius and the stimulation frequency for the MUs are exponentially distributed with basis 1.02, similar to the MU size. The fiber radius increases from 40 µm to 55 µm, and the stimulation frequency decreases from 24 Hz to 7 Hz for MUs 1 to 100. A random frequency jitter of 10 % is assumed.

The surface to volume ratio A_m of the membrane is determined by assuming a cylindrical shape and can be computed from the fiber radius r as $A_m = 2/r$ [Klo20]. We model 70 % slow twitch and 30 % fast twitch fibers. Accordingly, the membrane capacitance C_m is set to $C_m = 0.58\,\frac{\mu\text{F}}{\text{cm}^2}$ for the 70 smallest MUs and to $C_m = 1\,\frac{\mu\text{F}}{\text{cm}^2}$ for the 30 largest MUs.

Table 8.1 lists the spatial discretization and parallel partitioning parameters. The first column shows the number of fibers. Their number increases, however, the mesh resolution of every 1D fiber mesh stays constant at 1480 elements per fiber. The stride that defines the 3D mesh is given in the second and third columns. The stride in radial direction of the muscle, i.e., in the x and y coordinate directions, stays constant. Because the fiber density increases, the 3D mesh is refined accordingly. The stride along the fibers, i.e., in z direction is reduced, such that the mesh widths of the 3D mesh in all three coordinate directions remain balanced.

The resulting EMG recordings of each simulation are described by 2D meshes, which contain the values of the 3D muscle meshes without fat layer on the surface at one side of the muscle. The fourth column in Tab. 8.1 lists the dimensions of these surface meshes.

The next two columns list the number of dofs in the 3D mesh and the number of dofs in all fibers. For these scenarios, it is not practical to output the 3D mesh or the 1D fiber meshes in regular time intervals, because this would produce large amounts of data that could hardly be processed. Instead, we only output the 2D surface mesh in the ParaView format every 10 ms.

The last two columns in Tab. 8.1 show the numbers of processes and compute nodes that are used on Hawk. One compute node contains 128 physical cores, four cores share a 16 MiB level three (L3) cache. However, we decide to use only 64 cores per compute node, i.e., two cores per L3 cache, because measurements showed that this reduces the overall computation times more than it increases the runtime due to the decreased parallelism. The total computation time of this scenario with a timespan of 1 s is 2 h 20 min for the scenario with 76 729 fibers and 7744 processes.

Figures 8.23 and 8.24 show the resulting surface EMG signals for different resolutions. The color visualizes the value of the extracellular potential ϕ_e according to the shown color bar. Because of sign conventions in the definitions of the electric potentials, the spikes in the EMG signals, which result from the action potentials, are negative.

The resulting electric potential in Fig. 8.23 exhibits different regions of higher activation that move over time from the center of the muscle towards its ends. The size of these regions at time $t = 179.5$ ms decreases from Fig. 8.23a to Fig. 8.23d as the mesh width decreases. Dark-colored strong signals can be seen, which mainly correspond to fibers that are located directly underneath the shown muscle surface. Apart from these strong signals, also weaker artifacts occur, which are shown in yellow and orange colors. They result from the superposition of several fibers of the same or different MUs. The number of recognizable weak signals is higher for the simulations with higher numbers of fibers and finer mesh resolution.

The four scenarios in Fig. 8.23 share the material parameters, territory centers and relative size distributions of the 100 MU and the activation scheme. However, the location of the neuromuscular junctions is determined randomly and varies between the scenarios. Therefore, the resulting EMG signals are not refined images of each other. However, a similarity of activated regions on a coarse scale can be observed in all scenarios.

Figure 8.24 shows two more scenarios with many fibers at times of $t \approx 1$ s and $t \approx 0.4$ s. The scenario in Fig. 8.24a simulates approximately $76 \cdot 10^3$ fibers, which is in the order of a third of the realistic number of fibers in the biceps muscle. This scenario can also be interpreted as only activating a third of the available fibers in the muscle, resulting in the respective lower percentage of maximum voluntary contraction force.

(a) 1369 fibers. (b) 4489 fibers. (c) 11 881 fibers. (d) 34 969 fibers.

Figure 8.23: Fiber based EMG simulation for the upper arm (biceps) model; simulated surface EMG signals for different numbers of fibers and different mesh widths of the 3D mesh, see Tab. 8.1 for details. The same color coding of the EMG signal ϕ_e is used in the four scenarios.

Figure 8.24b shows the scenario where a realistic number of $273 \cdot 10^3$ fibers was simulated. The computational effort for these two scenarios can only be tackled with High Performance Computing. The last two rows of Tab. 8.1 show that 121 and 421, respectively, compute nodes were used for the computations.

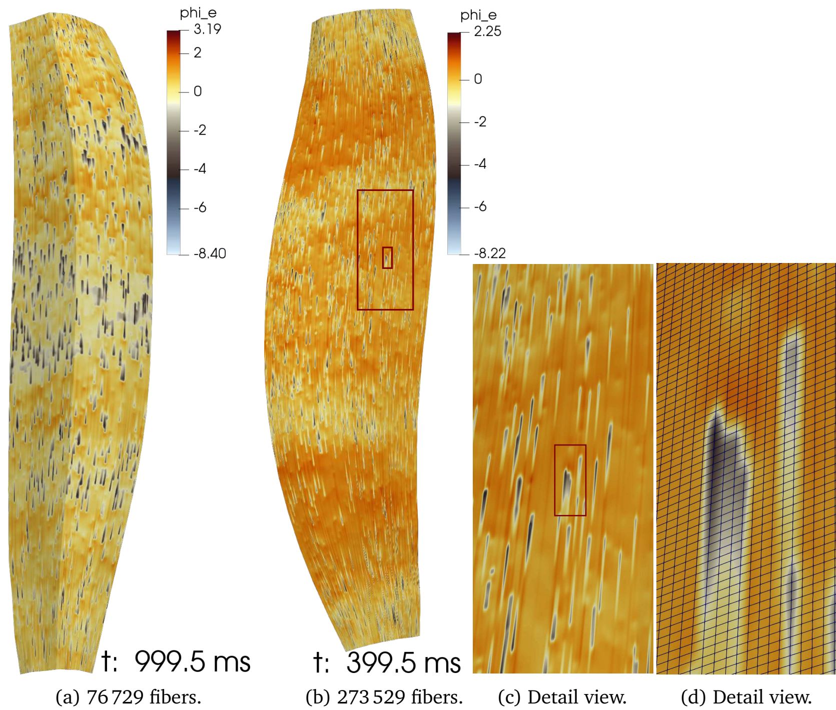
Figures 8.24c and 8.24d present details of the simulated surface EMG of the scenario in Fig. 8.24b. The cut-outs are indicated by the red boxes in Figures 8.24b and 8.24c. Figure 8.24d also visualizes the elements of the 2D and 3D meshes. In both scenarios of Fig. 8.24, the 3D mesh is as finely resolved in z direction (vertical in Fig. 8.24d) as the muscle fibers. In transversal direction (horizontal in Fig. 8.24d), the element sizes are twice as large as the spacing between the fibers.

The right side of Fig. 8.24d shows two almost aligned action potentials that propagate towards the top of the image. The upper action potential originates from a fiber that is located at the center between the element boundaries. Its electric potential is distributed to two adjacent nodes on the surface mesh, having the same activated values. In contrast, the lower action potential results from a fiber that is directly located on the element boundaries. It can be seen that the activation on the surface decays rapidly in transverse (horizontal) direction, as the neighbor elements already almost exhibit the same electric potential as the background level. This underlines the importance to use finely resolved meshes to accurately represent surface EMG signals.

How To Reproduce

Use the following commands to run the EMG simulation of the biceps muscle with fat layer and electrodes:

```
cd $OPENDIHU_HOME/examples/electrophysiology/fibers/fibers_fat_emg/
  ↪ build_release
mpirun -n 16 fibers_fat_emg ../settings_fibers_fat_emg.py 50mus.py
cd out/50mus
plot_emg.py ./electrodes.csv ./stimulation.log 25900 26000    # plot
  ↪  the result, here for time span 25.9s - 26s
```



(a) 76 729 fibers. (b) 273 529 fibers. (c) Detail view. (d) Detail view.

Figure 8.24: Fiber based EMG simulation for the upper arm (biceps) model; simulated surface EMG signals with realistic fiber counts, continued from Fig. 8.23. Figure (a) shows a scenario with 76 729 fibers, which is approximately a third of the realistic number for the biceps muscle. Figures (b)-(d) show the result with a realistic number of 273 529 fibers. (c) shows a detail view of (b) indicated by the outer red box. (d) shows another zoomed in view of (b) and (c), also indicated by the red boxes.

8.4.5 Simulation of EMG electrodes

While the surface EMG simulation results as presented in the last section in Fig. 8.23 are suited for insights into the temporal and spatial variation of the electric potential, real experiments are constraint to capture values at the discrete locations of electrodes. For some applications such as the evaluation of EMG decomposition algorithms, it is beneficial to obtain simulated values at electrode locations.

One possibility would be to extract nodal values from the simulated surface meshes to simulate electrodes. However, the distance between the nodes in the mesh is not constant in the whole mesh, whereas EMG electrode arrays have a fixed inter-electrode spacing. We, therefore, follow a different approach and allow to directly specify a grid of electrodes close to the muscle surface. These points are then mapped onto the surface of the muscle and the respective values are calculated by evaluating the finite element interpolant at the respective locations.

In OpenDiHu, a 2D grid of surface electrodes can be defined in the Python settings file by specifying the grid parameters and inter electrode distances. As a result, the simulation distributes the electrodes to the processes according to the parallel partitioning of the 3D mesh, evaluates the computed EMG values at the respective locations and outputs them in a single text file of comma separated values.

Figure 8.25 shows simulation results of the fiber based electrophysiology model with 49 fibers, fat layer and an array of 12×32 electrodes. The electrodes are visualized as spheres. The muscle fibers below the fat layer are colored according to the transmembrane voltage V_m. Only the upper surface of the fat layer is shown and colored according to the extracellular potential ϕ_e. The EMG electrodes capture the values of the scalar field ϕ_e at their locations. The color coding for the electrodes has a different *EMG* color scale to make the resulting signals more distinguishable. Two activated bands across the muscle surface can be seen, which are also present in the electrode values.

To visually evaluate the simulated EMG signals at the electrodes, OpenDiHu provides utilities to create the visualizations shown in Fig. 8.26. Figure 8.26a shows a single frame from an animation. On the upper right, the grid of electrodes is displayed. The EMG signal at the electrodes is given by the colored tiles and changes over time. At the bottom of the image, the activation times of the MUs are visualized. Every horizontal line corresponds to one MU. The colored markers indicate when the respective MU fires. As the shown example visualizes data for 40 s, the individual firing times are not distinguishable. In the animation, a vertical bar moves over the time axis and indicates the current simulation

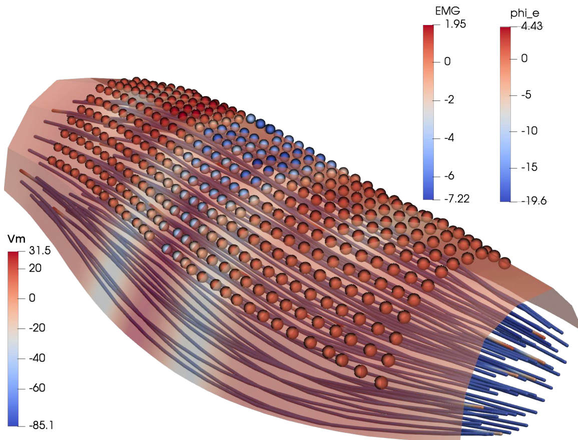


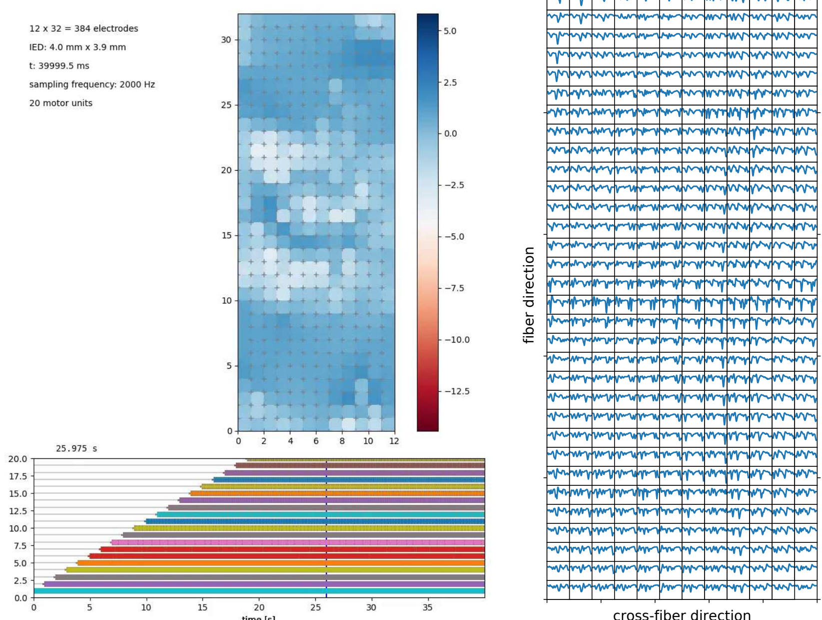
Figure 8.25: Fiber based EMG simulation for the upper arm (biceps) model; simulation of surface EMG and capturing electrodes. The scenario contains 49 muscle fibers, a fat layer, of which only the surface is shown, and a grid of 12×32 equidistant electrodes.

time. The picture displays the EMG values at time $t = 25.975\,\text{s}$. The upper left of the image shows a text with static information about the dataset, containing the electrode grid size, the inter electrode distance (IED), the end time, the sampling frequency of the electrodes, i.e., the frequency with which the computed EMG signals values are stored to the output file, and the number of MUs.

Figure 8.26b shows another, static visualization of simulated EMG data. The diagram contains boxes for all electrodes in the 12×32 grid. The value of the EMG signal is plotted over time in every box for the respective electrode. Figure 8.26b visualizes the data of Fig. 8.26a for the time interval $[25.9\,\text{s}, 26\,\text{s}]$. The diagram enables experts to visually identify propagating action potentials from the tile columns. The propagation velocity of the action potentials can be estimated from the time shift of matching spikes in vertically adjacent boxes.

8.4.6 Decomposition of Surface EMG Signals

Surface EMG recordings are a valuable tool to gain insights into the neuromuscular system. They are used, e.g., for the diagnosis of muscular disorders and in clinical studies that aim to advance biomedical understanding.



(a) Snapshot of an animation of the simulation results that can be generated with the utility of OpenDiHu. The top part visualizes the current values of the EMG electrodes at time $t = 25.975\,\text{s}$. The bottom part shows the MU activation ramp, each colored line shows to the firing time range of one MU.

(b) Surface EMG at 12×32 electrodes in the time range $[25.9\,\text{s}, 26\,\text{s}]$.

Figure 8.26: Fiber based EMG simulation for the upper arm (biceps) model; results of surface EMG obtained at simulated electrodes.

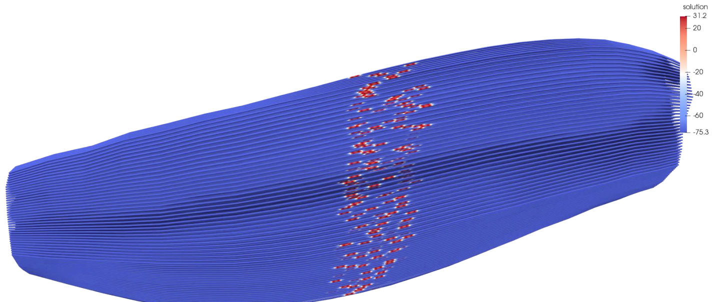


Figure 8.27: Fiber based EMG simulation for the upper arm (biceps) model; a simulation result that reveals the locations of the neuromuscular junctions. The figure depicts 1369 fibers after 1 ms, which have initially been stimulated at the neuromuscular junction. The color coding corresponds to the membrane potential V_m, which has a positive value near the points of stimulation.

As described earlier, the EMG signals on the skin surface originate from the activated muscle fibers. Effects from volume conduction of action potentials on all muscle fibers are superpositioned and contribute to the EMG signal. The scaling of the contributions to the overall signal depends on several factors such as the distance of the fibers to the skin surface. As all fibers in the same MU get activated simultaneously, each MU‘s contribution shows a characteristic “shape” in the resulting surface EMG signal. This shape is influenced by the number and location of the muscle fibers relative to the electrodes and the location of the neuromuscular junctions.

In our simulation, the location of the neuromuscular junctions is chosen pseudo-randomly (but deterministic) during initialization in the central 10 % of every muscle fiber. Figure 8.27 shows the state of a simulation with 1369 fibers at $t = 1$ ms, where all fibers have been activated at $t = 0$ ms. The color coding indicates the potential V_m of the membrane, which at the shown time has only depolarized near the locations of the neuromuscular junctions.

Methods exist that seek to decompose the surface EMG signal into the contributions of the individual MUs. Given a surface EMG recording, such methods output a number of recovered MUs and their firing times. In our simulation studies, all relevant information is available that determines the EMG signal resulting from MU activity: the location of the fibers and their association to MUs, the positions of the neuromuscular junctions and the innervation pulses for each MU. Thus, our simulation can be used to validate and evaluate EMG decomposition methods.

One popular EMG decomposition method is *Gradient Convolution Kernel Compensation* (gCKC) [Hol07a; Hol07b], which, in the following, will be outlined and then applied on simulated data.

Most decomposition methods, including the gCKC algorithm, assume that the EMG signal at an electrode is composed of the convolutional mixture of the activity of N MUs. The activity of each MU $k \in \{1,\dots,N\}$ is described by the innervation pulse trains, which activate the fibers of MU k, given as a point process of neural inputs at stimulation times φ_r. The source signal s_k in the muscle, which represents the effect of MU k is described as a spike $s_k(t) = \sum_r \delta(t-\varphi_r)$, where δ is the dirac delta function.

The vector of observed EMG value $\mathbf{x} \in \mathbb{R}^m$ at a time t is composed of the temporal convolution over L time-shifted sources $\mathbf{s}$ and a term $\boldsymbol{\omega}$ of additive Gaussian noise:

$$\mathbf{x}(t) = \sum_{\ell=0}^{L-1} \mathbf{H}(\ell)\,\mathbf{s}(t-\ell) + \boldsymbol{\omega}(t).$$

Here, $\mathbf{H}$ is the $m \times n$ mixing matrix for m observations and n MU sources and $\mathbf{s} = (s_k)_{1,\dots,n}$ is the vector of source signals. The sum over L previous values in this convolutive mixture can be reformulated by moving the summation into the matrix-vector product. The dimensions of the matrix $\mathbf{H}$ and the vector $\mathbf{s}$ are extended accordingly. An optimization problem yields the separation vectors, with which the innervation pulse trains φ_r of the MUs can be recovered from the recorded EMG signals $\mathbf{x}$. The gCKC algorithm determines the inverse effect of applying the unknown mixing matrix by solving a derived optimization problem using a gradient descent scheme.

The gCKC decomposition algorithm is implemented in the DEMUSE software, a commercial, MATLAB based tool that allows automatic and semi-automatic EMG decomposition [Hol08]. In collaboration with Lena Lehmann from the *Institute of Signal Processing and System Theory* and the *Institute for Modelling and Simulation of Biomechanical Systems*, we evaluated the performance of gCKC decomposition on simulated surface EMG signals.

We simulate fiber based electrophysiology scenarios with fat layer and 1369 fibers using the same model parameter as in Sec. 8.4.4. In the first scenario, a fat layer with thickness of 1 cm is modelled. The simulated EMG signal is sampled in an electrode array with a frequency of 2 kHz and a grid size of 12×32 fibers, as shown in Fig. 8.26.

Figure 8.28 shows the firing times of the 20 MUs in the first 10 s. The different MUs are initially activated every 100 ms to generate the shown “ramp” activation pattern, which

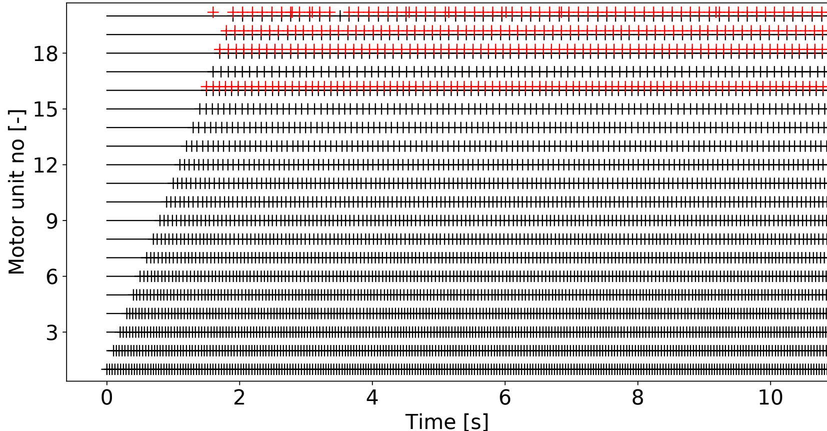


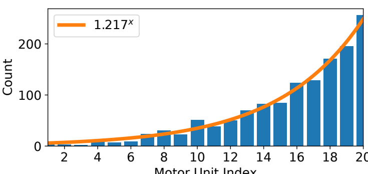
Figure 8.28: Validation experiment for EMG decomposition based on fiber-based simulations of the biceps brachii muscle: Match of EMG decomposition results with simulated data. The true firing pattern over time for the 20 MUs in the simulation is shown by black markers. The recovered firing times of the gradient convolution kernel compensation algorithm are given by the red markers. The algorithm detected the four MUs 16, 18, 19 and 20.

later helps to identify the recovered MUs from the decomposition. From $t = 1.8\,\text{s}$ on, all MUs fire with their respective constant frequency, subject to jitter values of 10 %.

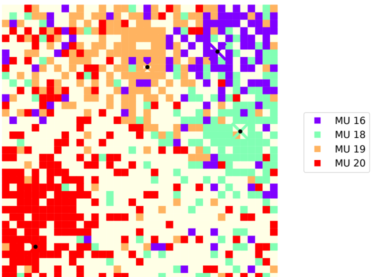
In this first scenario, the gCKC decomposition algorithm was applied on the first $t = 40\,\text{s}$ of simulated EMG data. The preconfigured algorithm in DEMUSE was used without manual intervention. While the simulated EMG recording consisted of an electrode grid of 12×32 fibers, only a rectangular subset of 5×13 channels at the lower center of the grid was used for the decomposition to mimic a realistic electrode array size.

In reality, some of the recorded channels may measure invalid data due to inappropriate surface contact of the electrodes, noisy signals at the particular measurement location or other experimental difficulties. The DEMUSE tool can automatically detect such channels and discard the corresponding data from the decomposition scheme. Despite our simulation does not contain invalid channels, the DEMUSE software discarded four of the 65 simulated channels.

Figure 8.28 shows the innervation pulses that were detected by DEMUSE as red vertical markers. A time span of 50 s was simulated of which only the first 11 s are visualized in Fig. 8.28. DEMUSE found four MUs in this scenario, i.e., 20 % of the 20 simulated



(a) Number of fibers of the 20 MUs in this scenario, following an exponential progression with basis 1.217.



(b) Spatial location of the fibers of the four MUs that were recovered by the EMG decomposition algorithm, indicated by different colors. The territory center points of the MUs used in the generation algorithm are indicated by black dots. The yellow background area corresponds to other MUs, which were not detected.

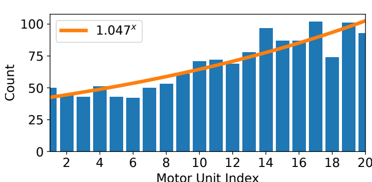
Figure 8.29: Validation experiment for EMG decomposition based on fiber-based simulations of the biceps muscle: Association of the fibers with motor units for the first scenario with 20 MUs, given in Fig. 8.28.

MUs. The recovered MUs were identified in the set of simulated MUs by matching the average firing frequency and the activation onset time in the ramp scheme. A first visual comparison with the original stimulation times given by the black markers shows a good agreement.

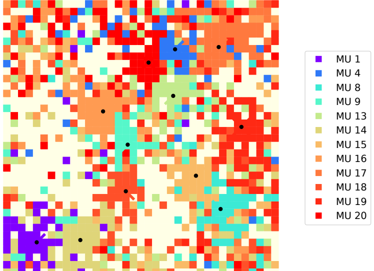
In this scenario, the association of fibers with MUs followed an exponential MU size progression with a basis of approximately 1.2, as shown in Fig. 8.29a. The smallest MU contained two fibers and the largest MU had 256 fibers. The method 1 described in Sec. 4.2 was used to generate the association between fibers and MUs.

Figure 8.29b depicts the location of the four MUs that were detected by DEMUSE. The detected MUs have the indices 16, 18, 19 and 20 and correspond to four of the five largest MUs. It can be seen that MUs 18 to 20 are located mainly in the upper half of the muscle cross-section, in proximity to the electrode array at the top of the diagram. The MU with the most fibers, MU 20, was detected by the decomposition algorithm even though it is located at the lower left of diagram at a large distance to the skin surface.

Two further scenarios were simulated with the same parameters as the first scenario in Fig. 8.28, but instead with 50 and 100 MUs. In these datasets, DEMUSE was able to



(a) Progression of MU sizes. The number of fibers per MU is more balanced in this scenario than in Fig. 8.29a.



(b) Spatial fiber arrangement of the MUs that were detected by the EMG decomposition algorithm, with their center points given by black dots.

Figure 8.30: Validation experiment for EMG decomposition based on fiber-based simulations of the biceps brachii muscle: Association of the fibers with motor units for the second scenario with 20 MUs, given in Fig. 8.31.

detect 8 and 12 MUs, which corresponds to 16 % and 12 %.

Moreover, another scenario with 20 MUs was computed, but the fat layer was varied to have a thickness of only 2 mm instead of 1 cm. In addition, the association scheme between MUs and fibers was changed to the one shown in Fig. 8.30. The exponential distribution of MU sizes only varied between 42 and 102 fibers per MU, corresponding to a basis in the exponential function of approximately 1.05 instead of 1.2.

Figure 8.31 shows the results of the EMG decomposition with the gCKC algorithm for this second scenario with 20 MUs. DEMUSE successfully decomposed the signal into 13 MUs, corresponding to 65 % of the 20 simulated MUs. DEMUSE also determined two additional MUs, which we do not consider part of the set of successfully recovered MUs. The first dataset only consists of ten innervation pulses, and the second pulse train contains high frequency oscillations. In this scenario, the software marked only one EMG recording channel as invalid, which means that more data were considered by the decomposition algorithm than in the first scenario with 20 MUs.

Similar to the previously presented scenario, the larger MUs were detected with a higher probability than the smaller MUs. In this scenario, the eight largest MUs were successfully found. Figure 8.30b shows the spatial arrangement of the detected MUs. The area of the muscle cross-section that is occupied by undetected MUs is again located more distantly to the skin surface at the upper boundary. However, the recovered MUs 1

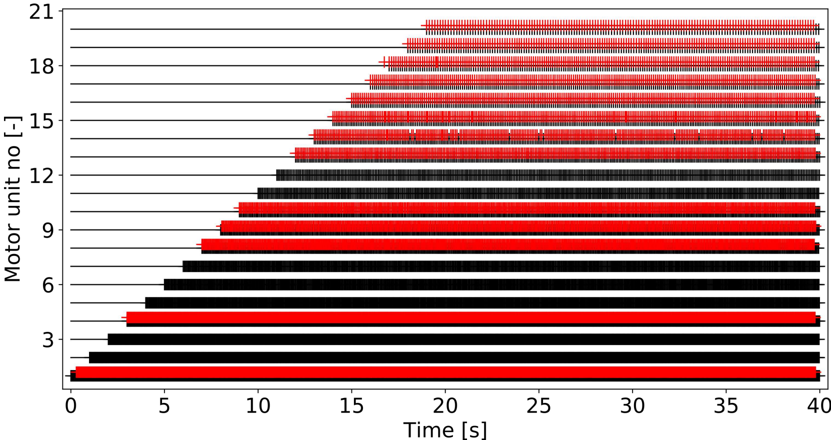
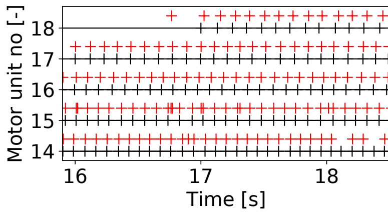


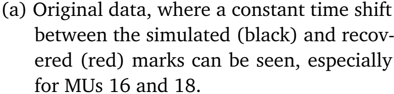
Figure 8.31: Validation experiment for EMG decomposition based on fiber-based simulations of the biceps muscle: Activation pattern for the second scenario with 20 MUs. The activation times used in the simulation are shown as black markers, the recovered activation pulses of the EMG decomposition algorithm are shown as red markers.

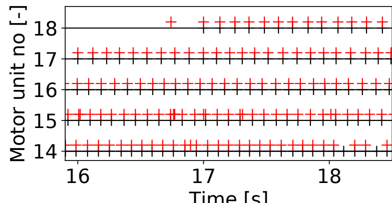
and 14 are nevertheless located at the lower boundary, i.e., in the most distant area from the EMG electrodes.

Next, we evaluate the quality of the innervation pulse trains that were recovered by the gCKC algorithm in our scenarios. We compare the stimulation times calculated by DEMUSE with the stimulation times of the simulation. Figure 8.32a shows an excerpt of the detected pulse trains of the second scenario with 20 MUs in Fig. 8.31, where the gCKC algorithm recovered 13 MUs. For same MUs, we observe that the recovered stimulation times are consistently shifted in time. This effect is especially visible for MUs 16 and 18.

The reference times given by the black markers in Fig. 8.32a correspond to the times when the fibers were stimulated in the simulation in OpenDiHu. The detected MU activations in DEMUSE, however, correspond to the times when the MU action potential shapes in the EMG recording reached their maximum. Moreover, the exact times when particular MUs reach particular EMG electrodes depend on the distance of the electrodes to the innervation points of the MUs. The further the electrodes are away from the neuromuscular junctions along the muscle, the higher is the delay of the recorded spikes to the corresponding innervation pulses. Thus, the constant time shifts in the pulses detected




(a) Original data, where a constant time shift between the simulated (black) and recovered (red) marks can be seen, especially for MUs 16 and 18.



(b) Same data as in (a) with applied time offset correction. The applied time shifts for MUs 16 and 18 are −12.9 ms and −24.9 ms, respectively.

Figure 8.32: Validation experiment for EMG decomposition based on fiber-based simulations of the biceps muscle: Excerpts of the detected firing times of MUs 14 to 18 in the second scenario with 20MUs. The stimulation times of the simulation are given by black markers, the recovered times are visualized by red crosses.

by the gCKC algorithm are valid and have to be accounted for in the evaluation of the decomposition performance.

We correct for these time shifts by adding constant time offsets Δt_k to the recovered innervation pulse trains. For every MU k, the algorithm finds the matching pairs of simulated and recovered pulses and optimizes the value of Δt_k such that the time differences in these pairs after shift correction get minimal.

Figure 8.32b shows the same extract of MU activity as in Fig. 8.32a with applied time offsets. The time offsets for MUs 14 to 18 in this example are given as

$$\Delta t_{14} = -2.4\,\text{ms}, \qquad \Delta t_{15} = -1.1\,\text{ms}, \qquad \Delta t_{16} = -12.9\,\text{ms},$$
$$\Delta t_{17} = -2.9\,\text{ms}, \quad \text{and} \quad \Delta t_{18} = -24.9\,\text{ms}.$$

Figure 8.32b shows that the recovered pulses now match the simulated data very well. The non-matching pulses are clearly false positive detections.

To compare the recovered MU times between the scenarios, we evaluate metrics such as the rate of agreement. The MU firing times in the simulation serve as the ground truth, to which we compare the recovered MU times. We identify true positive (TP), false positive (FP) and false negative (FN) recovered pulses, depending on whether a matching time to a recovered pulse can or cannot be found in the simulation data within a tolerance of $\varepsilon = 5\,\text{ms}$. The rate of agreement (RoA) between the gCKC algorithm output and the

ground truth data is then computed by

$$\mathrm{RoA} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP} + \mathrm{FN}}.$$

In the first scenario with 20 MUs in Fig. 8.28, the RoA for MUs 16,18 and 19 is above 99.7 % and slightly lower at 82.2 % for MU 20. Here, only 296 of the 334 detected pulses were true positives, corresponding to a precision of 88.6 %.

In the second scenario with 20 MUS presented in Fig. 8.31, all valid MUs except one have RoA values of above 94.5 %. Five MUs are even detected perfectly with 100 % rate of agreement. MU 9 is the only detected MU with a degraded RoA of approximately 57.9 %. However, the RoA improves to 98.1 %, if the tolerance ε for matching pulses is relaxed to 10 ms. This shows that the RoA metric also depends on a proper value for the tolerance ε, and that innervation pulse trains detected by DEMUSE can have varying accuracy in a range of less than 10 ms.

While the gCKC algorithm can be used for EMG decomposition of previously recorded signals in a controlled environment, it is less suited for real-time applications. The separation vectors that decompose the electrode signals and infer the MU innervation pulse trains can be computed in a training phase. However, their application on new data requires a certain history of previously captured signals to calculate the decomposed MU pulses. As a consequence, the predictions are delayed, which is usually undesirable in real-time applications. Furthermore, the system is sensitive to noisy data.

A fundamentally different approach to EMG decomposition is the use of sequence-to-sequence learning methods provided by recurrent neural networks. The authors of [Cla21] used a gated recurrent unit (GRU) network for this task. The network was trained using the output of the gCKC algorithm and was subsequently able to decompose surface EMG signals into innervation pulse trains. The approach was shown to be robust and to outperform gCKC for low signal-to-noise ratios.

To assess, whether our simulations of surface EMG can be used for the supervised learning of GRU networks for EMG decomposition, we tried in a first step to reproduce the studies of [Cla21], where the GRU is trained with the output of the gCKC algorithm. Additionally, we trained a GRU network directly on the simulated EMG data. These tasks were carried out in the masters project of Srijay Kolvekar and were supervised by Lena Lehmann and me. For details on the methods and results, we refer to the literature [Cla21] and the project report [Kol21].

In this project, the EMG decomposition of a GRU network trained with raw innervation pulse trains obtained from the gCKC algorithm, similar to the literature, showed many false positive and false negative predictions. However, a different setup using MU labels instead of raw pulse trains showed promising results. Every discrete point in time (according to the EMG sampling frequency) was either associated with the class of the currently active MU or with the background class, when no MU was activated at the time. This classification problem had a large class imbalance, as the background class was active for 86 % of the timesteps. The issue was mitigated by using class weights. The GRU network was trained with simulation data and yielded per-class rates of agreement of up to 72 % for the two scenarios with 20 MUs shown in Figures 8.28 and 8.31, i.e., with the test data set also generated by our simulation.

Figure 8.33 presents an excerpt of the resulting predictions of a GRU network that was trained with simulation results. We used the simulation of the second scenario with 20 MUs, which is shown in Fig. 8.31. The black markers in Fig. 8.33 indicate the stimulation times used in the simulation. The red markers correspond to the recovered times by the gCKC algorithm. Out of the shown MUs, only MUS 9 and 10 were recovered by the gCKC algorithm. The blue markers denote the GRU predictions. Correction of time offsets was performed for both the gCKC and GRU outputs.

Figure 8.33 shows the best agreement between the two prediction methods for MU 10 with a RoA of 99.6 % for the gCKC algorithm and 72.2 % for the GRU network. In contrast to the gCKC algorithm, the GRU network predicts firings for all MUs. However, the quality is only acceptable for MUs that could also be detected by the gCKC algorithm. For MUs 11 and 12, the RoA for the GRU network is around 30 %.

In future work, the decomposition performance of the GRU networks could be improved by using different training data. For example, the ramp activation in the training data could be replaced by constant tetanic stimulations. Moreover, different network architectures, such as convolutional recurrent neural networks could be investigated.

In conclusion, the gCKC algorithm is able to decompose artificially generated surface EMG signals. This means that our simulation can be used to evaluate the performance of EMG decomposition algorithms.

The number of detected MUs depends on the relation between MU sizes and on the distance of the MU territories to the electrodes. If the variance of the sizes of the activated MUs is small, such as in Fig. 8.30a, also MUs that are far away from the electrodes are detected. If, in the opposite case, the sizes of active MUs are distributed over a large range such as in Fig. 8.29a, only the largest MUs are detectable.

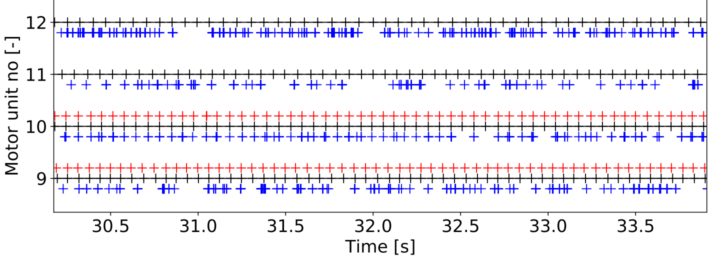


Figure 8.33: Comparison of innervation pulse train predictions of the gCKC algorithm (red), a GRU network (blue) and the ground truth data (black).

In addition, the amount of adipose tissue between the electrodes and the muscle influences the number of MUs that can be recovered. In our studies, the performance of EMG decomposition was lower for all scenarios with thicker fat layer than for the scenario with a thin fat layer.

The rate of agreement of the determined pulse trains of the DEMUSE software was above 95 % in most of the cases. Correspondingly, the rate of false positives was low. A time shift between the recovered times and the ground truth data was observed for some pulse trains, which can be explained with the delay from first activation to the onset of the EMG signal. As a result, the time shift was corrected for the rate of agreement measurement.

A proof-of-concept implementation of GRU networks showed promising performance for predicting MU firing times from artificial EMG recordings. The GRU network predicted firing times also for MUs that were not detected by the gCKC algorithm, however the rate of agreement was low for these MUs. In future work, the GRU decomposition method has to be refined to be comparative to the gCKC algorithm.

8.4.7 Simulation of Electrophysiology with a Phenomenological Fiber Model

Instead of the previously presented numerical model of action potential propagation on the muscle fibers, the respective physiological process can also be described with a phenomenological approach.

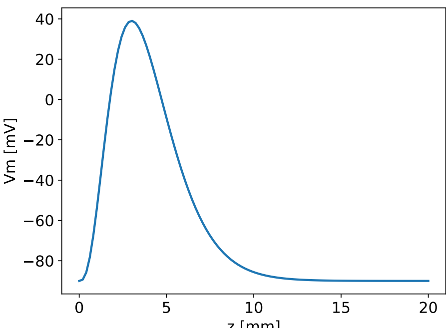


Figure 8.34: Phenomenological muscle fiber model: analytic model by Rosenfalck of an action potential shape on a 1D fiber.

The model formulated by Rosenfalck [Ros69] describes the action potential shape on a 1D domain by the following function:

$$G(z) = \begin{cases} 96\,z^3\,\exp(-z) - 9 & \text{for } z \geq 0, \\ -90 & \text{for } z < 0. \end{cases}$$

Figure 8.34 shows the graph of this function. The resulting value of G specifies the membrane voltage in millivolts. The coordinate $z = x + v\,t$ depends on the distance x to the neuromuscular junction on the fiber, the conduction velocity v and the time t after the last stimulation. In our simulation, we use the proposed propagation velocity of $4\,\frac{\text{m}}{\text{s}}$.

The advantage of using an analytic model is its fast calculation compared with the runtime of the numerical model. On the downside, such a model cannot accurately describe fatigue effects in tetanic stimulations or the action potential shape changing properties of more advanced subcellular models.

One use case of an analytic model is to study the effect of muscle fiber arrangements in a muscle volume on the surface EMG signal. In this case, the exact, possibly time-varying shapes of the motor unit action potentials are less important than the location and orientation of the fibers. We provide an exemplary scenario in OpenDiHu, which uses the Rosenfalck model on multiple 1D muscle fibers that are embedded in a 3D domain. As in the previous sections, the 3D bidomain model given by Eq. (5.9a) is coupled to the

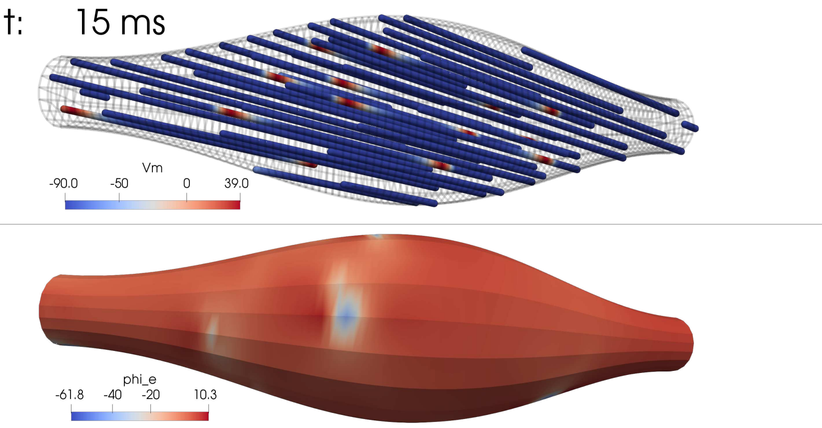


Figure 8.35: Simulation of EMG signals on the upper arm; artificial muscle geometry with a phenomenological model of action potential propagation. The upper image shows the muscle fibers, colored according to the transmembrane potential V_m. The lower image shows the extracellular potential ϕ_e on the surface. This scenario can be computed very fast and can, e.g., be useful to investigate the effects of different fiber orientations.

fibers and used to simulate EMG recordings on the surface.

In addition to the simplified model of action potential propagation, we use a simplified description of the muscle geometry. Figure 8.35 shows the artificial geometry, which is constructed by rotating a transformed sine curve around the z axis. Our program embeds the fibers automatically inside the volume and orients them according to specified spherical coordinates. The orientation angles, the number of fibers and the spacings between the fibers can be adjusted in the Python settings script of the simulation.

Figure 8.35 shows the location of the fibers inside the artificial muscle belly in the upper image and a simulation result of muscular activity at $t = 15\,\text{ms}$ in the lower image. The resulting EMG signal can be seen on the surface. This simulation scenario provides means to quickly study electrophysiology and generation of EMG for a generic muscle. Solving the model only consists of evaluations of the Rosenfalck function and repeated computation of the 3D model, but no further costly computations of numerical electrophysiology models. Moreover, no mesh file has to be generated and loaded, which simplifies the handling of the scenario.

How To Reproduce

The simulation in Fig. 8.35 can be started as follows.

```
cd $OPENDIHU_HOME/examples/electrophysiology/fibers/
  ↪ analytical_fibers_emg/build_release
./analytical_fibers_emg ../
  ↪ settings_analytical_fibers_emg_custom_geometry.py geometry_round
  ↪ .py
```

The options can be set in the `variables/geometry_round.py` settings file. Other artificial geometries are available by using other scripts under `variables`.

8.4.8 Conclusion

In the present section, surface EMG signals were computed using the fiber based electrophysiology model. We showed, how the qualitative nature of the signals depends on the spatial distribution of the MUs, and that the thickness of the body fat layer influences the smoothness of the recorded EMG signals. Furthermore, we investigated the effect of the mesh resolution and the number of fibers in the simulation. The results showed that only a high mesh resolution can resolve the small features in the EMG signal on the skin surface, which result from fiber action potentials. The number of these features in the result increased with the mesh width and, thus, the two finest discretizations with $77 \cdot 10^3$ $274 \cdot 10^3$ fibers yielded the most accurate results. As a consequence, High Performance Computing simulations are required, if an accurate 2D surface EMG signal should be computed.

Further studies with EMG decomposition algorithms showed another use case of our fiber based surface EMG simulations: They can be used as a validation tool for existing decomposition algorithms, and they can serve as a data generator to develop novel, data based decomposition methods. The decomposition software DEMUSE was tested, and we quantified the rate of agreement of its predictions with our simulation. Similarly, we evaluated a neural network based approach and compared its performance on the simulated data with the previous method.

8.5 Simulation of the Multidomain Model

The multidomain model is an alternative to the fiber based electrophysiology model discussed in the last section. As introduced in Sec. 5.1.4, it does not explicitly resolve muscle fibers, but considers activity in the muscle domain in a homogenized view. On every point in the 3D muscle mesh, separate values V_m^k of the transmembrane potential exist for every MU $k \in \{1, \dots, N_{\text{MU}}\}$, in addition to the value ϕ_e for the extracellular electric potential. The computational domain considers the muscle volume Ω_M and the body domain Ω_B, which represents adipose tissue on top of the muscle. To solve the multidomain model, a large linear system has to be solved in every timestep, as described in Sec. 5.3.5.

In the following, Sec. 8.5.1 discusses the model setup for a scenario with four MUs. Section 8.5.2 demonstrates a larger simulation scenario with 25 MUs, which can be used to simulate surface EMG signals. We discuss differences between the multidomain approach and the fiber based electrophysiology model in Sec. 8.5.3.

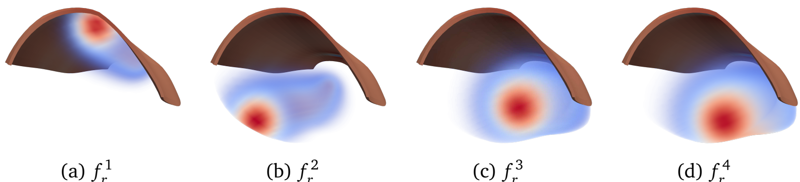
8.5.1 Components of the Computational Model

In the following, we consider a multidomain simulation with muscle and body fat domains and four MUs. We use a muscle mesh with $16 \times 16 \times 74 = 18\,944$ elements and linear finite element ansatz functions and a fat mesh with $32 \times 4 \times 74 = 9472$ elements, which are partitioned to 128 subdomains for 128 processes.

The scenario uses the following electric conduction tensors $\boldsymbol{\sigma}_i$ and $\boldsymbol{\sigma}_e$ for the intracellular domain and the extracellular domain, respectively:

$$\boldsymbol{\sigma}_i = \begin{pmatrix} 8.93 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \frac{\text{mS}}{\text{cm}}, \qquad \boldsymbol{\sigma}_e = \begin{pmatrix} 6.7 & 0 & 0 \\ 0 & 6.7 & 0 \\ 0 & 0 & 6.7 \end{pmatrix} \frac{\text{mS}}{\text{cm}}. \tag{8.5}$$

In this scenario, we use the subcellular model of Hodgkin and Huxley [Hod52a] and solve it using Heun's method. We discretize the multidomain equations using the Crank-Nicolson scheme with $\theta = \frac{1}{2}$. We solve the resulting linear system of equations by a GMRES solver with the parallel incomplete LU factorization preconditioner *Euclid* [Hys01] from the HYPRE package [Fal02]. A tight residual norm tolerance of 10^{-15} is used in the abortion criterion of the GMRES solver. Such a low tolerance is required, as, for higher tolerances, spurious artificial stimulations can be observed. Timestep widths of



(a) f_r^1 (b) f_r^2 (c) f_r^3 (d) f_r^4

Figure 8.36: Simulation of electrophysiology with the multidomain model: Value of the occupancy factors f_r^k for MUs 1 to 4. The color coding encodes the maximum value by red color and the decreasing value along the radius by increased transparency and the color transition to blue color. The locations where the factor vanishes, $f_r^k = 0$, correspond to full transparency.

$dt_{\text{0D}} = dt_{\text{multidomain}} = dt_{\text{splitting}} = 10^{-3}$ ms are used. The computation for a simulation end time of $t_{\text{end}} = 20$ ms in this scenario takes approximately 27 min.

In the multidomain model, we have to specify which point in the 3D mesh belongs to which MU to which extent. This is achieved by the relative occupancy factors f_r^k for MU k. In our implementation, the occupancy factors are computed by Python code in the settings file of the simulation. The location of a MU in the 3D domain is specified by choosing a 1D muscle fiber, which is considered to be the center of the MU territory. This is possible, as the nodes in the structured 3D mesh can also be interpreted as a set of adjacent 1D fibers.

For every MU k, the occupancy factors $f_r^k(x,y,z)$ in every muscle cross-section with fixed z coordinate are defined by a radial function $f(|(x,y)^\top|/d(z))$, which reaches a configurable maximum value at the location of the specified fiber. The argument of the radial function is scaled by the diameter $d(z)$ of the muscle at the considered cross-section. The factor $f_r^k(x,y,z)$ is constant in longitudinal direction of the muscle (z axis). Before the simulation, all factors f_r^k are scaled, such that the maximum of their sum is equal to one:

$$\max_{(x,y,z)\in\Omega_M} \sum_{k=1}^{N_{\text{MU}}} f_r^k(x,y,z) = 1.$$

Figure 8.36 shows the MU occupancy factors for the four MUs in the considered example scenario. It can be seen that the individual MU territories only occupy a small fraction of the muscle domain and are centered around fibers in longitudinal direction of the muscle.

Results of the simulation at $t = 14\,\text{ms}$ are given in Fig. 8.37. In the considered scenario, the first and second MU are stimulated at $t = 0\,\text{ms}$ and $t = 10\,\text{ms}$, respectively. At the time of the displayed images, MUs 3 and 4 have not yet been stimulated.

Upon stimulation, we prescribe the membrane voltage V_m for one timestep as 20 mV at the stimulated nodes in the mesh of the respective MU. In this scenario, the stimulated nodes are located in the middle of the muscle in longitudinal direction in three adjacent cross-sectional layers of mesh elements.

The upper two images in Fig. 8.37 show the locations of the propagated action potential fronts at $t = 14\,\text{ms}$ for MU 1 and MU 2, given by the values of V_m^k. While the action potentials span the entire cross-section of the muscle domain, they contribute to the EMG value scaled by their locally varying occupancy factor f_r^k.

As the V_m^k values of all MUs $k \in \{1, \dots, N_{\text{MU}}\}$ are strongly coupled, the active MUs, MU 1 and MU 2, influence the V_m^k scalar fields of the inactive MUs, MU 3 and MU 4. The lower two images in Fig. 8.37 show the computational domain of the muscle with several layers of 3D elements removed. It can be seen that, at some regions in the interior of the domain, the values of V_m^3 and V_m^4 correspond to the negated value of V_m^2 with a smaller absolute value. Note the different color scales for V_m^1, V_m^3 and V_m^4 in these images.

Figure 8.38a shows the values of the extracellular potential ϕ_e on the muscle domain. The contributions from the two active MUs can be seen. Figure 8.38b gives an impression of the used mesh and shows the value of ϕ_e for all nodes. It can be seen that the ϕ_e values span a larger value range in the interior of the domain than on the boundary, as previously shown in Fig. 8.38a. Correspondingly, the color coding in Fig. 8.38b uses a larger range than in Fig. 8.38a.

Figure 8.38c shows the EMG values ϕ_b on the surface of the body domain mesh. The effect of the body domain is revealed by comparing the electric potential on the boundary of the muscle mesh in Fig. 8.38a with the values in Fig. 8.38c. The signals get locally smoothed by the fat layer.

8.5.2 Simulation of EMG Signals

In the second scenario, we simulate a higher number of 25 MUs. The MUs are activated at random times within the first 20 ms. We use a mesh with $12 \times 12 \times 74 = 10\,656$ elements in the muscle domain and $24 \times 4 \times 74 = 7104$ elements in the body domain. Compared to the mesh in the previous scenario, the spatial resolution in radial direction is chosen

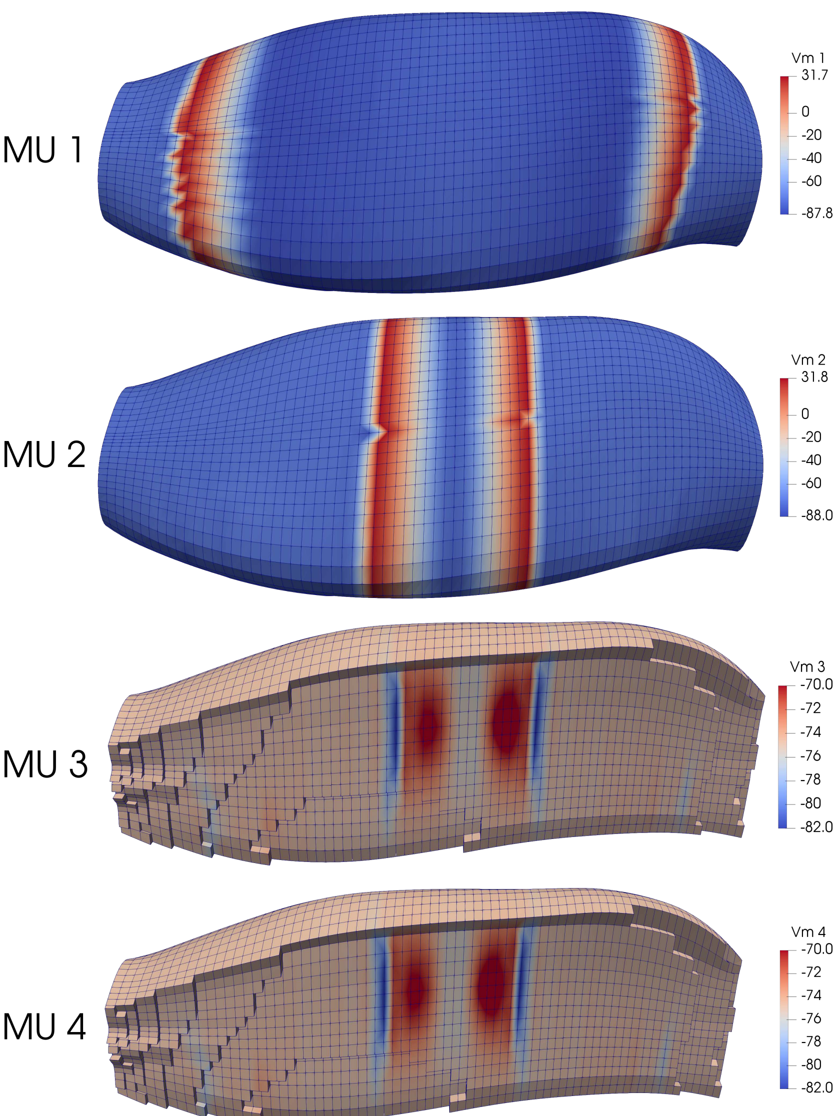
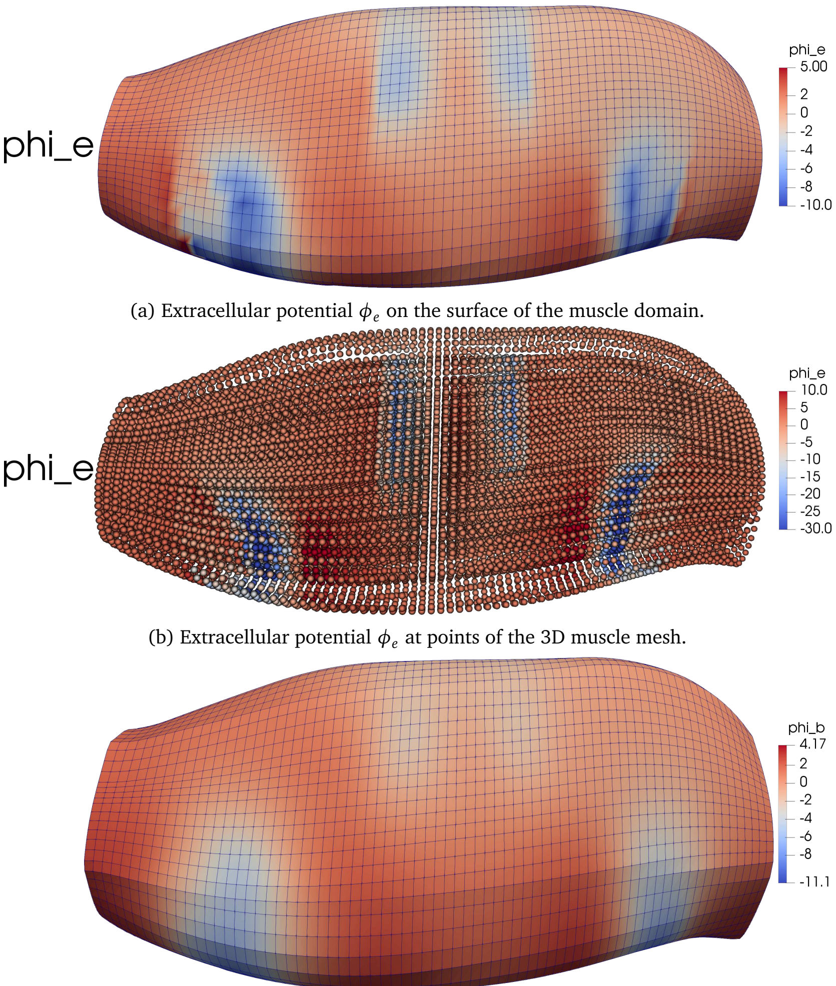


Figure 8.37: Simulation of electrophysiology with the multidomain model: Transmembrane potential V_m^k of MUs 1 to 4 at $t = 14\,\text{ms}$.



(a) Extracellular potential ϕ_e on the surface of the muscle domain.

(b) Extracellular potential ϕ_e at points of the 3D muscle mesh.

(c) EMG signal ϕ_b on the surface of the body fat domain. The comparison with (a) shows the effect of the fat layer.

Figure 8.38: Simulation of electrophysiology with the multidomain model: Simulation results at $t = 14\,\text{ms}$ for a scenario with 4 MUs.

slightly coarser, to speed up the computation. The other model parameters, discretization schemes and solvers are chosen as before.

In this scenario, the stimulated nodes are no longer located in the middle of the muscle, but randomly varied by up to 10 % of the muscle length using a uniform distribution. This approach to model the neuromuscular junctions is analog to the approach in Sec. 8.4.2 for the fiber based electrophysiology. Figure 8.39a shows the membrane voltage V_m^1 of MU 1 shortly after the MU has been activated. Because of the different locations of the stimulated points, no uniform action potential "front" as in Fig. 8.37 is seen. Instead, a characteristic 2D MU action potential forms.

The spike of the depolarized membrane voltage at every stimulated point propagates along the fiber direction and additionally diffuses in transverse direction. This yields the cone-like structures of lower V_m values as seen in Fig. 8.39a. The origin of the transverse propagation is the electric conduction in the extracellular space, which is governed by the isotropic conduction tensor $\boldsymbol{\sigma}_e$ in Eq. (8.5). This isotropic conduction is strongly coupled to the directed action potential propagation within every MU compartment.

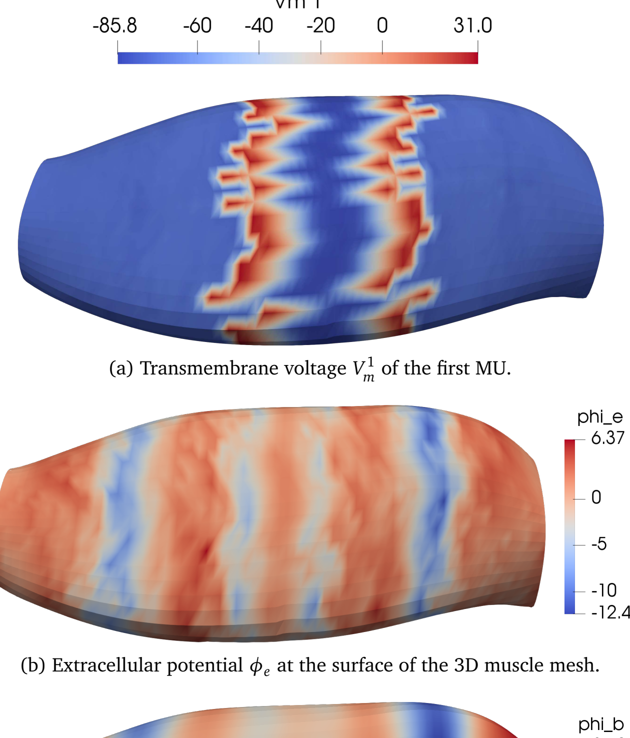
The resulting EMG values ϕ_e on the muscle boundary and ϕ_b on the skin surface are given in Fig. 8.39b and Fig. 8.39c, respectively. The fat layer again smooths out the signal, as observed in the last section and in Sec. 8.4.3 for the fiber based electrophysiology model.

The two blue vertical stripes in Fig. 8.39b with lower ϕ_e values correspond to the action potentials of multiple MUs at the respective location. It can be seen that the resulting EMG signal varies more in longitudinal direction than in transverse direction. This can be explained by the wide MU territories in this scenario.

8.5.3 Comparison of the Fiber Based Electrophysiology Model and the Multidomain Model

EMG signals on the upper arm can be simulated by both the fiber based electrophysiology model, as demonstrated in Sec. 8.4 and by the multidomain model, as shown in the previous sections. The two approaches have several similarities and differences.

Both model approaches have in common that they are based on biophysical principles. They both involve a detailed subcellular model, which describes the biochemical processes on the muscle fiber membranes. The subcellular model is solved at discrete points in the 3D domain and the model instances are coupled to a description of electric volume



(a) Transmembrane voltage V_m^1 of the first MU.

(b) Extracellular potential ϕ_e at the surface of the 3D muscle mesh.

(c) EMG signal ϕ_b at the surface of the body fat domain.

Figure 8.39: Simulation of electrophysiology with the multidomain model: Simulation results at $t = 20\,\text{ms}$ for a scenario with 25 MUs.

conduction in the muscle domain and the body fat layer. Both the fiber based approach and the multidomain model are, thus, multi-scale descriptions.

Both models also resolve the physiological structure of muscle activity given by multiple MUs. Action potential propagation is computed separately for different MUs. In a comprehensive simulation of the neuromuscular system, motor neuron models can be coupled to drive the activation of the MUs.

The two domains of the electrically active muscle tissue and the passive layer of skin and adipose tissue are also considered in both modeling approaches. In the fiber based approach, electric volume conduction is described by the 3D bidomain equation Eq. (5.9a) for the muscle domain and a strongly coupled 3D Laplace equation Eq. (5.17) for the body domain. The multidomain approach for electric volume conduction generalizes the bidomain model and yields the bidomain equation as a special case, if only one MU is considered. The Laplace equation for the body domain is coupled in the same way as in the fiber based model. In summary, both approaches are very similar regarding the 3D electric conduction part.

The major difference between the models is, that the multidomain approach considers only 3D domains, whereas the fiber based approach resolves individual 1D muscle fibers. Another difference lies in the coupling between the model components. For the fiber based approach, action potential propagation on the 1D fibers is unidirectionally coupled to the 3D volume conduction part of the model. In the multidomain model, all components are bidirectionally coupled. This allows, e.g., to simulate externally applied stimulations by active electrodes on the skin surface. The effects of the external currents on the electric potentials in the 3D muscle volume and down to the 0D subcellular behavior can only be described accurately by the multidomain approach.

The difference in coupling between the action potential propagation model part and the electric volume conduction in the extracellular space can also be seen in the computed EMG signals. A comparison of EMG simulations using the fiber based model, e.g., Fig. 8.24b, and the multidomain model, e.g., Fig. 8.39b, shows that the multidomain approach yields less sharp artifacts in the 2D EMG signal on the muscle surface than the fiber based method. In the fiber based model, the action potentials of individual fibers are visible in the signal. In the multidomain simulations, the regions of similar activity are more clustered in the resulting EMG signals.

Other differences between the two model approaches exist in terms of the computational performance properties of their solvers. In the fiber based approach, action potential propagation can be computed independently for all muscle fibers, which enables

large speedups by parallelization and makes large problem sizes with realistic numbers of muscle fibers feasible. For example, Sec. 8.4.4 presents the simulation of 270 000 muscle fibers with 27 000 compute cores. In the multidomain approach, on the other hand, a large linear system of equations has to be solved in every timesteps. This can also be parallelized, but requires communication between the involved processes, which limits the parallel scalability for large problem sizes.

In the multidomain model, the computational effort increases, in good approximation, linearly with both the number of MUs and the number of nodes in the mesh. In the fiber based approach, the amount of computational work mainly corresponds to the number of fibers, not to the number of MUs. The 3D problem and, thus, the mesh width of the 3D mesh, typically plays a minor role in the total runtime for the fiber based approach, as the 3D problem is only solved according to the desired EMG sampling frequency. In the multidomain approach, no separate timestep widths can be chosen for the computations of the extracellular and body domain electric potentials, ϕ_e and ϕ_b, as they are computed as a solution of the same linear system of equations.

For example, the computation of 24 ms of the multidomain scenario in Sec. 8.5.2 with 25 MUs and 126 processes has a runtime of approximately 106 min. The fiber based approach with the same 3D mesh and the same parallel partitioning with 126 processes has a total runtime of 6 s for 169 fibers or 20 s for 1369 fibers. A scenario with 169 fibers leads to a fiber spacing that corresponds to the 3D mesh width in the compared multidomain scenario. The speedup between the models in this case is approximately 1000. Note that only the computation of the fiber based approach is highly optimized in this work, and a better performance of the multidomain solver could be achieved in future work. However, the structural properties of the models facilitate highly parallel simulations only for the fiber based approach.

As a result, if the simplifications of a unidirectional coupling of the extracellular potential ϕ_e from the muscle fibers to the 3D volume can be tolerated, the fiber based approach should be used, as it exhibits significantly lower runtimes. The fiber based approach is (considering the current implementation) the only possible choice for scenarios with at least two of the three requirements (i) long simulation time spans in the range of seconds, (ii) large number of MUs in the range of multiple dozens, and (iii) finely resolved 3D meshes in the range of several 10^6 degrees of freedom.

The multidomain approach, on the other hand, can describe phenomena that are not accurately captured by the fiber based model, as described earlier. Moreover, the multidomain model is potentially easier to handle for more irregular geometries, where only a

structured 3D mesh and no physiologically oriented fibers are given. Another advantage of the multidomain approach is its ability to fine tune the MU territories. The multidomain model can also possibly simulate a given MU distribution with the same accuracy with less 3D points than the fiber based approach. However, investigations in this direction are subject of future research. If large runtimes are not an issue, the multidomain approach can be used to yield more physically accurate results than the fiber based approach, ultimately advancing the means to describe the neuromuscular system as detailed and accurately as possible.

How To Reproduce

The two simulations in this section with 4 and 25 MUs, respectively, which are visualized in Figures 8.37 to 8.39, can be executed by the following commands:

```
cd $OPENDIHU_HOME/examples/electrophysiology/multidomain/
  ↪ multidomain_with_fat/build_release
mpirun -n 128 multidomain_with_fat ../settings_multidomain_with_fat.
  ↪ py 4mus.py --n_subdomains 8 1 16
mpirun -n 126 ./multidomain_with_fat_emg ../
  ↪ settings_multidomain_with_fat.py all_active.py --n_subdomains 6
  ↪ 1 21
```

For other available numbers of processes, the subdomains at the end of the commands have to be adjusted.

8.6 Simulation of Coupled Electrophysiology and Solid Mechanics

Simulating muscle contraction with a detailed model, which accurately describes motor recruitment, yields the basis for new insights into the neuromuscular orchestration of processes that lead to muscle force generation.

We couple the two model approaches for electrophysiology, the fiber based model presented in Sec. 8.4 and the multidomain model presented in Sec. 8.5, with a solid mechanics model. In Sec. 8.2, we demonstrated the solver for nonlinear hyperelasticity models in simulations of the passive behavior of muscle tissue. The current section aims at simulating active muscle contraction.

Section 8.6.1 couples the fiber based electrophysiology model with a model of muscle contraction. Section 8.6.2 discusses an algorithm to add prestress to the description. Section 8.6.3 demonstrates the coupling of the multidomain model with the model of muscle contraction. Section 8.6.4 and Sec. 8.6.5 describe simulations using the numerical coupling library preCICE.

8.6.1 Fiber Based Electrophysiology and Muscle Contraction

We begin with coupling the fiber based electrophysiology solver with the solid mechanics model to simulate muscle contraction as a result of the activation of muscle fibers. We use the subcellular model of Shorten et al. [Sho07]. It computes the microscopic activation parameter $\gamma \in [0,1]$, which is related to the concentration of attached cross-bridges in the sarcomeres. The parameter γ is mapped and homogenized from the 0D subcellular points to $\bar{\gamma}$ on the 3D mesh. In the macroscopic 3D mechanics description, the factor is multiplied with a maximum active stress parameter $S_{\text{max,active}}$ and a force-velocity characteristic $f_\ell(\lambda_f)$, as described in Sec. 5.2.7. The 3D mechanics model updates the geometry of the 3D domain and transfers the fiber stretch value λ_f and the contraction velocity $\dot{\lambda}_f$ back to the subcellular model.

In this scenario, we aim to simulate a rapid and strong contraction of the biceps muscle. The scenario contains 169 fibers, which are associated with 15 MUs. This association is generated by method 1 in Sec. 4.2. All MUs are subsequently activated in a ramp in the first 1.4 s.

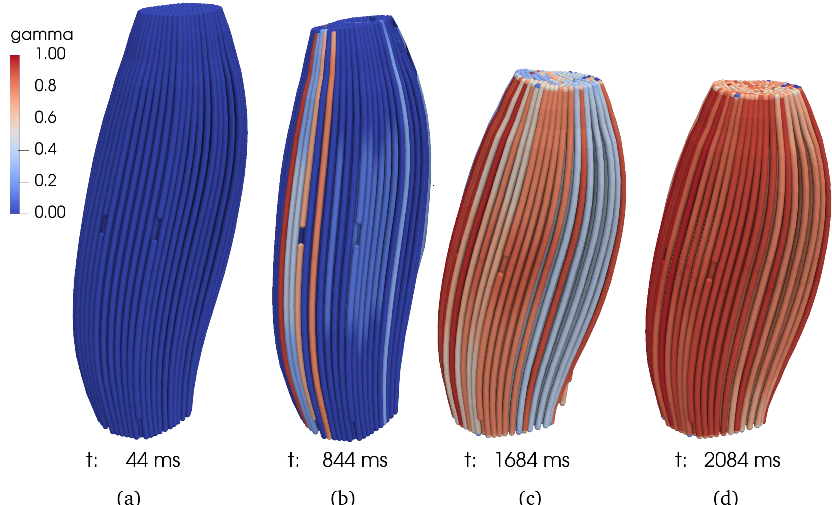
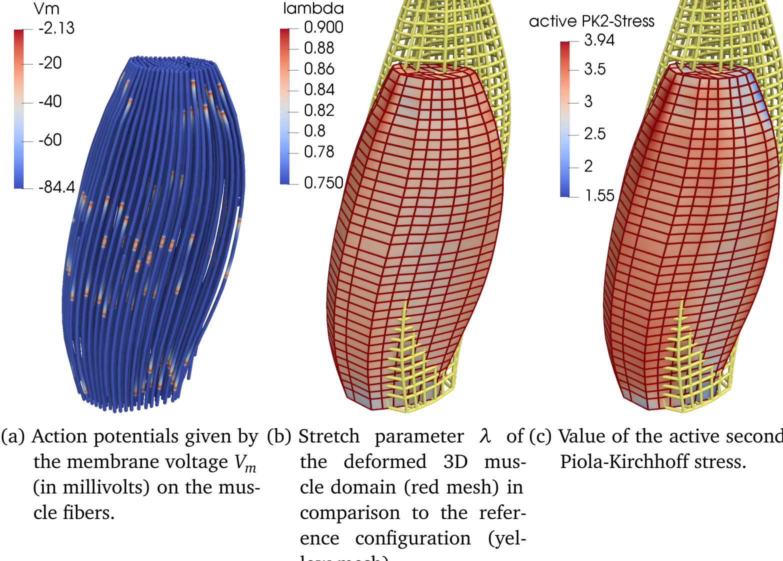


Figure 8.40: Simulation of fiber based electrophysiology and muscle contraction: Activation of the muscle fibers and overall deformation at various timesteps.

The muscle geometry is fixed at its lower end and no external forces are considered in this scenario. The dynamic formulation with the transversely isotropic Mooney-Rivlin material is used, as described in Sec. 5.2.7. The 3D muscle mesh contains $2 \times 3 \times 18 = 108$ elements with quadratic finite element ansatz functions and 1295 nodes in total and is partitioned into subdomains for four processes. Time step widths of $dt_{0\text{D}} = dt_{1\text{D}} = dt_{\text{splitting}} = 10^{-4}\,\text{ms}$ and $dt_{3\text{D}} = 1\,\text{ms}$ are used. The used numerical solvers and other settings of the electrophysiology and contraction models are equal to the described scenarios in Sections 8.4.2 and 8.4.4 and Sec. 8.2.1.

Figure 8.40 shows the fibers of the contracting muscle at four different timesteps between $t = 44\,\text{ms}$ and $t = 2.084\,\text{s}$. The fibers are colored according to the resulting activation parameter γ, which is a measure for the generated force on the sarcomere level. Between $t = 44\,\text{ms}$ and $t = 844\,\text{ms}$, shown in Figures 8.40a and 8.40b, the smallest MUs are activated, which, in this example, are mainly located on the left-hand side. As a consequence, the muscle domain initially bends slightly to the left. As more MUs become active at $t = 1684\,\text{ms}$, depicted in Fig. 8.40c, the deformation increases and the bending direction is reversed. However, the fibers on the left-hand side still exhibit the highest γ value, as they have been stimulated most often at that time. At $t = 1684\,\text{ms}$,



(a) Action potentials given by the membrane voltage V_m (in millivolts) on the muscle fibers.

(b) Stretch parameter λ of the deformed 3D muscle domain (red mesh) in comparison to the reference configuration (yellow mesh).

(c) Value of the active second Piola-Kirchhoff stress.

Figure 8.41: Simulation of fiber based electrophysiology and muscle contraction: Simulation results at $t = 2084\,\text{ms}$.

visualized in Fig. 8.40d, almost all fibers have a γ value close to one, corresponding to full activation.

Figure 8.41 shows several variables at the simulation end time of $t = 2084\,\text{ms}$. Figure 8.41a visualizes the transmembrane voltage V_m on the muscle fibers. Action potentials can be seen on almost all fibers, as the whole muscle is activated at this time. Figure 8.41b shows a comparison between the reference configuration given by the yellow mesh and the current configuration given by the red mesh. The muscle domain is colored according to the stretch λ, which has a nearly constant value of $\lambda \approx 85\,\%$ at the end time of this scenario. A similar visualization is given in Fig. 8.41c for the active stress in the muscle.

Because of the high level of activation and the corresponding active stress distribution in the muscle, our mechanics solver only converges up to the shown simulation time of 2084 ms in this scenario. The aim of the scenario is to simulate the contraction of a fully activated muscle. Other scenarios, where the activation is applied more slowly, allow for a convergence of the mechanics solver during longer simulation time spans.

The presented scenario showed that a fully activated biceps muscle contracted to about 85 % of its original length. However, in reality, larger contractions are possible. In the shown scenario, the muscle was initially in a stress-free configuration. More realistic scenarios can incorporate pretension forces, where the undeformed reference configuration is subject to a constant stress level in the muscle's direction of the line of action. This is considered in the next scenario.

How To Reproduce

The simulation in this section can be run as follows:

```
cd $OPENDIHU_HOME/examples/electrophysiology/fibers/
  ↪ fibers_contraction/no_precice/build_release
mpirun -n 4 ./biceps_contraction ../settings_biceps_contraction.py
  ↪ ramp.py
```

8.6.2 Simulation of Prestress

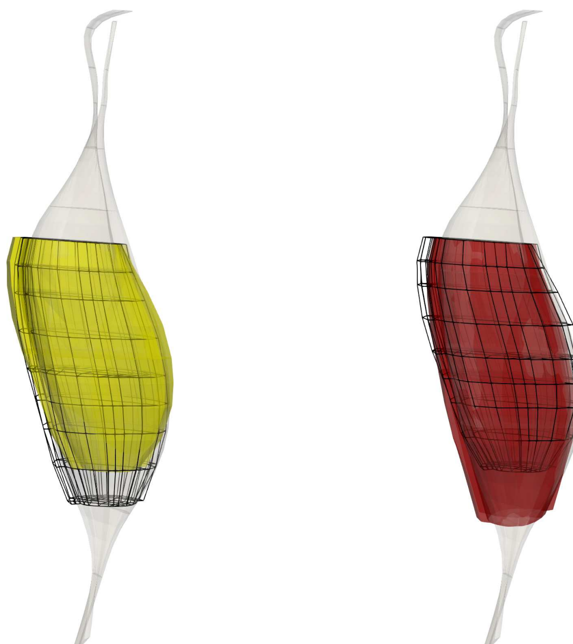
To obtain more realistic ranges of muscle contraction, a nonzero, constant prestress can be considered in the undeformed configuration of the muscle. In our solid mechanics formulation, the reference configuration always has zero stress. Thus, we need to construct a separate, first reference configuration of a shorter muscle geometry. We stretch it to the original muscle length by applying external forces. The resulting, second configuration resembles the original muscle geometry and has the desired prestress characteristics.

The detailed steps of this algorithm are visualized in Fig. 8.42 and are described in the following. We begin with the given geometry of the muscle with body fat layer, which is shown as black wireframe mesh in Fig. 8.42a. In a first static simulation step, a constant active stress $\alpha_{\text{pre}}\, S_{\text{max,active}}$ is prescribed in the entire muscle volume. The resulting muscle deformation is computed, using the usual nonlinear hyperelastic muscle material. $S_{\text{max,active}}$ refers to the maximum active stress value as used in the mechanics model description in Eq. (5.42). The result of this first step is a shortened muscle with the same volume as the original geometry. Figure 8.42 shows the result by the yellow volume for $\alpha_{\text{pre}} = 0.3$. It can be seen that the length of the muscle has shortened by approximately 13 %.

In the second step, we reuse the computed deformed geometry of the first step as new stress-free reference configuration and re-extend it by applying a constant surface load F_{pre} pointing to the bottom on the lower face in the setting of Fig. 8.42. The value of F_{pre} corresponding to α_{pre} has to be estimated by numerical experiments.

This step is again solved as a static problem. The result is a similar muscle geometry as the original one, and contains prestress according to the applied force. The muscle volume is exactly preserved due to the incompressible material formulation. Figure 8.42b shows the starting point for the second step by the black wireframe mesh and the resulting geometry for a total applied force of $F_{\text{pre}} = 30\,\text{N}$ by the red volume. The comparison of the original, black mesh in Fig. 8.42a with the red volume in Fig. 8.42b shows a good match of the geometry. For the subsequent dynamic simulations of, e.g., muscle contraction, the surface load has to be constantly applied. It corresponds to the tendon forces and the loads of the musculoskeletal system acting on the muscle.

The active stress parameter α_{pre} and the corresponding preload force F_{pre} can be chosen according to the desired amount of prestress. However, the higher these values are chosen, the more difficult is it for the nonlinear solid mechanics solvers to converge to a solution. Especially for irregular or large mechanics meshes, a lower stress factor of,



(a) In the first step, the original mesh (black wireframe) is contracted by an artificial active stress $\alpha_{\text{pre}}\, S_{\text{max,active}}$ to yield a shortened muscle (yellow mesh).

(b) In the second step, the mesh is extended again by an external surface load. The black wireframe corresponds to the yellow volume in (a), the red volume is the resulting geometry.

Figure 8.42: Simulation of biceps muscle geometry with prestress: The two steps of the algorithm to generate a reference geometry with prestress, shown with the geometry of the tendons for reference.

e.g., $\alpha_{\text{pre}} = 0.1$ has to be chosen. To improve convergence, we apply the load in the second step of the algorithm incrementally by several load steps. In addition, reducing the number of unknowns and increasing the mesh width in the mechanics problem can help, as this improves the conditioning of the problem.

8.6.3 Coupling of the Multidomain Model and Solid Mechanics Model with Prestress

In the following, we present a scenario that uses the prestress algorithm of the last section and couples the multidomain and mechanics models to simulate surface EMG signals on the skin surface over a contracting muscle.

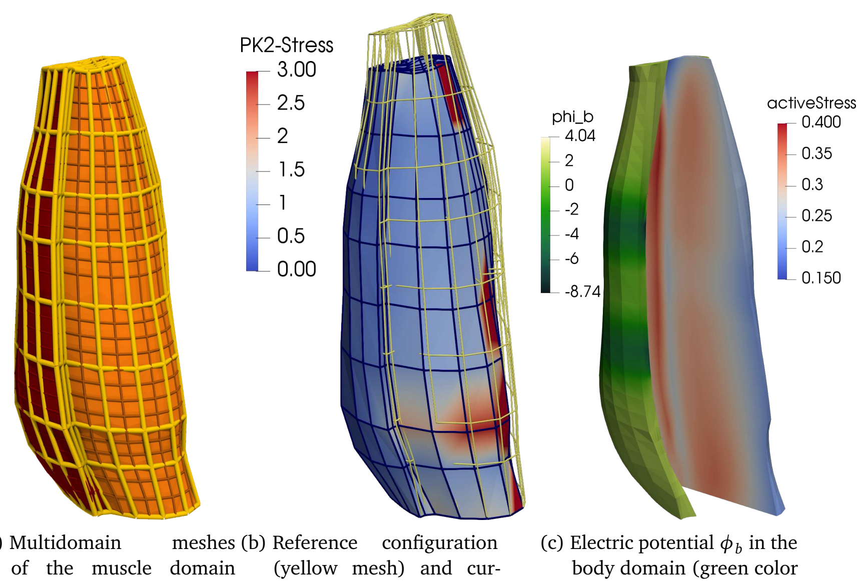
We choose $\alpha_{\text{pre}} = 0.1$ and apply the prestretch force $F_{\text{pre}} = 10\,\text{N}$ in three load steps. The multidomain model considers 5 MUs with stimulation frequencies between 7 Hz and 24 Hz and a 3D mesh of $8 \times 8 \times 28 = 1792$ elements. We execute the simulation with four processes. All other parameters and settings of the multidomain model and the solid mechanics model that are not explicitly mentioned in the following are chosen the same as in Sec. 8.5.1 and Sec. 8.2.1.

For the discretization of the mechanics model, we use a coarser mesh than for the multidomain model. Furthermore, we use quadratic elements instead of linear elements. The Python implementation of the settings script of this example contains functionality to create the mechanics mesh by subsampling the multidomain mesh with specified factors. In the current scenario, we set these factors for the x, y and z directions to 0.7, 0.7, and 0.3, respectively. As a result, we get meshes with $5 \times 7 \times 9 = 315$ elements for the muscle and $5 \times 1 \times 4 = 20$ elements for the body fat domain. Figure 8.43a visualizes all meshes used in this scenario: The orange muscle mesh and the red body mesh are used for the multidomain model, and the yellow mesh is used for the solid mechanics model.

Figures 8.43b and 8.43c depict results of the simulation at time $t = 920\,\text{ms}$. Figure 8.43b shows the reference geometry by the yellow wireframe after applying the prestress. The muscle is colored according to the value of the second Piola-Kirchhoff stress. During this dynamic simulation, the muscle bends elastically slightly to the left and right, as it is only fixed at its bottom in Fig. 8.43. This explains the stress distribution at the snapshot for $t = 920\,\text{ms}$ in Fig. 8.43b, where higher stresses occur on the right-hand side.

Figure 8.43c shows the electric potential ϕ_b of the body domain by the green color scale on the left of the image. The visible part of the fat layer shows two action potentials, visualized by the two dark green stripes.

Moreover, Fig. 8.43c displays the total active stress $\mathbf{S}_{\text{active}}$ in the interior of the muscle by the color scale that ranges from blue to red color. In the multidomain model, $\mathbf{S}_{\text{active}}$



(a) Multidomain meshes of the muscle domain (orange), body fat domain (red) and the coarser mesh used for the solid mechanics model (yellow).

(b) Reference configuration (yellow mesh) and current configuration of the muscle colored according to the distribution of the second Piola-Kirchhoff stress.

(c) Electric potential ϕ_b in the body domain (green color scale, in millivolts) and active stress in the interior of the muscle (blue-red color scale, in $\frac{\text{N}}{\text{cm}^2}$).

Figure 8.43: Simulation of muscle contraction based on the multidomain model with prestressed muscle geometry: Used meshes and simulation results at $t = 920\,\text{ms}$ of a scenario of the multidomain electrophysiology model coupled to the solid mechanics model.

is calculated as a weighted sum over the contributions $\mathbf{S}^k_{\text{active}}$ of the MU compartments, scaled by the occupancy factors f_r^k (cf. Sec. 5.1.4):

$$\mathbf{S}_{\text{active}} = \sum_{k=1}^{N_{\text{MU}}} f_r^k \, \mathbf{S}^k_{\text{active}}.$$

The muscle domain in Fig. 8.43c is cut open, such that interior distribution at the cut plane can be seen. The image shows two regions of higher active stress, which run vertically through the muscle, given by red color. They are a result on the location of the MUs in this scenario. The legend shows that the active stress inside the muscle is below $0.4\,\frac{\text{N}}{\text{cm}^2}$, while the maximum active stress parameter is chosen as $S_{\text{max,active}} = 7.3\,\frac{\text{N}}{\text{cm}^2}$. This low activation level is a result of the chosen MU recruitment. As a result, the muscle only slightly contracts, as can be seen in Fig. 8.43b.

In summary, both the fiber based electrophysiology model and the multidomain model can be coupled with the nonlinear solid mechanics model to simulate muscle contraction, as presented in Sec. 8.6.1 and in this section. The computational efficiency considerations discussed in the comparison of the fiber based and multidomain approaches in Sec. 8.5.3 also apply to coupled simulations with muscle contraction. For longer simulation times, the fiber based approach in Sec. 8.6.1 is, therefore, favored.

How To Reproduce

The simulation can be run with the following commands. Instead of four processes also other numbers are possible. A lot of parameters can be fine-tuned in the `../variables/multidomain.py` settings file.

```
cd $OPENDIHU_HOME/examples/electrophysiology/multidomain/
  ↪ multidomain_prestretch/build_release
mpirun -n 4 ./multidomain_prestretch ../
  ↪ settings_multidomain_prestretch.py multidomain.py
```

8.6.4 Coupling of Solid Mechanics Models using the Software preCICE

One problem of multi-scale simulations with solid mechanics models is the limited amount of parallelism, if a coarse mechanics mesh with a low number of elements is chosen. The domain can only be partitioned into as many subdomains as there are elements in the 3D mechanics mesh. While this is not an issue for small scale simulations like the ones shown in the previous sections, it prohibits exploitation of High Performance Computing resources, e.g., if numerous muscle fibers are considered as in Sec. 8.4.4.

The reason for the limited parallelism lies in the partitioning scheme, where every node in the 3D domain corresponds to the subdomain of exactly one process, regardless of the mesh. OpenDiHu does not allow to partition, e.g., the finely resolved 1D muscle fiber meshes differently than the coarse 3D mechanics mesh. However, this restriction can be circumvented by using multiple OpenDiHu programs with different partitioning schemes and by performing the data transfer between the meshes using an external coupling software.

We provide support for the black-box coupling library preCICE [Bun16]. This open source library allows mapping data between different meshes, can communicate values between subdomains that reside on different processors, and implements implicit numerical coupling schemes with quasi-Newton methods. The implementation is known to scale well on small-scale clusters and supercomputers. The preCICE library targets a minimally-invasive approach, where the user application implements a preCICE adapter. Multiple, potentially different solver codes can be coupled numerically and compute individual model parts of a joint multi-physics simulation. Moreover, preCICE has an active and growing community where experiences and codes are shared, and open source adapters are available for several popular solvers.

This makes the library suited for our use case. We provide two different types of preCICE adapters in OpenDiHu, one for surface coupling of 2D meshes and one for volume coupling of 3D meshes. These adapters integrate with the structure of nested solvers and can be positioned anywhere in the solver tree (cf. Fig. 6.8). The meshes and variables that are exposed to preCICE can be configured in the settings file.

In the current section, we show how to use the volume coupling adapter to resolve the initially stated issue of limited scalability for coupled simulations with electrophysiology and mechanics models. Subsequently, the next section presents a simulation that uses surface coupling. Details can also be found in [Mai22].

We simulate muscle contraction and surface EMG of the biceps muscle using the fiber based electrophysiology model. To fully exploit the capabilities of an 18-core Intel Core i9-10980XE processor, we compute the electrophysiology model using 16 processes and the mechanics model using 2 processes. The data mapping between the differently partitioned 3D meshes is performed by preCICE.

Figure 8.45 shows the structure of the simulation components with the used meshes and the exchanged variables in this simulation. Two different OpenDiHu programs are executed at the same time, given by the gray boxes. The program corresponding to the left box solves the electric conduction problem, given by the bidomain equation Eq. (5.9a) on the 3D domain and the action potential propagation model, given by the monodomain equation Eq. (5.11) on a large number of 1D muscle fiber meshes. The 3D and the 1D mesh in this program are partitioned into 16 subdomains for the 16 processes.

Figure 8.45 visualizes the meshes and their partitioning to the different processes by the colored inset images. It can be seen that the fibers meshes and the 3D mesh used in the OpenDiHu program in the left box have corresponding subdomains.

The second OpenDiHu program visualized by the right box in Fig. 8.45 only solves the solid mechanics problem using a coarse 3D mesh. The mesh of this problem is partitioned to two processes, as shown by the image.

The three model parts are numerically coupled and need to exchange several variables. The action potential propagation model, shown at the lower left of Fig. 8.45, computes the transmembrane voltage V_m and the activation parameter γ and maps them from the 0D points on the fibers to the 3D mesh using the mapping scheme described in Sec. 7.8. The activation parameter γ is needed in the solid mechanics model. It is transferred between the two OpenDiHu programs using the functionality of preCICE. After the solid mechanics solver has computed a new deformation of the coarse solid mechanics mesh, preCICE maps the node positions to the finer 3D mesh in the left program. The geometries of the 3D and 1D meshes in the left program are updated accordingly. The preCICE couplings in this example use serial explicit coupling and radial basis functions for the data mapping.

The presented scheme in Fig. 8.44 allows us to simulate muscle contraction and surface EMG signals. The volume coupling with preCICE is configured between the two 3D meshes. Even for scenarios where EMG signals are not of interest and only the muscle contraction resulting from the activated muscle fibers should be simulated, the presented approach can be used.

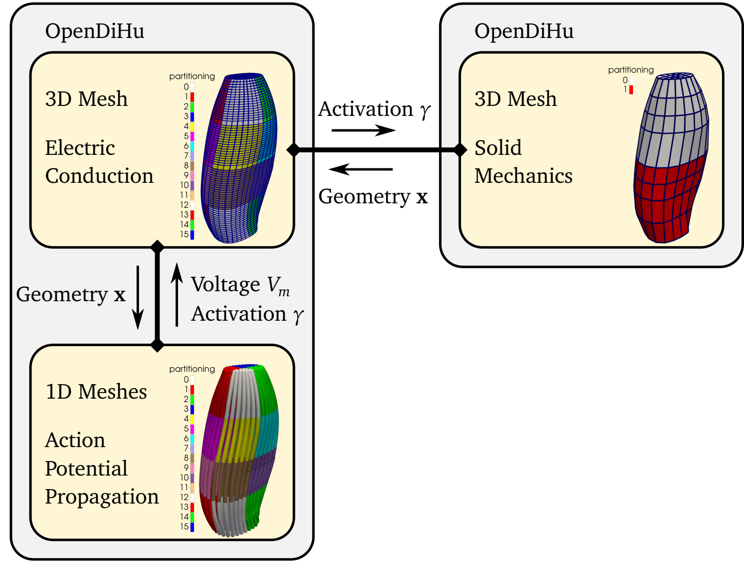


Figure 8.44: Simulation of muscle contraction: Structure of a coupled simulation with the coupling library preCICE on 18 processes, consisting of the two independent OpenDiHu programs indicated by the gray boxes. The program in the left box solves the electric conduction model using the shown 3D mesh and the action potential propagation model using the shown 1D fiber meshes. Both meshes are partitioned to 16 subdomains as shown by the colors. The program in the right box solves the mechanics problem on a coarse 3D mesh, which is partitioned into 2 subdomains. The arrows between the models indicate the exchanged variables. The coupling within the left gray box is implemented in OpenDiHu, the coupling between the gray boxes is realized using preCICE.

An alternative approach, where preCICE instead couples directly between the fiber meshes and the solid mechanics 3D mesh is also implemented. This approach neither includes the electric conduction model nor the fine 3D mesh for the left program. However, the mapping between the solid mechanics mesh and the set of 1D fiber meshes is more costly than the mapping between the two 3D meshes, as the fibers contain more data points in total than the 3D mesh of the electric conduction problem. A quantitative analysis of this effect is subject to work in progress.

Apart from ensuring better parallel scalability, the OpenDiHu model setup using preCICE also allows to exchange the solid mechanics solver by a different solver code, e.g., a commercial solver. The black-box approach of preCICE allows to exchange the mechanics solver without any changes to the electrophysiology simulation, contributing to the extensibility goal of combining modular model components.

How To Reproduce

The two programs with preCICE coupling can be used as follows. Note that the compilation of preCICE has to be enabled in the `user-variables.scons.py` configuration file for the `scons` build system in the `$OPENDIHU_HOME` directory.

```
cd $OPENDIHU_HOME/examples/electrophysiology/fibers/
  ↪ fibers_contraction/with_precice_volume_coupling/build_release
mpirun -n 2 ./muscle_contraction ../settings_muscle_contraction.py
  ↪ ramp.py
mpirun -n 16 ./fibers_with_3d ../settings_fibers_with_3d.py ramp.py
```

8.6.5 Simulation of a Muscle-Tendon Complex using Surface Coupling with preCICE

In all previously presented simulations of muscle contraction, the biceps muscle was considered in isolation. In the following, we present a physiologically more correct scenario that includes a 3D description of the tendon mechanics. The simulation consists of four individual solvers in OpenDiHu for the distal tendon, the two proximal tendons, and the muscle belly. The coupling library preCICE is used to numerically couple the parts.

An advantage of simulations of an entire muscle-tendon complex is their more realistic line of action of the muscle force, compared with a model of the muscle belly without tendons. The goal of the simulation described in this section is to predict the progression of the total muscle force as result of MU recruitment.

For the simulation of the muscle contraction part, we couple the fiber based electrophysiology model with the nonlinear solid mechanics model as described in Sec. 8.6.1. The electrophysiology part of the muscle uses the subcellular model of Shorten et al. [Sho07]. The solid mechanics description of the three tendons uses the hyperelastic Saint-Venant Kirchhoff material, which is the extension of the linear elastic formulation given in Sec. 5.2.5 to the geometrically nonlinear regime. The proximal tendons are fixed at their insertion points to the skeletal system. We apply corresponding Dirichlet boundary conditions. At the lower end of the distal tendon, a downwards pulling force is applied. We gradually increase the value of this force in the corresponding Neumann boundary conditions from zero up to the maximum value 100 N during the first 100 ms of the simulation.

The muscle fibers are associated with 10 MUs and activated in a ramp during the first 1.8 s. After each MU has been activated for the first time, it fires with a MU specific frequency between 7.66 Hz and 23.92 Hz plus a random jitter value of 10 %. This setup replicates the progressive recruitment scenario in [Klo20].

The four simulation programs are connected using an implicit Neumann-Dirichlet multi-coupling scheme in preCICE with a constant relaxation factor of 0.5. At the interfaces between the muscle and the tendons, the implicit numerical coupling ensures continuity for the displacements, velocities and stresses. The tendon solvers send their computed displacement and velocity values to the muscle model, where the corresponding Dirichlet

boundary conditions are applied. The muscle model computes traction forces by integrating the stress values over the surface and sends the values to the tendon models, where corresponding Neumann boundary conditions are applied.

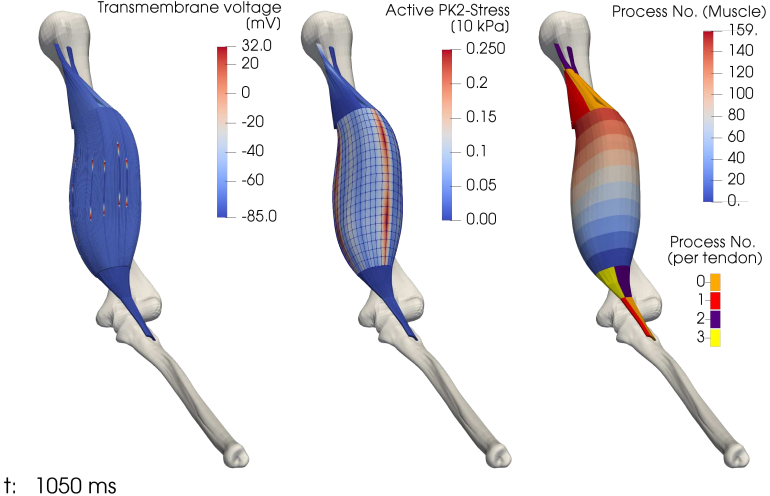
We configure preCICE to use Gaussian radial basis functions for the consistent mapping of the variables between the surface meshes of the muscle and the tendons. An error threshold of $\varepsilon = 0.1$ for the coupled displacement values is used to terminate the implicit coupling scheme. As a consequence, the scheme requires approximately two iterations per timestep on average to reach the error threshold. The coupling step is repeated with a timestep width of $dt_{\text{coupling}} = 1\,\text{ms}$.

We simulate two scenarios of this model. The first scenario considers a high spatial resolution of 1089 muscle fibers and a simulation time of approximately 1 s, while the second scenario considers only 81 fibers but a longer simulation time span of 10 s.

In the first scenario, we use a 3D mesh with $9 \times 9 \times 21 = 1701$ nodes, which are partitioned into 160 subdomains. The meshes of the three tendons each consists of 125 nodes and are each partitioned to four subdomains. We run the computation using 172 processes on four compute nodes of the supercomputer Hawk at the High Performance Computing Center Stuttgart. The hardware is described in more detail in Sec. 8.4.4. The simulation time span of 1 s has a runtime of approximately 7 h 20 min.

Figure 8.45 presents the simulation results of this scenario at the simulation time $t = 1.05\,\text{s}$. Figure 8.45a shows the muscle fibers, which are attached to the tendons at both ends. Several action potentials can be seen on the fibers. Figure 8.45b displays the distribution of active stresses in the 3D mesh at the same simulation time. One vertical red line of higher active stress values can be seen at the foreside of the muscle belly, which corresponds to a region of higher muscle activity. This muscle activity results from a MU that is activated early on in the simulation scenario. A corresponding active fiber at that location can also be identified in Fig. 8.45a. Figure 8.45c visualizes the parallel partitioning of the 3D domains of muscle and tendons into 160 subdomains for the muscle mesh and 4 subdomains for each of the three tendon meshes.

The second scenario uses a coarser 3D mesh with 525 nodes and a parallel partitioning into eight subdomains. We simulate the resulting muscle force, measured at the top insertion point of the proximal tendons, over a longer time period of 10 s. Figure 8.46 shows the resulting relative force progression over time. The plot shows the total force as a moving average function over 0.1 s. It can be seen that the force initially increases, as more and more MUs get activated. A short delay between the onset of the last MU at 1.8 s and the maximum force at 2.39 s can be seen. The muscle force exhibits large



(a) Muscle fibers colored by the value of the transmembrane potential V_m.

(b) Active stress in the 3D muscle mesh.

(c) Parallel partitioning scheme.

Figure 8.45: Simulation of a muscle-tendon complex. Muscle and tendon geometries of the scenario with 1089 muscle fibers, embedded in the skeletal system comprising the ulna bone (lower end) and humerus bone (upper end), result of the simulation at $t = 1.05\,\mathrm{s}$.

oscillations during the period of high muscle activation. They result from the lower firing frequencies of the later activated, large MUs and their higher contribution to the overall activity, compared to the smaller MUs.

Figure 8.46 also shows that the generated muscle force rapidly decreases after the maximum is reached. This is a result from muscle fatigue, which is described by the Shorten subcellular model. The observed decrease to below 60 % after 10 s can also be found in experimental studies of healthy subjects, e.g., in [Eno08].

In conclusion, several biophysical simulation scenarios of MU activity induced muscle contraction have been presented in the previous sections. OpenDiHu allowed us to couple the computationally efficient fiber based electrophysiology description with the solid mechanics model of muscle deformation, as well as the biophysically more accurate multidomain model. An algorithm to include prestresses was presented and the coupling software preCICE was used to numerically couple individual parts of the multi-scale

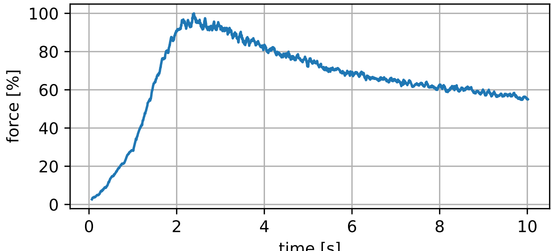


Figure 8.46: Simulation of muscle force in a muscle-tendon complex. Resulting relative muscle force of the biceps muscle with attached tendons using the second scenario with 81 muscle fibers.

model.

The last presented scenario simulated the generated force of a muscle-tendon complex for a simulation time span of 10 s. It can be used in the future to test hypotheses on the influence of various processes along the pathway from MU recruitment over muscle activation to force generation and macroscopic deformation. The simulated force progression related to the maximum voluntary contraction force is a macroscopic quantity, which can be easily measured in in vivo studies. Thus, a connection between the simulation domain and the experimental domain is given, and the microscopic subcellular processes in the muscle fibers are linked to a quantifiable outcome that can be compared with experiments.

How To Reproduce

The simulations in this section were carried out on the supercomputer Hawk in Stuttgart. The job scripts can be found in the repository at github.com/dihu-stuttgart/performance in the directory `opendihu/15_precice_biceps/with_electrophysiology`. To run similar simulations on other computers, run commands that are similar to the following (adjust the numbers of processes):

```
cd $OPENDIHU_HOME/examples/electrophysiology/fibers/
  ↪ fibers_contraction/with_tendons_precice/
  ↪ multiple_tendons_with_electrophysiology
mpirun -n 1 muscle_electrophysiology_precice settings_muscle.py ramp
  ↪ .py
mpirun -n 1 tendon_linear_precice_dynamic settings_tendon_bottom.py
mpirun -n 1 tendon_linear_precice_dynamic settings_tendon_top_a.py
mpirun -n 1 tendon_linear_precice_dynamic settings_tendon_top_b.py
```

Chapter 9

Performance Analysis

In this chapter, we measure the performance of all implemented solvers and evaluate the different actions that were carried out to improve their runtimes and memory characteristics. We consider the performance of instruction-level parallelism, evaluate parallelization strategies for shared and distributed memory parallelism, offloading to a GPU, and hybrid CPU-GPU approaches. We measure weak and strong parallel scaling from using small distributed-memory systems up to the large supercomputers Hazel Hen and Hawk at the High Performance Computing Center Stuttgart. Furthermore, we consider the numerical scaling behavior of several solvers.

Section 9.1 presents numerical studies and improvements in the baseline software OpenCMISS. The numerical properties found in this section are later also used in simulations with OpenDiHu. Section 9.2 evaluates the runtime performance and various optimization options for the electrophysiology solver in OpenDiHu. The best found optimizations are then compared to the OpenCMISS baseline solver in Sec. 9.3. Section 9.4 addresses the computation on the GPU and compares the performance with the CPU computations. Section 9.5 conducts parallel weak scaling studies on supercomputers. Section 9.6 evaluates options and corresponding speedups in the solver of the mechanics model. Section 9.7 conducts numerical studies for the fiber based electrophysiology model and evaluates different solvers for the multidomain model.

9.1 Performance Studies with OpenCMISS Iron

We begin with performance studies on OpenCMISS Iron as the baseline solver, which also implements parts of the multi-scale model considered in this work. The work of [Hei13] describes the implementation of the fiber based electrophysiology model coupled to a quasi-static hyperelastic material model with OpenCMISS. The implementation is

parallelized for a hard-coded number of four processes and serves as the baseline code for the following studies.

We improved the performance of this solver for the multi-scale model by two actions: First, we evaluated and optimized the employed numerical schemes. Second, we implemented parallel partitioning for an arbitrary number of processes and evaluated different parallelization strategies. These changes were directly implemented in the OpenCMISS code. The improvements were also presented in a publication [Bra18]. In the following sections, Sections 9.1.1 and 9.1.2, we describe the numerical improvements and the parallel partitioning strategies. In Sec. 9.1.3, we discuss parallel weak scaling and memory consumption properties.

9.1.1 Numerical Improvements

The first numerical improvement is to replace the GMRES solver, which is used to solve the 1D electric conduction problem on the muscle fibers, by a faster direct solver.

As observed in Sec. 7.4.2, the 1D electric conduction problem of the monodomain equation yields a tridiagonal system that can be solved with linear time complexity. The baseline solver code employs the restarted GMRES solver of PETSc, which is the default linear system solver in OpenCMISS Iron, as it is a robust choice for arbitrary system matrices. However, more efficient solvers for symmetric positive definite systems exist such as the conjugate gradient solver. Furthermore, the MUMPS package [Ame01], which can be interfaced in PETSc, provides a parallel implementation of a direct, multi-frontal linear solver, which is able to exploit the banded structure of the system matrix.

We study the runtime of these three solvers for different problem sizes of the 1D problem. The monodomain equation is solved on a single muscle fiber and the number of 1D elements is varied from 15 to 2807. The used timestep widths are $dt_{\text{0D}} = 10^{-4}\,\text{ms}$ and $dt_{\text{1D}} = 5 \cdot 10^{-3}\,\text{ms}$. The end time of the simulation is 3 ms, yielding a total of 600 calls to the linear solver in the simulated time. The study is executed on an Intel Xeon E7540 processor with 24 cores, clock frequency of 1064 MHz and 506 GiB RAM.

Figure 9.1 shows the runtimes of GMRES, the conjugate gradient solver and the direct solver for this problem in a double-logarithmic plot. It can be seen, that, for coarse discretizations with a low number of 1D elements per fiber, GMRES and the conjugate gradient solver are faster than the direct solver. For finer discretizations, the conjugate gradient solver and the direct solver outperform the GMRES solver. For fibers with more

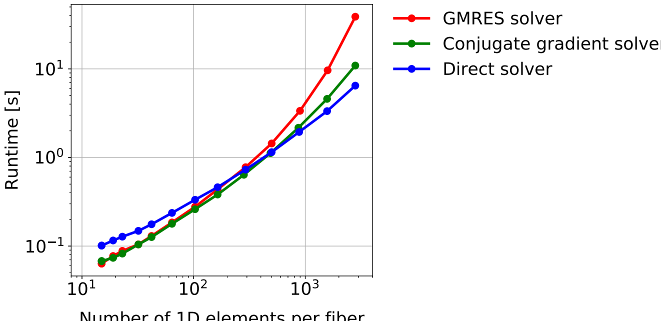


Figure 9.1: Numerical improvements in OpenCMISS: Runtime evaluation of different linear system solvers for a single muscle fiber with varying spatial resolution.

than approximately 500 elements, the direct solver has the lowest runtime. Moreover, the direct solver exhibits an almost linear runtime complexity in terms of the problem size. This indicates that the solver is able to exploit the tridiagonal structure of the system matrix.

The second numerical improvement is the exchange of first-order accurate timestepping schemes by second-order schemes. For this exchange, we implemented the Strang operator splitting scheme and use it with the existing Crank-Nicolson implementation in OpenCMISS Iron and a new implementation of the Heun method by Aaron Krämer.

Numerical studies by Aaron Krämer presented in [Bra18] show that the relation $K = dt_{1\text{D}}/dt_{0\text{D}}$ between the timestep width $dt_{1\text{D}}$ of the 1D electric conduction problem and the timestep width $dt_{0\text{D}}$ of the 0D subcellular model has to be set to $K = 2$ and $K = 5$ for the Godunov and Strang splitting schemes, respectively, such that the errors of the 0D and 1D subproblems are balanced. To achieve a total error for the membrane potential V_m of approximately $8 \cdot 10^{-2}$, we can increase the required splitting timestep width $dt_{\text{splitting}}$ from $5 \cdot 10^{-4}$ ms for the Godunov splitting to $4 \cdot 10^{-3}$ ms for the Strang splitting scheme. This results in a runtime speedup of approximately 7.5.

To evaluate the total speedup of the described numerical improvements, we compare the runtimes without and with the improvements for a complete simulation of the fiber based electrophysiology model coupled with the elasticity model. A cuboid 3D domain is discretized by $2 \times 2 \times 2 = 8$ finite elements for the elasticity model, and we embed $6 \times 6 = 36$ 1D fiber meshes. The number of 1D elements per fiber is varied between 576

and 239 400 to study the scaling behavior of the solvers related to the problem size. The problem is solved in serial to avoid runtime effects introduced by the parallelization.

The baseline implementation uses the Godunov splitting with forward and backward Euler schemes for the 0D subcellular model and the electric conduction model, respectively. The linear system in the 1D problem is solved by a GMRES solver with relative residuum tolerance of 10^{-5} and restart after 30 iterations. Timestep widths of $dt_{0\text{D}} = 10^{-4}\,\text{ms}$ and $dt_{\text{splitting}} = dt_{1\text{D}} = 5 \cdot 10^{-4}\,\text{ms}$ are used. The improved scheme uses the Strang operator splitting with Heun and Crank-Nicolson schemes and timestep widths of $dt_{0\text{D}} = 2 \cdot 10^{-3}\,\text{ms}$ and $dt_{\text{splitting}} = dt_{1\text{D}} = 4 \cdot 10^{-3}\,\text{ms}$. The direct solver is used for the linear system in the 1D problem. The solver for the 3D elasticity problem is the same for both implementations: A Newton scheme with residual tolerance of 10^{-8} is used and coupled to the 0D and the 1D solver with a coupling timestep width of $dt_{3\text{D}} = 1\,\text{ms}$.

The present study and the studies in the next section are executed on the supercomputer *Hazel Hen* at the High Performance Computing Center Stuttgart. This Cray XC40 system contains compute nodes with two Intel Haswell E5-2680v3 processors with a base frequency of 2.5 GHz, 12 cores per CPU, 24 cores per compute node and 128 GB RAM per node.

Figure 9.2 shows the results of this study. In the upper part, the runtimes for different components of the simulation are indicated by different colors in a plot with double logarithmic scale. The runtimes for the baseline implementation are shown by solid lines and the runtimes of the implementation where the improvements have been incorporated are shown by dashed lines. In the lower plot, the speedups from the baseline to the improved implementation are given.

The total runtime of the simulation is given by the black lines in the upper plot. It can be seen that the total runtime results almost completely from the 0D model solver, which is shown by the yellow lines. The 1D solver, given by the red lines, has the second highest contribution. The effects of the data mapping operations between the 3D mesh and the 1D fibers on the runtime are negligible. These data mapping operations consists of the homogenization step from the 1D fibers to the 3D mesh and the interpolation step from the 3D mesh to the 1D fibers.

The runtimes for almost all problem parts increase linearly for increasing mesh resolution of the 1D fibers. Only the runtime of the 3D problem stays constant, as the 3D mesh is unchanged for the different runs.

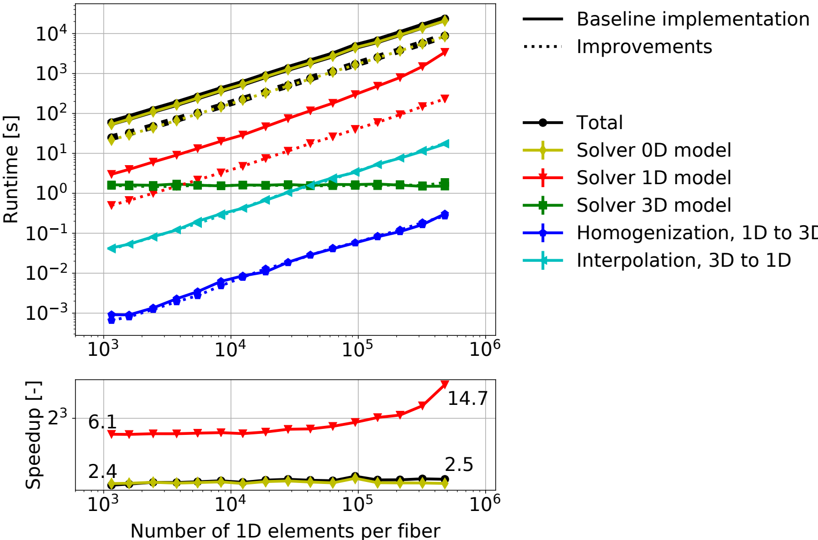


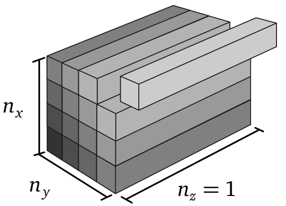
Figure 9.2: Numerical improvements in OpenCMISS: Study to evaluate the speedup of the improved implementation of the fiber based electrophysiology and mechanics model in OpenCMISS.

Significant runtime improvements of the new implementation compared to the baseline implementation can be seen in the lower plot of Fig. 9.2 for the 0D solver and the 1D solver. The speedup for the 0D solver is constant at approximately 2.5. The speedup resulting from the improved linear system solver in the 1D problem is approximately 6.1 for coarse meshes and increases to 14.7 for the finest mesh. This increase for high mesh resolutions results from the higher runtime of the GMRES solver for large problem sizes in the baseline implementation. The overall speedup is similar to the speedup of the 0D problem, as the 0D solver exhibits the dominant runtime contribution to the overall computation.

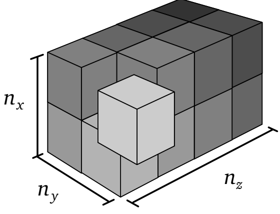
This study shows how numerical investigations can help to reduce the total runtime, in this case by a factor of 2.5. Moreover, the solver of the 0D model has the highest potential for improvements that further speed up the computation.

9.1.2 Parallel Partitioning Strategies

To exploit parallelism and, thus, further reduce the computation times, we implemented a generic domain decomposition for the studied problem in OpenCMISS Iron. Like in OpenDiHu, the 3D mesh can be partitioned to an arbitrary number of $n_x \times n_y \times n_z$ subdomains. The embedded 1D fibers are aligned with the z axis and are partitioned by the same cut planes as the 3D mesh.



(a) “Pillar-like” domain decomposition with $n_z = 1$.



(b) “Cube-like” domain decomposition.

Figure 9.3: Fiber-based electrophysiology and mechanics model in OpenCMISS: Different partitioning strategies for parallelization that have been implemented in OpenCMISS. This figure shows two approaches to partition the domain into 16 subdomains.

Figure 9.3 shows two exemplary partitioning approaches. If the domain is only partitioned in x and y direction, the individual fibers are not split into multiple subdomains. As a result, we get “pillar” subdomains as shown in Fig. 9.3a. An alternative approach is to subdivide the domain in all three coordinate directions, such that the subdomains are approximately cuboid, as shown in Fig. 9.3b.

OpenCMISS Iron already provides the functionality to create parallel partitioned, unstructured meshes. However, every mesh has to be partitioned into non-empty subdomains for all processes. Thus, it is not possible to use individual meshes for the 1D fibers. In the baseline implementation of the model by [Hei13], all 1D fiber meshes are however realized as a single mesh, whose node positions are set according to the positions of the individual fibers. This facilitates the implementation of the 0D subcellular model solvers and 1D model solvers, as the implementation has to deal with only a single mesh.

To allow for an arbitrary partitioning as in Fig. 9.3, we assigned the 1D elements of the single fiber mesh to the same processes as the corresponding subdomains of the 3D

mesh. Furthermore, we reimplemented the data mapping between the 1D mesh and the 3D mesh, which was hard-coded for four processes.

In the following, we investigate the effect of different partitioning strategies on the overall runtime of the solver. The idea is that, for pillar-like partitionings as in Fig. 9.3a, the 1D problems could potentially be solved faster, as the fibers, which are aligned in z-direction, are not subdivided to multiple processes. On the other hand, the partitioning to cubes in Fig. 9.3b requires less communication in the solution of the 3D problem as the cubes minimize the surface of each subdomain and, in consequence, the amount of data to be exchanged. We evaluate how these effects influence the runtimes for the pillar-like partitioning, the cube partitioning and all other possible partitionings specified by numbers of subdomains $n_x \times n_y \times n_z$.

Our test case uses a 3D mesh with 12 × 12 × 144 elements. To reduce the runtime contribution of the 0D/1D electrophysiology problem and the memory consumption of the solver, only two 1D elements per 3D element are included. The numerical parameters are the same as for the improved scenario presented in Fig. 9.2. The simulations are executed on 12 compute nodes of the supercomputer Hazel Hen with 12 processes per node.

We partition the 3D domain to 144 processes using different combinations of n_x, n_y and n_z such that $n_x\, n_y\, n_z = 144$. For every partitioning, we compute the average surface area of the boundary of every subdomain. Figure 9.4 shows the resulting runtime in relation to this average boundary area. The pillar-like partitioning uses 12 × 12 × 1 subdomains and exhibits the largest boundary surface area, corresponding to the last point in Fig. 9.4. The cube partitioning consists of 6 × 6 × 4 subdomains and corresponds to the first data point with the smallest boundary area.

The plot shows that the runtime of the 3D solver increases approximately linearly with the amount of communication, which is expected. The partitioning with the largest average surface area has a runtime that is approximately four times larger than the runtime for the smallest surface area.

Moreover, the plot shows that the partitioning scheme has no significant influence on the runtime of the 1D solver. The reason is that the implementation does not fully reflect the decoupled nature of the individual problems of the fibers. As noted before, one big linear system has to be solved that contains the degrees of freedom of all fibers. The degrees of freedom are ordered by PETSc, such that the nodes within every subdomain

[1]This figure and the following figures have also been published in [Bra18] under a creative commons license.

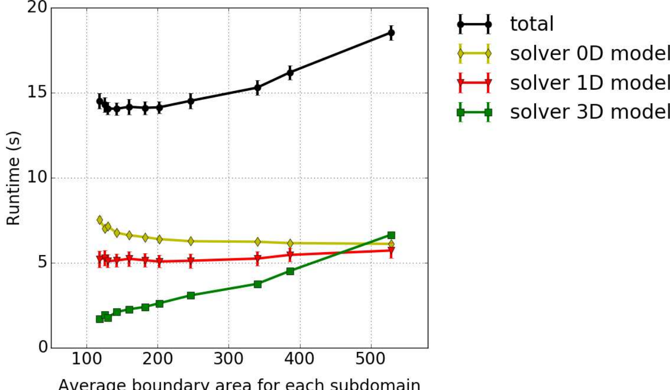


Figure 9.4: Fiber-based electrophysiology and mechanics model in OpenCMISS: Runtime of the solvers for different partition shapes, from cube partitions on the left to pillar partitions on the right.[1]

are consecutive. If a subdomain contains (parts of) multiple fibers, the degrees of freedom of a single fiber are not necessarily consecutive in the solution vector and communication is required in the linear solver.

9.1.3 Weak Scaling Study and Memory Consumption

Next, we evaluate the parallel weak scaling properties of the overall solver. We increase the number of elements in the 3D mesh from 1232 to 8640 and the total number of 1D elements in all fibers from 14 784 to 103 680. Correspondingly, the number of processes increases from 24 to 192, such that the amount of work per process stays approximately constant. Each scenario is computed with two different partitioning schemes, once with pillar-like partitioning and once with cuboid partitioning. For the exact problem sizes, numbers of cores and numbers of elements in the partitions, we refer to the paper [Bra18].

Figure 9.5 shows the resulting runtimes of the different components of the simulation. It can be seen that the runtime stays approximately the same for all problem sizes. The observable differences in runtime within the same solver, especially for the last two data

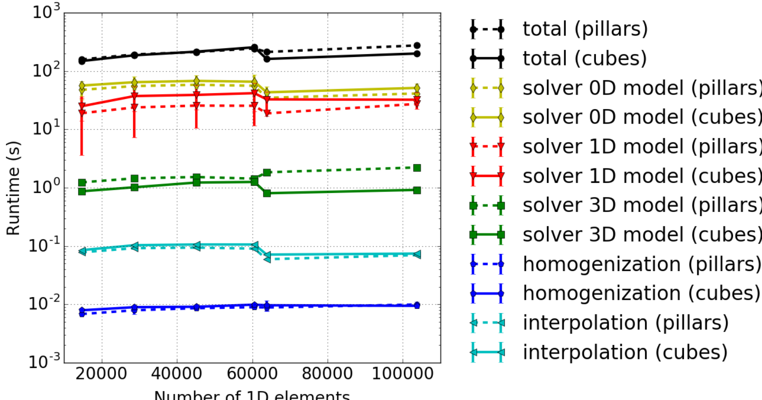


Figure 9.5: Fiber-based electrophysiology and mechanics model in OpenCMISS: Parallel weak scaling study of a scenario with the pillar and cube partitionings.

points, can be explained by slightly different ratios of element counts to process counts, which result from the goal to use the pillars and cube partitioning schemes while not exceeding the available main memory.

The runtimes of the pillar and cube partitioning schemes are depicted by dashed and solid lines, respectively. The pillar partitioning exhibits shorter runtimes for the 1D solver and longer runtimes for the 3D solver compared to the cube partitioning. In total, the runtime is not significantly different for the different partitioning strategies.

A limiting factor for the construction of weak scaling studies with this implementation is the high memory consumption. Figure 9.6 shows the total memory consumption per process at the end of the runtime of the simulations in Fig. 9.5. The used memory is visualized by purple lines. The dashed line again corresponds to the pillar partitioning and the solid line corresponds to the cube partitioning.

A difference between the pillar partitions and the cube partitions is the size of the subdomain surfaces and the corresponding size of the ghost layer. Fig. 9.6 shows the number of 3D ghost elements for the scenarios with cubes and pillars by the black lines. In OpenCMISS, a ghost element on a process is an element that contains ghost nodes, which are owned by a different process. The ghost elements serve as data buffers for communication during the assembly of the finite element matrices, similar to OpenDiHu.

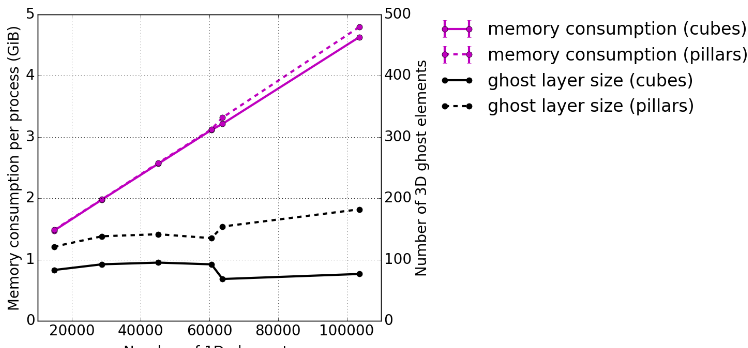


Figure 9.6: Fiber-based electrophysiology and mechanics model in OpenCMISS: Memory consumption per process at the end of the simulation corresponding to the weak scaling study of Fig. 9.5

The plot in Fig. 9.6 shows that the number of ghost elements is higher for the pillar partitioning scheme than for the cubes scheme, as expected. As a consequence, the memory consumption per process is also slightly higher for the pillar partitioning. However, this effect is negligible compared to the high absolute value of the required memory and does not explain this effect.

As can be seen, the memory consumption per process monotonically increases with the total number of 1D elements. At the same time, however, the number of elements per process stays approximately constant in this weak scaling setting. The last data point is close to the memory limit of 128 GB/24 ≈ 4.967 GiB, which is reached when 24 processes are executed on a compute node of the supercomputer Hazel Hen.

The observed large increase in memory consumption results from the organization of parallel partitioned data in OpenCMISS Iron. On every process, global mesh topology information such as mappings between global indexing and local indexing is stored for the element numbers, node numbers and degree of freedom numbers. While this overhead in storage is negligible for moderately parallel scenarios, it counteracts the domain decomposition approach for higher degrees of parallelism.

Numerous functions and algorithms in the OpenCMISS Iron code rely on this type of global information. Thus, eliminating the parallelism constraint by reorganizing the data structures is a highly involved task. Especially the initialization of the parallel partitioning

heavily uses this global information. This initialization includes, e.g., the distribution of elements and nodes to the subdomains on the processes, the determination of the ghost layers and dofs to send to and receive from neighbor processes, and the setup of local numbers for elements, nodes and degrees of freedom.

We addressed the elimination of this use of global topology information in the initialization steps and developed and implemented appropriate local algorithms in OpenCMISS Iron. This resulted in major code changes that are difficult to oversee, also because of the lacking object orientation in the code base and the difficulty to comprehensively test the functionality. Creating the required set of unit tests for nearly all functionality of OpenCMISS would be a large task that remains to be done. Thus, these code changes could not be merged into the main trunk of OpenCMISS.

Even with these code changes, the memory problem is not yet solved. Another problem prior to the initialization step is that the mesh has to be specified from the user code in a global data structure. It is currently not possible to specify a mesh in a distributed way. Thus, OpenCMISS Iron can only use meshes that initially fit into the main memory on every single core.

Moreover, another issue is concerned with the data structures for matrices. Each process stores its local row indices and additionally a map from global to local row indices for all dofs of the global problem. This global-to-local map also contributes to the bad weak memory scaling and has to be eliminated as well. One possible approach is to use hash maps and only store the relevant portion of the mapping on every process. Work towards resolving this issue has been started by Lorenzo Zanon at the former SimTech Research Group on Continuum Biomechanics and Mechanobiology at the University of Stuttgart.

One reason for the generic mapping of matrix rows, which uses global information, is that OpenCMISS Iron does not restrict discretization schemes to the finite element method, where the system matrix can be assembled from local element matrices within the subdomains. An example for a different supported scheme is the boundary element method.

In addition, there exist more parts in the code that use a similar global-to-local mapping and would also have to be changed to allow for a constant memory consumption per process, e.g, the boundary condition handling and the data mapping between the 3D mesh and the fibers.

In summary, fixing the issue of non-scaling memory consumption in OpenCMISS Iron, which was revealed in Fig. 9.6, corresponds to redeveloping a significant portion of the

code. To preserve the generic functionality of OpenCMISS, some changes would require new algorithmic considerations and complex workarounds. This development effort would have to be quick enough to keep up with the independent development of the normal OpenCMISS branch. After completion, the merge back into the main software trunk would only be possible if the branches had not diverged too far and after significant efforts have been put into testing and preserving the feature set of OpenCMISS.

On the other hand, developing the missing functionality from scratch and making sensible restrictions on the generality of the solved problems and used methods requires possibly less effort and allows considering design goals such as performance, usability and extensibility from the beginning. In this sense, the OpenDiHu software project can be seen as a complement to OpenCMISS Iron with better performance characteristics. The mentioned restrictions for OpenDiHu are, e.g., the exclusive use of the finite element method and Cartesian coordinates and the use of parallel partitioned structured meshes instead of the more complex parallelization of unstructured meshes.

9.2 Performance Studies of the Electrophysiology Solver in OpenDiHu

After the previous studies with OpenCMISS, we now consider the performance of the OpenDiHu software. In the following sections, we investigate the runtime performance of the solvers for the electrophysiology part of the multi-scale model in OpenDiHu.

9.2.1 Evaluation of Compiler Optimizations

One difference in the data organization in OpenDiHu compared to OpenCMISS Iron lies in the transposed memory layout for the storage of multiple instances of the 0D subcellular model. If the `simd` optimization type in the `CellmlAdapter` class is used, the components of the state vector $\mathbf{y}$ of all 0D model instances are stored consecutively. This storage order is the SoA memory layout, which was described in Sec. 7.6.2. It enables the compiler to automatically employ SIMD instructions and, thus, exploit instruction-level parallelism.

We study the auto-vectorization performance of the GNU, Intel and Cray compilers to determine the effect of these SIMD instructions on the total runtimes of the solver. The simulated scenario consists of one muscle fiber mesh with 2400 nodes, on which

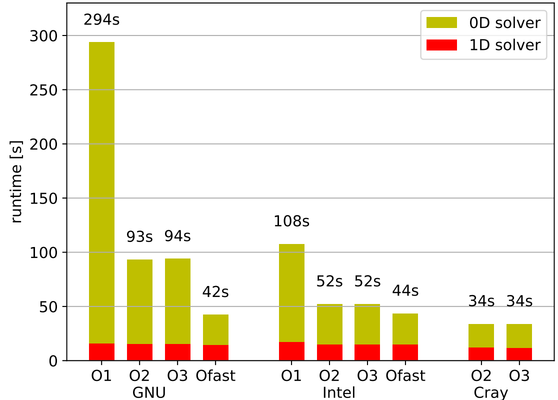


Figure 9.7: Electrophysiology Solver in OpenDiHu: Comparison of auto-vectorization in different compilers. Runtime of the 0D and 1D solvers in the fiber based electrophysiology model with simd optimization type for different compilers and optimization flags.

the monodomain equation Eq. (5.11) is solved. The subcellular model of Shorten et al. [Sho07] is used. The used timestep widths are $dt_{0D} = 10^{-3}\,\text{ms}, dt_{1D} = dt_{\text{splitting}} = 3 \cdot 10^{-3}\,\text{ms}$, and the model is computed up to a simulation end time of $t_{\text{end}} = 20\,\text{ms}$.

We run the study on one compute node of the supercomputer Hazel Hen at the High Performance Computing Center in Stuttgart. This Cray XC40 system contains two 12-core Intel Haswell E-2680v3 CPUs with clock frequency of 2.5 GHz per dual-socket node, yielding 24 processors per compute node and contains 128 GB memory per compute node.

Figure 9.7 shows the runtime of the 0D and 1D model solvers for the three different compilers with varying optimization flags. As expected, the runtime of the 1D solver is not affected by the choice of the compiler. The runtime of the 0D solver, however, varies greatly, as the compilers with different optimization flags are able to vectorize the code to a different extent.

For all compilers, the runtime decreases when a higher optimization level is chosen. A significant drop to less than half of the runtime is observed when switching from the O1 to the O2 optimization level for the GNU and for the Intel compiler. This is mainly the result of the SIMD instructions, which are enabled starting from the O2 levels. The change to

the aggressive optimization levels `O3`, which enables all available optimizations such as inlining and code transformations does not improve the runtime any further, for all three evaluated compilers. Thus, vectorization is the main driver for good subcellular solver performance.

Another significant decrease in runtime can be observed for the `Ofast` optimization flag. For the GNU compiler, the runtime decreases again to less than half of the previous value. For the Intel compiler, the decrease is less prominent with approximately 15 %.

The `Ofast` level performs optimizations that potentially change the behavior of the code. Especially floating-point arithmetic does no longer comply to the standardization rules of IEEE and ISO. Only finite numbers can be represented and the compiler is allowed to perform transformations in formulas that are mathematically correct, but not in terms of propagating rounding errors. The calculated values are correct as long as no invalid operations such as divisions by zero occur. The precision may decrease or even increase compared to `O3`. This is usually not an issue for the given simulations, however, divergence of the numerical solvers is not automatically detectable with `Ofast` in our code as no infinity values can be represented.

The comparison between the compilers shows that the Intel compiler creates faster assembly code than the GNU compiler, and the Cray compiler creates faster assembly code than the Intel compiler for the same optimization levels. The performance of the `Ofast` flag is comparable between the GNU and the Intel compiler. In total, the Cray compiler yields the best performance on the Cray hardware used in this evaluation.

The Cray compiler has a “whole-program mode”, which collects static information about all compilation units and allows, e.g., application-wide inlining during the linking step. The faster runtime is traded for longer compilation times. In our example, the compilation duration increases from approximately 10 min for the GNU and Intel compilers to over 2 h for the Cray compiler.

For all further simulations, we use the GNU compiler with the `Ofast` optimization flag, as it is freely available on all systems, has fast compilation times and showed good performance.

9.2.2 Evaluation of Code Generator Optimizations

Apart from the automatic optimizations by the compiler, the code can also be manually optimized by using efficient data structures and algorithms. Section 7.6.2 presents various

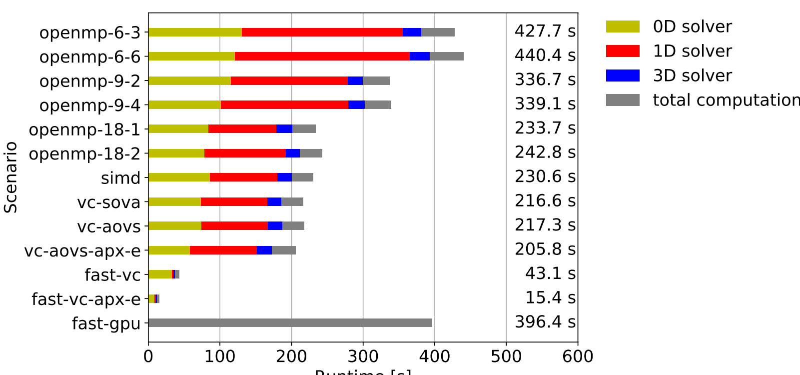


Figure 9.8: Electrophysiology Solver in OpenDiHu: Evaluation of various code optimizations for the subcellular model solver. Comparison of runtimes for the 0D, 1D and 3D model solvers with different optimization types in the code generator.

optimization options in our code generator, which potentially have an influence on the runtime of the subcellular model solver. We compare all optimization options for a scenario of a comprehensive surface EMG simulation.

The considered scenario solves the monodomain equation Eq. (5.11) on every 1D muscle fiber domain and is coupled to a 3D mesh where the bidomain equation Eq. (5.9a) is solved. No body fat domain is considered in this scenario. We simulate 625 muscle fibers with 1481 nodes per fiber mesh and the subcellular model of Hodgkin and Huxley [Hod52a]. This leads to a total number of 3 702 500 degrees of freedom to be solved for the 0D and 1D models. We run the code in parallel with 18 processes and a parallel partitioning of the 3D domain into $3 \times 2 \times 3$ subdomains. Thus, every muscle fiber domain is distributed to three different processes. The 3D mesh contains 5239 nodes. Timestep widths of $dt_{1D} = 10^{-3}\,\text{ms}$, $dt_{3D} = dt_{\text{splitting}} = 3 \cdot 10^{-3}\,\text{ms}$ and an end time of $t_{\text{end}} = 10\,\text{ms}$ are used, and file output is disabled for this study.

We use an Intel Core i9-10980XE processor with 18 cores, base frequency of 3 GHz, maximum boost frequency of 4.8 GHz, cache sizes of 24.8 MiB, 18 MiB and 576 KiB and 31 GiB main memory. This processor is listed in the upper price segment of consumer hardware and can be considered a typical hardware for individual workstations in scientific research.

Figure 9.8 presents the results of the study for all available optimization types in our code generator. For every scenario, the bar chart shows the runtimes of the 0D subcellular solver in yellow color, the runtime of the 1D electric conduction solver in red color, the runtime for the 3D bidomain solver in blue color and the remaining runtime of the coupled solver scheme, which involves, e.g., data transfer between data structures and inter-process communication, in gray color. The presented runtimes are averaged over several runs and over all processes per run.

The first six bars correspond to the `openmp` optimization type, which places OpenMP pragmas in the code and employs thread-based, shared memory parallelism. The scenario `openmp-`i`-`j refers to i MPI processes in total with j threads on every process. The problem is partitioned into i subdomains and the j OpenMP threads per subdomain simultaneously operate on the shared data structures of the subdomain. As a result, in the scenarios `openmp-6-3`, `openmp-9-2` and `openmp-18-1`, 18 threads are executed in total on the processor with 18 physical cores. The other scenarios, `openmp-6-6`, `openmp-9-4` and `openmp-18-2`, employ 36 threads.

It can be seen that each set of two scenarios with the same number i of processes and varying number j of threads, i.e., `openmp-6-3` and `openmp-6-6`, `openmp-9-2` and `openmp-9-4`, and `openmp-18-1` and `openmp-18-2` has similar total runtimes. This shows that the runtime is reduced mainly as a result of MPI parallelization. The distribution of the runtime to the solvers allows further insights. Between the two scenarios with the same number of processes, the runtime of the 0D solver decreases. This is a result of the higher number of OpenMP threads that is used to perform the same amount of work. At the same time, the runtimes of the 1D solvers increase, which is due to the multi-threaded solution of the 1D problem in the solver library PETSc, which we consider as a black box.

The effect of OpenMP parallelism on the 1D solver is even higher than on the 0D solver in this example. As the code generator using OpenMP parallelism is only responsible for the 0D problem, the performance of the 1D problem depends only on the partition size and workload defined by the parallel partitioning with i MPI processes. A reduction of the MPI parallelism has more impact on the runtime than the resulting increased parallelism of the 0D solver. Thus, the scenarios with high degrees of OpenMP parallelism, e.g., scenario `openmp-6-6`, show a worse performance than the scenarios with higher MPI parallelism, e.g., scenario `openmp-18-1`.

The next bar in Fig. 9.8 presents the runtime of the `simd` optimization type. The code uses the `SoA` memory layout and the program is run with 18 MPI processes. As in all

scenarios of this study, the GNU compiler with the `Ofast` flag is used and automatically vectorizes the subcellular model equations. The `simd` scenario is very similar to the `openmp-18-1` scenario, except that the OpenMP pragmas are omitted in the generated code. As a result, the runtimes are also similar to this scenario. A slight reduction in runtime is seen that results from the missing OpenMP initializations before every loop.

While the `simd` scenario relies on the auto-vectorization capabilities of the compiler, the `vc` scenarios, which are considered next, explicitly employ vector instructions, abstracted by the *Vc* and *std-simd* libraries.

The `vc-sova` scenario uses the Struct-of-Vectorized-Array (SoVA) memory layout and the bar chart shows a slightly lower runtime of the 0D solver compared to the Array-of-Vectorized-Struct (AoVS) memory layout in the `vc-aovs` scenario.

The next considered scenario is `vc-aovs-apx-e`. It is the same as `vc-aovs` except that the exponential function is approximated by $\exp^*(x) = (1 + x/n)^n$ for $n = 1024$, as given in Eq. (7.7). The results show that this reduces the runtime of the 0D solver from 74.24 s to 58.02 s, which is a reduction by approximately 22 %.

Instead of generating code only for the 0D subcellular model and solving the 1D subcellular model using a direct solver of PETSc, as in all considered scenarios so far, we can also directly generate combined solver code for the 0D and 1D models and use the Thomas algorithm for the computation of the 1D model. This is done in the `fast-vc` scenario and reduces the runtime by a factor of nearly 5. In this approach, the exponential function can also be exchanged by the approximation in Eq. (7.7). This is done in the `fast-vc-apx-e` scenario and further decreases the total runtime to now only 15.4 s.

The two `fast-vc` scenarios demonstrate the performance of the AVX-512 vector instruction set that is available on the used Intel processor. The study shows that its potential is only fully exploited, if the explicit vector instructions are generated in the code, as done in the `vc` scenarios.

The solution times for the last two mentioned scenarios can be further reduced if only those subcellular model instances are computed that are not in equilibrium. If enabled, this reduction depends on the activation pattern of the fibers. For the sake of the present study, which aims to compare runtimes of the code generator, this option is not evaluated and, thus, disabled.

The last considered optimization type in the code generator is presented in the scenario `fast-gpu`. In this scenario, the program is only run with one MPI process. The total computation of the 0D and 1D models is offloaded to a GPU using OpenMP 4.5 pragmas

in the generated code. We use the same simulation scenario and CPU hardware for this run as for the other scenarios. The used computer is equipped with an NVIDIA GeForce RTX 3080 GPU with 8704 CUDA cores, 10 GB of memory and a Thermal Design Power (TDP) of 320 W. The processing power is 29.77 TFLOPS for single precision and 465.1 GFLOPS for double precision operations. We use only double precision operations for the computation of the models.

In this scenario, only the total runtime is measured. The bar chart shows a total solver runtime of 396 s, which is slower than the optimized CPU computations. Possible reasons are that the used GPU is targeted at single precision performance, and that the employed GPU code by the OpenMP functionality of the GNU compiler is not optimal.

In the previously considered example, which uses the Hodgkin and Huxley subcellular model with a state vector $\mathbf{y} \in \mathbb{R}^4$, the amount of computational work in the 0D and in the 1D solver was in the same range. Other 0D subcellular models exist that have higher workloads. In the next study, we repeat the same measurements as before with the subcellular model of Shorten et al. [Sho07], which has a state vector $\mathbf{y} \in \mathbb{R}^{57}$. Whereas the solver for the model of Hodgkin and Huxley needs to compute 4 ODEs and 9 algebraic equations in every timestep, the solver for the Shorten model computes 57 ODEs and 71 algebraic equations in every timestep.

As the computational effort to solve one instance of the subcellular model increases, we adjust the simulation scenario for the next study. We use 49 fibers with 1481 nodes each and a 3D finite element mesh with linear ansatz functions and a total of 23 696 degrees of freedom. The total number of degrees of freedom in all meshes is 4 087 560, which is similar to the number 3 707 739 in the previous study. The simulation end time is 3 ms. For this subcellular model, smaller timestep widths of $dt_{\text{splitting}} = dt_{\text{1D}} = dt_{\text{0D}} = 2.5 \cdot 10^{-5}$ ms and $dt_{\text{3D}} = 10^{-1}$ ms are used as required to ensure convergence of the solver for this subcellular model.

Figure 9.9 shows the resulting runtimes for different scenarios in a bar chart analog to Fig. 9.8. It can be seen that the solver time for the 0D model now dominates the total runtime in all scenarios. In the openmp-i-j scenarios, the runtime for the 0D solver decreases as before, if more threads are used in total. Contrary to the previous study, the total runtime profits from this runtime reduction, as the 0D part is significant enough for the total runtime. Another difference to the results of the previous study is that the durations for the 0D model are nearly the same for every combination of number of MPI processes i and number of OpenMP threads j. This shows that the overhead of starting

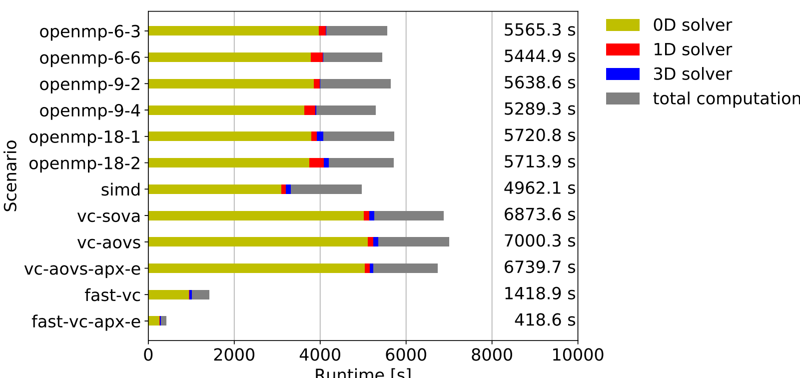


Figure 9.9: Electrophysiology Solver in OpenDiHu: Comparison of runtimes for different optimizations in the code generator, for the compute-intense Shorten subcellular model.

the OpenMP threads, which in the previous study was responsible for larger compute times of the 0D models, is now amortized by the larger overall workload.

The performance in the `simd` scenario is, again, comparable to the performance of the `openmp-18-1` scenario and shows a slightly smaller runtime due to the missing OpenMP thread initializations.

A difference to the previous study can be seen for the `vc` scenarios. In the present study with the subcellular model of Shorten et al., the runtimes for the `vc-sova`, `vc-aovs`, and `vc-aovs-apx-e` are all higher than for the auto-vectorized scenarios. In contrast, the `vc` scenarios showed a large reduction in runtime in the study with the Hodgkin and Huxley subcellular model.

This effect originates from the operations required to evaluate the subcellular equations. The Shorten model contains many $\log(x)$ function evaluations. These are especially compute intense and, in addition, not supported in the abstraction layer of the AVX-512 instructions provided by the *std-simd* library. Instead, the library employs their non-vectorized counterparts. The auto-vectorization of the compilers, however, is able to employ the respective vectorized functions, which explains the better performance in the `openmp` and `simd` scenarios.

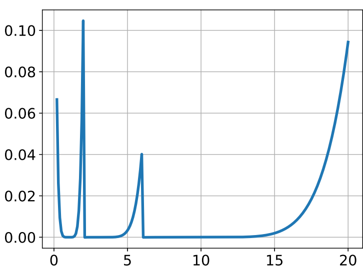


Figure 9.10: Electrophysiology Solver in OpenDiHu: Relative error of the piecewise Taylor approximation of the log function as used in the vectorized simulation code.

We expect that, in the future, the respective functionality will become available in the *std-simd* library, which would automatically increase the performance for these optimization types. For processors without AVX-512 support, but with the AVX2 instruction set, the library *Vc* is used, which supports the respective functions and, thus, yields the expected performance in the `vc` scenarios. Whereas AVX-512 has a SIMD lane width of eight double values, AVX2 only supports SIMD lanes with 4 double values.

To mitigate the effect of the missing $\log(x)$ vectorization, we replace the log function by a numerical approximation, in addition to the approximated exp function. We define the approximated logarithm function $\log^*(x)$ by its piecewise Taylor polynomials of sixth order around the points $x = 1$, 3 and 9 with discontinuities at the points $x = 2$ and $x = 6$. Figure 9.10 shows the absolute relative error for the range between 0.2 and 20 which, in this range, is bounded by 0.105. However, better convergence of the 0D-1D problem is achieved, if the approximated log function $\log^*$ is the inverse of the approximated exponential function $\exp^*$. Therefore, we apply one Newton iteration of the problem

$$F(y) = \exp^*(y) - x \stackrel{!}{=} 0$$

to the log value y computed by the Taylor approximation. The Newton iteration consists of subtracting $(1 - x/\exp^*(x))$ from the computed result y. Thus, it only involves one evaluation of the approximated exponential function.

The scenario `fast-vc` in Fig. 9.9 generates unified solver code for both 0D and 1D models, but does not include this approximation. The approximated exponential and logarithm functions are included in the scenario `fast-vc-apx-e`. As a result, it can be seen that the total runtime is largely reduced compared to the auto-vectorized scenarios.

How To Reproduce

The simulations in this section use the example `examples/electrophysiology/fibers/fibers_emg` with the variables files `optimization_type_study.py` and `shorten.py`. The commands for the individual runs are executed by the following scripts:

```
cd $OPENDIHU_HOME/examples/electrophysiology/fibers/fibers_emg/
  ↪ build_release
../old_scripts/run_optimization_type_study.sh
../old_scripts/run_optimization_type_study_shorten.sh
```

The utility to create the plots from the generated `logs/log.csv` files can be found in the repository at github.com/dihu-stuttgart/performance in the directory `opendihu/18_fibers_emg`:

```
./plot_optimization_type_study_shorten.py
./plot_optimization_type_study.py
```

9.3 Parallel Strong Scaling and Comparison with OpenCMISS Iron

After the performance of different optimization types has been evaluated for a scenario with a single number of MPI processes in the last section, we now conduct a strong scaling study with the optimization type `fast-vc-apx-e`, which was found to be the most performant, and compare the runtimes to the reference software OpenCMISS Iron.

9.3.1 Evaluation of Runtimes

For a fair comparison, we take care to exactly compute the same scenario with both software packages. The simulated scenario uses the same model as in the previous section: fiber based electrophysiology with the monodomain model given by Eq. (5.11) on every muscle fiber, including the 1D electric conduction along the fibers and the 0D subcellular model of Shorten et al. [Sho07]. The fibers are coupled with the bidomain equation in Eq. (5.9a), which is solved on the 3D domain to yield the EMG signals. No fat layer is considered in this study, as this feature is not available in our OpenCMISS implementation.

The simulated scenario contains 81 fibers with 1480 elements each, a coarse 3D mesh with 775 nodes and 6 718 591 degrees of freedom in total. We use our improved OpenCMISS setup, which is discussed in Sec. 9.1.1 and employs the second order numerical timestepping schemes and the improved linear solvers: The monodomain model is solved using a Strang operator splitting with Crank-Nicolson and Heun's methods. A conjugate-gradient solver is used for the linear system of the bidomain equation.

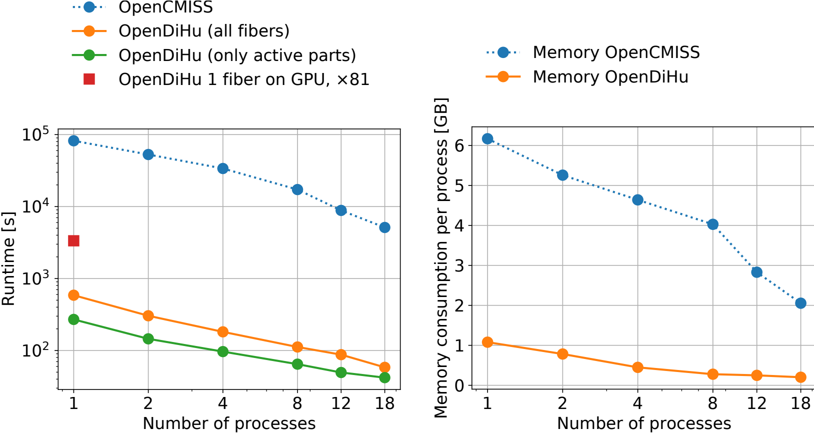
We use timestep widths of $dt_{0D} = 10^{-4}$ ms, $dt_{1D} = dt_{\text{splitting}} = 5 \cdot 10^{-4}$ ms, $dt_{3D} = 10^{-1}$ ms and a simulation end time of $t_{\text{end}} = 2$ ms. During this time, the resulting values are written to output files after every 0.1 ms. The fibers are assigned to 10 MUs that are activated in a ramp every 0.2 ms from $t = 0$ ms to $t = 1.8$ ms.

Within the strong scaling study, the same scenario is computed with different numbers of processes, ranging from one to 18 in this case. We use the cubes partitioning strategies presented in Sec. 9.1.2 in both OpenCMISS and OpenDiHu. The study is executed on the same Intel Core i9-10980XE processor as the studies in the previous section. We measure the total user time of the simulation program, which includes the runtimes for initialization, computation of system matrices and the duration of file output. However, the majority of the runtime in this scenario is spent in the numerical solvers.

In OpenDiHu, we use the setup corresponding to the `fast-vc-apx-e` scenario in the last section with enabled approximation of log and exp functions. We measure the runtime of two variants. The first variant computes all subcellular models and performs the same work as the OpenCMISS Iron implementation. In the second variant, the adaptive computation described in Sec. 7.4.3 is enabled, which only computes fibers that have been activated and the subcellular models that are not in equilibrium. For the chosen ramp activation pattern of the MUs, the second variant computes approximately only half of the subcellular model instances.

Figure 9.11a shows the resulting runtimes in this study. It can be seen that the runtime decreases monotonically for higher numbers of processes for all three tested simulations. The OpenDiHu implementation exhibits lower runtimes for all numbers of processes. The reduction in runtime between OpenCMISS Iron and the first OpenDiHu variant is given by a factor of approximately 100 with a maximum factor of 186 for 4 processes. In addition, the second OpenDiHu variant approximately halves the runtimes as expected, because only half of the subcellular models are computed.

In summary, the improvements to the EMG simulation software, which are described in this work, include the numerical improvements in Sec. 9.1.1 with a speedup of 2.5, the software improvements with a speedup of over 100 and a measured maximum of 186,



(a) Runtime of the simulation programs for OpenCMISS (blue) and two variants of OpenDiHu (orange and green), see the description in the text for details.

(b) Memory consumption per process at the end of the program.

Figure 9.11: Electrophysiology Solver in OpenDiHu: Strong scaling study of fiber based electrophysiology and comparison between the implementations of OpenDiHu and OpenCMISS Iron. The same scenario is solved with both software packages and for increasing numbers of processes from one to 18.

and the algorithmic improvement of adaptive 0D model computations, whose speedup factor is scenario dependent. In the present study, the two latter factors, i.e., the speedup between the improved OpenCMISS Iron software and the OpenDiHu scenario with adaptive computation, give a combined maximum speedup of 363 for the measurement with two processes.

Moreover, the computation of this study was also carried out with the gpu optimization type in OpenDiHu, using the same GPU as in the last section. One process was started on the CPU, which offloaded the computational work of the 0D and 1D problems to the GPU. However, the GPU memory was not sufficient for the computation of all 81 fibers. Therefore, we only compute one fiber, but keep the rest of the simulation scenario equal to the other measurements. The req square in Fig. 9.11a shows the measured runtime multiplied by the factor 81 for compensation. As in the studies of the previous section, the computation on the GPU has higher runtimes than the computation on the CPU.

As the memory consumption was a limiting factor for parallelism in OpenCMISS Iron

as shown in Sec. 9.1.3, we also measure the memory consumption per process at the end of the simulation in both software frameworks. Figure 9.11b shows the result for OpenCMISS and OpenDiHu. The two variants of OpenDiHu have the same memory consumption characteristics, as the only difference between the variants is that the computation of certain subcellular models is switched on or off.

It can be seen that the increased parallelism leads to a reduction of the used memory per process in both pieces of software. OpenDiHu approaches a saturation value of 200 MiB for eight and more processes. For OpenCMISS, the memory consumption is higher, but reduces more quickly also for higher numbers of processes. However, the relation between the two curves increases from a value of 6.168 GiB : 1.078 GiB $= 5.7$ for one process to 2.054 GiB : 0.202 GiB $= 10.2$ for 18 processes.

As a result, this study shows a large memory efficiency improvement in OpenDiHu compared with the OpenCMISS Iron software. For OpenCMISS, the memory scaling in this parallel strong scaling scenario is not as bad as in the parallel weak scaling considered in Fig. 9.6 in Sec. 9.1.3. However, the total memory for all processes still increases to approximately $18 \cdot 2.054\,\text{GiB} \approx 37\,\text{GiB}$, which is higher than the main memory capacity of the used processor.

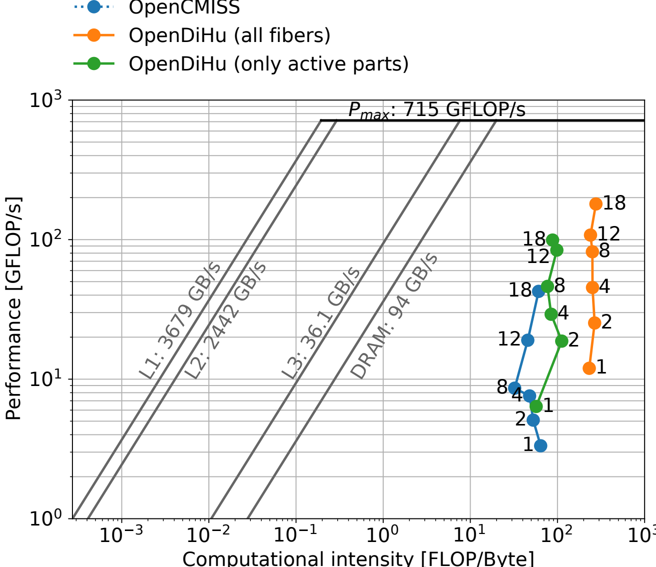


Figure 9.12: Electrophysiology model in OpenDiHu: Roofline model of the strong scaling study. The blue, green and orange data points correspond to the runs of the OpenCMISS and OpenDiHu variants, as given in Fig. 9.11a.

9.3.2 Roofline Model

To further investigate the computational behavior, we also present the performance measurements of the solvers in a roofline model. Figure 9.12 shows the resulting diagram with the data points of all CPU runs in the strong scaling study of Fig. 9.11a. The x axis shows the computational intensity of the simulation, which is measured in double-precision floating-point operations (FLOP) per byte of data that are transferred between the CPU and the main memory and caches. The y axis measures the performance in GFLOP per second. The highest possible performance is given by the peak performance of the processor, which is $P_{\text{max}} = 715\,\frac{\text{GFLOP}}{\text{s}}$ in this case. Furthermore, the performance is limited by the amount of payload data that can be transferred to the CPU over the memory bus. The memory bandwidths of the L1, L2 and L3 caches and the main memory (DRAM) correspond to the shown diagonals in Fig. 9.12 and form the “roofline” of the model.

We measured the memory bandwidths of the Caches and the peak performance using the Empirical Roofline Tool [Yan20]. The main memory bandwidth was retrieved from the processors' documentation.

To locate the simulation runs of the study in the roofline model, we used hardware counters to count floating point instructions and memory access operations. For the runs with OpenCMISS Iron and OpenDiHu, we started the hardware counters 90 s, respectively 15 s after the beginning of the simulation, such that the initialization phase was not included in the measured data. The counters were kept active for 15, 30 and 60 seconds, depending on the expected runtimes of the different runs. The counted numbers of events were then divided by the acquisition time to yield the required rates of memory bandwidth and floating-point performance.

Figure 9.12 shows the measured points in the roofline model corresponding to the curves in Fig. 9.11a. All data points are located at the right-hand side of the memory bandwidth limits, which indicates that the simulation is compute bound. The highest computational intensity and performance are both achieved by the OpenDiHu variant given in orange color, which computes all fibers and subcellular models regardless of their activation state. The values for the adaptive variant given in green color are lower, as fewer computations are performed and a higher portion of the runtime and compute power is spent on determining which subcellular model has to be computed. The two metrics are lowest for the OpenCMISS runs given in blue color.

The roofline diagram shows the data points for all parallel runs and the number of processes is noted in the plot. The run with 18 processes is the most meaningful, as this means that the whole processor is employed. The largest performance for the OpenDiHu run in orange color has a value of 180.157 $\frac{\text{GFLOP}}{\text{s}}$ which corresponds to 25.2 % of the peak performance and is a very good value. The rated 100 % of peak performance for processors are practically unreachable. For example, the peak performance assumes only fused multiply add operations and requires a power management that maximizes the employment of the boost clock frequency in the processor. These conditions are not fulfilled in our computations of realistic models and scenarios. The performance values of the runs with 18 processes for the green and blue data points are 13.9 % and 6.0 %, respectively.

Furthermore, the measurements with lower process counts can also be assessed with a scaled down peak performance according to the fraction of used cores. However, this assessment is slightly off, as, e.g., the CPU can use a higher clock frequency, if only the heat dissipation of one active core has to be compensated. The performance for the OpenDiHu

run with one process given by the orange point is $11.966\,\frac{\text{GFLOP}}{\text{s}}$, which corresponds to 30.1 % of the fractional peak performance of $715\,\frac{\text{GFLOP}}{\text{s}}/18 = 39.7\,\frac{\text{GFLOP}}{\text{s}}$.

How To Reproduce

The scripts to run the studies in this scenario and to create the plots are available in the repository at github.com/dihu-stuttgart/performance in the directory `opendihu/20_fibers_emg_avx_opencmiss`:

```
./0_run.sh
```

The directory also contains a script that performs all steps to install OpenCMISS, if needed.

Note, the studies in the previous and the current sections were carried out on the computer with hostname `pcsgs05` in the institute network at the time of writing.

9.4 Performance Measurements on the GPU

In the previous two sections, measurements were made on a GPU, which produced worse results than the CPU code. The used GPU was an NVIDIA GeForce RTX 3080, a high-end consumer graphics card, which mainly targets graphics rendering performance using single-precision operations. The ratio between double-precision and single-precision performance is 1:64. However, single-precision calculations were found to not yield a stable subcellular model solver, as the precision is too low.

In this section, we conduct two studies, where the first study uses the same GPU hardware as before. The second study is executed on an NVIDIA Quadro GP100 GPU, which has a double-precision to single-precision performance ratio of 1:2. The rated double-precision performance of the Quadro card is 5.168 TFLOPS, which is ten times higher than the value of 465.1 GFLOPS for the GeForce card.

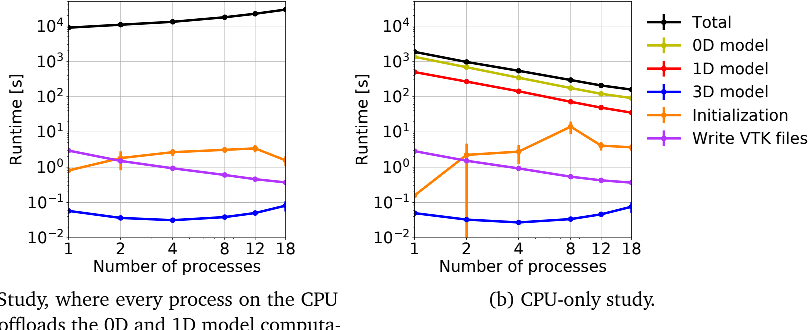
The CPU hardware connected with the Quadro card contains a dual-socket CPU with two 12-core Intel Xeon Silver 4116 processors with 2.1 GHz base frequency and 3 GHz maximum turbo frequency, yielding a total core count of 24, and being equipped with main memory of 188 GiB.

Computational hardware can be compared by its average thermal design power dissipation (TDP). For the studies in the previous two sections, the TDP values for the used CPU and GPU were 165 W and 320 W. For the second study in the current section, the values for CPU and GPU are $2 \cdot 85\,\mathrm{W} = 170\,\mathrm{W}$ and 235 W. This indicates that the employed hardware is in a comparable electrical power range. However, the GPU is more specialized for our double-precision needs.

9.4.1 Strong Scaling with the GPU for the Hodgkin-Huxley Model

While the studies in the last two sections only used one process on the CPU, which managed the offloaded computations on the GPU, we now additionally consider parallelism on the CPU.

The first study compares the strong scaling with and without GPU acceleration. The runs with GPU acceleration partition the computational domain as usual to multiple subdomains, which are handled by dedicated processes on the CPU. The solution of the 0D and 1D models is performed on the GPU, and every process independently transfers its part of the computational work to the same GPU. The 3D model is fully solved using



(a) Study, where every process on the CPU offloads the 0D and 1D model computations to the GPU.

(b) CPU-only study.

Figure 9.13: Electrophysiology model in OpenDiHu: Strong scaling study with and without GPU usage. A scenario with 169 fibers and the subcellular model of Hodgkin and Huxley is simulated. The vertical bars indicate the standard deviation of the runtimes in the set of measurements, which consists of multiple runs and the values of all processors in every run.

MPI parallelism on the CPU. We compare this setup with a pure CPU based strong scaling study.

The scenario solves the fiber based electrophysiology model without fat layer with 169 fiber meshes of 1480 elements each and a 3D mesh with 1984 elements. The subcellular model of Hodgkin and Huxley [Hod52a] is used. The computation uses the same numerical parameters as in the first study in Sec. 9.2.2, a 3D solver timestep of $dt_{3D} = 4 \cdot 10^{-1}$ ms and a simulation end time of 10 ms. Moreover, the setup equals the settings of the `fast-vc` and `fast-gpu` scenarios in Sec. 9.2.2.

Figure 9.13 presents the results for the two studies with and without GPU usage. Figure 9.13b shows the runtimes of different parts in the simulation of the CPU-only strong scaling study. It can be seen that the 0D computations account for most of the runtime, followed by the 1D computations. The 0D computations involve the solution of the Hodgkin-Huxley subcellular model. The 1D computations consists of solving 1D problems in serial using the Thomas algorithm. The 3D solver time is negligible as the 3D problem is only solved every 13 333 timesteps. The 3D solution is only required right before the VTK file output step for the EMG values. This step occurs every 0.4 ms, which corresponds to an EMG sampling frequency of 2.5 kHz. The runtimes for initialization and the file output itself are also very low compared with the runtimes of the computations.

Figure 9.13b shows that the total runtime decreases with higher process counts in

this strong scaling study. The parallel efficiency $E_p = T_1/(T_p\, p)$ reaches $E_p = 65.2\,\%$ for $p = 18$ processes. We observe that the 0D and the 1D solver and the VTK file output have good strong scaling properties, whereas the initialization and the solution of the 3D model contain serial code portions that prohibit optimal scaling.

Figure 9.13a shows the analog study, where the 0D and 1D computations are offloaded to the GPU. The runtimes of these individual model parts are not explicitly measured, only the total runtime is known.

The plot shows an increasing total runtime for higher CPU parallelism. The runtimes for initialization, file output and the 3D solver are equal to the CPU-only study. The increasing total runtime shows that the GPU is better at solving the complete 0D and 1D problems given by one CPU process than at the same computation, but split to several parts and provided by different MPI processes. The benefit of using multiple CPU processes to interface the GPU in this study is, thus, only that the VTK output functionality gets parallelized. However, this effect is negligible.

An absolute comparison between the runtimes in Fig. 9.13a and Fig. 9.13b also reveals that the scenarios for one to 18 processes using the GPU have 4.8 to 181 times longer total runtimes. In this study, the memory transfer between the CPU and the GPU has a low influence on the total runtime, as this transfer only happens before and after the 3D model is solved. The measured runtimes, therefore, correspond to the computation on the GPU.

9.4.2 Evaluation of Hybrid CPU and GPU Computation for the Shorten Model

Whereas previously, the subcellular model of Hodgkin and Huxley was solved on the GPU, we now switch to the more compute intense model of Shorten et al. [Sho07]. This model has higher memory demands, such that it is not possible to solve it with OpenMP 4.5 for a muscle fiber mesh with 1481 nodes on the GeForce RTX 3080 GPU. As noted before, we use the NVIDIA Quadro GP100 GPU for the next study. This GPU is also not capable of solving the whole set of 1481 models instances per fiber for 169 fibers at the same time. For one instance of the subcellular model, 57 state and rate variables each, and 71 intermediate variables have to be stored, along with other data, such as element lengths for every element.

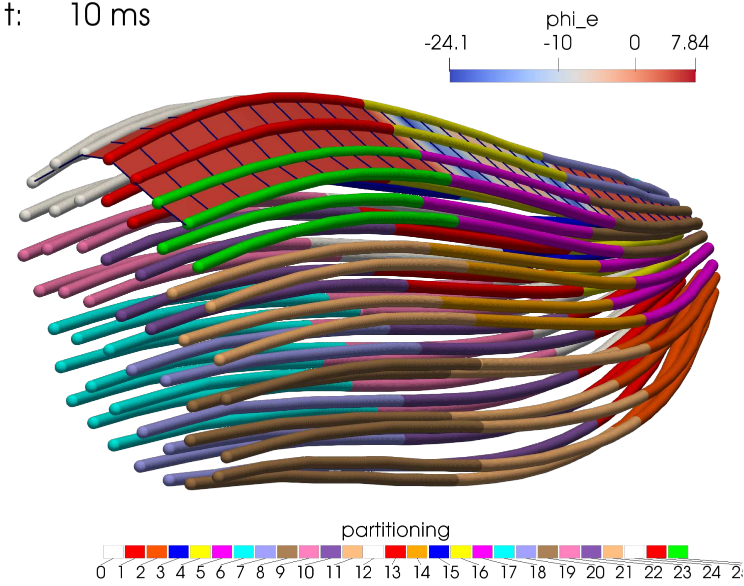


Figure 9.14: Electrophysiology solver in OpenDiHu: Partitioning of the 169 fibers to 27 processes used in the runtime study with hybrid CPU-GPU usage. The image also shows the resulting EMG signals ϕ_e on the muscle surface.

Thus, we follow a hybrid approach. We parallelize the scenario to 27 processes on the CPU. Only one process offloads its subdomain to the GPU. In this way, both the CPU and the GPU take part in the computation and the available hardware capabilities are fully exploited.

The scenario and the numerical parameters are the same as described for the study with the Shorten model in Sec. 9.2.2. 49 muscle fibers are used and parallelized to $3\times3\times3 = 27$ subdomains. Figure 9.14 visualizes the partitioning of the fibers by different colors and the EMG values ϕ_e on the muscle surface at the simulation end time of $t_{\text{end}} = 10\,\text{ms}$.

Figure 9.15 visualizes the runtimes of two runs. The first bar only employs the CPU and provides the reference for the measured runtimes. The second bar corresponds to the hybrid run, where one process employs the GPU. The 0D and 1D solver runtimes in the second bar are averaged over the CPU computations. The total runtime involves both the

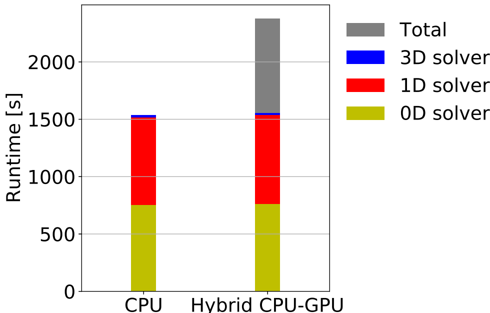


Figure 9.15: Electrophysiology model in OpenDiHu: Runtime study for a CPU-only computation and a hybrid parallelization that employs both the CPU and the GPU. For the hybrid approach, one of the 27 partitions was computed on the GPU. The compute-intense subcellular model of Shorten et al. is simulated.

CPU and the GPU computations. It can be seen that the total runtime given by the total bar heights is higher for the hybrid run. In the hybrid runs, the CPU processes have to wait at the synchronization point in the solution of the 3D problem until the GPU process has completed the 0D/1D computations.

9.4.3 Conclusion

Several scenarios with computations of the 0D subcellular and 1D electric conduction models on the GPU have been evaluated. Section 9.2.2 compared the runtime for the Hodgkin-Huxley subcellular model on 625 fibers on the GPU with implementations on the CPU. In Sec. 9.3.1, the computation on the GPU with the Shorten subcellular model was measured for one fiber. Section 9.4.1 conducted a strong scaling study with the Hodgkin-Huxley model on 169 fibers and Sec. 9.4.2 evaluated a hybrid approach, where the Shorten model on different fibers was computed on the GPU and the CPU at the same time. This last study used a GPU with higher double-precision performance than the previous studies.

In all of these studies, the GPU computations could not compete with their CPU counterparts. The GPU implementation of the models relied on target-specific CUDA code, which was automatically generated by the OpenMP 4.5 pragmas produced by the code generator in OpenDiHu. The CPU computations used highly optimized CPU code with

explicit vector instructions, an approach that is close to optimal as shown by the roofline model in Sec. 9.3.2. Thus, the comparison considers different levels of optimization. We cannot conclude in general that the GPU is less suited to solve the fiber based electrophysiology models than the CPU. However, the GPU support in OpenMP is not competitive with our optimized CPU implementation.

The required GPU support of OpenMP in the GNU compiler is functional to the extent needed in our studies only since GCC version 11, which, at the time of writing, is still experimental and not yet released. Further performance gains can be expected in the future as compiler development advances. One problem is also the high memory requirement for the subcellular models, which only allows a certain number of subcellular model instances to be computed on the given hardware. It is also not clear, whether the high memory consumption is also an artifact of the compiler and will reduce with later compiler versions.

Despite the lower performance, it was shown that OpenDiHu can be used to solve the monodomain equation Eq. (5.11) with different subcellular models on the GPU. Switching between the CPU and GPU variants can be accomplished by only changing the `optimizationType` parameter between `"vc"` and `"gpu"`. Hybrid strategies, where some processes use `"vc"` and others `"gpu"`, have been demonstrated.

In future work, the performance issue of the GPU computations can be addressed by using different technologies to access the compute power of GPUs. Examples are to directly use the CUDA programming language or the C++ based abstraction layer for acceleration hardware SYCL [Khr19]. OpenDiHu already provides a reference implementation for such improvements by the code generator, which outputs model specific code with OpenMP pragmas for GPU offloading. This code implements proper, economical data transfer between the devices and contains hints how to distribute the workload on the GPU by the respective pragma placements. It could be used as a starting point to integrate further “optimization types” in the code generator.

How To Reproduce

The scripts to run the studies and to create the plots for Figures 9.13 and 9.15 are available in the repository at github.com/dihu-stuttgart/performance in the directories `opendihu/16_hodgkin_huxley_gpu` and `opendihu/17_shorten_gpu`.

9.5 Parallel Scaling of the EMG Model Using High Performance Computing

After the parallel scaling of the multi-scale model has been investigated in moderately parallel scenarios with up to 27 processes in the previous sections, we now study the parallel scalability in High Performance Computing scenarios with larger degrees of parallelism. We simulate the fiber based electrophysiology model consisting of a 0D subcellular model, the 1D electric conduction problem and the 3D bidomain equation, as described in Sec. 5.1. We conduct these studies on the supercomputer Hawk at the High Performance Computing Center Stuttgart (HLRS). The system contains a total of 5632 compute nodes. Each compute node consists of two AMD EPYC 7742 processors with 64 cores each, a clock frequency of 2.25 GHz and 256 GB memory per node.

In the following, Sec. 9.5.1 presents a weak scaling study, which scales the problem size up to a realistic number of 270 000 muscle fibers in a biceps muscle. Then, Sec. 9.5.2 shows measurements of the scaling behavior for the 1D model solver and gives details on MPI rank placement policies.

9.5.1 Weak Scaling of the Fiber Based Electrophysiology Model

We simulate the fiber based electrophysiology model with EMG values on the muscle surface and the subcellular model of Hodgkin and Huxley [Hod52a]. Corresponding simulation results of this scenario, also for the highly parallel runs, are presented in Sec. 8.4.4.

In this weak scaling study, the number of fibers and number of processes is varied while their relation is kept approximately constant. The scenarios are constructed such that there are approximately 10 fibers per process, while maintaining a cube partitioning scheme in the 3D domain. The 0D subcellular and the 1D electric conduction problems on the fibers are solved with the `FastMonodomainSolver` class and the `"vc"` optimization type, which is the fastest available option in OpenDiHu. The Strang operator splitting scheme with Heun's method and the implicit Euler scheme are used. The 3D problem is solved using the conjugate gradient solver of PETSc and a relative tolerance on the residual norm of 10^{-5}.

The used timestep widths are $dt_{0D} = 10^{-3}$ ms, $dt_{1D} = dt_{\text{splitting}} = 2 \cdot 10^{-3}$ ms and $dt_{3D} = 1$ ms. Because only the scaling behavior is of interest in this study, the simulation end time is set to $t_{\text{end}} = 2$ ms.

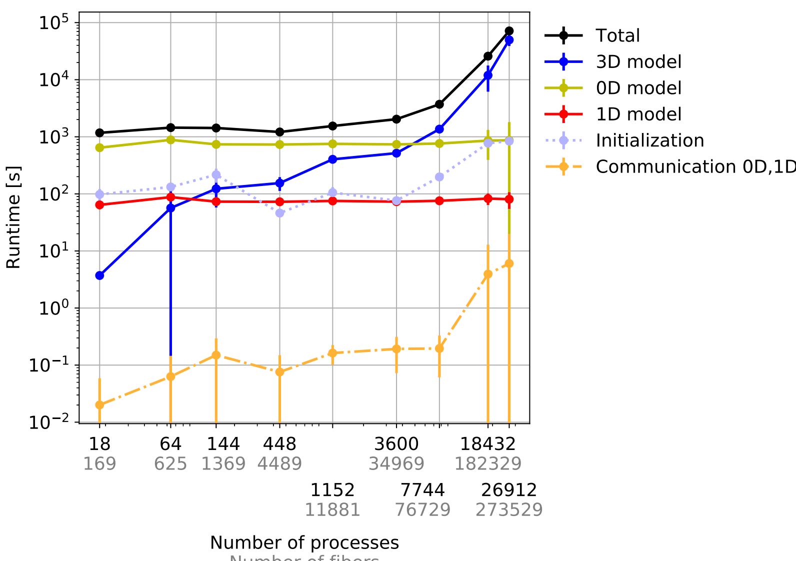


Figure 9.16: Weak scaling of the fiber based electrophysiology model on the supercomputer Hawk simulating up to more than 270 000 muscle fibers.

Figure 9.16 presents the resulting runtimes for the different parts of the simulation program: the solvers of the 0D, 1D and 3D models, the runtime for initialization and the runtime for the communication in the `FastMonodomainSolver`, as explained in Sec. 7.4.2. To relate the initialization runtime to the runtime of the solvers in a realistic scenario with longer simulation times, all runtimes except for the initialization are scaled to a simulation end time of 1 s.

The results show perfect weak scaling properties of the 0D and the 1D solver, given by the yellow and red lines. This is expected due to the construction of the algorithm and the parallel partitioning. The 0D problems are “embarrassingly parallel” and are solved independently of each other. In the 1D problem solver, the values are transferred to a

# processes	18	64	144	448	1152	3600	7744	26 912
# iterations	72	115	176	339	561	1056	1636	2807

Table 9.1: Scaling study of the fiber based muscle model: Number of iterations of the conjugate gradient solver for the 3D bidomain model in the weak scaling study presented in Fig. 9.16.

dedicated process, where the serial Thomas algorithm is employed for each fiber as a whole. Thus, the solution of all 1D problems is also performed independently of each other, but an additional communication step is required, before and after running the solver in each implicit time step of the 1D problem. The plot shows a very small runtime for this communication even for higher parallelism, which is given by the orange dashed line.

The initialization of the computation involves parsing the Python script, which for the last data point requires 35.1 s, parallel file access and read operations of the mesh file, assembly of the 3D stiffness matrix and solution of the potential flow problem to obtain the fiber directions in the 3D mesh, which contains approximately 10^8 dofs for the last data point, code generation, compilation, linking and loading of the shared library for the subcellular problem, and initialization of all internal data structures.

Loading the mesh input file from the file system is the part of the initialization, which requires the most runtime. The dotted light blue line in Fig. 9.16 shows that the initialization time increases to a maximum value of 839 s for the largest problem size.

The runtime of the 3D model is shown by the blue line in Fig. 9.16. This part of the model is responsible for the highest portion of the total runtime starting from the scenario with 7744 fibers and 76 729 processes. This increase is two-fold: first, the communication cost increases for a larger number of processes. Second, the number of iterations in the conjugate gradient solver increases for a larger number of unknowns.

In this weak scaling study, the number of conjugate gradient solver iterations increases from 72 for the first data point to 2807 for the last data point, as listed in Tab. 9.1. The 3D problem of the last data point has 10^8 dofs. The exact numbers of dofs are also listed in Tab. 8.1 in Sec. 8.4.4. Currently, the solution of the 3D problem uses no preconditioner. In future work, a multigrid solver could be employed for preconditioning, which could improve the weak scaling for large problem sizes.

In summary, the solution of the multi-scale model for fiber based electrophysiology without fat layer exhibits a very good weak scalability for up to 35 000 fibers. For larger problem sizes, the solution of the 3D problem dominates and the weak scaling behavior

deteriorates. However, the solution times are still feasible, as such large problems have been successfully solved in Sec. 8.4.4 of this work.

9.5.2 Weak Scaling of the 1D Solver and MPI Rank Placement

Next, we compare the different approaches to solve the 1D electric conduction part of the monodomain equation on the fibers in a High Performance Computing setting. Figure 9.17 shows a similar weak scaling study to the one in the last section with slightly different process counts. The study was carried out on the supercomputer Hazel Hen, a Cray XC30 system, which was installed until 2020 as the predecessor to Hawk at the High Performance Computing Center Stuttgart. The same problem as in the last section is solved, and, again, the relation between fibers and processes is 10:1. The partitionings were chosen in accordance with the compute nodes of Hazel Hen, which had 24 cores. The detailed problem sizes and partitionings can be found in [Mai19].

Figure 9.17 shows the runtimes of the same 0D and 1D solvers as in the last section by the solid yellow line for the 0D problem, the solid red line below for the 1D problem and the solid orange line for the communication. The perfect weak scaling for these parts is in line with the previous observations.

In addition, we compare the weak scaling of the 1D solution, if a parallel conjugate gradient solver of PETSc is used for the problem on every fiber, instead of the serial Thomas algorithm. This setup corresponds to the vc-aovs scenario presented in Sec. 9.2.2. As already noted earlier, the performance of this approach is worse. The dashed and dotted lines in Fig. 9.17 present the runtimes of this approach for two different MPI rank placement strategies, but for the identical program. It can be seen that the runtimes increase with higher numbers of processes in both curves. This effect is the result of the 1D fiber problems being distributed to more processes, as the total number of processes increases.

For example, in the scenarios with 1200 and 3468 processes, all fibers are distributed to 12 different processes. For the last three data points of the dashed curve, all fibers are distributed to 24 processes. As a result, the runtime to solve the 1D problems in the measurements with 1200 and 3468 processes is approximately equal, but lower than the runtime for the last three data points, where twice the amount of processors takes part in the solution of a single 1D problem.

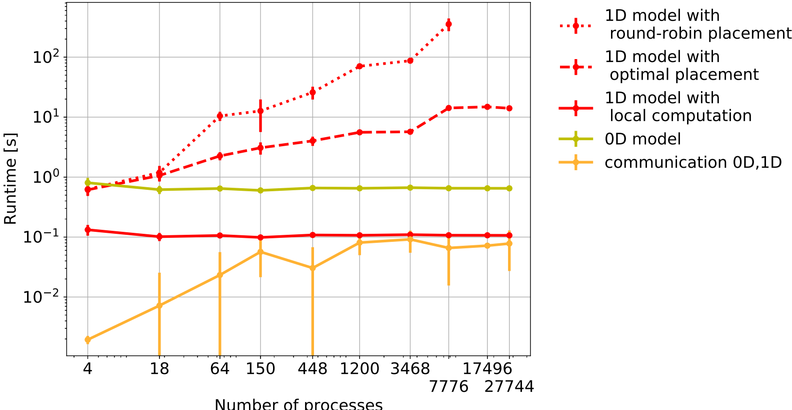


Figure 9.17: Scaling study of the fiber based muscle model: Weak scaling behavior of solvers for the 1D problem. Runtimes for the 1D solver with different rank placement strategy (dotted and dashed red lines) and the optimized runtimes of the `FastMonodomainSolver`, which combines the solution of the 0D and 1D problem (yellow and solid red lines, respectively).

The difference between the dotted and the dashed red curves is a different strategy to place the processes on the compute nodes. The dashed curve with the lower runtime corresponds to a placement of all fiber sharing processes on the same compute node. As the subdomain indices in the $n_x \times n_y \times n_z$ partitioning increase fastest in x-direction, then in y-direction and then in z-direction, and the fibers are aligned with the z direction, the set of processes on a compute node contains MPI ranks that are offset with a constant stride of $n_x n_y$. This has to be ensured in the job scripts on the supercomputers by explicit MPI rank pinning.

If no such measures are taken, the default placement of MPI ranks on the compute nodes proceeds consecutively by filling the compute nodes in the order of the MPI ranks. This corresponds to a round-robin placement of the fibers on the compute nodes, which is the worst possible way of distributing MPI ranks to compute nodes. All processes that compute a fiber are potentially located on different nodes and have to communicate over compute node boundaries. Figure 9.17 shows the resulting runtimes by the upper, dotted red curve. The difference in runtime increases to one order of magnitude for the highest number of cores.

Thus, it is important to properly handle MPI rank placement on compute clusters with

multiple compute nodes. As a consequence, we also configured rank placement on the supercomputer Hawk accordingly in all our studies on this system.

9.6 Performance Studies of the Solid Mechanics Solver

Next, we address the performance of the solid mechanics solver. Its runtime in OpenDiHu is given by the call to the nonlinear solver of PETSc. PETSc, in turn, calls two functions in OpenDiHu, which evaluate the nonlinear function to be solved and the Jacobian of the nonlinear function for a given vector of unknowns. In the following, Sec. 9.6.1 compares an analytic and a numerical computation scheme for the Jacobian, and Sec. 9.6.2 studies the impact of vectorization in this computation.

9.6.1 Analytic and Numerical Computations of the Jacobian

The computation of the Jacobian can be done in two ways. The first possibility is done by PETSc, which uses finite differences to numerically estimate the value of the Jacobian. The second possibility is to evaluate the respective analytic formulation within OpenDiHu. This analytic formulation is derived in Sec. 5.4 and uses the SEMT library [Gut12] to differentiate the mechanics model given by a strain energy function at compile time.

To compare both approaches, we simulate a tension test, where a cuboid domain is extended by an applied force. The box is shown in Fig. 9.18 and has physical dimensions $2\,\text{cm} \times 2\,\text{cm} \times 5\,\text{cm}$. The left face of the box is fixed at $z = 0$ and the two edges of the left face at $x = 0$ and $y = 0$ are also fixed to prevent rotation of the body. On the right face, a constant surface load with a total force of 10 is applied. The material model is the incompressible transversely isotropic Mooney-Rivlin description given in Eq. (5.40), which is also used for the muscle tissue. However, no active stress is considered. The material parameters are chosen as $c_1 = 2, c_2 = 3, b = 4$ and $d = 5$. The fiber direction lies in the $y-z$ plane and has an angle of 40° to the z axis. As a result, the box slightly bends in negative y direction, as can be seen in Fig. 9.18. In this figure, the volume of the deformed object is colored according to the displacement in y direction.

We solve the problem using the Newton solver of PETSc with a secant line search over the L_2 norm of the function. The absolute and relative residual tolerances are set to

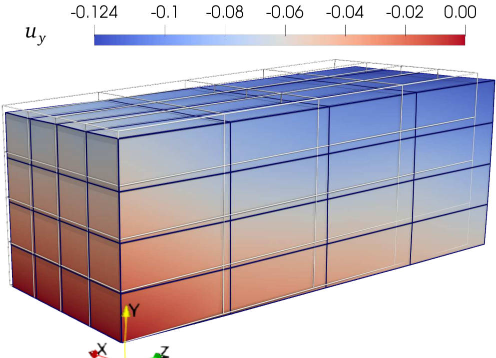


Figure 9.18: Scaling study for solid mechanics: Visualization of the solid mechanics problem used in the weak scaling studies. The box is extended by a surface load at the right end. The wireframe shows the reference configuration, the colored mesh shows the current configuration where the color corresponds to the displacements u_y in vertical direction.

10^{-5}, which leads to approximately 5 Newton iterations. The parallel direct solver of the MUMPS package [Ame17] is used to solve the linear system in every Newton iteration.

We compare the runtimes of numerically and analytically computing the Jacobian in this problem. We vary the number of processes and the problem size in a weak scaling setting, such that there are exactly 8 3D elements per process. A dual-socket AMD EPYC 7742 64-core processor with clock speed of 2.25 GHz, a total core count of 128 and 1.96 TB RAM is used. We vary the number of processes between one and 256 processes. For more than 128 processes, hyperthreading is used.

Figure 9.19 shows the runtimes to solve the nonlinear system by the orange and purple lines in a double logarithmic plot. An increasing runtime for higher process counts is observed for both computation approaches. The curves have a higher slope for more than 128 processes, when the computation uses more processes than physical cores in the processor.

The computation of the Jacobian requires $\mathcal{O}(n)$ non-zero entries to be calculated as the matrix has a banded sparsity structure, where n is the number of degrees of freedom or elements in the 3D mesh. The number of non-zero Jacobian entries is, thus, constant in the weak scaling setup. However, Fig. 9.19 shows a linearly increasing runtime for the numerical computation of the Jacobian, which can be seen by the comparison with the

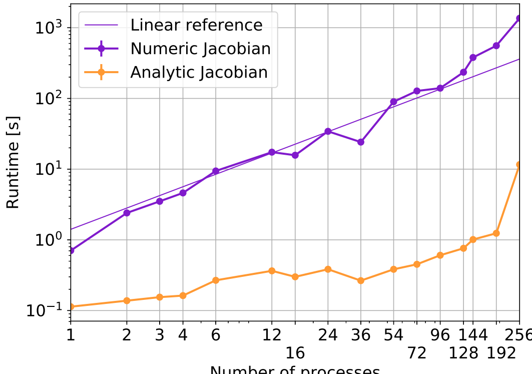


Figure 9.19: Weak scaling of the nonlinear mechanics solver. The runtime to solve the nonlinear system equation is given for the numerical and for the analytic computation approaches for the Jacobian matrix.

plotted linear function. The finite difference scheme of PETSc has no information about the sparsity pattern and estimates all $\mathcal{O}(n^2)$ matrix entries.

The runtimes for the analytic computation are only slightly increasing in the weak scaling, which is as expected. This approach only computes the non-zero entries of the Jacobian. Moreover, the values of the material elasticity tensor $\mathbb{C}$ can be computed once and reused for every entry of the Jacobian.

The comparison between the approaches shows lower computation times for the analytic approach by factors between 6.2 for one process and 450 for 192 processes. In conclusion, the analytic formula for the Jacobian, which is implemented in OpenDiHu, allows us to speed up the mechanics computations and to compute larger problem sizes in feasible runtimes.

9.6.2 Vectorization of the Analytic Jacobian Computations

In the assembly of finite element system matrices of any kind, contributions are computed on an element level and then combined to form a global system matrix. In OpenDiHu, this algorithm can be vectorized by performing analog but independent computations for multiple elements concurrently in multiple SIMD lanes. This vectorization uses the *Vc*

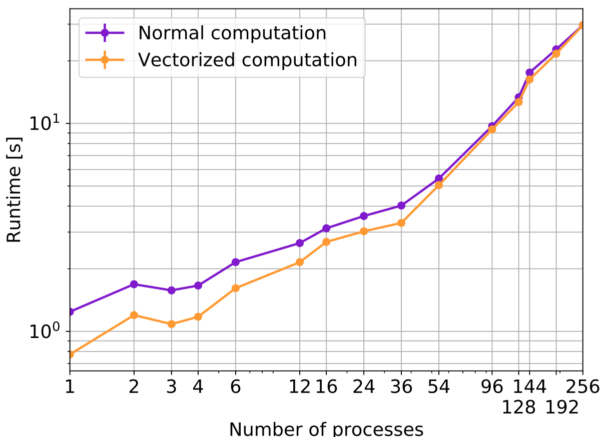


Figure 9.20: Scaling Study for the solid mechanics solver: Weak scaling study to evaluate vectorization in the computation of the Jacobian.

library, similar to the optimizations of the subcellular model solver presented in Sec. 9.2. As this vectorization significantly increases compilation times, the feature is turned off by default and has to be enabled before compilation.

In this section, we evaluate the effect of vectorized system matrix computations for the Jacobian matrix of the nonlinear solid mechanics solver. We conduct a similar weak scaling study as before using the same scenario, but an eight times larger number of elements for each measurement. We compare the runtime to solve the nonlinear problem using analytic Jacobian computations with disabled and enabled vectorization.

Figure 9.20 presents the resulting runtimes for the entire nonlinear solver. The vectorized computation shows speedups from 1.6 for one process to 1.06 for 128 processes. The theoretically possible speedup is four, as the processor supports the AVX2 instruction set with a vector register length of four double-precision values. The measured speedups are lower, because the computation of the Jacobian is only one portion of the computations in the nonlinear solver.

In summary, the vectorized computation of Jacobian matrices can reduce runtimes for the nonlinear solver. A runtime reduction of 38 % (corresponding to the speedup of 1.6) was observed for the serial scenario. The performance gain is largest for small

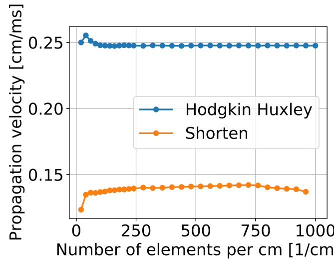
problem sizes and small numbers of processes, and gets less prominent for larger degrees of parallelism.

How To Reproduce

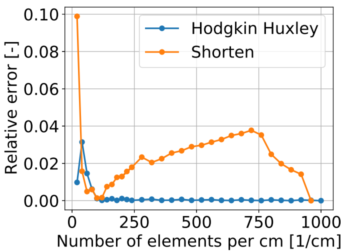
The scripts to run the studies in this scenario and to create the plots of Figures 9.19 and 9.20 are available in the repository at github.com/dihu-stuttgart/performance in the directory `opendihu/22_solid_mechanics_vectorization`.

9.7 Numerical Studies

Next, we perform numerical studies to evaluate mesh resolutions and linear solvers. Section 9.7.1 addresses the mesh width of the 1D problem. Section 9.7.2 evaluates different numerical solver choices for the multidomain model.



(a) Propagation velocities over spatial resolution of the 1D mesh.



(b) Relative error of the propagation velocities over spatial resolution of the 1D mesh.

Figure 9.21: Influence of the mesh width on the propagation velocity of the action potential for the subcellular models of Shorten and Hodgkin-Huxley in the fiber-based electrophysiology simulation. This study is used to determine the 1D mesh width.

9.7.1 Effect of the Mesh Width on the Action Potential Propagation Velocity

The numerical parameters of a simulation scenario such as mesh widths and timestep widths should be chosen, such that the resulting numerical errors are balanced between all model components. In the simulation of activated muscle fibers, the propagation velocity of the action potentials is an important quantity, which also influences the macroscopic outcome of the EMG recordings. Thus, we investigate the effect of numerical parameters on the propagation velocity.

In the first study, we consider a single fiber given by a 1D mesh, where the monodomain equation Eq. (5.11) is used. We measure the error of the propagation velocity depending on the mesh width. A fiber with a physical length of 4 cm is used and discretized by different numbers of 1D elements. We stimulate the fiber at its center at the beginning of the simulation and run the simulation until an end time of 28 ms. An action potential propagates along the fiber. We determine the location of the propagating peak at the end and compute the propagation velocity.

Figure 9.21a shows the resulting values of the propagation velocity for the Hodgkin-Huxley and Shorten subcellular models for varying mesh resolutions. It can be seen that the velocities level out at a constant value for finer mesh discretizations. The absolute

value of the propagation velocity is different for the two subcellular models and depends on various model parameters.

For a quantitative evaluation, we examine the error of the propagation velocity, which is estimated by comparing each run with the value from the finest simulation. One issue with comparing propagation velocities on discretized meshes is that the measured velocity can only be calculated as number of elements traversed per time interval. Thus, the measurements for different mesh resolutions have different accuracies, not only due to the usual discretization error in the finite element approach. This issue can be reduced if a long enough time is simulated, such that the action potential propagates a large enough distance in terms of multiples of the element width.

Figure 9.21b shows the resulting relative error of the propagation velocity. It can be seen that, for the Hodgkin Huxley model, the error gets close to zero for 100 elements per centimeter of fiber length. For the Shorten model, this mesh resolution also exhibits a low relative error. However, the error increases again for higher mesh resolutions before decreasing again starting at approximately 750 elements per centimeter.

As a result, we use a 1D mesh resolution of 100 elements per centimeter for all muscle fibers in all our simulations. For a biceps muscle of 14.8 cm length, this leads to muscle fiber discretizations with 1480 elements and 1481 nodes.

How To Reproduce

The scripts for this study are available in the repository at github.com/dihu-stuttgart/performance in the directory `opendihu/02_propagation_velocity`:

```
./run.sh
./plot_propagation_velocity.py
```

9.7.2 Linear Solvers for the Multidomain Problem

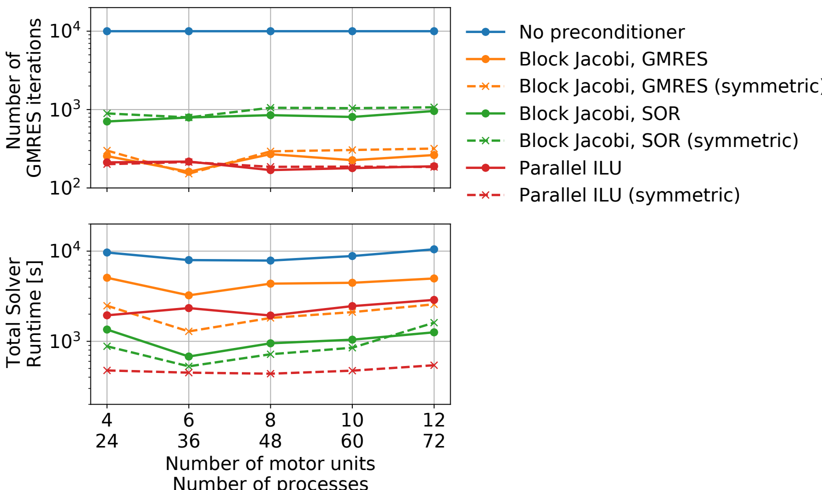


Figure 9.22: Evaluation of preconditioners for the multidomain model. The system is solved with a GMRES solver and different preconditioners. The upper plots shows the remaining number of iterations in the GMRES solver, the lower plot measures the total runtime of preconditioning and solution. Three different preconditioners given by orange, green and red lines are compared to the blue reference curve of using no preconditioner.

In the next study, we investigate the parallel weak scaling behavior for the multidomain model with a particular focus on the total runtimes for different choices of the preconditioner that is used in the solution of the linear system of equations. Our goal is to select the fastest solver-preconditioner combination to speed up the computation of the multidomain model.

In contrast to classical weak scaling, we increase the number of MUs with the number of processes, i.e., the size of the blocks in the resulting block structured matrix remains constant, whereas the number of blocks increases.

We simulate the multidomain model with a fat domain as given in Sec. 5.3.5 and with the subcellular model of Shorten et al. [Sho07]. The model is discretized by a Strang operator splitting with Heun's method for the subcellular model and an implicit Euler scheme for the multidomain equations. The timestep widths of all schemes are $dt_{\text{0D}} =$

$dt_{\text{multidomain}} = dt_{\text{splitting}} = 5 \cdot 10^{-4}$ ms, and an end time of 10^{-1} ms is used, corresponding to 200 invocations of the linear solver.

We partition the 3D computational domain, consisting of the muscle mesh with 50 024 nodes and the fat layer mesh with 37 000 nodes, to 24, 36, 48, 60 and 72 subdomains. In a weak scaling setup, we correspondingly simulate 4,6,8,10 and 12 MUs. As the square-shaped system matrix contains one row of blocks for every MU plus one row of blocks for the fat mesh, the total number of rows does not scale exactly linearly with the number of MUs. In consequence, the system matrices in the five scenarios contain 279 720, 379 768, 479 816, 579 864 and 679 912 rows and columns. Thus, the problem size per process is only approximately constant in this weak scaling study.

We solve the linear system of the multidomain equations by a GMRES solver, because the system matrix is non-symmetric. The stopping threshold on the residual norm is set to 10^{-15} and the specified maximum number of iterations is 10^4. Different preconditioners are applied and the resulting number of GMRES iterations and the total runtime for preconditioner and solver are measured. Figure 9.22 shows the number of GMRES iterations in the upper plot and the total runtimes in the lower plot.

For every preconditioner, the preconditioning is either performed based on the non-symmetric system matrix (solid lines) or based on a symmetric matrix that is obtained by taking all diagonal blocks of the system matrix (dashed lines), as described in Sec. 7.5.3.

The reference measurement is given by the GMRES solver without any preconditioner, visualized by the blue lines in both plots. The upper plot of Fig. 9.22 indicates the maximum number of 10^4 iterations for all measurements of the GMRES solver without preconditioner. This means that the specified tolerance of 10^{-15} is not reached in the given number of iterations. Thus, a preconditioner is required to obtain an accurate solution.

The first examined preconditioner is the block Jacobi scheme. A block Jacobi preconditioner divides the system matrix into blocks on the diagonal, yielding smaller problems that can each be solved individually. The scheme is an iterative solver, which starts with an initial solution $\mathbf{x}^{(0)}$ and successively computes approximations $\mathbf{x}^{(i+1)} = \Phi(\mathbf{x}^{(i)})$ of the solution until the residual norm reaches the specified threshold.

For a model problem $A\mathbf{x} = \mathbf{b}$, where the system matrix $A = D + L + U$ is decomposed into a matrix D with blocks on the diagonal and lower and upper triangular block matrices L

and U, respectively, the preconditioning is based on the following iterative computation scheme $\Phi : \mathbf{x}^{(i)} \mapsto \mathbf{x}^{(i+1)}$:

$$D\,\mathbf{x}^{(i+1)} = \mathbf{b} - (L+U)\mathbf{x}^{(i)}. \tag{9.1}$$

Because of the structure of the block diagonal matrix D, the system is decoupled and every process can solve its own linear system of equations using the respective diagonal block as system matrix. If the symmetric option for the system matrix used in the preconditioner is chosen, the matrices L and U vanish and the solution of Eq. (9.1) is trivial.

The preconditioner is constructed using the reordered matrix layout described in Sec. 7.5.2 and the symmetric matrix is obtained as discussed in Sec. 7.5.3. The remaining blocks on the matrix diagonal belong to the subdomains of the parallel partitioning. Each block corresponds to a part of the mesh for all MU compartments.

Two versions of block Jacobi preconditioners provided by the PETSc library are evaluated. The first variant, shown by the orange lines in Fig. 9.22, employs a GMRES solver for the resulting smaller linear systems of equations in Eq. (9.1). The second variant, shown by the green lines in Fig. 9.22, uses a SOR (successive over-relaxation) solver with over-relaxation parameter $\omega = 1$, i.e., a Gauß-Seidel scheme.

The upper plot in Fig. 9.22 shows that the number of GMRES solver iterations is reduced more for the GMRES solver than for the Gauß-Seidel solver, as the constant number of GMRES iterations in the preconditioner yields a better approximation to the solution than the same number of Gauß-Seidel iterations. The lower plot shows a smaller total runtime for the block Jacobi scheme with Gauß-Seidel solver than for the block Jacobi scheme with GMRES solver. This means that the lower runtime of the Gauß-Seidel solver in the preconditioner outweighs the larger number of GMRES iterations in the solver, compared to the GMRES preconditioning scheme. For both preconditioners, the total runtime for the variant with the symmetric matrix is lower than for the variant with the full system matrix.

Another solver is *Euclid* [Hys01] from the HYPRE package, shown by the red lines in Fig. 9.22. It is a parallel implementation of incomplete LU factorization using graph partitioning and a two-level ordering strategy.

The plots in Fig. 9.22 show a low number of remaining GMRES iterations after the preconditioner has been applied, similar to the block Jacobi scheme with GMRES solver.

However, the total runtime of Euclid is significantly lower than the runtime of the GMRES-block Jacobi scheme. If the symmetric system matrix is used for preconditioning, the total runtime is the lowest of all measured preconditioner combinations.

Regarding the parallel weak scaling, the lower plot in Fig. 9.22 shows overall good scaling properties for all considered preconditioners. The runtimes slightly decrease from the first to the second data points, as the system matrix size per process also slightly decreases. Then, a trend of slightly increasing runtimes in the weak scaling setup can be seen, which indicates that the preconditioners and the GMRES solver perform slightly more computations and communication for larger problem sizes. The accuracy of the preconditioning step is not affected by the overall problem size, as can be seen by the constant numbers of GMRES iterations in the upper plot of Fig. 9.22.

As a result, we use the combination of Euclid preconditioner and GMRES solver in all solutions of the multidomain model in this work, because this is the fastest of the tested combinations. Apart from the three presented preconditioners, more available choices in the software packages PETSc and HYPRE were tested, but yielded worse performance. These include the *Parallel Incomplete Factorization preconditioner* (PILUT) from the HYPRE package and the combinations of the block Jacobi scheme with an algebraic multigrid method or the Euclid preconditioner for the subproblems. Tests with the parallel algebraic multigrid method *BoomerAMG* from the HYPRE package also showed promising results with even lower total runtimes than the Euclid preconditioner, but suffered from occasional long runtimes and divergence in a non-deterministic fashion. However, the full set of possible parameters such as different coarsening and interpolation options and settings for the smoother have not yet been evaluated and, after fine-tuning, corresponding performance improvements could be possible in future work.

How To Reproduce

The script for this study is available in the repository at github.com/dihu-stuttgart/performance in the directory `opendihu/07_multidomain_solver`:

```
./run_mu.sh
```

Chapter 10

Conclusion and Future Work

To conclude this work, we summarize the presented models, algorithms, implementations, studies and the main findings. Moreover, we give an outlook on future work and additional research questions that can be approached building on our work.

10.1 Summary of this Work

The overarching goal of this work was to enable simulations of the neuromuscular system using detailed, biophysical multi-scale models with high resolutions. The simulations should compute numerically accurate results, run efficiently on various hardware and allow parallel scaling to large problem sizes, which should be solved on supercomputers.

As a result, this work established a computational framework for multi-scale modeling of skeletal muscles, their neural activation, muscle contraction and generation of EMG signals on the skin surface. Our approach combined existing models for various parts of the neuromuscular system into a comprehensive multi-scale model framework. Scalability and parallel efficiency of our software were ensured by efficient algorithms, suitable, parallelized numerical schemes and by our accompanying performance analyses.

We described the following topics in this work: After the introduction in Chap. 1, we compared two modeling approaches to describe the movement of the upper arm in Chap. 2. Based on data of experimental trials we conducted during a graduate school workshop, we developed a first, data-driven model using Gaussian process regression and a second model based on a biophysical simulation with two muscle models. The parameters for the biophysical simulation were fitted to experimental training data using numerical optimization. The comparison of the two approaches revealed a slightly better fit for the biophysical simulation model. This approach had the additional benefit of giving biophysical insights into the functioning of the system and provided estimates for

subject-specific muscle parameters. While this study used Hill-type muscle models, which describe muscle forces on a 1D line of action, we considered more accurate multi-scale models in the remainder of this work.

Chapter 3 dealt with the generation of structured 3D meshes and embedded 1D meshes for muscle fibers. The approach of only using structured meshes, which allowed for a simple domain decomposition proved to be beneficial for the parallel performance of our simulations. We described a workflow, how to obtain these meshes from biomedical imaging data. We developed a serial algorithm and a parallel algorithm to construct the required meshes and to ensure a good mesh quality, even for meshes with high resolutions. The algorithms were based on our novel approach of using harmonic maps to transform reference meshes to cross-sectional slices of the muscle mesh.

In Chap. 4, we described ways to associate muscle fibers with motor units (MUs) in a physiological manner. We developed efficient algorithms for this task for different premises, and employed the algorithms to associate up to 270 000 muscle fibers to 100 MUs for the subsequent use in our simulations.

In Chap. 5, we first described all equations of the state-of-the models that we used, and how they can be combined into a multi-scale description. Then, we described their discretization using the finite element method for the spatial derivative terms and various timestepping and operator splitting schemes for the temporal derivatives. One original contribution is the derivation of the finite element formulation for the multidomain equation. Further, we gave a detailed description of the nonlinear solid mechanics discretization, which we used in our implementation.

Next, we presented details on our simulation software OpenDiHu, which we used to solve various combinations of the described multi-scale model framework to simulate the neuromuscular system. Chapter 6 gave an introduction to the design and usage of the software and demonstrated its application using various example problems.

Chapter 7 described the implementation of OpenDiHu in more detail, motivated various design decisions, introduced the data handling and several algorithms, e.g., to construct a parallel domain decomposition or to map data between meshes, and described the implementation of various solvers for particular parts of the multi-scale model.

Chapter 8 presented numerical results, which were obtained using our simulation software. We simulated the passive mechanical behavior of muscle tissue, subcellular models given in CellML description, electrophysiology on muscle fibers, electric conduction in the muscle and the adipose tissue to obtain surface EMG signals, electrophysiology using the

3D homogenized multidomain description, and coupled scenarios of electrophysiology and muscle contraction. We discussed effects of model and structural parameters and interpreted the obtained simulation results.

In Chap. 9, we analyzed the computational performance of our software in general and various solvers in particular. We conducted numerical studies of universal convergence properties with the software OpenCMISS, which also helped to parameterize the numerical solvers in OpenDiHu. Further studies on mesh widths and used linear solvers were carried out directly using OpenDiHu. We evaluated various optimization options in OpenDiHu and compared the most optimized settings in OpenDiHu with the baseline solver OpenCMISS, yielding a high speedup of more than two orders of magnitude. Moreover, we investigated the computational performance of our models on the GPU, and conducted parallel strong scaling and parallel weak scaling tests on small clusters and the supercomputers at the High Performance Computing Center Stuttgart.

10.2 Summary of Main Findings

The present work simulated numerous scenarios with various model combinations, which provided different insights. In the following, we summarize the observed findings. We address the biophysical observations in Sec. 10.2.1 and results of the performance measurements in Sec. 10.3.

10.2.1 Observations from the Fields of Biophysics and Biomechanics

The comparison of the linear and nonlinear mechanics models in Sec. 8.2 showed qualitatively different results and demonstrated that the accurate behavior of deforming muscle tissue can only be described by a proper nonlinear anisotropic solid mechanics model.

Initially, an open question was also how to relate the accuracy of the simulated EMG signals to the number of fibers and the mesh resolution. Our numerical studies in Sec. 9.7.1, which compared the resulting action propagation velocity for different mesh widths of the 1D muscle fiber meshes showed that a mesh width of 100 µm or 100 elements per cm gives reasonably accurate results.

To evaluate the 3D mesh width and the spacing between the muscle fibers, we conducted simulations with different 3D mesh resolutions and numbers of fibers in Sec. 8.4.4. The number of fibers was scaled up to the realistic number of 270 000 fibers in a biceps brachii muscle. We concluded that the most accurate solution is obtained for a mesh width as fine as possible, as the EMG results were qualitatively different for every refinement step. This emphasizes the need for highly resolved simulation scenarios (representing the real number of fibers in a muscle accurately) for realistic EMG computations and, as a result, the need for High Performance Computing techniques.

However, if the EMG is to be sampled by electrodes, i.e., if the EMG recording process should also be part of the simulation, lower mesh widths might be possible, as the EMG is only captured at the locations of the electrodes.

One possible approach to reduce the computational effort for EMG simulations would be to only consider the muscle tissue down to a certain depth below the surface with the EMG electrodes. We observed in Sec. 8.4.2, that the EMG signal is highly influenced by MUs, whose territories are located close to the electrodes. However, our numerical experiments with EMG decomposition algorithms in Sec. 8.4.6 showed that large MUs located opposite to the EMG electrodes at the deepest muscle tissue layers are detectable in the surface EMG signals. Thus, neglecting the deeper parts of the muscle would remove relevant information from the system and is, therefore, not a valid approach to reduce the computational load.

Furthermore, the layer of adipose tissue on top of the muscle showed a smoothing effect on EMG recordings in our simulations, both with the fiber based approach in Sec. 8.4.3 and with the multidomain approach in Sections 8.5.1 and 8.5.2. One advantage of our simulations compared to experimental studies is that the thickness of the fat layer is known exactly and can also be adjusted.

Simulations of muscle contraction with coupled electrophysiology and solid mechanics models showed a spatially inhomogeneous contraction for the biceps muscle while the muscle activation is ramped up. The simulation in Sec. 8.6.1 of an isolated, contracting muscle belly without tendons showed transverse bending, alternating between the left and right-hand sides, as a result of the subsequently activated MUs at the different sides of the muscle. We also simulated the biceps brachii muscle together with its tendons and observed a ripple in the generated muscle force, which is caused by the same inhomogeneous MU activity.

The simulations of muscle contraction also showed that, if the muscle is initially in a stress-free state, the model can only achieve a maximum contraction of approximately

85 %. However, the muscles of the musculoskeletal system are known to exhibit pre-stresses in their relaxed states. Accordingly, we added prestress to our simulations. The amount of prestress is adjustable in the simulation settings, and the required amount can be determined by a comparison with experimental studies.

10.3 Summary of Performance Results

A major part of the work was also concerned with improving the performance of the simulation software, and, thus, enabling larger simulation scenarios in shorter runtimes.

Previously, literature on biophysical, multi-scale models of skeletal muscles was mainly focused on modelling and interpretation of the results, rather than targeting efficient computations. The work of Röhrle et al. [Röh12] introduced the multi-scale model, which we based our work on, and simulated the tibialis anterior muscle using a 3D mechanics mesh with 12 elements. The work of Heidlauf et al. [Hei13] considered the same geometry and simulated 400 muscle fibers. The authors parallelized their OpenCMISS based implementation for a fixed number of four processes. We built upon this work with the goal to push the limits of feasible problem sizes, and, in Sec. 8.4.4, executed our optimized simulation with 26 912 processes, 273 529 muscle fibers and a 3D mesh for the electrophysiology model with approximately 10^8 degrees of freedom.

The performance analyzes in Chap. 9 showed that the subcellular model contributes a large portion to the total runtime and, thus, is the most crucial part to optimize. By using proper memory layouts, vectorization is possible. Our approach of using explicit vector instructions outperformed the auto-vectorization capabilities of the compiler. The approximation of the exponential function and an improved parallelization scheme for the 1D electric conduction problem additionally contributed to a high speedup. The comparison to the baseline solver OpenCMISS Iron in a strong scaling study in Sec. 9.3.1 revealed a maximum speedup of 363 for the purely implementation-specific improvements and an additional speedup of 2.5, shown in Sec. 9.1.1, by using more efficient numerical methods.

In addition, the memory characteristics of the solvers were investigated in Sec. 9.3.1. The linear increase in memory consumption of the baseline solver in a weak scaling setting was improved to a nearly constant scaling. Our analysis using a roofline performance model showed that our solvers are compute bound and achieve a computational performance of approximately 25 % peak performance, which is a very good value.

Moreover, hybrid shared/distributed memory parallelism and computations on the GPU were investigated, but both approaches were found to be not competitive with our highly optimized distributed memory parallelization. For the GPU, potentially more efficient approaches than our approach using OpenMP exist, such that a performance improvement in the future could be possible.

The modularity of the CellML infrastructure, where computational models can be shared among researchers and are interchangeable in multi-scale simulations was preserved during all optimization endeavors. Our approach was to implement a source-to-source code generator, which transformed the given CellML code into optimized code for the CPU or the GPU.

For the solution of the multidomain model, we evaluated various preconditioners and selected the most performant preconditioner-solver combination for our computations. One previously unforeseen result is the large discrepancy of required runtime between the fiber based and the multidomain based electrophysiology models, presented in Sec. 8.5. We measured by a factor of 1000 longer computation times for the multidomain model, which result from the structure of the model. Despite the high computational effort, the multidomain model is useful in practice as it can simulate effects that are not captured by the fiber based model. We gave a detailed comparison between both approaches in Sec. 8.5.3.

In summary, we provided a computationally efficient and scalable tool for applied biophysics researchers to solve problems in the domains of EMG generation and muscle contraction. For example, the effect of different muscle fiber organizations and MU recruitment strategies can be tested with our software. We demonstrated its use with state-of-the-art EMG decomposition algorithms, which provide the bridge to the experimental domain. Thus, we hope to contribute one step on the pathway of complementing in vivo with in silico experiments to increase the understanding of the neuromuscular system.

10.4 Outlook and Future Work

The presented work could be extended in multiple directions, spanning performance improvements and model extensions.

First, some ideas for further performance improvements could be implemented and evaluated. The monodomain equation could be solved with implicit-explicit (IMEX)

schemes, which could potentially achieve higher precision. The numerically stiff subcellular model is currently solved explicitly. Implicit schemes could be developed, and the implicit iteration equations could be solved symbolically in a preprocessing step using the parsed CellML code.

To improve the performance of the multidomain model, the following algorithmic improvements are promising options. The 3D problems of action potential propagation in the muscle volume for every compartment could be restricted to the subset of nodes, where the occupancy factors are above a certain threshold, effectively reducing the problem sizes, and reducing the effect of higher MU counts on the runtime. However, this would bring difficulties to ensure a balanced parallel domain decomposition. Instead of the current parallel partitioning of the domain, the multidomain model could also be parallelized by distributing the MUs to different processes or by a combination of both approaches.

On the numerical side, an extended error analysis could be carried out for all model parts, and the timestep widths, which are currently chosen conservatively, could potentially be increased, while keeping the numerical error below a given threshold. Error estimators could be developed, which would allow an adaptive adjustment of the timestep widths. The 3D model solvers for the 3D electrophysiology and multidomain problems could be enhanced with geometric or algebraic multigrid preconditioners.

Since all subcellular points in a muscle are usually in similar states at any time, a hybrid approach using analytic descriptions of action potential propagation, as in Sec. 8.4.7, and a fully numerical treatment could be chosen, and surrogate models could be adaptively added to the computational description.

On the technical side, computations on the GPU could be re-evaluated in the future using the existing OpenMP approach with more mature compiler versions or different accelerator targeting programming technologies.

Second, the range of simulated models could be extended. The simulations could be applied to further muscle geometries such as the triceps brachii or the tibialis anterior muscles. Muscles with more complex geometries and fiber arrangements could be investigated. A mechanically coupled problem of agonist-antagonist pair could be considered such as a system of biceps and triceps brachii. Apart from the mechanical coupling, a coupling of the neural recruitment involving sensory organs in the muscles could be implemented and used to approach further biomechanical research questions. Such a neuromuscular feedback loop could also be investigated first for a single muscle, e.g., by extending the preliminary implementation in OpenDiHu for the biceps muscle.

Pathological conditions could be simulated to understand muscular diseases and neuromuscular electrical stimulation of the muscle for stroke rehabilitation could be considered.

By using the preCICE adapters in OpenDiHu, more advanced mechanics solvers could be coupled to an electrophysiology simulation in OpenDiHu, allowing to, e.g., study mechanical effects of surrounding tissue.

On a larger scale, the interplay of more organs could be taken into account. Blood perfusion and muscle metabolism could be added, and coupled by models of the lung and general metabolism in the organism. Thus, a digital human model can be envisioned, which allows to study the effects of anomalies and to develop new therapies, effectively utilizing simulation technology for human wellbeing.

Bibliography

[80008] **80000-13:2008**, I.: *Quantities and Units—Part 13: Information Science and Technology*, Standard, Geneva, CH: International Organization for Standardization / International Electrotechnical Commission, 2008

[Ahr05] **Ahrens**, J.; **Geveci**, B.; **Law**, C.: *ParaView: an end-user tool for large data visualization*, Visualization Handbook, ed. by **Hansen**, C.; **Johnson**, C. R., Elsevier, 2005

[Ala20] **Alappat**, C. et al.: *A recursive algebraic coloring technique for hardware-efficient symmetric sparse matrix-vector multiplication*, ACM Trans. Parallel Comput. 7.3, 2020, ISSN: 2329-4949, doi:10.1145/3399732, https://doi.org/10.1145/3399732

[All05] **Alliez**, P. et al.: *Variational tetrahedral meshing*, ACM SIGGRAPH 2005 Papers, SIGGRAPH '05, Los Angeles, California: Association for Computing Machinery, 2005, pp. 617–625, isbn:9781450378253, doi:10.1145/1186822.1073238, https://doi.org/10.1145/1186822.1073238

[Aln15] **Alnæs**, M. et al.: *The fenics project version 1.5*, Archive of Numerical Software 3.100, 2015

[Alt20] **Altair Engineering, Inc**: *High Fidelity Finite Element Modeling | Altair Hypermesh*, 2020, https://www.altair.com/hypermesh/

[Ame01] **Amestoy**, P. R. et al.: *A fully asynchronous multifrontal solver using distributed dynamic scheduling*, SIAM Journal on Matrix Analysis and Applications 23.1, 2001, pp. 15–41, doi:10.1137/S0895479899358194

[Ame17] **Amestoy**, P. et al.: *On the complexity of the block low-rank multifrontal factorization*, SIAM Journal on Scientific Computing 39.4, 2017, pp. 1710–1740, doi:16M1077192

[Ame19] **Amestoy**, P. R. et al.: *Performance and scalability of the block low-rank multifrontal factorization on multicore architectures*, ACM Trans. Math. Softw. 45.1, 2019, ISSN: 0098-3500, doi:10.1145/3242094, https://doi.org/10.1145/3242094

[And05] **Andreasen**, D. S.; **Alien**, S. K.; **Backus**, D. A.: *Exoskeleton with emg based active assistance for rehabilitation*, 9th International Conference on Rehabilitation Robotics, 2005. ICORR 2005. 2005, pp. 333–336, doi:10.1109/ICORR.2005.1501113

[Bal15] **Balay**, S. et al.: *PETSc users manual*, tech. rep. ANL-95/11 - Revision 3.6, Argonne National Laboratory, 2015

[Bal16] **Balay**, S. et al.: *PETSc Web page*, http://www.mcs.anl.gov/petsc, 2016, http://www.mcs.anl.gov/petsc

[Bal97] **Balay**, S. et al.: *Efficient management of parallelism in object oriented numerical software libraries*, Modern Software Tools in Scientific Computing, ed. by **Arge**, E.; **Bruaset**, A. M.; **Langtangen**, H. P., Birkhäuser Press, 1997, pp. 163–202

[Bar03] **Bar-Cohen**, Y.; **Breazeal**, C.: *Biologically inspired intelligent robots*, Smart Structures and Materials 2003: Electroactive Polymer Actuators and Devices (EAPAD), ed. by **Bar-Cohen**, Y., vol. 5051, International Society for Optics and Photonics, SPIE, 2003, pp. 14–20, doi:10.1117/12.484379, https://doi.org/10.1117/12.484379

[Bar04] **Bar-Cohen**, Y.; **SPIE**, eds.: *Electroactive polymer (eap) actuators as artificial muscles: reality, potential, and challenges*, Englisch, Mode of access: World Wide Web, Bellingham, Wash. <1000 20th St. Bellingham WA 98225-6705 USA>, 2004, http://dx.doi.org/10.1117/3.547465

[Bar96] **Barber**, C. B.; **Dobkin**, D. P.; **Huhdanpaa**, H.: *The quickhull algorithm for convex hulls*, ACM Trans. Math. Softw. 22.4, 1996, pp. 469–483, ISSN: 0098-3500, doi:10.1145/235815.235821, https://doi.org/10.1145/235815.235821

[Bay17] **Bayer**, A. et al.: *The influence of biophysical muscle properties on simulating fast human arm movements*, Computer Methods in Biomechanics and Biomedical Engineering 20.8, 2017, PMID: 28387534, pp. 803–821, doi:10.1080/10255842.2017.1293663

[Bes12] **Bessmeltsev**, M. et al.: *Design-driven quadrangulation of closed 3d curves*, ACM Trans. Graph. 31.6, 2012, ISSN: 0730-0301, doi:10.1145/2366145.2366197, https://doi.org/10.1145/2366145.2366197

[Bla93] **Blacker**, T. D.; **Meyers**, R. J.: *Seams and wedges in plastering: a 3-d hexahedral mesh generation algorithm*, Engineering with computers 9.2, 1993, pp. 83–93

[Ble05a] **Blemker**, S. S.; **Pinsky**, P. M.; **Delp**, S. L.: *A 3D model of muscle reveals the causes of nonuniform strains in the biceps brachii*, Journal of Biomechanics 38.4, 2005, pp. 657–665, doi:10.1016/j.jbiomech.2004.04.009

[Ble05b] **Blemker**, S. S.; **Delp**, S. L.: *Three-dimensional representation of complex muscle architectures and geometries*, Annals of biomedical engineering 33.5, 2005, pp. 661–673

[Böl08] **Böl**, M.; **Reese**, S.: *Micromechanical modelling of skeletal muscles based on the finite element method*, Computer methods in biomechanics and biomedical engineering 11.5, 2008, pp. 489–504

[Böl12] **Böl**, M. et al.: *Compressive properties of passive skeletal muscle—the impact of precise sample geometry on parameter identification in inverse finite element analysis*, Journal of Biomechanics 45.15, 2012, pp. 2673–2679, ISSN: 0021-9290, doi:https://doi.org/10.1016/j.jbiomech.2012.08.023, https://www.sciencedirect.com/science/article/pii/S002192901200471X

[Bra11] **Bradley**, C. et al.: *Opencmiss: a multi-physics & multi-scale computational infrastructure for the vph/physiome project*, Progress in Biophysics and Molecular Biology 107.1, 2011, Experimental and Computational Model Interactions in Bio-Research: State of the Art, pp. 32–47, ISSN: 0079-6107, doi:https://doi.org/10.1016/j.pbiomolbio.2011.06.015, https://www.sciencedirect.com/science/article/pii/S0079610711000629

[Bra18] **Bradley**, C. P.; **Emamy**, N.; **Ertl**, T.; **Göddeke**, D.; **Hessenthaler**, A.; **Klotz**, T.; **Krämer**, A.; **Krone**, M.; **Maier**, B.; **Mehl**, M.; **Rau**, T.; **Röhrle**, O.: *Enabling detailed, biophysics-based skeletal muscle models on HPC systems*, Frontiers in Physiology 9.816, 2018, doi:10.3389/fphys.2018.00816

[Bra69] **Brandstater**, M.; **Lambert**, E.: *A histochemical study of the spatial arrangement of muscle fibers in single motor units within rat tibialis anterior muscle*, Bull Am Assoc Electromyogr Electrodiag 82, 1969, pp. 15–16

[Bun16] **Bungartz**, H.-J. et al.: *Precice – a fully parallel library for multi-physics surface coupling*, Computers & Fluids 141, 2016, Advances in Fluid-Structure Interaction, pp. 250–258, ISSN: 0045-7930, doi:https://doi.org/10.1016/j.compfluid.2016.04.003, https://www.sciencedirect.com/science/article/pii/S0045793016300974

[Byr95] **Byrd**, R. H. et al.: *A limited memory algorithm for bound constrained optimization*, SIAM Journal on scientific computing 16.5, 1995, pp. 1190–1208

[Car14] **Cardiff**, P. et al.: *Nonlinear solid mechanics in openfoam*, 9th OpenFOAM Workshop, University of Zagreb, Croatia, 2014

[Car17] **Carniel**, T. A.; **Fancello**, E. A.: *A transversely isotropic coupled hyperelastic model for the mechanical behavior of tendons*, Journal of Biomechanics 54, 2017, pp. 49–57, ISSN: 0021-9290, doi:https://doi.org/10.1016/j.jbiomech.2017.01.042, https://www.sciencedirect.com/science/article/pii/S0021929017300726

[Cav05] **Cavallaro**, E. et al.: *Hill-based model as a myoprocessor for a neural controlled powered exoskeleton arm - parameters optimization*, Proceedings of the 2005 IEEE International Conference on Robotics and Automation, 2005, pp. 4514–4519

[Cav06] **Cavallaro**, E. E. et al.: *Real-time myoprocessors for a neural controlled powered exoskeleton arm*, IEEE Transactions on Biomedical Engineering 53.11, 2006, pp. 2387–2396

[Cel21] **CellMLTeam**: *The CellML project*, 2021, https://www.cellml.org/

[Che97] **Chew**, L. P.: *Guaranteed-quality delaunay meshing in 3d (short version)*, Proceedings of the thirteenth annual symposium on Computational geometry, 1997, pp. 391–393

[Chi04] **Childers**, M. K.: *Targeting the neuromuscular junction in skeletal muscles*, American Journal of Physical Medicine & Rehabilitation 83.10, 2004, ISSN: 0894-9115, https://journals.lww.com/ajpmr/Fulltext/2004/10001/Targeting_the_Neuromuscular_Junction_in_Skeletal.6.aspx

[Cho13] **Choi**, H. F.; **Blemker**, S. S.: *Skeletal muscle fascicle arrangements can be reconstructed using a laplacian vector field simulation*, PLOS ONE 8.10, 2013, pp. 1–7, doi:10.1371/journal.pone.0077576

[Cis08] **Cisi**, R. R.; **Kohn**, A. F.: *Simulation system of spinal cord motor nuclei and associated nerves and muscles, in a web-based architecture*, Journal of computational neuroscience 25.3, 2008, pp. 520–542

[Cla21] **Clarke**, A. K. et al.: *Deep learning for robust decomposition of high-density surface emg signals*, IEEE Transactions on Biomedical Engineering 68.2, 2021, pp. 526–534, doi:10.1109/TBME.2020.3006508

[Coh96] **Cohen**, S. D.; **Hindmarsh**, A. C.; **Dubois**, P. F.: *Cvode, a stiff/nonstiff ode solver in c*, Computers in physics 10.2, 1996, pp. 138–143

[Coo06] **Cooper**, J.; **McKeever**, S.; **Garny**, A.: *On the application of partial evaluation to the optimisation of cardiac electrophysiological simulations*, Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '06, Charleston, South Carolina: Association for Computing Machinery, 2006, pp. 12–20, isbn:1595931961, doi:10.1145/1111542.1111546, https://doi.org/10.1145/1111542.1111546

[Coo15] **Cooper**, J.; **Spiteri**, R. J.; **Mirams**, G. R.: *Cellular cardiac electrophysiology modeling with Chaste and CellML*, Frontiers in Physiology 5, 2015, p. 511, doi:10.3389/fphys.2014.00511

[Cor92] **Cordes**, C.; **Kinzelbach**, W.: *Continuous groundwater velocity fields and path lines in linear, bilinear, and trilinear finite elements*, Water resources research 28.11, 1992, pp. 2903–2911

[Cra47] **Crank**, J.; **Nicolson**, P.: *A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type*, Mathematical Proceedings of the Cambridge Philosophical Society 43.1, 1947, pp. 50–67, doi:10.1017/S0305004100023197

[Cue03] **Cuellar**, A. A. et al.: *An overview of CellML 1.1, a biological model description language*, SIMULATION 79.12, 2003, pp. 740–747, doi:10.1177/0037549703040939

[Das20] **Dassault Systèmes**: *Abaqus Unified FEA - SIMULIA™ by Dassault Systèmes®*, 2020, https://www.3ds.com/products-services/simulia/products/abaqus/

[De 06] **De Luca**, C. J. et al.: *Decomposition of surface emg signals*, Journal of Neurophysiology 96.3, 2006, PMID: 16899649, pp. 1646–1657, doi:10.1152/jn.00009.2006, eprint: https://doi.org/10.1152/jn.00009.2006, https://doi.org/10.1152/jn.00009.2006

[Del07] **Delp**, S. L. et al.: *Opensim: open-source software to create and analyze dynamic simulations of movement*, IEEE Transactions on Biomedical Engineering 54.11, 2007, pp. 1940–1950

[Del34] **Delaunay**, B. et al.: *Sur la sphere vide*, Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk 7.793-800, 1934, pp. 1–2

[Did10] **Dideriksen**, J. L. et al.: *An integrative model of motor unit activity during sustained submaximal contractions*, Journal of Applied Physiology 108.6, 2010, PMID: 20360437, pp. 1550–1562, doi:10.1152/japplphysiol.01017.2009

[Dim98] **Dimitrov**, G. V.; **Dimitrova**, N. A.: *Precise and fast calculation of the motor unit potentials detected by a point and rectangular plate electrode*, Medical engineering & physics 20.5, 1998, pp. 374–381

[Don05] **Dong**, S. et al.: *Quadrangulating a mesh using laplacian eigenvectors*, tech. rep., 2005

[Eds68] **Edström**, L.; **Kugelberg**, E.: *Histochemical composition, distribution of fibres and fatiguability of single motor units. anterior tibial muscle of the rat*, eng, Journal of neurology, neurosurgery, and psychiatry 31.5, 1968, PMC496396[pmcid], pp. 424–433, ISSN: 0022-3050, doi:10.1136/jnnp.31.5.424, https://doi.org/10.1136/jnnp.31.5.424

[Eme02] **Emery**, A. E.: *The muscular dystrophies*, The Lancet 359.9307, 2002, pp. 687–695, ISSN: 0140-6736, doi:https://doi.org/10.1016/S0140-6736(02)07815-7

[Eme91] **Emery**, A. E.: *Population frequencies of inherited neuromuscular diseases—a world survey*, Neuromuscular Disorders 1.1, 1991, pp. 19–29, ISSN: 0960-8966

[Eno01] **Enoka**, R. M.; **Fuglevand**, A. J.: *Motor unit physiology: some unresolved issues*, Muscle & Nerve 24.1, 2001, pp. 4–17, doi:10.1002/1097-4598(200101)24:1<4::AID-MUS13>3.0.CO;2-F

[Eno08] **Enoka**, R. M.; **Duchateau**, J.: *Muscle fatigue: what, why and how it influences muscle function*, The Journal of Physiology 586.1, 2008, pp. 11–23, doi:https://doi.org/10.1113/jphysiol.2007.139477

[Epp99] **Eppstein**, D.: *Linear complexity hexahedral mesh generation*, Computational Geometry 12.1-2, 1999, pp. 3–16

[Fal02] **Falgout**, R. D.; **Yang**, U. M.: *Hypre: a library of high performance preconditioners*, International Conference on Computational Science, Springer, 2002, pp. 632–641

[Fal16] **Falisse**, A. et al.: *Emg-driven optimal estimation of subject-specific hill model muscle-tendon parameters of the knee joint actuators*, IEEE Transactions on Biomedical Engineering PP, 2016, pp. 1–1, doi:10.1109/TBME.2016.2630009

[Far01] **Farina**, D.; **Merletti**, R.: *A novel approach for precise simulation of the emg signal detected by surface electrodes*, IEEE Transactions on Biomedical Engineering 48.6, 2001, pp. 637–646

[Far10] **Farina**, D. et al.: *Decoding the neural drive to muscles from the surface electromyogram*, Clinical Neurophysiology 121.10, 2010, pp. 1616–1623, ISSN: 1388-2457, doi:https://doi.org/10.1016/j.clinph.2009.10.040

[Fei55] **Feinstein**, B. et al.: *Morphologic studies of motor units in normal human muscles*, Cells Tissues Organs 23.2, 1955, pp. 127–142

[Fer18] **Fernandez**, J. et al.: *Musculoskeletal modelling and the physiome project*, Multiscale Mechanobiology of Bone Remodeling and Adaptation, ed. by **Pivonka**, P., Cham: Springer International Publishing, 2018, pp. 123–174, isbn:978-3-319-58845-2, doi:10.1007/978-3-319-58845-2_3, https://doi.org/10.1007/978-3-319-58845-2_3

[Fie88] **Field**, D. A.: *Laplacian smoothing and delaunay triangulations*, Communications in applied numerical methods 4.6, 1988, pp. 709–712

[Gab04] **Gabriel**, E. et al.: *Open MPI: goals, concept, and design of a next generation MPI implementation*, Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, 2004, pp. 97–104

[Gar03] **Garner**, B. A.; **Pandy**, M. G.: *Estimation of musculotendon properties in the human upper limb*, Annals of Biomedical Engineering 31.2, 2003, pp. 207–220, ISSN: 1573-9686, doi:10.1114/1.1540105, https://doi.org/10.1114/1.1540105

[Gar08] **Garny**, A. et al.: *CellML and associated tools and techniques*, Philos Trans A Math Phys Eng Sci 366.1878, 2008, pp. 3017–3043

[Gar15] **Garny**, A.; **Hunter**, P. J.: *OpenCOR: a modular and interoperable approach to computational biology*, Frontiers in Physiology 6, 2015, p. 26, doi:10.3389/fphys.2015.00026

[Gha20] **Ghadam Soltani**, E.: *Modelling Thermoregulation of the Human Body*, PhD thesis, The University of Auckland, 2020

[God20] **Godoy**, W. F. et al.: *Adios 2: the adaptable input output system. a framework for high-performance data management*, SoftwareX 12, 2020, p. 100561, ISSN: 2352-7110, doi:https://doi.org/10.1016/j.softx.2020.100561, https://www.sciencedirect.com/science/article/pii/S2352711019302560

[Goo20] **Google**: *Googletest user's guide*, https://google.github.io/googletest/, 2020

[Gre11] **Gregson**, J.; **Sheffer**, A.; **Zhang**, E.: *All-hex mesh generation via volumetric polycube deformation*, Computer Graphics Forum 30.5, 2011, pp. 1407–1416, doi:https://doi.org/10.1111/j.1467-8659.2011.02015.x

[Gro09] **Grosland**, N. M. et al.: *IA-FEMesh: an open-source, interactive, multiblock approach to anatomic finite element model development*, Computer methods and programs in biomedicine 94.1, 2009, pp. 96–107

[Gui03] Godunov-type Schemes, ed. by **Guinot**, V., Amsterdam: Elsevier, 2003, pp. 471–480, isbn:978-0-444-51155-3, doi:https://doi.org/10.1016/B978-044451155-3/50015-3, https://www.sciencedirect.com/science/article/pii/B9780444511553500153

[Gun07] **Gunther**, M.; **Schmitt**, S.; **Wank**, V.: *High-frequency oscillations as a consequence of neglected serial damping in hill-type muscle models*, Biological Cybernetics 97.1, 2007, pp. 63–79, ISSN: 1432-0770, doi:10.1007/s00422-007-0160-6, https://doi.org/10.1007/s00422-007-0160-6

[Gut04] **Gutterman**, Z.: *Symbolic pre-computation for numerical applications*, Technion-Israel Institute of Technology, Faculty of Computer Science, 2004

[Gut12] **Gutterman**, Z.: *Semt - compile-time symbolic differentiation via c++ templates*, https://github.com/st-gille/semt, 2012

[Hae14] **Haeufle**, D. et al.: *Hill-type muscle model with serial damping and eccentric force–velocity relation*, Journal of Biomechanics 47.6, 2014, pp. 1531–1536, ISSN: 0021-9290, doi:https://doi.org/10.1016/j.jbiomech.2014.02.009

[Hæg07] **Hægland**, H. et al.: *Improved streamlines and time-of-flight for streamline simulation on irregular grids*, Advances in Water Resources 30.4, 2007, pp. 1027–1045, ISSN: 0309-1708, doi:https://doi.org/10.1016/j.advwatres.2006.09.002, http://www.sciencedirect.com/science/article/pii/S0309170806001709

[Han17] **Handsfield**, G. G. et al.: *Determining skeletal muscle architecture with laplacian simulations: a comparison with diffusion tensor imaging*, Biomechanics and Modeling in Mechanobiology 16.6, 2017, pp. 1845–1855, ISSN: 1617-7940, doi:10.1007/s10237-017-0923-5

[Har20] **Harris**, C. R. et al.: *Array programming with NumPy*, Nature 585.7825, 2020, pp. 357–362, doi:10.1038/s41586-020-2649-2, https://doi.org/10.1038/s41586-020-2649-2

[Hat77] **Hatze**, H.: *A myocybernetic control model of skeletal muscle*, Biological cybernetics 25.2, 1977, pp. 103–119

[Hei03] **Heine**, R.; **Manal**, K.; **Buchanan**, T. S.: *Using hill-type muscle models and emg data in a forward dynamic analysis of joint moment*, Journal of Mechanics in Medicine and Biology 03.02, 2003, pp. 169–186, doi:10.1142/S0219519403000727

[Hei13] **Heidlauf**, T.; **Röhrle**, O.: *Modeling the Chemoelectromechanical Behavior of Skeletal Muscle Using the Parallel Open-Source Software Library OpenCMISS*, Computational and Mathematical Methods in Medicine 2013, 2013, pp. 1–14, doi:10.1155/2013/517287, http://dx.doi.org/10.1155/2013/517287

[Hei14] **Heidlauf**, T.; **Röhrle**, O.: *A multiscale chemo-electro-mechanical skeletal muscle model to analyze muscle contraction and force generation for different muscle fiber arrangements*, Frontiers in Physiology 5.498, 2014, pp. 1–14, doi:10.3389/fphys.2014.00498, http://dx.doi.org/10.3389/fphys.2014.00498

[Hei15] **Heidlauf**, T., ed.: *Chemo-electro-mechanical modelling of the neuromuscular system*, Englisch, Text (nur für elektronische Ressourcen), Online publiziert 2016, Stuttgart, 2015, http://nbn-resolving.de/urn:nbn:de:bsz:93-opus-104496

[Hei16] **Heidlauf**, T. et al.: *A multi-scale continuum model of skeletal muscle mechanics predicting force enhancement based on actin–titin interaction*, Biomechanics and Modeling in Mechanobiology 15.6, 2016, pp. 1423–1437, doi:10.1007/s10237-016-0772-7, http://dx.doi.org/10.1007/s10237-016-0772-7

[Her13] **Hernández-Gascón**, B. et al.: *A 3D electro-mechanical continuum model for simulating skeletal muscle contraction*, Journal of Theoretical Biology 335, 2013, pp. 108–118

[Hil38] **Hill**, A. V.: *The heat of shortening and the dynamic constants of muscle*, Proceedings of the Royal Society of London. Series B - Biological Sciences 126.843, 1938, pp. 136–195, doi:10.1098/rspb.1938.0050

[Hin76] **Hinton**, E.; **Rock**, T.; **Zienkiewicz**, O. C.: *A note on mass lumping and related processes in the finite element method*, Earthquake Engineering & Structural Dynamics 4.3, 1976, pp. 245–249, doi:10.1002/eqe.4290040305

[Hob19] **Hoberock**, J.: *Working Draft, C++ Extensions for Parallelism Version 2*, tech. rep. N4808, International Organization for Standardization (ISO), International Electrotechnical Commission (IEC), 2019, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/n4808.pdf

[Hod13] **Hodkinson**, L.: *Scons-config*, https://github.com/furious-luke/scons-config, 2013

[Hod52a] **Hodgkin**, A. L.; **Huxley**, A. F.: *A quantitative description of membrane current and its application to conduction and excitation in nerve.* The Journal of Physiology 117.4, 1952, pp. 500–544

[Hod52b] **Hodgkin**, A. L.; **Huxley**, A. F.: *Propagation of electrical signals along giant nerve fibres*, Proceedings of the Royal Society of London. Series B, Biological Sciences, 1952, pp. 177–183

[Hol00] **Holzapfel**, A. G.: *Nonlinear solid mechanics*, 2000

[Hol07a] **Holobar**, A.; **Zazula**, D.: *Multichannel blind source separation using convolution kernel compensation*, IEEE Transactions on Signal Processing 55.9, 2007, pp. 4487–4496, doi:10.1109/TSP.2007.896108

[Hol07b] **Holobar**, A.; **Zazula**, D.: *Gradient convolution kernel compensation applied to surface electromyograms*, Independent Component Analysis and Signal Separation, ed. by **Davies**, M. E. et al., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 617–624, isbn:978-3-540-74494-8

[Hol08] **Holobar**, A.; **Zazula**, D.; **Merletti**, R.: *Demusetool-a tool for decomposition of multichannel surface electromyograms*, 2008

[Hun04] **Hunter**, P.: *The iups physiome project: a framework for computational physiology*, Progress in Biophysics and Molecular Biology 85.2, 2004, Modelling Cellular and Tissue Function, pp. 551–569, ISSN: 0079-6107, doi:https://doi.org/10.1016/j.pbiomolbio.2004.02.006, https://www.sciencedirect.com/science/article/pii/S0079610704000318

[Hun07] **Hunter**, J. D.: *Matplotlib: a 2d graphics environment*, Computing in Science & Engineering 9.3, 2007, pp. 90–95, doi:10.1109/MCSE.2007.55

[Hys01] **Hysom**, D.; **Pothen**, A.: *A scalable parallel algorithm for incomplete factor preconditioning*, SIAM Journal on Scientific Computing 22.6, 2001, pp. 2194–2215, doi:10.1137/S1064827500376193, eprint: https://doi.org/10.1137/S1064827500376193, https://doi.org/10.1137/S1064827500376193

[Ino15] **Inouye**, J.; **Handsfield**, G.; **Blemker**, S.: *Fiber tractography for finite-element modeling of transversely isotropic biological tissues of arbitrary shape using computational fluid dynamics*, 2015

[ISO18] **ISO-9241-11:2018(en)**: *Ergonomics of human-system interaction – Part 11: Usability: Definitions and concepts*, Standard, Geneva, CH: International Organization for Standardization, 2018

[Jam15] **Jamin**, C. et al.: *Cgalmesh: a generic framework for delaunay mesh generation*, ACM Trans. Math. Softw. 41.4, 2015, ISSN: 0098-3500, doi:10.1145/2699463, https://doi.org/10.1145/2699463

[Jas07] **Jasak**, H.; **Jemcov**, A.; **Tukovic**, Z., et al.: *Openfoam: a c++ library for complex physics simulations*, International workshop on coupled methods in numerical dynamics, vol. 1000, IUC Dubrovnik Croatia, 2007, pp. 1–20

[Joh00] **Johansson**, T.; **Meier**, P.; **Blickhan**, R.: *A finite-element model for the mechanical analysis of skeletal muscles*, Journal of Theoretical Biology 206.1, 2000, pp. 131–49

[Jua06] **Juanes**, R.; **Matringe**, S. F.: *Unified formulation of velocity fields for streamline tracing on two-dimensional unstructured grids*, Comput. Methods. Appl. Mech. Engrg., submitted, 2006

[Kaw08] **Kawamura**, Y.; **Islam**, M. S.; **Sumi**, Y.: *A strategy of automatic hexahedral mesh generation by using an improved whisker-weaving method with a surface mesh modification procedure*, Engineering with Computers 24.3, 2008, pp. 215–229

[Khr19] **KhronosGroup**: *SYCL Specification, SYCL integrates OpenCL devices with modern C++*, "Version 1.2.1", 2019

[Kis20] **Kislan**, T.: *Base64 encoding and decoding for c++ projects*, https://github.com/tkislan/base64, 2020

[Klo20] **Klotz**, T. et al.: *Modelling the electrical activity of skeletal muscle tissue using a multi-domain approach*, Biomechanics and Modeling in Mechanobiology 19.1, 2020, pp. 335–349, ISSN: 1617-7940, doi:10.1007/s10237-019-01214-5

[Kol21] **Kolvekar**, S.: *Gated Recurrent Unit Network for Decomposition of Synthetic High-Density Surface Electromyography Signals*, MA thesis, Pfaffenwaldring 47, 70569 Stuttgart, Germany: Institute for Signal Processing and System Theory, University of Stuttgart, 2021

[Kov11] **Kovacs**, D.; **Myles**, A.; **Zorin**, D.: *Anisotropic quadrangulation*, Computer Aided Geometric Design 28.8, 2011, Solid and Physical Modeling 2010, pp. 449–462, ISSN: 0167-8396, doi:https://doi.org/10.1016/j.cagd.2011.06.003

[Krä21] **Krämer**, A.; **Maier**, B.; **Rau**, T.; **Huber**, F.; **Klotz**, T.; **Ertl**, T.; **Göddeke**, D.; **Mehl**, M.; **Reina**, G.; **Röhrle**, O.: *Multi-physics multi-scale HPC simulations of skeletal muscles (accepted)*, High Performance Computing in Science and Engineering '20, Transactions of the High Performance Computing Center, Stuttgart (HLRS) 2020, ed. by **Nagel**, W.; **Kröner**, D.; **Resch**, M., Springer International Publishing, 2021

[Kre02] **Krebs**, H. I. et al.: *Robot-aided neurorehabilitation: from evidence-based to science-based rehabilitation*, Topics in Stroke Rehabilitation 8.4, 2002, PMID: 14523730, pp. 54–70, doi:10.1310/6177-QDJJ-56DU-0NW0

[Kre12] **Kretz**, M.; **Lindenstruth**, V.: *Vc: A C++ library for explicit vectorization*, Software: Practice and Experience 42.11, 2012, pp. 1409–1430, doi:https://doi.org/10.1002/spe.1149

[Kre15] **Kretz**, M.: *Extending C++ for explicit data-parallel programming via SIMD vector types*, PhD thesis, Frankfurt am Main: Johann Wolfgang Goethe-Universität Frankfurt

am Main, 2015, p. 256, https://publikationen.ub.uni-frankfurt.de/frontdoor/index/index/docId/38415

[Kus06] **Kuss**, M.: *Gaussian process models for robust regression, classification, and reinforcement learning*, PhD thesis, echnische Universität Darmstadt Darmstadt, Germany, 2006

[Kus19] **Kusterer**, J.: *Extraktion anatomischer Strukturen und Darstellung durch NURBS*, Deutsch, Bachelorarbeit: Universität Stuttgart, Institut für Parallele und Verteilte Systeme, Simulation großer Systeme, Bachelorarbeit, 2019, http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=BCLR-2019-19&engl=0

[Lad16] **Ladd**, D.: *An open-source vascular modelling framework: from imaging to multiscale CFD*, PhD thesis, The University of Auckland, 2016

[Led08] **Ledoux**, F.; **Weill**, J.-C.: *An extension of the reliable whisker weaving algorithm*, Proceedings of the 16th International Meshing Roundtable, Springer, 2008, pp. 215–232

[Leo15] **Leonardis**, D. et al.: *An emg-controlled robotic hand exoskeleton for bilateral rehabilitation*, IEEE Transactions on Haptics 8.2, 2015, pp. 140–151, doi:10.1109/TOH.2015.2417570

[Lin02] **Lin Wang**; **Buchanan**, T. S.: *Prediction of joint moments using a neural network model of muscle activations from emg signals*, IEEE Transactions on Neural Systems and Rehabilitation Engineering 10.1, 2002, pp. 30–37

[Llo03] **Lloyd**, D. G.; **Besier**, T. F.: *An emg-driven musculoskeletal model to estimate muscle forces and knee joint moments in vivo*, Journal of Biomechanics 36.6, 2003, pp. 765–776, ISSN: 0021-9290, doi:https://doi.org/10.1016/S0021-9290(03)00010-1

[Llo04] **Lloyd**, C. M.; **Halstead**, M. D.; **Nielsen**, P. F.: *Cellml: its future, present and past*, Progress in biophysics and molecular biology 85.2, 2004, pp. 433–450

[Low02] **Lowery**, M. M. et al.: *A multiple-layer finite-element model of the surface emg signal*, IEEE Transactions on Biomedical Engineering 49.5, 2002, pp. 446–454

[Maa12] **Maas**, S. A. et al.: *Febio: finite elements for biomechanics*, Journal of Biomechanical Engineering 134.1, 2012, 011005, ISSN: 0148-0731, doi:10.1115/1.4005694, https://doi.org/10.1115/1.4005694

[Maa17] **Maas**, S. A.; **Ateshian**, G. A.; **Weiss**, J. A.: *Febio: history and advances*, Annual review of biomedical engineering 19, 2017, pp. 279–299

[Mac06] **MacIntosh** B., R.; **Gardiner** P., F.; **McComas** A., J.: *Skeletal Muscle: Form and Function*, Second, Human Kinetics, 2006

[Mac84] **MacDougall**, J. D. et al.: *Muscle fiber number in biceps brachii in bodybuilders and control subjects*, Journal of Applied Physiology 57.5, 1984, PMID: 6520032, pp. 1399–1403, doi:10.1152/jappl.1984.57.5.1399

[Mai19] **Maier**, B.; **Emamy**, N.; **Krämer**, A. S.; **Mehl**, M.: *Highly parallel multi-physics simulation of muscular activation and EMG*, COUPLED PROBLEMS 2019, 2019, pp. 610–621, isbn:978-84-949194-5-9, http://hdl.handle.net/2117/190149

[Mai21a] **Maier**, B.: *Input data for OpenDiHu simulations*, version 1.3, Zenodo, 2021, doi:10.5281/zenodo.4705945, https://doi.org/10.5281/zenodo.4705945

[Mai21b] **Maier**, B.: *OpenDiHu*, version 1.3, Zenodo, 2021, doi:10.5281/zenodo.4706049, https://doi.org/10.5281/zenodo.4706049

[Mai21c] **Maier**, B.: *OpenDiHu Online Documentation*, https://opendihu.readthedocs.io/, 2021

[Mai21d] **Maier**, B.; **Göddeke**, D.; **Huber**, F.; **Klotz**, T.; **Röhrle**, O.; **Schulte**, M.: *OpenDiHu - Efficient and Scalable Software for Biophysical Simulations of the Neuromuscular System (in preparation)*, Journal of Computational Physics, 2021

[Mai21e] **Maier**, B.; **Mehl**, M.: *Mesh generation and multi-scale simulation of a contracting muscle-tendon complex (under review)*, Journal of Computational Science, 2021

[Mai21f] **Maier**, B.; **Stach**, M.; **Mehl**, M.: *Real-time, dynamic simulation of deformable linear objects with friction on a 2d surface*, Mechatronics and Machine Vision in Practice 4, 2021, doi:10.1007/978-3-030-43703-9

[Mai22] **Maier**, B.; **Schneider**, D.; **Schulte**, M.; **Uekermann**, B.: *Bridging scales with volume coupling – scalable simulations of muscle contraction and electromyography (under review)*, High Performance Computing in Science and Engineering '21, 2022

[Mak91] **Maker**, B. N.: *Nike3d: a nonlinear, implicit, three-dimensional finite element code for solid and structural mechanics*, 1991

[Mar10] **Marchandise**, E. et al.: *Quality meshing based on stl triangulations for biomedical simulations*, International Journal for Numerical Methods in Biomedical Engineering 26.7, 2010, pp. 876–889

[Mar11] **Marchandise**, E. et al.: *High-quality surface remeshing using harmonic maps—part ii: surfaces with high genus and of large aspect ratio*, International Journal for Numerical Methods in Engineering 86.11, 2011, pp. 1303–1321, doi:10.1002/nme.3099

[Mar94] **Marsden**, J. E.; **Hughes**, T. J.: *Mathematical foundations of elasticity*, Courier Corporation, 1994

[Men16] **Meng**, M.; **He**, Y.: *Consistent quadrangulation for shape collections via feature line co-extraction*, Computer-Aided Design 70, 2016, SPM 2015, pp. 78–88, ISSN: 0010-4485, doi:https://doi.org/10.1016/j.cad.2015.07.010, http://www.sciencedirect.com/science/article/pii/S0010448515001104

[Mer04] **Merletti**, R.; **Parker**, P.: *Electromyography - Physiology, Engineering, and Noninvasive Applications*, ed. by **Akay**, M., John Wiley & Sons, 2004, doi:10.1002/0471678384, http://dx.doi.org/10.1002/0471678384

[Mes06] **Mesin**, L.; **Farina**, D.: *An analytical model for surface emg generation in volume conductors with smooth conductivity variations*, IEEE transactions on biomedical engineering 53.5, 2006, pp. 773–779

[Mes13] **Mesin**, L.: *Volume conductor models in surface electromyography: computational techniques*, Computers in Biology and Medicine 43.7, 2013, pp. 942–952, ISSN: 0010-4825, doi:https://doi.org/10.1016/j.compbiomed.2013.02.002

[Meu17] **Meurer**, A. et al.: *Sympy: symbolic computing in python*, PeerJ Computer Science 3, 2017, e103, ISSN: 2376-5992, doi:10.7717/peerj-cs.103, https://doi.org/10.7717/peerj-cs.103

[Mil06a] **Mileusnic**, M. P.; **Loeb**, G. E.: *Mathematical models of proprioceptors. ii. structure and function of the golgi tendon organ*, Journal of Neurophysiology 96.4, 2006, PMID: 16672300, pp. 1789–1802, doi:10.1152/jn.00869.2005, eprint: https://doi.org/10.1152/jn.00869.2005, https://doi.org/10.1152/jn.00869.2005

[Mil06b] **Mileusnic**, M. P. et al.: *Mathematical models of proprioceptors. i. control and transduction in the muscle spindle*, Journal of Neurophysiology 96.4, 2006, PMID: 16672301, pp. 1772–1788, doi:10.1152/jn.00868.2005, https://doi.org/10.1152/jn.00868.2005

[Mil10] **Miller**, A. K. et al.: *An overview of the CellML API and its implementation*, BMC Bioinformatics 11, 2010, p. 178

[Mil73] **Milner-Brown**, H. S.; **Stein**, R. B.; **Yemm**, R.: *The orderly recruitment of human motor units during voluntary isometric contractions*, eng, The Journal of physiology 230.2, 1973, PMC1350367[pmcid], pp. 359–370, ISSN: 0022-3751, doi:10.1113/jphysiol.1973.sp010192, https://doi.org/10.1113/jphysiol.1973.sp010192

[Mir13] **Mirams**, G. R. et al.: *Chaste: an open source c++ library for computational physiology and biology*, PLOS Computational Biology 9.3, 2013, pp. 1–8, doi:10.1371/journal.pcbi.1002970, https://doi.org/10.1371/journal.pcbi.1002970

[Mir18] **Mirvakili**, S. M.; **Hunter**, I. W.: *Artificial muscles: mechanisms, applications, and challenges*, Advanced Materials 30.6, 2018, p. 1704407, doi:https://doi.org/10.1002/adma.201704407, eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/adma.201704407, https://onlinelibrary.wiley.com/doi/abs/10.1002/adma.201704407

[Möl97] **Möller**, T.; **Trumbore**, B.: *Fast, minimum storage ray-triangle intersection*, Journal of Graphics Tools 2.1, 1997, pp. 21–28, doi:10.1080/10867651.1997.10487468

[Mör12] **Mörl**, F. et al.: *Electro-mechanical delay in hill-type muscle models*, Journal of Mechanics in Medicine and Biology 12.05, 2012, p. 1250085, doi:10.1142/S0219519412500856

[Mor15] **Mordhorst**, M.; **Heidlauf**, T.; **Röhrle**, O.: *Predicting electromyographic signals under realistic conditions using a multiscale chemo-electro-mechanical finite element model*, Interface Focus 5.2, 2015, pp. 1–11, doi:10.1098/rsfs.2014.0076, http://dx.doi.org/10.1098/rsfs.2014.0076

[Mor17] **Mordhorst**, M. et al.: *POD-DEIM reduction of computational EMG models*, Journal of Computational Science 19, 2017, pp. 86–96, doi:10.1016/j.jocs.2017.01.009, http://dx.doi.org/10.1016/j.jocs.2017.01.009

[Mul05] **Mulas**, M.; **Folgheraiter**, M.; **Gini**, G.: *An emg-controlled exoskeleton for hand rehabilitation*, 9th International Conference on Rehabilitation Robotics, 2005. ICORR 2005. 2005, pp. 371–374, doi:10.1109/ICORR.2005.1501122

[Nat20] **National Institute for Research in Computer Science and Automation (Inria), France**: *Nuages Wegpage*, 2020, https://www-sop.inria.fr/prisme/logiciel/nuages.html.en

[Naw10] **Nawab**, S. H.; **Chang**, S.-S.; **De Luca**, C. J.: *High-yield decomposition of surface emg signals*, Clinical Neurophysiology 121.10, 2010, pp. 1602–1615, ISSN: 1388-2457, doi:https://doi.org/10.1016/j.clinph.2009.11.092, https://www.sciencedirect.com/science/article/pii/S138824571000338X

[Neg11] **Negro**, F.; **Farina**, D.: *Decorrelation of cortical inputs and motoneuron output*, Journal of neurophysiology 106.5, 2011, pp. 2688–2697

[Nic14] **Nickerson**, D. P. et al.: *Using CellML with OpenCMISS to simulate multi-scale physiology*, Frontiers in Bioengineering and Biotechnology 2, 2014

[Owe98] **Owen**, S. J.: *A survey of unstructured mesh generation technology.* IMR 239, 1998, p. 267

[Per07] **Perry**, J. C.; **Rosen**, J.; **Burns**, S.: *Upper-limb powered exoskeleton design*, IEEE/ASME Transactions on Mechatronics 12.4, 2007, pp. 408–417

[Pes79] **Peskoff**, A.: *Electric potential in three-dimensional electrically syncytial tissues*, Bulletin of mathematical biology 41.2, 1979, pp. 163–181

[Pie12] **Piegl**, L.; **Tiller**, W.: *The NURBS book*, Springer Science & Business Media, 2012

[Pit09] **Pitt-Francis**, J. et al.: *Chaste: a test-driven approach to software development for biological modelling*, Computer Physics Communications 180.12, 2009, 40 YEARS OF CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architectures, pp. 2452–2471, ISSN: 0010-4655, doi:https://doi.org/10.1016/j.cpc.2009.07.019, https://www.sciencedirect.com/science/article/pii/S0010465509002604

[Pol88] **Pollock**, D. W.: *Semianalytical computation of path lines for finite-difference models*, Groundwater 26.6, 1988, pp. 743–750, doi:10.1111/j.1745-6584.1988.tb00425.x

[Pon09] **Pontonnier**, C.; **Dumont**, G.: *Inverse dynamics method using optimization techniques for the estimation of muscles forces involved in the elbow motion*, International Journal on Interactive Design and Manufacturing (IJIDeM) 3.4, 2009, p. 227, ISSN: 1955-2505, doi:10.1007/s12008-009-0078-4, https://doi.org/10.1007/s12008-009-0078-4

[Pri95] **Price**, M.; **Armstrong**, C. G.; **Sabin**, M.: *Hexahedral mesh generation by medial surface subdivision: part i. solids with convex edges*, International Journal for Numerical Methods in Engineering 38.19, 1995, pp. 3335–3359

[Pri97] **Price**, M. A.; **Armstrong**, C. G.: *Hexahedral mesh generation by medial surface subdivision: part ii. solids with flat and concave edges*, International Journal for Numerical Methods in Engineering 40.1, 1997, pp. 111–136

[Ram18] **Ramasamy**, E. et al.: *An efficient modelling-simulation-analysis workflow to investigate stump-socket interaction using patient-specific, three-dimensional, continuum-mechanical, finite element residual limb models*, Frontiers in Bioengineering and Biotechnology 6, 2018, p. 126, ISSN: 2296-4185, doi:10.3389/fbioe.2018.00126, https://www.frontiersin.org/article/10.3389/fbioe.2018.00126

[Ras05] **Rasmussen**, C. E.; **Williams**, C. K. I.: *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*, The MIT Press, 2005, isbn:026218253X

[Rem10] **Remacle**, J.-F. et al.: *High-quality surface remeshing using harmonic maps*, International Journal for Numerical Methods in Engineering 83.4, 2010, pp. 403–425, doi:10.1002/nme.2824

[Röh07] **Röhrle**, O.; **Pullan**, A. J.: *Three-dimensional finite element modelling of muscle forces during mastication*, Journal of Biomechanics 40.15, 2007, pp. 3363–3372

[Röh08] **Röhrle**, O.; **Davidson**, J. B.; **Pullan**, A. J.: *Bridging scales: a three-dimensional electromechanical finite element model of skeletal muscle*, SIAM Journal on Scientific Computing 30.6, 2008, pp. 2882–2904, doi:10.1137/070691504, http://dx.doi.org/10.1137/070691504

[Röh12] **Röhrle**, O.; **Davidson**, J. B.; **Pullan**, A. J.: *A physiologically based, multi-scale model of skeletal muscle structure and function*, Frontiers in Physiology 3, 2012

[Röh17] **Röhrle**, O.: *DiHu - Towards a digital human, Project Website*, 2017, https://ipvs.informatik.uni-stuttgart.de/SGS/digital_human/index.php

[Röh19] **Röhrle**, O. et al.: *Multiscale modeling of the neuromuscular system: coupling neurophysiology and skeletal muscle mechanics*, WIREs Systems Biology and Medicine 11.6, 2019, e1457, doi:https://doi.org/10.1002/wsbm.1457, eprint: https://onlinelibrary.

wiley.com/doi/pdf/10.1002/wsbm.1457, https://onlinelibrary.wiley.com/doi/abs/10.1002/wsbm.1457

[Ros01] **Rosen**, J. et al.: *A myosignal-based powered exoskeleton system*, IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans 31.3, 2001, pp. 210–222

[Ros69] **Rosenfalck**, P.: *Intra-and extracellular potential fields of active nerve and muscle fibres: A physico-mathematical analysis of different models*, Acta Physiologica Scandinavica. Supplementum 321, 1969, pp. 1–168, ISSN: 0302-2994

[Ros99] **Rosen**, J.; **Fuchs**, M. B.; **Arcan**, M.: *Performances of hill-type and neural network muscle models—toward a myosignal-based exoskeleton*, Computers and Biomedical Research 32.5, 1999, pp. 415–439, ISSN: 0010-4809, doi:https://doi.org/10.1006/cbmr.1999.1524, http://www.sciencedirect.com/science/article/pii/S0010480999915240

[Rup95] **Ruppert**, J.: *A delaunay refinement algorithm for quality 2-dimensional mesh generation*, Journal of Algorithms 18.3, 1995, pp. 548–585, ISSN: 0196-6774, doi:https://doi.org/10.1006/jagm.1995.1021

[Sai18] **Saini**, H. et al.: *Predicting skeletal muscle force from motor-unit activity using a 3d fe model*, PAMM 18.1, 2018, e201800035, doi:10.1002/pamm.201800035, https://onlinelibrary.wiley.com/doi/abs/10.1002/pamm.201800035

[Sar12] **Sartori**, M. et al.: *Emg-driven forward-dynamic estimation of muscle force and joint moment about multiple degrees of freedom in the human lower extremity*, PLOS ONE 7.12, 2012, pp. 1–11, doi:10.1371/journal.pone.0052618, https://doi.org/10.1371/journal.pone.0052618

[Sch06] **Schroeder**, W. J.; **Martin**, K.; **Lorensen**, B.: *The Visualization Toolkit (4th ed.)* Kitware, 2006, isbn:978-1-930934-19-1

[Sch82] **Schock**, R.; **Brunski**, J.; **Cochran**, G.: *In vivo experiments on pressure sore biomechanics: stresses and strains in indented tissues*, Advances in Bioengineering; Winter Annual Meeting, 1982, pp. 88–91

[Sch96] **Schneiders**, R.: *A grid-based algorithm for the generation of hexahedral element meshes*, Engineering with computers 12.3-4, 1996, pp. 168–177

[Sch97] **Schneiders**, R.: *An algorithm for the generation of hexahedral element meshes based on an octree technique*, 6th International Meshing Roundtable, 1997, pp. 195–196

[Sed11] **Sedgewick**, R.; **Wayne**, K.: *Algorithms*, Addison-wesley professional, 2011

[Ser21] **Services**, A. W.: *Easylogging++ - Single header C++ logging library*, https://github.com/amrayn/easyloggingpp, 2021

[She02] **Shewchuk**, J. R.: *Delaunay refinement algorithms for triangular mesh generation*, Computational Geometry 22.1, 2002, 16th ACM Symposium on Computational Geometry, pp. 21–74, ISSN: 0925-7721, doi:https://doi.org/10.1016/S0925-7721(01)00047-5, http://www.sciencedirect.com/science/article/pii/S0925772101000475

[She96] **Shewchuk**, J. R.: *Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator*, Applied Computational Geometry: Towards Geometric Engineering, ed. by **Lin**, M. C.; **Manocha**, D., vol. 1148, Lecture Notes in Computer Science, From the First ACM Workshop on Applied Computational Geometry, Springer-Verlag, 1996, pp. 203–222

[Sho07] **Shorten**, P. R. et al.: *A mathematical model of fatigue in skeletal muscle force contraction*, Journal of Muscle Research and Cell Motility 28.6, 2007, pp. 293–313, doi:10.1007\%2Fs10974-007-9125-6, http://dx.doi.org/10.1007%5C%2Fs10974-007-9125-6

[Sie08] **Siebert**, T. et al.: *Nonlinearities make a difference: comparison of two common hill-type models with real muscle*, Biological Cybernetics 98.2, 2008, pp. 133–143, ISSN: 1432-0770, doi:10.1007/s00422-007-0197-6, https://doi.org/10.1007/s00422-007-0197-6

[Smi04] **Smith**, N. et al.: *Multiscale computational modelling of the heart*, Acta Numerica 13, 2004, p. 371

[Son05] **Song**, R.; **Tong**, K. Y.: *Using recurrent artificial neural network model to estimate voluntary elbow torque in dynamic situations*, Medical and Biological Engineering and Computing 43.4, 2005, pp. 473–480, ISSN: 1741-0444, doi:10.1007/BF02344728, https://doi.org/10.1007/BF02344728

[Spi96] **Spitzer**, V. et al.: *The Visible Human Male: A Technical Report*, Journal of the American Medical Informatics Association 3.2, 1996, pp. 118–130, ISSN: 1067-5027, doi:10.1136/jamia.1996.96236280, https://doi.org/10.1136/jamia.1996.96236280

[Sta06] **Staten**, M. L. et al.: *Unconstrained paving and plastering: progress update*, proceedings of the 15th International Meshing Roundtable, Springer, 2006, pp. 469–486

[Sta10] **Staten**, M. L. et al.: *Unconstrained plastering hexahedral mesh generation via advancing front geometry decomposition*, International journal for numerical methods in engineering 81.2, 2010, pp. 135–171

[Str68] **Strang**, G.: *On the construction and comparison of difference schemes*, SIAM Journal on Numerical Analysis 5.3, 1968, pp. 506–517, doi:10.1137/0705041, eprint: https://doi.org/10.1137/0705041, https://doi.org/10.1137/0705041

[Sus87] **Sussman**, T.; **Bathe**, K.-J.: *A finite element formulation for nonlinear incompressible elastic and inelastic analysis*, Computers & Structures 26.1, 1987, pp. 357–409, ISSN: 0045-7949, https://www.sciencedirect.com/science/article/pii/0045794987902653

[Tak13] **Takaza**, M. et al.: *The anisotropic mechanical behaviour of passive skeletal muscle tissue subjected to large tensile strain*, Journal of the Mechanical Behavior of Biomedical Materials 17, 2013, pp. 209–220, ISSN: 1751-6161, doi:https://doi.org/10.1016/j.jmbbm.2012.09.001, https://www.sciencedirect.com/science/article/pii/S1751616112002457

[Tau96] **Tautges**, T.; **Blacker**, T.; **Mitchell**, S.: *The whisker weaving algorithm a connectivity based method for constructing all hexahedral finite element meshes*, International Journal for Numerical Methods in Engineering 39.19, 1996, pp. 3327–3349

[Tho90] **Thomas**, C. K. et al.: *Twitch properties of human thenar motor units measured in response to intraneural motor-axon stimulation*, Journal of Neurophysiology 64.4, 1990, PMID: 2258751, pp. 1339–1346, doi:10.1152/jn.1990.64.4.1339

[Til08] **Till**, O. et al.: *Characterization of isovelocity extension of activated muscle: a hill-type model for eccentric contractions and a method for parameter determination*, Journal of Theoretical Biology 255.2, 2008, pp. 176–187, ISSN: 0022-5193, doi:https://doi.org/10.1016/j.jtbi.2008.08.009

[Tun78] **Tung**, L.: *A bi-domain model for describing ischemic myocardial dc potentials.* PhD thesis, Massachusetts Institute of Technology, 1978

[Unt13] **Untaroiu**, C. D.; **Yue**, N.; **Shin**, J.: *A finite element model of the lower limb for simulating automotive impacts*, Annals of biomedical engineering 41.3, 2013, pp. 513–526

[Val18] **Valentin**, J. et al.: *Gradient-based optimization with b-splines on sparse grids for solving forward-dynamics simulations of three-dimensional, continuum-mechanical musculoskeletal system models*, Int J Numer Method Biomed Eng, 2018, e2965, doi:10.1002/cnm.2965

[Van06] **Van Loocke**, M.; **Lyons**, C.; **Simms**, C.: *A validated model of passive muscle in compression*, Journal of Biomechanics 39.16, 2006, pp. 2999–3009, ISSN: 0021-9290, doi:https://doi.org/10.1016/j.jbiomech.2005.10.016

[Van08] **Van Loocke**, M.; **Lyons**, C.; **Simms**, C.: *Viscoelastic properties of passive skeletal muscle in compression: stress-relaxation behaviour and constitutive modelling*, Journal of Biomechanics 41.7, 2008, pp. 1555–1566, ISSN: 0021-9290, doi:https://doi.org/10.1016/j.jbiomech.2008.02.007

[Van09] **Van Rossum**, G.; **Drake**, F. L.: *Python 3 Reference Manual*, Scotts Valley, CA: CreateSpace, 2009, isbn:1441412697

[Van14] **Van Campen**, A. et al.: *A new method for estimating subject-specific muscle–tendon parameters of the knee joint actuators: a simulation study*, International Journal for Numerical Methods in Biomedical Engineering 30.10, 2014, pp. 969–987, doi:10.1002/cnm.2639, https://onlinelibrary.wiley.com/doi/abs/10.1002/cnm.2639

[Vei21] **Veillard**, D.: *The XML C parser and toolkit of Gnome*, http://xmlsoft.org/, 2021

[Ven05] **Venture**, G.; **Yamane**, K.; **Nakamura**, Y.: *Identifying musculo-tendon parameters of human body based on the musculo-skeletal dynamics computation and hill-stroeve muscle model*, 5th IEEE-RAS International Conference on Humanoid Robots, 2005. 2005, pp. 351–356

[Vic12] **Vichot**, F. et al.: *Cardiac interventional guidance using multimodal data processing and visualisation: medinria as an interoperability platform*, 2012

[Vir20] **Virtanen**, P. et al.: *SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python*, Nature Methods 17, 2020, pp. 261–272, doi:10.1038/s41592-019-0686-2

[Wag05] **Wagner**, H. et al.: *Isofit: a model-based method to measure muscle–tendon properties simultaneously*, Biomechanics and Modeling in Mechanobiology 4.1, 2005, pp. 10–19, ISSN: 1617-7940, doi:10.1007/s10237-005-0068-9

[Wal20] **Walter**, J. R.; **Saini**, H.; **Maier**, B.; **Mostashiri**, N.; **Aguayo**, J. L.; **Zarshenas**, H.; **Hinze**, C.; **Shuva**, S.; **Köhler**, J.; **Sahrmann**, A. S.; **Chang**, C.-m.; **Csiszar**, A.; **Galliani**, S.; **Cheng**, L. K.; **Röhrle**, O.: *Comparative study of a biomechanical model-based and black-box approach for subject-specific movement prediction**, 2020 42nd Annual International Conference of the IEEE Engineering in Medicine Biology Society (EMBC), 2020, pp. 4775–4778, doi:10.1109/EMBC44109.2020.9176600

[Wey16] **Weyl**, H.: *Über die gleichverteilung von zahlen mod. eins*, Mathematische Annalen 77.3, 1916, pp. 313–352, ISSN: 1432-1807, doi:10.1007/BF01475864, https://doi.org/10.1007/BF01475864

[Wey40] **Weyl**, H.: *The method of orthogonal projection in potential theory*, Duke Math. J. 7.1, 1940, pp. 411–444, doi:10.1215/S0012-7094-40-00725-6

[XBr20] **XBraid**, T.: *XBraid: parallel multigrid in time*, http://llnl.gov/casc/xbraid, 2020

[XYZ20] **XYZ Scientific Applications, Inc**: *TrueGrid Homepage*, 2020, http://truegrid.com/

[Yan20] **Yang**, C.: *Empirical Roofline Tool (ERT)*, (online), 2020, https://crd.lbl.gov/departments/computer-science/par/research/roofline/software/ert/

[Zaj89] **Zajac**, F. E.: *Muscle and tendon properties models scaling and application to biomechanics and motor*, Critical reviews in biomedical engineering 17.4, 1989, pp. 359–411

[Zha03] **Zhang**, Y.; **Bajaj**, C.; **Sohn**, B.-S.: *Adaptive and quality 3d meshing from imaging data*, Proceedings of the Eighth ACM Symposium on Solid Modeling and Applications, SM '03, Seattle, Washington, USA: Association for Computing Machinery, 2003, pp. 286–291, isbn:1581137060, doi:10.1145/781606.781653, https://doi.org/10.1145/781606.781653

[Zha05] **Zhang**, Y.; **Bajaj**, C.; **Sohn**, B.-S.: *3d finite element meshing from imaging data*, Computer Methods in Applied Mechanics and Engineering 194.48, 2005, Unstructured Mesh Generation, pp. 5083–5106, ISSN: 0045-7825, doi:https://doi.org/10.1016/j.cma.2004.11.026, http://www.sciencedirect.com/science/article/pii/S0045782505000800

[Zha14] **Zhang**, J. et al.: *The map client: user-friendly musculoskeletal modelling workflows*, Biomedical Simulation, ed. by **Bello**, F.; **Cotin**, S., Cham: Springer International Publishing, 2014, pp. 182–192, isbn:978-3-319-12057-7

[Zha18] **Zhang**, K. et al.: *System framework of robotics in upper limb rehabilitation on poststroke motor recovery*, Behavioural Neurology 2018, 2018, p. 6737056, ISSN: 0953-4180, doi:10.1155/2018/6737056, https://doi.org/10.1155/2018/6737056

[Zie05] **Zienkiewicz**, O. C.; **Taylor**, R. L.; **Zhu**, J. Z.: *The finite element method: its basis and fundamentals*, Elsevier, 2005

[Zie77] **Zienkiewicz**, O. C.; **Taylor**, R. L.: *The finite element method*, vol. 3, McGraw-hill London, 1977