Contents

1 Usage and Implementation of the Software OpenDiHu

The simulation of complex multi-scale models requires the combination of tailored numerical solution schemes: Spatial mesh resolutions and timestep widths have to be chosen carefully to avoid instabilities. For a given error tolerance, timestep widths should be chosen as large as possible to allow a sufficiently long simulation time span in a feasible runtime. 1D, 2D and 3D meshes have to be combined and mapping of data between these meshes is necessary. The simulation should be able to run in parallel on multiple cores to efficiently exploit today's hardware and reduce the runtime to a minimum. Input and output has to be processed in proper data formats for different purposes, ranging from convenient debugging using small and simple scenarios to efficiently storing and visualizating large datasets in production runs. The configuration of model and solver parameters has to be well organized and documented to allow an efficient workflow.

In the following, we present details on the use and implementation of the software OpenDiHu, which aims to fulfill the mentioned requirements. OpenDiHu is an Open Source software framework that solves static and dynamic multi-physics problems using the Finite Element Method. It is used to simulate the multi-scale models presented in the last chapter: biophysical problems describing the biomechanics and neurophysiology of the musculoskeletal system.

1.1 Design Goals

In addition to the functional requirements that are mentioned in the introduction, further demands can be formulated from the developer perspective. The program code should be modular and well documented to allow discovery, reuse and extension in the future. The implemented models should produce correct results and the correctness should be testable in order to be preserved during code changes.

With these user and developer requirements in mind we develop our software framework named OpenDiHu. The name originates from the Digital Human project that is entitled 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].

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 introduced field of requirements from user centric to developer centric properties.

1.1.1 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 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 postprocessed directly in the Python settings script.

1.1.2 Perfomance

The second design goal is to achieve good performance. OpenDiHu satisfies this goal by supporting parallel execution on the one hand and 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* [Bal97a] for the parallel storage of vectors and matrices and for linear algebra. PETSc provides a large collection of preconditioners, linear and nonlinear solvers that can be chosen at runtime. At the same time, it allows low-level access to the locally stored data, which, e.g., allows us to optimize data transfer between different vectors.

For multiscale 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 the most runtime should be the actual computations and not memory allocations or data transfer operations. Using these insights, the framework was constructed in an efficient way, e.g., by avoiding repeated memory initializations and expensive copy operations whenever possible.

1.1.3 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 numeric coupling library *PreCICE* [Bun16] to numerically couple to solvers in different external software packages. The surface coupling adapter allows to couple 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 forgoes 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 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 to write 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 if 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.

1.1.4 Overview of this Chapter

The aim of this chapter is to describe how the software is used and how it is implemented. We begin with an introduction of its usage in Sec. 1.2. The remainder of this chapter addresses the implementation. Section 1.4 presents details on the data handling based on the library PETSc. Section 1.5 describes the assembly of Finite Element matrices and shows an algorithm for Dirichlet boundary conditions. TODO

1.2 Usage of OpenDiHu

We begin with a description of the aspects of OpenDiHu that are relevant from a user's perspective. This section outlines the basic organization of the repository in Sec. 1.2.1, the installation procedure in Sec. 1.2.2 and demonstrates the usage of given example scenarios in Sections 1.2.3, 1.2.4, and 1.2.6. Section 1.2.7 summarizes all available solver classes.

1.2.1 Organization of the Repository

All code is contained in a single git repository which is hosted on github at `https://github.com/maierbn/opendihu`. In addition, documentation is hosted on the "Read the Docs" website under `https://opendihu.readthedocs.io` [Mai21]. 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 approximately the contents shown in Fig. 1.1, which will be explained in the following. The software consists of a core library that provides all functionality such as solvers and data handling. In addition, examples are created that setup 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 circa 90 000 code lines, 24 000 blank lines and 19 000 comment lines contained in circa 700 files and structured in a directory tree with circa 70 total subdirectories.

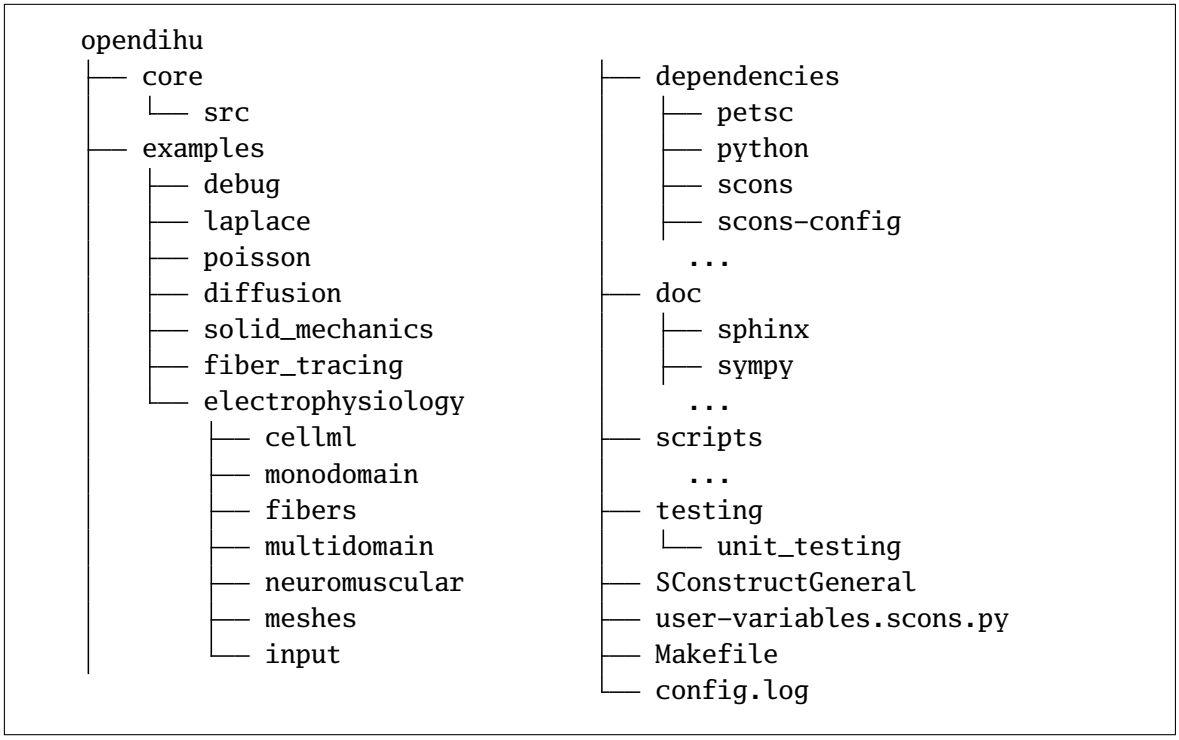


Figure 1.1: Contents of the main opendihu directory.

The `examples` directory contains all simulation scenarios that are packaged with OpenDiHu. Each of the circa 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. 1.1 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 **??**, and the electrophysiology models. The electrophysiology examples are further structured as given in Fig. 1.1 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 and 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 [Mai21] 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.

1.2.2 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.

The third step performs a similar action, as it builds and links three 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* [Bal97a]; [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* [Kre12a]; [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 scripts 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 setup OpenDiHu on a new Linux computer by only executing a single "scons" command.

The top-level files listed in Fig. 1.1 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 determine the 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. 1.2.3, a simulation of muscle contraction in Sec. 1.2.4 and a simulation of the neuromuscular system in Sec. 1.2.6.

1.2.3 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 is listed in Figures 1.2 and 1.3. 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 1.2 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. 1.3. 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.

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. 1.3, 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 [Mai21] and also sketched by the comments in the file. Some parameters will be presented in the following.

```
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 1.2: Example source file of an OpenDiHu solver for the 2D Laplace problem.

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 specifies 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 from 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 1.4 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.

```
nx = 10;       ny = nx          # number of elements
mx = 2*nx + 1; my = 2*ny + 1    # number of nodes

# specify boundary conditions
import numpy as np
bc = {}
for i in range(mx):
  x = i/mx

  # bottom line
  bc[i] = np.sin(x*np.pi)

  # top line
  bc[(my-1)*mx + i] = np.sin(x*np.pi)

  print("{}, bc: {}, {}".format(i, bc[i], bc[(my-1)*mx + i]))

config = {
  "solverStructureDiagramFile":     "solver_structure.txt",      # diagram file
  "logFormat":                      "csv",     # "csv" or "json", format of log
  "scenarioName":                   "laplace", # scenario name for log file
  "mappingsBetweenMeshesLogFile":   None,      # a log file about mappings
  "FiniteElementMethod": {
    # mesh parameters
    "nElements":                    [nx, ny],   # number of elements in x and y
    "inputMeshIsGlobal":            True,       # if nElements is a global number
    "physicalExtent":               [1.0, 1.0], # physical domain size
    "physicalOffset":               [0, 0],     # physical location of origin

    # problem parameters
    "dirichletBoundaryConditions": bc,          # Dirichlet BC as dict
    "dirichletOutputFilename":     None,        # output file for Dirichlet BC
    "neumannBoundaryConditions":   [],          # Neumann BC
    "prefactor":                   1,           # constant prefactor c in c∆u

    # linear solver parameters
    "solverType":                  "gmres",     # linear solver scheme
    "preconditionerType":          "none",      # preconditioner scheme
    "relativeTolerance":           1e-15,       # stopping criterion, rel. tol.
    "absoluteTolerance":           1e-10,       # stopping criterion, abs. tol.
    "maxIterations":               1e4,         # maximum number of iterations
    "dumpFormat":                  "default",   # format for data dump
    "dumpFilename":                None,        # filename for dump
    "slotName":                    None,        # connector of solver

    # output writers
    "OutputWriter" : [
      {"format": "Paraview",   "filename": "out/laplace", "binary": False},
      {"format": "PythonFile", "filename": "out/laplace", "binary": False},
    ]
  }
}
```

Figure 1.3: Python settings file of the Laplace example corresponding to the source file in Fig. 1.2.

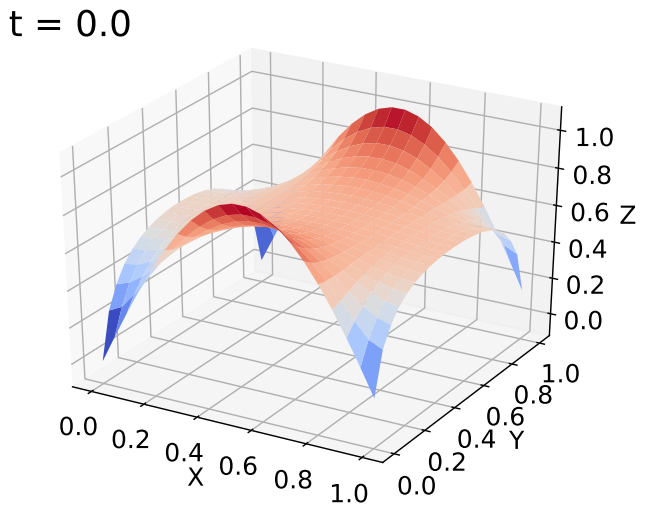


Figure 1.4: Visualization of the solution of the exemplary 2D Laplace problem. The plot was created using the `plot` command on the output of the `PythonFile` output writer.

1.2.4 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 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 1.5 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 here 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

```
#include <cstdlib>
#include "opendihu.h"

int main(int argc, char *argv[])
{
  // 3D multidomain coupled with contraction

  // initialize everything, handle arguments and parse settings from input file
  DihuContext settings(argc, argv);

  typedef Mesh::StructuredDeformableOfDimension<3> MeshType;

  Control::Coupling
  <
    OperatorSplitting::Strang<
      Control::MultipleInstances<     // subcellular model
        TimeSteppingScheme::Heun<
          CellmlAdapter<
            57,71,  // nStates,nAlgebraics: 57,71 = Shorten, 4,9 = Hodgkin Huxley
            FunctionSpace::FunctionSpace<MeshType,BasisFunction::LagrangeOfOrder<1>>
          >
        >
      >,
      TimeSteppingScheme::MultidomainWithFatSolver<       // multidomain
        SpatialDiscretization::FiniteElementMethod<       // FEM for initial potential flow
          MeshType,
          BasisFunction::LagrangeOfOrder<1>,
          Quadrature::Gauss<3>,
          Equation::Static::Laplace
        >,
        SpatialDiscretization::FiniteElementMethod<       // anisotropic conduction
          MeshType,
          BasisFunction::LagrangeOfOrder<1>,
          Quadrature::Gauss<5>,
          Equation::Dynamic::DirectionalDiffusion
        >,
        SpatialDiscretization::FiniteElementMethod<       // isotropic conduction in fat layer
          MeshType,
          BasisFunction::LagrangeOfOrder<1>,
          Quadrature::Gauss<5>,
          Equation::Dynamic::IsotropicDiffusion
        >
      >
    >,
    MuscleContractionSolver<          // solid mechanics
      Mesh::CompositeOfDimension<3>
    >
  > problem(settings);

  problem.run();

  return EXIT_SUCCESS;
}
```

Figure 1.5: 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 **??**, 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 **??**. 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 **??**. In 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 setup the stiffness and mass matrices for these subproblems.

Several different meshes are involved in the definition of this example. As described in **??**, the computational domain consists of a muscle domain and a fat domain. Both these domains are discretized by a 3D structured mesh, which is given as `MeshType` in line 11. This type is referenced for the subcelullar model in line 20 and for the conduction parts in lines 26, 32, and 38. Only in the anisotropic conduction class it refers to the fat domain, in the other uses it specifies the muscle domain. For the muscle contraction solver in

```
config = {
  "scenarioName": "multidomain_contraction",
  "Solvers": {
    "potentialFlowSolver": {...},
  },
  "Coupling": {
    "timeStepWidth": 1e-3,
    "Term1": {            # multidomain
      "StrangSplitting": {

        "Term1": {        # subcellular model
          "MultipleInstances": {
            "nInstances": variables.n_compartments,
            "instances": [        # settings for each motor unit
            {
              "ranks": list(range(n_ranks)),
              "Heun": {
                "CellML" : {
                }
              }
            } for compartment_no in range(variables.n_compartments)]
          },
        },
        "Term2": {        # conduction term of multidomain
          "MultidomainSolver": {

            "PotentialFlow": {
              "FiniteElementMethod": {
                "solverName": "potentialFlowSolver",
              },
            },

            "OutputWriter": [
            ]
          },
        }
      },
    },
    "Term2": {        # solid mechanics
      "MuscleContractionSolver": {

        # the actual solid mechanics solver
        "DynamicHyperelasticitySolver": {
         }
      }
    }
  }
}
```

Figure 1.6: Excerpt of the settings file for the multidomain and solid mechanics solver.

line 46, we use a different, “composite” type of mesh. This type is a combination of the two structured meshes for the body and the fat domains, 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. 1.6. 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 CellML methods, not shown in Fig. 1.6, 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 is 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. 1.2.3, also appear in the example of this section, once for each of the three occurences of this class. The excerpt of the settings file in Fig. 1.6 shows one of these specifications, for 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. 1.3. To avoid duplication of this information and to share linear solvers and meshes, these parameters are not repeated for every class where 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 where they are used. In the example settings in Fig. 1.6, 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 whereever the solver is referenced by its solver name.

The meshes use an analogous approach where all meshes can be defined under a global

"Meshes" keyword (not shown in Fig. 1.6) and referenced in the solver objects by their "meshName". This is helpful especially for meshes with a large number of node positions that can be specified once and reused throughout all solvers.

Output of the results is written to files by output writers that are defined as shown in the previous Laplace example in Fig. 1.3. Almost all solver classes allow to configure associated output writers. In Fig. 1.6, 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 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. 1.2.3, 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. 1.6 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. 1.6 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 from 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. These type of command line arguments 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 possiblity 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.

1.2.5 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 that 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 1.7 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 one solver to a data connector slot of the other solver. 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 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
│   │   │   (...)                                        :::::::                  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 1.7: Solver structure diagram that shows the data connections of the solvers.

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 "¤" and a number on the right, together with their custom name.

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. 1.7, the first four data corresponding slots 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. 1.7 shows the field variables for slots 0 to 2 ending in "`_0`" for compartment $k = 0$. Similar field variables exist for $k = 1, \ldots, N_{\text{MU}}$. 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. Thus, 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. 1.7, 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. **??**).

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.

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 subscellular 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 has 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 possiblities 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. 1.7 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 possiblity are also listed at the top of the diagram, here in lines 5 and 6.

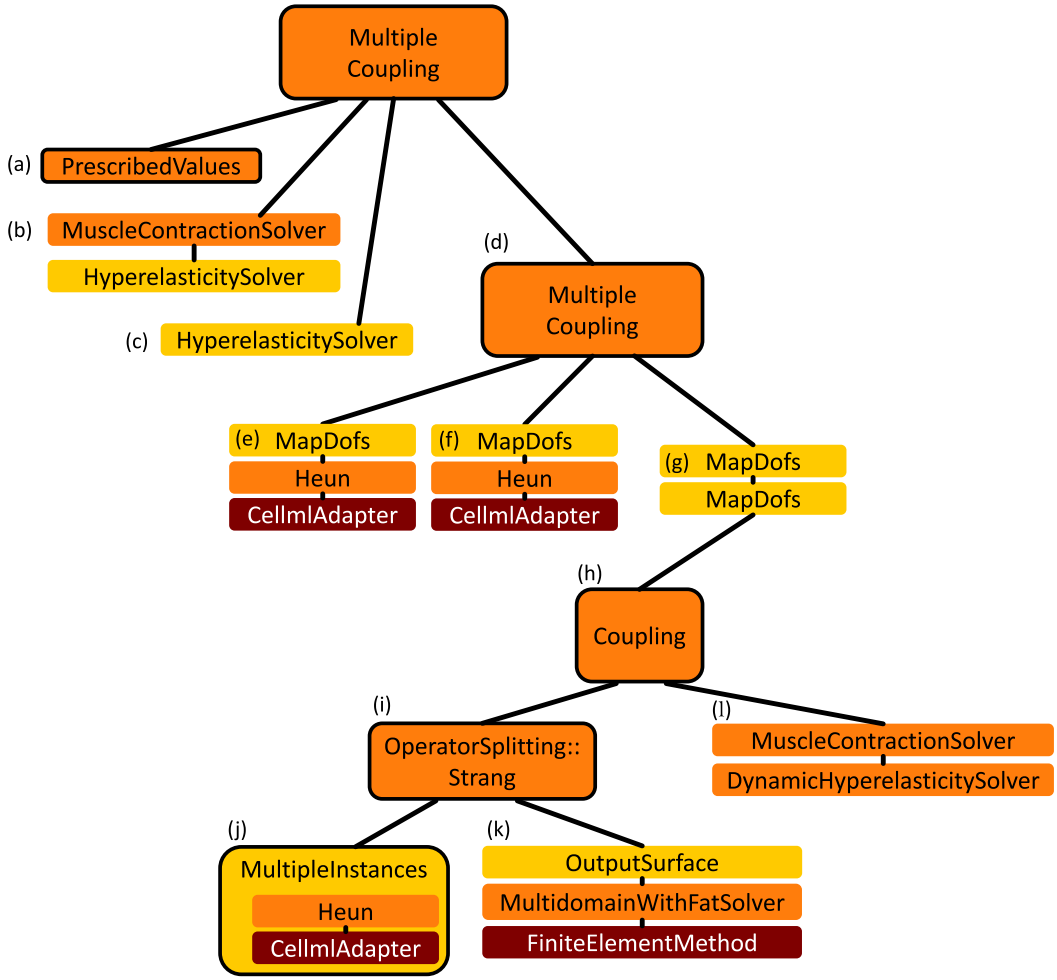


Figure 1.8: Solver tree for a simulation of the neuromuscular system. Classes of type `DiscretizableInTime` are shown as red boxe, `TimeSteppingScheme`s are given by orange boxes.

1.2.6 Exemplary Usage: Neuromuscular System

In the example in the last sections Sections 1.2.4 and 1.2.5, 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. 1.8. 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 any degrees of freedom between two field variables, apply custom python functions on the values and communicate values between processes in parallel execution.

The subtree under (h) is identical to the example presented in Sec. 1.2.4. 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. 1.8 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. 1.8, further solver classes are available in OpenDiHu. A comprehensive list of all available solver classes is given in the following section.

1.2.7 Summary of Existing Solver Classes

All the timestepping schemes introduced in **??** 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. 1.2.6, `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 **??**. The `MultidomainSolver` and `MultidomainWithFatSolver` classes solve the multidomain models **????** 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 **??**.

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 **??**. The `HyperelasticitySolver` class solves the static hyperelasticity formulation for any material model, as presented in **??**. The `DynamicHyperelasticitySolver` class inherits from the `HyperelasticitySolver` class and adds functionality to solve the dynamic hyperelasticity formulation shown in **??**. 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 equations in analytic form 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 numeric 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 extract 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 implicitely or explicitely 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 [Mai21].

TODO mention GUI

1.3 Usage of CellML models

1.4 Data Handling with PETSc

OpenDiHu processes various types of values: 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 this 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]; [Bal97b] 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 and the rank of a process is its number in this communicator, i.e., a consecutive number starting with zero.

1.4.1 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, structured meshes with linear ansatz functions are considered.

The partitioning of a structured, 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. 1.9 a 1D mesh with $N_x^{\text{el}} = 6$ elements is partitioned into three subdomains with two elements each. Figure 1.10 (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. 1.10 (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 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 ranks, the node is stored as so called *ghost* node. In contrast, the other local nodes are in the following called *non-ghost* nodes.

The assignment of nodes to processes leads to the 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 having one layer of nodes more than other subdomains. This effect is opposite to the number of elements which is potentially higher for subdomains with lower index. Therefore, the total number of nodes and dofs is approximately equally distributed. Moreover, in the limit for $N_x^{\text{el}}, N_y^{\text{el}}, N_z^{\text{el}} \to \infty$ the imbalance vanishes totally.

In the examplary partitionings in Fig. 1.9 (b) and Fig. 1.10 (b), owned nodes are represented by black circles and numbers, ghost nodes are represented by yellow circles and numbers. In the 1D example in Fig. 1.9, the three subdomains with two elements each have three, two and two nodes. In the 2D example in Fig. 1.10 the six subdomains have

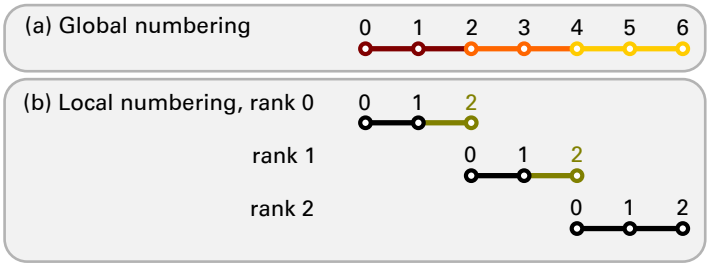


Figure 1.9: Partitioning and local and global numberings of a 1D mesh with $N_x^{\text{el}} = 6$ elements partitioned to $n_x = 3$ processes.

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 nodes count.

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 $\tilde{\mathbf{v}} \in \mathbb{R}^{n_{\text{global}}}$ 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 $n_{\text{local\_without\_ghosts}} \leq n_{\text{global}}$ values of the global vector of dofs. Therefore, every dof is *owned* by exactly one rank. 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 $0, \ldots, n_{\text{local\_without\_ghosts}} - 1$, the ghost dofs follow at positions $n_{\text{local\_without\_ghosts}}, \ldots, n_{\text{local\_with\_ghosts}} - 1$. 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 1.9 (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 in given in Figure 1.9 (a). It is needed if global operations have to be performed with the Vec, e.g., computing matrix vector products.

The computation of mass and stiffness matrices for the Finite Element Method proceeds by iterating over the elements of a mesh and computing contributions at the dofs of every element. Additional material data stored at the dofs may be used in this process, such

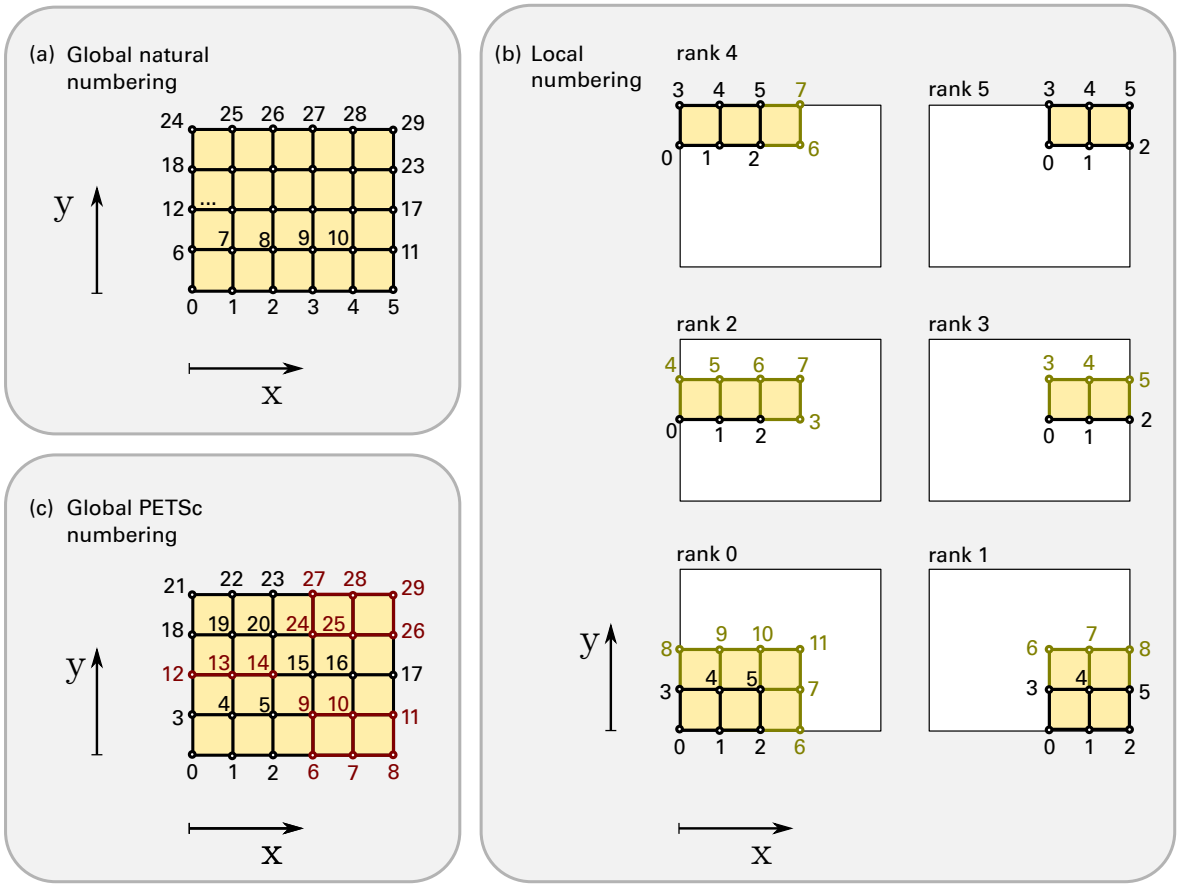
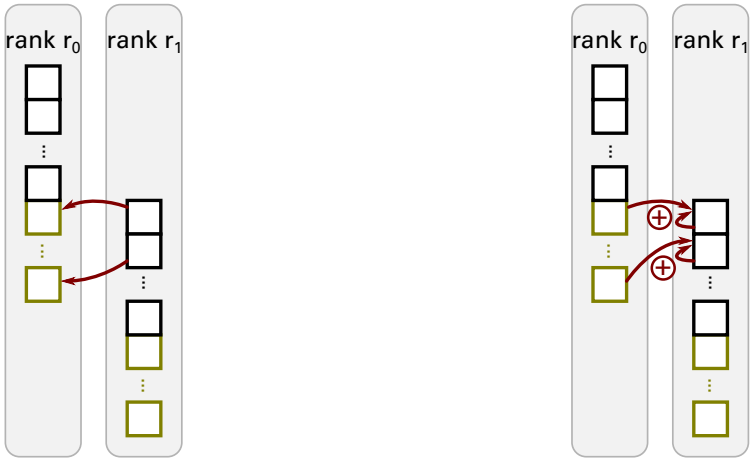


Figure 1.10: Subdomains and numberings of a 2D mesh with $N_x^{\text{el}} \times N_y^{\text{el}} = 5 \times 4$ elements partitioned to $n_x \times n_y = 2 \times 3$ processes. (a)-(c) show the different numberings needed for (a) boundary condition specification, (b) identification of local non-ghost dofs (black) and local ghost dofs (yellow), and (c) identification of global dofs.

as values of a diffusion tensor. During matrix assembly, the contributions of all elements that are adjacent to a given dof need to be added up to yield the corresponding matrix entry. Some of the dofs are ghost dofs. To yield a correct computation, the ghost dofs initially need to receive the material data from the corresponding non-ghost dof. After all element contributions have been computed, the value collected at a ghost dof needs to be communicated back and added to the corresponding dof value on the rank where the dof is non-ghost.

PETSc provides functionality for these two operations: First, communicating the values from the owning rank to the ghost buffers at all other ranks where the respective dofs are ghosts. Second, 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. Figure 1.11a and Fig. 1.11b visualize the data flow in the two operations.



(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 1.11: 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 (yellow). The red arrows indicate the data transfer.

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()`.

1.4.2 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. 1.9 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. 1.10 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. 1.10 (a) for a 2D mesh and in Fig. 1.12 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 easy understandable from a users's perspective and independent of the partitioning.

The second numbering scheme is the *local* numbering. An example is given in Fig. 1.10 (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

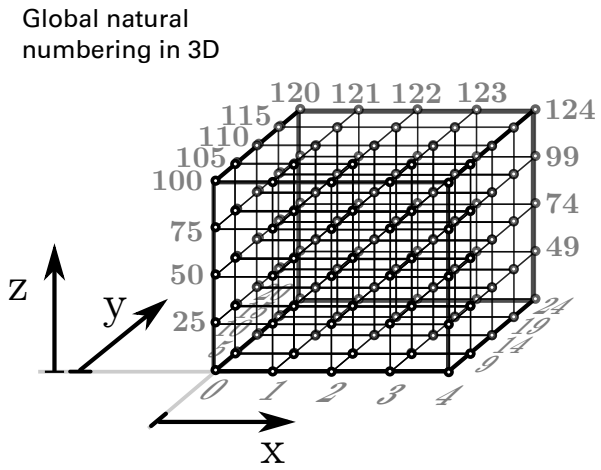


Figure 1.12: Global natural numbering of nodes in a mesh with $4 \times 4 \times 4$ linear elements.

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. 1.10 (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.

1.4.3 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. 1.13 with the data representation in raw memory at the top and increasing abstraction towards the bottom of the figure.

In the two situations where 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. 1.13, 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. 1.13, 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. Because of the shared memory, no costly value copy operation is needed for this action.

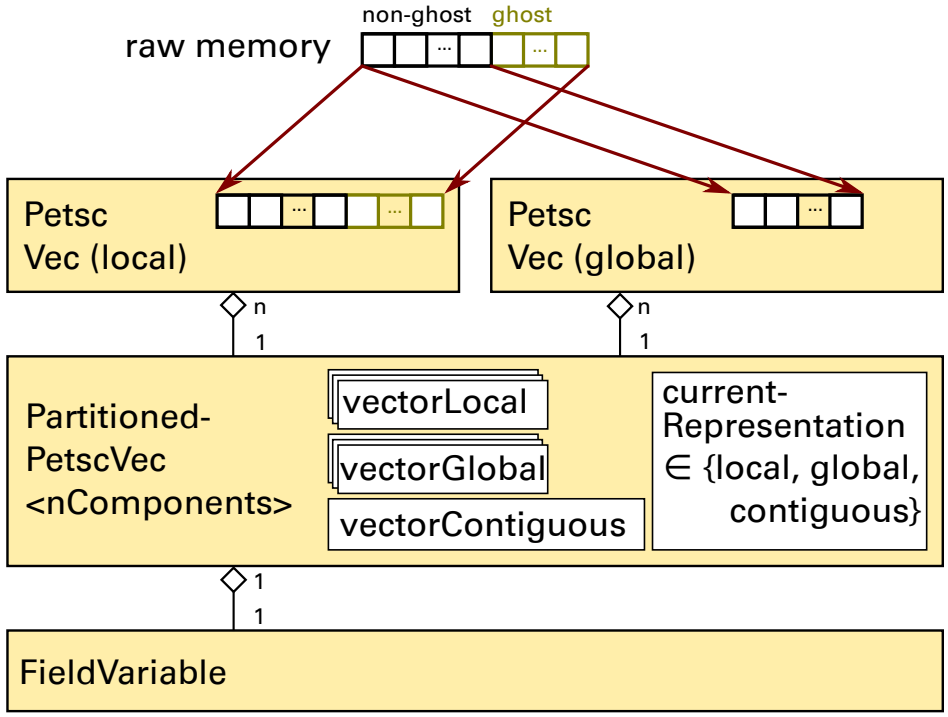


Figure 1.13: Classes in OpenDiHu that represent vectors in parallel execution. The abstraction layer increases from raw memory at the top to the `FieldVariable` at the bottom.

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 that 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 coherent 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 storage. 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 representation is fast. Despite occuring 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. 1.13 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.

1.4.4 Discussion of Several Design Decisions

In the following, some of the previously presented design decisions 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 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 $n_{\text{global}} \to \infty$, wherease the runtime in the approach of OpenDiHu stays constant.

For OpenDiHu, the AoS approach with separate Vecs was chosen for three reasons. First, it is more cache efficient than the alternative during computation of the subcellular model, as explained in Sec. 1.4.3.

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 **??**. 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 Vecs, 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 but rather stored in variable, 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 FieldVariables 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.

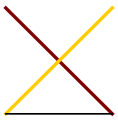
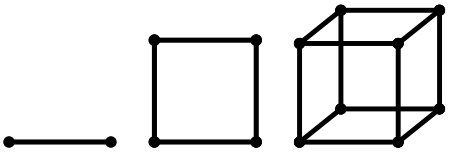
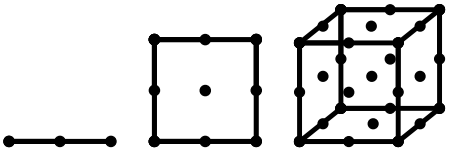
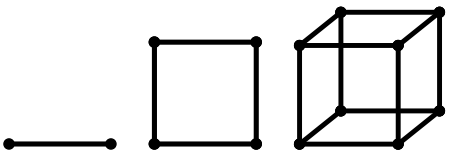
Ansatz functions	Element shapes 1D 2D 3D
 $\phi_0(\xi) = 1-\xi,$ $\phi_1(\xi) = \xi$	
 $\phi_0(\xi) = (2\xi-1)(\xi-1),$ $\phi_1(\xi) = 4(\xi-\xi^2),$ $\phi_2(\xi) = 2\xi^2-\xi$	
 $\phi_0(\xi) = 2\xi^3-3\xi^2+1,$ $\phi_1(\xi) = \xi(\xi-1)^2,$ $\phi_2(\xi) = \xi^2(3-2\xi),$ $\phi_3(\xi) = \xi^2(\xi-1)$	

Table 1.1: 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.

1.4.5 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 1.1 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 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. 1.1.

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 based 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 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., $f \in C^0(\Omega, \mathbb{R})$.

For quadratic Lagrange and cubic Hermite basis functions, the numbering schemes presented in Sec. 1.4.2 have to be adjusted such that at every node all dofs are enumerated in sequence before the numbering continues at the next node.

1.4.6 Implemented Types of Meshes

Meshes of different types can be selected independently of the choice of basis functions. Three types are supported. Figure 1.14 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) or a rectangular (2D) or 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

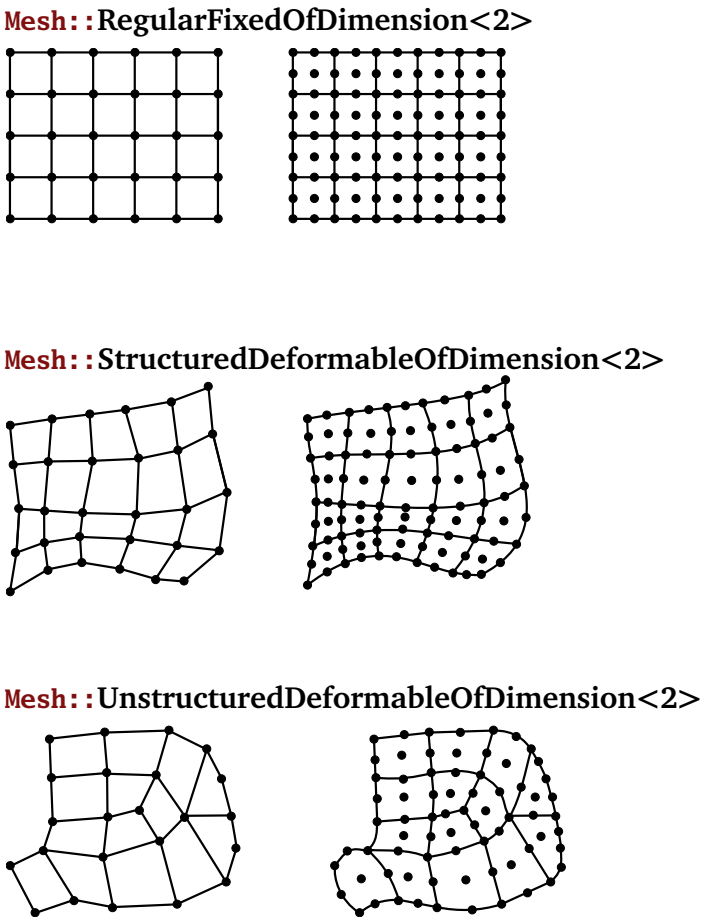


Figure 1.14: The three implemented mesh types in OpenDiHu, each time for 2D linear Lagrange or Hermite ansatz functions (left) and for 2D quadratic Lagrange ansatz functions (right).

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. 1.14.

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 implicitely. The example at the lower third of Fig. 1.14 shows capabilities of 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 commuinty 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 explicitely while it is given implicitely 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.

1.4.7 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 1.15a 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

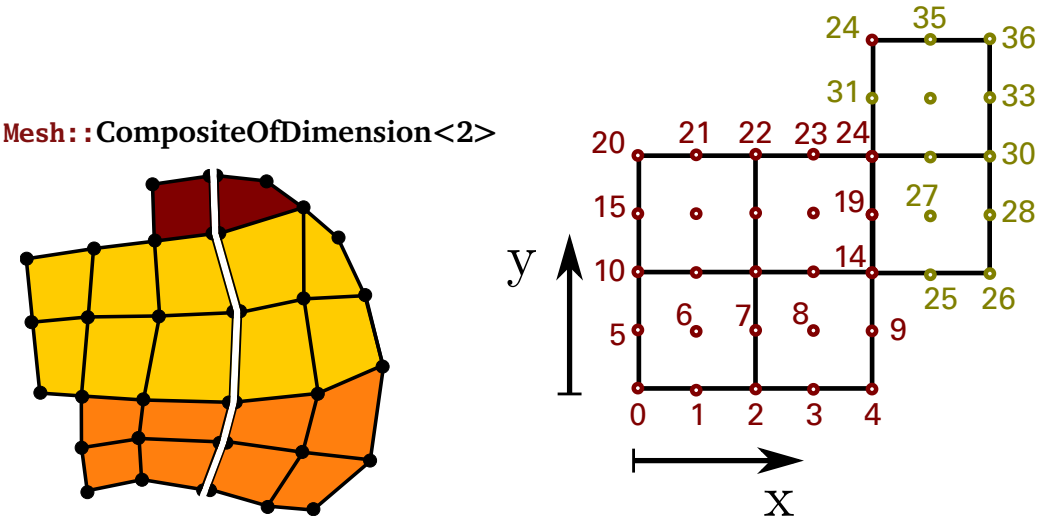
over all remaining nodes of the second submesh and so on, until all nodes are numbered. The numbering of dofs is analogous. Figure 1.15b shows an example with two quadratic submeshes with four and two elements. The resulting composite mesh has six elements. The node numbers in the first structured mesh 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 reqiured by the parallel PETSc `Vec`s.

For the construction of these numberings, the shared nodes of different submeshes that 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 $1 \cdot 10^{-5}$ 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 to 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 1.15a shows a valid partitioning of the exemplary composite mesh to two subdomains.



(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 that is created from two quadratic meshes.

Figure 1.15: Examples for composite meshes that combine the advantages of structured and unstructured meshes.

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.

An application of composite meshes is the biceps muscle with a fat and skin layer. Figure 1.16 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.

1.5 Finite Element Matrices and Boundary Conditions

Another important mathematical object besides the vector that 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.

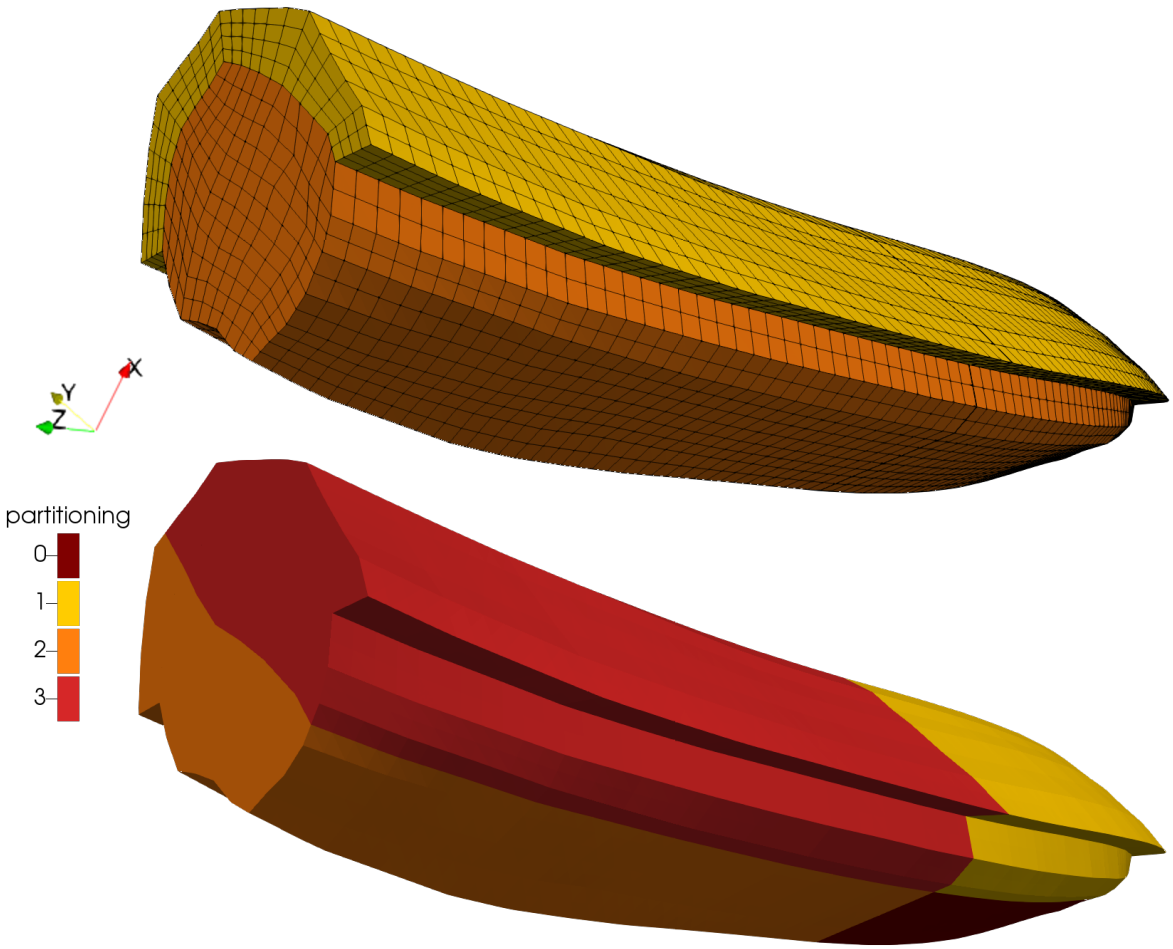


Figure 1.16: Composite mesh of the biceps muscle. Top: the two structured meshes from which the composite mesh is created, bottom: partitioning to four processes.

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.

1.5.1 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 stiffness 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 $\mathbf{f}$. 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:

$$\begin{aligned} n_{\text{1D\_overlaps}} &= (2\, n_{\text{dofs\_per\_basis}} - 1) \cdot n_{\text{dofs\_per\_node}}, \\ n_{\text{non-zeros}} &= \left(n_{\text{1D\_overlaps}}\right)^d. \end{aligned} \tag{1.1}$$

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

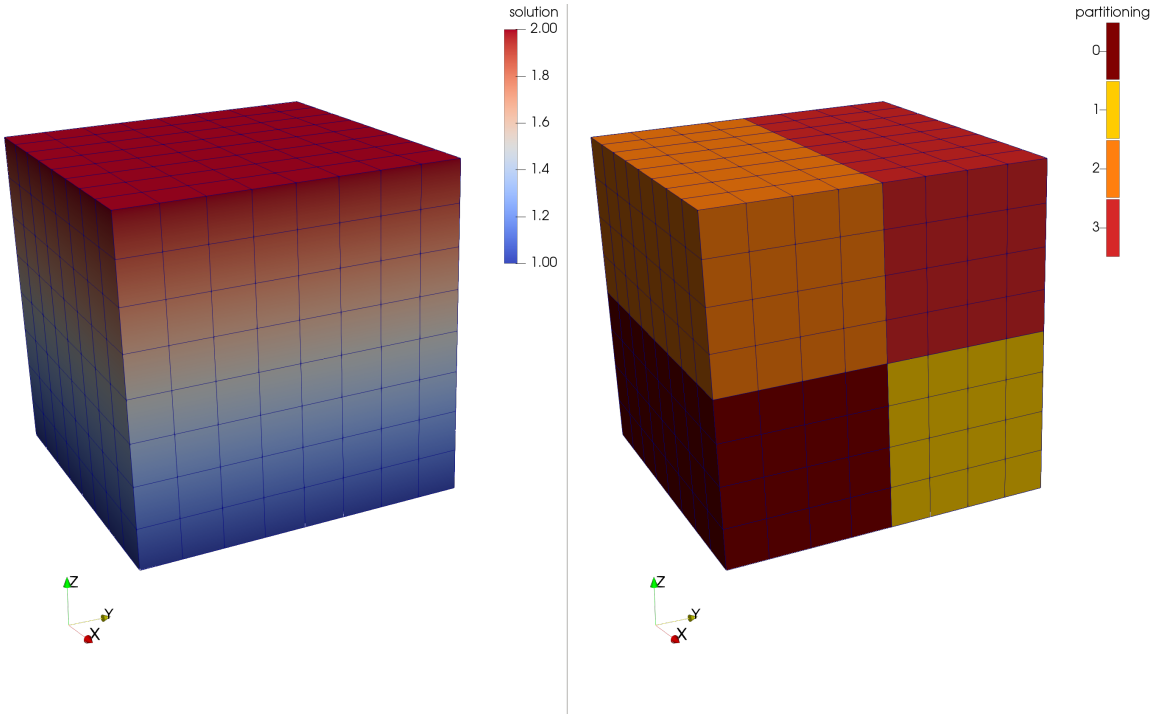


Figure 1.17: Solution of the Laplace equation $\Delta \mathbf{u} = \mathbf{f}$ with prescribed values $\mathbf{u}|_{\text{bottom}} = 1$ and $\mathbf{u}|_{\text{top}} = 2$. The mesh consists of $4 \times 4 \times 4$ quadratic elements and, thus, 9^3 nodes. Left: Solution, right: Partitioning to four processes.

$n_{\text{non-zeros}}$ 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 examplary 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 the top. The solution is visualized in the left of Fig. 1.17. The computation is performed with four processes. Figure 1.17 shows the partitioning on the right.

The non-zero estimates computed by Eq. (1.1) are $n_{\text{1D\_overlaps}} = 5$ and $n_{\text{non-zeros}} = 125$. The mesh has 4 elements and, thus, 9 nodes per coordinate direction and therefore $n_{\text{global}} = 9^3 = 729$ dofs. Figure 1.18 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 are to 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

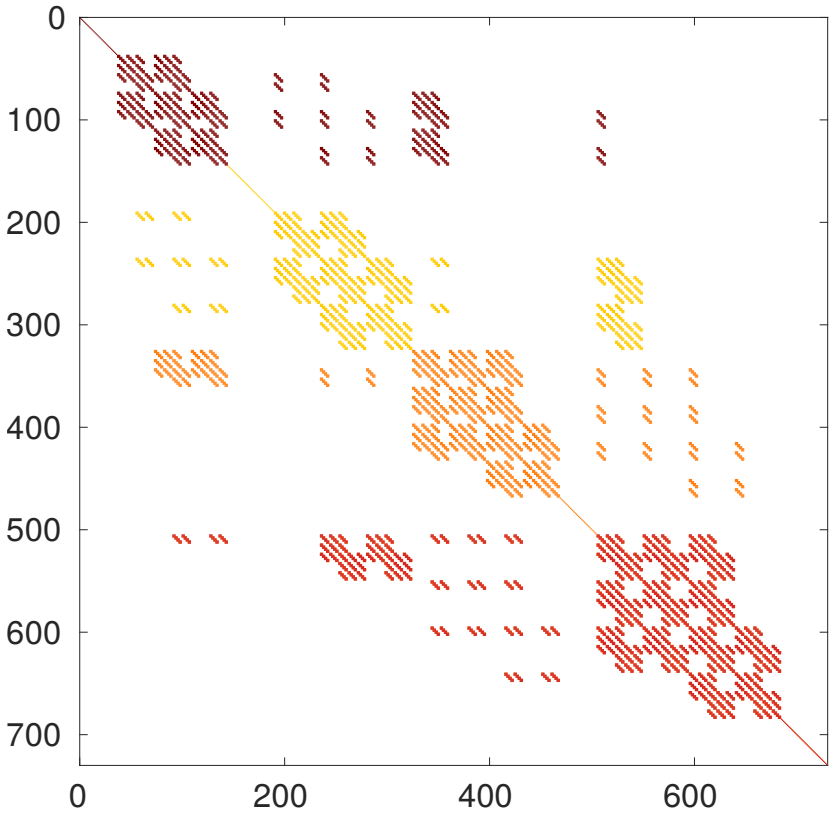


Figure 1.18: Sparsity pattern, i.e., locations of non-zero entries of the 729×729 stiffness matrix $\mathbf{K}$ for the example problem in Fig. 1.17. The rows for the four processes are given by different colors matching the partitioning in Fig. 1.17. The processes have 144, 180, 180 and 225 local dofs.

of non-zero entries is 35 937 which is circa 20 % of the preallocated values.

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. 1.18, 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.

1.5.2 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

$$m_{i,j} = \int_{\Omega} I(\mathbf{x})\,\mathrm{d}\mathbf{x}, \tag{1.2}$$

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. 1. Multiple variants of this algorithm that only differ in their achieved performance have been implemented for evaluation purposes. They will be discussed in Sec. 1.5.3. The listed algorithm in Alg. 1 shows the fastest variant.

Algorithm 1 Finite Element matrix assembly

```
1  procedure Assemble FE system matrix
2    for elements e = {e_1, e_2, e_3, e_4} 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 e_1, e_2, e_3, e_4
10         columns = dofs j of elements e_1, e_2, e_3, e_4
11         matrix[rows,columns] = matrix_entries[i,j]
12   Call PETSc final matrix assembly
```

The main loop in line 1.2 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* [Kre12b]; [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. (1.2), 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 1.3 - 1.5 iterates over the respective sampling points $\xi \in [0,1]^d$ in the element coordinate system. In line 1.4, 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 1.5, 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 $i,j \in \{0,1,\ldots,n_{\text{dofs\_per\_element}}-1\}$ 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 1.6. 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 1.7 - 1.11 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 1.9 and 1.10 and the respective computed value is assigned in line 1.11. 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 1.12 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 1.2, 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. (1.2). 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 explicitely stored in the elements. To assemble the global matrix, PETSc uses the mapping from local to global numbering, which it can maintained 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 that would require memory of $\mathcal{O}(n_{\text{global}})$ is avoided in all algorithms to ensure good parallel weak scaling properties.

1.5.3 Performance of the Algorithm for Parallel Matrix Assembly

In the following, the performance of two variations of the algorithm in Alg. 1 will be examined. The first variation is to not use explicit vectorization and, thus, in line 1.2 to iterate over the elements one by one instead of the groups of four elements.

The second variation is to not accumulate multiple values for the application of the quadrature scheme in line 1.6. Instead, the loop over the sampling points in line 1.3 is made the inner-most loop and placed inside the loop in line 1.8. 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 1.4 and 1.5 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. 1.17 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

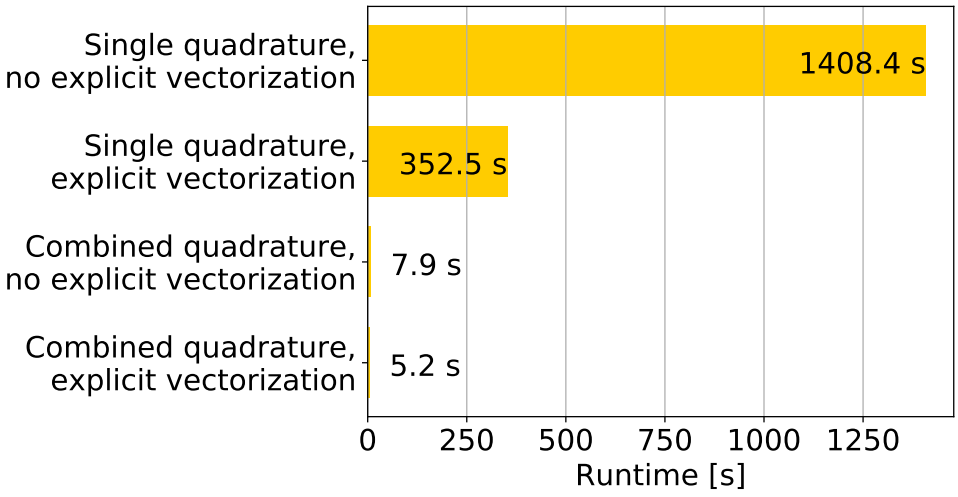


Figure 1.19: Runtimes of different optimizations for the algorithm to assemble the FEM stiffness matrix.

total was used. The program was executed with four processes on a 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 bandwith 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 1.19 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. 1.19) the runtime reduces to less than 0.6 %, in the case of explicit vectorization (second and fourth bar from the top in Fig. 1.19) 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. 1.19. 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. 1.19) to the most optimized version (bottom bar in Fig. 1.19) equals a speedup of over 270. Together with the solution of the linear system using an algebraic multigrid preconditioner and a GMRES solver with a residual norm tolerance of $1 \cdot 10^{-10}$, 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. 1.19 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.

1.5.4 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. (1.2) can be precomputed analytically and no numeric 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 combinations 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 1.2 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 to 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. 1.2, 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. 1.3.

Dim.	Element contribution	Node stencil
1D	$\begin{bmatrix} -1 & \underline{1} \end{bmatrix}$	$\begin{bmatrix} 1 & \underline{-2} & 1 \end{bmatrix}$
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 1.2: Stencils of the Finite Element stiffness matrix of Δu for a mesh with uniform resolution 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 1.3: Stencils of the Finite Element mass matrix for a mesh with uniform resolution 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 that otherwise uses the numerical algorithm presented in Sec. 1.5.2. 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. 1. 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.

1.5.5 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 occured have been brought to the right hand side and u_4 has been substituted by the prescribed value $\hat{u}_4$.

Thus, enforcing 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 from 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 to communicate is illustrated with an example.

Figure 1.20 (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

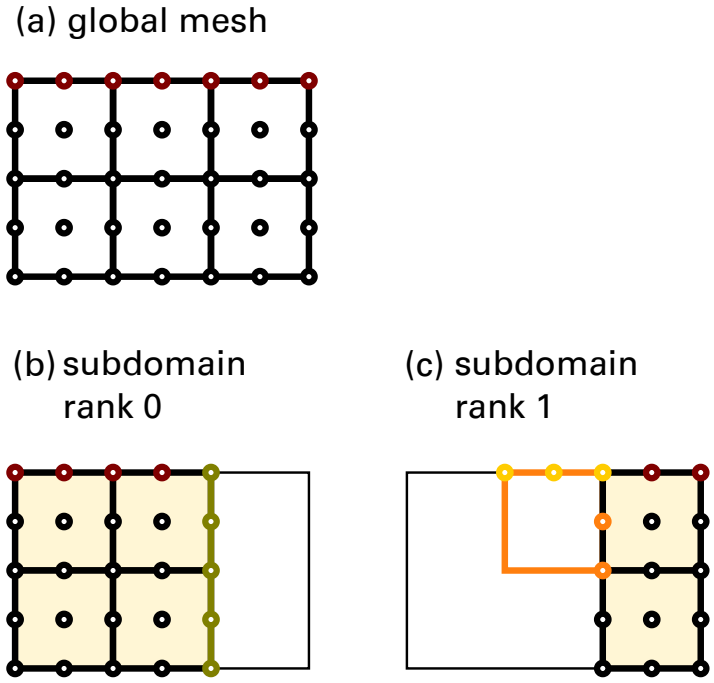


Figure 1.20: Example to illustrate ghost element transfer that is needed for handling Dirichlet boundary conditions in parallel. A mesh with Dirichlet boundary conditions on the red nodes is given in (a). The subdomains for two processes are given in (b) and (c). (c) shows a ghost element in orange that is sent from rank 0 to rank 1.

are partitioned to two processes with subdomains containing four and two elements, as shown in Fig. 1.20 (b) and Fig. 1.20 (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. 1.20 (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. 1.20 (c) by the orange element with the two orange nodes and the three yellow boundary condition nodes and values has to be communicated from rank 0 to rank 1. This element is called *ghost element*. Rank 0 knowns 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 is 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 peformed. 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 that 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 involve a small number of elements depending on the number of boundary conditions. Of these elements, only a subset of nodes are 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.

Is 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.

1.5.6 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:

$$\big(\boldsymbol{\sigma}\,\mathrm{grad}\,u(\mathbf{x})\big)\cdot\mathbf{n} = f(\mathbf{x}) \quad \text{for } \mathbf{x}\in\Gamma_f\,. \tag{1.3}$$

Here, $\boldsymbol{\sigma}$ is a conductivity tensor that 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 **??**, these boundary integrals were summarized by the matrix $\mathbf{B}_{\boldsymbol{\sigma}}$. By using the definition of the Neumann boundary condition in Eq. (1.3), we can derive the following equation for the boundary integral:

$$-\sum_{j=1}^{M} u_j \int_{\Gamma_f} (\boldsymbol{\sigma} \operatorname{grad} \varphi_j \cdot \mathbf{n})\, \varphi_k \,\mathrm{d}\mathbf{x} = -\sum_{j=1}^{M} \int_{\Gamma_f} f_j\, \psi_j\, \varphi_k \,\mathrm{d}\mathbf{x} \tag{1.4a}$$

$$=: \mathrm{rhs}_k, \tag{1.4b}$$

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 (1.4a) is equivalent to the matrix equation **??** and Eq. (1.4b) 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 **??** 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 that is equivalent to Eq. (1.4b) can be formulated as:

$$\mathrm{rhs}_{aM} := -\int_{\partial\Omega} T_a^L\, \psi^L(\mathbf{x})\, \delta u_M \,\mathrm{d}\mathbf{x}. \tag{1.5}$$

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.

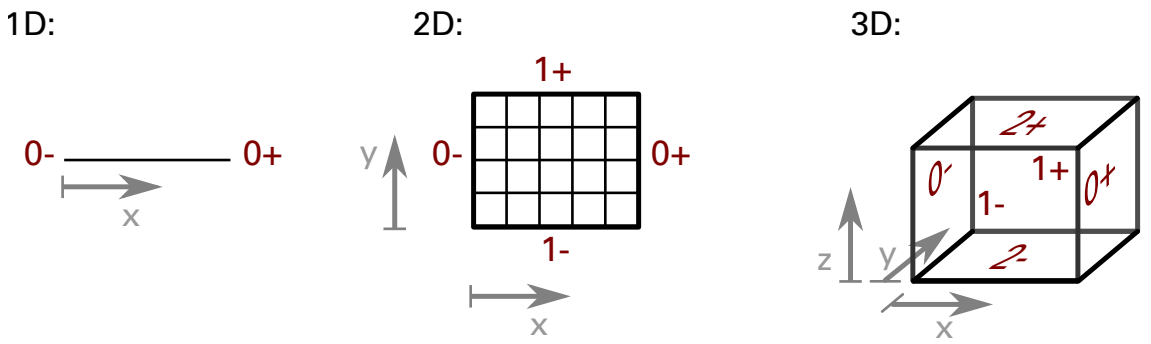


Figure 1.21: Notion of the faces of 1D, 2D and 3D elements, as used in the definition of Neumann boundary conditions.

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. (1.4b) or the vectorial case in Eq. (1.5).

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 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. 1.21.

For elasticity problems where the function $f(\mathbf{x})$ is interpreted as traction force, two more options can be set. The first option is `"divideNeumannBoundaryConditionValues ByTotalArea"`. If set to `True`, the traction force vector is interpreted as a total force on the whole surface. The value of T in Eq. (1.5) 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.

The second option, `"isInReferenceConfiguration"`, allows to switch 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 $\mathbf{F}$:

$$\mathbf{T} = \mathbf{F}^{-1}\,\mathbf{t}. \tag{1.6}$$

Because the implemented model uses the Lagrangian formulation with the right hand side term defined in Eq. (1.5), the transformation in Eq. (1.6) and subsequently the right hand side have to be computed in every timestep of a dynamic problem.

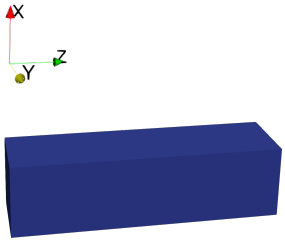
Figure 1.22 illustrates the difference between Neumann boundary conditions that are specified in reference and current configurations. A horizontal, cuboid rod is fixed at its right end and a traction force in positive x direction acts on the surface on its other end. A dynamic hyperelasticity model with isotropic Mooney-Rivlin material is solved.

Figures 1.22a to 1.22c show the simulation results where the traction force is specified in reference configuration, Figures 1.22d to 1.22f shows 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 wheras in the other version the direction of the applied force in the global coordinate system stays constant.

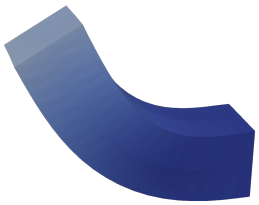
How To Reproduce

Use the following commands to create the results in Fig. 1.22.

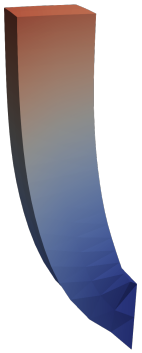
```
cd $OPENDIHU_HOME/examples/electrophysiology/fibers/
  ↪ fibers_contraction/with_tendons_precice/meshes
./create_cuboid_meshes.sh     # create the cuboid mesh
cd $OPENDIHU_HOME/examples/electrophysiology/fibers/
  ↪ fibers_contraction/with_tendons_precice/
  ↪ traction_current_or_reference_configuration
mkorn && srr       # build
./muscle_precice settings_current_configuration.py ramp.py
./muscle_precice settings_reference_configuration.py ramp.py
```



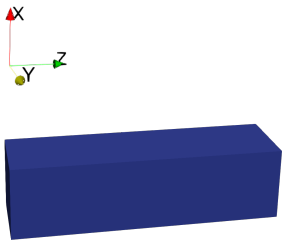
(a) Boundary condition given in reference configuration, $t = 1\,\text{ms}$



(b) Boundary condition given in reference configuration, $t = 30\,\text{ms}$

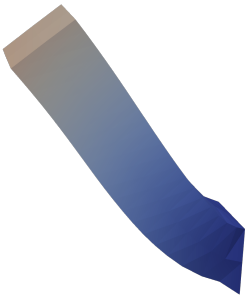


(c) Boundary condition given in reference configuration, $t = 60\,\text{ms}$



(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 1.22: Influence of whether external traction force are defined in current or reference configuration. Simulation of a dynamic solid mechanics problem with a Mooney Rivlin material model and a constant traction force in x direction.

1.6 Parallel Partitioning and Sampling

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 behaviour 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. Sections 1.6.1 and 1.6.2 set the scene and define our requirements for well-behaved parallelized meshes. Sections 1.6.3 and 1.6.4 give details on the implemented algorithms and Sec. 1.6.5 addresses the configuration for the user. Section 1.6.6 concludes by comparing the resulting partitionings for different parameters.

Afterwards, Sections 1.7 and 1.8 TODO present dedicated parallel solvers for various parts of the multi-scale model. Sections (TODO: reference) describe structures that are needed for the interplay of the individual model components.

1.6.1 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. 1.4.1, 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. 1.2.3.

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 $n_{\text{opt}} = n_{\text{proc}}^{1/3}$. The partitioning with the lowest value of p is selected among all partitions as it leads to subdomains with the best volume-to-surface ratio for a cube-shaped domain. An advantage of this method is that it is independent of the mesh size.

1.6.2 Requirements for Partitioning and Sampling of the 3D Mesh

Simulation scenarios with fiber-based electrophysiology use a 3D muscle mesh and embedded 1D fiber meshes that are generated from the same node positions, as described in **??**. The binary input file contains a structured grid of points that 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 is 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.

For illustration, Fig. 1.23 shows the initial fine mesh consisting of 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 parallel partitioned into the colored subdomains. Furthermore, the coarse mesh consists of quadratic elements that are formed from two by two white elements in the cross section each. Hence, every subdomain contains an even number of the white elements in horizontal and vertical directions.

The requirements to the sampling and partitioning algorithm are as follows:

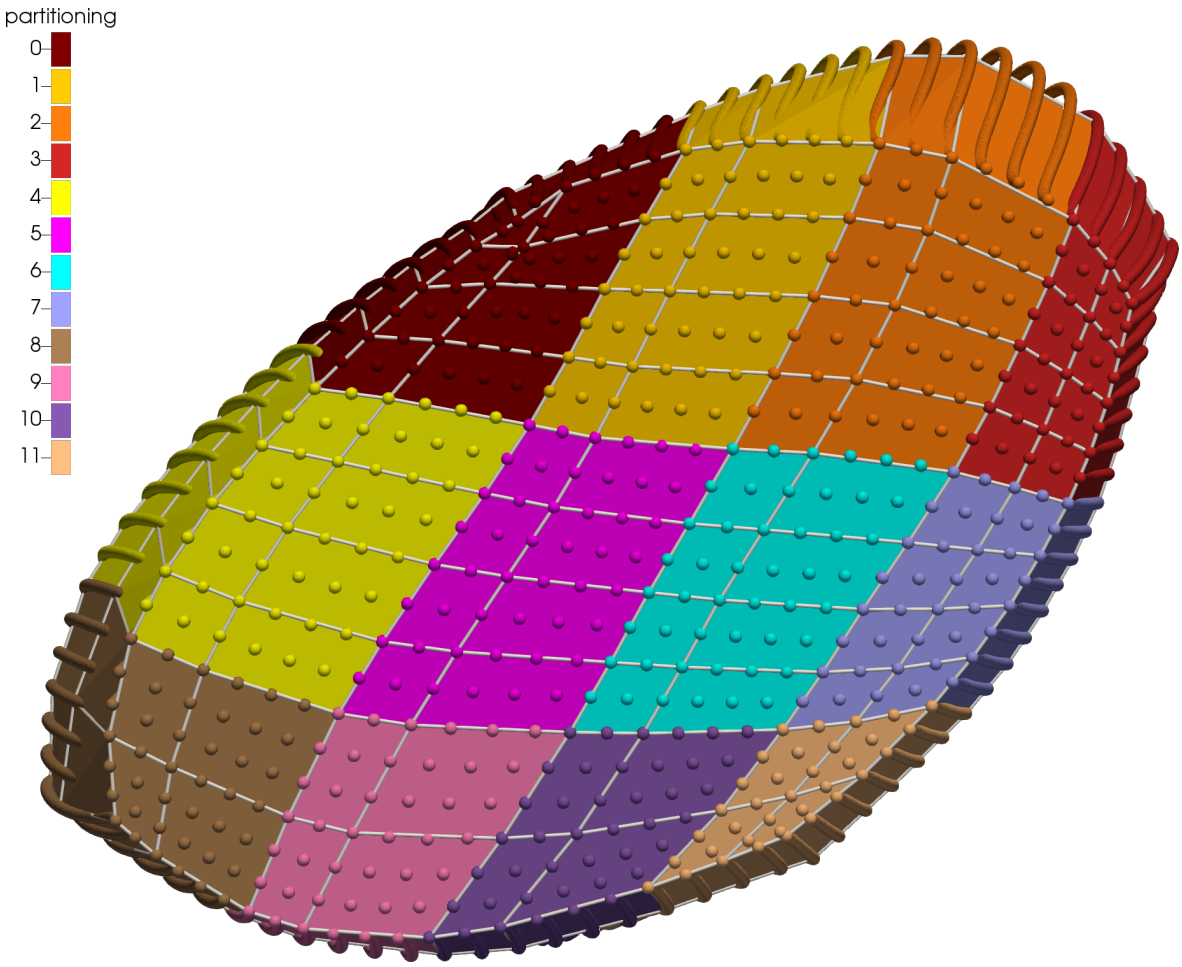


Figure 1.23: Partitioning and subsampling of a fiber mesh to twelve proceses. The fiber data indicated by the spheres is sampled with a stride parameter of two to obtain the partitioned quadratic coarse mesh given by the white elements. The subdomains are indicated by different colors.

(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 mesh is required, e.g., for solid mechanics models, the number of (linear) 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 numeric 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 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 always two neighboring points are in the same element. Then, subdomain boundaries and element boundaries can only occur at every second node.

1.6.3 Algorithm for Determining the Subdomains

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.

The algorithm for the first step is given in Alg. 2. 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.

Figure 1.24 provides a visualization of the algorithmic steps, corresponding to the partitioning in vertical direction of the mesh shown in Fig. 1.23. Figure 1.24 (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. 1.24 (a).

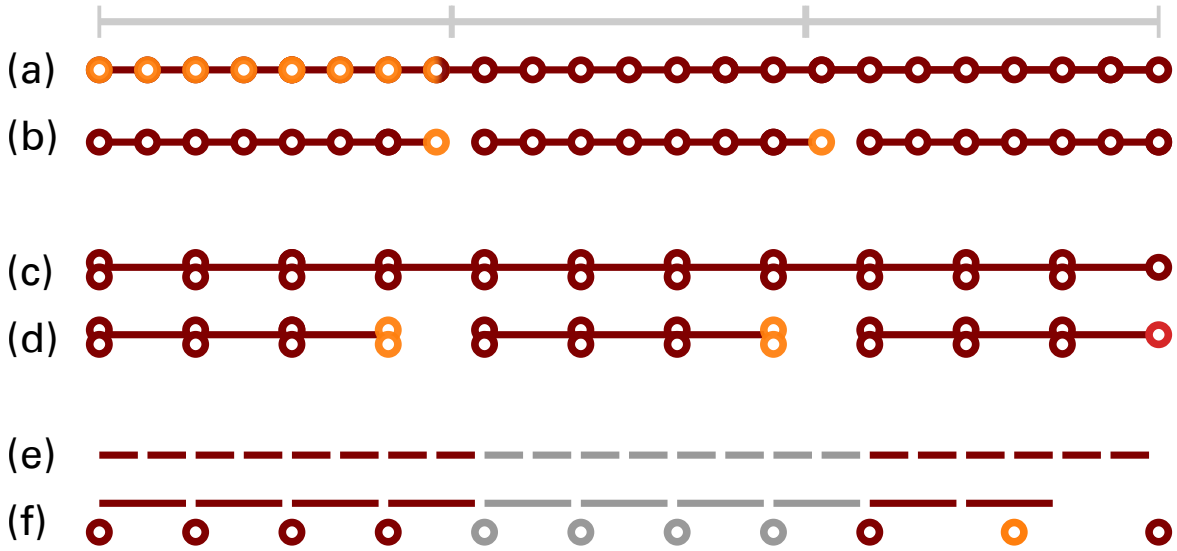


Figure 1.24: Visualization of the steps of the partitioning algorithms given by Algorithms 2 and 3 that yield the partitioning shown in Fig. 1.23.

Algorithm 2 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
```

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. 1.24 (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. 2 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. 1.24 (c) shows this grouping. Because the considered example has an odd total number of 23 nodes, one 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. 2 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. 1.24 (d), the subdomains obtain either $\alpha = 6$ or α+ `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.

1.6.4 Algorithm for Sampling Points from the Fine 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 3 lists the function that determines the number of sampled points in a given subdomain. Similar to Alg. 2, the input is a 1D subdomain coordinate. The output is the number of sampled points in this subdomain.

Algorithm 3 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. 3 calls Alg. 2 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. 1.23 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. 3 decrement the value of `n` to yield the correct number of elements.

The corresponding visualization in Fig. 1.24 (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. 1.24 (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. 1.24 (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. 1.24 (f), this case occurs in the last subdomain. The orange node is sampled at the middle between the two neighboring dark red nodes. This behaviour can also be observed in the corresponding partitioning in Fig. 1.23 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.

1.6.5 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 that each subdomain obtains the same number of fibers (ii) and that the least possible number of remainder elements is generated (iii) can be adjusted. By default, we set the granularity parameters to the same value as the sampling parameters and additionally ensure for quadratic meshes that the granularities are a multiple of two. This setting typically yields partitionings with equally sized elements, however the number of fibers per subdomain is not always optimal.

To allow users to enforce a partitioning, where every rank gets the exact same number of fibers, except for the last subdomains in each coordinate direction, which potentially gets one layer of fibers 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.

1.6.6 Results

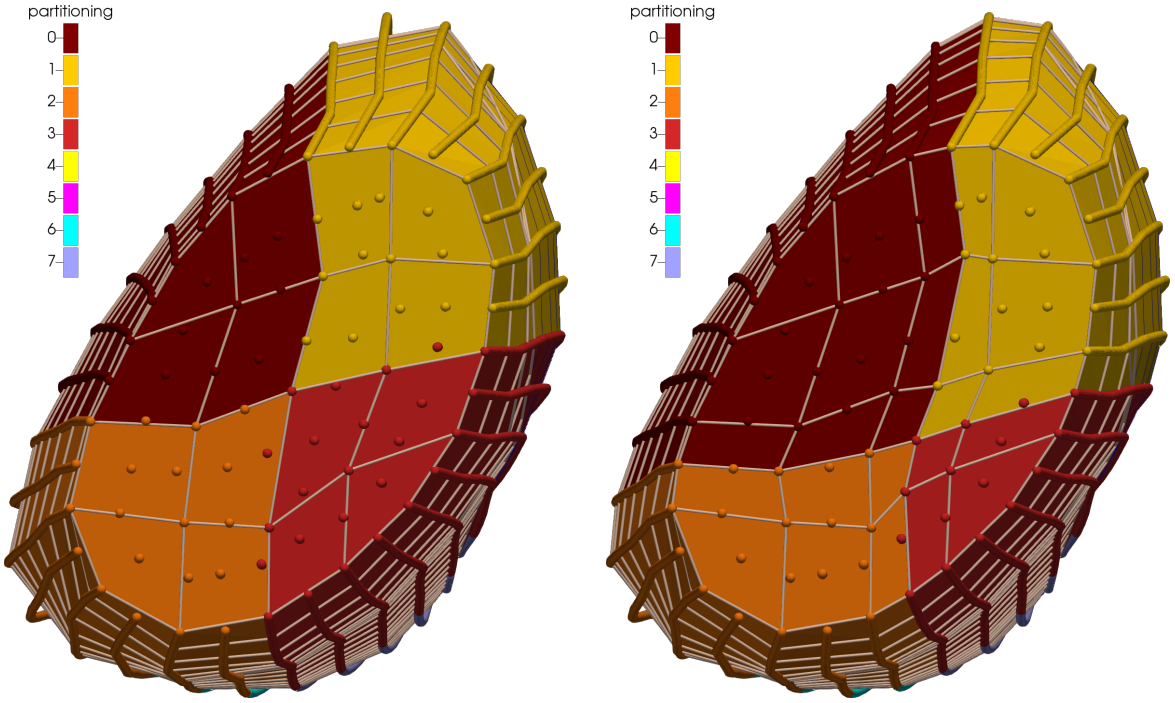
The different results are demonstrated in Figures 1.25 and 1.26. Figure 1.25 shows the automatic partitioning, where a simulation of fiber based electrophysiology with a grid of 9×9 fibers is executed with eight processes and the stride values `sampling_stride_x` and `sampling_stride_y` are set to two. 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. 1.25a. Only the first four subdomains can be seen in the visualization, the other four are located behind.

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 1.25b 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. In 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. 1.25a, except for a different order. However, for larger partitionings 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. 1.25a and Fig. 1.25b in terms of load balancing, the 3D mesh quality of the generated partitioning is worse for Fig. 1.25b. As can be seen in Fig. 1.25b, the first and third subdomains 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. 1.26, 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 1.26a and 1.26b show the resulting partitionings without and with the `distribute_nodes_equally` option.



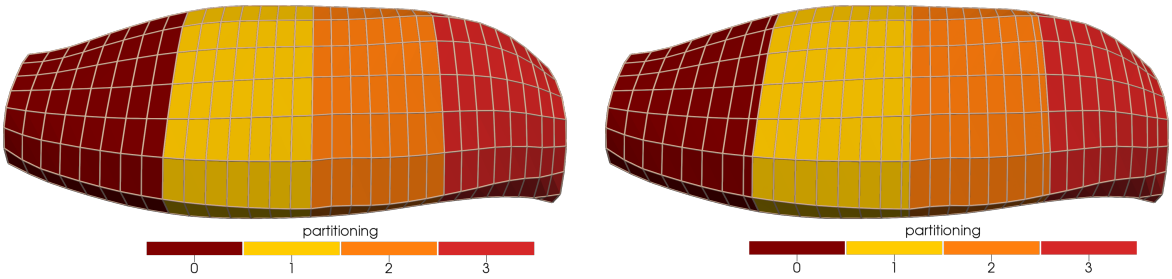
(a) Resulting sampled mesh with option `distribute_nodes_equally=False`.

(b) Resulting sampled mesh with option `distribute_nodes_equally=True`.

Figure 1.25: 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.

Again, the second scenario shows "remainder" elements with smaller mesh widths at the end of every subdomain. The distribution of nodes is 400, 350, 350 and 381 nodes per subdomain in Fig. 1.26a and 371, 370, 370 and 370 nodes per subdomain for the scenario in Fig. 1.26b 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 that impose high computational loads because of the subcellular model, the load balancing aspect is more important and the option `distribute_nodes_equally` should be set to `False`. 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 `True`.



(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 1.26: Sampling a mesh along the fiber direction. The original mesh has 1481 nodes and is sampled with a stride value of 50.

How To Reproduce

The partitioning in Fig. 1.23 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 1.25 and 1.26 are created by the following simulations. For Figures 1.25a and 1.26a, edit the variables file `partitioning_demo.py` and set `distribute_nodes_equally = False`. For Figures 1.25b and 1.26b, 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
```

1.7 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. Section 1.7.1 begins with a description of the solver structure and the parallelization. Subsequently, performance improvements considering the parallel execution of the solver are discussed. Section 1.7.2 presents a variant, where a faster solver is employed for the 1D part of the computation. Section 1.7.3 shows how the computational load can be reduced by only computing activated parts of the muscle.

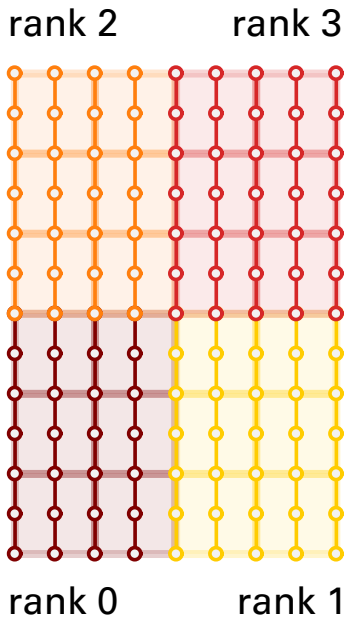
1.7.1 Parallel Solver Structure

For better visualization, we consider the 2D setting of a mesh and embedded 1D fibers partitioned to 2×2 processes as shown in Fig. 1.27a by different colors. However, all discussions are also valid for the real 3D setting shown in the last section and for abitrary partitionings to $n_x \times n_y \times n_z$ processes.

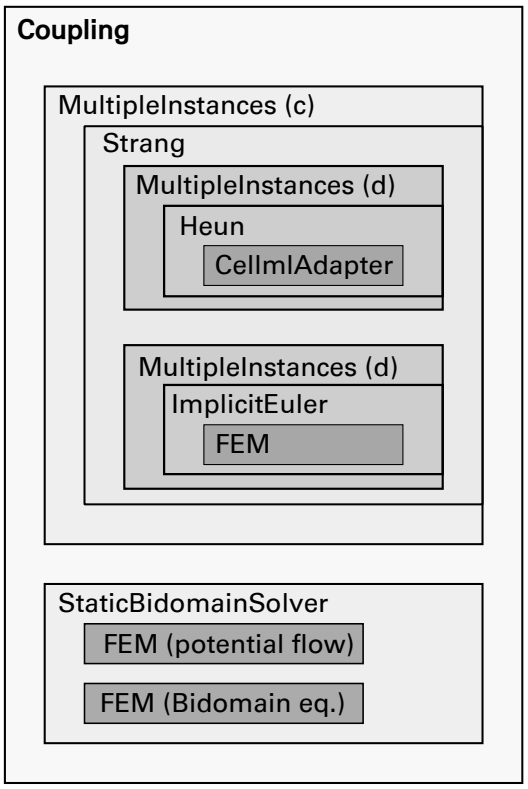
Figure 1.27b shows the program structure of the example that solves the fiber based electrophysiology model. The outer class is a `Coupling` that alternates between computing the monodomain equation on the 1D fibers and computing the static bidomain equation on the 3D domain. The second part, the bidomain solver, is given in Fig. 1.27b 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. 1.27b consists of a tree of a `Multiple Instances` class that 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`.

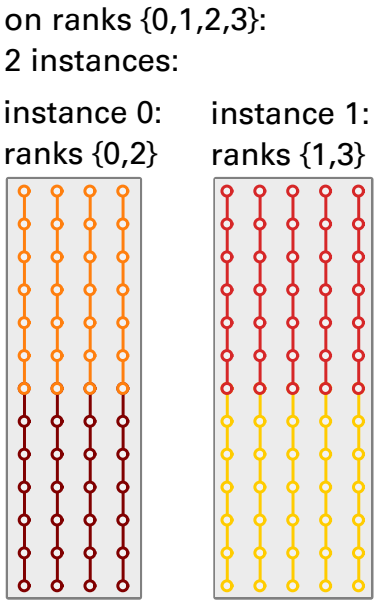
A `MultipleInstances` class can be used to apply a solver to more than one problem of the same kind. The class allows to specify a number of instances of its nested solver. Each instance can be given a subset of processes that will take part in the computation of



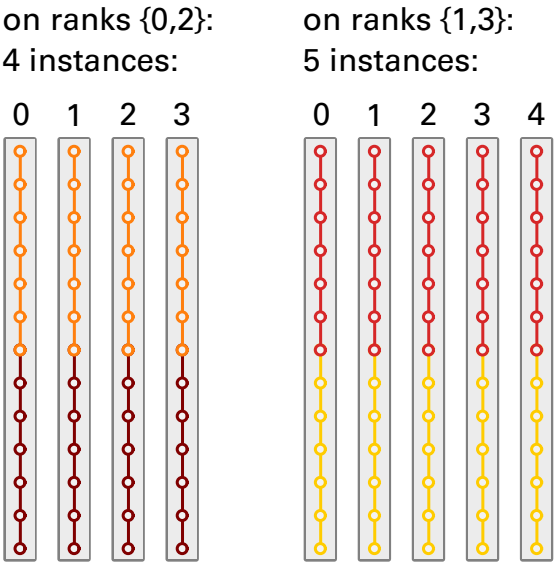
(a) Visualization of the 3D mesh with embedded 1D fibers, partitioned to four ranks.



(b) Structure of the OpenDiHu example program to solve the fiber based electrophysiology model.



(c) Instances of the outer `MultipleInstances` class in Fig. 1.27b.



(d) Instances of the inner `MultipleInstances` classes in Fig. 1.27b.

Figure 1.27: Visualizations for the discussion of the program structure and partitioning used for fiber based electrophysiology simulations.

the instance. Each process then iterates over all instances where 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 and it is called in parallel and independently for instances that have disjoint subsets of ranks.

Furthermore, the class provides a common output writer that collectively writes the data of all instances. This allows, e.g., to create a single output file in every timestep containing the data of all fibers. Especially for large scenarios, this is more practical than having as many output files as there are 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. 1.27b are shown in Figures 1.27c and 1.27d, respectively. As can be seen in Fig. 1.27c, 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. 1.27a, the subdomains of ranks 0 and 2 can be computed independently of the subdomains of ranks 1 and 3. In consequence, all processes specify that their `MultipleInstances` class has two instances. At rank 0, the list of instance settings contains in the first list item the settings of the nested Strang solver with all information of rank 0's subdomain and in the second list item the value `None`, as rank 0 has no information about fibers outside of its subdomain. Ranks 1, 2 and 3 specify their subdomain accordingly, as shown in Fig. 1.27c.

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 1D electric conduction solvers on multiple fibers. As shown in Fig. 1.27d, 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, $n_x \cdot n_y$.

1.7.2 Improved Parallel Solver Scheme using the Thomas Algorithm

The monodomain model that is solved on each fiber consists of a reaction-diffusion equation that is solved using the Strang operator splitting. The diffusion part uses an implicit timestepping scheme, which leads to a linear system of equation 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. 1.27b, this solution step occurs in the solvers under the second inner `MultipleInstances` class. As can be seen in Fig. 1.27d, 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 elemination 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 taylored to the parallel solution of fiber based electrophysiology using the Thomas algorithm. The parallel partitioning of the fibers is carried out normally, as described in

Sec. 1.7. Before the computation, the fiber data in Fig. 1.27d are communicated such that every fiber is completely present 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. In result, every process has approximately the same number of complete fibers. 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.

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. 1.27a. 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.

During initialization, the `FastMonodomainSolver` class initializes its nested solver tree as normal. At the beginning of the first timestep, the communication to gather complete fiber data on single processes is carried out. Then, the monodomain equation is solved 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 is communicated back to the original partitioned fibers. 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 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 $dt_{3D} = 0.5\,\text{ms}$. The timestep widths dt_{0D} of the subcellular model and dt_{1D} of the diffusion term have to be set at maximum to $1 \cdot 10^{-3}\,\text{ms}$, yielding 500 timesteps of computations on the fiber between subsequent communication steps. As a result, the communication cost is negligible.

1.7.3 Adaptive Computation of the Subcellular Model

During simulations of the fiber based electrophysiology model, often only a small fraction of the given fibers are activated. The reason is that in physiological conditions the smaller 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 if 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 $1 \cdot 10^{-5}$, 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 one 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. 1.28.

An active point stays active, if the solution has changed in the last numeric 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 time step. An inactive state cannot be activated by a check on the point itself, as this state implies that no computation and 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”. Then, point A changes to “neighbor is 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 1.29 shows a simulation where the effect of both improvements are 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

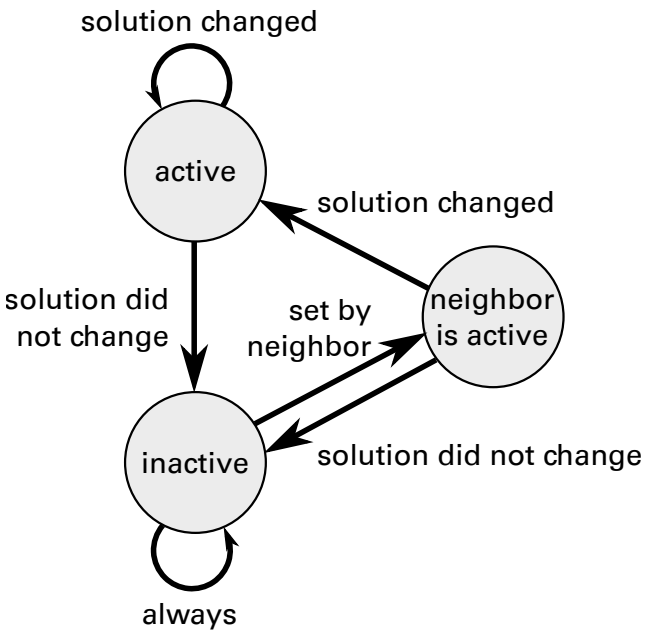


Figure 1.28: Transition diagram for the adaptive computation of the subcellular model.

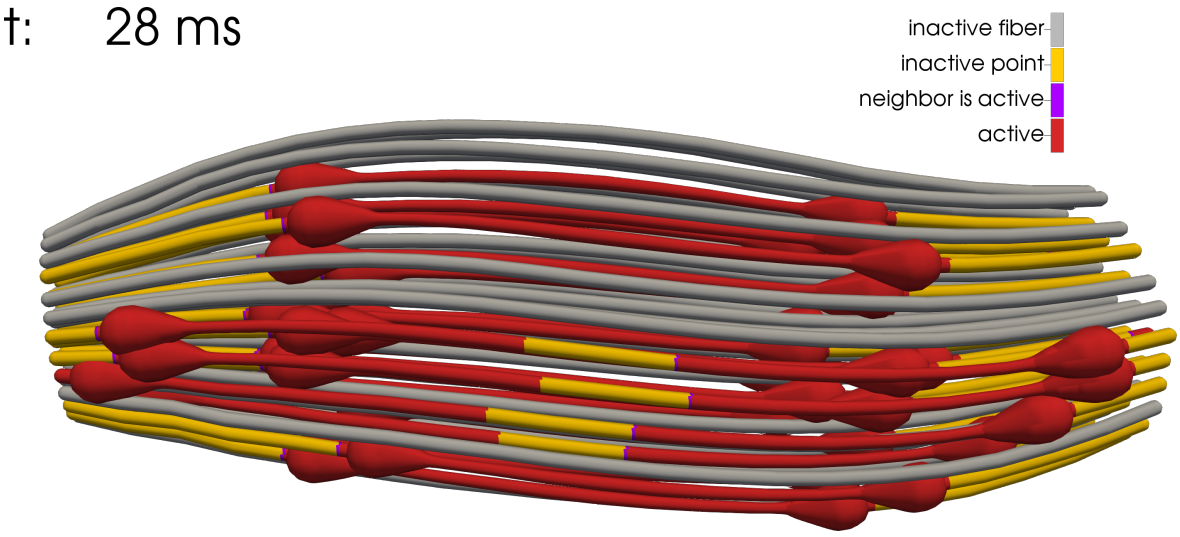


Figure 1.29: Simulation scenario that demonstrates the adaptive computation method of fibers and subcellular points. A simulation of the monodomain equation on a set of 49 fibers with the subcellular model of Hodgkin and Huxley is shown. The transmembrane potential is visualized by the fiber radius. 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. 1.29 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.

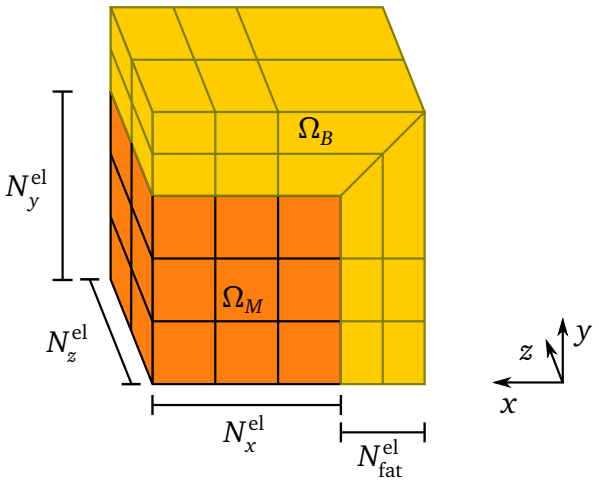


Figure 1.30: Layout of the composite 3D mesh for the multidomain model with fat layer. The orange elements belong to the mesh of the muscle domain Ω_M, the yellow elements are added on top to represent the body domain Ω_B.

1.8 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 1.6 and 1.7, 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. 1.2.4.

1.8.1 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. 1.4.7. Figure 1.30 shows the layout 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 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 in **??**. 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 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`.)

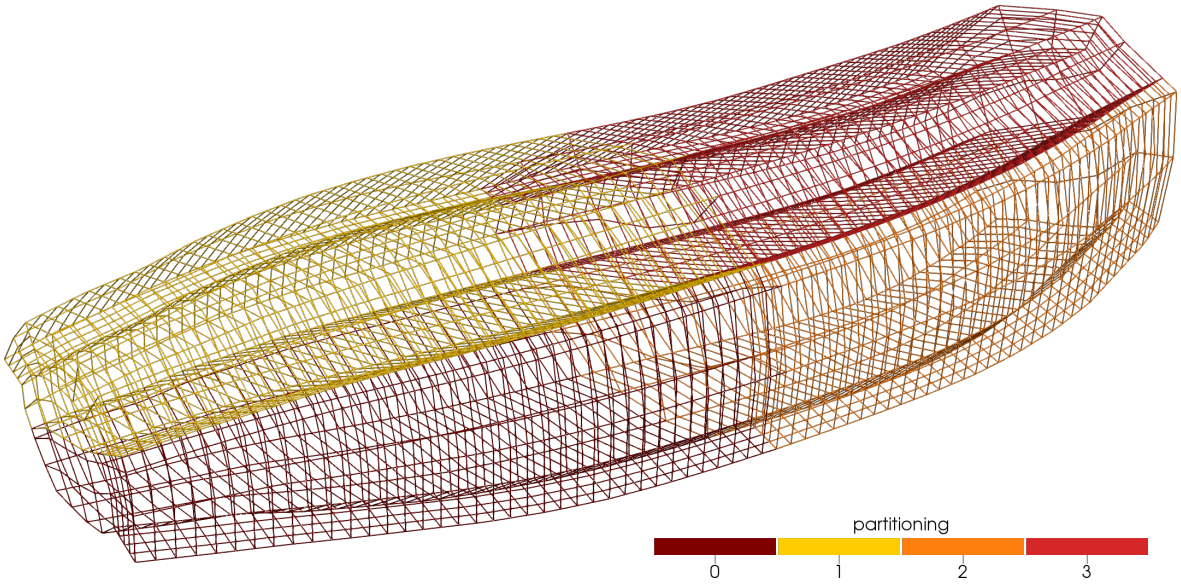


Figure 1.31: Composite mesh of the multidomain example, partitioned to four processes.

Figure 1.31 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. 1.6.1 with stride values of 3, 3 and 20 in x, y and z directions and `distribute_nodes_equally=True`. In 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, yielding a

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 to the same number of processes. If, e.g., a partitioning of $n_x = n_y = 2$ is chosen, the cube in Fig. 1.30 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. 1.31 is partitioned to $2 \times 1 \times 2$ subdomains as shown by the different colors.

1.8.2 Structure of the System Matrix

The `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 **??** in **??**. The solver calculates the matrix block entries using the stiffness and mass matrices computed by the nested `FiniteElementMethod` classes.

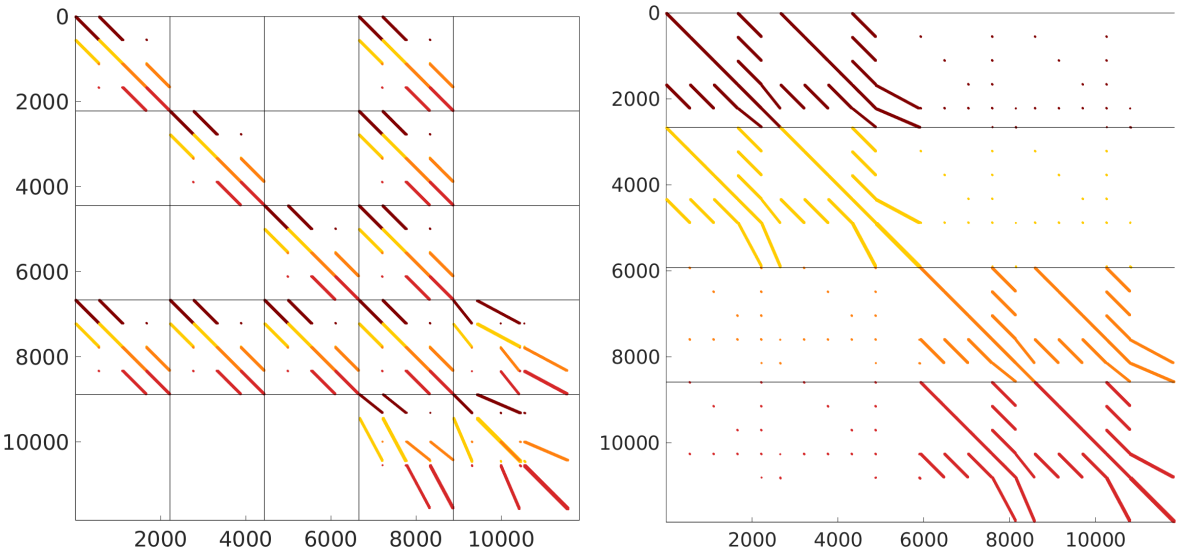
Figure 1.32a shows the location of non-zeros in the resulting sparse matrix for three MUs. The matrix blocks are indicated by boxes and can be identified by comparison with **??**. In this visualization, it may seem that most of the blocks only have three non-zero entries per row, however, the actual number is higher 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. 1.18.

The colors in Fig. 1.32a correspond to the four processes, as defined in the partitioned mesh in Fig. 1.31. The entries in every block are all partitioned in the same way to the 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 that 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, in consequence, the matrix entries. The same permutation is applied to the rows and the columns of the matrix. The result of this operation is shown in Fig. 1.32b. 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.

1.8.3 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



(a) Original matrix layout (b) Reordered matrix layout

Figure 1.32: Nonzero structure of the system matrix of the multidomain problem.

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 1.33 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. 1.33 shows a large difference between the largest and the smallest eigenvalues. The condition number of A can be computed by $\text{cond}(A) = |\lambda_{\max}|/|\lambda_{\min}| = 161.2576/0.0116 \approx 1.4 \cdot 10^5$. Thus, the problem is ill-conditioned and can benefit from for preconditioning. The condition number is also dependend 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. Figure 1.34a shows the non-zero structure of the resulting matrix. As all diagonal blocks are symmetric matrices, the resulting matrix A' is also symmetric in contrast to A. The permutation of rows and columns to obtain the parallel data layout leads to the matrix structure shown in Fig. 1.34b. The permutated matrix still contains only

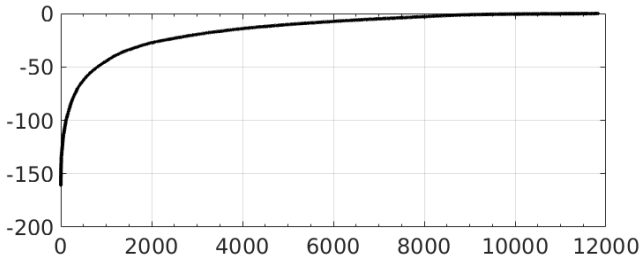
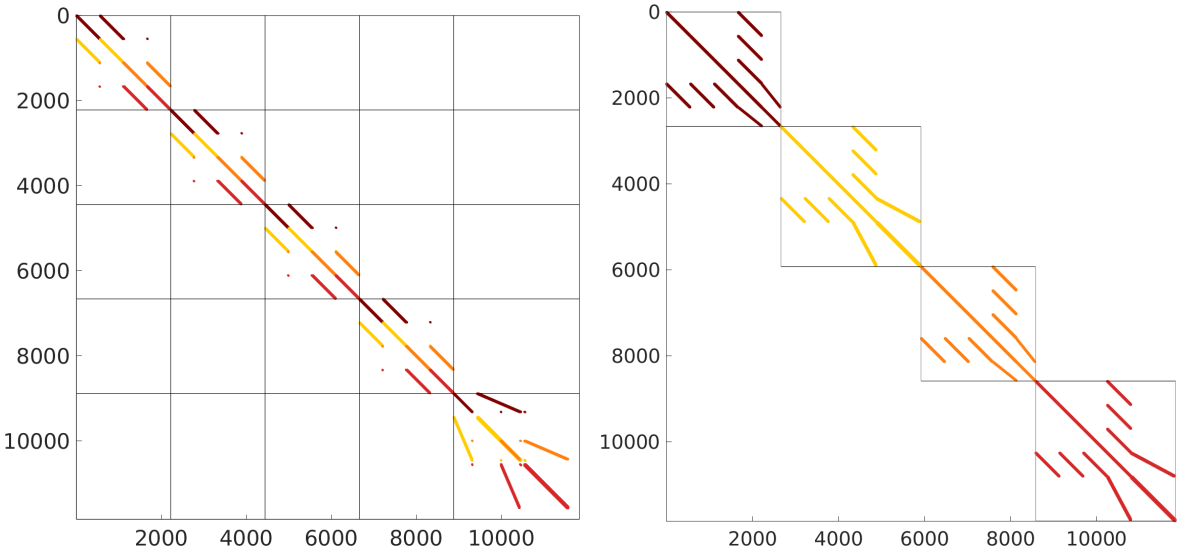


Figure 1.33: Real parts of the eigenvalues sorted by magnitude and corresponding to the example in Fig. 1.31. The non-zero eigenvalue with largest and smallest absolute values are $\lambda_{\text{max}} = -161.2576$ and $\lambda_{\text{min}} = -0.0116$.



(a) Original matrix layout

(b) Reordered matrix layout

Figure 1.34: Nonzero structure of the symmetric preconditioner matrix of the multidomain problem.

diagonal block, because entries in the original diagonal blocks are never permutated out of this block only in their row or column but are always moved by the same permutation in both rows and columns. Figure 1.34b shows that the entries on every rank are now decoupled, which potentially allows for a faster computation in the application of the preconditioner.

1.8.4 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. 1.31 to 16 processes. The resulting partitioning of the mesh is given in Fig. 1.35. Figure 1.36 shows the non-zero structure of the system matrix

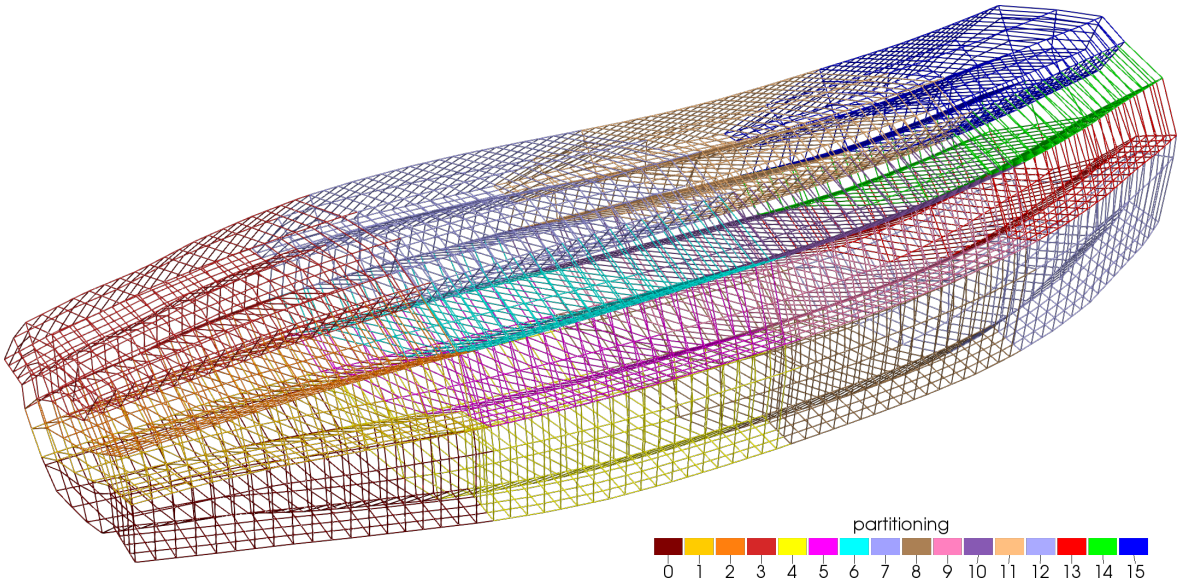
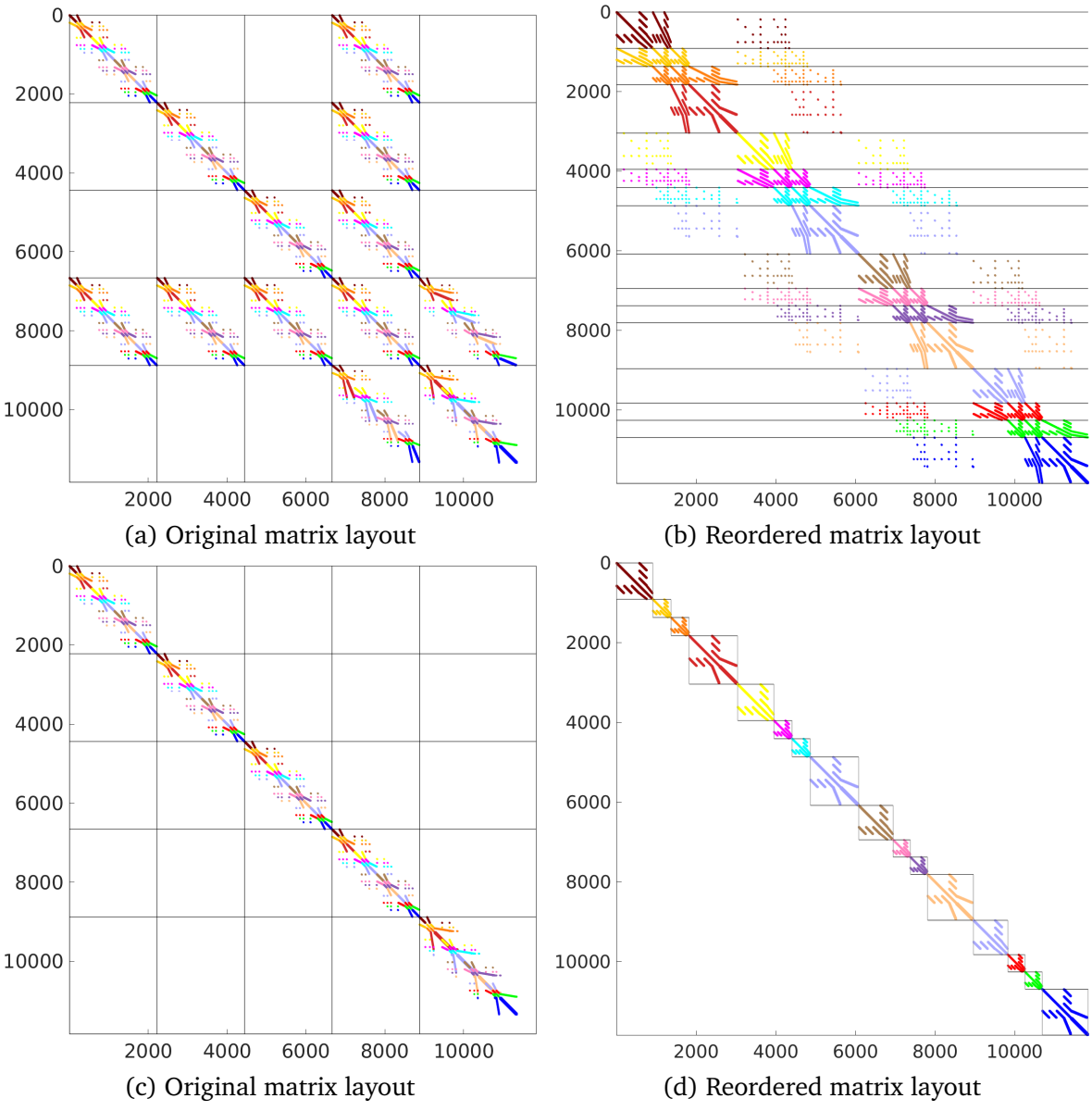


Figure 1.35: Partitioning to 16 processes of the mesh in the multidomain example.

and the diagonal matrices for the preconditioner. Figure 1.36a contains the original matrix structure that is permutated to the structure in Fig. 1.36b. Reordering only the diagonal blocks of the original matrix in Fig. 1.36c leads to the structure in Fig. 1.36d. A comparison with Fig. 1.34 shows that the width of the non-zero band decreases for higher parallelizations.



(a) Original matrix layout

(b) Reordered matrix layout

(c) Original matrix layout

(d) Reordered matrix layout

Figure 1.36: Nonzero structure of the symmetric preconditioner matrix of the multidomain problem partitioned to 16 processes.

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 1.32b and 1.36b. The MATLAB script reverses the permutation that was applied in OpenDiHu to generate the plots of Figures 1.32a and 1.36a.

1.9 CellML Adapter

1.10 Solid Mechanics Solver

1.11 Mapping Between Meshes

1.12 Output Writers

Bibliography

[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, eprint: https://doi.org/10.1137/S0895479899358194, https://doi.org/10.1137/S0895479899358194

[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

[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

[Bal97a] **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

[Bal97b] **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

[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

[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

[Fal02] **Falgout**, R. D.; **Yang**, U. M.: *Hypre: a library of high performance preconditioners*, International Conference on Computational Science, Springer, 2002, pp. 632–641

[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

[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

[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

[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

[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

[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

[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

[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

[Kis20] **Kislan**, T.: *Base64 encoding and decoding for c++ projects*, https://github.com/tkislan/base64, 2020

[Kre12a] **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, eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.1149, https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.1149

[Kre12b] **Kretz**, M.; **Lindenstruth**, V.: *Vc: a c++ library for explicit vectorization*, Software: Practice and Experience 42.11, 2012, pp. 1409–1430, doi:10.1002/spe.1149, eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.1149, https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.1149

[Kre15] **Kretz**, M.: *Extending C++ for explicit data-parallel programming via SIMD vector types*, PhD thesis, Frankfurt am Main: Goethe University Frankfurt, 2015, https://publikationen.ub.uni-frankfurt.de/frontdoor/index/index/docId/38415

[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

[Mai21] **Maier**, B.: *OpenDiHu Online Documentation*, https://opendihu.readthedocs.io/, 2021

[Röh17] **Röhrle**, O.: *DiHu - Towards a digital human, Project Website*, 2017, https://ipvs.informatik.uni-stuttgart.de/SGS/digital_human/index.php

[Ser21] **Services**, A. W.: *Easylogging++ – Single header C++ logging library*, https://github.com/amrayn/easyloggingpp, 2021

[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

[Van09] **Van Rossum**, G.; **Drake**, F. L.: *Python 3 Reference Manual*, Scotts Valley, CA: CreateSpace, 2009, isbn:1441412697

[Vei21] **Veillard**, D.: *The XML C parser and toolkit of Gnome*, http://xmlsoft.org/, 2021

[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

[XBr20] **XBraid**, T.: *XBraid: parallel multigrid in time*, http://llnl.gov/casc/xbraid, 2020