

# Contents

<b>1</b>	<b>Implementation</b>	<b>3</b>
1.1	Introduction	3
1.1.1	Design Goals of OpenDiHu	4
1.1.2	Overview of this Chapter	5
1.2	Data Handling with PETSc	5
1.2.1	Organization of Parallel Partitioned Data	6
1.2.2	Numbering Schemes for Nodes and Degrees of Freedom	10
1.2.3	Parallel Data Structures in OpenDiHu	11
1.2.4	Discussion of Several Design Decisions	14
1.2.5	Implemented Basis Functions	16
1.2.6	Implemented Types of Meshes	18
1.2.7	Composite Meshes	20
1.3	Finite Element Matrices and Boundary Conditions	22
1.3.1	Storage of Matrices	22
1.3.2	Assembly of Finite Element Matrices	27
1.3.3	Performance of the Algorithm for Parallel Matrix Assembly	29
1.3.4	Assembly of Finite Element Matrices for Regular Meshes	32
1.3.5	Algorithm for Dirichlet Boundary Conditions	34



# 1 Implementation

The simulation of complex multi-scale models often requires the combination of tailored numerical solution schemes. Spatial mesh resolutions and time step widths have to be chosen carefully to avoid instabilities and at the same time allow to compute a sufficiently long simulation time span in a feasible runtime. Meshes with different dimensionalities have to be combined, data mapping between the meshes can be 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 such as convenient debugging as well as efficient storage and visualization of large datasets in production runs. The configuration of parameters of models and solvers has to be organised and documented in a convenient way.

In the following, we present details on our implementation of the software *OpenDiHu*, which aims to fulfill the mentioned requirements. It is used to simulate various biophysical problems describing the biomechanics and neurophysiology of the musculoskeletal system.

## 1.1 Introduction

The previously mentioned requirements are imposed from a user's perspective. Additional requirements follow from a developer's perspective. The program code should be modular and well documented to allow reuse and extension in the future. The implemented functionality should function correctly and the correctness should be testable in order to be preserved during code changes.

With these demands in mind we develop our software framework named *OpenDiHu*. In the different contexts of reporting and software development, the presented capitalized name and an all lower case version of the name are used, respectively. The name originates from the Digital Human project that advanced the field of biomechanics by “providing

new possibilities to improve the understanding of the neuromuscular system by switching from small-sized cluster model problems to realistic simulations on HPC clusters” [Röh17]. The software has been introduced in a publication [Mai19].

### 1.1.1 Design Goals of OpenDiHu

OpenDiHu is an Open Source software framework that solves static and dynamic multi-physics problems using the Finite Element Method. It is a simulation framework targeted at multi-scale skeletal muscle models for simulating EMG signals and muscle contraction. The design goals can be summarized under the keywords *usability*, *performance* and *extensibility*. They span the mentioned field of requirements from user to developer-centric properties.

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 are 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 are scientific and educational studies. The program can be run in serial or in parallel on a small-scale compute server or compute cluster or on a supercomputer.

We restrict usage of the framework to using command line tools 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. The advantage of a command line only program is that it can be easily used in automated studies with different parameter combination and it simplifies usage on remote computers such as compute servers and supercomputers.

In this context, good usability is ensured by using the Python programming language for the configuration of the simulation. The core code of every simulation is written in C++, in order to achieve good performance. The user can configure all parameters using Python scripting. The Python3 interpreter is linked to the C++ program and the configuration script of the simulation run, also called *Python settings* gets parsed by this interpreter

initially at runtime of the simulation. Thus, the user is able to use clear variables for parameters, compute derived parameter values, organize the settings in multiple files, and define own command line parameters for every example. Input and results of the simulation can be preprocessed and postprocessed directly in the Python settings script.

The next design goal of good performance is satisfied by supporting parallel execution and structured meshes. Efficient algorithms are implemented. The data handling avoids expensive copy operations whenever possible. Instruction level parallelism is employed. Highly optimized solvers are imported from external libraries.

Extensibility refers to the possibility to implement new solvers for new physical processes in the existing framework. By properly abstracting concepts in the code by object orientation and templates, structures like meshes, function spaces and solvers can be reused. For example, the software framework provides infrastructure to store and manipulate scalar and vector fields, assemble system matrices, use various timestepping schemes, and handle input and output in various file formats. Open standards like CellML for subcellular or neuron models can be accessed. The coupling library *PreCICE* can be used to combine the simulation with external solvers.

### 1.1.2 Overview of this Chapter

All the presented properties are discussed in more detail in the remainder of this chapter. Details are given on the implementation, the underlying algorithms, and the application from a user's perspective. For selected topics, a comparison to other existing simulation frameworks for biomechanical problems such as OpenCMISS is given.

[Section 1.2](#) presents the underlying data structures to handle vectors and meshes. [Section 1.3](#) visits the assembly of Finite Element matrices and an algorithm for Dirichlet boundary conditions. (...)

## 1.2 Data Handling with PETSc

Software for parallel Finite Element simulations processes various types of values: geometry data, the computed quantities, and 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 the same regardless of the number of processes that execute the program.

For parallel data handling and solvers of linear and nonlinear systems, the *Portable, Extensible Toolkit for Scientific Computation (PETSc)* [Bal16]; [Bal15]; [Bal97] is used. PETSc provides a large collection of solvers and preconditioners that can be selected and configured at runtime. More solvers are accessible through interfaces to external software, such as the *Multifrontal Massively Parallel Sparse Direct Solver (MUMPS)* [Ame01]; [Ame19] and the preconditioner library *HYPRE* [Fal02]. PETSc natively supports MPI parallelism and provides parallel data structures for vectors and matrices. Numerous operations on the data are provided including value communication and access, house-keeping, 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*. The rank of a process is a consecutive number starting with zero, the MPI communicator is a subset of processes that can communicate with each other.

### 1.2.1 Organization of Parallel Partitioned Data

Basic building blocks in the implementation of OpenDiHu are *field variables* that represent scalar fields. A scalar field  $v : \Omega \rightarrow \mathbb{R}$  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  $\Omega$ , 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)* values. 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  $N_i^{\text{el}}$  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

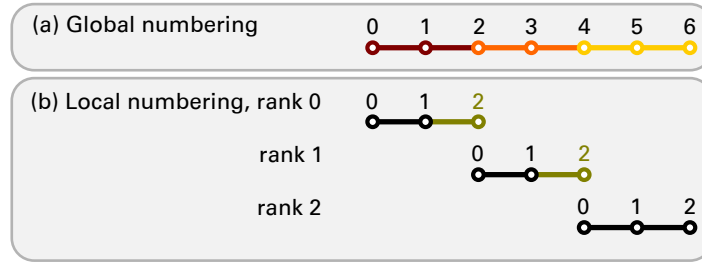


Figure 1.1: Partitioning and local and global numberings of a 1D mesh with  $N_x^{\text{el}} = 6$  elements partitioned to  $n_x = 3$  processes.

with smaller index in  $x$ ,  $y$  and  $z$  direction potentially have one layer of elements more than other subdomains.

For example, in Fig. 1.1 a 1D mesh with  $N_x^{\text{el}} = 6$  elements is partitioned into three subdomains with two elements each. Figure 1.2 (a) shows a 2D mesh with  $N_x^{\text{el}} \times N_y^{\text{el}} = 5 \times 4$  elements, a partitioning to  $n_x \times n_y = 2 \times 3$  processes is given in Fig. 1.2 (b).

The nodes of the mesh are assigned to the same subdomains as their adjacent elements. The assignment of the nodes that lie on the cutting planes between the subdomains 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  $N_x^{\text{el}}, N_y^{\text{el}}, N_z^{\text{el}} \rightarrow \infty$  the imbalance vanishes totally.

In the exemplary partitionings in Fig. 1.1 (b) and Fig. 1.2 (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. 1.1, the three subdomains with two elements each have three, two and two nodes. In the 2D example in Fig. 1.2 the six subdomains have either three (ranks 2 and 3) or six (ranks 0,1,4 and 5) nodes while the number of elements varies between six (rank 0) and two (rank 5). This demonstrates the construction of nearly equally sized subdomains in terms of 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

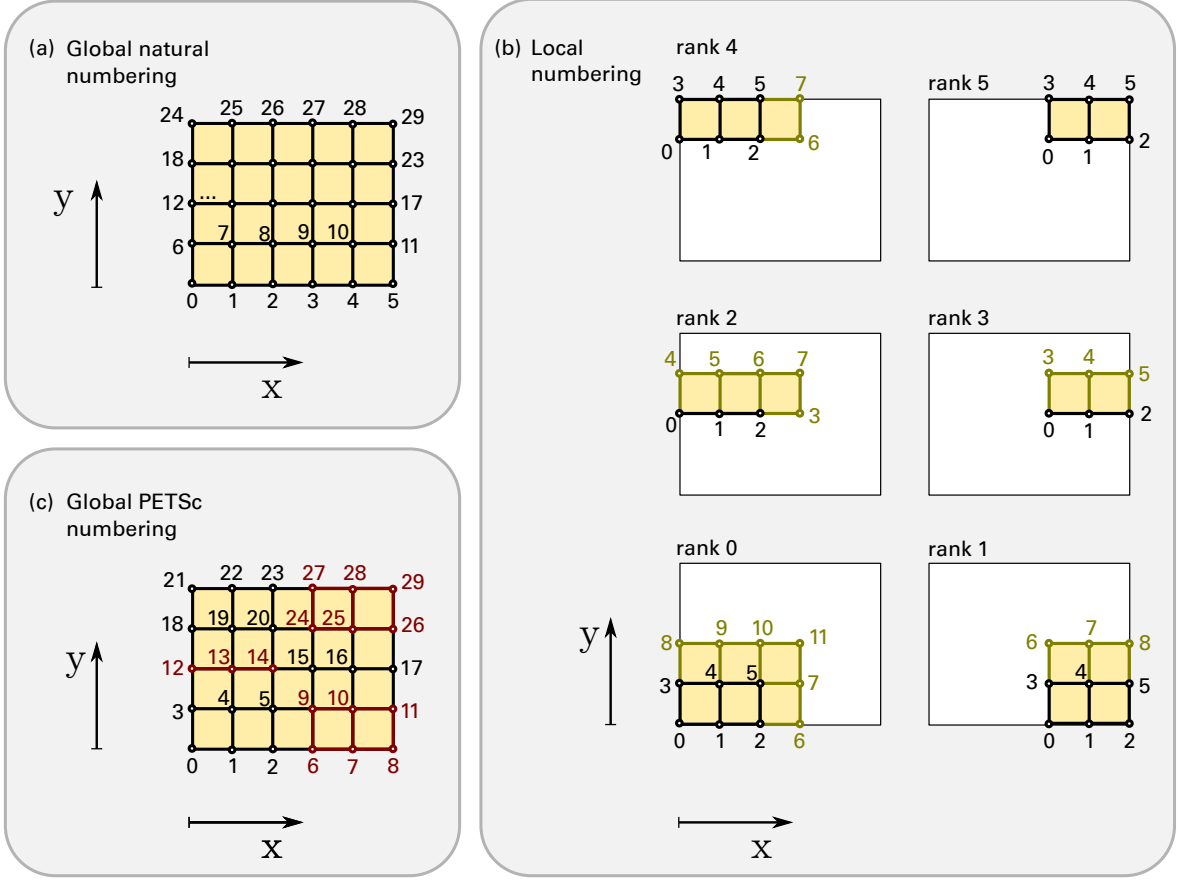


Figure 1.2: Subdomains and numberings of a 2D mesh with  $N_x^{\text{el}} \times N_y^{\text{el}} = 5 \times 4$  elements partitioned to  $n_x \times n_y = 2 \times 3$  processes. (a)-(c) show the different numberings needed for (a) boundary condition specification, (b) identification of local non-ghost dofs (black) and local ghost dofs (yellow), and (c) identification of global dofs.

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  $n_{\text{global}}$  values. The vector is distributed to  $n_{\text{proc}}$  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  $n_{\text{ghosts}}$  ghost dofs that are owned by other ranks. PETSc is able to communicate corresponding values between all ranks where the dof is present either as ghost or non-ghost dof.

In total, the local buffer of a **Vec** stores  $n_{\text{local\_with\_ghosts}} = n_{\text{local\_without\_ghosts}} + n_{\text{ghosts}}$  values. The non-ghost dofs are located at array positions  $0, \dots, n_{\text{local\_without\_ghosts}} - 1$ , the ghost



dofs follow at positions  $n_{\text{local\_without\_ghosts}}, \dots, n_{\text{local\_with\_ghosts}} - 1$ . This array is consecutive in memory. The latter part for the ghost dofs is called the *ghost buffer*.

The local dofs in every subdomain are numbered according to the layout of this buffer. Figure 1.1 (b) shows the local dof numbering on the three ranks. It proceeds through all non-ghost dofs followed by the ghost dofs. A global numbering of all dofs is given in Figure 1.1 (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 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 1.3a and Fig. 1.3b visualize the data flow in the two operations.

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.

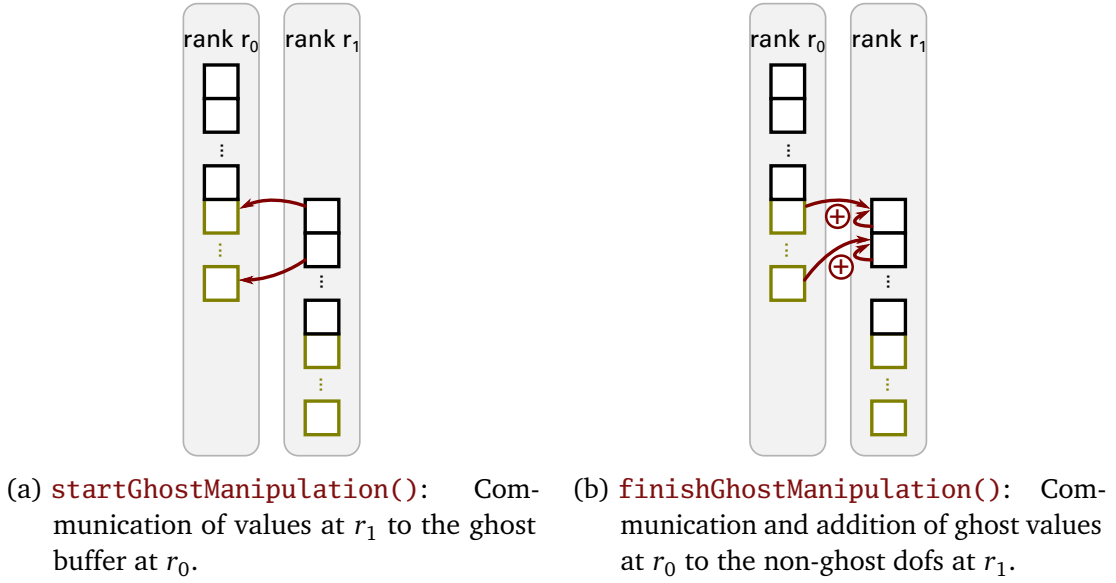


Figure 1.3: Communication operations for ghost values in an example with two ranks  $r_0$  and  $r_1$ . Depicted are the vectors of local storage for non-ghost (black) and ghost values (yellow). The red arrows indicate the data transfer.

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()`.

## 1.2.2 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. 1.1 seems natural, for 2D and 3D meshes a more complex ordering of local dofs is needed.

Three different numbering schemes for nodes and dofs exist within OpenDiHu. They are visualized in Fig. 1.2 for a 2D mesh. The first is the *global natural* numbering scheme, which numbers all  $n_{\text{global}} = N_x^{\text{dofs}} \times N_y^{\text{dofs}} \times N_z^{\text{dofs}}$  global dofs in the structured mesh. It starts with zero and iterates through the mesh using the triple of coordinate indices  $(i, j, k)$  for the  $x$ ,  $y$  and  $z$  axis with the ranges  $i \in \{0, \dots, N_x^{\text{dofs}} - 1\}$ ,  $j \in \{0, \dots, N_y^{\text{dofs}} - 1\}$  and  $k \in \{0, \dots, N_z^{\text{dofs}} - 1\}$ . The numbering proceeds fastest in  $x$  or  $i$  direction, then in  $y$  or  $j$  direction and then in  $z$  or  $k$  direction. Examples are shown in Fig. 1.2 (a) for a 2D mesh and in Fig. 1.4 for a 3D mesh.

The intention of this first numbering is to facilitate the problem description by the user. If values for a variable in the whole computational domain should be specified, the order of the given value list will be interpreted according to this numbering. Boundary conditions can be given for some dofs by simply specifying the corresponding dof numbers in global natural numbering. The advantage is that this numbering scheme is 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. 1.2 (b). It specifies the order of dofs in the local PETSc **Vec** and is defined locally on every subdomain for the non-ghost and ghost dofs. At first, all non-ghost dofs are numbered with the order equal to the one in the global natural scheme. Then, all ghost dofs are numbered, again in the order of the global natural scheme. This numbering has the counter-intuitive property of jumps between some neighboring nodes.

The third numbering scheme is called *global PETSc* numbering and is defined by PETSc. It is the numbering used to access global **Vecs**. It is also the ordering of the rows and columns of matrices. The numbering starts with all local non-ghost numbers on rank 0, then proceeds over all non-ghost numbers of rank 1 and continues like this for all remaining ranks. An example for this numbering is given in Fig. 1.2 (c). The portions of local dofs for the different ranks are indicated by the grid of red and black colors. This numbering depends on the partitioning and, thus, on the number of processes. For serial execution it is identical to the global natural numbering.

### 1.2.3 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

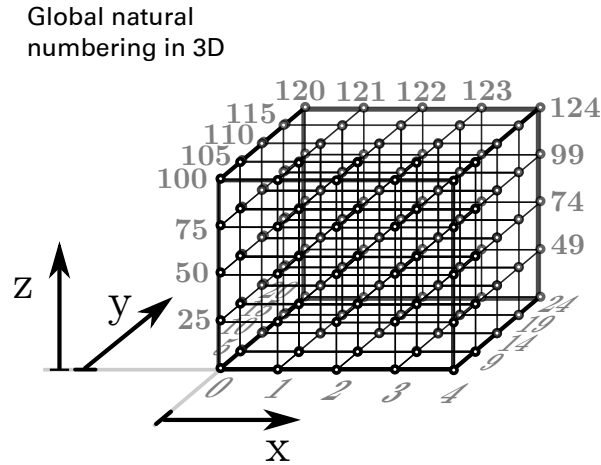


Figure 1.4: Global natural numbering of nodes in a mesh with  $4 \times 4 \times 4$  linear elements.

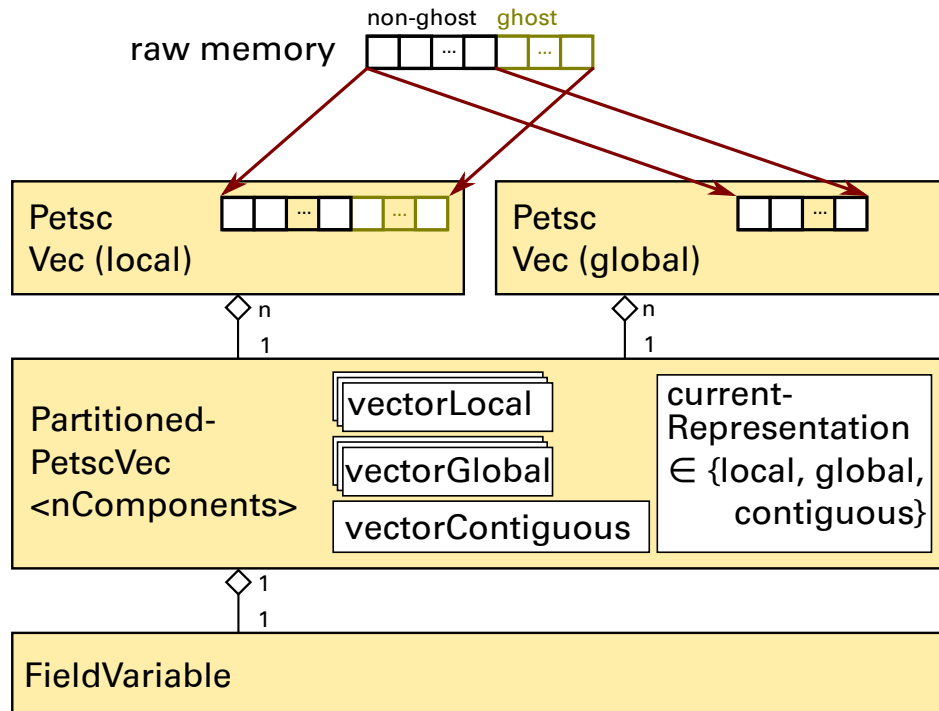


Figure 1.5: Classes in OpenDiHu that represent vectors in parallel execution. The abstraction layer increases from raw memory at the top to the **FieldVariable** at the bottom.

classes are visualized in Fig. 1.5 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. 1.5, these two **Vecs** 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 **Vecs** use the same memory to store their data. However, as shown at the top of Fig. 1.5, the memory range of the local **Vec**'s buffer includes the ghost buffer, which is never accessed by the global **Vec**. As mentioned, PETSc functions are available to switch the valid state between the two **Vec**'s, involving communication of ghost values. Because of the shared memory, no costly value copy operation is needed for this action.

The next abstracting class is *PartitionedPetscVec<nComponents>*. It represents a discretized vector field  $\mathbf{v} : \Omega \rightarrow \mathbb{R}^c$  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 **Vecs** is stored in the variables *vectorLocal* and *vectorGlobal*. The global number of entries in each of the **Vecs** is given by the number  $n_{\text{global}}$  of dofs in the mesh that discretizes the domain  $\Omega$ . 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 occurring often during transient simulations, profiling of simulations has shown negligible runtime for the action of switching between these representations.

The top level class in the value storage hierarchy as shown in Fig. 1.5 is the `FieldVariable`, which contains a `PartitionedPetscVec` and adds numerous methods to facilitate access to the data container. Model formulations use this class to manipulate scalar and vector fields. At the same time, the underlying global PETSc `Vec` can still be obtained from a `FieldVariable`. Vector operations such as addition, norms and matrix-vector products are performed using the low-level PETSc functions on the global `Vec` obtained from the `FieldVariables`.

## 1.2.4 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  $n_{\text{global}}$  dofs of the domain. The same loop is executed in parallel by all  $n_{\text{proc}}$  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  $n_{\text{global}} \rightarrow \infty$ , whereas 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. 1.2.3](#).

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.

### 1.2.5 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. Fur-



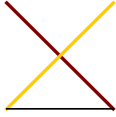

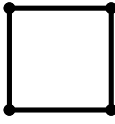
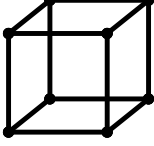


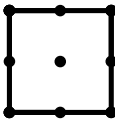
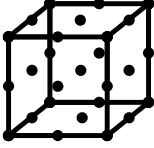


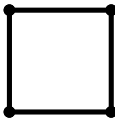
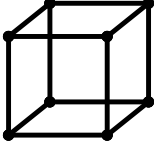
Ansatz functions	Element shapes		
	1D	2D	3D
 $\phi_0(\xi) = 1 - \xi,$ $\phi_1(\xi) = \xi$			
 $\phi_0(\xi) = (2\xi - 1)(\xi - 1),$ $\phi_1(\xi) = 4(\xi - \xi^2),$ $\phi_2(\xi) = 2\xi^2 - \xi$			
 $\phi_0(\xi) = 2\xi^3 - 3\xi^2 + 1,$ $\phi_1(\xi) = \xi(\xi - 1)^2,$ $\phi_2(\xi) = \xi^2(3 - 2\xi),$ $\phi_3(\xi)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

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.

thermore, 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  $f : \Omega \rightarrow \mathbb{R}$  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. 1.2.2](#) have to be adjusted such that at every node all dofs are enumerated in sequence before the numbering continues at the next node.

## 1.2.6 Implemented Types of Meshes

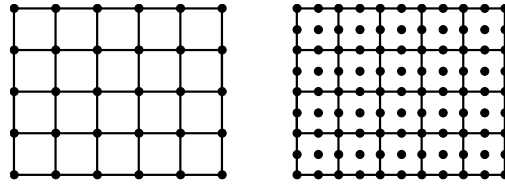
Meshes of different types can be selected independently of the choice of basis functions. Three types are supported. [Figure 1.6](#) visualizes meshes of these types with linear and quadratic elements.

The first type is `Mesh::RegularFixedOfDimension<D>` where  $D \in \{1, 2, 3\}$  is a compile-time constant of the dimension. This type describes a rectilinear, regular structured mesh that is defined by a fixed mesh width  $h$  in all coordinate directions. This mesh is “fixed”, which means that the positions of the nodes cannot change after the mesh object was created. Regular fixed meshes describe a line (1D) 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.

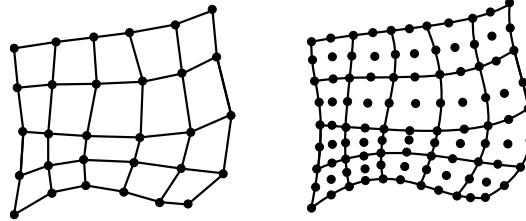
The second mesh type is `Mesh::StructuredDeformableOfDimension<D>`. The structured deformable mesh is a generalization of the regular fixed mesh. The mesh again has a structure of  $N_x^{\text{el}} \times N_y^{\text{el}} \times N_z^{\text{el}}$  elements. Contrary to the regular fixed mesh, the nodes 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 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. 1.6](#).

The third mesh type is `Mesh::UnstructuredDeformableOfDimension<D>`. In contrast to the two other types, this mesh is unstructured implying that element adjacency is no longer given implicitly. The example at the lower third of [Fig. 1.6](#) 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

**Mesh::RegularFixedOfDimension<2>**



**Mesh::StructuredDeformableOfDimension<2>**



**Mesh::UnstructuredDeformableOfDimension<2>**

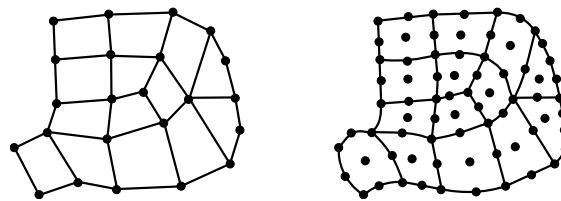


Figure 1.6: The three implemented mesh types in OpenDiHu, each time for 2D linear Lagrange or Hermite ansatz functions (left) and for 2D quadratic Lagrange ansatz functions (right).

possibility to specify these meshes. A pair of **exelem** and **exnode** files, which are common in the OpenCMISS community can be loaded.

A disadvantage of unstructured meshes is that the simple parallel partitioning scheme of subdividing the domain according to element index ranges is not applicable. Instead, the set of elements for every subdomain needs to be computed individually. Typically, this is done using graph partitioning methods in order to minimize subdomain border lengths while ensuring equal subdomain sizes. Another disadvantage is that information about neighbor elements and neighbor subdomains has to be stored explicitly while it is given implicitly in structured meshes. For these reasons, unstructured meshes can be used in OpenDiHu only for serial computation. The construction of parallel partitionings is only possible with the other two, structured mesh types.

The choice which mesh type to use in a simulation has to be made at compile time.

A simulation program can be easily compiled for different meshes by substituting the type in the main C++ source file. By proper abstraction in the code, all implemented algorithms are independent of the used mesh type when run in serial. Some algorithms, e.g., streamline tracing, are specialized for structured meshes to exploit the structure and lead to more efficient code. Unit tests ensure the correct solution of a Laplace problem with all combinations of mesh type, dimensionality and ansatz function.

## 1.2.7 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 1.7a](#) shows a 2D example where three structured meshes are combined to a composite mesh. As can be seen, the submeshes can have different numbers of elements. The nodes on the borders between touching structured meshes are shared between the individual meshes. Thus, these nodes contain only a single set of dofs like every other node in the mesh.

In the code, composite meshes reuse the implementation of structured meshes by defining different numbering schemes for nodes and dofs over the whole composite domain. The numbering of nodes starts with all nodes of the first submesh, then proceeds over all remaining nodes of the second submesh and so on, until all nodes are numbered. The numbering of dofs is analogous. [Figure 1.7b](#) shows an example with two quadratic submeshes with four and two elements. The resulting composite mesh has six elements. The node numbers in the first structured mesh are identical to the corresponding nodes in the composite mesh. The numbering continues in the set of remaining nodes of the second structured mesh and the shared nodes on the border between the meshes are skipped in the numbering, as they already have a number assigned. The shared nodes have the numbers 14, 19 and 24.

In parallel execution, this scheme is executed first on the non-ghost and then on the ghost nodes of the subdomains of all submeshes. Thus, the local numbering of the composite scheme visits the non-ghost nodes of all subdomains first before iterating over the ghost dofs on all subdomains. Thus, the ghost buffer is consecutive in memory as required by the parallel PETSc `Vecs`.

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 1.7a](#) shows a valid partitioning of the exemplary composite mesh to two subdomains.

To configure composite meshes in the settings, their submeshes have to be specified as usual for structured meshes. Then, a list of all submeshes is given for the composite mesh. In parallel execution, a proper partitioning that fulfills the requirements has to be constructed in the Python script of the settings as well.

An application of composite meshes is the biceps muscle with a fat and skin layer. [Figure 1.8](#) visualizes the composite mesh. It consists of two structured submeshes for the muscle belly and the body layer on top, as visualized in the top image. The bottom image shows a partitioning to four processes. As can be seen, the domain can be split along the  $x$  and  $z$  coordinate axes to produce valid partitionings. Using this decomposition strategy, any number of subdomains (limited by the number of elements, though) is possible.

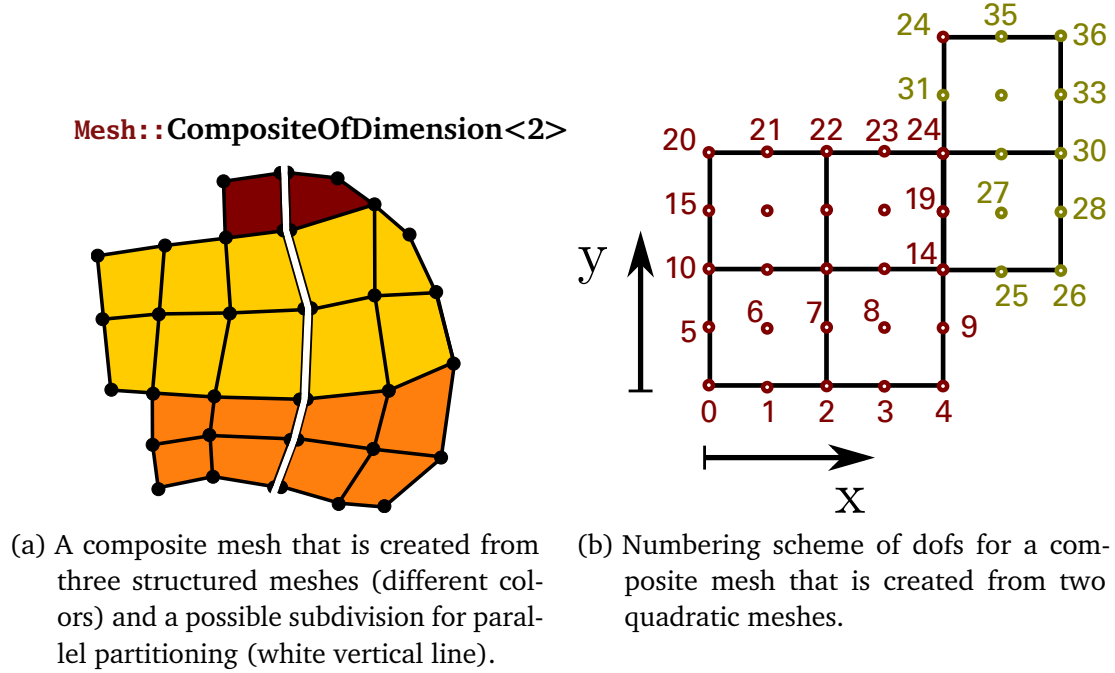


Figure 1.7: Examples for composite meshes that combine the advantages of structured and unstructured meshes.

## 1.3 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.

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.

### 1.3.1 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 for every row of the matrix the non-zero locations and their values.

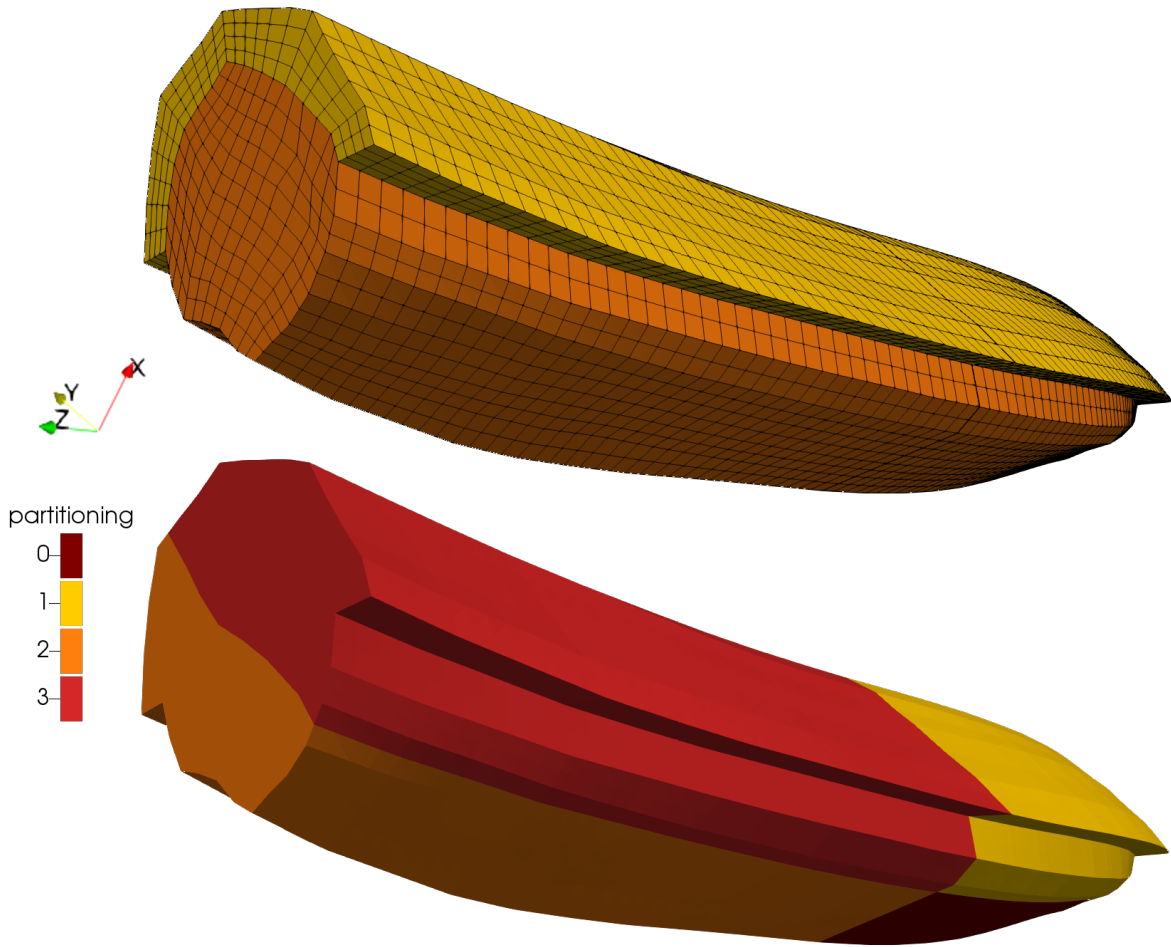


Figure 1.8: Composite mesh of the biceps muscle. Top: the two structured meshes from which the composite mesh is created, bottom: partitioning to four processes.

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:

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

Here, the number  $n_{dofs\_per\_basis}$  of dofs per 1D element is 2 and 3 for linear and quadratic Lagrange bases and 4 for cubic Hermite basis functions. The number  $n_{dofs\_per\_node}$  of dofs per node is 1 for Lagrange basis functions and 2 for Hermite basis functions. The value  $n_{1D\_overlaps}$  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  $n_{non-zeros}$  of non-zero entries per row is computed by exponentiation of  $n_{1D\_overlaps}$  with the dimensionality  $d$ .

Because no assumption can be made about how the bands of non-zero entries in the matrix are distributed to the diagonal and off-diagonal storage parts, the same value of  $n_{non-zeros}$  is used as estimate to preallocate both the diagonal and the off-diagonal part of the local matrix storage.

In the following, the non-zero structure of an exemplary stiffness matrix is shown. A 3D regular fixed mesh of  $4 \times 4 \times 4$  elements with quadratic Lagrange basis functions is considered. The Laplace equation ?? is solved with Dirichlet boundary conditions at the bottom and top planes of the volume. The prescribed values are 1 at the bottom and 2 at the top. The solution is visualized in the left of Fig. 1.9. The computation is performed with four processes. Figure 1.9 shows the partitioning on the right.

The non-zero estimates computed by Eq. (1.1) are  $n_{1D\_overlaps} = 5$  and  $n_{non-zeros} = 125$ . The mesh has 4 elements and, thus, 9 nodes per coordinate direction and therefore  $n_{global} = 9^3 = 729$  dofs. Figure 1.10 shows the resulting sparsity pattern of the stiffness matrix  $\mathbf{K}$ . The portions of the four processes are indicated by different colors. The



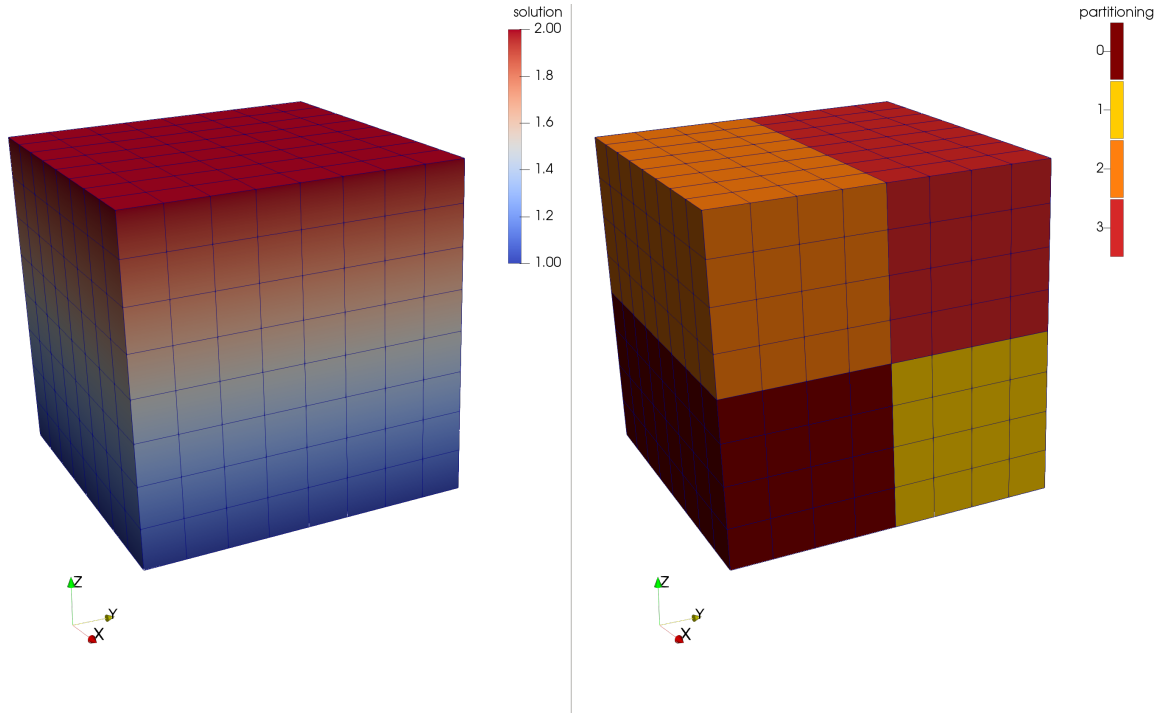


Figure 1.9: Solution of the Laplace equation  $\Delta \mathbf{u} = \mathbf{f}$  with prescribed values  $\mathbf{u}|_{\text{bottom}} = 1$  and  $\mathbf{u}|_{\text{top}} = 2$ . The mesh consists of  $4 \times 4 \times 4$  quadratic elements and, thus,  $9^3$  nodes. Left: Solution, right: Partitioning to four processes.

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

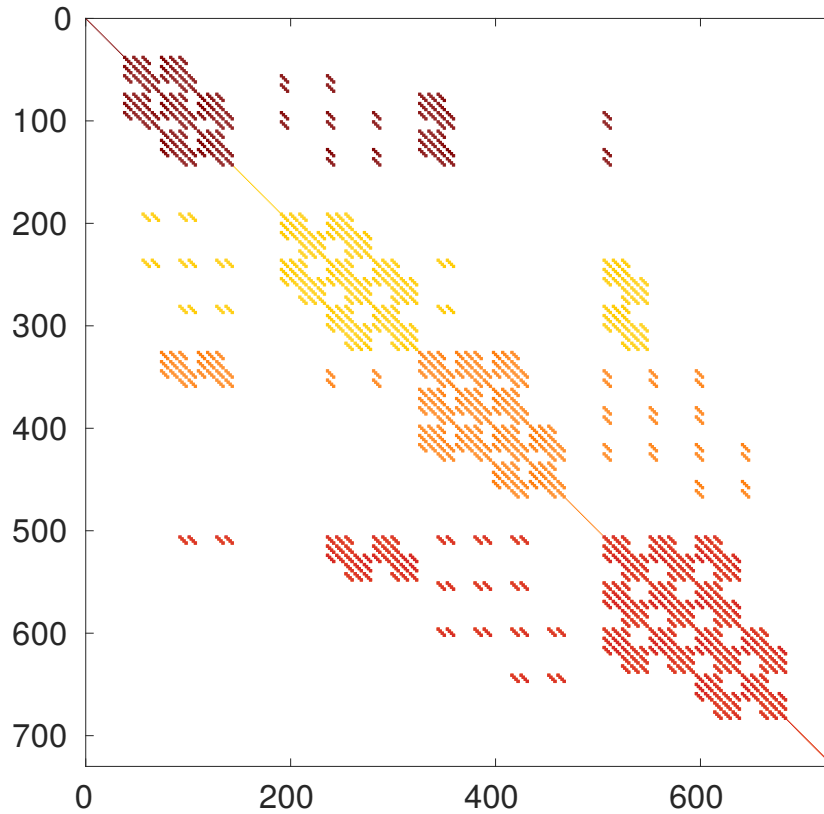


Figure 1.10: Sparsity pattern, i.e., locations of non-zero entries of the  $729 \times 729$  stiffness matrix  $K$  for the example problem in Fig. 1.9. The rows for the four processes are given by different colors matching the partitioning in Fig. 1.9. The processes have 144, 180, 180 and 225 local dofs.

### How To Reproduce

In any example, the system matrix can be written to a MATLAB compatible file by specifying the settings `'dumpFormat': 'matlab', 'dumpFilename': 'out'`. To get the non-zero structure for the example in Fig. 1.10, compile the *laplace3d* example and run the following:

```
cd $OPENDIHU_HOME/examples/laplace/laplace3d/build_release
mpirun -n 4 ./laplace_quadratic ../settings_quadratic_matrix_output.
  ↪ py -ksp_view
```

The flag `-ksp_view` is parsed by PETSc and outputs matrix statistics such as the number of preallocated and actual non-zeros. A file `out_matrix_000.m` is created that can be loaded in MATLAB. Use `spy(stiffnessMatrix)` to plot the non-zero structure.

### 1.3.2 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

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

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. 1.3.3](#). 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  $\xi$  do
4       Compute Jacobian  $J_e(\xi)$ 
5       Evaluate integrand  $I_{e,i,j}(\xi) = c \cdot I(J_e, \xi)$  ↪for all elements  $e$ /dofs  $(i, j)$  at once
6       matrix_entries[ $i, j$ ] = Quadrature( $I_{e,i,j}(\xi)$ ) ↪for all el.  $e$ /dofs  $(i, j)$  at once
7     for dof  $i = 0, \dots, n_{\text{dofs\_per\_element}} - 1$  do
8       for dof  $j = 0, \dots, n_{\text{dofs\_per\_element}} - 1$  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 [\[Kre12\]](#); [\[Kre15\]](#) is used. Vc provides zero-overhead C++ types for explicitly data-parallel programming and directly employs the respective vector instructions where these types are used.

To compute the integral in [Eq. \(1.2\)](#), 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  $\xi$  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  $i, j \in \{0, 1, \dots, n_{\text{dofs\_per\_element}} - 1\}$  with the number  $n_{\text{dofs\_per\_element}}$  of dofs per element. The set of  $4(n_{\text{dofs\_per\_element}})^2 \cdot (n_{\text{sampling\_points}})^d$  computed values is passed to the implementation of the  $d$ -dimensional quadrature rule in line 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 explicitly stored in the elements. To assemble the global matrix, PETSc uses the mapping from local to global numbering, which it can maintain by storing the constant offset in the global numbering on every subdomain. Mappings from global dof or node numbers to local numbers are not needed in this algorithm. In general, storing global information that would require memory of  $\mathcal{O}(n_{\text{global}})$  is avoided in all algorithms to ensure good parallel weak scaling properties.

### 1.3.3 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. 1.9](#) was run except for a larger number of  $50 \times 50 \times 50$  elements. This setup lead to a total number of  $n_{\text{global}} = 1\,030\,301$  dofs. Gauss quadrature with three sampling points per coordinate direction and, thus,  $3^3 = 27$  sampling points in

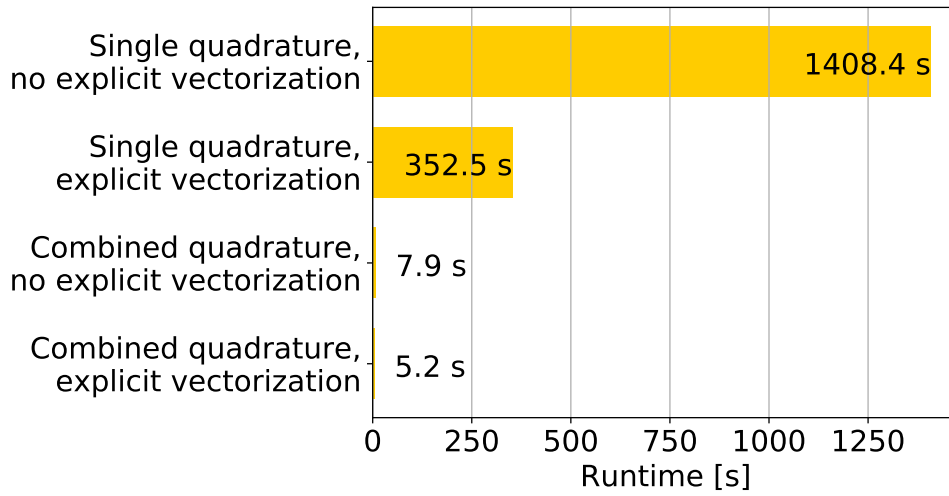


Figure 1.11: Runtimes of different optimizations for the algorithm to assemble the FEM stiffness matrix.

total was used. The program was executed with four processes on 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 bandwidth of  $204.8 \frac{\text{GB}}{\text{s}}$  per socket. The runtime for the assembly of the stiffness matrix with dimensions  $n_{\text{global}} \times n_{\text{global}}$  was measured for all four variants. Figure 1.11 presents the resulting runtimes.

It can be seen that a large difference in runtime exists between the variants with quadrature of single values compared to the combined quadrature. In the case of no explicit vectorization (first and third bar from the top in Fig. 1.11) the runtime reduces to less than 0.6 %, in the case of explicit vectorization (second and fourth bar from the top in Fig. 1.11) the runtime reduces to less than 1.5 %. The reason for this enormous gain in performance is three-fold. First, the values of the Jacobian can be reused for the same element and sampling point. Second, the combined quadrature for multiple values yields more cache-efficient memory access because the vector of values is stored consecutively in memory and can be fetched from the cache by less load operations. For the single quadrature, the individual values are fetched at different times from different memory locations. Third, the compiler is able to employ SIMD instructions for the combined quadrature, a process called auto-vectorization.

The performance improvements from the second variation, the use of explicit vectorization by simultaneously computing the entries for four elements at once can be seen by comparing the first and second bars and the third and forth bars in Fig. 1.11. The run-

time reduction of explicit vectorization with single quadrature from 1408.4 s to 352.5 s is exactly by the expected factor of four. This shows that explicit vectorization works as expected and that no auto-vectorization could be performed by the compiler for the single quadrature. The runtime reduction of explicit vectorization from 7.9 s to 5.2 s during combined quadrature corresponds to a speedup of only approximately 1.5. This shows that combined quadrature without explicit vectorization already allows the compiler to employ some auto-vectorization. However, using the explicit vectorization approach on the level of different elements instead of the level of quadrature values still has a positive effect.

In total, the performance gain from the most naive implementation (top bar in Fig. 1.11) to the most optimized version (bottom bar in Fig. 1.11) 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. 1.11 can be reproduced as follows. The explicit vectorization can be turned on and off with the variable `USE_VECTORIZED_FE_MATRIX_ASSEMBLY` in the configuration of the scons build system in the file `$OPENDIHU_HOME/user-variables.scons.py` (ca. line 75). Normally, only the variant with combined quadrature is implemented. To test the single quadrature, checkout the git branch `fem_assembly_measurement`. The single quadrature is on by default, to change back to the combined quadrature, edit the following line:

```
vi $OPENDIHU_HOME/core/src/spatial_discretization/
  ↪ finite_element_method/01_stiffness_matrix_integrate.tpp +17
```

For all variants of the algorithm, compile and run the following example:

```
cd $OPENDIHU_HOME/examples/laplace/laplace3d/build_release
mpirun -n 4 ./laplace_quadratic ../settings_quadratic.py
```

The duration of the algorithm for stiffness matrix assembly will be printed.

### 1.3.4 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  $(-\int_{\Omega^{\text{el}}} \nabla \phi_i \cdot \nabla \phi_j d\xi)$  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  $(-\int_{\Omega} \nabla \phi_i \cdot \nabla \phi_j d\xi)$  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  $\Omega$  is the sum of integrals over all elements  $\Omega^{\text{el}}$  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  $\int \phi_i \phi_j d\xi$  can be precalculated. The element and node stencils for the mass matrix are given in Tab. 1.3.



Dim.	Element contribution	Node stencil
1D	$\begin{bmatrix} \underline{-1} & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & \underline{-2} & 1 \end{bmatrix}$
2D	$\frac{1}{6} \begin{bmatrix} 1 & 2 \\ \underline{-4} & 1 \end{bmatrix}$	$\frac{1}{3} \begin{bmatrix} 1 & 1 & 1 \\ 1 & \underline{-8} & 1 \\ 1 & 1 & 1 \end{bmatrix}$
3D	center: $\frac{1}{12} \begin{bmatrix} 0 & 1 \\ \underline{-4} & 0 \end{bmatrix}$ bottom: $\frac{1}{12} \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$	top: $\frac{1}{12} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 0 & 2 \\ 1 & 2 & 1 \end{bmatrix}$ center: $\frac{1}{12} \begin{bmatrix} 2 & 0 & 2 \\ 0 & \underline{-32} & 0 \\ 2 & 0 & 2 \end{bmatrix}$ bottom: same as top

Table 1.2: Stencils of the Finite Element stiffness matrix of  $\Delta u$  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	$\frac{1}{6} \begin{bmatrix} 2 & 1 \end{bmatrix}$	$\frac{1}{6} \begin{bmatrix} 1 & \underline{4} & 1 \end{bmatrix}$
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. 1.3.2](#). 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.

### 1.3.5 Algorithm for Dirichlet Boundary Conditions

The assembled system matrix needs to be adjusted when Dirichlet boundary conditions are specified. Dirichlet boundary conditions are ensured by replacing equations involving the prescribed dofs by the definition of the boundary conditions. This involves changes in the system matrix and right hand side of the Finite Element formulation.

Consider the following matrix equation resulting from a FE discretization with four dofs  $u_1$  to  $u_4$ :

$$\begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{pmatrix}.$$

We assume a Dirichlet boundary condition for the last dof,  $u_4 = \hat{u}_4$ . Enforcing this condition is accomplished by the following adjusted system of equations:

$$\begin{pmatrix} m_{11} & m_{12} & m_{13} & 0 \\ m_{21} & m_{22} & m_{23} & 0 \\ m_{31} & m_{32} & m_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} f_1 - m_{14} \hat{u}_4 \\ f_2 - m_{24} \hat{u}_4 \\ f_3 - m_{34} \hat{u}_4 \\ \hat{u}_4 \end{pmatrix}.$$

The last equation has been replaced by  $u_4 = \hat{u}_4$ , all summands in the other equations where  $u_4$  occurred have been brought to the right hand side and  $u_4$  has been substituted by the prescribed value  $\hat{u}_4$ .

Thus, 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.

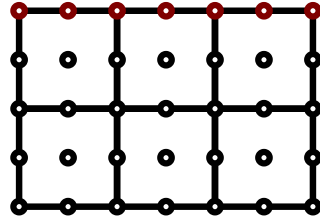
This method also works for more prescribed values as demonstrated with the additional Dirichlet boundary condition  $u_2 = \hat{u}_2$ . Executing the scheme results in the following system:

$$\begin{pmatrix} m_{11} & 0 & m_{13} & 0 \\ 0 & 1 & 0 & 0 \\ m_{31} & 0 & m_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} f_1 - m_{12} \hat{u}_2 - m_{14} \hat{u}_4 \\ \hat{u}_2 \\ f_3 - m_{32} \hat{u}_2 - m_{34} \hat{u}_4 \\ \hat{u}_4 \end{pmatrix}.$$

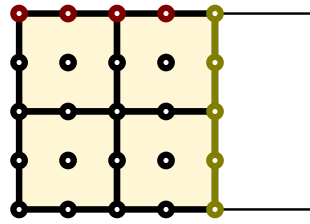
During parallel execution, only a distinct subset of rows of the matrix equation is accessible on every rank. However, for the subtractions from the right hand side, the full vector of prescribed boundary conditions values is needed on every rank. Additionally, the corresponding matrix entries are required. 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 1.12 (a) shows a 2D quadratic mesh with  $3 \times 2$  elements. The top layer of nodes has prescribed Dirichlet boundary conditions, marked by the red circles. The elements

(a) global mesh



(b) subdomain rank 0



(c) subdomain rank 1

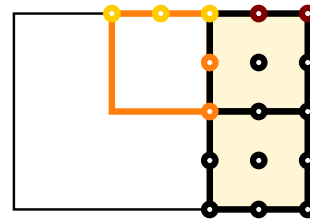


Figure 1.12: Example to illustrate ghost element transfer that is needed for handling Dirichlet boundary conditions in parallel. A mesh with Dirichlet boundary conditions on the red nodes is given in (a). The subdomains for two processes are given in (b) and (c). (c) shows a ghost element in orange that is sent from rank 0 to rank 1.

are partitioned to two processes with subdomains containing four and two elements, as shown in Fig. 1.12 (b) and Fig. 1.12 (c).

On rank 0, the right-most layer of nodes consists of ghost nodes. All other nodes are non-ghost and correspond to the matrix rows and right hand side entries that have to be manipulated by this rank. In every of these matrix rows  $i$ , only entries in columns  $j$  that correspond to nodes in the same element as  $i$  are non-zero. For these columns  $j$ , the prescribed Dirichlet boundary condition values  $\hat{u}_j$  need to be known such that the product of matrix entry  $m_{ij}$  and prescribed value  $\hat{u}_j$  can be subtracted from the right hand side at the corresponding row  $i$ . This is fulfilled for rank 0 since the required boundary condition values are all part of the subdomain. The top right boundary condition node in the top right element of rank 0's subdomain is stored as ghost value, the other four are non-ghosts.

As can be seen in Fig. 1.12 (c), the subdomain of rank 1 has no ghost nodes. The rank owns three boundary condition nodes in the top layer. It has to perform right hand side subtractions for the twelve rows of the matrix equation that correspond to the  $3 \times 4$  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. 1.12 \(c\)](#) 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 knows that rank 1 will need this information because it stores the right-most yellow node and the orange nodes in subdomain 1 as ghost nodes in its own subdomain. Therefore, no request from rank 1 is necessary.

In general, every rank constructs ghost elements from own elements that contain both at least one boundary condition node and at least one ghost node without boundary condition. The global indices of all nodes of these two kinds and the corresponding boundary condition values are packaged as ghost element and sent to the rank of the neighboring subdomain. Every process potentially sends multiple ghost elements to multiple neighboring ranks.

Because a rank does not know the number of ghost elements it will receive a-priori, one-sided communication is employed, which was introduced with the MPI 2.0 standard. More specifically, *passive target* communication is used where only the sending rank is explicitly involved in the transfer.

After the data 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 performed. The algorithm has to ensure that the same subtraction is not executed multiple times when the particular pair of nodes is obtained once from the local subdomain and once from a received ghost element. After solving the linear system with the updated stiffness matrix and right hand side, the dofs on nodes with Dirichlet boundary conditions will have the prescribed values.

Considering the overhead for ensuring Dirichlet boundary conditions, the question may arise whether the partitioning scheme should be designed in a better way to simplify the presented algorithm. However, applying Dirichlet boundary conditions is the only process where the subdomains including ghost nodes 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.

In summary, the presented algorithm fits our design goals of good performance. It is used in our implementation to enforce Dirichlet boundary conditions for static and dynamic problems. The algorithm is executed after assembling the stiffness matrix. In consequence, for static problems the linear system solver sets the prescribed values at the respective dofs and the boundary conditions are fulfilled. For dynamic problems with constant stiffness matrices, Dirichlet boundary conditions have to be ensured in every timestep. After running the algorithm on the system matrix once, the operation on the right hand side vector needs to be repeated in every timestep on the updated right hand side.

# 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>
- [Bal97] **Balay**, S. et al.: *Efficient management of parallelism in object oriented numerical software libraries*, Modern Software Tools in Scientific Computing, ed. by **Arge**, E.; **Bruaset**, A. M.; **Langtangen**, H. P., Birkhäuser Press, 1997, pp. 163–202
- [Fal02] **Falgout**, R. D.; **Yang**, U. M.: *Hypre: a library of high performance preconditioners*, International Conference on Computational Science, Springer, 2002, pp. 632–641
- [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
- [Kre12] **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, Johann Wolfgang Goethe-Universität Frankfurt am Main, 2015, p. 256
- [Mai19] **Maier**, B. et al.: *Highly parallel multi-physics simulation of muscular activation and EMG*, COUPLED PROBLEMS 2019, 2019, pp. 610–621
- [Röh17] **Röhrle**, O.: *DiHu - Towards a digital human*, Project Website, 2017, [https://ipvs.informatik.uni-stuttgart.de/SGS/digital\\_human/index.php](https://ipvs.informatik.uni-stuttgart.de/SGS/digital_human/index.php)
- [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%2Fs10974-007-9125-6>