Adaptive Finite Elements for Systems of PDEs: Software Concepts, Multi-level Techniques and Parallelization

DISSERTATION

zur Erlangung des akademischen Grades

Doctor rerum naturalium
(Dr. rer. nat.)

vorgelegt

der Fakultät Mathematik und Naturwissenschaften
der Technischen Universität Dresden

von

Dipl.-Inf. Simon Vey

geboren am 22. Juli 1973 in Neunkirchen-Seelscheid

Gutachter:

Eingereicht am:

Tag der Disputation:

ii

Für Stephanie und Johanna

iv

Contents

Chapter 1

Introduction

In the recent past, the field of scientific computing has become of more and more importance for scientific as well as for industrial research, playing a comparable role as experiment and theory do. This success of computational methods in scientific and engineering research is next to the enormous improvement of computer hardware to a large extend due to contributions from applied mathematicians, who have developed algorithms which make real life applications feasible. Examples are adaptive methods, high order discretization, fast linear and non-linear solvers and multi-level methods.

The application of these methods in a large class of problems demands for suitable and robust tools for a flexible and efficient implementation. Today, there exist several different simulation packages for the numerical solution of partial differential equations by the above mentioned algorithms, which are suitable to solve real world problems. For an example of an adaptive finite element software, I refer to [44].

In order to play a crucial role in scientific and engineering research, besides efficiency in the numerical solution, also efficiency in problem setup and interpretation of simulation results is of utmost importance. As modeling and computing comes closer together, efficient computational methods need to be applied to new sets of equations. The problems to be addressed by simulation methods become more and more complicated, ranging over different scales, interacting on different dimensions and combining different physics. Such problems need to be implemented in a short period of time, solved on complicated domains and visualized with respect to the demand of the user. Only a modular abstract simulation environment will fulfill these requirements and allow to setup, solve and visualize real-world problems appropriately.

In this work, the concepts and the design of the C++ finite element toolbox AMDiS (adaptive multidimensional simulations) are described. It is shown, how abstract data structures and modern software concepts can help to design user-friendly finite element software, which provides large flexibility in problem definition while on the other hand efficiently solves these problems. Attempts to modularize finite element codes have recently intensified, see e.g. [11, 4]. The basic principles and advantages of modularization are commonly known and acknowledged. However, a widely accepted modular design of finite element software has not been developed yet.

AMDiS extends some of the mathematical concepts of the adaptive finite element C-library ALBERTA [44] and realizes them in a modular object oriented design. The main design goals of AMDiS are:

- **High level of abstraction**: The problem definition can be done on a very high abstraction level in a dimension independent way keeping numerical issues away from the user as far as possible.

- **Generality**: With AMDiS, a very large class of problems can be implemented in an intuitive way. Linear and nonlinear problems, stationary as well as time dependent problems can be treated. Different problems of the same or of different dimension can be coupled in one

code, and also higher order equations can be solved as system of second order equations.

- **Extensibility and reusability**: A flexible interface oriented design allows an easy extension of the code to fulfill new requirements and guarantees a high reusability of existing code in a modified context. Many of the software design patterns proposed in [23] are applied to achieve this goal.

- **Efficiency**: To solve the problems with as little computational effort as possible, many different techniques are applied. Some important examples are the use of adaptive mesh refinements, multigrid methods and parallelization.

The remainder of this work is structured as follows: In Chapter 2, the finite element discretization of partial differential equations (PDEs) of second order including different kinds of boundary conditions is described. The discretization of scalar problems is closely related to the one proposed in [44]. But in AMDiS, also general systems of second order PDEs can be discretized and solved. Such systems can be used to model the coupling of different problem aspects like pressure and velocity in fluid dynamics, or to solve higher order problems by writing them as system of second order equations. In AMDiS, the different components of a system can be discretized on different finite element spaces.

Chapter 3 treats the implementation of finite element spaces and the assemblage of the corresponding linear systems of equations. Here, many of the concepts of [44] are implemented in an object oriented design, again extended by the support for PDE systems. The clear distinction between problem definition interfaces and assembler implementations leads to a high abstraction level and to general and reusable assembling routines.

Topic of Chapter 4 is the implementation of adaptation loops. Adaptation loops are the highest abstraction level in the simulation. They control the adaptive process including assemblage and solution of the linear system of equations, computation of a posteriori error indicators, local mesh adaptation and time step control. Since such adaptation strategies can become rather complicated, it is important to implement them in a reusable way. The adaptation loop accesses the concrete problem classes through so-called iteration and time interfaces. These interfaces e.g. allow to define the coupling of different problems or the linearization of nonlinear problems without modifications in the adaptation loop.

The most time-consuming part in most simulations is the solution of the linear system of equations. Multi-level methods use discretization hierarchies to solve the system of equations in a very efficient way. In many cases multi-level or *multigrid* methods linearly scale with the number of discrete nodes. In Chapter 5, the implementation of a multigrid solver in AMDiS is described. Special care here is taken of the adaptive finite element context.

Besides the development of more efficient algorithms also the growing hardware capabilities lead to an improvement of simulation possibilities. Modern computing clusters contain more and more processors and also personal computers today are often equipped with multi-core processors. In Chapter 6, the concept of full domain covering meshes is introduced which allows the parallelization of sequential code in a very easy way. Furthermore, the needed communication overhead compared to classical parallelization approaches is reduced.

In Chapter 7, the Shared Mesh Interface (SMI) is introduced. SMI is an interface that allows the distributed management of shared meshes. It is not part of the AMDiS library but can be used to couple arbitrary applications that share the same logical mesh and values that are defined on the mesh. One application example is the coupling of a simulation code with a separated result visualization.

Chapter 8 includes several simulation examples which demonstrate the possibilities of AMDiS. Furthermore, the results concerning the multigrid and parallelization concepts are discussed. Finally, in Chapter 9, some conclusions are drawn and an outlook to future work is given.

Parts of this work have already been published in [40, 41, 47, 54, 55, 56].

Chapter 2

Finite element discretization

In AMDiS, systems of second order PDEs can be solved. Since often it is possible to write higher order problems as system of second order problems, also more general problems can be treated. In this chapter, the finite element discretization of systems of PDEs is described, including the discretization of boundary conditions and the treatment of instationary, non-linear and higher order problems.

The general second order problem with $\mathcal{N}$ components reads:

$$\sum_{q=1}^{\mathcal{N}} \left(-\nabla \cdot A^{p,q} \nabla u^q + b^{p,q} \cdot \nabla u^q + c^{p,q} u^q \right) = f^p \quad \text{in } \Omega \qquad p = 1, \ldots, \mathcal{N}, \qquad (2.1)$$

$$u^p = g^p \quad \text{on } \Gamma_D^p \quad p = 1, \ldots, \mathcal{N}, \qquad (2.2)$$

$$\sum_{q=1}^{\mathcal{N}} \left(A^{p,q} \nabla u^q \cdot \nu \right) = h^p \quad \text{on } \Gamma_N^p \quad p = 1, \ldots, \mathcal{N}. \qquad (2.3)$$

The problem is defined on Ω which is a domain in $\mathbb{R}^d$. The solution u^p of the p-th equation as well as $c^{p,q}$, f^p, g^p, h^p are real valued functions that can depend on space x and time t (for time dependent problems). $A^{p,q}$ is a $d \times d$ matrix and $b^{p,q}$ a vector of dimension d, both also depending on space and time. In principle, all functions can depend on arbitrary components of the solution. This non-linearity can be linearized over adaptive iterations or timesteps or using non-linear methods like the Newton method. Further assumptions on these functions are made in Section 2.4. $\Gamma_D^p \subset \partial\Omega$ is the Dirichlet boundary for the solution component u^p. $\Gamma_N^p \subset \partial\Omega$ is the Neumann boundary of u^p. ν is the outward unit surface normal to $\partial\Omega$. For $\mathcal{N} = 1$ and omitting p and q we obtain the scalar problem:

$$-\nabla \cdot A\nabla u + b \cdot \nabla u + cu = f \quad \text{in } \Omega \qquad (2.4)$$

$$u = g \quad \text{on } \Gamma_D \qquad (2.5)$$

$$A\nabla u \cdot \nu = h \quad \text{on } \Gamma_N. \qquad (2.6)$$

In this chapter we describe the way from the abstract problem formulation to a linear system of equations for a given mesh $\mathcal{T}$ that approximates Ω by simplicial elements. The assemblage of the system of equations is done automatically in AMDiS, keeping numerical issues away from the user. The user only has to specify the formal problem.

When the system of equations has been assembled, it can be solved by an iterative solver. The solve-step results in a vector containing one solution vector for each component of the system (a vector of $\mathcal{N}$ vectors, see Section 2.4). With U_i^p we denote the i-th entry of the p-th solution vector.

The true solution u^p of component p is approximated by u_h^p which is a linear combination of the global basis functions ϕ_i^p that belong to the finite element space of component p (see Section
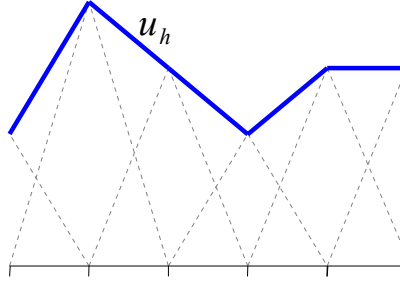
Figure 2.1: u_h as linear combination of linear basis functions in 1d.

2.2):

$$u_h^p := \sum_i U_i^p \phi_i^p. \tag{2.7}$$

Figure 2.1 illustrates the construction of the approximate solution for a scalar problem in 1d with linear Lagrange basis functions.

2.1   Simplex, standard simplex and reference simplex

AMDiS meshes consist of simplices (lines in 1d, triangles in 2d, tetrahedra in 3d). Simplices allow an easy way to approximate complex domains, to construct local Lagrange basis functions using barycentric coordinates, and to perform local refinements by bisection without the need of hanging nodes.

Simplex   A *simplex* S of dimension d is given by its $d+1$ vertex coordinates $a_1, \ldots, a_{d+1} \in \mathbb{R}^d$, assuming that $a_2 - a_1, \ldots, a_{d+1} - a_1$ are linear independent vectors:

$$S := \text{conv hull}\{a_1, \ldots, a_{d+1}\}. \tag{2.8}$$

Standard simplex   Integration is always done over the *standard simplex*:

$$\hat{S} := \text{conv hull}\{0, e_1, \ldots, e_d\}, \tag{2.9}$$

where e_i are the unit vectors in $\mathbb{R}^d$. So basis functions have to be evaluated only once at each integration point. Integration over an arbitrary simplex S is done using the mapping $F_S : \hat{S} \to S$, which is defined by:

$$F_S(\hat{x}) := A_S \hat{x} + a_1, \tag{2.10}$$

where A_S is the matrix $[a_2 - a_1 \ldots a_{d+1} - a_1]$. Now the integral transformation from S to $\hat{S}$ for a function f can be done:

$$\int_S f(x)dx = \int_{\hat{S}} f(F_S(\hat{x}))|DF_S|d\hat{x}, \tag{2.11}$$

where DF_S is the Jacobian of F_S. DF_S is constant here, because F_S is an affine mapping.

In the context of parametric finite elements (see Section 3.7), the mapping F_S can be defined in a different way than described here. In particular, it may map into a higher world dimension which allows e.g. 2d meshes in a 3d world, or it can be time dependent which allows moving meshes.
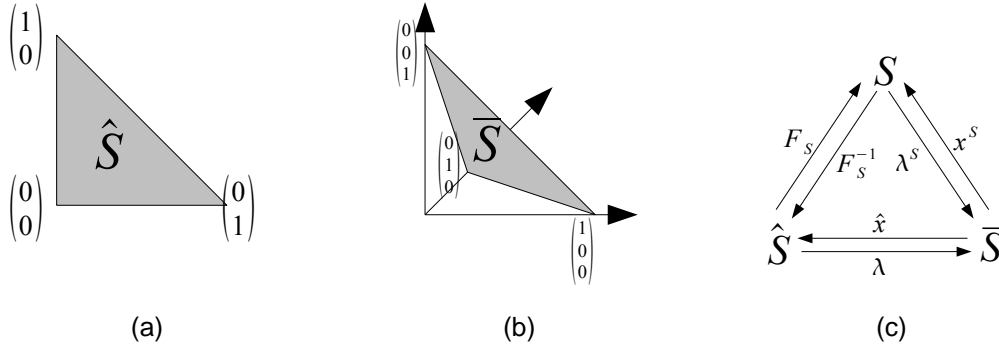
Figure 2.2: (a): Standard simplex $\hat{S}$, (b): Reference simplex $\bar{S}$, (c): Mappings between S, $\hat{S}$ and $\bar{S}$

(a) local basis functions   (b) global basis functions
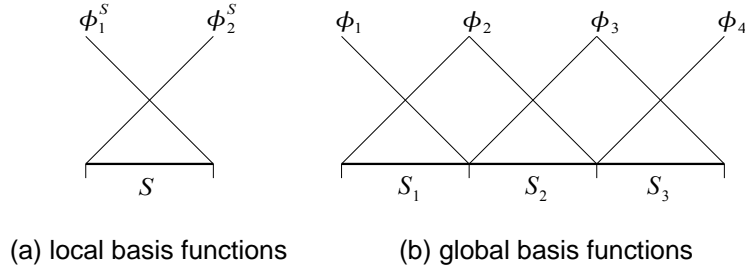
Figure 2.3: Local and global basis functions for linear finite elements in 1d

Reference simplex   Basis functions are defined and evaluated in barycentric coordinates on the *reference simplex*:

$$\bar{S} := \{(\lambda_1, \cdots, \lambda_{d+1}) \ : \ \lambda_k \geq 0 \wedge \sum_{k=1}^{d+1} \lambda_k = 1\}. \tag{2.12}$$

$\bar{S}$ is a subset of a hyper surface in $\mathbb{R}^{d+1}$. On barycentric coordinates it is easy to construct polynomial basis functions that are 1 at the point where the function is located and 0 on points where other local basis functions are located. This property of the basis functions directly leads to $u^i(x_k) = U_k^i$, where x_k are the coordinates of global node k.

For a point $x \in \mathbb{R}^d$ in simplex S the barycentric coordinates $\lambda^S(x)$ can be determined by the $d+1$ equations:

$$\sum_{k=1}^{d+1} \lambda_k^S(x) a_k = x \quad \text{and} \quad \sum_{k=1}^{d+1} \lambda_k^S(x) = 1. \tag{2.13}$$

The world coordinates x of barycentric coordinates λ of simplex S can be computed by $\sum_{k=1}^{d+1} \lambda_k a_k$.

Figure 2.2 shows the standard simplex $\hat{S}$, the reference simplex $\bar{S}$ (both in 2d) and the mappings between S, $\hat{S}$ and $\bar{S}$.

2.2   Finite elements and finite element spaces

In the following we define finite elements and finite element spaces, similar to the definitions in [14] by Ciarlet.

Finite element   A finite element S consists of a non empty domain Ω^S and a set Φ^S of local basis functions $\{\phi_i^S : \Omega^S \to \mathbb{R} \ : \ i = 1, \ldots, n\}$. The domain Ω^S in AMDiS is always described by

a d dimensional simplex. The value of n depends on the finite element type and the dimension. For linear Lagrange elements in 1d n is 2 (see Figure 2.3(a)).

Like mentioned in Section 2.1, local basis functions in AMDiS are defined once on the reference simplex $\bar{S}$. The local basis functions on the reference element are denoted by $\bar{\phi}_i$ with $i = 1, \ldots, n$. The local basis functions ϕ_i^S for an element S can be obtained by

$$\phi_i^S(x) := \bar{\phi}_i(\lambda^S(x)). \tag{2.14}$$

Every basis function is located at given coordinates in the element. These locations are called the *nodes* of an element. In AMDiS, we assume the following additional properties of the basis functions:

- Every basis function is 1 at the node where it is located.

- Every basis function is 0 at any node where another basis function is located.

These properties enable an easy discretization of Dirichlet boundary conditions and assure that the approximate solution at mesh nodes is equal to the corresponding entries of the solution vector U (see Section 2.4).

Finite element space   A finite element space V is a triple consisting of a set $\mathcal{T}$ of finite elements $\{S_1, \ldots, S_M\}$, a set Φ of global basis functions $\{\phi_1, \ldots, \phi_N\}$ and a mapping $\mathcal{G}^V : \mathcal{T} \times \{1, \ldots, n\} \rightarrow \{1, \ldots, N\}$ from local to global basis function indices. $\mathcal{T}$ is given by the mesh together with the definition of local basis functions. The mapping from local to global basis functions is described in Section 2.3. We define $\mathcal{G}_S^V(\cdot) := \mathcal{G}^V(S, \cdot)$ as the mapping from the local indices of element S to the corresponding global indices.

When a system of PDEs has to be solved, each component can be discretized on a different finite element space. The mesh must be the same for each component, but the type of basis functions can vary.

2.3   Global and local basis function indices

At each element S of a given finite element space V the local basis functions $\phi_1^S, \ldots, \phi_n^S$ are located. Different elements that share a common node in the discretization all contain a local basis function for this common node. The global basis function at this node is the sum over all these local basis functions. In Figure 2.3 ϕ_2 is the sum of $\phi_2^{S_1}$ and $\phi_1^{S_2}$. In other words: local basis functions of an element S can be seen as global basis functions restricted to Ω^S.

The numbering of global basis functions is independent of the local numberings. So, we define a mapping to switch from local to global indices. The mapping $\mathcal{G}_S^V(i)$ provides the global index of local index i in element S of finite element space V. In Figure 2.3 $\mathcal{G}_{S_3}^V(1)$ is 3 and $\mathcal{G}_{S_3}^V(2)$ is 4.

The mappings $\mathcal{G}_S^V$ are stored locally at the elements. For each local node of an element the global index is stored.

2.4   Discretization of stationary 2nd order problems

In this section, the finite element discretization of second order problems is described. First, the discretization of scalar problems is explained in Section 2.4.1. The discretization of systems of PDEs can be reduced to the scalar discretization in a straightforward way, which is described in Section 2.4.2.

2.4.1 Scalar problems

In the scalar case we have to discretize the following equation:

$$-\nabla \cdot A\nabla u + b \cdot \nabla u + cu = f \quad \text{on } \Omega \subset \mathbb{R}^d \tag{2.15}$$
$$u = g \quad \text{on } \Gamma_D \tag{2.16}$$
$$A\nabla u \cdot \nu = h \quad \text{on } \Gamma_N \tag{2.17}$$

where d is the dimension, Ω is a domain in $\mathbb{R}^d$, $A \in L^\infty(\Omega, \mathbb{R}^{d\times d})$, $b \in L^\infty(\Omega, \mathbb{R}^d)$, $c \in L^\infty(\Omega)$, $f \in L^2(\Omega)$. $\Gamma_D \subset \partial\Omega$ is the Dirichlet boundary ($|\Gamma_D| \neq 0$) with values $g : \Gamma_D \to \mathbb{R}$. We assume that g has an extension to some function $g \in H^1(\Omega)$. $\Gamma_N = \partial\Omega \setminus \Gamma_D$ is the Neumann boundary and $h \in L^2(\partial\Omega)$. By ν we denote the outer normal vector on $\partial\Omega$.

We define $X := H^1(\Omega)$ and $\hat{X} := \{v \in H^1(\Omega) \ : \ v = 0 \text{ on } \Gamma_D\}$. The weak formulation of equation (2.15) reads: Find $u \in g + \hat{X}$, such that

$$\int_\Omega -\nabla \cdot A\nabla u\phi dx + \int_\Omega b \cdot \nabla u\phi dx + \int_\Omega cu\phi dx = \int_\Omega f\phi dx \quad \forall\phi \in \hat{X}. \tag{2.18}$$

Existence and uniqueness results for this equation can be obtained by the Lax-Milgram theorem (see [13]). The same kind of equation can result from linearizations of nonlinear elliptic problems and from time discretizations of parabolic problems.

The equation should be discretized on the finite element space V with mesh $\mathcal{T} = \{S_1, \ldots, S_M\}$ (a triangulation of Ω) and the set of global basis functions $\Phi = \{\phi_1, \ldots, \phi_N\}$. We make the following assumptions considering the finite element space:

- $\phi_i \in X \quad i = 1, \ldots, N$.

- Each basis function is 1 at the node where it is located and 0 at all nodes where other basis functions are located (see Section 2.2).

- $\Phi_{\Gamma_D} \subset \Phi$ ($\Phi_{\Gamma_D} \neq \emptyset$) contains the basis functions located at the Dirichlet boundary Γ_D.

We define

$$X_h := \text{span}\{\phi_i : \phi_i \in \Phi\} \quad \text{and} \tag{2.19}$$
$$\hat{X}_h := \text{span}\{\phi_i : \phi_i \in \Phi \setminus \Phi_{\Gamma_D}\}. \tag{2.20}$$

The function $g_h \in X_h$ is an approximation of $g \in X$. The discrete version of equation (2.18) then reads: Find $u_h \in g_h + \hat{X}_h$, such that

$$\int_\Omega -\nabla \cdot A\nabla u_h\phi_i dx + \int_\Omega b \cdot \nabla u_h\phi_i dx + \int_\Omega cu_h\phi_i dx = \int_\Omega f\phi_i dx \quad \forall\phi_i \in \Phi \setminus \Phi_{\Gamma_D} \tag{2.21}$$

Using integration by parts, the first integral can be rewritten as $\int_\Omega A\nabla u_h\nabla\phi_i dx - \int_{\partial\Omega} \phi_i A\nabla u_h\nu \, d\sigma$. The boundary integral is non-zero only at the Neumann boundary. The discretization of Neumann boundary conditions is topic of Section 2.5.2. For the rest of this section we assume the boundary integral to be zero.

The discrete solution u_h is a linear combination of the global basis functions:

$$u_h = \sum_{j=1}^N U_j\phi_j, \tag{2.22}$$

with the coefficient vector $U = (U_i)_{i=1,\ldots,N} \in \mathbb{R}^N$. In the same way g_h is represented by the coefficient vector $G = (G_i)_{i=1,\ldots,N}$. Let I_{Γ_D} be the set of basis function indices in Φ that are

located at Γ_D and I_Ω the set of basis function indices of $\Phi \setminus \Phi_{\Gamma_D}$. Then we have the N discrete equations

$$\sum_{j=1}^{N} U_j \left(\int_\Omega \nabla \phi_i \cdot A \nabla \phi_j dx + \int_\Omega \phi_i b \cdot \nabla \phi_j dx + \int_\Omega \phi_i c \phi_j dx \right) \quad = \quad \int_\Omega f \phi_i dx \qquad \forall i \in I_\Omega \quad (2.23)$$

$$U_i \quad = \quad G_i \qquad \qquad \forall i \in I_{\Gamma_D}.(2.24)$$

With

$$M_{i,j} := \begin{cases} \int_\Omega \nabla \phi_i \cdot A \nabla \phi_j dx + \int_\Omega \phi_i b \cdot \nabla \phi_j dx + \int_\Omega \phi_i c \phi_j dx & i \in I_\Omega \\ 1 & i \in I_{\Gamma_D}, i = j \\ 0 & i \in I_{\Gamma_D}, i \neq j \end{cases} \qquad (2.25)$$

and

$$F_i := \begin{cases} \int_\Omega f \phi_i dx & i \in I_\Omega \\ G_i & i \in I_{\Gamma_D} \end{cases} \qquad (2.26)$$

and $M := (M_{i,j})_{i,j=1,\dots,N}$ and $F := (F_i)_{i=1,\dots,N}$ we can write the N equations as one matrix-vector equation:

$$M \cdot U = F. \qquad (2.27)$$

This is the linear system of equations that has to be assembled. The solution of this system of equations is the coefficient vector U. The linear combination $u_h = \sum_{j=1}^{N} U_j \phi_j$ is the wanted approximation to u. To assemble the linear system of equations, we have to compute $M_{i,j}$ and F_i.

The domain Ω is approximated by the mesh $\mathcal{T}$. Hence, integrals over Ω can be replaced by the sum of integrals over all mesh elements ($\int_\Omega f(x) dx \approx \sum_{S \in \mathcal{T}} \int_S f(x) dx$). Furthermore, we replace the global basis functions by local basis functions defined on the elements. As mentioned in Section 2.3, each global basis function can be seen as the sum of all local basis functions defined at the same node as the global function. The local basis functions on element S are $\{\phi_1^S, \dots, \phi_n^S\}$. The assemblage of the system of equations is done element-wise like shown in Algorithm 1.

The local basis functions are given implicitly by the basis functions $\{\bar\phi_1(\lambda), \dots, \bar\phi_n(\lambda)\}$ defined on the reference simplex $\bar S$ together with the mapping λ^S (see Section 2.1). After transformation of the integrals to the standard simplex $\hat S$, we can replace line 6 by

$$M_{\mathcal{G}_S^V(k),\mathcal{G}_S^V(l)} \; +=\; \int_{\hat S} \nabla_\lambda \bar\phi_k(\lambda(\hat x)) \cdot \Lambda(\hat x) A(F_S(\hat x)) \Lambda^t(\hat x) \nabla_\lambda \bar\phi_l(\lambda(\hat x)) |DF_S(\hat x)| \, d\hat x, \qquad (2.28)$$

line 7 by

$$M_{\mathcal{G}_S^V(k),\mathcal{G}_S^V(l)} \; +=\; \int_{\hat S} \bar\phi_k(\lambda(\hat x)) \Lambda(\hat x) b(F_S(\hat x)) \cdot \nabla_\lambda \bar\phi_l(\lambda(\hat x)) |DF_S(\hat x)| \, d\hat x, \qquad (2.29)$$

line 8 by

$$M_{\mathcal{G}_S^V(k),\mathcal{G}_S^V(l)} \; +=\; \int_{\hat S} c(F_S(\hat x)) \bar\phi_k(\lambda(\hat x)) \bar\phi_l(\lambda(\hat x)) |DF_S(\hat x)| \, d\hat x \qquad (2.30)$$

and line 9 by

$$F_{\mathcal{G}_S^V(k)} \; +=\; \int_{\hat S} f(F_S(\hat x)) \bar\phi_k(\lambda(\hat x)) |DF_S(\hat x)| \, d\hat x, \qquad (2.31)$$

where Λ is the Jacobian of the barycentric coordinates λ on S. ∇_λ is the derivative with respect to λ. F_S is the mapping from the standard element $\bar S$ to the world element S and DF_S its Jacobian.

Algorithm 1 Element-wise assemblage of the linear system of equations

---

1: set $M_{i,j}$ and F_i to zero for $i, j = 1, \ldots, N$
2: **for** all elements S in $\mathcal{T}$ **do**
3:    **for** $k = 1, \ldots, n$ **do**
4:       **if not** $\mathcal{G}_S^V(k) \in I_{\Gamma_D}$ **then**
5:          **for** $l = 1, \ldots, n$ **do**
6:             $M_{\mathcal{G}_S^V(k), \mathcal{G}_S^V(l)} \ +\!= \ \int_S \nabla \phi_k^S \cdot A \nabla \phi_l^S dx$
7:             $M_{\mathcal{G}_S^V(k), \mathcal{G}_S^V(l)} \ +\!= \ \int_S \phi_k^S b \cdot \nabla \phi_l^S dx$
8:             $M_{\mathcal{G}_S^V(k), \mathcal{G}_S^V(l)} \ +\!= \ \int_S \phi_k^S c \phi_l^S dx$
9:             $F_{\mathcal{G}_S^V(k)} \ +\!= \ \int_S f \phi_k^S dx$
10:          **end for**
11:       **else**
12:          **for** $l = 1, \ldots, n$ **do**
13:             **if** $k = l$ **then**
14:                $M_{\mathcal{G}_S^V(k), \mathcal{G}_S^V(l)} = 1$
15:                $F_{\mathcal{G}_S^V(k)} = G_{\mathcal{G}_S^V(k)}$
16:             **else**
17:                $M_{\mathcal{G}_S^V(k), \mathcal{G}_S^V(l)} = 0$
18:             **end if**
19:          **end for**
20:       **end if**
21:    **end for**
22: **end for**

2.4.2   Systems of PDEs

The general vector valued problem is described by the equations (2.1)-(2.3). We want to discretize all components on the same triangulation $\mathcal{T}$ but allow different types of basis functions for the different components (e.g. different polynomial degree of Lagrange basis functions). The finite element space of component p is denoted by V^p with the set of global basis functions $\Phi^p = \{\phi_1^p, \ldots, \phi_{N^p}^p\}$.

In principle, one could discretize the different components also on different triangulations (e.g. different refinements of the same macro mesh). This would allow a better adaptation to the different component properties. But this is much more complex and may be the content of future work.

Analog to equation (2.27) in the last section, we can discretize the problem in a matrix-vector equation:

$$M \cdot U = F, \tag{2.32}$$

with $M := (M^{p,q})_{p=1,\ldots,\mathcal{N}, q=1,\ldots,\mathcal{N}}$, $U := (U^p)_{p=1,\ldots,\mathcal{N}}$ and $F := (F^p)_{p=1,\ldots,\mathcal{N}}$. Each $M^{p,q}$ is a $N^p \times N^q$ matrix with the entries

$$M_{i,j}^{p,q} := \begin{cases} \int_\Omega \nabla \phi_i^p \cdot A^{p,q} \nabla \phi_j^q dx + \int_\Omega \phi_i^p b^{p,q} \cdot \nabla \phi_j^q dx + \int_\Omega \phi_i^p c^{p,q} \phi_j^q dx & i \in I_\Omega^p \\ 1 & i \in I_{\Gamma_D^p}, i = j, p = q \\ 0 & \text{otherwise.} \end{cases} \tag{2.33}$$

Each F^p is a N^p dimensional vector with the entries

$$F_i^p := \begin{cases} \int_\Omega f \phi_i^p dx & i \in I_\Omega^p \\ G_i^p & i \in I_{\Gamma_D^p}. \end{cases} \tag{2.34}$$

Each U^p is a N^p dimensional vector which will contain the solution of component p after the linear system of equations is solved. $I_{\Gamma_D}^p$ is the set containing all global basis function indices of Φ^p that

are located at Γ_D^p. The set I_Ω^p contains all non Dirichlet indices of Φ^p. G_i^p represents the Dirichlet function g^p evaluated at the location of ϕ_i^p.

Like in Section 2.4.1, Ω is approximated by the triangulation $\mathcal{T} = \{S_1, \ldots, S_M\}$, the integrals are evaluated over the standard simplex $\hat{S}$ and the local basis functions are given in barycentric coordinates on the reference simplex $\bar{S}$. Algorithm 2 illustrates how the corresponding linear system of equations is assembled.

The values of n^p and n^q describe the number of local basis function indices at one element in the finite element spaces V^p and V^q. The sets $\{\bar{\phi}_1^p, \ldots, \bar{\phi}_{n^p}^p\}$ and $\{\bar{\phi}_1^q, \ldots, \bar{\phi}_{n^q}^q\}$ contain the local basis functions defined on the reference element $\bar{S}$ for the two finite element spaces.

Algorithm 2 Assemblage of the linear system of equations for vector valued problems

---

1: **for** $p = 1, \ldots, \mathcal{N}$ **do**
2:   set F_i^p to zero for $i = 1, \ldots, N^p$
3:   **for** $q = 1, \ldots, \mathcal{N}$ **do**
4:     set $M_{i,j}^{p,q}$ to zero for $i = 1, \ldots, N^p$ and $j = 1, \ldots, N^q$
5:     **for** all elements S in $\mathcal{T}$ **do**
6:       **for** $k = 1, \ldots, n^p$ **do**
7:         **if not** $\mathcal{G}_S^{V^p}(k) \in I_{\Gamma_D^p}$ **then**
8:           **for** $l = 1, \ldots, n^q$ **do**
9:             $\gamma_k = \mathcal{G}_S^{V^p}(k)$
10:            $\gamma_l = \mathcal{G}_S^{V^q}(l)$
11:            $M_{\gamma_k,\gamma_l}^{p,q} \mathrel{+}= \int_{\hat{S}} \nabla_\lambda \bar{\phi}_k^p(\lambda(\hat{x})) \cdot \Lambda(\hat{x}) A^{p,q}(F_S(\hat{x})) \Lambda^t(\hat{x}) \nabla_\lambda \bar{\phi}_l^q(\lambda(\hat{x})) |DF_S(\hat{x})| \, d\hat{x}$
12:            $M_{\gamma_k,\gamma_l}^{p,q} \mathrel{+}= \int_{\hat{S}} \bar{\phi}_k^p(\lambda(\hat{x})) \Lambda(\hat{x}) b^{p,q}(F_S(\hat{x})) \cdot \nabla_\lambda \bar{\phi}_l^q(\lambda(\hat{x})) |DF_S(\hat{x})| \, d\hat{x}$
13:            $M_{\gamma_k,\gamma_l}^{p,q} \mathrel{+}= \int_{\hat{S}} c^{p,q}(F_S(\hat{x})) \bar{\phi}_k^p(\lambda(\hat{x})) \bar{\phi}_l^q(\lambda(\hat{x})) |DF_S(\hat{x})| \, d\hat{x}$
14:            **if** $p = q$ **then**
15:              $F_{\gamma_k}^p \mathrel{+}= \int_{\hat{S}} f^p(F_S(\hat{x})) \bar{\phi}_k^p(\lambda(\hat{x})) |DF_S(\hat{x})| \, d\hat{x}$
16:            **else**
17:              **for** $l = 1, \ldots, n$ **do**
18:                **if** $k = l$ **and** $p = q$ **then**
19:                  $M_{\gamma_k,\gamma_l}^{p,q} = 1$
20:                  $F_{\gamma_k}^p = G_{\gamma_k}^p$
21:                **else**
22:                  $M_{\gamma_k,\gamma_l}^{p,q} = 0$
23:                **end if**
24:              **end for**
25:            **end if**
26:          **end for**
27:        **end if**
28:      **end for**
29:    **end for**
30: **end for**
31: **end for**

The resulting matrix M is a $\mathcal{N} \times \mathcal{N}$ matrix, where each entry $M^{p,q}$ is a $N^p \times N^q$ matrix with real valued entries. In principle, one could discretize systems also in a different way, such that the corresponding large matrix for each node contains a small $\mathcal{N} \times \mathcal{N}$ matrix. The advantage of this approach would be that the components in the solution vector for one node also would be grouped together. But the approach described in this work has the following advantages:

- The different components can be discretized on different finite element spaces. This would not be possible in the alternative approach, because the small matrices can only be defined if each component has the same discretization points.

- The structure of the discretization allows a straightforward re-use of scalar code in the vector

valued case (see Section 3.5).

- If $A^{p,q}$, $b^{p,q}$ and $c^{p,q}$ all are not defined (equal to zero for all values of x and t) for a given pair (p, q), the corresponding entries in the small matrices of the alternative approach would always be zero. This leads to unnecessary additional memory usage or to additional effort to store the small matrices in a sparse structure. In the approach made in this work, the whole matrix $M^{p,q}$ is zero and can be omitted in such a case.

2.5 Boundary conditions

Now the discretization of different types of boundary conditions is described. Dirichlet boundary conditions are treated in Section 2.5.1, Neumann boundary conditions are topic of Section 2.5.2. In Section 2.5.3, the implementation of other types of boundary conditions, like Robin or periodic boundary conditions, is explained.

2.5.1 Dirichlet boundary conditions

Dirichlet boundary conditions are already included in the discretizations described in Section 2.4. If the i-th global index of component p is located at Γ_D^p, a complete row of the linear system of equations represents the trivial equation $1 \cdot U_i^p = G_i^p$:

$$
\begin{pmatrix}
\vdots & & \vdots & \vdots & \vdots & & \vdots \\
0 & \ldots & 0 & 1 & 0 & \ldots & 0 \\
\vdots & & \vdots & \vdots & \vdots & & \vdots
\end{pmatrix}
\begin{pmatrix}
\vdots \\
U_i^p \\
\vdots
\end{pmatrix}
=
\begin{pmatrix}
\vdots \\
G_i^p \\
\vdots
\end{pmatrix}.
\tag{2.35}
$$

2.5.2 Neumann boundary conditions

With Neumann boundary conditions the total flow for the p-th equation can be specified. If the i-th global node of V^p is located at Γ_N^p, we test equation (2.3) with ϕ_i^p:

$$
\sum_{q=1}^{N} \int_{\Gamma_N^p} \left(A^{p,q} \nabla u^q \cdot \nu \right) \phi_i^p \, d\sigma = \int_{\Gamma_N^p} h^p \phi_i^p \, d\sigma.
\tag{2.36}
$$

The left hand side is just the sum over all boundary integrals that were omitted in Section 2.4. This sum is replaced by the right hand side, which is discretized using surface quadratures (see Section 3.5.1). Then it is added to F_i^p.

2.5.3 General boundary conditions

Beside Dirichlet and Neumann boundary conditions, it is also possible to implement any other kind of boundary conditions. Two examples that already are implemented in AMDiS are:

- **Robin boundary conditions:** Robin boundary conditions are a combination of Dirichlet and Neumann boundary conditions. They specify a linear combination of the solution and its normal derivative on the boundary. For a scalar problem, Robin boundary conditions are given by

$$
\alpha u + A \nabla u \cdot \nu = j \quad \text{on } \Gamma_R,
\tag{2.37}
$$

where α and j are real valued functions and $\Gamma_R \subset \partial\Omega$ is the Robin boundary. In weak formulation tested with ϕ_i and after bringing αu to the right hand side, the equation reads

$$
\int_{\Gamma_R} A \nabla u \cdot \nu \phi_i \, d\sigma = \int_{\Gamma_R} j \phi_i \, d\sigma - \int_{\Gamma_R} \alpha u \phi_i \, d\sigma \quad \text{on } \Gamma_R.
\tag{2.38}
$$

The integral on the left hand side is the boundary integral that was assumed to be zero in Section 2.4. Now it is replaced by the two right hand side integrals. The first integral is computed analog to the Neumann boundary integral described in Section 2.5.2. The second integral also is a boundary integral that can be computed using surface quadratures. Since it depends on u, it contributes to the matrix of the linear system of equations.

For systems of PDEs, Robin boundary conditions are defined by

$$\sum_{q=1}^{\mathcal{N}} \left(A^{p,q} \nabla u^q \cdot \nu \right) = j^p - \alpha^p u^p \quad \text{on } \Gamma_R^p \qquad p = 1, \ldots, \mathcal{N}. \tag{2.39}$$

- **Periodic boundary conditions:** Sometimes the solution is assumed to be periodically repeated in some directions. This is realized by using periodic boundary conditions. In AMDiS, there are two implementations of periodic boundary conditions.

  In the first implementation, the mesh topology is changed in such a way, that corresponding periodic boundaries are connected with each other. After this topological change, periodic boundaries don't have to be considered anymore. Geometric mesh information, like vertex coordinates, are kept unchanged.

  In the second implementation, the mesh topology keeps unchanged. Instead, periodic associations to corresponding periodic vertices are stored at vertices of the periodic boundary. After the linear system of equations has been assembled, these periodic associations are considered to modify the system of equations in a periodic sense.

  In Section 3.6.3, the implementation of periodic boundary conditions is described in more detail.

2.6   Time discretization

For time dependent problems, the solution u^p of equation p and all other functions that depend on x can also depend on the time t. An initial solution $u_0^p(x) := u^p(x, 0)$ must be given analytically or as solution of a stationary problem. In every equation p, the time derivative $\partial_t u^p(x, t)$ can appear which must be discretized at application level using e.g. the Runge-Kutta method or the fractional step Θ-scheme.

The control of the timestep size is done within the adaptation loop described in 4.1.

2.7   Linearization of non-linear problems

In principle, each of the $A^{p,q}$, $b^{p,q}$, $c^{p,q}$ or $f^{p,q}$ can depend on one or more solution components. Such non-linearities must be linearized globally before they can be discretized. One possibility is to apply a non-linear solving method, like the Newton method, within each iteration of the adaptation loop (see Section A.4.5). Another possibility e.g. for time dependent problems is to linearize the non-linear equation using the linear part of its Taylor series about the solution of the last timestep.

2.8   Discretization of higher order problems

In Section 2.4, the discretization of second order problems is described. However, many relevant problems are of higher order. One can often formulate such higher order problems as systems of second order PDEs, which then can be solved using the default discretization techniques.

A very simple example is the fourth order equation

$$\Delta^2 u = f \tag{2.40}$$

which can be written as

$$\Delta w = f \tag{2.41}$$
$$w - \Delta u = 0 \tag{2.42}$$

by substituting $w := \Delta u$. Some examples of more complex problems of higher order that have been solved with AMDiS are given in Sections 8.1.5 and 8.1.6.

Chapter 3

Implementation of finite element spaces

In this chapter, the implementation of finite element spaces including the assemblage of the corresponding linear system of equations is described. For the user of the AMDiS library, most of these aspects are usually hidden, because the problem discretization is done automatically.

The software design is often illustrated using the *Unified Modeling Language* (UML). In [20], a detailed overview over the UML is given. To obtain a flexible, extensible and reusable software structure, many object oriented design patterns are used in the AMDiS implementation, see [23] for a detailed description of these patterns.

3.1   Degrees of freedom

As mentioned in Chapter 2, a finite element space consists of a set of finite elements, a set of global basis functions, and a mapping from local to global basis functions. The set of elements is given by the mesh. The local basis functions on one element S are given by the basis functions defined on the reference element together with the mapping λ^S from world coordinates to barycentric coordinates of element S. One local basis function can be interpreted as the restriction of a global basis function to the domain of element S. The mapping from local to global basis functions is done by so called *degrees of freedom* or *DOF indices*.

A DOF index represents the index of a global basis function. Global basis functions are not explicitly stored, but are constructed from all local basis functions associated to the same global index. In Figure 3.1 (a), the vertex with global number 4 is shared by all elements. Thus, each element defines a local basis function on its local domain. In this example, it is the function with local index 2 for all elements. The global basis function with index 4 is the aggregation of all those local basis functions.

The mapping from local to global indices is done by storing the global indices at element nodes, which can be located at vertices, edges, faces or in the center of an element. Figure 3.1 (b) shows the local nodes of a triangle for Lagrange basis functions of degree 3 (see Section 3.4). To avoid redundant storage, each element stores only one pointer to an array, containing DOF indices, at each element position (vertex, edge, face, center). Two elements sharing e.g. one edge, store a pointer to the same array at the common edge. This array contains all DOF indices located at this edge. To be able to assign the global indices in the array to local element indices, the global indices have to satisfy a global order: on an edge the indices are sorted in ascending order from the vertex with the smaller global index to the vertex with the higher global index.

If more than one finite element space is defined on the mesh, the DOFs of the different spaces are stored successively in the arrays. First, all DOFs of space one at the given position, then all DOFs of space two at the same position, are stored, and so on. To distinguish the DOFs of the
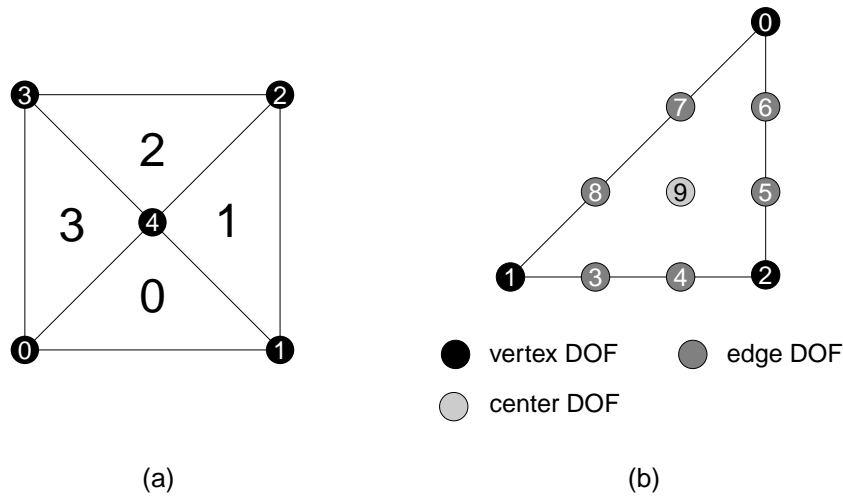
Figure 3.1: (a): A two dimensional macro mesh with vertex indices and element numbers, (b): A triangle with local node numbering for Lagrange basis functions of degree 3.

different finite element spaces, the offsets at the different positions for each finite element space must be stored (one integer vector of size $d + 1$ for each space).

Figure 3.2 (a) shows the global indices of Lagrange basis functions of degree 3 for the two dimensional example. In Figure 3.2 (b), one can see the index arrays and the corresponding pointers for two finite element spaces. The first is of degree 1, the second of degree 3. The first finite element space defines DOFs only at vertices, the second at vertices, edges and in element centers.

In the remainder of this section, we describe the main AMDiS concepts concerning DOF indices. Section 3.1.1 describes the dynamic administration of free and used DOF indices. In Sections 3.1.2 and 3.1.3, interfaces for objects which are indexed by DOF indices and for objects containing DOF indices are introduced. These interfaces provide a unified access to such objects through the whole code. Different classes implementing the same interface, can be treated in a unique way, and new classes can be added without any changes in the existing library code. An efficient way for iterating through DOF indexed objects is provided by DOF iterators, described in Section 3.1.4. The most important DOF indexed objects are DOF vectors and DOF matrices. These are the components of the global linear system of equations, that represents the discretization of the abstract problem (see Chapter 2). DOF vectors and matrices are topic of Section 3.1.5.

3.1.1 DOF administration

Every finite element space has its own DOF administration. This DOF administration manages the global DOF indices. If a new element is created, e.g. during adaptive refinement, new indices have to be created. When elements are deleted during a coarsening operation, some indices are no longer used. In addition, during the refinement process, DOF indices may be deleted: For higher order elements some indices of a parent element do not belong to any of the two child elements. If AMDiS is not told to preserve such coarse DOF indices, they are freed after refinement. Preservation of coarse DOFs is necessary if computations are performed not only at the finest possible mesh level. Examples are multigrid solvers (see Chapter 5) or the parallelization technique described in Chapter 6.

Thus, during the adaptation process (see Chapter 4) new DOF indices have to be created and others have to be freed. To keep the space of used DOF indices as small as possible, freed indices are reused in later allocation calls. For this purpose, the DOF administration contains a dynamic vector dofFree, that stores for every index, whether it is used or not. If DOF index i
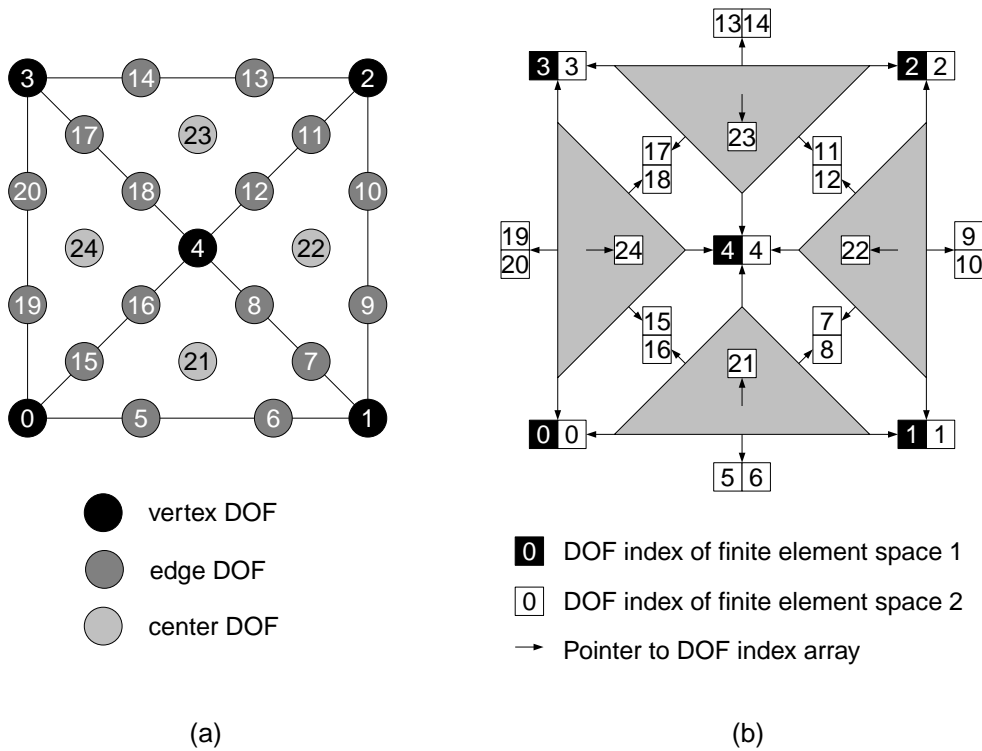
Figure 3.2: (a): Global DOF indices for Lagrange basis functions of degree 3. (b): DOF index arrays, stored on element nodes, for two finite element spaces on the same mesh. The first space is of degree 1, the second of degree 3.

is used, `dofFree[i]` is `false`. Figure 3.3 shows such a vector. The index `firstFree` points to

Figure 3.3: The vector stores used and free DOF indices.

the first unused index. This index is returned by the next index allocation. The index `lastUsed` points to the last used DOF index. This number determines the size of DOF indexed objects (see Section 3.1.2).

When a DOF index is freed or a new index is created, all objects that contain or are indexed by DOF indices, have to be informed, so that they can perform necessary reactions, like deleting, creating or adapting corresponding content. Furthermore, all objects indexed by DOF indices may have to be resized. For this reason, the DOF administration manages lists of all DOF indexed and DOF containing objects belonging to the same finite element space as the administration. The interaction with these objects is done through well defined interfaces described in Sections 3.1.2 and 3.1.3. Figure 3.4 shows the UML diagram of class `DOFAdmin`, which is the implementation of
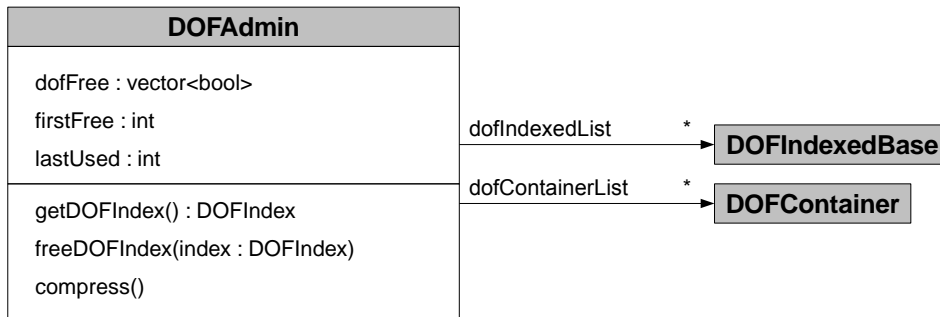
Figure 3.4: DOF administration class DOFAdmin

the DOF administration.

The member variables of class `DOFAdmin` are:

`dofIndexedList`: List of all objects belonging to the same finite element space that are indexed by DOF indices.

`dofContainerList`: List of all objects belonging to the same finite element space that contain DOF indices.

`dofFree, firstFree, lastUsed`: Member variables for management of free and used DOFs (described above).

The methods are:

`getDOFIndex()`: Returns the first DOF index, that is unused at the moment (`firstFree`) and marks the index as used.

`freeDOFIndex()`: Frees the given DOF index. The freed index may be reserved again by a future call of `getDOFIndex()`.

`compress()`: Eliminates all holes of unused DOF indices. The relative order of used indices keeps unchanged. After compression, all DOF indexed and DOF containing objects, that belong to the corresponding finite element space, are informed about the new index structure.

3.1.2   DOF indexed objects

The interface for DOF indexed objects is split into two parts. `DOFIndexedBase` is the template type independent base class of the templated sub class `DOFIndexed<ContentType>`. The template parameter determines, what content type is indexed by the DOF indices. The template type independent base class is used as an interface to other classes or modules that do not care about the content type of the object, or that manage different DOF indexed object types in one list. Examples are the DOF administration or the mesh adaptation module. The templated sub class is used as an interface for the unified content access.

In Figure 3.5 the UML class diagram for the two classes is shown. The methods of interface `DOFIndexedBase` are:

`getSize()`: Returns the current size of the object.

`resize()`: Resizes the object to the given value. Used by the DOF administration to adapt the object to the right size.

DOFIndexedBase

getSize() : int

resize(size : int)

compressDOFIndexed(newDOF : vector<DOFIndex>)

addDOFContent(index : DOFIndex)

freeDOFContent(index : DOFIndex)

refineInterpol(patch : RCPatch)

coarseRestrict(patch : RCPatch)

getFESpace() : FESpace*

DOFIndexed<ContentType>

operator[](index : DOFIndex) : ContentType&
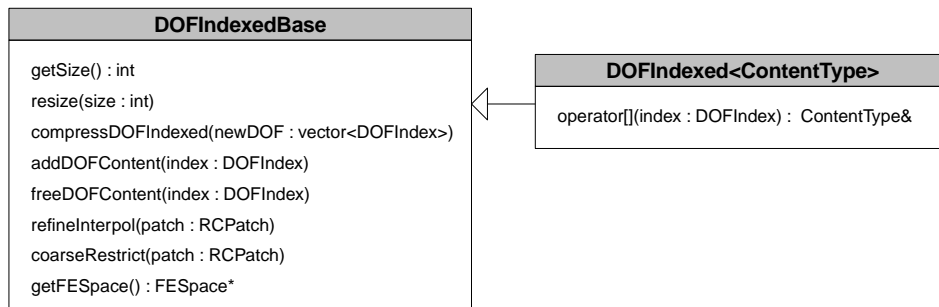
Figure 3.5: Interfaces for DOF indexed objects

DOFContainer

freeDOFIndex(index : DOFIndex)

compressDOFContainer(newDOF : vector<DOFIndex>)
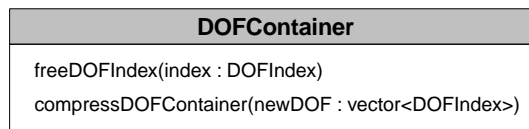
Figure 3.6: Class DOFContainer.

compressDOFIndexed(): Called by the DOF administration within its compress method. The DOF indexed object here performs all necessary operations to adapt to the new DOF indices. The argument newDOF is a vector storing the new DOF index for each old index.

addDOFContent(): Called by the DOF administration when a new DOF index was allocated. The DOF indexed object may allocate some corresponding content then. If a new DOF indexed object registers with an existing DOF administration, the method is called for each DOF index that is marked as used at this time.

freeDOFContent(): Called by the DOF administration after the given index was freed. The DOF indexed object may free some corresponding content then.

refineInterpol(), coarseRestrict: Called by the adaptation module after each mesh refinement and coarsening. The method determines, what effect the adapt operation has on the content. The argument patch contains the set of elements, involved in one refinement or one coarsening operation.

getFESpace(): Returns a pointer to the finite element space the object belongs to.

The interface DOFIndexed<ContentType> has only one method:

operator[]: This operator is used for the index based content access.

3.1.3 DOF containers

DOFContainer is the interface for objects that contain DOF indices. If indices are freed or changed, a DOF containing object is informed from the DOF administration through this interface. It is not task of the DOF administration to fill a DOF container, since it does not know which DOF indices to add and in which context the container is used. But if existing indices are changed or freed during compression, also the container entries must be adapted, to keep the container content consistent to the current DOF state. Figure 3.6 shows the UML diagram of class DOFContainer. Its methods are:

DOFIterator<ContentType>
mode : IterationMode
getDOFIndex() : DOFIndex
isDOFFree() : bool
reset()
end() : bool
operator++() : DOFIterator<ContentType>&
operator*() : ContentType&

admin → **DOFAdmin**

iteratedObject → **DOFIndexed<ContentType>**
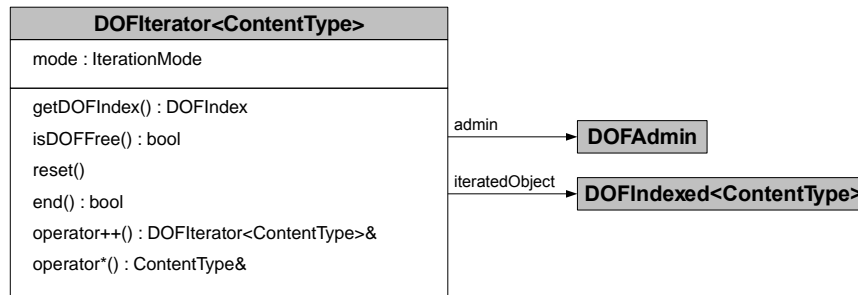
Figure 3.7: UML diagram of class DOFIterator

`freeDOFIndex()`: Called by the DOF administration after the given DOF index was freed.

`compressDOFContainer()`: Called by the DOF administration after a index compression was performed. The vector `newDOF` contains the new DOF indices indexed by the old indices.

3.1.4 DOF iterators

DOF iterators provide a unified and efficient way for traversing through DOF indexed objects. Each DOF iterator has access to the DOF indexed object that should be iterated and to the corresponding DOF administration. Depending on the value of member `mode` (`USED_DOFS`, `FREE_DOFS`, `ALL_DOFS`), the iterator can be used to visit only used DOF indices, only free DOF indices, or all DOF indices (free and used). Figure 3.7 shows the UML diagram of class `DOFIterator<ContentType>`. The template parameter `ContentType` specifies the content type of the DOF indexed object that should be iterated. The methods of class `DOFIterator<ContentType>` are:

`getDOFIndex()`: Returns the DOF index corresponding to the entry the iterator points to at the moment.

`isDOFFree()`: Returns, whether the current DOF index is used or free. Useful in mode `ALL_DOFS`.

`reset()`: Resets the iterator. After this method is called, the iterator points to the first relevant entry for the iterator's iteration mode.

`end()`: Returns `true` if the iterator points behind the last relevant entry for the iteration mode. Otherwise, it returns `false`.

`operator++()`: Increments the iterator to the next relevant entry for the iteration mode.

`operator*()`: Dereferencing operator. Returns a reference to the value the iterator points to at the moment.

The following example shows how a DOF indexed object is traversed. Each entry corresponding to a used DOF is incremented by 1. `obj` here is a pointer to a `DOFIndexed<double>` object. Note that the DOF administration is implicitly passed to the iterator by the object, which knows its finite element space and therefore, the corresponding DOF administration, too.

```
DOFIterator<double> dofIt(obj, USED_DOFS);
for(dofIt.reset(); dofIt.end() == false; ++dofIt) {
  *dofIt += 1.0;
}
```
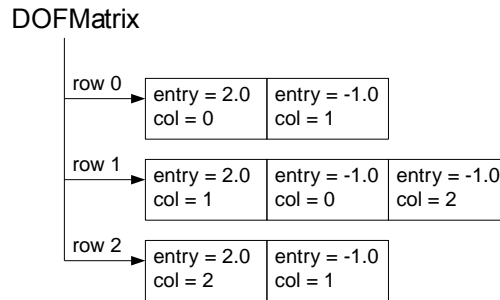
Figure 3.8: Example of a DOF matrix.

3.1.5 DOF vectors and DOF matrices

DOF vectors and DOF matrices are the components of the linear system of equations. The class `DOFVector<ContentType>` is an implementation of the interface `DOFIndexed<ContentType>`.

DOF matrices are stored in sparse form. Only non-zero entries are stored. In AMDiS, a variation of the *Compressed Row Storage* scheme (CRS) is used. In the CRS format, three arrays are stored: `entry`, `column` and `offset`. The array `entry` stores all non-zero entries of the matrix consecutively. The array `column` has the same size and stores the corresponding columns of each entry. The entry `offset` has an entry for each row of the matrix, storing row offsets for the first two arrays. The matrix

$$\begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix}$$

in the CRS format looks like:

```
entry  : 2 -1 -1  2 -1 -1  2
column : 0  1  0  1  2  1  2
offset : 0  2  5.
```

In AMDiS, due to adaptivity, a more flexible scheme is needed. Here each matrix row is stored as a dynamic STL vector of matrix entries. One matrix entry consists of the actual matrix entry and the corresponding column number. A DOF matrix is stored as a vector of such matrix rows. Since the diagonal elements of a matrix often plays a special role (e.g. in pre-conditioning within the solving process), they are stored always as first entries in the corresponding vectors. In Figure 3.8 the above matrix is shown like it would be stored in AMDiS.

The column entry can have two special entries: `UNUSED_ENTRY` and `NO_MORE_ENTRIES`, represented by the negative numbers -1 and -2. `UNUSED_ENTRY` marks an entry, which is not used anymore, because the corresponding DOF index was freed. `NO_MORE_ENTRIES` marks an unused entry, which is not followed by any used entry in this row.

Conceptually, each DOF matrix row could be seen as a DOF container and the vector of rows as a DOF indexed object. However, for efficiency reasons the class `DOFMatrix` only implements the `DOFIndexed` interface. The methods `freeDOFIndex` and `compressDOFContainer` of the `DOFContainer` interface are implicitly implemented within the implementations of the methods `compressDOFIndexed` and `freeDOFContent`.

3.2   Hierarchical mesh structure

In AMDiS, meshes are stored in a hierarchical way. The macro mesh is represented by a list of macro elements which stores geometrical as well as topological information about the elements. If an element is refined (bisected), the two new elements are stored as children of the bisected element. In this way a binary tree arises.

To store the whole hierarchical structure, needs more memory than storing only the elements of the fine mesh, but it has the following advantages:

- Much element information can be constructed from macro elements during mesh traversal. Therefore, it does not have to be stored for each element.

- Coarsening can be implemented as the inverse operation of refinement just by deleting the corresponding child elements.

- The hierarchical structure can be used in order to implement fast multigrid solvers (see Chapter 5).

In the following sections, the classes that are used for the mesh storage are described.

3.2.1   MacroElement

A `MacroElement` object stores all geometrical and topological information for one macro element. These are:

- A unique index of the macro element.

- A pointer to the corresponding `Element` object, which is the root of the binary tree.

- The world coordinates of each macro element vertex.

- The boundary condition identifiers for each vertex, edge or face that is part of the domain boundary. These numbers are used to assign the boundary condition to the corresponding parts of the elements.

- Neighbor information: For each side of a macro element (vertices in 1d, edges in 2d, faces in 3d), a pointer to the neighbor element at this side is stored (`NULL`, if it is a boundary side). Furthermore, the local index of the vertex in the neighbor element is stored, that lies opposite to the common side (*oppVertex*). This information is used for mesh adaptation.

- Projections: Can be used to automatically map the vertex coordinates in a given way. Boundary projections are applied to the boundary vertices, volume projections are applied to all vertices of the macro element (and all corresponding child elements, see Section A.4.8).

3.2.2   Element

`Element` is the abstract base class for all elements. In AMDiS, the three concrete element classes `Line`, `Triangle` and `Tetrahedron` are implemented. Every element stores:

- A unique element index.

- Two pointers to the two child elements if the element is refined. If the element is not refined, these pointers are set to `NULL`.

- A pointer to an array of DOF indices for each position (for each vertex, edge, face and for the element center, see Section 3.1).

- An integer value which can store markings for refinement and coarsening.

- Pointer to an `ElementData` object where additional element data can be stored (e.g. the local error estimation for leaf elements). The pointer can be `NULL`.

- A Tetrahedron has an additional element type number, which is used during refinement to assign the children's vertex numbers.

Concrete element types must provide:

- A `clone()` method, which returns a copy of the element. This method is used to implement the prototype design pattern: The mesh manages an element prototype which can be cloned to create new elements during refinement.

- Topological information about the element type (e.g. the index of the side of the i-th child that corresponds to side j of the parent element).

3.2.3 ElementData

The class `ElementData` is the abstract base class for all data that can be dynamically assigned to elements. Element data can be assigned to every element in the binary tree, not only to leaf elements. Every implementation of `ElementData` must provide:

- A `clone()` method, which is used by the mesh to dynamically create `ElementData` objects for new elements.

- A `refineElementData()` method. This is used to determine how the element data are refined to the new child elements. The method returns a boolean value that indicates whether the parent `ElementData` object has to be deleted after refinement. If it is deleted, a new `ElementData` object must be created by the `clone()` method when the mesh is coarsened.

- A `coarsenElementData()` method, which determines how the element data are coarsened.

To be able to use different types of element data at the same time, every `ElementData` object can be decorated by another `ElementData` object, which in turn can be decorated (and so on). Therefore, every element can be provided with an arbitrary combination of element data dynamically at runtime. The accessibility of element data is assured by a chain of responsibility.

An example for element data are the local error estimations, which have to be stored only at leaf elements.

3.2.4 Mesh

A `Mesh` object manages the complete hierarchical mesh structure. It stores:

- The dimension of the mesh (can be different from the dimension of the world, see Section 3.7).

- The list of macro elements, that represent the macro mesh.

- An element prototype, which can be cloned to produce new elements during refinement.

- An element data prototype, which is used to create the default element data for new elements.

- A parametrizer (optional), that adds parametric information to `ElInfo` objects during mesh traversal (see Section 3.7).

In Figure 3.9, an instance diagram of a simple example mesh is shown.
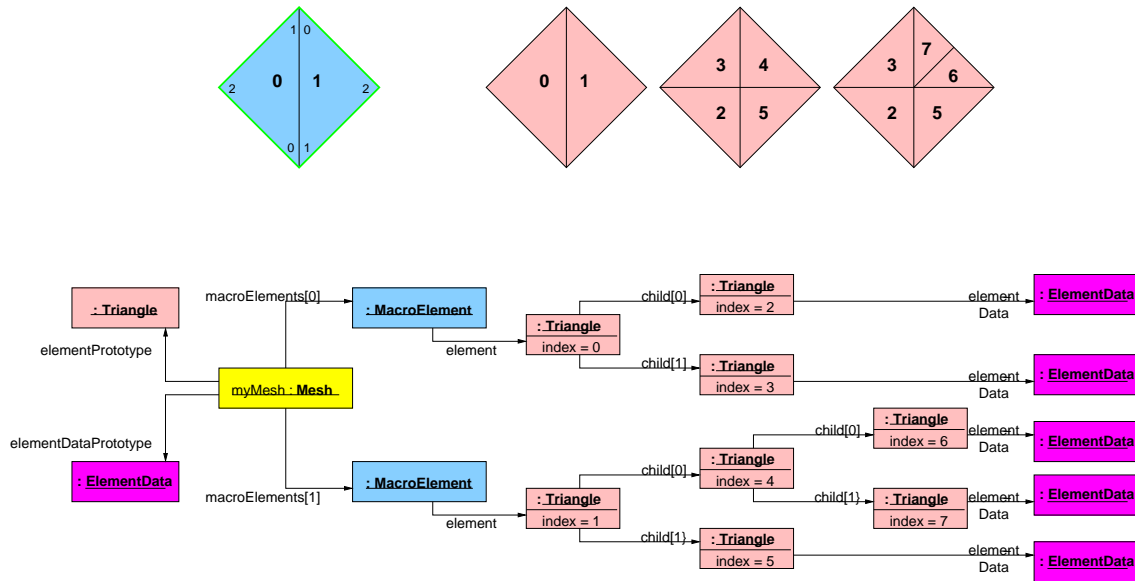
Figure 3.9: Instance diagram of an adaptively refined two dimensional mesh.

3.3 Mesh traversal

This section describes how iteration over mesh elements is done in AMDiS. This iteration is called *mesh traversal*. As mentioned in Section 3.2, element information that can be computed during mesh traversal, are not stored in the elements. Such information is constructed based on the information stored at macro elements and the current position in the binary tree. Then it is written to `ElInfo` objects, which are returned to the traverse client. Each `ElInfo` object represents one mesh element.

Before the traversal starts, one has to determine

- which elements are called in which order and

- which element information should be constructed during traversal.

This information is given by the *traverse flag*, which can contain the following values:

- `CALL_EVERY_EL_PREORDER`: Every element is called in pre-order (parent-$child_1$-$child_2$).

- `CALL_EVERY_EL_INORDER`: Every element is called in in-order ($child_1$-parent-$child_2$).

- `CALL_EVERY_EL_POSTORDER`: Every element is called in post-order ($child_1$-$child_2$-parent).

- `CALL_LEAF_EL`: Every leaf element is called.

- `CALL_LEAF_EL_LEVEL`: Every leaf element of a given level is called.

- `CALL_EL_LEVEL`: Every element of a given level is called.

- `FILL_COORDS`: Add vertex coordinates to the `ElInfo` object.

- `FILL_BOUND`: Add boundary information to the `ElInfo` object.

- `FILL_NEIGH`: Add neighbor information to the `ElInfo` object.

- `FILL_OPP_COORDS`: Add coordinates of the *opposite vertices* to the `ElInfo` object (see Section 3.2.1).

- `FILL_DET`: Compute and add $\det DF_S$ to the `ElInfo` object.

- `FILL_GRD_LAMBDA`: Compute and add Λ to the `ElInfo` object.

Since every of these values represents one bit in the flag, multiple entries can be set in the same traverse flag. However, if more than one `CALL`-flag is set, this results in an error, because the general traverse type must be unique.

The following information is filled into the `ElInfo` object in every traverse independent of the traverse flag:

- Pointer to the corresponding `Element` object.

- Level of the element in the binary tree (elements at macro level have level 0).

- Pointer to the corresponding `MacroElement` object.

- Pointer to the parent `Element` object.

3.3.1 Traverse stack

In principle, there are two possibilities to implement the mesh traversal: recursively or iteratively. In AMDiS, both methods are implemented. But in most cases, the iterative implementation is the better choice because it is more flexible and easier to use. The following listing shows an example of iterative mesh traversal using a traverse stack.

```
TraverseStack stack;
ElInfo *elInfo = stack.traverseFirst(mesh, level, flag);
while(elInfo) {

  // do work for current element represented by elInfo

  elInfo = stack.traverseNext(elInfo);
}
```

The traverse stack stores the current traversal state (see below). The method `traverseFirst()` returns the first `ElInfo` object of this traversal. The first argument is a pointer to the mesh that should be traversed. The second argument is the level number which should be used for level traverses (e.g. `CALL_EL_LEVEL`). It is ignored if no level traverse is used. The third argument is the traverse flag. If e.g. every leaf element should be visited and only vertex coordinates should be computed the flag would be `flag = Mesh::CALL_LEAF_EL | Mesh::FILL_COORDS`. The prefix `Mesh::` is necessary because the flags are defined in the scope of class `Mesh`. The flags are combined by the bit-wise or-operator (`|`), which ensures that both bits are set in the resulting flag.

The method `traverseNext()` returns the next `ElInfo` object for the current traversal. The current `ElInfo` is given as argument to `traverseNext()`. If the traverse is finished, `traverseNext()` returns a `NULL` pointer. The traversal now can be done as a while-loop. In the last statement of the loop body, `traverseNext()` is called to get the new `ElInfo` object. The loop terminates, after a `NULL` pointer has been returned.

Each entry of the traverse stack stores one `ElInfo` object and one counter. The counter is used in order to count the number of visitations of the corresponding element. If the counter is 0, the element is visited for the first time. If it is 1, the element is visited for the second time which means that the sub tree of the first child has already been processed. If the counter is 2, the element is visited for the third time which means that the sub trees of both children have been processed. In Algorithm 3, pseudo code for the method `traverseFirst()` is shown. To simplify matters, level-traverses are left out here.

The traverse type (in-order, pre-order, post-order, leaf) is considered in the `returnElement()` algorithm shown in Algorithm 5. In `createMacroInfo()`, an `ElInfo` object is created for the given macro element.

Algorithm 3 traverseFirst(traverseMesh, traverseFlag)

 1: mesh = traverseMesh
 2: flag = traverseFlag
 3: stack.clear()
 4: currentMacro = 0
 5: elInfo = createMacroInfo(mesh.macro[0], flag)
 6: stack.push(elInfo, 0)
 7: **if** returnElement(elInfo, counter, flag) **then**
 8:    **return** elInfo;
 9: **else**
10:    **return** traverseNext(elInfo);
11: **end if**

Algorithm 4 shows the pseudo code implementation of `traverseNext()`. The method `createElInfo()` creates and fills the `ElInfo` object for the i-th child of the element corresponding to the current `ElInfo`. Depending on the counter on the top of the stack, the first or the second child is visited next.

Algorithm 4 traverseNext(elInfo)

 1: **while not** stack.isEmpty() **do**
 2:    **if** (stack.top.counter == 2) **or** stack.top.elInfo.element.isLeaf() **then**
 3:       stack.pop()
 4:    **else**
 5:       nextElInfo = createElInfo(stack.top.counter, elInfo)
 6:       elInfo = nextElInfo
 7:       stack.top.counter += 1
 8:       stack.push(elInfo, 0)
 9:    **end if**
10:    **if** stack.isEmpty() **and** currentMacro $<$ mesh.macro.size() - 1 **then**
11:       currentMacro += 1
12:       nextElInfo = createMacroInfo(mesh.macro[currentMacro], flag)
13:       elInfo = nextElInfo
14:       stack.push(elInfo, 0)
15:    **end if**
16:    **if** returnElement(elInfo, counter, flag) **then**
17:       **return** stack.top.elInfo
18:    **end if**
19: **end while**
20: **return** NULL

In Figure 3.10, the binary tree for a simple example mesh with only one macro element is given. Table 3.1 shows the different steps of the mesh traversal and the corresponding states of the stack (`ElInfo` objects here are represented by the corresponding element indices). Furthermore, one can see which element is returned in which step depending on the traversal type.

3.3.2   Neighbor traverse

Within the body of a normal mesh traversal, a neighbor traverse can be performed if information about neighbor elements (or neighbors of neighbors) is needed. The statement

```
elInfo = stack.traverseNeighbor(elInfo, i);
```
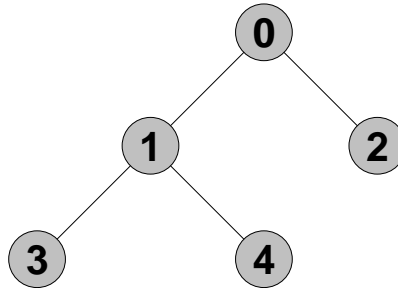
Figure 3.10: Example binary tree of a mesh with one macro element.

					pre-order	in-order	post-order	leaf
step 1	ElInfo	0			0			
	counter	0						
step 2	ElInfo	0	1		1			
	counter	1	0					
step 3	ElInfo	0	1	3	3	3	3	3
	counter	1	1	0				
step 4	ElInfo	0	1			1		
	counter	1	1					
step 5	ElInfo	0	1	4	4	4	4	4
	counter	1	2	0				
step 6	ElInfo	0	1				1	
	counter	1	2					
step 7	ElInfo	0				0		
	counter	1						
step 8	ElInfo	0	2		2	2	2	2
	counter	2	0					
step 9	ElInfo	0					0	
	counter	2						

Table 3.1: Traverse steps for the binary tree shown in Figure 3.10. For every traverse step one can see the states of the traverse stack (element numbers are shown instead of the stored ElInfo objects) and the element numbers for which an ElInfo object is returned for the different traverse types.

Algorithm 5 returnElement(elInfo, counter, flag)

1: **if** flag.isSet(CALL_EVERY_EL_PREORDER) **then**
2:     **return** (counter == 0)
3: **end if**
4: **if** flag.isSet(CALL_EVERY_EL_INORDER) **then**
5:     **return** (counter == 1)
6: **end if**
7: **if** flag.isSet(CALL_EVERY_EL_POSTORDER) **then**
8:     **return** (counter == 2)
9: **end if**
10: **return** elInfo.element.isLeaf()

---

returns an `ElInfo` object for the neighbor located at side `i` (vertex i in 1d, edge i in 2d, face i in 3d) of the current element. Two conditions must be fulfilled for neighbor traversal:

- It can be performed only on leaf level.

- Before `traverseNext()` is called the next time, the original element (corresponding to `elInfo` before the first call of `traverseNeighbor()`) must be reached again.

3.3.3   Dual traverse

The concept of dual traverses allows it to traverse two different meshes simultaneously. Both meshes must have the same macro mesh, but they can be refined independently of each other. One application could be the independent refinement for the different finite element spaces of a system of PDEs (not yet implemented in AMDiS).

The two meshes can also be defined by the same mesh object but on different levels. Necessary condition for the dual traverse is that the elements of both *virtual* meshes build a triangulation of the same domain. Such a dual traverse is used e.g. for the partition of unity method in parallel computations (see Chapter 6). Here, the partition of unity is computed on the leaf level considering the basis functions of the coarser partitioning level.

Figure 3.11 shows the two application cases of the dual traverse: In Figure 3.11 (a), two meshes with the same macro triangulation but different refinements are traversed simultaneously. In Figure 3.11 (b), both meshes are defined by the same object. The first mesh consists of all elements of level 1, the second mesh of all leaf elements.

Figure 3.11: (a): Dual traverse of two independent refinements of the same macro mesh, (b): Dual traverse of two different virtual meshes: the first mesh defined by all elements of level 1 and the second mesh defined by all leaf elements.

The following listing shows how a dual traverse is done in AMDiS.

```
DualTraverse dualStack;
```
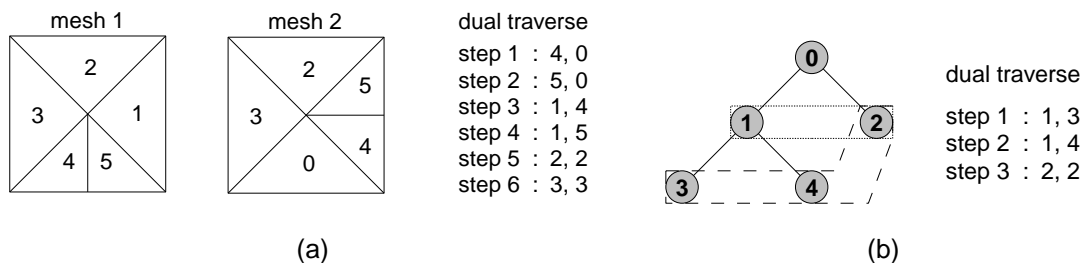
```
ElInfo *elInfo1, *elInfo2;
ElInfo *elInfoSmall, *elInfoLarge;
bool cont = dualStack.traverseFirst(mesh1, mesh2,
                                    level1, level2,
                                    flag1, flag2,
                                    &elInfo1, &elInfo2,
                                    &elInfoSmall, &elInfoLarge);
while(cont) {

  // do work

  cont = dualStack.traverseNext(&elInfo1, &elInfo2,
                                &elInfoSmall, &elInfoLarge);
}
```

The class `DualTraverse` manages two standard traverse stacks which are created in the `traverseFirst()` method. The arguments are pointers to the two meshes (can be pointers to the same mesh), the two traverse levels (ignored, if no level traverse is used), the two traverse flags, and four pointers to pointers to `ElInfo` objects (it must be pointers to pointers, because the pointers to the `ElInfo` objects are changed within the method). The first two pointers are the addresses of the pointers to the current `ElInfo` objects for mesh 1 and mesh 2. The third and the fourth pointers are used to tell the traverse client, which of the corresponding elements is the smaller (on higher mesh level) and which the larger one (on lower mesh level). The method returns `true` if the traverse has not finished, yet, and `false` otherwise.

The arguments of `traverseNext()` have similar meanings as in `traverseFirst()`. But here, they must contain the values that were returned by the last call of `traverseFirst()` or `traverseNext()`. The return value has the same meaning as in `traverseFirst()`.

3.4 Basis functions

In AMDiS, the local basis functions are defined on the reference simplex $\bar{S}$ in barycentric coordinates (see Sections 2.1 and 2.2). To construct a set of local basis functions, one has to define for each function $\bar{\phi}_i$ the barycentric coordinates λ^i of the node where it is located at, and the mapping $\bar{\phi}_i : \bar{S} \to \mathbb{R}$.

Together with the parameterization F_S, the definition of the basis functions on the reference element implicitly defines the local basis functions for every mesh element S. Since the assemblage for systems of PDEs can be reduced to the scalar case (see Section 2.4.2), no specialized basis functions are needed for vector valued problems.

Every basis function $\bar{\phi}_i$ should fulfill the following properties:

- $\bar{\phi}_i(\lambda^i) = 1$,

- $\bar{\phi}_i(\lambda^j) = 0 \quad \forall j = 1, \ldots, n, \ j \neq i$ where n is the number of local basis functions on $\bar{S}$.

These properties assure that the value of the solution vector at a given index is equal to the finite element solution at the corresponding node coordinates, and that Dirichlet boundary conditions can be easily discretized as described in Section 2.4.

So far in AMDiS, Lagrange basis functions of degree 1 - 4 have been implemented. The construction of the Lagrange polynomials is described in the next section.

3.4.1 Construction of Lagrange basis functions

In the following, Lagrange basis functions for dimension d and of degree k are constructed. First, we define the barycentric node coordinates. For every node λ^i
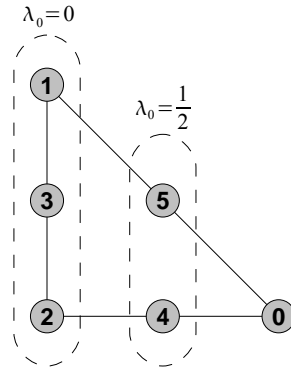
Figure 3.12: Nodes of Lagrange basis functions of order 2 in 2d shown on the standard simplex $\hat{S}$. For nodes 1, 2 and 3 the first component of the corresponding barycentric coordinates is 0. For nodes 4 and 5 it is $\frac{1}{2}$.

λ^0	=	(	1	,	0	,	0	)
λ^1	=	(	0	,	1	,	0	)
λ^2	=	(	0	,	0	,	1	)
λ^3	=	(	0	,	$\frac{1}{2}$	,	$\frac{1}{2}$	)
λ^4	=	(	$\frac{1}{2}$	,	0	,	$\frac{1}{2}$	)
λ^5	=	(	$\frac{1}{2}$	,	$\frac{1}{2}$	,	0	)

Table 3.2: Barycentric node coordinates of quadratic Lagrange basis functions in 2d.

- every coordinate component λ_j^i must satisfy $0 \leq \lambda_j^i \leq 1$,

- every coordinate component must be a multiple of $1/k$ and

- the sum over all coordinate components must be 1.

The set of all distinct coordinates that fulfill these properties are the nodes of the Lagrange basis functions. In Table 3.2, the node coordinates of quadratic Lagrange functions in 2d are given. In Figure 3.12, the nodes are visualized on the standard element $\hat{S}$.

Now, exemplarily the construction of the corresponding Lagrange polynomial $\bar{\phi}_0(\lambda)$ is demonstrated. The polynomial must be 0 at all nodes λ^i, $i \neq 0$. We start with the polynomial λ_0 (first component of λ). This polynomial is 0 at all nodes λ^i where $\lambda_0^i = 0$ (nodes 1, 2 and 3). Now, we multiply it by $\lambda_0 - \frac{1}{2}$. The resulting polynomial $\lambda_0(\lambda_0 - \frac{1}{2})$ is also zero at all nodes where $\lambda_0^i = \frac{1}{2}$ (nodes 4 and 5) and it is equal to $\frac{1}{2}$ at λ^0. After multiplication with two, we have obtained the final polynomial with the desired properties.

In principle, the construction of Lagrange basis function could be automated. To avoid construction every time a function is evaluated, the basis functions should be constructed once and stored in a symbolic way. The evaluation of such constructed basis functions would be slower than hard coded basis functions, but it would allow basis functions of arbitrary degrees in arbitrary dimensions.

So far, the construction of basis function is not automated in AMDiS. This may be content of future work.

3.5 Assemblage of the linear system of equations

The abstract problem definition, including Neumann boundary conditions, is done by defining operators. These operators are used to fill (assemble) the left hand side matrix and the right hand

side vector of the linear system of equations.

In AMDiS, operators can be defined by the user at runtime on a very high abstraction level, that directly reflects the corresponding PDE. For every operator, one assembler will be constructed automatically, optimized to the properties of its operator. The assembler computes the contribution of the operator to the system of equations. Numerical issues are hidden to the user as far as possible.

The strict separation of operators and assemblers leads to a strict separation of the problem definition and the discretization in given circumstances (triangulation, finite element spaces, ...). This in turn leads to a high reusability.

3.5.1   Numerical integration

Integration in AMDiS is done using numerical quadratures. Since basis functions are defined in barycentric coordinates on the reference simplex, also quadratures are defined on the reference simplex. In this section, different ways for the numerical integration are introduced which are all based on the quadrature principle.

Quadratures

Quadratures are used for the numerical computation of integrals. If $\int_\Omega f(x)dx$ has to be calculated, it can be approximated by a sum $\sum_{i=1}^n w_i f(x_i)$. The accuracy of this approximation depends on the number n of quadrature points and the choice of the weights w_i and the coordinates x_i. The quadrature is defined as

$$Q_\Omega := \{(w_i, x_i); \ \ i = 1, \ldots, n\}. \tag{3.1}$$

If the function f fulfills certain properties, one can construct quadratures that provide an exact computation of the integral.

In the context of finite elements, basis functions are usually involved in the integrals, like

$$\int_\Omega f(x)\phi(x)dx, \tag{3.2}$$

where ϕ is a global basis function.

The integration is done element-wise and the local basis functions are defined in barycentric coordinates. If S is the current element and $\bar\phi$ is the local basis function corresponding to ϕ, we can write the integral as

$$\int_S f(x)\bar\phi(\lambda^S(x))dx. \tag{3.3}$$

After transformation to the standard simplex the integral reads

$$\int_{\hat S} f(F_S(\hat x))\bar\phi(\lambda(\hat x))|DF_S|d\hat x. \tag{3.4}$$

If F_S is an affine mapping ($|DF_S|$ than is constant), we can extract $|DF_S|$ from the integral and approximate equation (3.4) by

$$|DF_s| \sum_{i=1}^n w_i f(F_S(\hat x_i))\bar\phi(\lambda(\hat x_i)). \tag{3.5}$$

Since the local basis functions are given in barycentric coordinates, we also define the quadrature points in barycentric coordinates:

$$\hat Q_{\hat S} := \{(w_i, \lambda_i); \ i = 1, \ldots, n\}, \tag{3.6}$$
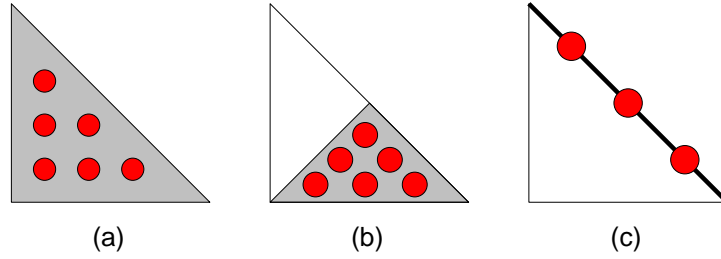
Figure 3.13: The different quadrature types: Standard quadrature (a), scaled quadrature for integration over a sub triangle (b), surface quadrature for integration over an edge (c).

with $\lambda_i := \lambda(x_i)$. This leads to the approximation

$$|DF_s| \sum_{i=1}^{n} w_i f(x^S(\lambda_i)) \bar{\phi}(\lambda_i). \tag{3.7}$$

Generally spoken, the quadrature $\hat{Q}_{\hat{S}}$ like defined in equation (3.6) approximates the integral $\int_{\hat{S}} f(\hat{x}) d\hat{x}$ by $\sum_{i=1}^{n} w_i f(x^S(\lambda_i))$.

Pre-calculated basis functions

The local basis functions are defined in barycentric coordinates on the reference simplex $\bar{S}$. For a given quadrature the local basis functions have to be evaluated at the same points for every element S. Thus, every basis function is calculated once at each quadrature point and the result is written into an array. The array access is much faster than evaluating the basis function every time it is used. A quadrature that additionally stores the values of all basis functions at each quadrature point is called a *fast quadrature*.

If derivatives of the basis function are part of the integral, they can be pre-calculated and stored in the same way.

Pre-calculated integrals

If the function f of the above example is piecewise constant for every element, this constant (f^S) can be extracted from the integral:

$$\int_{\hat{S}} f(F_S(\hat{x})) \bar{\phi}(\lambda(\hat{x})) |DF_S| d\hat{x} = f^S |DF_S| \int_{\hat{S}} \bar{\phi}(\lambda(\hat{x})) d\hat{x}. \tag{3.8}$$

The integral now no longer depends on the element S and can be pre-calculated. In principle, the same holds for integrals containing derivatives of ϕ.

Scalable quadratures and surface quadratures

In composite finite elements (see Section 3.8), where simplices can be cut by implicit described surfaces, integration only over a part of an element is needed. Furthermore, if meshes for different solution components can be refined independently from each other (may be implemented in future AMDiS versions), integration over sub elements becomes necessary. This can be done by scalable quadratures which scale the coordinates λ_i to fit to the sub triangle.

If surface integrals should be computed, e.g. for Neumann boundary conditions or for implicit boundaries using composite finite elements, the integration can be calculated by a lower dimensional surface quadrature with quadrature points located at the surface of the element.

Note, that fast quadratures and pre-calculated integrals can not be used together with scalable or surface quadratures, because here the quadrature points are not fixed on the reference simplex which would be necessary for the optimizations.

In Figure 3.13, quadrature points of the different quadrature types are visualized on the standard simplex $\hat{S}$ in 2d.

3.5.2 Element matrices and element vectors

The assemblage of the linear system of equations is done element-wise. Element matrices and vectors are used as data capsule to store element contributions to the global system matrix and to the global right hand side vector. If an operator is evaluated several times for the same element, it can store its own element matrix and element vector to avoid redundant computations.

Element matrix

An element matrix instance contains

- m: a real valued $n^p \times n^q$ matrix containing the contributions of the current element to the global matrix,

- r: a vector of size n^p containing the global row indices corresponding to the local row indices on this element matrix and

- c: a vector of size n^q containing the global column indices corresponding to the local column indices on this element matrix.

Element vector

An element vector instance contains

- v: a real valued vector of size n^p containing the contributions of the current element to the global right hand side vector and

- r: a vector of size n^p containing the global indices corresponding to the local indices on this element vector.

3.5.3 Operators and assemblers

In Section 2.4, we have seen that the discretization of second order equations leads to integrals of the following form:

$$\text{SOT} = \int_{\hat{S}} \nabla_\lambda \bar{\phi}_k^p(\lambda(\hat{x})) \cdot \Lambda(\hat{x}) A(F_S(\hat{x})) \Lambda^t(\hat{x}) \nabla_\lambda \bar{\phi}_l^q(\lambda(\hat{x})) |DF_S(\hat{x})| \, d\hat{x}, \tag{3.9}$$

$$\text{FOT}_{01} = \int_{\hat{S}} \bar{\phi}_k^p(\lambda(\hat{x})) \Lambda(\hat{x}) b(F_S(\hat{x})) \cdot \nabla_\lambda \bar{\phi}_l^q(\lambda(\hat{x})) |DF_S(\hat{x})| \, d\hat{x}, \tag{3.10}$$

$$\text{ZOT} = \int_{\hat{S}} c(F_S(\hat{x})) \bar{\phi}_k^p(\lambda(\hat{x})) \bar{\phi}_l^q(\lambda(\hat{x})) |DF_S(\hat{x})| \, d\hat{x} \quad \text{and} \tag{3.11}$$

$$\text{RHS} = \int_{\hat{S}} f(F_S(\hat{x})) \bar{\phi}_k^p(\lambda(\hat{x})) |DF_S(\hat{x})| \, d\hat{x}. \tag{3.12}$$

The integrals are computed for $k = 1, \ldots, n^p$ and $l = 1, \ldots, n^q$ for every element $S \in \mathcal{T}$. The results are added to the system matrix entry $M_{\mathcal{G}_S^{V^p}(k), \mathcal{G}_S^{V^q}(l)}^{p,q}$ or to the right hand side vector entry $F_{\mathcal{G}_S^{V^p}(k)}^p$. SOT stands for second order term, FOT for first order term, ZOT for zero order term

and RHS for right hand side term. FOT_{01} denotes first order terms where the basis function $\bar{\phi}_l^q$ is derived. In principle, also FOT_{10} can appear, where the basis function $\bar{\phi}_k^p$ is derived:

$$FOT_{10} \quad = \quad \int_{\hat{S}} \nabla_\lambda \bar{\phi}_k^p(\lambda(\hat{x})) \cdot \Lambda(\hat{x}) b(F_S(\hat{x})) \bar{\phi}_l^q(\lambda(\hat{x})) |DF_S(\hat{x})| \, d\hat{x}. \tag{3.13}$$

This happens e.g. if integration by parts is applied to a first order term. Right hand side terms can be realized by applying a zero order term to the right hand side vector instead of the system matrix.

If the finite element spaces are given, the above integrals are uniquely determined by the definitions of A, b, c and f. Hence, an intuitive design is, that A, b, c and f are defined by operator terms. Each operator contains one list of second order terms, one for each type of first order terms and one for zero order terms. The operator terms then are used by assemblers to compute the integrals for the given discretization. Every operator has one assembler for its second order terms, one for each types of its first order terms and one for its zero order terms.

In AMDiS, the separation between operators and assemblers is slightly different:

- Second order terms compute not only $A(F_S(\hat{x}))$ but $\Lambda(\hat{x}) A(F_S(\hat{x})) \Lambda^t(\hat{x})$.

- First order terms compute not only $b(F_S(\hat{x}))$ but $\Lambda(\hat{x}) b(F_S(\hat{x}))$.

The reason for that is, that the computation of $\Lambda(\hat{x}) A(F_S(\hat{x})) \Lambda^t(\hat{x})$ and $\Lambda(\hat{x}) b(F_S(\hat{x}))$ then can be optimized within the operator terms. Assume e.g. that A is equal to the identity, like it is the case for the Laplace operator. Then the operator term has only to compute $\Lambda(\hat{x}) \Lambda^t(\hat{x})$.

The assemblage of the DOF matrix $M^{p,q}$ can be written on a high abstraction level as shown in Algorithm 6.

Algorithm 6 assemble DOF matrix $M^{p,q}$

1: set $M_{i,j}^{p,q} = 0$ for $i = 1, \ldots, N^p$ and $j = 1, \ldots, N^q$
2: **for** all elements S in $\mathcal{T}$ **do**
3:    initialize element matrix m^S
4:    **for** all operators $\mathcal{O}$ of $M^{p,q}$ **do**
5:       add contribution of operator $\mathcal{O}$ for element S to m^S
6:    **end for**
7:    add element matrix m^S to global matrix $M^{p,q}$
8: **end for**

The initialization of the element matrix in line 3 is described in more detail in Algorithm 7. Here, the matrix entries are reseted and the global row and column indices are set for the current element.

Algorithm 7 initialize element matrix m^S

1: set $m^S.m_{k,l} = 0$ for $k = 1, \ldots, n^p$ and $l = 1, \ldots, n^q$
2: set $m^S.r_k = \mathcal{G}_S^{V^p}(k)$ for $k = 1, \ldots, n^p$
3: set $m^S.c_l = \mathcal{G}_S^{V^q}(l)$ for $l = 1, \ldots, n^q$

After the element matrix has been computed, it is added to the DOF matrix (line 7 of Algorithm 6). This is described by Algorithm 8.

The algorithms describe only the assemblage of DOF matrices. The assemblage of DOF vectors is done in an analog way.

The most complex step of the assemblage is the computation of the element matrix (line 5 of Algorithm 6) and of the element vector, which is described in the rest of this section.

In Figure 3.14, one can see the class diagram of the main operator and assembler classes.

Algorithm 8 add element matrix m^S to global matrix $M^{p,q}$

1: **for** $k = 1, \ldots, n^p$ **do**
2:   **for** $l = 1, \ldots, n^q$ **do**
3:     $M^{p,q}_{m^S.r_k, m^S.c_l} \mathrel{+}= m^S.m_{k,l}$
4:   **end for**
5: **end for**
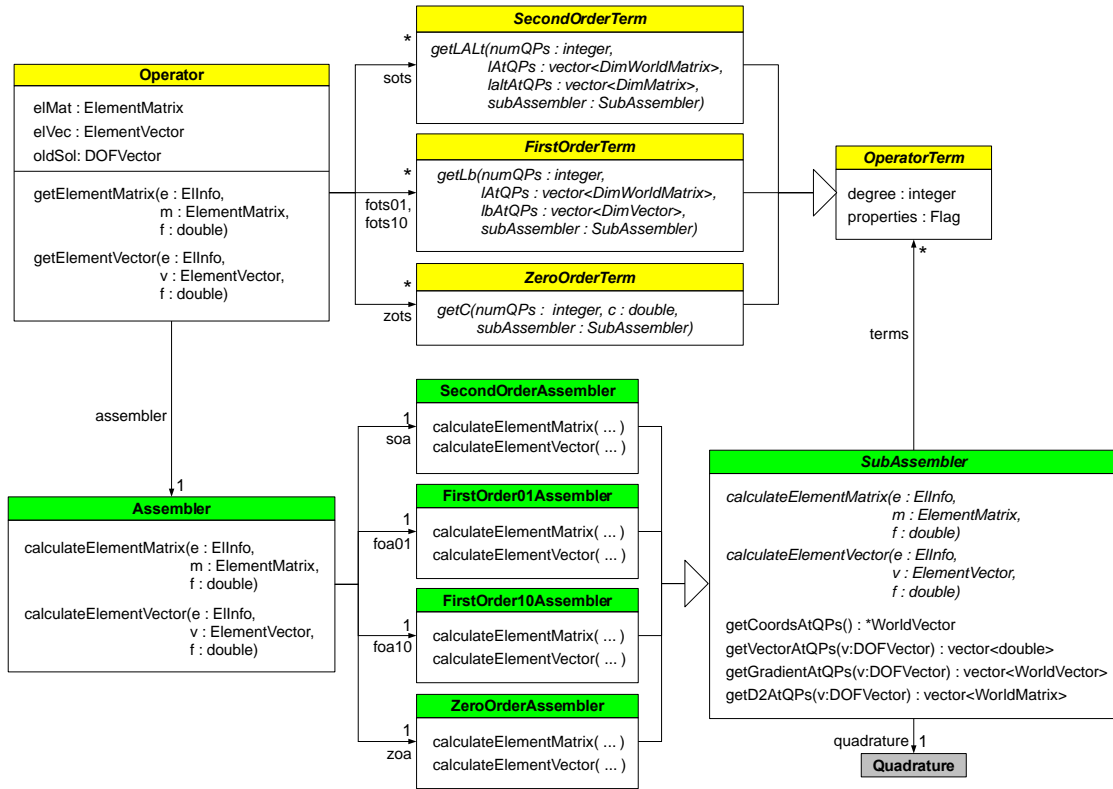
Figure 3.14: UML diagram for the operator and assembler classes.

If d is the dimension of the mesh and w is the dimension of the world the mesh lives in (can be different for parametric elements, see Section 3.7), the vectors and matrices in Figure 3.14 are of the following types:

- `WorldVector`: Double valued w dimensional vector.

- `WorldMatrix`: Double valued $w \times w$ dimensional matrix.

- `DimVector`: Double valued $(d + 1)$ dimensional vector.

- `DimMatrix`: Double valued $(d + 1) \times (d + 1)$ dimensional matrix.

- `DimWorldMatrix`: Double valued $(d + 1) \times w$ dimensional matrix.

- `vector<T>`: Vector with an entry of type `T` for every quadrature point.

- `ElementVector` and `ElementMatrix`: See Section 3.5.2.

In the following the classes shown in Figure 3.14 are described in detail.

OperatorTerm

`OperatorTerm` is the base class for all second, first and zero order operator terms.

Members:

- `degree`: The polynomial degree with which the term is considered in the assemblage. The degree of the operator terms is necessary to determine the degree needed for the quadrature of the corresponding `SubAssembler`.

- `properties`: This is a flag, that stores term properties, which can be used for assembler optimizations. In the current AMDiS version the properties `SYMMETRIC` and `PW_CONST` (piecewise constant) are defined (see Section 3.5.4).

AMDiS already includes a operator term library which covers a large class of problems. It is also possible to implement new operator terms by deriving from one of the base classes `SecondOrderTerm`, `FirstOrderTerm` or `ZeroOrderTerm`. In general, it holds: The more specialized an operator term class is, the smaller its application range is, but also the more efficient it can be implemented.

SecondOrderTerm

`SecondOrderTerm` is a sub class of `OperatorTerm` and base class of all concrete second order terms. A concrete second order term must implement the computation of $\Lambda(\hat{x})A(F_S(\hat{x}))\Lambda^t(\hat{x})$ at all quadrature points of the current element. Since the quadrature points are given in barycentric coordinates, for each quadrature point coordinate λ_i the $(d+1) \times (d+1)$ dimensional matrix $\Lambda(\hat{x}(\lambda_i))A(x^S(\lambda_i))\Lambda^t(\hat{x}(\lambda_i))$ is computed.

Methods:

- `getLALt` (abstract): Computation of $\Lambda(\hat{x}(\lambda_i))A(x^S(\lambda_i))\Lambda^t(\hat{x}(\lambda_i))$. Must be overridden by concrete second order terms. Called by a `SecondOrderAssembler` instance.

  Parameters:

  - `numQPs` (in): Number of quadrature points.
  - `lAtQPs` (in): $\Lambda(\lambda_i)$ at all quadrature points λ_i.
  - `laltAtQPs` (in/out): $\Lambda(\hat{x}(\lambda_i))A(x^S(\lambda_i))\Lambda^t(\hat{x}(\lambda_i))$ is computed for all quadrature points and added to `laltAtQPs`.
  - `subAssembler`: Reference to the calling `SubAssembler` object. Used for callback to get quadrature point information.

The most general second order term in AMDiS so far is implemented by the class `General_SOT`. It represents the term $-\nabla \cdot A(x, t, v_1, \ldots, v_n, \nabla w_1, \ldots, \nabla w_m)\nabla u$. The world matrix A is given by the user as function that can depend on x, t, an arbitrary set of functions $v_i : \Omega \to \mathbb{R}$, $i = 1, \ldots, n$, and an arbitrary set of gradients of functions $w_i : \Omega \to \mathbb{R}$, $i = 1, \ldots, m$. The v_i and w_i are given in discretized form as DOF vectors.

FirstOrderTerm

`FirstOrderTerm` is a sub class of `OperatorTerm` and base class of all concrete first order terms. A concrete first order term must implement the computation of $\Lambda(\hat{x})b(F_S(\hat{x}))$ at all quadrature points of the current element. Since the quadrature points are given in barycentric coordinates, for each quadrature point coordinate λ_i the $(d+1)$ dimensional vector $\Lambda(\hat{x}(\lambda_i))b(x^S(\lambda_i))$ is computed.

Methods:

- `getLb` (abstract): Computation of $\Lambda(\hat{x}(\lambda_i))b(x^S(\lambda_i))$. Must be overridden by concrete first order terms. Called by a `FirstOrderAssembler` instance.

  Parameters:

  - `numQPs` (in): Number of quadrature points.
  - `lAtQPs` (in): $\Lambda(\lambda_i)$ at all quadrature points λ_i.
  - `lbAtQPs` (in/out): $\Lambda(\hat{x}(\lambda_i))b(x^S(\lambda_i))$ is computed for all quadrature points and added to `lbAtQPs`.
  - `subAssembler`: Reference to the calling `SubAssembler` object. Used for callback to get quadrature point information.

The most general first order term in AMDiS so far is implemented by the class `General_FOT`. It represents the term $b(x, t, v_1, \ldots, v_n, \nabla w_1, \ldots, \nabla w_m) \cdot \nabla u$. The world vector b is given by the user as function that can depend on x, t, an arbitrary set of functions $v_i : \Omega \to \mathbb{R}$, $i = 1, \ldots, n$, and an arbitrary set of gradients of functions $w_i : \Omega \to \mathbb{R}$, $i = 1, \ldots, m$. The v_i and w_i are given in discretized form as DOF vectors.

ZeroOrderTerm

`ZeroOrderTerm` is a sub class of `OperatorTerm` and base class of all concrete zero order terms. A concrete zero order term must implement the computation of $c(F_S(\hat{x}))$ at all quadrature points of the current element. Since the quadrature points are given in barycentric coordinates, for each quadrature point coordinate λ_i the value $c(x^S(\lambda_i))$ is computed.

Methods:

- `getC` (abstract): Computation of $c(x^S(\lambda_i))$. Must be overridden by concrete zero order terms. Called by a `ZeroOrderAssembler` instance.

  Parameters:

  - `numQPs` (in): Number of quadrature points.
  - `cAtQPs` (in/out): $c(x^S(\lambda_i))$ is computed for all quadrature points and added to `cAtQPs`.
  - `subAssembler`: Reference to the calling `SubAssembler` object. Used for callback to get quadrature point information.

The most general zero order term in AMDiS so far is implemented by the class `General_SOT`. It represents the term $c(x, t, v_1, \ldots, v_n, \nabla w_1, \ldots, \nabla w_m)$. The function c can depend on x, t, an arbitrary set of functions $v_i : \Omega \to \mathbb{R}$, $i = 1, \ldots, n$, and an arbitrary set of gradients of functions $w_i : \Omega \to \mathbb{R}$, $i = 1, \ldots, m$. The v_i and w_i are given in discretized form as DOF vectors.

Operator

Members:

- `assembler`: Reference to the corresponding `Assembler` object.
- `sots`: List of second order terms of this operator.
- `fots01`: List of first order terms of type FOT_{01} of this operator.
- `fots10`: List of first order terms of type FOT_{10} of this operator.
- `zots`: List of zero order terms of this operator.

- `elMat`: Stores the element matrix of this operator for the current element. Used for optimization (see Section 3.5.4). If it is not needed, it can be a `null`-reference.

- `elVec`: Stores the element vector of this operator for the current element. Used for optimization (see Section 3.5.4). If it is not needed, it can be a `null`-reference.

- `oldSol`: Used for matrix-vector assemblage (see Section 3.5.3).

Methods:

- `getElementMatrix`: Provides the element matrix for the current element. The computation is delegated to `assembler`.

  Parameters:

  - `e` (in): `ElInfo` of the current element.
  - `m` (in/out): Element matrix to which the result should be added.
  - `f` (in): Factor by which the result is multiplied before it is added to `m`.

- `getElementVector`:

  Parameters: Provides the element vector for the current element. The computation is delegated to `assembler`.

  - `e` (in): `ElInfo` of the current element.
  - `v` (in/out): Element vector to which the result should be added.
  - `f` (in): Factor by which the result is multiplied before it is added to `v`.

SubAssembler

`SubAssembler` is the base class of all second, first and zero order assemblers. It manages quadrature point information for all terms.

Members:

- `terms`: List of all operator terms treated by this sub assembler.

- `quadrature`: Reference to the used quadrature. The quadrature is applied to all operator terms in `terms`.

Methods:

- `calculateElementMatrix` (abstract): Computation of the element matrix for the current element. Must be implemented by concrete sub assemblers.

  Parameters:

  - `e` (in): `ElInfo` of the current element.
  - `m` (in/out): Element matrix to which the result should be added.
  - `f` (in): Factor by which the result is multiplied before it is added to `m`.

- `calculateElementVector` (abstract): Computation of the element vector for the current element. Must be implemented by concrete sub assemblers.

  Parameters:

  - `e` (in): `ElInfo` of the current element.

- **–** `v` (in/out): Element vector to which the result should be added.

- **–** `f` (in): Factor by which the result is multiplied before it is added to `v`.

- `getCoordsAtQPs`: Provides barycentric coordinates of all quadrature points of `quadrature`.
  Returns: Vector containing a $(d + 1)$ dimensional `DimVector` for each quadrature point.

- `getVectorAtQPs`: Provides `DOFVector` v evaluated at each quadrature point of `quadrature`.
  Parameters:

  - **–** `v`: `DOFVector` to be evaluated.

  Returns: Vector containing one `double` value for each quadrature point of `quadrature`.

- `getGradientAtQPs`: Provides the gradient of `DOFVector` v evaluated at each quadrature point of `quadrature`.
  Parameters:

  - **–** `v`: `DOFVector` of which the gradient should be computed.

  Returns: Vector containing a w dimensional `WorldVector` for each quadrature point.

- `getD2AtQPs`: Provides the Hessian matrix of `DOFVector` v evaluated at each quadrature point of `quadrature`.
  Parameters:

  - **–** `v`: `DOFVector` of which the Hessian matrix should be computed.

  Returns: Vector containing a $w \times w$ dimensional `WorldMatrix` for each quadrature point.

SecondOrderAssembler

Implementation of `SubAssembler` for second order terms.

FirstOrder01Assembler

Implementation of `SubAssembler` for first order terms of type FOT_{01}.

FirstOrder10Assembler

Implementation of `SubAssembler` for first order terms of type FOT_{10}.

ZeroOrderAssembler

Implementation of `SubAssembler` for zero order terms.

Assembler

Members:

- `soa`: Second order assembler. Used for the assemblage of the second order terms in the corresponding operator.

- `foa01`: First order assembler. Used for the assemblage of the first order terms of type FOT_{01} in the corresponding operator.

- `foa10`: First order assembler. Used for the assemblage of the first order terms of type FOT_{10} in the corresponding operator.

- `zoa`: Zero order assembler. Used for the assemblage of the zero order terms in the corresponding operator.

Methods:

- `calculateElementMatrix`: Iterates through all sub assemblers and sums up its element matrices.

  Parameters:

  - `e` (in): `ElInfo` of the current element.
  - `m` (in/out): Element matrix to which the result is added.
  - `f` (in): Factor by which the result is multiplied before it is added to m.

- `calculateElementVector`: Iterates through all sub assemblers and sums up its element vectors.

  Parameters:

  - `e` (in): `ElInfo` of the current element.
  - `v` (in/out): Element vector to which the result is added.
  - `f` (in): Factor by which the result is multiplied before it is added to v.

A little example

Consider the equation $-\nabla \cdot A(x, v(x))\nabla u(x) + c_1(x)u(x) + c_2(x)u(x) = 0$. The second order term depends on a function $v(x)$ which is given as a `DOFVector` in the discretization. There are two zero order terms and no first order terms. In Figure 3.15, the assemblage of the element matrix is visualized as UML sequence diagram.
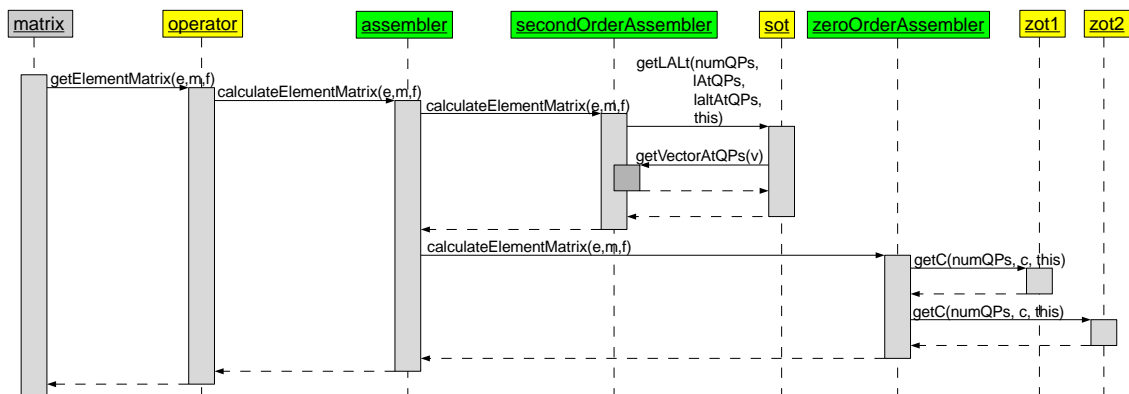
Figure 3.15: Example of a sequence diagram for the element matrix assemblage.

The assemblage consists of the following steps:

- The `DOFMatrix matrix` is assembled by traversing over all elements and summing up the corresponding element matrices. For every element `getElementMatrix(e, m, f)` of every operator that is applied to `matrix` is called (in this example, we have only one operator with three operator terms). The factor `f` is 1.0 in this example.

	matrix terms	vector terms	
		regular	mat-vec
SOT	$\left(\int_\Omega \nabla\phi_i^p \cdot A\nabla\phi_j^q dx\right)_{i,j}$	-	$\left(\int_\Omega \nabla\phi_i^p \cdot A\nabla\phi_j^q dx\right)_{i,j} \left(U_j^{old}\right)_j$
FOT$_{10}$	$\left(\int_\Omega \nabla\phi_i^p \cdot b\phi_j^q dx\right)_{i,j}$	$\left(\int_\Omega \nabla\phi_i^p \cdot bdx\right)_i$	$\left(\int_\Omega \nabla\phi_i^p \cdot b\phi_j^q dx\right)_{i,j} \left(U_j^{old}\right)_j$
FOT$_{01}$	$\left(\int_\Omega \phi_i^p b \cdot \nabla\phi_j^q dx\right)_{i,j}$	-	$\left(\int_\Omega \phi_i^p b \cdot \nabla\phi_j^q dx\right)_{i,j} \left(U_j^{old}\right)_j$
ZOT	$\left(\int_\Omega \phi_i^p c\phi_j^q dx\right)_{i,j}$	$\left(\int_\Omega \phi_i^p f dx\right)_i$	$\left(\int_\Omega \phi_i^p c\phi_j^q dx\right)_{i,j} \left(U_j^{old}\right)_j$

Table 3.3: Discretization of the different operator term types.

- The operator delegates the computation to its assembler by calling `calculateElementMatrix(e, m, f)`.

- The assembler has two sub assemblers, one for the second order term and one for the two zero order terms. Both sub assemblers are called consecutively and the results are added to m.

- The second order assembler calls `getLALt(numQPs, lAtQPs, laltAtQPs, this)` of its second order term.

- Since the second order term needs the vector v at quadrature points, it calls `getVectorAtQPs(v)` of the second order assembler (third argument of `getLALt()`).

- The zero order assembler calls `getC(numQPs, c, this)` for both zero order terms.

Matrix-vector assemblage

Let $O(u)$ be an arbitrary operator applied to the left hand side of the equation, where u is the searched solution. If the operator is applied to a known function v instead of u, $\mathcal{O}(v)$ can be put on the right hand side of the equation. This e.g. often is used for the linearization of nonlinear problems or for the time discretization of instationary problems. In the first case, v is the solution of the last fix-point iteration. In the second case, it is the solution of the last timestep. In both cases, we replace v by u^{old}.

The vector u^{old} is discretized by $\sum_j U_j^{old}\phi_j$, with U^{old} the solution vector of the last timestep or fix-point iteration and ϕ_j the global basis functions of the corresponding finite element space. Therefore, $O(u^{old})$ can be assembled by first assembling the DOF matrix and then multiply it by U^{old}. Since the DOF Matrix is the sum over all element matrices, we can do this matrix-vector multiplication element-wise. The element matrix has to be assembled only once, also if the operator appears several times in the equation (see Section 3.5.4).

In Table 3.3, the discretizations of all operator term types are shown. In the right column, the vector terms assembled by matrix-vector multiplication are listed.

3.5.4 Assembler optimizations

Rembering element matrices and vectors

If an operator is used more than once (e.g. like described in Section 3.5.3), multiple discretizations of this operator should be avoided. For that reason, every operator can store the element matrix and the element vector for the current element. If the same operator is called again for the same element within the same mesh traverse, the stored element matrix and vector can be used (even with a different factor).

Avoidance of redundant quadrature point evaluation

If different operator terms depend on the same DOF vector or derivatives of this vector, this vector should be evaluated at the quadrature points only once. Therefore, this evaluation is not done by the operator terms but by the sub assembler, which knows all operator terms corresponding to its quadrature.

Symmetric assamblage

If all second order terms of a second order assembler have a symmetric matrix A, the assemblage can be reduced to the upper triangular element matrix. The other values are filled with corresponding copies.

Precalculated basis functions

As long as normal quadratures are used (no scaled quadratures or surface quadratures), the basis functions only have to be evaluated once at the reference element (see Section 3.5.1).

Precalculated integrals

If all operator terms of a sub assembler are piecewise constant on each element, the integrals can be pre-calculated (see Section 3.5.1).

3.5.5 Assemblage for systems of PDEs

If a system of PDEs is to be solved, the matrix of the linear system of equations is a $\mathcal{N} \times \mathcal{N}$ dimensional matrix of DOF matrices. The DOF matrix in row p and column q has the row finite element space V^p and the column finite element space V^q. The right hand side vector and the solution vector are vectors of DOF vectors. The p-th components are defined on finite element space V^p.

The assemblage of the system can be reduced to the assemblage of DOF matrices and DOF vectors. The vector valued assemblage is described by Algorithm 9.

Algorithm 9 assemble system of a vector valued problem

1: **for** $p = 1, \ldots, \mathcal{N}$ **do**
2:   **for** $q = 1, \ldots, \mathcal{N}$ **do**
3:     assemble global matrix $M^{p,q}$
4:   **end for**
5:   assemble global vector F^p
6: **end for**

3.6 Assemblage of boundary conditions

In the macro file, boundary regions are identified by assigning boundary region numbers to the sides of macro mesh elements. In the application, These identifiers then are assigned to boundary conditions that should be satisfied at the corresponding boundary regions.

In the mesh, the boundary condition numbers are associated to element sides (vertices in 1d, edges in 2d, faces in 3d). The evaluation of boundary conditions must be done for element nodes. If different boundary conditions coincide in one node, the condition with the highest number prevails. This is illustrated in Figure 3.16. Although two sides of the macro mesh are assigned to boundary region 1, all boundary vertices of the macro mesh are assigned to region 2, since it

is the larger number. After one global refinement of the macro mesh, in region 1 new vertices appear, that are not dominated by region 2.
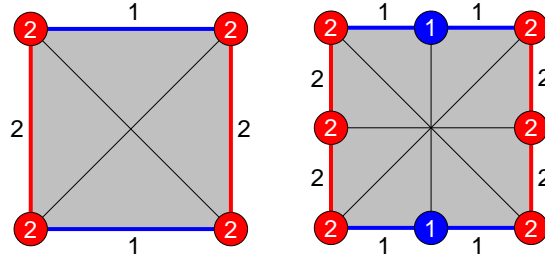
Figure 3.16: Boundary condition numbers for element sides and nodes of the macro mesh (left hand side), and of a refined mesh (right hand side).

The boundary conditions of the element nodes are provided by the mesh traverse through `ElInfo` objects (see Section 3.3).

Every DOF matrix and every DOF vector has its own boundary manager, which manages all boundary conditions that must be applied to the matrix or vector. The assemblage of boundary conditions is done in three steps:

1. **Initialization**: In this step, needed memory can be allocated and the assemblage can be prepared.

2. **Element-wise assemblage**: Within a mesh traversal, element contributions of the boundary condition are filled to the DOF matrix or vector.

3. **Finalizing**: Memory that was allocated in the initialization step must be freed here.

In Figure 3.17, the relationships between DOF matrices, DOF vectors, boundary managers and boundary conditions are visualized.

3.6.1 Dirichlet boundary conditions

Dirichlet boundary conditions play a special role during the assemblage of the linear system of equations. The whole row for Dirichlet DOF index i in component p is replaced by the row representing $1 \cdot U_i^p = G_i^p$, where U_i^p is the solution at the Dirichlet index and G_i^p the discretized Dirichlet

Figure 3.17: Every DOF matrix and every DOF vector has its own boundary manager which manages the corresponding boundary conditions.
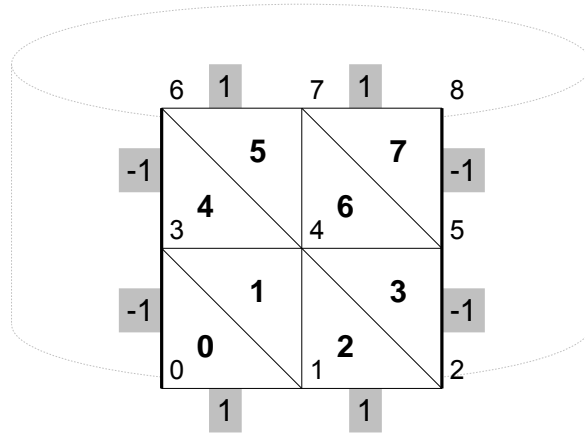
Figure 3.18: Macro mesh with periodic boundaries.

function at this index. As described in Section 2.4, this has already been considered during the assemblage of the linear system of equations. To identify Dirichlet boundary conditions within the assemblage, every boundary condition must implement the method `isDirichlet()` returning a boolean value.

3.6.2   Neumann boundary conditions

For the assemblage of Neumann boundary conditions, surface integrals have to be computed (see Section 2.5.2). These integrals contribute to the right hand side of the system of equations. Hence, Neumann boundary conditons are applied to the right hand side DOF vector.

3.6.3   Periodic boundary conditions

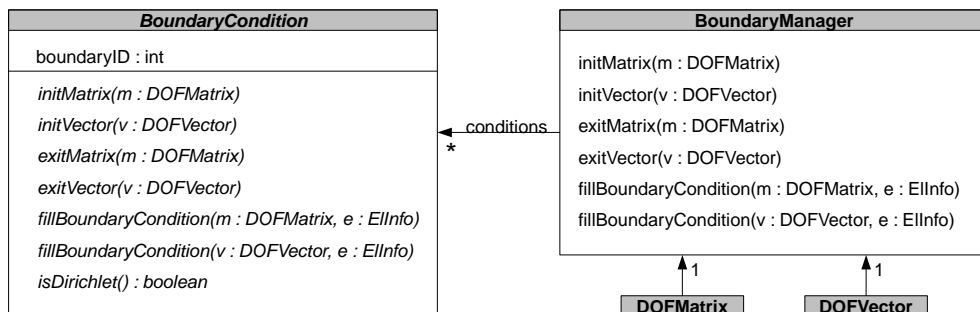Periodic boundary conditions are used, if the solution is assumed to be periodically continued through given boundaries. Since periodic boundary conditions actually are not real boundary conditions (the domain is assumed to be continued behind the periodic boundary), they must have negative numbers. This deactivates some mesh consistency checks which are not usefull in the periodic case (e.g./ usually elements must not have neighbors at boundary sides).

In Figure 3.18 an example macro mesh with periodic and non-periodic boundary conditions is shown. The periodic boundaries are marked by -1, the non-periodic by 1.

In AMDiS periodic boundary conditions can be implemented in two ways:

Periodic boundary conditions by changing mesh topology

The first way is to change the topology of the macro mesh. As mentioned in Section 3.1, at each position of an element (vertices, edges, faces, center) one pointer to an array of DOF indices is stored. In this array, the DOF indices for each node of this position are stored.

In the example of Figure 3.18, periodic boundary conditions could be implemented by deleting the DOF index arrays cooresponding to vertices 2, 5 and 8. After that the pointer at vertex 2 is reassigned to point to the same DOF index array as the pointer of vertex 0. In the same way the pointers for vertices 5 and 8 are reassigned to the arrays of vertices 3 and 6. The vertex coordinates that are stored in the macro elements keeps unchanged. Hence, one DOF index can have different coordinates within different elements.

After the topology has been changed in this way, periodic boundaries do not have to be considered again. Therefore, no implementation of class `BoundaryCondition` is necessary.

Periodic boundary conditions by associating DOF indices

If periodic boundary conditions are used together with parametric elements (see Section 3.7), the vertex coordinates may be stored in DOF vectors and not in the macro elements. If we change the mesh topology, as described in the last section, we have only one DOF index for all vertices that belong to the same periodic vertex. Thus, in Figure 3.18, nodes 0 and 2 share the same coordinates as well as nodes 3 and 5, and nodes 6 and 8.

To avoid this geometric limitation, the DOF index arrays and the pointers to it remain unchanged. Instead of that, periodic associations are stored. In the example, vertex 0 and vertex 2 are associated with each other as well as the vertices 3 and 5, and the vertices 6 and 8. When the mesh is refined, new associations must be created for the new vertices that are located at periodic boundaries.

Furthermore, an implementation of the `BoundaryCondition` base class is necessary. This periodic boundary condition must be applied to the system matrix as well as to the right hand side vector. It assures that associated DOFs behave in a periodic manner.

During mesh traversal within the matrix version of `fillBoundaryCondition()`, the vertex associations are used to construct periodic associations for all DOF indices (also for non-vertex DOF indices). In a_i^p the DOF index associated to DOF index i in finite element space V^p is stored. If there is no associated DOF index to i, a_i^p is set to i. The mapping a_j^q is defined in the same way for finite element space V^q.

For systems of PDEs the mapping a^p is used for every matrix $M^{p,\cdot}$ in row p and the mapping a^q is used for every matrix $M^{\cdot,q}$ in column q. If the same finite element spaces are used for component p and q, also the mappings a^p and a^q are identic. Hence, it is useful to construct the mappings only once for each finite element space.

In Algorithm 10, the periodic boundary condition is applied to matrix $M^{p,q}$. This is done in the `exitMatrix()` method. To simplify matters, the sparse structure of the matrix is not considered here.

Algorithm 10 Apply periodic boundary condition to matrix $M^{p,q}$ (dense version)

1: **for** $i = 1, \ldots, N^p$ **do**
2:     $i_a = a_i^p$
3:     **for** $j = 1, \ldots, N^q$ **do**
4:        $j_a = a_j^q$
5:        **if** $(i < i_a) \vee ((i = i_a) \wedge (j < j_a))$ **then**
6:           $m = \frac{1}{2}(M_{i,j}^{p,q} + M_{i_a,j_a}^{p,q})$
7:           $M_{i,j}^{p,q} = m$
8:           $M_{i_a,j_a}^{p,q} = m$
9:        **end if**
10:     **end for**
11: **end for**

In Algorithm 11, the periodic boundary condition is applied to the right hand side vector F^p. This is implemented in the `exitVector()` method.

This implementation of periodic boundary conditions has the following properties:

- The resulting system of equations is equivalent to the system of equations in the case of changed mesh topology.

- Symmetric properties of the system matrix are conserved, which is important for some iterative solver methods.

Algorithm 11 Apply periodic boundary condition to vector F^p

1: **for** $i = 1, \ldots, N^p$ **do**
2:     $i_a = a_i^p$
3:     **if** $i < i_a$ **then**
4:         $m = \frac{1}{2}(F_i^p + F_{i_a}^p)$
5:         $F_i^p = m$
6:         $F_{i_a}^p = m$
7:     **end if**
8: **end for**
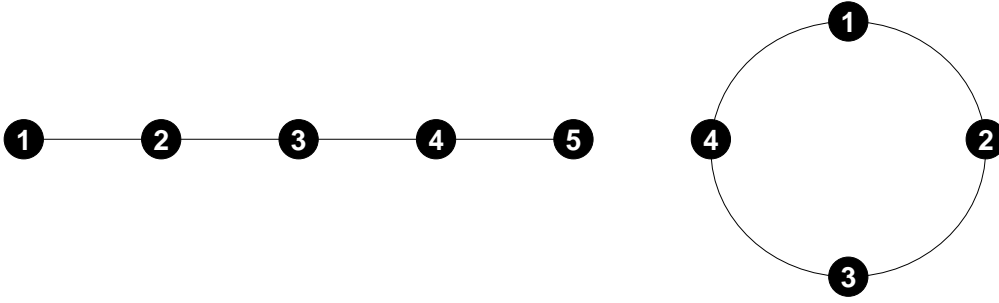
Figure 3.19: The two different mesh topologies used for periodic boundary conditions.

This is illustrated in the following simple example. Consider a one dimensional mesh consisting of 4 elements. Each element has length 1. We use linear Lagrange basis functions for the finite element discretization of the Poisson equation $-\Delta u = f$, and we want to apply periodic boundary conditions. In Figure 3.19 the two possible mesh topolgies are shown. On the left hand side, one can see the original mesh topolgy, on the right hand side, the changed topology after DOF index 5 was removed.

First, we assemble the system of equations for the case of changed topology:

$$\begin{pmatrix} 2 & -1 & 0 & -1 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ -1 & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{pmatrix}. \tag{3.14}$$

The right hand side entry F_1 contains contributions from element $1-2$ and from element $4-1$.

If we assemble the system of equations for the unchanged mesh topology and without periodic boundary condition, the system of equations looks like:

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \\ U_5 \end{pmatrix} = \begin{pmatrix} F_1' \\ F_2 \\ F_3 \\ F_4 \\ F_5' \end{pmatrix}, \tag{3.15}$$

where F_1' contains only the contribution from element $1-2$ and F_5' only the contribution from element $4-5$. It holds $F_1 = F_1' + F_5'$

Now, we apply the above algorithms, and the system of equations is modified to:

$$\begin{pmatrix} 1 & -\frac{1}{2} & 0 & -\frac{1}{2} & 0 \\ -\frac{1}{2} & 2 & -1 & 0 & -\frac{1}{2} \\ 0 & -1 & 2 & -1 & 0 \\ -\frac{1}{2} & 0 & -1 & 2 & -\frac{1}{2} \\ 0 & -\frac{1}{2} & 0 & -\frac{1}{2} & 1 \end{pmatrix} \begin{pmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \\ U_5 \end{pmatrix} = \begin{pmatrix} \frac{1}{2}(F_1' + F_5') \\ F_2 \\ F_3 \\ F_4 \\ \frac{1}{2}(F_1' + F_5') \end{pmatrix}. \tag{3.16}$$

We see:

- The matrix is still symmetric.

- If we subtract line 1 of the system of equations from line 5, we get $U_1 = U_5$.

- Since $U_1 = U_5$, line 1 and line 5 of the system of equations are equivalent to line 1 of system of equations (3.14) (the whole line devided by 2).

- Since $U_1 = U_5$, line 2 is equivalent to line 2 of system of equations (3.14) and line 4 is equivalent to line 4 of system of equations (3.14).

- Line 3 is obviously equivalent to line 3 of of system of equations (3.14). Hence, the whole system of equations is equaivalent to system of equations (3.14).

3.7 Parametric Finite Elements

As described in Chapter 2, local basis functions for an element S are given by the local basis functions defined on the reference simplex $\bar{S}$ together with the mapping $F_S : \hat{S} \to S$ from the standard simplex $\hat{S}$ to S. By default, F_S is the affine mapping uniquely defined by the vertex coordinates of S (see Section 2.1). A flexible redefinition of F_S can enable computations

- on arbitrary manifolds where the dimension of the mesh may be smaller than the dimension of the world,

- on meshes with curved elements (if F_S is not affine),

- on moving meshes (if F_S is time dependent).

A redefinition of the parametrization F_S has the following consequences on the finite element computation:

- The vertex coordinates of an element, which are filled during mesh traversal based on macro element coordinates, may have to be modified.

- The coordinate transformation functions `coordToWorld()` and `worldToCoord()` must take the parametrization into account. If F_S is affine, the default implementation can still be used, because it is based on the (already parametrized) vertex coordinates. Otherwise, these functions have to be reimplemented.

- The determinant $\det DF_S$ of the Jacobian of F_S may have to be modified (if F_S is affine, the default implementation is sufficient).

- The Jacobian of λ^S (Λ) may have to be modified, since λ^S depends on F_S. If F_S is not affine, Λ is not constant on S and it has to be computed at every quadrature point.

All these aspects are managed by the `ElInfo` classes (see Section 3.3). Thus, we define a parametrizer interface with the following methods:

```
class Parametric {
public:
   ElInfo* addParametricInfo(ElInfo* elInfo) = 0;
   ELInfo* removeParametricInfo(ElInfo* elInfo) = 0;
}
```
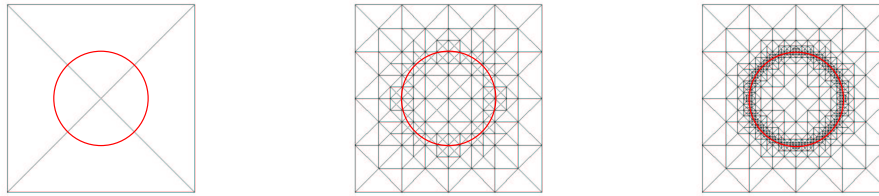
Figure 3.20: Different levels of refinement of the underlying grid and boundary $\mathcal{M}_t$ of the domain.

The method `addParametricInfo()` is directly called before an `ElInfo` object is returned to the traverse client. Instead of the original `ElInfo` object, the object returned by `addParametricInfo()` is returned. This can be the same object just with modified member values, or it can be a new object (e.g. an instance of a parametrized sub class of `ElInfo`) which decorates the original `ElInfo`.

The method `removeParametricInfo()` removes the parametric information from a parametrized `ElInfo` object. It is called as first step within the `traverseNext()` function for the given old `ElInfo` object. Furthermore, memory that was allocated when parametric information was added, has to be freed here (e.g. the decorating parametrized object). The method returns the original (non-parametrized) `ElInfo` object.

If a parametrizer is given, the parametrization is totally transparent for the user. A mesh traversal looks the same no matter if it is parametric or not. To parametrize a mesh in an existing non-parametrized application, only the parametrizer has to be added to the mesh at the beginning of computations. The rest of the code remains unchanged.

Currently in AMDiS, parametric elements are implemented for affine F_S. The parametrization is given as a vector, which stores the world coordinates for each vertex of the triangulation. During mesh traversal, these coordinates are used in order to compute all relevant element data.

Examples of the usage of parametric finite elements are given in Sections 8.1.2, 8.1.5 and A.4.9.

3.8 Composite Finite Elements

Creating a mesh is always the first step in a wide range of applications concerned with scientific computing and computer graphics. In engineering as well as medical applications usually quite complex three-dimensional geometries need to be meshed. Fully automatic mesh generators for such applications leading to high quality meshes for the numerical problem are still rare. The codes are usually quite complex and nearly inaccessible by the user which suppresses the possibility to combine the mesh generation with the numerical solution and visualization. The advantage of this interaction necessary for multilevel treatments lies in the ability to adaptively describe the complexity of the geometry as well as the solution. We therefore follow a different way which circumvents the meshing of a complex domain. An essential decision in this way is how to represent the geometry. We use a *signed distance function* $d(x, y, z, t)$, which is negative inside the region, to represent our domain. Our numerical grid is now always generated from a regular simplicial grid in a larger box, which is adaptively refined according to $d(x, y, z, t)$, see Figure 3.20 for an example. The signed distance function can either be given analytically, be computed for implicitly given boundaries by equations $f(x, y, z, t) = 0$ or be provided in a discrete form by values on the grid, which is common in *level set* applications [35], where PDEs efficiently model geometries with moving boundaries.

Let the boundary of the domain at time t be given by the zero level set of the signed distance
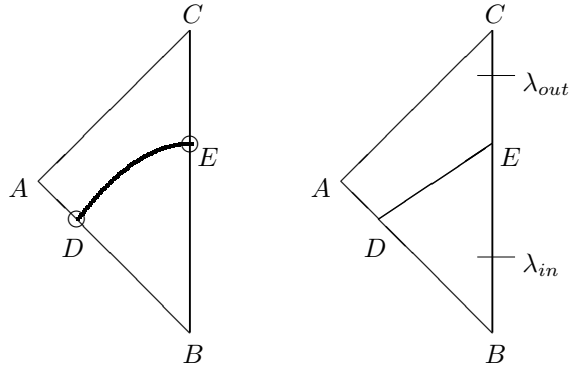
Figure 3.21: Element S, boundary Γ_C, and definition of λ.

function $\mathcal{M}_0 = \{(x, y, z) \in \Omega \mid d(x, y, z, t) = 0\}$. This boundary does not coincide with the boundary of the computational domain, which is a box embedding $\mathcal{M}_0$, nor can it be represented by nodes of the underlying grid. Figure 3.20 shows an illustrative two-dimensional example of a circle represented by $d(x, y, t) = (x^2 + y^2)^{1/2} - 1$ in a computational domain $[-2, 2] \times [-2, 2]$ for different refinements.

The geometry is adaptively resolved with increasing refinements, but at no level can be represented by the nodes of the grid. Instead we are confronted with the situation of $d(x, y, t) = 0$ within an element S, as pointed out in Figure 3.21. If parameters vary across $\mathcal{M}_0$ integrals of the form $\int_S \lambda \phi$, with λ a discontinuous function and ϕ a smooth function, have to be evaluated. The method used is similar as in [6] and is explained in Figure 3.21:

$$\int_S \lambda \, \phi \approx \int_{\triangle(DBE)} \lambda_{in} \phi + \int_{\square(ADEC)} \lambda_{out} \phi$$

$$= \int_{\triangle(DBE)} \lambda_{in} \phi + \int_S \lambda_{out} \phi - \int_{\triangle(DBE)} \lambda_{out} \phi.$$

Note that this formula avoids the explicit integration over quadrilaterals and requires only integration over triangles, and can be thus performed in a nearly standard way.

If in addition boundary conditions are specified at the domain wall given by $\mathcal{M}_0$, a penalty method is applied in order to fulfill them. We approximate the introduced line integral along $\mathcal{M}_0$ within an element S by an integration along the straight line DE, see Figure 3.21. This involves nothing else than a standard integration of an element in one dimension and therefore leads to no further complications. The same approach can be carried over to three dimensions.

A more rigorous approach to circumvent resolving the boundary of complex geometries by the mesh can be found in [24], where the notion *composite finite elements* has been introduced.

Chapter 4

The adaptation loop

The goal of using adaptive meshes is to achieve a solution, which satisfies a given quality criterion, with as little computational effort as possible. The quality criterion usually is given as an upper bound for the global error in a given norm. This norm is estimated by an a posteriori error estimator that also computes local error indicators on every mesh element. As long as the global criterion is not reached, the mesh is refined in regions with a high local error indicator. In regions with a very small local error, the mesh may be coarsened again. This is especially useful for instationary problems, where solution properties can change over time. The resulting loop is called *adaptation loop* and consists of the following steps:

- Assemblage of the linear system of equations.

- Solution of the system of equations.

- Estimation of local and global error indicators.

- Adaptation of the mesh according to local error estimates.

The implementation of adaptation loops in AMDiS is done in a flexible and reusable way on a high abstraction level. It is described in Section 4.1.

The assemblage of the linear system of equations has already been described in Section 3.5. For the solution of this system, several iterative Krylov-subspace methods are implemented. For a detailed description of these methods see e.g. [30]. In Chapter 5, the implementation of a multigrid solver is described, which uses the hierarchical mesh structure of AMDiS for an efficient solution of the linear system of equations.

The implementation of error estimators is topic of Section 4.2. Which elements are marked for refinement and coarsening based on local error estimates is decided by the used adaptation strategy. In Section 4.3, different adaptation strategies are introduced.

Mesh refinement in AMDiS is done by bisection of simplices. In contrast to red-green refinement, the bisection method is relatively easy to implement without the need of hanging nodes, even in 3d. In principle, it is generalizable to simplices of any dimension, but one must take care that the refinement algorithm does not result in an endless recursion. The refinement and coarsening algorithms described in Section 4.4 are based on the algorithms described by A. Schmidt and K. G. Siebert in [44]. In Figure 4.1, simplices for 1d, 2d and 3d and the corresponding bisections are shown.

4.1 Implementation of adaptation loops

The adaptation loop is the highest abstraction level in the simulation. Here, the decisions are made when the equation system is assembled, when it is solved, when the error is estimated, and
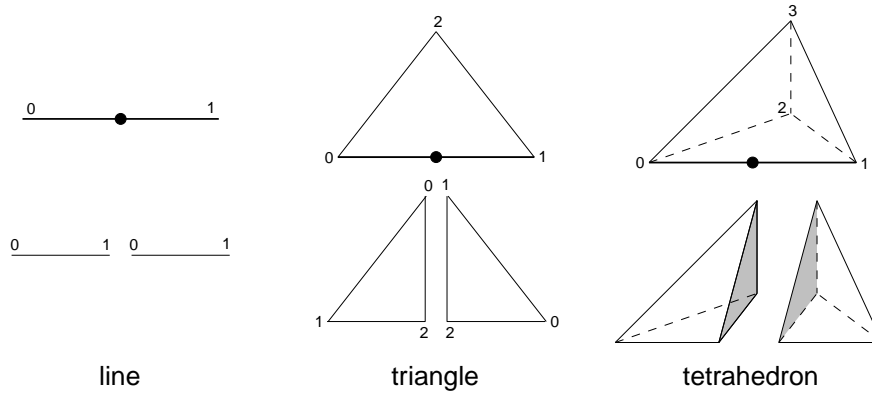
Figure 4.1: Simplices in 1d, 2d and 3d, and their bisections. The •-symbol marks the midpoints of the refinement edges. The numbers are the local vertex indices of parent and child elements. The numbering of children vertices in 3d depends on the element type and is not shown here.
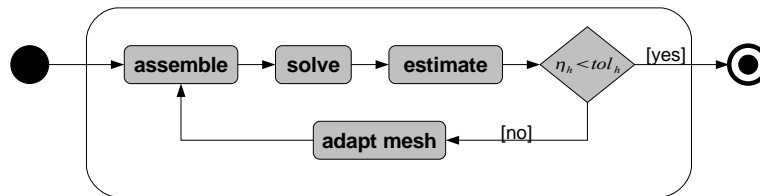

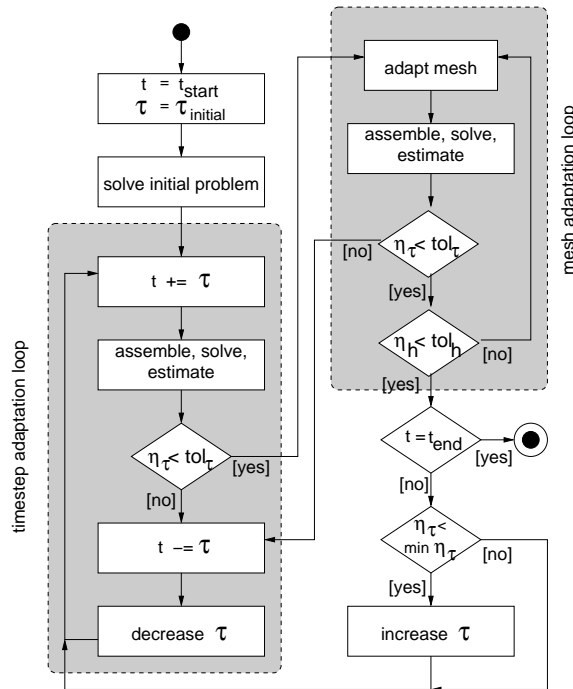Figure 4.2: Adaptation loop of a stationary problem.

Figure 4.3: Implicit time strategy.

when the mesh is adapted. In Figure 4.2, the adaptation loop for a stationary problem is shown. Furthermore, time dependent aspects of instationary problems are controlled by the adaptation loop. Since adaptation loops can be quite complicated, it is important that they can be formulated in a reusable way. In Figure 4.3, the implicit time strategy for instationary problems is shown, like it is implemented in AMDiS. This is an example of a complex adaptation strategy with two nested loops, one for the time step adaptation and one for the mesh adaptation. It is not practicable to re-implement such a strategy for every problem.

In this section, a software design is introduced which allows an adaptation loop formulation in a flexible and reusable way. The main ideas have already been published in [56].

The remainder of this chapter is organized as follows: In Section 4.1.1 the data structure *AdaptInfo* is introduced which is used as capsule to exchange information about the adaptation state between the different involved software modules. The concept of iteration interfaces described in 4.1.2 allows an independent implementation of adaptation loops and space iterations (e.g. coupled or nonlinear iterations). They can be combined at runtime in a flexible way. The time interface described in Section 4.1.3 allows the adaptation loop to access time dependent aspects of instationary problems. Finally, in Section 4.1.4 standard implementations for different types of problems are depicted. These can be used as starting point for the implementation of user defined problems.

4.1.1 AdaptInfo

During the simulation process, many information about the current adaptation state have to be exchanged between the different parts of AMDiS. Furthermore, user defined parameters have to be known.

The class *AdaptInfo* is used as data capsule which stores all theses information and which is exchanged between the software modules.

An instance of *AdaptInfo* stores the following information about the current adaptation state:

- Current iteration numbers (space iterations, timestep iterations)

- Last global error estimates (space discretization error, time discretization error)

- Information about the last solve step (convergence status, number of needed iterations)

- Current simulation time

- Current time step size

- Current time step number

The user defined control parameters in *AdaptInfo* are

- Maximal iteration numbers

- Error tolerances (for the adaptation loop, for the solver)

- Parameters for the mesh adaptation (refinement strategy, number of bisections per refinement step, ...)

4.1.2 Iteration interface

In order to get reusable components that can be combined in a flexible way, it is important to accurately separate different concerns.

We start with a simple example of a stationary adaptation loop, which has already been shown in Figure 4.2. The first idea could be, to implement the loop as part of the problem. But then every problem has to implement its own adaptation loop. This could be avoided by defining the loop

ProblemIterationInterface

beginIteration(adaptInfo:AdaptInfo)

oneIteration(adaptInfo:AdaptInfo, toDo:Flag)

endIteration(adaptInfo:AdaptInfo)
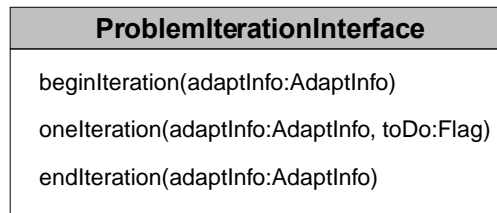
Figure 4.4: UML diagram of *ProblemIterationInterface*.

in a base class, common for all problems or for a class of problems. But then the problem is intrinsically tied to the adaptation strategy. To allow a flexible coupling between problem and loop, the principle of delegation can be applied: The adaption loop knows a problem instance and delegates the single adaptation steps to it. The concrete implementation of the steps is part of the problem. The adaptation loop only determines the calling order.

Now let's assume that we have two coupled problems, where the second problem needs the solution of the first problem to assemble its own equation system. Furthermore, let's assume that both problems are defined on the same mesh and that the first problem is responsible for error estimation and mesh adaptation. In this case, we need an adaptation loop which knows both problems. An implementation could look like the following C++ code.

```
adaptationLoop(AdaptInfo *adaptInfo)
{
  do {
    problem1->assemble(adaptInfo);
    problem1->solve(adaptInfo);
    problem2->assemble(adaptInfo);
    problem2->solve(adaptInfo);
    problem1->estimate(adaptInfo);
    if(adaptInfo->spaceToleranceReached() == false) {
      problem1->adaptMesh(adaptInfo);
    }
  } while(adaptInfo->spaceToleranceReached() == false)
}
```

Now, we have to re-implement the adaptation loop for every kind of problem coupling. The solution is to introduce a new abstraction level with an interface for one iteration. The adaptation loop only knows an implementation instance of this iteration interface. In Figure 4.4, the class diagram of the iteration interface is shown. It has the following methods:

- `beginIteration()`: Called by the adaptation loop at the beginning of each iteration.

- `oneIteration()`: Called by the adaptation loop to perform one space iteration. The flag `toDo` specifies which parts of the iteration should be executed.

- `endIteration()`: Called by the adaptation loop at the beginning of each iteration.

The stationary adaptation loop now looks like:

```
adaptationLoop(AdaptInfo *adaptInfo)
{
  do {
    iterationIF->beginIteration(adaptInfo);
    iterationIF->oneIteration(adaptInfo, ASSEMBLE|SOLVE|ESTIMATE|ADAPT);
    iterationIF->endIteration(adaptInfo);
```

ProblemTimeInterface
initTimestep(adaptInfo:AdaptInfo)
closeTimestep(adaptInfo:AdaptInfo)
setTime(adaptInfo:AdaptInfo)
solveInitialProblem(adaptInfo:AdaptInfo)
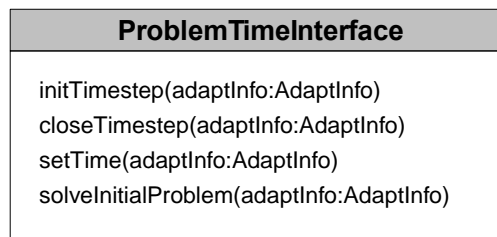
Figure 4.5: UML diagram of *ProblemTimeInterface*.

```
  } while(adaptInfo->spaceToleranceReached() == false)
}
```

The standard implementation of `oneIteration` is:

```
oneIteration(AdaptInfo *adaptInfo, Flag toDo)
{
  if(toDo.isSet(ASSEMBLE)) problem->assemble(adaptInfo);
  if(toDo.isSet(SOLVE)) problem->solve(adaptInfo);
  if(toDo.isSet(ESTIMATE)) problem->estimate(adaptInfo);
  if(adaptInfo->spaceToleranceReached() == false) {
    if(toDo.isSet(ADAPT)) problem->adaptMesh(adaptInfo);
  }
}
```

If two problems should be coupled in the way stated above, the implementation looks like:

```
oneIteration(AdaptInfo *adaptInfo, Flag toDo)
{
  if(toDo.isSet(ASSEMBLE)) problem1->assemble(adaptInfo);
  if(toDo.isSet(SOLVE)) problem1->solve(adaptInfo);
  if(toDo.isSet(ASSEMBLE)) problem2->assemble(adaptInfo);
  if(toDo.isSet(SOLVE)) problem2->solve(adaptInfo);
  if(toDo.isSet(ESTIMATE)) problem->estimate(adaptInfo);
  if(adaptInfo->spaceToleranceReached() == false) {
    if(toDo.isSet(ADAPT)) problem->adaptMesh(adaptInfo);
  }
}
```

The usefulness of this separation becomes clearer for more complex adaptation strategies as the implicit time strategy.

Another example application for the iteration interface is the nonlinear problem described in Section A.4.5. The problem is linearized by the Newton method. In every space iteration, several Newton steps are applied until a given tolerance is reached. In each Newton step, the equation system must be re-assembled and solved. But this is hidden to the adaptation loop.

4.1.3 Time interface

The time interface is used to access time dependent aspects of instationary problems. In Figure 4.5, the class diagram of the time interface is shown. It has the following methods:

- `initTimestep()`: Called by the adaptation loop at the beginning of each timestep.

- `closeTimestep()`: Called by the adaptation loop at the end of each timestep.
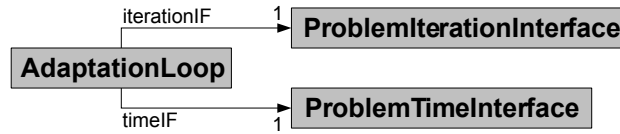
Figure 4.6: The adaptation loop knows one iteration interface and one time interface.

	scalar	vector valued
stationary	*ProblemScal*	*ProblemVec*
instationary	*ProblemInstatScal*	*ProblemInstatVec*

Table 4.1: The problem classes.

- `setTime()`: Called by the adaptation loop to set the current time. An implementation e.g. should update the time value in time dependent functions.

- `solveInitialProblem()`: Solves the initial problem which delivers the initial solution. An implementation e.g. can solve a stationary problem to create the initial solution. But it also can set the initial solution directly or interpolate a given function to it.

In Figure 4.6, the dependencies between the adaptation loop, the iteration interface and the time interface are shown: The adaptation loop knows one iteration interface and one time interface.

4.1.4   Problem classes

In AMDiS, the problem classes are default problem implementations that can be used as starting points for user defined stationary and instationary problems.

The default implementation of scalar stationary problems is the class *ProblemScal*, the vector-valued version is the class *ProblemVec*.

Stationary problems are called by any implementation of the iteration interface (see Section 4.1.2) and implement the steps *assemble*, *solve*, *estimate* and *adaptMesh*. Furthermore, they contain the problem definition (operators, boundary conditions), the finite element spaces (meshes, basis functions, DOF administrations) and the needed DOF vectors and matrices.

The default implementations of time dependent (instationary) problems are the classes *ProblemInstatScal* and *ProblemInstatVec*.

An instationary problem implements the time interface 4.1.3. It creates an initial solution (initial condition of the PDE) and it knows a stationary problem which is used as *space problem* that is solved within every time step. When the current simulation time is changed by the calling adaptation loop, the instationary problem sets the new time where necessary (e.g in time dependent functions).

In Table 4.1, the four problem classes are categorized.

Figure 4.7 shows the four main abstraction levels of AMDiS: The highest level is the adaptation loop which calls methods of the iteration and time interface (second level). On the third level, the problem classes implement the needed methods using concrete algorithmic classes (assembler, solver, ...) working on concrete data classes (finite element spaces, operators, vectors, ...).

4.2   Error estimation

In principle, one can implement specialized error estimators for every concrete problem which are matched to the concrete properties of the problem. But often it is sufficient to use predefined
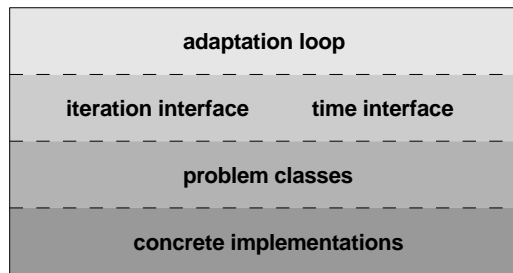
Figure 4.7: The four main abstraction levels in AMDiS.

estimators that can be automatically constructed using the abstract problem definition given by its operators (see Section 3.5.3). An a posteriori error estimator in AMDiS must provide:

- An estimation η of the global error $||u - u_h||$ in a given norm. The global error estimation is used as abort criterion for the adaptation loop.

- Estimations η_S of local element errors for every element $S \in \mathcal{T}$. The local estimations are used as basis for the decision which elements have to be refined or coarsened. They are stored as `ElementData` objects at leaf elements in the mesh data structure.

- An estimation of the time discretization error η_t which is used for timestep adaptive strategies.

- An estimation of the coarsening error η_S^c (optionally). Coarsening of mesh elements can lead to additional errors. If an upper bound for this error is known, it should be considered by the adaptation strategy when the mesh is coarsened.

In AMDiS, by default a residual based estimator is used which contains:

- The norm of the pointwise residual $L(u_h) - f$ (L^2 norm or H^1 semi-norm), where L is the operator containing all left hand side terms of the PDE, f is the right hand side of the PDE and u_h is the discrete solution.

- A jump residual which considers the jump of the normal component of $A\nabla u_h$ across interior edges/faces.

- A boundary residual which is the analog to the jump residual at boundary edges/faces.

One can prove that under suitable assumptions such residual based estimates are an upper bound for the real error in L^2 norm (see [9]) or in H^1 semi-norm (see [53]).

In the context of parallel computations in AMDiS (see Chapter 6), every partition computes on the whole domain but on a very coarse mesh outside of its partition. But the solution quality within the local partition can also depend on the mesh resolution outside of the partition. A very coarse mesh outside of the partition can cause a so called *pollution error* within the partition. In [41] this aspect is considered by so called *goal oriented error estimation*.

Another estimator that is implemented in AMDiS is the *recovery estimator*, which is based on the recovery gradient of the solution. In general, the gradient of the finite element solution is discontinuous at element boundaries. The recovery gradient is a smoothed continuous version of this gradient.

For Systems of PDEs, every component in the system can have its own estimator. On every element the local estimations from all estimators are stored separately.

4.3 Adaptation strategies

The adaptation strategy determines which elements of mesh $\mathcal{T}$, consisting of the current leaf elements, are marked for coarsening or refinement based on the local error estimates. The global error estimate is denoted by η, the local estimate for element S by η_S. Coarsening can lead to additional errors. η_S^c denotes the upper bound of this coarsening error for element S.

So far in AMDiS, the following adaptation strategies are implemented, which are already described in [44]:

- **Global refinement strategy:** Every element of the mesh is marked for refinement. This leads to the best error reduction but also to the highest computational effort. It can be useful for some initial global refinements of the macro mesh. Coarsening is not done in this strategy.

- **Maximum strategy:** An element S is marked for refinement if

$$\eta_S > \gamma \max_{S' \in \mathcal{T}} \eta_{S'} \tag{4.1}$$

  for a given parameter $\gamma \in (0, 1)$. A small γ leads to many refinements and to a fast error reduction but maybe also to too many unknowns. A larger γ leads to more adaptive iterations but to a better mesh. A common value for γ is $\frac{1}{2}$.

  The element is marked for coarsening if

$$\eta_S + \eta_S^c \leq \gamma_c \max_{S' \in \mathcal{T}} \eta_{S'} \tag{4.2}$$

  for a given parameter $\gamma_c \in (0, 1)$, $\gamma_c < \gamma$.

- **Equidistribution strategy:** Let the global error estimation η be given by

$$\eta = \left(\sum_{S \in \mathcal{T}} \eta_S^p \right)^{1/p}, \quad p \in [1, \infty), \tag{4.3}$$

  and let N be the number of elements in $\mathcal{T}$. If the error is equidistributed over all mesh elements, i.e. $\eta_S = \eta_{S'}$ for all $S, S' \in \mathcal{T}$, then

$$\eta = N^{1/p} \eta_S. \tag{4.4}$$

  If we further assume that $\eta = tol$ with tol the adaptation tolerance, it holds

$$\eta_S = \frac{tol}{N^{1/p}}. \tag{4.5}$$

  An element S is marked for refinement if

$$\eta_S > \theta \frac{tol}{N^{1/p}}. \tag{4.6}$$

  It is marked for coarsening if

$$\eta_S + \eta_S^c \leq \theta_c \frac{tol}{N^{1/p}}. \tag{4.7}$$

  The parameters θ and θ_c are used to make the method more robust. They both are chosen from the interval $(0, 1)$ with $\theta_c < \theta$ and $\theta \approx 1$.
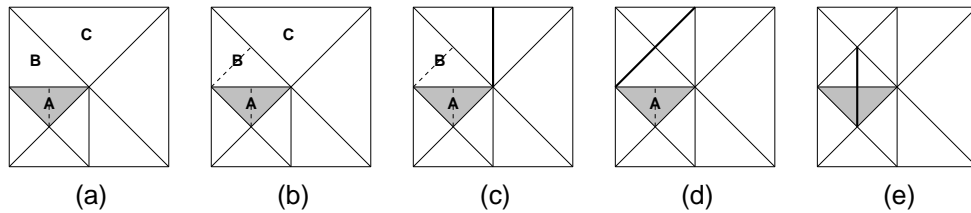
Figure 4.8: Refinement of element A at its longest edge. To avoid hanging nodes also elements B and C have to be bisected at their refinement edges.

- **Guaranteed error reduction strategy:** For most marking strategies, convergency is not guaranteed. Dörfler introduced in [18] a method with a guaranteed error reduction for the Poisson equation if the properties of the problem are sufficiently resolved by the current mesh and if every edge of a marked element is at least bisected once in one refinement step. For details of this strategy, see [44].

For systems of PDEs, every component can have its own adaptation strategy. Since all components are discretized on the same mesh, refinements are done if at least one element of a refinement patch is marked for refinement by at least one component. Coarsening is done only if all elements of a coarsening patch are marked for coarsening by all components.

AMDiS is not restricted to the adaptation strategies described here. One can implement his own strategy by deriving from the class `Marker`.

4.4 Refinement and coarsening

The bisection of a simplex is always done by bisecting a designated refinement edge of the element. The midpoint of the refinement edge then is the new vertex of both new elements (see Figure 4.1).

The refinement edge can be chosen in several ways. One possibility is to choose the longest edge of the element. In AMDiS, the refinement edge is always the edge connecting the vertices with local indices 0 and 1. So, the refinement edge of an element is given by the definition of the corresponding macro element together with the vertex numbering during refinement. In 1d and 2d, the numbering of child vertices is straightforward (see Figure 4.1). In 3d, it depends on the element type of the parent tetrahedron, which can be 0, 1 or 2 (the children's type then is (parent's type + 1) modulo 3). This element type distinction allows a refinement algorithm that definitely terminates (see [44]).

If an element is marked for refinement, all elements that share the refinement edge of this element have to be refined, too. These elements build the *refinement patch*. The refinement is allowed only if the common edge is the refinement edge of all elements in the patch. An element in the patch that has a different refinement edge first has to be refined. Then it is replaced by its child that is located at the common edge. The corresponding recursive algorithm can result in endless recursion if the refinement edges are not chosen properly. One can avoid an endless recursion by redefining the refinement edges of the maybe refined macro mesh, which is done in AMDiS automatically if necessary. For this reason, also the different element types are necessary in 3d.

In Figure 4.8, the recursive refinement algorithm in 2d is illustrated. Element A should be bisected at its longest edge. To avoid a hanging node, first element B has to be bisected. But the refinement edge of element B is not the common edge with element A. To be able to refine element B, first Element C has to be refined.

Analog to the refinement patch a *coarsening patch* is constructed to avoid hanging nodes during coarsening. A refinement operation is performed if one element in the refinement patch is marked for refinement. A coarsening operation is performed only if all elements in the coarsening

patch are marked for coarsening. This ensures that only elements with a sufficient small error are coarsened.

In the following, the main operations are listed that must be applied during refinement and coarsening.

- **Changes of mesh topology:** Elements must be created during refinement and deleted during coarsening. The corresponding topological changes must be considered in the hierarchical mesh structure.

- **DOF administration:** During refinement, new DOF indices have to be created and assigned to the elements. DOF indices of coarser levels that are not used on finer levels may be deleted, if allowed (not allowed e.g. if multigrid or parallelization is used). During coarsening, DOF indices of deleted elements are freed, if they are not part of other, still existing elements. Coarse level DOF indices that were freed during refinement must be reallocated.

- **Transfer of geometric data:** As described in Sections 3.2 and 3.3, geometric data usually is stored only at macro level. During mesh traversal, the corresponding data for finer levels is computed.

  In the case of curved boundaries, the coordinates of new vertices must be stored explicitly in the mesh structure. Furthermore, if parametric finite elements are used, the vertex coordinates may be stored in DOF vectors that must be actualized during refinement and coarsening.

- **Transfer of finite element data:** DOF vectors have to be interpolated to the finer mesh after refinement and restricted to the coarser mesh after coarsening. This is necessary to have a useful initial guess for the DOF vectors on the new mesh.

To separate the mesh data structure from the used refinement and coarsening algorithms, the visitor design pattern is used (see [23]). The algorithms are implemented within refinement and coarsening managers (*visitors*) that know the mesh structure and how its elements can be iterated. This separation allows to implement new refinement and coarsening algorithms without modifications of the mesh data structure.

Chapter 5

Multigrid

The multigrid method is an efficient way to solve large linear systems of equations as they appear in AMDiS. The main idea is to solve the system using a hierarchy of grids.

Classical iterative methods often have a strong smoothing effect on the error even if they have a bad convergence rate. The idea of multigrid is to treat the high frequencies of the solution by smoothing on a fine grid, whereas the low frequency components are computed with less computational effort on a coarser grid. If n is the number of unknowns in the linear system of equations, the solution can be obtained in $\mathcal{O}(n)$ time by multigrid methods.

I start with a description of some iterative methods in Section 5.1. In Section 5.2, the basics of the multigrid method are introduced. Section 5.3 describes the multigrid implementation in AMDiS.

5.1   Iterative methods

Consider the linear system of equations $Ax = b$ with matrix $A \in \mathbb{R}^{n \times n}$ and right hand side $b \in \mathbb{R}^n$. The matrix A is given by its entries a_{ij} with $i, j = 1, \ldots, n$. Iterative methods compute successive approximations to the solution by repeatedly applying the mapping

$$x_{m+1} = \phi(x_m, b), \quad m \in \mathbb{N}_0 \tag{5.1}$$

with given initial guess x_0.

The iterative method is called linear if matrices $M, N \in \mathbb{R}^n$ exist such that

$$\phi(x, b) = Mx + Nb \tag{5.2}$$

holds. M is called the iteration matrix.

5.1.1   Splitting methods

In splitting methods, the matrix A is written as

$$A = B + (A - B), \ B \in \mathbb{R}^n. \tag{5.3}$$

The linear system of equations $Ax = B$ then can be written as

$$Bx = (B - A)x + b \tag{5.4}$$

and if B is invertible as

$$x = B^{-1}(B - A)x + B^{-1}b. \tag{5.5}$$

Now a linear iterative method can be defined as

$$x_{m+1} = \phi(x_m, b) = M x_m + N b, \quad m \in \mathbb{N}_0 \tag{5.6}$$

with

$$M = B^{-1}(B - A) \tag{5.7}$$

and

$$N = B^{-1}. \tag{5.8}$$

In the following sections, two splitting methods are introduced: The Jacobi method and the Gauss-Seidel method.

Jacobi method

In the Jacobi method, the matrix B is set to

$$D = (d_{ij})_{i,j=1,\ldots,n} \ \text{ with } \ d_{ij} = \left\{ \begin{array}{ll} a_{ij}, & i = j \\ 0, & \text{otherwise.} \end{array} \right. \tag{5.9}$$

The resulting iterative method is

$$x_{m+1} = D^{-1}(D - A)x_m + D^{-1}b, \quad m \in \mathbb{N}_0. \tag{5.10}$$

In component notation:

$$x_{m+1,i} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1, j \neq i}^{n} a_{ij} x_{m,j} \right), \quad i = 1, \ldots, n. \tag{5.11}$$

Gauss-Seidel method

In the Gauss-Seidel method the matrix A is written as $R + D + L$ with

$$R = (r_{ij})_{i,j=1,\ldots,n} \ \text{ with } \ r_{ij} = \left\{ \begin{array}{ll} a_{ij}, & i < j \\ 0, & \text{otherwise} \end{array} \right. \tag{5.12}$$

and

$$L = (l_{ij})_{i,j=1,\ldots,n} \ \text{ with } \ l_{ij} = \left\{ \begin{array}{ll} a_{ij}, & i > j \\ 0, & \text{otherwise.} \end{array} \right. \tag{5.13}$$

D is defined as in equation (5.9).

B is set to $D + L$. The resulting iterative method is:

$$x_{m+1} = -(D + L)^{-1} R x_m + (D + L)^{-1} b, \quad m \in \mathbb{N}_0. \tag{5.14}$$

In component notation:

$$x_{m+1,i} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_{m+1,j} - \sum_{j=i+1}^{n} a_{ij} x_{m,j} \right), \quad i = 1, \ldots, n. \tag{5.15}$$

5.1.2 Damped methods

We rewrite equation 5.6 as

$$x_{m+1} = x_m + r_m \tag{5.16}$$

with

$$r_m = B^{-1}(b - Ax_m). \tag{5.17}$$

The vector x_{m+1} can be seen as correction of vector x_m with correction vector r_m. We modify equation (5.16) by weighting the correction with the relaxation parameter $\omega \in \mathbb{R}^+$:

$$
\begin{aligned}
x_{m+1} &= x_m + \omega r_m \\
&= x_m + \omega B^{-1}(b - Ax_m) \\
&= (I - \omega B^{-1}A)x_m + \omega B^{-1}b.
\end{aligned}
\tag{5.18}
$$

Damped Jacobi method

The damped Jacobi method in component notation reads:

$$x_{m+1,i} = (1 - \omega)x_{m,i} + \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_{m,j} \right), \quad i = 1, \ldots, n. \tag{5.19}$$

Damped Gauss-Seidel method

The damped Gauss-Seidel method in component notation reads:

$$x_{m+1,i} = (1 - \omega)x_{m,i} + \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_{m+1,j} - \sum_{j=i+1}^{n} a_{ij} x_{m,j} \right), \quad i = 1, \ldots, n. \tag{5.20}$$

If A is a self-adjoint and positive-definite matrix, the damped Gauss-Seidel methods is convergent if and only if $\omega \in (0, 2)$.

5.2 Multigrid basics

The multigrid basics explained in this section are based on the principles described in [51] in the context of finite differences. These concepts are adjusted and expanded to fit in the finite element context using the hierarchical mesh structure of AMDiS.

First, in Section 5.2.1 the main multigrid principles are explained, followed by a short enumeration of needed components for the multigrid method in Section 5.2.2. Basis for a multigrid cycle is the two-grid cycle which is introduced in Section 5.2.3. Afterwards, the multigrid cycle is explained in Section 5.2.4. To obtain a good initial guess for the multigrid method, the *full multigrid method* can be applied which is topic of Section 5.2.5.

5.2.1 Multigrid principles

The multigrid idea can very shortly be summarized by the following two principles:

- **Smoothing principle**: Classical iterative methods often have a strong smoothing effect on the error.

- **Coarse grid principle**: A quantity that is smooth on a certain grid can be approximated on a coarser grid without any essential loss of information.

	0 iterations	5 iterations	10 iterations	15 iterations
approx. solution				
pointwise error				
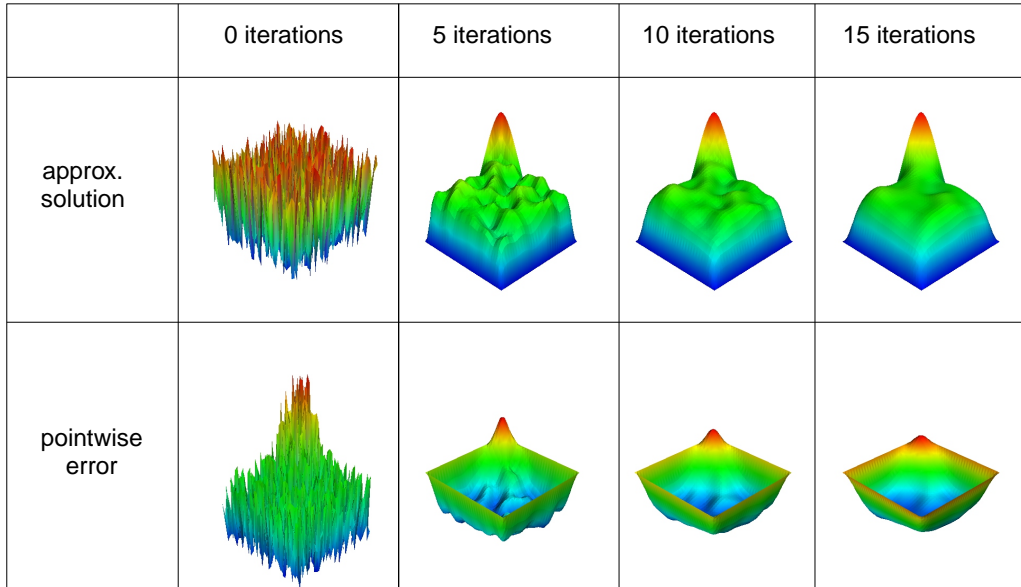
Figure 5.1: Influence of the Gauss-Seidel method on the error.

Iterative methods with a strong smoothing effect e.g. are splitting methods as the Jacobi method or the Gauss-Seidel method. In Figure 5.1 one can see the influence of some Gauss-Seidel iterations on the pointwise error for the Poisson equation $-\Delta u = f$ with $f = (400x^2 - 40)e^{-10x^2}$ on $\Omega = [0,1]^2$ starting with a randomized initial guess. The method has a bad convergence rate, but the error becomes smooth very quickly. Now the second principle can be applied: If the error is smooth on a given grid, it can be approximated on a coarser grid. In Section 5.2.3 it is shown, how the error can be computed on the coarser grid as solution of the defect equation, using the restricted residual of the fine grid equation. The error then is prolongated to the fine grid and used as correction for the fine grid solution. These ideas are used in the so-called *two-grid cycle*:

1. Perform a few smoothing steps on the finest level (pre-smoothing).

2. Compute the residual and transfer it to a coarser level.

3. Compute a correction on the coarser level by solving a so called defect equation.

4. Transfer the correction to the fine level and add it to the fine level solution.

5. Perform a few smoothing steps on the finest level (post-smoothing).

To solve the defect equation in step 3, multigrid can be applied recursively, until the coarsest level is reached. On the coarsest level with only a few degrees of freedom, a direct method as Gauss elimination can be applied. Alternatively, just some additional smoothing iterations can be performed. The full recursion is called a *multigrid cycle*.

5.2.2   Multigrid components

The multigrid initialization phase consists of two main steps:

- **Creating different level grids**: First, the grids of the different multigrid levels must be created. Starting from the finest level, grids of coarser levels are created until the coarsest multigrid level is reached. In the context of the hierarchical mesh structure in AMDiS, the given mesh levels can be used to construct the different level grids.

- **Creating level operators**: At each multigrid level, a defect equation must be solved. The left hand side of this equation is a coarser version of the left hand side operator of the finest level. Since it does not depend on the current approximation, it can be computed in advance in the initialization phase.

After the initialization phase, some multigrid iterations or *multigrid cycles* are performed until a given tolerance criterion or the maximal cycle number is reached. The components of a multigrid cycle are:

- **Smoothing**: Reduces the high frequencies in the error on fine grids.

- **Restriction**: The residual of the fine grid equation is transferred to the coarse grid. In the Full Approximation Scheme (described in Section 5.2.3) also the smoothed approximation to the solution is restricted. The Full Approximation Scheme can be used for nonlinear PDEs and for an efficient implementation of multigrid together with adaptively refined meshes.

- **Coarse grid right hand side construction**: To compute a coarse grid correction, the defect equation is solved on the coarse level (see Section 5.2.3). The left hand side of this equation is given by the level operators created in the initialization phase. The right hand side depends on the restricted residual and is constructed within the multigrid cycle.

- **Solving the coarse grid equation**: The defect equation on the coarser level is solved by multigrid recursively until the coarsest level is reached. The solution of the coarse grid equation is used for the correction of the fine grid solution.

- **Prolongation**: The correction is transferred to the fine grid and added to the fine grid solution.

5.2.3   The two-grid cycle

Before we define the multigrid cycle, we first define the two-grid cycle for the fine grid level l and the coarse grid level $l-1$.

The conventional two-grid cycle

Consider the linear discrete problem $L_l u_l = f_l$ on the fine level. We start with a guess u_l^m for the solution u_l. The upper index m denotes the iteration number. The result of the two-grid cycle will be an improved guess u_l^{m+1} for the fine level.

For any approximation u_l^m, we denote the error $\nu_l^m := u_l - u_l^m$ and the residual (or defect) $d_l^m := f_l - L_l u_l^m$. Then the following holds:

$$
\begin{aligned}
L_l u_l = f_l \quad &\Leftrightarrow\quad L_l(u_l^m + \nu_l^m) = f_l \\
&\Leftrightarrow\quad L_l \nu_l^m = f_l - L_l u_l^m \\
&\Leftrightarrow\quad L_l \nu_l^m = d_l^m.
\end{aligned}
\tag{5.21}
$$

The last equation is called the defect equation and can be solved on the coarse level if the error ν_l^m is sufficiently smooth (see Section 5.2.1). The conventional two-grid cycle consists of the following steps:

1. Apply some pre-smoothing steps on the fine level ($u_l^m \to \bar{u}_l^m$).

2. Compute the residual $d_l^m = f_l - L_l \bar{u}_l^m$.

3. Restrict the residual to the coarse level ($d_{l-1}^m := R_l^{l-1} d_l^m$).

4. Solve the defect equation $L_{l-1}\nu_{l-1}^m = d_{l-1}$ on the coarse level.
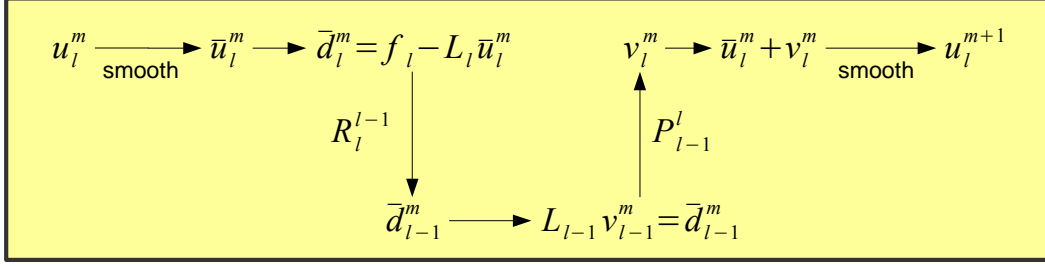
Figure 5.2: Structure of the conventional two-grid cycle.

5. Prolongate the solution of the defect equation to the fine level ($\nu_l^m := P_{l-1}^l \nu_{l-1}^m$).

6. Correct the solution on the fine level ($\bar{u}_l^m = \bar{u}_l^m + \nu_l^m$).

7. Apply some post-smoothing steps on the fine level ($\bar{u}_l^m \to u_l^{m+1}$).

In Figure 5.2 the structure of the two-grid cycle is illustrated.

Full Approximation Scheme

Consider the (not necessarily linear) discrete problem $N_l u_l = f_l$. Since we do not assume that N_l is a linear operator, the defect equation 5.21 may not hold. Here, the defect equation is constructed as follows:

$$
\begin{aligned}
L_l u_l = f_l \quad &\Leftrightarrow \quad N_l(u_l^m + \nu_l^m) = f_l \\
&\Leftrightarrow \quad N_l(u_l^m + \nu_l^m) - N_l u_l^m = d_l^m.
\end{aligned}
\tag{5.22}
$$

We define $w_l^m := u_l^m + \nu_l^m$ and rewrite equation 5.22:

$$
N_l w_l^m = d_l^m + N_l u_l^m.
\tag{5.23}
$$

This equation now can be solved on the coarse level, after the residual and the solution have been restricted to the coarse grid. But the smoothing steps have a smoothing effect on the error and not on the approximation of the solution. So the solution may not be approximated well by w_{l-1}^m on the coarse level. Therefore, after solving the defect equation on the coarse level, we compute the correction $\nu_{l-1}^m = w_{l-1}^m - u_{l-1}^m$, which is an approximation to the error and is represented well on the coarse grid. Then we prolongate the correction to the fine level and add it to the fine level solution.

The two-grid cycle for the Full Approximation Scheme then reads:

1. Apply some pre-smoothing steps on the fine level ($u_l^m \to \bar{u}_l^m$).

2. Compute the residual $d_l^m = f_l - N_l \bar{u}_l^m$.

3. Restrict the residual and the solution to the coarse level ($d_{l-1}^m := R_l^{l-1} d_l^m$, $\bar{u}_{l-1}^m := R_l^{l-1} \bar{u}_l^m$).

4. Solve the defect equation $N_{l-1} w_{l-1}^m = d_{l-1} + N_{l-1} \bar{u}_{l-1}^m$ on the coarse level.

5. Compute the correction $\nu_{l-1}^m := w_{l-1}^m - \bar{u}_{l-1}^m$ on the coarse level.

6. Prolongate the correction to the fine level ($\nu_l^m := P_{l-1}^l \nu_{l-1}^m$).

7. Correct the solution on the fine level ($\bar{u}_l^m = \bar{u}_l^m + \nu_l^m$).
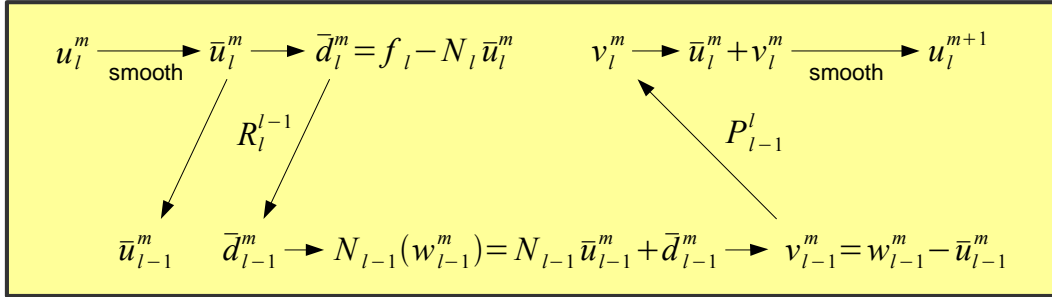
Figure 5.3: Structure of the two-grid cycle for the Full Approximation Scheme.

8. Apply some post-smoothing steps on the fine level ($\bar{u}_l^m \rightarrow u_l^{m+1}$).

If N_l is a nonlinear operator, also the pre- and post-smoothing steps are implemented by nonlinear relaxation procedures. In the Full Approximation Scheme the solution of the defect equation is a full approximation to the solution and not only a correction. This fact is used for the efficient multigrid implementation in the context of adaptively refined meshes, as they are used in AMDiS (see Section 5.3).

In Figure 5.3 the structure of the Full Approximation Scheme two-grid cycle is illustrated.

5.2.4   The multigrid cycle

The two-grid cycle, described in Section 5.2.3, is an iterative method to successively improve the solution on a fine grid by solving the defect equation on a coarser grid. In a multigrid cycle, the defect equation on the coarser level now is solved recursively using multigrid. On the coarsest level a direct solver is applied. The multigrid cycle is described by Algorithm 12. The only parameter in this view is the multigrid level l. The algorithm initially is called for the finest level.

Algorithm 12 multiGridCycle(l)

 1: **if** $l = minLevel$ **then**
 2:    exact solve on level l
 3: **else**
 4:    σ_{pre} pre-smoothing steps on level l
 5:    compute residual on level l
 6:    restrict residual (and possibly solution) to level $l-1$
 7:    compute right hand side for level $l-1$
 8:    **for** $i = 0$ to $cycleIndex$ **do**
 9:       **multiGridCycle**($l-1$)
10:    **end for**
11:    compute correction on level $l-1$
12:    prolongate correction to level l
13:    fix solution on level l
14:    σ_{post} post-smoothing steps on level l
15: **end if**

Depending on the used cycle scheme (conventional or Full Approximation Scheme) some lines in the algorithm have different meanings. In line 6, the residual is restricted to the coarser level. If the Full Approximation Scheme is used, also the solution is restricted. Since the defect equation differs for the two cycle schemes, in line 7 the right hand side for the coarse level is computed like it is needed for the used cycle scheme. The computation of the correction in line 11 is necessary
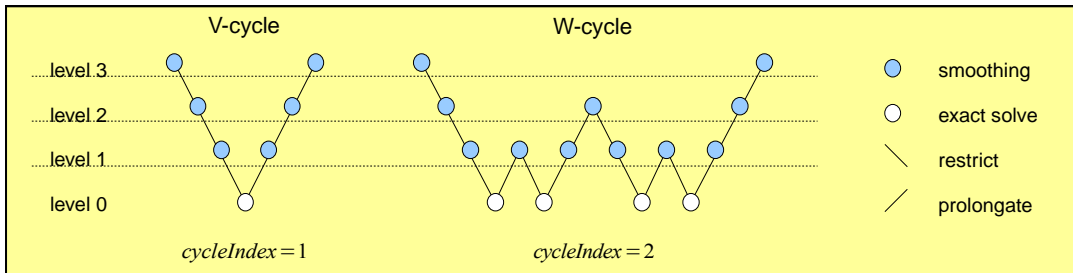
Figure 5.4: V-cycle ($cycleIndex = 1$) and W-cycle ($cycleIndex = 2$).
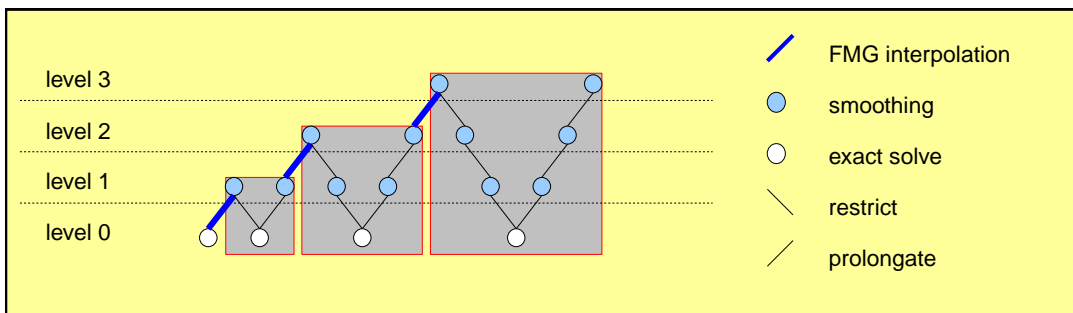
Figure 5.5: Structure of Full Multigrid.

only for the Full Approximation Scheme. In the conventional scheme, the correction is the solution of the defect equation.

In the loop from line 8 to line 10, **multiGridCycle** is called $cycleIndex$ times with level $l - 1$. The value of $cycleIndex$ determines the multigrid cycle type. If $cycleIndex$ is 1, the algorithm describes a so-called V-cycle. If the value is 2, this results in a W-cycle. Figure 5.4 illustrates a V- and W-cycle starting on level 3.

5.2.5 Full Multigrid

So far, the multigrid method was started with an arbitrary initial guess. To get a good initial guess the method of *full multigrid* (FMG) can be applied. The FMG algorithm starts at the coarsest multigrid level and computes an exact solution there. Then this solution is prolongated to the next finer level and it is used as initial guess for one multigrid cycle starting at this level. The solution after this cycle again is prolongated to the next finer level and again a multigrid cycle is applied. This procedure is repeated until the finest multigrid level is reached. In Algorithm 13, the FMG method is shown. The operator Π_{i-1}^{i} in line 3 is the FMG prolongation from level $i - 1$ to level i.

Algorithm 13 fullMultiGrid($level$)

1: Solve $L_{minLevel}u_{minLevel} = f_{minLevel}$
2: **for** $i = minLevel + 1$ to $i = level$ **do**
3:     $u_i = \Pi_{i-1}^{i}u_{i-1}$
4:     **multiGridCycle**(i)
5: **end for**

In Figure 5.5, the structure of FMG with $cycleIndex = 1$ is illustrated.

In the context of adaptive finite elements, usually the solution of the last adaptive iteration can be used as initial guess. Therefore, here FMG in most cases is not necessary.

(a) DOF locations in the parent triangle.   (b) DOF locations in the two child elements.

Figure 5.6: DOF locations for parent and child triangles with Lagrange basis functions of degree 2. The DOF at the longest parent edge by default is deleted after refinement, but must be preserved for multigrid. The • symbol stands for vertex DOFs, the ∘ symbol for edge DOFs.

5.3 Multigrid in AMDiS

In this section, the multigrid implementation in the context of AMDiS is discussed. Using the hierarchical mesh data structure of AMDiS, the different multigrid levels can be constructed in an easy way. But the adaptivity of AMDiS meshes also causes some difficulties, because finer levels in general cover a smaller part of the domain than coarser levels do.

5.3.1 Preservation of coarse DOFs

Usually, in finite element computations one is interested in the solution on the finest mesh. In the hierarchical mesh structure of AMDiS, this is the mesh consisting of all leaf elements. If Lagrange basis functions with $degree > 1$ are used, coarser elements always contain DOFs that do not belong to the leaf mesh. In Figure 5.6, this is illustrated for two dimensional elements of degree 2. In Figure 5.6 (a), a triangle with one DOF located at each vertex and at each edge is shown. Figure 5.6 (b) shows the two children of the element and the corresponding DOFs.

The parent DOF at the longest edge is not part of one of the two children. It is replaced by a new *vertex* DOF (with new index) at the finer level. By default, the parent DOF is freed after refinement for efficiency reasons.

In the multigrid context, solutions have to be computed also on coarse mesh levels, so one has to ensure that coarse DOFs are not freed, even if they are not part of finer mesh levels.

5.3.2 Choice of level grids

AMDiS meshes are stored in a hierarchical data structure. For each macro element a binary tree is stored, which contains the elements of the different refinement levels of this macro element (see Section 3.2). This structure can be used to construct the different multigrid levels in two different ways:

- Use all elements (leaf or not leaf) of a given level in the hierarchical structure to build the corresponding level mesh.

- Use all elements of a given level together with all leaf elements of coarser levels to build the composite level mesh.

Figure 5.7 illustrates the first approach. In Figure 5.7 (a), an example of an adaptively refined mesh M is shown. The Figures 5.7 (b)-(f) show the different refinement levels of this example. Here, M_l is the union of all mesh elements of level l, not only the leaf elements of level l but also further refined elements within this level.

mesh M M_0 M_1 M_2 M_3 M_4

(a)   (b)   (c)   (d)   (e)   (f)
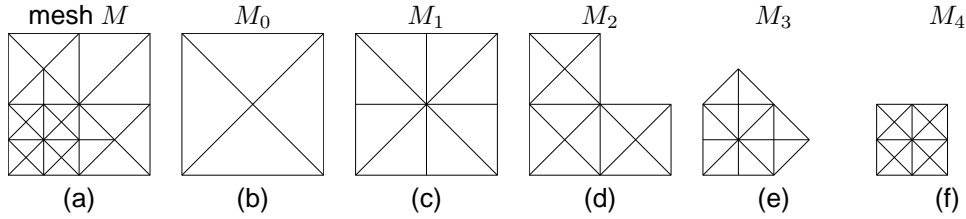
Figure 5.7: Hierarchy of mesh levels for an AMDiS mesh.

CM_0 CM_1 CM_2 CM_3 CM_4

(a)   (b)   (c)   (d)   (e)
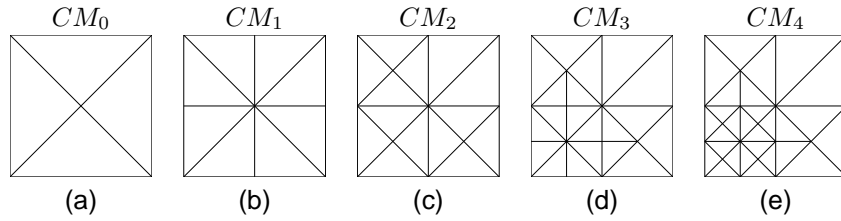
Figure 5.8: Composite meshes for the different levels.

The second approach is illustrated in Figure 5.8. Let L_i be the set of all leaf elements within level i. Then the composite mesh CM_l is defined as follows:

$$CM_l := M_l \cup \bigcup_{i=0}^{l-1} L_i. \tag{5.24}$$

In the first approach, less work has to be done at each level (if the mesh is not globally refined on the whole domain). But the coarse grid correction can be computed only in regions where a fine grid solution exists. So the Full Approximation Scheme (see Section 5.2.3) must be applied to get a useful coarse grid equation.

In the second case, the conventional two-grid cycle can be used, but every multigrid level covers the whole problem domain Ω.

In Section 5.3.4, the coarse grid correction schemes for these two methods are discussed.

To ensure that every edge length is halved between two multigrid levels, it is necessary to skip some mesh levels. If d is the dimension of the mesh, two multigrid levels should have a distance of d mesh levels.

In the second approach ignoring the mesh levels between two multigrid levels can lead to wrong results because there may exist discretization points (DOFs) that only belong to the skipped levels. Those points then never would be smoothed or corrected. Therefore, a multigrid level is defined as the union of mesh levels. To which multigrid level a given mesh level l belongs can be computed by equation

$$MGL(l) := \left\lceil \frac{l}{d} \right\rceil. \tag{5.25}$$

The mesh of multigrid level l then is defined by

$$MG_l := M_{d \cdot l} \cup \{L_i \mid MGL(i) = l \ \wedge \ i \neq d \cdot l\}. \tag{5.26}$$

In Figure 5.9, the meshes MG_0, MG_1 and MG_2 for the above example are shown.

If composite meshes are used to build the multigrid levels, the leaf elements of all coarser levels are already included ($MG_l \subset CM_{d \cdot l}$). In this case, the multigrid level l is defined by:

$$CMG_l := CM_{d \cdot l}. \tag{5.27}$$
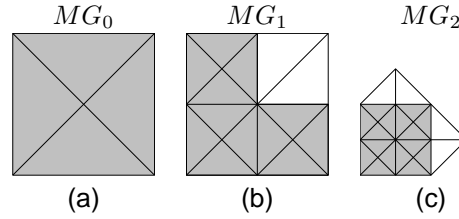
Figure 5.9: Multigrid levels for the two dimensional example. The grey elements correspond to M_{2l} ($l = 0, 1, 2$), the white elements to L_{2l-1}.
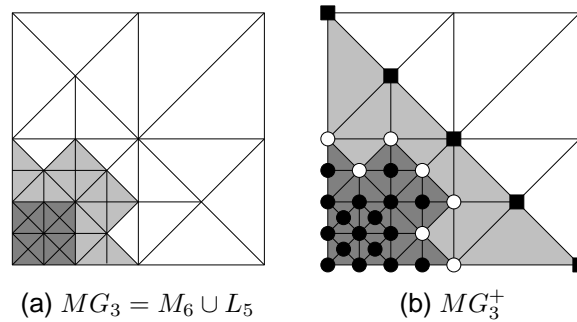
Figure 5.10: (a): Adaptively refined two dimensional mesh. (b): Extended multigrid level MG_3^+. •: Inner level points of MG_3. ○: Interface points of multigrid level 3 to coarser mesh levels. ■: Points of level extension.

5.3.3 Smoothing

In AMDiS, damped versions of the Jacobi method and of the Gauss-Seidel method are implemented. The smoothing is straightforward if composite meshes are used as multigrid levels (see Section 5.3.2). It has to be applied to all DOFs of the corresponding level.

   If multigrid levels are constructed using only the refined elements, also the smoothing should be applied only for these elements. In this case, one has to take special care of DOFs lying at the interface to coarser levels.

   The naive approach would be to smooth only the 'inner' DOFs of a multigrid level and treat the interface DOFs as Dirichlet boundary DOFs set by lower levels. But if the interface DOFs are only set by coarser levels and never corrected or smoothed on the finer level, the solution accuracy at such points is always limited by the coarse grid discretization.

   For that reason, the smoothing on a given multigrid level has to be applied also for DOFs lying at the interface to coarser levels. For a given DOF i the result of one smoothing step depends on all DOFs j belonging to elements that contain DOF i, because the corresponding matrix entries (i, j) in general are not zero. For the interface DOFs this means that also DOFs of coarser mesh levels have to be considered.

   Therefore, for multigrid level l the *extended multigrid level* MG_l^+ contains all elements of MG_l and all elements of coarser levels that include at least one interface DOF of the multigrid level. DOFs within the extension that are no interface DOFs are set by coarser levels. Furthermore, matrix contributions of the extension elements have to be added to the DOF matrix of multigrid level l.

   In Figure 5.10, this concept is illustrated by a simple example. Figure 5.10 (a) shows a adaptively refined triangular mesh. The mesh levels 5 and 6 are highlighted (light grey: level 5, dark grey: level 6). Multigrid level 3 is built by these two mesh levels (see Section 5.3.2). In Figure 5.10 (b), one can see multigrid level 3 (dark grey) and the corresponding extension (light grey). The DOFs marked by • are inner DOFs of the multigrid level, the DOFs marked by ○ are interface
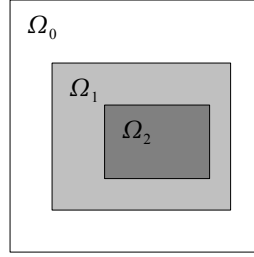
Figure 5.11: Nested multigrid level domains.

DOFs to coarser mesh levels. The ■ symbol marks the DOFs of the extension.

5.3.4 Coarse grid correction

Also the computation of the coarse grid correction depends on how the meshes for the different multigrid levels are constructed (see Section 5.3.2).

If the composite meshes are used, the conventional two-grid cycle can be applied. Here, on the coarse multigrid level $l-1$ the defect equation $L_{l-1}\nu_{l-1} = d_{l-1}$ is solved, where d_{l-1} is the residual of the fine grid solution on multigrid level l restricted to multigrid level $l-1$.

If multigrid levels are constructed only from elements of the corresponding mesh levels (without leaf elements of coarser levels), the corresponding meshes may not cover the whole problem domain Ω. If Ω_l denotes the domain covered by the mesh of multigrid level l it holds

$$\Omega_l \subset \Omega_{l-1} \subset \cdots \subset \Omega_0 = \Omega, \tag{5.28}$$

like illustrated in Figure 5.11.

As described in Section 5.3.3, smoothing on multigrid level l is done only for DOFs that belong to the mesh MG_l, which covers the domain $\Omega_l \subset \Omega_{l-1}$. At level l, an approximative solution and the corresponding residual can be computed only within Ω_l. Also the restricted residual d_{l-1} is defined only for the coarse grid DOFs in Ω_l. Solving $L_{l-1}u_{l-1} = f_{l-1}$ in $\Omega_{l-1} \setminus \Omega_l$ would lead to a wrong solution, because two inconsistent problems would be solved in one system of equations: The defect equation within Ω_l and the original (finest level) equation on the rest of Ω_{l-1}.

Here, the Full Approximation Scheme introduced in Section 5.2.3 can be used. Since the solution of the coarse grid correction is a full approximation to the solution of the fine grid equation, the coarse grid equations within Ω_l and in $\Omega_{l-1} \setminus \Omega_l$ now are consistent.

On the coarse level $l-1$, we solve:

$$\begin{aligned} L_{l-1}w_{l-1}^m &= d_{l-1} + L_{l-1}u_{l-1} & \text{in } \Omega_l \\ L_{l-1}u_{l-1} &= f_{l-1} & \text{in } \Omega_{l-1} \setminus \Omega_l. \end{aligned} \tag{5.29}$$

A coarse grid correction in needed only within Ω_l. In the rest of Ω_{l-1}, the level solution u_{l-1} is already the finest level solution. Following the Full Approximation Scheme, we compute the coarse grid correction by $\nu_{l-1} = w_{l-1} - u_{l-1}$, with u_{l-1} the restricted fine grid solution, and prolongate it to the finer level.

5.3.5 Transfer between the grids

The prolongation operator from the coarse multigrid level $l-1$ to the fine level l is denoted by P_{l-1}^l. The restriction operator from fine level l to coarse level $l-1$ is denoted by R_l^{l-1}. A coarse level vector u_{l-1} is prolongated to the finer level by

$$u_l = P_{l-1}^l u_{l-1}. \tag{5.30}$$
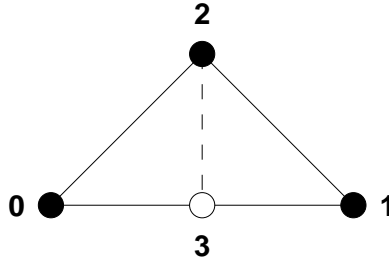
Figure 5.12: A refined triangle with vertex indices.

A fine level vector is restricted to the coarse level by

$$u_{l-1} = R_l^{l-1} u_l. \tag{5.31}$$

In the following, the indices of the operators are omitted ($P := P_{l-1}^l$, $R := R_l^{l-1}$).

The choice of P and R has a strong influence on the convergence rate of the multigrid method. Usually, they are defined such that P and R are adjoint operators, which means that

$$P = R^t \tag{5.32}$$

holds. In this case, also $u_{l-1} \cdot R u_l = P u_{l-1} \cdot u_l$ holds. This property is needed in many convergence proofs.

The default prolongation and restriction operators used in the AMDiS adaptation module do not fulfill this property. This is illustrated for a simple example in 2d with linear Lagrange basis functions located at element vertices. Consider a macro mesh consisting of one triangle which is refined once (see Figure 5.12). When the macro mesh is refined, a vector v is prolongated to the finer level by interpolating it from its coarse mesh values ($v[3] := \frac{1}{2}(v[0] + v[1])$). When the finer mesh is coarsened, the coarse grid values are interpolated from the fine grid values. In this linear example, this means that $v[3]$ is discarded and the other entries keep unchanged. The resulting matrices for P and R are

$$P = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ \frac{1}{2} & \frac{1}{2} & 0 \end{pmatrix} \text{ and } R = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{5.33}$$

Obviously equation 5.32 does not hold in this case. To let P and R be adjoint operators, we remain P unchanged and set $R := P^t$. The restriction matrix for the example then is

$$R = \begin{pmatrix} 1 & 0 & 0 & \frac{1}{2} \\ 0 & 1 & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{5.34}$$

For an efficient implementation of P and R two ideas are applied:

- P and R are sparse matrices and should be stored in a sparse way.

- Prolongation and restriction can be performed element-wise.

To illustrate the implementation, we look on a little more complex example with two macro elements and four fine elements, shown in Figure 5.13. For this example P and R are defined as follows:

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 \end{pmatrix}, \quad R = \begin{pmatrix} 1 & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}. \tag{5.35}$$
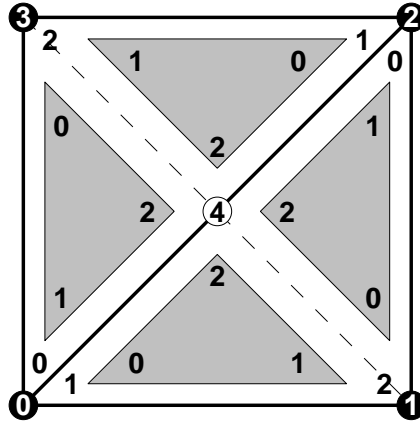
Figure 5.13: A macro mesh consisting of two triangles, which are bisected once. The numbers on the vertices are the global node indices. The numbers within the small grey triangles are the local node indices of the refined elements. The numbers outside of the small triangles are the local node indices of the coarse elements.

Now we define prolongation and restriction element matrices:

$$
p_1 = \left(
\begin{array}{c|ccc}
 & 2 & 0 & 1 \\
\hline
2 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 \\
4 & \frac{1}{2} & \frac{1}{2} & 0
\end{array}
\right), \quad
p_2 = \left(
\begin{array}{c|ccc}
 & 0 & 2 & 3 \\
\hline
0 & 1 & 0 & 0 \\
2 & 0 & 1 & 0 \\
3 & 0 & 0 & 1 \\
4 & \frac{1}{2} & \frac{1}{2} & 0
\end{array}
\right)
\tag{5.36}
$$

and correspondingly

$$
r_1 = \left(
\begin{array}{c|cccc}
 & 2 & 0 & 1 & 4 \\
\hline
2 & 1 & 0 & 0 & \frac{1}{2} \\
0 & 0 & 1 & 0 & \frac{1}{2} \\
1 & 0 & 0 & 1 & 0
\end{array}
\right), \quad
r_2 = \left(
\begin{array}{c|cccc}
 & 0 & 2 & 3 & 4 \\
\hline
0 & 1 & 0 & 0 & \frac{1}{2} \\
2 & 0 & 1 & 0 & \frac{1}{2} \\
3 & 0 & 0 & 1 & 0
\end{array}
\right),
\tag{5.37}
$$

where p_1 and r_1 are the matrices for the coarse element $2 - 0 - 1$, and the matrices p_2 and r_2 belong to the element $0 - 2 - 3$. The numbers before the first column and above the first row specify the global row and column indices of the element matrix entries.

Adding up the element matrix contributions to global prolongation and restriction matrices would lead to wrong results for P and R. Consecutive execution of element matrix–vector multiplication would produce the right result for the prolongation, but still the wrong for the restriction process (the value of DOF 4 is added twice to the DOFs 0 and 2 with factor $\frac{1}{2}$).

The problem is multiple appliance of rows (in the prolongation element matrices) and columns (in the restriction element matrices) which belong to the same DOF. This can be avoided by marking the corresponding rows/columns in the element matrices as *exclusive*. An exclusive row or column is treated only once per DOF. DOFs that are already treated are marked as visited and the corresponding exclusive rows/columns are ignored in later element matrices.

We can write the prolongation element matrix p and the restriction element matrix r in a unique way such that only the global row and column indices have to be filled for each element:

$$
p = \left(
\begin{array}{c|ccc|c}
 & cd_0 & cd_1 & cd_2 & ex \\
\hline
rd_0 & 1 & 0 & 0 & true \\
rd_1 & 0 & 1 & 0 & true \\
rd_2 & 0 & 0 & 1 & true \\
rd_3 & \frac{1}{2} & \frac{1}{2} & 0 & true
\end{array}
\right), \quad
r = \left(
\begin{array}{c|cccc}
 & cd_0 & cd_1 & cd_2 & cd_3 \\
\hline
rd_0 & 1 & 0 & 0 & \frac{1}{2} \\
rd_1 & 0 & 1 & 0 & \frac{1}{2} \\
rd_2 & 0 & 0 & 1 & 0 \\
\hline
ex & true & true & true & true
\end{array}
\right).
\tag{5.38}
$$

The vectors rd and cd store the global row and column indices corresponding to the local indices in the element matrix. They are filled in a mesh traversal during the prolongation/restriction process.

The matrix entries are stored in a sparse storage format. We store all exclusive rows in the matrix row^{ex} in the compressed row storage format (CRS) and all exclusive columns in the matrix col^{ex} in the compressed column storage format (CCS). All other entries are stored in the matrix row stored in CRS.

The prolongation element matrix in this example has only exclusive rows. It looks like:

$$\begin{aligned} row_0^{ex} : &\quad \{col = 0; \; entry = 1\} \\ row_1^{ex} : &\quad \{col = 1; \; entry = 1\} \\ row_2^{ex} : &\quad \{col = 2; \; entry = 1\} \\ row_3^{ex} : &\quad \left\{col = 0; \; entry = \tfrac{1}{2}\right\}, \quad \left\{col = 1; \; entry = \tfrac{1}{2}\right\}. \end{aligned} \tag{5.39}$$

The corresponding restriction matrix col^{ex} reads

$$\begin{aligned} col_0^{ex} : &\quad \{row = 0; \; entry = 1\} \\ col_1^{ex} : &\quad \{row = 1; \; entry = 1\} \\ col_2^{ex} : &\quad \{row = 2; \; entry = 1\} \\ col_3^{ex} : &\quad \left\{row = 0; \; entry = \tfrac{1}{2}\right\}, \quad \left\{row = 1; \; entry = \tfrac{1}{2}\right\}. \end{aligned} \tag{5.40}$$

This storage format has the following nice properties:

- Given an arbitrary element matrix the adjoint matrix can be computed automatically:

  - The vectors rd and cd have to be exchanged.
  - The matrix row^{ex} of the adjoint operator matrix is set to $(col^{ex})^t$ of the original operator matrix.
  - The matrix col^{ex} of the adjoint operator matrix is set to $(row^{ex})^t$ of the original operator matrix.
  - The matrix row of the adjoint operator matrix is set to row^t of the original operator matrix.

- Prolongation and restriction can be implemented by the same procedure **applyOperator** just using different element matrices.

The procedure **applyOperator**, described in Algorithm 14, implements the prolongation and restriction based on a prolongation element matrix p and the corresponding restriction matrix p^t. If $transposed$ is $false$, the procedure prolongates a vector v from mesh level $l - 1$ to mesh level l. Otherwise, the procedure restricts a vector v from mesh level l to mesh level $l - 1$. The result is written to vector w.

In line 1, all entries of the result vector w are set to zero. This is necessary because the result contributions are successively added to it. In the lines 2 and 3, all entries of the vectors $rowVisited$ and $colVisited$ are set to $false$. These vectors store which rows and columns are already treated. From line 4 to line 12, the local operator element matrix m is initialized. If $transposed$ is $false$, m is equal to p, otherwise it is set to p^t. In the loop starting at line 13, all non-leaf elements of the coarse mesh level $l - 1$ are traversed. The row and column DOF indices for m are set from line 14 to line 20 depending on the value of $transposed$. If $transposed$ is false, the row DOF indices of m are the fine level indices, and the column DOF indices of m are the coarse level indices. The order in which these global indices are stored is determined by the local node indices within the elements. Finally, the element-wise matrix-vector multiplication is applied in three steps. First, all exclusive rows are treated in line 21, then all exclusive columns are treated in line 22. The vectors $rowVisited$ and $colVisited$ are modified within the functions. Therefore, these arguments are handed over in a *call-by-reference* sense. In line 23, all non-exclusive entries are treated.

Algorithm 14 applyOperator(v, w, l, p, $transposed$)

1: clear w
2: clear $rowVisited$
3: clear $colVisited$
4: **if** $transposed == false$ **then**
5:    $m.row := p.row$
6:    $m.row^{ex} := p.row^{ex}$
7:    $m.col^{ex} := p.col^{ex}$
8: **else**
9:    $m.row := (p.row)^{t}$
10:   $m.row^{ex} := (p.col^{ex})^{t}$
11:   $m.col^{ex} := (p.row^{ex})^{t}$
12: **end if**
13: **for** all elements $el \in M_{l-1} \setminus L_{l-1}$ **do**
14:   **if** $transposed == false$ **then**
15:     fill $m.rd$ with the DOF indices of the children of el
16:     fill $m.cd$ with the DOF indices of el
17:   **else**
18:     fill $m.rd$ with the DOF indices of el
19:     fill $m.cd$ with the DOF indices of the children of el
20:   **end if**
21:   **treatExclusiveRows**(v, w, m, $rowVisited$)
22:   **treatExclusiveCols**(v, w, m, $colVisited$)
23:   **treatOtherEntries**(v, w, m)
24: **end for**

In Algorithm 15, the procedure **treatExclusiveRows** is shown. The procedures **treatExclusiveRows** and **treatOtherEntries** are implemented in a similar way.

To prolongate or restrict a vector between two multigrid levels, the procedure must be called d times, once for every mesh level that must be passed.

5.3.6   Coarse grid operators

A common way to construct the matrix A_{l-1} for the coarse level $l-1$ is to use the *Galerkin operator* $A_{l-1} = R \cdot A_l \cdot P$. The fine level matrix is restricted to the coarse level. The matrices P and R are the operators that are also used for the prolongation and restriction of vectors (see Section 5.3.5). The Galerkin operator minimizes the error in the energy norm over all coarse grid vectors after the coarse grid correction.

But the Galerkin operator is only applicable together with linear basis functions. For basis functions of higher degree, the level matrices becomes more and more dense for coarser levels, and the smoothing for each DOF becomes more and more expensive. The resulting multigrid method then would not be linear in the number of unknowns.

An alternative method to get the coarse grid operator is, to re-assemble the matrix for the elements of the extended coarse multigrid level (see Section 5.3.3). The matrix structure then is the same for each multigrid level: A matrix entry $A_l[i,j]$ can be different from zero only if there is at least one element on multigrid level l that contains both DOFs i and j.

5.3.7   Coarsest level solver

On the coarsest multigrid level, a direct solver like the Gauss elimination can be applied. If the coarsest mesh consist only of a very few DOFs, it can be sufficient to apply just some smoothing steps on the coarsest level.

Algorithm 15 treatExclusiveRows(v, w, m, $rowVisited$)

 1: **for** $i = 0, \ldots, \text{numRows}(m.row^{ex}) - 1$ **do**
 2:    **if** $rowVisited_{m.rd_i} == false$ **then**
 3:       $rowVisited_{m.rd_i} := true$
 4:       **for** $j = 0, \ldots, \text{numCols}(m.row_i^{ex}) - 1$ **do**
 5:          $c := m.row_{i,j}^{ex}.col$
 6:          $e := m.row_{i,j}^{ex}.entry$
 7:          $w_{m.rd_i} = w_{m.rd_i} + e \cdot v_{m.cd_c}$
 8:       **end for**
 9:    **end if**
10: **end for**
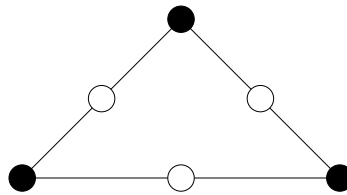
Figure 5.14: DOF locations of linear basis function (●) and additional locations of quadratic basis functions (○) on a triangle.

5.3.8 Sparse vectors

At different multigrid levels, vectors must store only entries for DOF indices that belong to the corresponding level. Therefore, it is useful to store the vectors in a sparse way, similar to the rows in a DOF matrix. If multiple vectors are stored at the same level (level solution, right hand side, correction vector, ...), they all have entries for the same indices. To store the sparse indices for every vector would produce redundant information. To avoid this redundancy, the sparse indices are stored only in one *master* vector at each level. All other vectors of this level only store a link to the the master indices.

5.3.9 Multigrid for systems of equations

The most multigrid components can be adopted to systems of PDEs in a straightforward way. In principle, this holds also for the smoothing. The generalization of the pointwise smoothing in the scalar case is the *pointwise collective* smoothing. Here, all DOFs that belong to the same mesh node are smoothed simultaneously. For a PDE system with N components this would result in a $N \times N$ system of equations for each node in each smoothing step. This small system of equations can be solved by a direct solver like the Gauss elimination.

In a *decoupled smoothing*, all DOFs are smoothed consecutively. This is easier to implement but less robust than the pointwise collective smoothing.

The collective smoothing is only possible if the different components have DOFs at the same mesh nodes. This is the case if the components are defined on the same finite element space with the same basis functions. If different basis functions are used, one can apply *box smoothing*. Here, all DOFs are smoothed simultaneously that belong to a set of mesh nodes. This set in AMDiS can be defined by all mesh nodes corresponding to one element. In Figure 5.14, one can see the DOF locations of linear and quadratic Lagrange basis functions on a triangle that are smoothed together. To avoid multiple smoothing of DOFs that belong to more than one element, already smoothed DOFs are marked as visited and will be ignored for further elements.

5.3.10 Multigrid as preconditioner

If multigrid provides a good convergence rate for a given problem, there is no need to use other solvers. But for complex problems, it is often very hard to choose the right multigrid components. Simple smoothers like pointwise damped Jacobi or Gauss-Seidel e.g. often lead to bad convergence rate for such problems. Also the type of prolongation and restriction can be important.

The goal of AMDiS is to provide a fast solver that is robust for a large class of problems and not to tailor specialized multigrid components for single problems.

A method to combine the robustness of Krylov subspace methods with the efficiency of multigrid is to use the multigrid method as a preconditioner for Krylov solvers. Preconditioners are used to reduce the condition number of the matrix and so to improve the solver convergence.

The linear system of equations $Ax = b$ is written as:

$$P_L A P_R x^P = P_L b \tag{5.41}$$
$$x = P_R x^P. \tag{5.42}$$

The matrix P_L is called *left preconditioner*, P_R is called *right preconditioner*.

In AMDiS, *implicit preconditioning* is used. This means, that the matrices P_L and P_R are not explicitly given but only their effect on a given vector. Let C be a right or left preconditioner. Its effect on a vector v is defined by the product $Cv = v'$. A useful choice of C should be an approximation to A^{-1}. Therefore, instead of computing the matrix vector product, we apply some multigrid iterations to the system of equations $Cv' = v$.

Chapter 6

Parallel concepts

In this chapter, a new parallelization concept for adaptive finite element methods is presented which has already been published in [54] and [40]. Compared to classical domain decomposition approaches, the concept of *adaptive full domain covering meshes* reduces the parallel communication overhead. Furthermore, it provides an easy way to transform sequential codes into parallel software by changing only a few lines of source code.

In traditional parallelization concepts, each process computes within a partition Ω_i of the full problem domain Ω. At partition boundaries a copy of needed neighboring data (shadow data) is stored to reduce the communication between the processes. But communication must be done anytime when the neighboring data changes. So, the parallel structure must be considered in many parts of the code. Therefore, parallelizing a sequential code is a complex matter if not only the solver is parallelized but the whole problem including setup, grid generation, assembly, solving, error estimation and visualization. Using adaptive full domain covering meshes can circumvent this complexity. Here, each process computes a solution on the whole domain Ω. But outside of the local partition Ω_i a relatively coarse mesh is used. At the end of computation, the different processes combine their solutions into one global solution by a partition of unity method.

Bank and Holst presented a similar technique in [5]. Their approach consists of the following steps: Firstly, the problem is solved on a relatively coarse mesh and local error estimates are computed. Next, partitions with approximately the same error are created, assuming that equal errors will lead to approximately equal future work. Then each process computes on the whole domain, but refinements are limited largely to its own partition. Finally, after parallel computations, a global solution is constructed on the union of the refined partitions. This can be done e.g. by a parallel multigrid solver or by a partition of unity method.

In contrast to the approach described here, no repartitioning is provided there, but only one partitioning based on local error estimates at the beginning. This makes it hard to obtain a good load balance, because local error estimates are only a rough estimate for the future work to be done in some region. Furthermore, repartitioning becomes useful in time dependent problems to consider instationary solution properties.

Mitchell introduced a concept called full domain partitions in [32]. It combines the adaptive finite element method with a parallel solver and a load balancing step in each iteration. Outside of the local partition the coarsest possible compatible mesh is used for each process. The global solution is built directly by a specialized parallel multigrid solver. So, no partition of unity is needed. A load balanced repartitioning is done in every iteration.

The concept of adaptive full domain covering meshes, presented in this chapter, combines the benefits of both approaches. No specialized solvers or other specialized modules are needed. The repartitioning strategy allows one to handle time dependent problems in a straightforward way. Load balance quality and repartitioning overhead can be balanced against each other. Furthermore, the concept of mesh structure codes (see Section 6.2.1) presents a very efficient way to exchange mesh information between the processes and further reduces the need for commu-
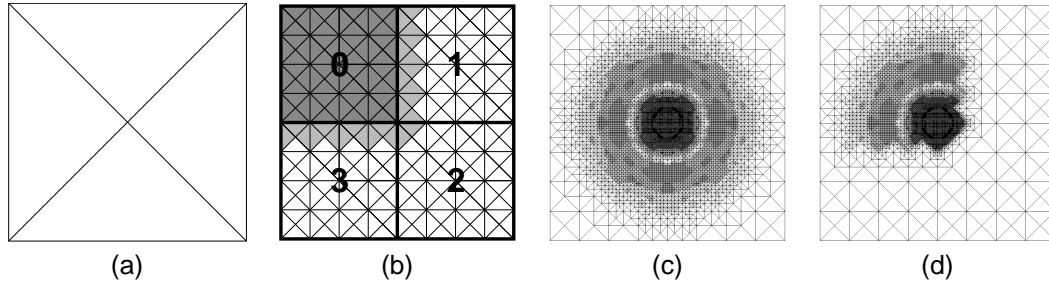
Figure 6.1: (a) A triangular macro mesh, (b) domain decomposition after six global refinements and overlap for partition 0, (c) composite mesh after adaptation loop, (d) local mesh of process 0 after adaptation loop

nication.

The remainder of this chapter is organized as follows: Section 6.1 gives a survey of the concept followed by implementation aspects presented in Section 6.2. In Section 6.3, a short source code example is given that should demonstrate the simplicity of parallelizing a sequential program.

6.1   Overview

The main idea of the parallelization approach presented in this work is the concept of adaptive full domain covering meshes. Starting with a very coarse macro mesh, a partitioning mesh is obtained by some global or adaptive refinement steps. The leaf elements of the resulting mesh build the partitioning level which is the basis for all future domain decompositions. After partitioning the mesh, each process computes on the whole domain, but refinements are allowed only on the local partition, including a certain overlap region, and some necessary propagation refinements. The overlap is needed to construct a global solution after the adaptation loop. The propagation refinements are needed to preserve mesh compatibility. Figure 6.1 illustrates the concept of adaptive full domain covering meshes. To simplify matters, in this example the partitioning did not change during the adaptation loop. In Section 6.2.4, it is shown how repartitionings within the adaptation loop are handled.

Due to adaptive refinements, the load may get out of balance during the adaptation loop. Then a new load balanced repartitioning will be necessary. The load balancing can be done before or after adapting the local meshes. If it is done before the adaptation step (predictive load balancing), the load situation after the adaptation has to be estimated considering the local error estimates at the elements. Here, regular load balancing has been applied in which the new partitioning is computed after mesh refinements. This leads to a little more communication, because a finer mesh has to be redistributed after repartitioning, but on the other hand the load balance is more accurate. After the adaptation loop, the global solution is constructed by a parallel partition of unity of all local solutions. Algorithm 16 describes the parallel adaptation loop on a high abstraction level. In the initialization step of line 1, the partitioning level is constructed and the first domain decomposition is done.

Initialization, repartitioning and building the global solution are described in more detail later in this paper. Without these three steps the algorithm would describe the usual non parallel adaptation loop for stationary problems in AMDiS.

In Figure 6.2, the domain notation used in this paper is shown. Ω stands for the whole problem domain and Ω_i for the local partition of process i. The local partition of process i including overlap is denoted by Ω_i^+. Ω_i^j is the sub domain of Ω_i that belongs to the overlap of Ω_j ($\Omega_i^j := \Omega_i \cap \Omega_j^+$).

Algorithm 16 parallel adaptation loop

1: initialize parallelization
2: assemble, solve, estimate
3: **while** tolerance not reached **do**
4:      adapt mesh
5:      **if** load out of balance **then**
6:          repartition mesh
7:      **end if**
8:      assemble, solve, estimate
9: **end while**
10: build global solution
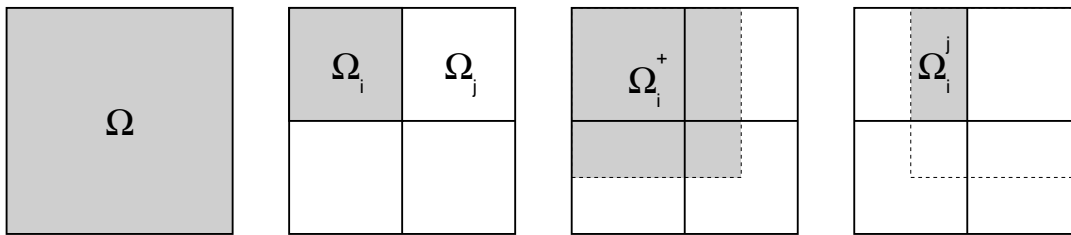
Figure 6.2: Domain notation used in this paper

6.2 Implementation

The parallelization was realized on top of the *Message Passing Interface* (MPI), which is the standard for message passing in distributed memory environments.

In Section 6.2.1, the concept of mesh structure codes is introduced which is an efficient way to exchange mesh information between the processes. Mesh structure codes can be used to construct global element and node indices, addressed in Section 6.2.2. Section 6.2.3 describes the *three level approach* which allows one to decouple partitioning level, overlap level and global coarse grid level. The repartitioning algorithm is the topic of Section 6.2.4. Section 6.2.5 addresses the construction of a global solution by a parallel partition of unity method. Finally, in Section 6.2.6 the parallelization of time dependent problems and systems of PDEs is discussed.

6.2.1 Mesh structure codes

At some points in the computation, e.g. when the global solution should be built or a repartitioning is performed, the current mesh states have to be communicated between the parallel processes. Therefore, a compact mesh representation is needed, which can be used for MPI communication. The concept of Mesh structure codes provides such a compact representation.

Like mentioned before, all processes start with the same coarse macro triangulation, which is refined in different ways during parallel computation. The only allowed refinement operation is bisection of elements. Every element has either two children or no children. The same holds for the nodes of the corresponding binary tree. For each node, we store a 1 if the corresponding element is refined and a 0 otherwise. To obtain a unique node order, we perform a pre-order traversal on the tree: First, the root of the tree is visited, then the left sub tree and, finally, the right sub tree is traversed in pre-order recursively. The resulting binary sequence can be interpreted as an (unsigned long) integer value which can be sent over MPI very efficiently. If the number of mesh elements exceeds the capacity of one integer variable, an array of integers has to be used. Using this code, the receiver can easily reconstruct the senders mesh or fit the local mesh to it. Figure 6.3 illustrates how to construct the mesh structure code of an adaptively refined triangle.
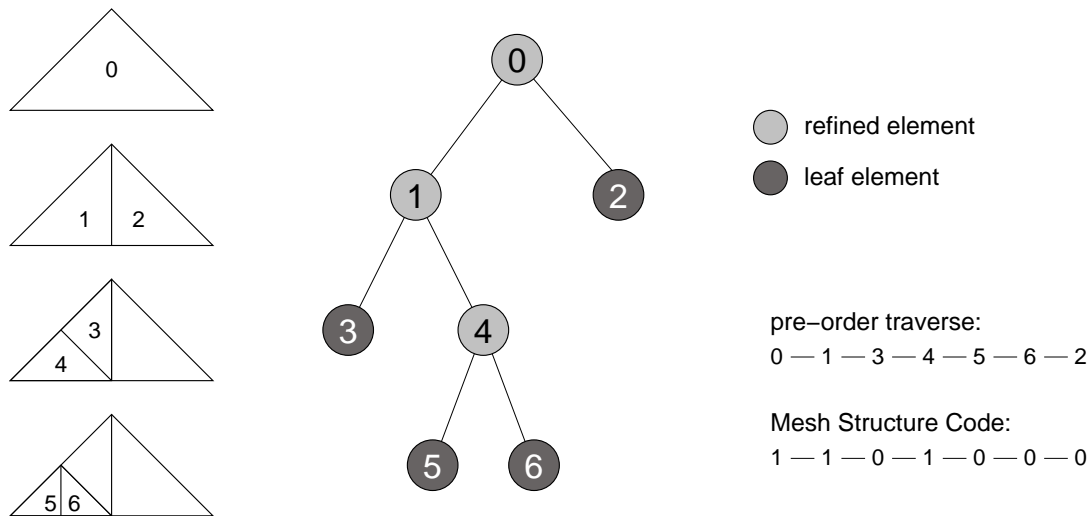
Figure 6.3: Mesh structure code of an adaptively refined triangle

The corresponding binary code 1101000 is represented by the integer value 104 together with the position of the first relevant bit within the binary representation. In this way, the code 1101000 can be distinguished from the code $0\ldots01101000$. The mesh structure code of a mesh is the concatenation of the mesh structure codes of its macro elements. The mesh structure code of the mesh showed in Figure 6.3 (b) is $110110000-0-0-101101000$ (the '-' character is used for a higher readability to separate the macro elements). Since the construction of mesh structure codes is not expensive, no update procedure for them is needed. A mesh structure code is constructed for the current mesh based on the hierarchical mesh data structure described in Section 3.2. It will be deleted after its usage.

Another feature of mesh structure codes is the possibility to build the composition of multiple meshes on a binary level. Therefore, in Algorithm 17, we firstly define a recursive algorithm to extract a sub code $subTreeCode$ which represents the sub tree starting from the node corresponding to position pos in the mesh structure code $structureCode$. If $subTreeCode$ is not empty at the beginning, the result is pushed to the back of $subTreeCode$. The return value of the algorithm is the first position within $structureCode$ after the extracted sub tree. The first recursive call of **get-**

Algorithm 17 getSubTreeCode($structureCode$, pos, $subTreeCode$)

 1: **if** $structureCode[pos] == 0$ **then**
 2:     $pos = pos + 1$
 3:     **if** $subTreeCode \neq NULL$ **then**
 4:         push 0 to back of $subTreeCode$
 5:     **end if**
 6: **else**
 7:     $pos = pos + 1$
 8:     **if** $subTreeCode \neq NULL$ **then**
 9:         push 1 to back of $subTreeCode$
10:     **end if**
11:     pos = **getSubTreeCode**($structureCode$, pos, $subTreeCode$)
12:     pos = **getSubTreeCode**($structureCode$, pos, $subTreeCode$)
13: **end if**
14: **return** pos

SubTreeCode delivers the first child's (or left) sub tree code, the second call the second child's

(or right) sub tree code. If $subTreeCode$ is not given ($subTreeCode == NULL$), the algorithm just skips the corresponding sub tree and returns the first position after it. This feature will be used e.g. in Algorithm 19 to skip unused sub trees in the mesh structure code. In Algorithm 18, **get-SubTreeCode** is now used to merge two codes into a composite mesh structure code. The result of this procedure is a code which represents the composition of the two meshes represented by $structureCode1$ and $structureCode2$.

Algorithm 18 merge($structureCode1$, $structureCode2$)

```
 1: result = empty structure code
 2: size1 = binary length of structureCode1
 3: size2 = binary length of structureCode2
 4: pos1 = 0, pos2 = 0
 5: while (pos1 < size1) and (pos2 < size2) do
 6:   if structureCode1[pos1] == structureCode2[pos2] then
 7:     push structureCode1[pos1] to back of result
 8:     pos1 = pos1 + 1
 9:     pos2 = pos2 + 1
10:   else
11:     if structureCode1[pos1] == 0 then
12:       pos1 = pos1 + 1
13:       pos2 = getSubTreeCode(structureCode2, pos2, result)
14:     else
15:       pos1 = getSubTreeCode(structureCode1, pos1, result)
16:       pos2 = pos2 + 1
17:     end if
18:   end if
19: end while
20: return result
```

To synchronize the local meshes during the parallel computation, each process computes its local mesh structure code. After that, the codes are exchanged between the processes via MPI. Now each process knows the mesh structure code of each other process and can build the composite code by merging all local codes. Then the local mesh can be locally adapted to fit to the composite code in some region. Usually, the local mesh is adapted only within the local partition including the corresponding overlap region. To skip the parts of the code that are not needed for local mesh refinements, Algorithm 17 can be used again. Now, the result in $subTreeCode$ can be ignored and only the new position returned by the algorithm is of interest.

6.2.2 Global Indexing

Different refinement and coarsening orders on the processes can lead to different element and node numerations on the processes, even if the final mesh structure is the same. To create global numerations, which are necessary for inter process communication, the concept of mesh structure codes described in Section 6.2.1 can be used.

Global element indices First the composite mesh structure code is built on each process by exchanging the local codes and merging them. In the little example illustrated in Figure 6.4, the local codes 10100 and 11000 are merged into the composite mesh structure code 1100100. Now a pre-order traversal is performed on the local mesh of each process. Simultaneously, the composite mesh structure code is traversed. If the current element in the local mesh is a leaf element but the structure code entry is 1 (for *refined element*), the corresponding sub tree of the code is skipped using Algorithm 17. The global index of the current local element is always its

mesh 1   **mesh 2**   **composite mesh**

(a)   (b)   (c)
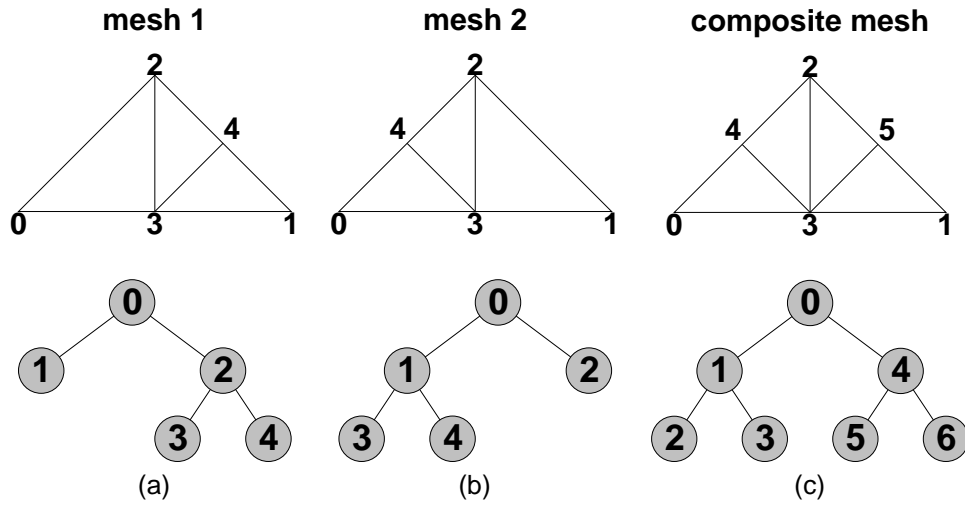
Figure 6.4: Node indices and binary trees with element indices for two differently refined meshes (a,b) and corresponding global node and element indices for the composite mesh (c).

composite code	1	1	0	0	1	0	0
global element index	0	1	2	3	4	5	6
mesh 1 element index	0	1	-	-	2	3	4
mesh 2 element index	0	1	3	4	2	-	-

Table 6.1: Global element indices for composite mesh structure code 1100100 and corresponding element indices for mesh 1 and mesh 2.

position in the composite mesh structure code. In Algorithm 19, this procedure is shown. The

---

Algorithm 19 create global element numeration

1: Create composite mesh structure code: $code$
2: $pos = 0$
3: **for all** elements el of local mesh (pre-order) **do**
4:    $globalElementIndex(localIndex_{el}) := pos$
5:    **if** el is leaf element **then**
6:       $pos :=$ **getSubTreeCode**($code, pos, NULL$)
7:    **else**
8:       $pos := pos + 1$
9:    **end if**
10: **end for**

---

mapping from local to global element indices in our example can be seen in Table 6.1.

Global node indices   To create a global node numeration, the composite mesh structure code is used to create a binary tree, corresponding to the structure of the composite mesh. At each node of this binary tree, the list of global node indices will be stored in local element order as a result of the algorithm. In the example, for global element 5 the list of global node indices $(3, 1, 5)$ is stored. Once such a binary tree is created, it is easy to obtain a mapping from local to global node indices. To create this binary tree, we can use the local node indices on macro elements as global indices, too, because the numeration at macro level is the same for each process. If the node indices of an element are known, the indices of its children can be constructed depending

global element index	0	1	2	3	4	5	6
global node indices	0,1,2	2,0,3	3,2,4	0,3,4	1,2,3	3,1,5	2,3,5

Table 6.2: Global node indices in local element order according to global element indices.

global node index	0	1	2	3	4	5
mesh 1 node index	0	1	2	3	-	4
mesh 2 node index	0	1	2	3	4	-

Table 6.3: Global and local node indices

on the element type. For linear Lagrange elements with one node at each vertex and parent node indices (p_0, p_1, p_2), the indices of the first child are $(p_2, p_0, newIndex)$ and those of the second child are $(p_1, p_2, newIndex)$. The counter $newIndex$ is set to the first number which is not used for the macro mesh nodes. It is incremented by one for each refined element. Now the whole binary tree for every macro element can be constructed and filled with global node indices in a recursive way. Table 6.2 lists the global node indices for the composite mesh elements for the example. This leads to the mappings illustrated in Table 6.3

6.2.3   Three level approach

As mentioned in Section 6.1, the domain decomposition is done on a fixed partitioning mesh. This partitioning mesh should be defined on a relatively coarse level for two reasons. First, the partitioning process is faster for fewer elements. Second, in time dependent problems it should be possible to coarsen the mesh in regions where such a fine mesh is not longer needed.

On the other hand, a large overlap would lead to a bad parallel speedup behavior, because on overlap regions more than one process computes on a fine grid. And defining the overlap on the coarse partitioning level would lead to large overlaps.

A third point is that the quality of the local solution of process i on Ω_i partially depends on the mesh level outside of Ω_i, see e.g. [41]. So, these three levels should be determined separately. First, the *partitioning level* is created and the domain decomposition is computed. Then the mesh is refined uniformly on the whole domain Ω to create the *global coarse grid*, which now is set to the coarsest mesh for future computations on Ω. In a third step the *local coarse grid* is constructed by uniform refinements within Ω_i, which builds the coarsest mesh for future computations within the partition. To ensure a smaller overlap due to local coarse grid construction, not only refinements within Ω_i are performed, but in every refinement step all elements with element distance 1 to Ω_i are refined, too. Then on the resulting mesh the overlap computation is performed.

In Figure 6.5, an example for this three level approach is shown. Notice the size of the final overlap compared to the size the overlap would have on partitioning level.

In Algorithm 20, we can now take a closer look at the initialization procedure needed for the parallel adaptation loop, introduced in Section 6.1.

After the creation of the partitioning level, an initial randomized partitioning is done. This first partitioning is needed to compute the first useful domain decomposition in parallel. The element weights set in line 3 are used for the partitioning. Every element of the partitioning mesh is weighted by 1 at this point because we want a partitioning with the same number of elements in each partition. At later stages of the computation, when further refinements of the partitioning mesh exist, the element weights are set to the number of leaf elements belonging to one partitioning element. In Section 6.2.4, setting the element weights for repartitioning is addressed in more detail.

In the last two steps, a parallel marker and a parallel estimator are created for each process. The marker is responsible for marking elements for coarsening and refinement depending on
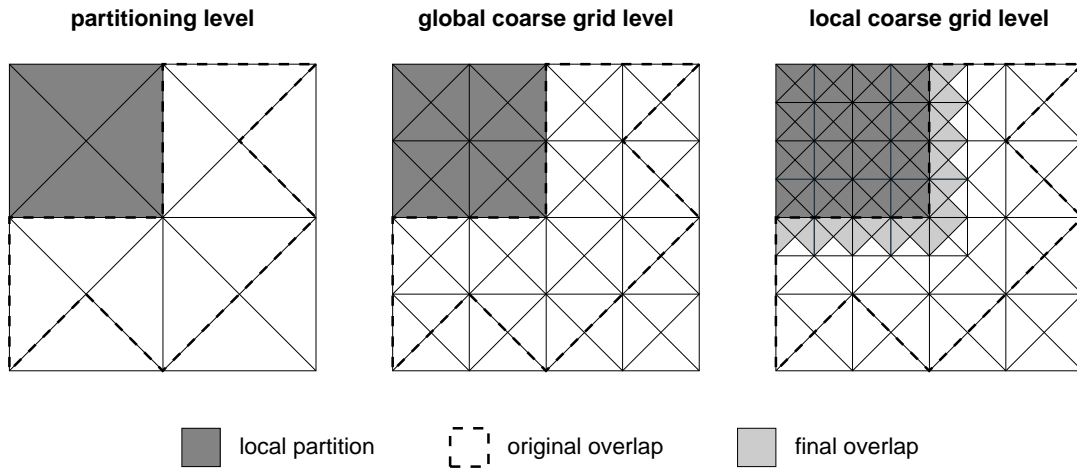
Figure 6.5: Example for the three levels of partitioning. The partitioning mesh is globally refined twice to get the global coarse grid level. Then another two global refinement steps applied on Ω_i and its direct neighbor elements (in each step) result in the local coarse grid level.

Algorithm 20 initialize parallelization (on process i)

1: create partitioning level
2: create initial partitioning
3: set element weights on initial Ω_i
4: create new partitioning (in parallel)
5: mark elements of new Ω_i
6: create global coarse grid level
7: create local coarse grid level
8: create overlap
9: create parallel estimator
10: create parallel marker

---

local error estimates. The parallel marker is responsible for marking elements for coarsening and refinement depending on local error estimates - with the requirements that only elements within Ω_i^+ can be adapted, and coarsening is limited by the local coarse grid. Propagation refinements are done by the refinement module automatically. So, no markings have to be done for them. The parallel estimator extends an arbitrary sequential estimator by communicating needed global values like estimation sums or maxima after each estimation step.

The parallel marker and estimator are created only once at the beginning of the parallelization. They use information about the current partitioning which is stored at the elements. This information is set in line 2 of Algorithm 20 and adapted within the adaptation loop after each repartitioning (line 6 of Algorithm 16).

6.2.4 Domain decomposition and repartitioning

For the domain decomposition the parallel graph partitioning library ParMETIS, described in [43], is used. First, a dual graph of the mesh that should be partitioned has to be constructed. Then the nodes of the dual graph, which corresponds to the elements of the mesh, are decomposed considering weight constraints on the graph nodes. The algorithm also tries to minimize the number (or edge weight sum) of graph edges that are cut by domain boundaries. Furthermore, a diffusive repartitioning of adaptively refined meshes is supported.

The goal of the domain decomposition is to decompose the domain Ω into n partitions (n is the number of processes in the parallel computation), such that the work load is approximately the same on every process. Each of the partitions should be connected and the boundaries between the partitions should be minimized to reduce the communication overhead and the size of the resulting overlap. In AMDiS, node weights are used to enable partitioning of a fine mesh on a coarser level (see Section 6.2.3). So far, no edge weights are used for partitioning. The partitioning algorithm in ParMETIS works in parallel. This means that an initial arbitrary partitioning must exist before the first call of ParMETIS. Each process then generates a new partition number for every element of its old partition. Redistribution of the new partitioning information is not part of ParMETIS and has to be done by the calling application. In the context of this work, every process has to collect its new partition elements from all other processes and mark them, including all descendants, as elements of the local partition.

When an element is refined, the partition status (IN, OUT or $OVERLAP$), is handed down to its children. Coarsening is only allowed if the resulting coarse element has a defined partition status. So, the partitioning mesh is the coarsest possible mesh for all future computations.

During the parallel adaptation each process adapts its local mesh due to local error indicators. In general, this leads to a more and more unbalanced load between the processes. Repartitioning then is applied to recover the optimal load balance. Like mentioned in Section 6.1, the basis of repartitioning is a fixed repartitioning mesh.

An optimal load balance is assumed if all partitions have the same number of leaf elements within their local partitions. One could imagine other criteria which could count the number of degrees of freedom or the number of all tree elements of the hierarchical mesh (not only leaf elements). Furthermore, one could consider the work to be done outside of the local partition on each process. To count the number of leaf elements within Ω_i probably is not the most accurate approximation, but it is very easy to implement and fast to execute.

In Figure 6.6, the element weights used for domain decomposition are shown for an adaptively refined macro triangle. One pre-order mesh traversal is needed to obtain these element weights. If a partitioning element is reached during the traversal, this element is stored as the current partitioning element. If a leaf element is reached, the weight of the current partitioning element (initially set to zero) is incremented by one. Note that in pre-order traversal the partitioning elements are visited before all of the corresponding leaf elements. The mesh of process i is not necessarily the finest on Ω_i because other processes have an overlap with Ω_i, in which they can refine also. Therefore, before the element weights are set, local mesh structure codes are exchanged and merged, and the local mesh on process i is adapted to the composite mesh within Ω_i.
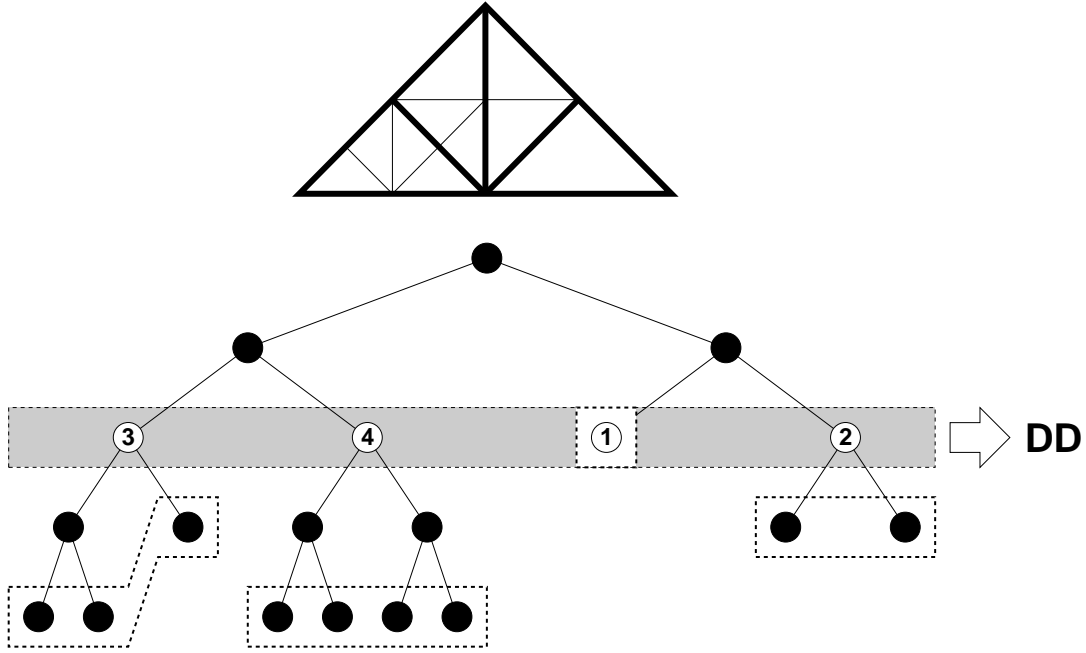
Figure 6.6: Element weights for an adaptively refined macro triangle. The partitioning mesh is built by the elements of level two.

The goal of repartitioning is to optimize the load balance and to reduce total computation time. But the repartitioning itself is an expensive procedure. To repartition after every mesh adaptation is not necessarily the best choice. Repartitioning is useful only if the time loss due to load imbalance is higher than the time needed for the total repartitioning process. To consider this aspect, we introduce a mechanism which after each mesh adaptation step decides whether to repartition or not. First, on every process the sum over all element weights is computed, and then the sum average over all processes is built. After that every process compares its local sum to this average. If the difference is too large for at least one of the processes, a repartitioning is scheduled. More precisely, a repartitioning is done if any of the local weight sums sum_i satisfies one of the following inequalities:

$$sum_i > rt_{high} \cdot sum_{average} \quad rt_{high} > 1 \tag{6.1}$$

$$sum_i < rt_{low} \cdot sum_{average} \quad 0 < rt_{low} < 1, \tag{6.2}$$

where rt_{high} and rt_{low} stand for upper and lower repartitioning thresholds and $sum_{average}$ is the average over all local weight sums.

6.2.5   Building the global solution

After the parallel adaptation loop, each process i has computed a solution on a fine mesh within Ω_i^+ and on a relatively coarse mesh outside of this region. We build one global solution u_{PU} out of the N rank solutions u_i by a partition of unity method (see [1]):

$$u_{PU}(x) := \sum_{i=0}^{N-1} \gamma_i(x) u_i(x) \quad \forall x \in \Omega \tag{6.3}$$

Algorithm 21 repartition on process i

1: adapt to composite mesh on Ω_i
2: set element weights on Ω_i
3: **if** repartitioning useful **then**
4:      compute new partitioning (in parallel)
5:      mark elements of new Ω_i
6:      create local coarse grid level
7:      create overlap
8:      adapt to composite mesh on new Ω_i^+
9:      exchange values
10:     coarsen outside of Ω_i^+
11: **end if**

Figure 6.7: Global view of W_0

where

$$\gamma_i(x) := \frac{W_i(x)}{\sum_{j=0}^{N-1} W_j(x)}. \tag{6.4}$$

Equation 6.4 ensures $\sum_{i=0}^{N-1} \gamma_i(x) = 1$ for all $x \in \Omega$, and $\gamma_i(x) \geq 0$ if $W_i(x) \geq 0$ for all $i \in [0:N-1]$ and for all $x \in \Omega$. We define

$$W_i(x) := \sum_{\phi \in \Phi_i^c} \phi(x), \tag{6.5}$$

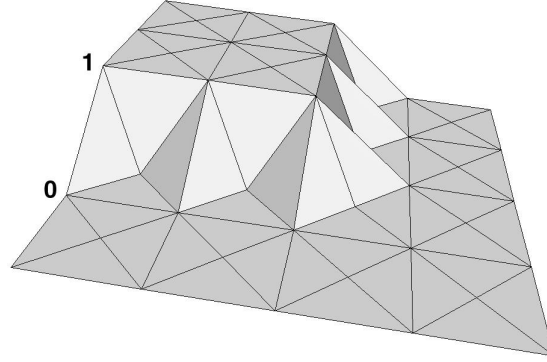where Φ_i^c is the set of all linear basis functions of the local coarse grid level of partition i located at vertices with an overlap distance to Ω_i smaller than the given overlap size. This choice leads to functions W_i which are constant 1 within Ω_i, constant 0 outside of Ω_i^+, and have a linear slope in the overlap region. Figure 6.7 shows such a function in the two dimensional case for an overlap size of 1. After construction of the γ_i functions on the local coarse grids, equation 6.3 can be evaluated at each discretization point of the fine composite mesh to obtain the final solution.

For a parallel computation of u_{PU}, each process i computes the partition of unity within its local partition Ω_i. For this purpose the process needs the local solution u_j of process j in $\Omega_i^j = \Omega_i \cap \Omega_j^+$ for all $j \neq i$. In Figure 6.8 the communication scheme for one process is illustrated.

In [28] an upper bound for the error in H^1 semi norm resulting from the partition of unity is given. Assume $u \in H^2(\Omega)$, then $\|u - u_{PU}\|_{H^1} \leq C(h + H^2)$, where h is the maximal edge size of mesh i in Ω_i and H is the maximal edge size of mesh i in $\Omega \setminus \Omega_i$. In particular, if $h \leq \sqrt{H}$:

$$\|u - u_{PU}\|_{H^1} \leq C(h). \tag{6.6}$$

As in [5], we do not require this constraint in the simulations in Section 8.3. However, the results still fulfill our tolerance requirements on the H^1-error with the analytic solution, which was set to be 10^{-3}.

send values to other partition

receive values from other partition
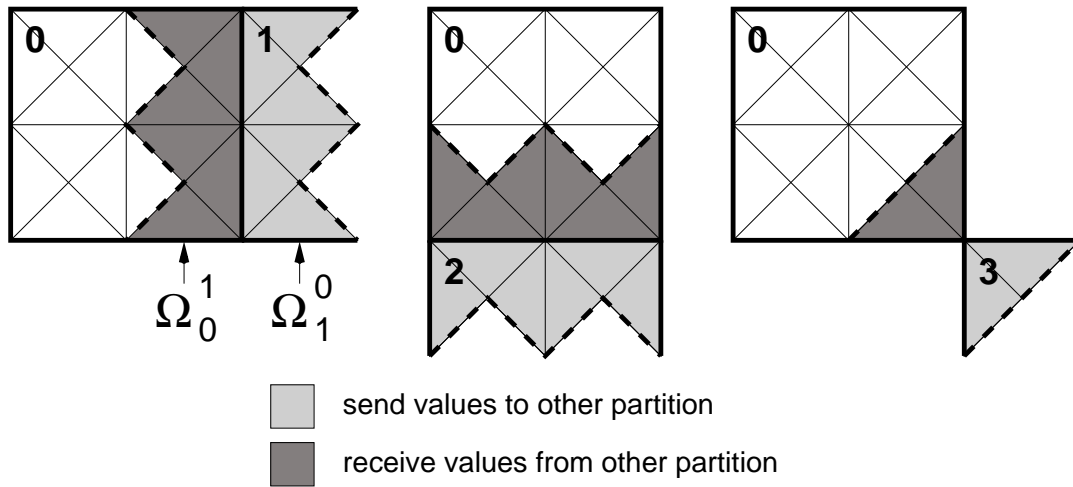
Figure 6.8: Necessary communication for process 0

6.2.6 Time dependent and vector valued problems

The parallelization of time dependent problems is straight forward. Because repartitioning is provided already in the stationary adaptation loop, the partitioning will also adapt to mesh changes over time automatically (see Section 6.2.4). The basis for repartitioning is always the relatively coarse partitioning level, defined at the beginning of computation. In time dependent problems not only refinement but also coarsening of elements can occur. But the coarsest possible mesh is defined by the local coarse gird level and the global coarse grid level respectively introduced in Section 6.2.3. Further coarsening decisions are ignored. After each timestep, the global solution is constructed by a partition of unity. And after each repartitioning, the solution of the last timestep is sent from the former owner process of a fine element to the new owner process of this element, like it is done for all relevant values located at mesh nodes or elements.

If a system of PDEs should be solved in one vector valued problem, all PDEs are discretized on one common mesh and result in one linear system of equations. So, all aspects concerning the mesh, e.g. the partitioning, are handled in exactly the same way as in the scalar case. Aspects concerning the values defined on the mesh, like value exchange and partition of unity, must be treated separately for each component, but also in the same way as in the scalar case.

6.3 Code example

This section gives an impression of how easy it is for the user to parallelize a given sequential code. The most work is done by a parallel problem class, which extends the original problem, and adds the needed parallelization abilities to it. Instead of the original problem, this parallel problem is handed over to the adaptation loop. Before the loop is started, an *initParallelization* routine has to be called. And after the loop has finished, an *exitParallelization* routine has to be called. In the following example the relevant code lines of a stationary scalar problem called *parallelellipt* are shown:

```
#include "mpi.h"                                              // added
#include "ParallelProblem.h"                                  // added
...
int main(int argc, char* argv[])
{
  MPI::Init(argc, argv);                                      // added
  ...
```

```
  ParallelProblemScal parallelellipt("ellipt->parallel",    // added
                                 &ellipt);

  AdaptStationary *adaptationLoop =
    NEW AdaptStationary("ellipt->adapt",
                        &parallelellipt,                      // modified
                        adaptInfo);

  parallelellipt.initParallelization(adaptInfo);             // added
  adaptationLoop->adapt();
  parallelellipt.exitParallelization(adaptInfo);             // added
  ...
  MPI::Finalize();                                           // added
}
```

In this example, *ellipt* is the name of the original sequential problem. After compiling and linking this code for MPI use, it can be started in parallel by *mpirun*. Needed parallelization parameters, like partitioning level or partitioning thresholds, can be set in a parameter file. Otherwise, they are set to predefined default values.

Chapter 7

SMI - Shared Mesh Interface

In AMDiS, no visualization of the simulation results is included. By default, the results are written to output files of specified formats. These files then can be read by a visualization tool or any other post-processing software. To avoid this detour over output files written to the hard disc, the *Shared Mesh Interface* (SMI) has been developed. SMI is not restricted to the use with AMDiS, but it can be used in every situation where two applications need a shared distributed mesh management. The connection of simulation software with a visualization tool is only one application example.

The goal of SMI is to provide an unified and distributed management for arbitrary meshes. So on the one hand, SMI provides an abstract interface which can handle any kind of unstructured meshes consisting of arbitrary and even mixed element types. On the other hand, this interface is already implemented efficiently in two ways:

1. In the **standard mode** the *shared mesh manager*, which implements the interface, is linked directly to the user program as a shared library. Note, that in this mode the mesh data can not be shared with other programs but only used in the program the manager is linked to. Different programs linking against the *shared mesh manager* will create different manager instances which have no knowledge of each other.

2. Therefore, in the **client-server mode** the program does not link against the *shared mesh manager* but against a *SMI client*. This client will connect to the *SMI server* which is a self contained program containing the *shared mesh manager*. Now different user programs running on even different computers can connect to this server and so share and exchange their mesh data.

This two communication modes are illustrated in Figure 7.1.
In Figure 7.2, one can see a *shared mesh manager* containing two meshes with different quantities and three applications.
In SMI, the following concepts are realized:

- **Nodes**: Every node consists of its coordinates and a (within its mesh) unique node ID.

- **Elements**: Besides the element ID, each element contains a list of node IDs. The number of nodes for one element depends on its element type.

- **Meshes**: Elements and nodes together form a mesh.

- **Quantities**: Quantities can be located at nodes or at elements of a mesh, or they can be mesh-global. A quantity located at nodes/elements, has values of a given type and dimension at every node/element of the corresponding mesh.

- **Applications**: SMI must know the involved applications to allow synchronized data access. Furthermore, applications can specify whether data changes from the applications point of view should be logged.
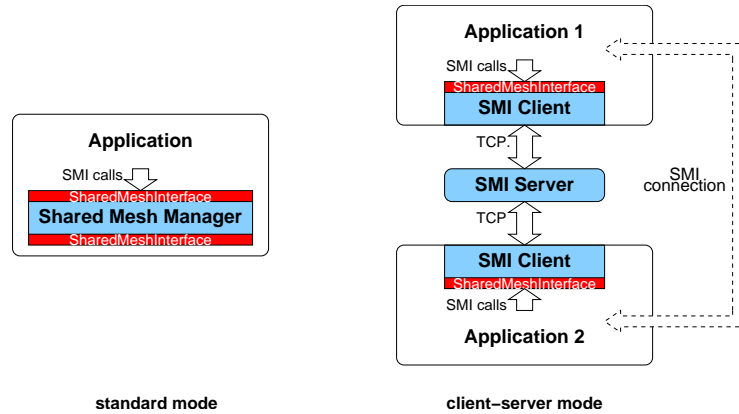
Figure 7.1: The two communication modes in SMI
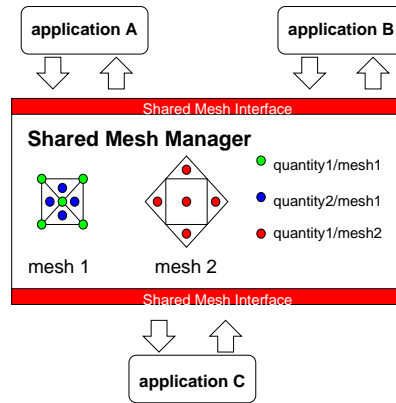
Figure 7.2: A shared mesh manager containing different meshes and quantities

- **Transactions**: Read and write transactions avoid problems with concurrent data accesses and manipulations.

- **Synchronization points**: Synchronization points allow the synchronization of different parallel running applications.

- **Relations**: Can be used to store additional informations how elements and nodes of a mesh are related.

- **Iterators**: Define a order in which elements or nodes should be traversed.

7.1   Nodes, elements, meshes

The nodes of a mesh are stored in a mapping which maps the unique ID of each node to its coordinates. The coordinate dimension is fixed within one mesh and it is defined when the first node is added to the mesh.

$$node : nodeIndices \mapsto \mathbb{R}^d \tag{7.1}$$

Before an element can be defined, its element type must be defined. The element type stores the number of nodes belonging to an element of this type:

$$nodesForElementType : elemTypeIDs \mapsto \mathbb{N} \tag{7.2}$$

No further semantics about an element type is stored in SMI. How the nodes and their order is interpreted is up to the applications. For example, a rectangular element in 2d may defined by its 4 vertex nodes and a triangular element by its 3 vertex nodes plus one node at the inner face (see Figure 7.3).
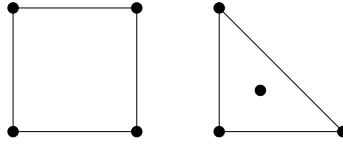
Figure 7.3: Different element types with 4 nodes

Both element types contain 4 nodes. The only difference is the element type ID for this types, which must be interpreted by the applications. The advantage of this approach is, that SMI can handle any element type, because only the needed number of nodes must be given. The main drawback is, that the applications must communicate the meaning beyond the element type IDs outside of SMI. An element now can be given by its type and a list of node indices:

$$elementType : elementIndices \mapsto elemTypeIDs \tag{7.3}$$

and

$$elementNodes : elementIndices \mapsto nodeIndices^{numNodes} \tag{7.4}$$

where $numNodes$ is the number of nodes needed for the element type the element belongs to:

$$numNodes(elementIndex) = nodesForElementType(elementType(elementIndex)). \tag{7.5}$$

So a mesh is stored as a list of elements and each element is stored as a list of nodes. Now, if one is interested in all elements containing a specific node the whole mesh must be traversed and each element must be checked for this node. To avoid this inefficient procedure, additionally a node-element mapping can be built. Here, for each node index a list of all element indices containing this node is stored:

$$nodeElements : nodeIndices \mapsto elementIndices^{numElements} \tag{7.6}$$

$numElements$ is the number of elements containing this node. This number can be different for each node index and change dynamically when elements are added or removed.

7.2 Quantities

In addition to the elements and nodes of a mesh, values associated to the mesh, like simulation or measurement results, may be of interest. Therefore, in SMI so-called quantities, which are sets of values with common properties, can be added to a mesh. Each quantity has the following properties:

1. **Type**: In SMI, different scalar data types like `float`, `double`, `int` or `char` are supported.

2. **Dimension**: Even vector valued quantities can be managed in SMI. Values of such a quantity then are vectors of the given dimension.

3. **Location**: Quantities can be located at nodes, elements or meshes. A quantity located at nodes/elements has one value for each node/element of the mesh. A quantity with global location has only one value for the whole mesh.

4. **Default value**: The default value is assumed for all values of a quantity that are not explicitly set by the user.

In SMI, *lazy memory allocation* is used for the values. This means, that memory is not allocated for a value, since it is set or retrieved the first time. If a value is retrieved, which was not set or retrieved before, first the needed memory will be allocated and then the default value for the corresponding quantity is copied into this memory. Finally, this default value will be returned to the calling application. This lazy procedure avoids allocation of memory for values that are never used, and it avoids the necessity of checking all defined quantities every time a node or element is added or removed.

7.3   Applications

The application concept is needed in SMI, because the managed data can be shared between different applications. To avoid inconsistent data states due to concurrent application accesses, read and write transactions are used (see Section 7.4). The concept of synchronization points, described in Section 7.5, enables the synchronization of parallel running applications. For example, a visualization tool which should show the results of a simulation has to wait until the simulation program has finished its computation.

Another application feature in SMI is the possibility to store changes from the applications point of view. Consider that the visualization tool should show the simulation results after each timestep and that the underlying mesh has changed only in a small region. Then it would be useful to update only changed elements and nodes instead of sending the whole mesh in each step. Such update informations are stored only for application which run in the so called *change-log-mode* (see Section B.2.2).

7.4   Transactions

The transaction concept ensures that concurrent application accesses can not lead to inconsistent data states within one mesh, and that data are not changed by one application while an other application tries to read or write data of that mesh. In SMI, there are two kinds of transactions. Data can be read only within a read transaction and they can be written only within a write transaction. A read transaction can be opened only if there is no open write transaction. A write transaction can be opened only if there is no open read **or** write transaction. Thus, concurrent read accesses are possible, but no concurrent read-write or write accesses.

If an transaction can not be opened immediately, either the opening function returns with an error value (non-blocking mode) or it waits until the transaction can be opened and then returns with `SMI_OK` (blocking mode).

In Figure 7.4, an example of blocking transactions is shown. First, application 1 opens a write transaction, then application 2 tries to open a read transaction. Because the write transaction is not yet finished, application 2 is blocked, until application 1 has closed its write transaction. Now, within the read transaction of application 2, application 1 tries to open another read transaction. This is done immediately, because concurrent read transactions are allowed. Finally, after closing the read transaction, application 2 tries to open a write transaction, but application 1 has not finished its read transaction yet. So, application 2 is blocked until the end of this read transaction.

7.5   Synchronization points

With synchronization points different applications which run in parallel can be synchronized at application level. A synchronization point is defined by the IDs of all application which should reach this point by a call of `SMI_Reach_sync_point()`. The only task of this function, is to block the calling application until all other corresponding applications has reached this point, too. When the last application has reached the synchronization point all applications can continue.
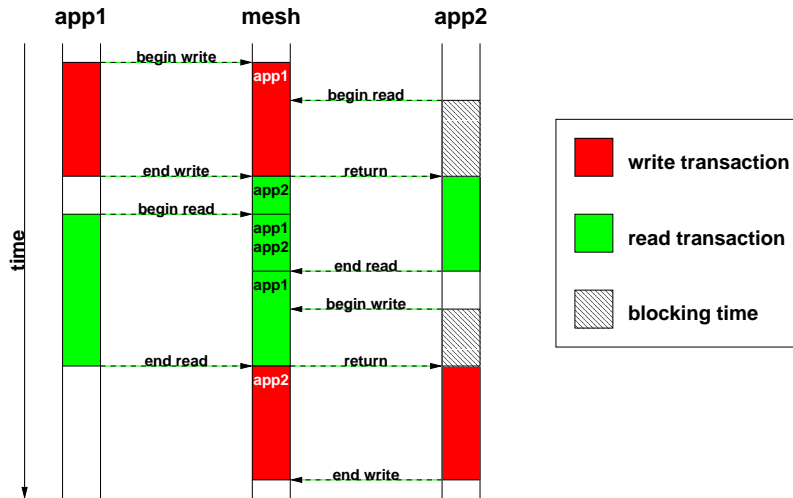
Figure 7.4: Two applications with blocking transactions

Imagine, that a visualization tool should show the simulation results of another application and that many time steps are computed which all have to be visualized. A result should be visualized as soon, as the simulation software has finished the corresponding computations and every time step should be visualized only once. Therefore, the visualization has to wait for the end of one computation step and the simulation program has to be sure, that the visualization is ready with the last step, before the new values are sent to SMI. The solution is, to define a synchronization point for the two applications. The simulation tool waits for this synchronization point after one simulation step and the visualization program before the results of one step are visualized. In Figure 7.5, this example is illustrated.
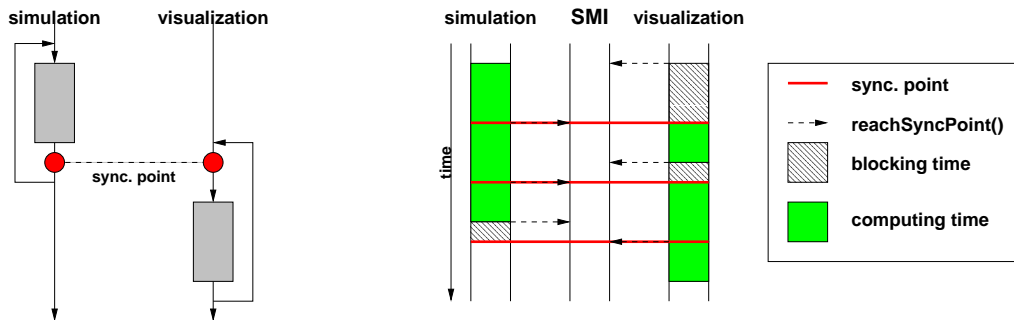
Figure 7.5: Synchronization points for coupled simulation and visualization

7.6 Relations

In SMI, relations can be used to store additional information which relate elements and nodes of one mesh. One example could be a parent-child relation which is used to store a hierarchical element structure in SMI. A relation is a set of tuples where each tuple includes a defined number n of element- and m of node-indices:

$$elementIndices^n \times nodeIndices^m. \tag{7.7}$$

After a relation is defined on a mesh, tuples can be added by `SMI_Add_relation_tuples()` and retrieved by `SMI_Query_relation_tuples()` (see Section B.2.9).

7.7   Iterators

Iterators define the order of traversing elements or nodes of a mesh. They are stored as lists of
integer values. The advantage, of storing iterators within SMI instead of storing them within the
application itself, is, that iterators defined by one application can be used by other applications,
as well. Figure 7.6 shows examples for element and node iterators.

element–iterator: 0–>1–>4–>5–>6–>3–>2   node–iterator: 3–>4–>6–>7–>4–>5–>2
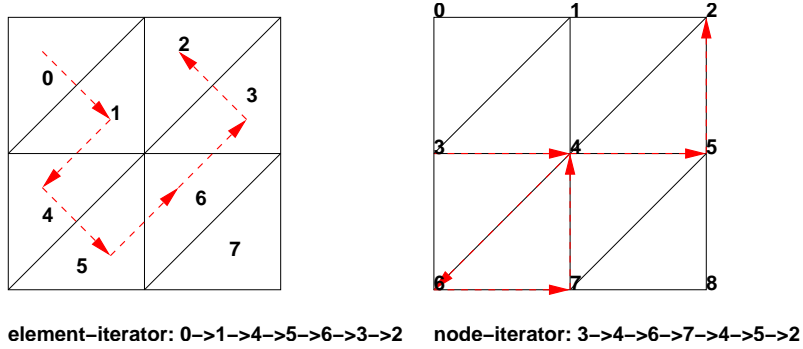
Figure 7.6: Example for an element iterator and a node iterator

In Appendix B, a complete SMI reference is given.

Chapter 8

Simulation examples

In this chapter, several simulation examples performed with AMDiS are presented. The chapter is divided into three parts. In the first part (Section 8.1), some general examples are given that demonstrate the possibilities of AMDiS. The second part in Section 8.2 discusses some numerical results concerning the multigrid concepts described in Chapter 5. Finally, in Section 8.3, several examples concerning the parallelization concepts introduced in Chapter 6 are given.

To get an impression of how easy a large class of problems can be implemented in AMDiS on a very high abstraction level, in the AMDiS tutorial in Appendix A, several examples concerning different simulation aspects are given. In this tutorial, also the corresponding application source codes, the used parameter files and macro triangulations are described.

AMDiS has already been used for the simulation results of several publications. Not all of these results can be described in detail in this work. Therefore, in the following a list of some of these publications is given:

- In [47] mean curvature flow and surface diffusion with non-convex anisotropies is simulated in two and three dimensions. Curvature regularization is applied to obtain rounded corners and edges.

- In [48] the numerical solution of a kinetic model is treated, which combines the effects of mean curvature flow and surface diffusion.

- In [49] a level set approach to anisotropic surface evolution with free adatoms is described.

- In [50] the so called geodesic evolution of curves under mean curvature flow and surface diffusion on arbitrary surfaces is investigated.

- In [2] the numerical solution of a phase field crystal (PFC) model has been computed. Here, a 6th order equation was solved as system of three 2nd order equations.

- In [39] a diffuse-interface approximation of surface diffusion was coupled with the diffusion of adatoms.

- In [38] the surface evolution of elastically stressed films under deposition has been computed. Here, two vector valued problems, namely the linear elasticity problem and the diffuse interface approximation of motion by surface diffusion, have been coupled in one code.

8.1 General AMDiS examples

The object of interest in the first examples is the Stanford Bunny. This bunny model is most commonly used in testing computer graphics techniques and it is complicated enough as a test
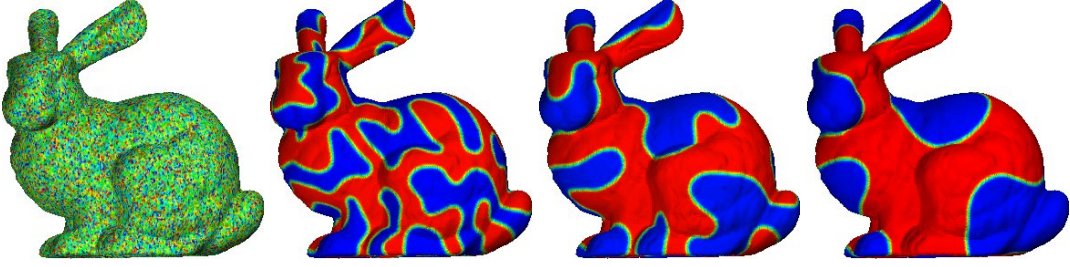
Figure 8.1: Coarsening in the Cahn-Hilliard equation. Initial condition $u = 0.5$ (slightly perturbed), blue denotes $u = 0$ and red denotes $u = 1$. The time steps are $t = 0, 0.001, 0.00463, 0.01163$. The simulations are performed by A. Rätz.

object for non-standard numerical simulations. With the techniques developed in [52], a polygonal mesh with 69.451 triangles of a bunny surface was created which after some modifications is here used as a macro triangulation for computations on parametrically defined domains. Besides this polygonal mesh, the bunny is also implicitly described by a signed distance function and used as a computational domain to demonstrate the applicability of the level set techniques.

8.1.1 Higher order models on polygonal meshes

Applications for higher order partial differential equations on surfaces are abundant. For some special problems related to materials science, biology and image processing we refer to [33, 25, 31]. Here, we will concentrate on a classical model for spinodal decomposition of a binary alloy, the Cahn-Hilliard equation. The model is applicable to describe coarsening dynamics in phase separation processes, which occur in quenched alloys. For numerical approaches, also see [10, 29, 19]. Here, such an equation is solved on a general surface S. The equation reads

$$u_t \;\;=\;\; \Delta_S \left(-\epsilon \Delta_S u + \epsilon^{-1} G'(u) \right) \tag{8.1}$$

with Δ_S the surface Laplacian, $G(u) = 18u^2(1 - u^2)$ a double well potential and ϵ a small parameter. Furthermore, $u = 0$ and $u = 1$ are the two stable steady states, representing the two phases. As initial conditions a small zero mean perturbation of $u = 0.5$ is used. The numerical approach is the same as for Euclidean geometries, see [37]. We write (8.1) as a system of two second order equations, discretize in space by linear finite elements, linearize the derivative of the double-well potential, apply a semi-implicit time-discretization and solve the resulting linear system by an iterative solver. The finite element representation reads

$$\int_S \frac{u_h^{n+1} - u_h^n}{\tau^n} \phi \;\;=\;\; \int_S \nabla_S w_h^{n+1} \cdot \nabla_S \phi \tag{8.2}$$

$$\int_S w_h^{n+1} \phi - \epsilon \int_S \nabla_S u_h^{n+1} \cdot \nabla_S \phi - \epsilon^{-1} \int_S G''(u_h^n) u_h^{n+1} \phi$$

$$=\;\; \epsilon^{-1} \int_S (G'(u_h^n) - G''(u_h^n) u_h^n \phi) \tag{8.3}$$

with ϕ test functions from the space of piecewise linear, globally continuous elements. To solve this problem in AMDiS, an implementation on an Euclidean grid can be used. No changes in the code are needed, only the parametric mesh has to be provided. Fig. 8.1 shows the evolution of u at different time steps. The solution quickly separates the surface S into two regions S_0 and S_1, where u takes the values of 0 and 1, respectively. The remaining part of S lies on an interface of width $\mathcal{O}(\epsilon)$ between the two regions. In later stages S_0 and S_1 change shape such that the length of the interface between the two regions decreases while maintaining the area of S_0 and S_1.
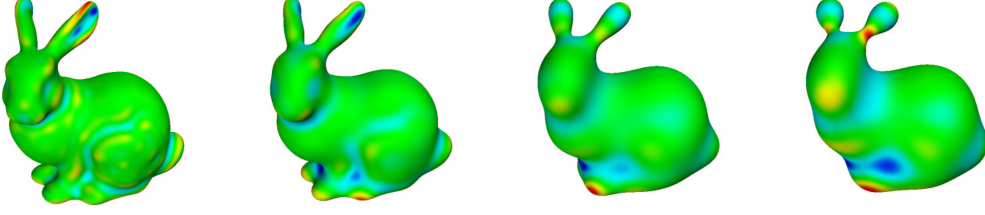
Figure 8.2: Surface evolution due to surface diffusion. The time steps are $t = 10^{-7}, 10^{-6}, 5 \cdot 10^{-6}, 10^{-5}$. red indicate a normal velocity inwards and blue outwards. The simulations are performed by F. Haußer.

8.1.2 Geometric evolution by parametric finite elements

The evolution of surfaces plays a major role in various applications, such as fluid dynamics, materials science and image processing [21, 42], and becomes even more important if smaller and smaller length scales are reached. Here, we will concentrate on surface diffusion, as an important mass transport mechanism in epitaxial growth.

$$v = -\Delta_S \kappa \tag{8.4}$$

with v the normal velocity and κ the curvature. For the numerical approaches see [8]. The fourth order equation is rewritten into a system of second order equations, discretized in space by linear parametric elements and semi-implicitly in time by treating the nonlinear operators and the surface normal explicitly but all other quantities implicitly. The resulting system for the vector and scalar valued unknowns $\vec{\kappa} = \vec{\kappa}_h^{n+1}, \kappa = \kappa_h^{n+1}, \vec{v} = \vec{v}_h^{n+1}$ and $v = v_h^{n+1}$ reads

$$\int_{S^n} \vec{\kappa}\vec{\phi} - \tau^n \int_{S^n} \nabla_{S^n}\vec{v} \cdot \nabla_{S^n}\vec{\phi} = \int_{S^n} \nabla_{S^n}\vec{x}^n \cdot \nabla_{S^n}\vec{\phi} \tag{8.5}$$

$$\int_{S^n} \kappa\vec{\phi} = \int_{S^n} \kappa \cdot \vec{n}^n \phi \tag{8.6}$$

$$\int_{S^n} v\phi = \int_{S^n} \nabla_{S^n}\kappa \cdot \nabla_{S^n}\phi \tag{8.7}$$

$$\int_{S^n} \vec{v}\vec{\phi} = \int_{S^n} v\vec{n}^n\vec{\phi} \tag{8.8}$$

with ϕ and $\vec{\phi}$ scalar and vector valued piecewise linear test functions. The first equation results from the geometric expression $\vec{\kappa}^{n+1} = -\Delta_S \vec{x}^{n+1}$, with $\vec{x}^{n+1} = \vec{x}^n + \tau^n \vec{v}^{n+1}$ the updated position vector. The resulting linear system is solved by a Schur-complement ansatz. The main difference to the implementation of (8.2) and (8.3) lies in the fact, that the parameterization of the surface changes in time, which is accounted for in treating the position vector $\vec{x}^n$ as an unknown. For further numerical details and an extension to anisotropic situations see [26]. Fig. 8.2 shows the evolution of the surface at different time steps. The normal velocity is plotted on the surface to visualize the smoothing properties in detail. The evolution quickly smoothes small surface features and afterwards evolves towards its equilibrium shape. The configuration of the bunny however will lead to a pinch-off off the left ear, which cannot be handled within the described method. For the demonstrated isotropic situation the surface area decreases, while keeping the volume constant.

8.1.3 Implicit description of surfaces

In several applications, today mainly related to volumetric medical imaging, surfaces are not given in parametric form. They are only defined through implicit functions. On the other hand due to
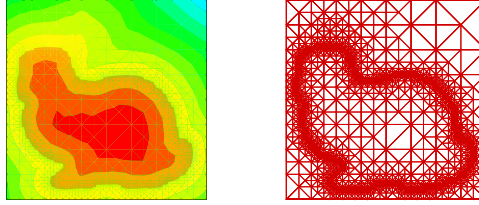
Figure 8.3: Signed distance function and adaptive mesh on cut through the bunny.

the success of level set methods [45, 34] in a wide range of applications, including computer vision, fluid dynamics, optimal design, and others where the simulation of moving interfaces plays a key role, an implicit description of domains and the solution of partial differential equations on such implicitly described domains is of importance. In constructing signed distance functions, as well as in solving problems by level set methods, efficient numerical methods of Hamilton-Jacobi equations are necessary. To derive such methods on unstructured meshes a finite-element discretization of [12] is used. Here, a linear finite element solution is constructed through a simplified local equation, which is solved by the Hopf-Lax formula. The proposed adaptive Gauss-Seidel iteration for the solution of the nonlinear system is modified and extended to three dimensional situations in [47] and used here to construct a signed distance function of the Stanford bunny from a given implicit representation. Fig. 8.3 shows on a cut through the bunny, the computed signed distance function and the adaptively refined mesh.

The computed data is used in the following sections as an initial description of the surface.

8.1.4 Geometric evolution by level sets

Again, we are concerned with surface evolution, but now concentrate on mean curvature flow, important for example to describe the motion of grain boundaries. For an isotropic situation the equation has the simple form

$$v = \kappa. \tag{8.9}$$

This equation is solved within a level set context. For related work see [46, 22, 16]. Starting from the L^2-gradient flow of the surface energy $e(S_0) = \int_{S_0} 1 ds$ and representing the surface $S_c = \{\vec{x} \in \Omega | u(\vec{x}) = c\}$ through the level set of u with value c, we can define a global energy $\mathcal{E}(u) = \int_{I\!R} e(S_c) dc = \int_{\Omega} \|\nabla u\| d\vec{x}$. Following [16] this can be used to derive a finite element formulation for isotropic mean curvature flow, which in a semi-implicit time discretization reads

$$\int_{\Omega} \frac{u_h^{n+1} - u_h^n}{\tau^n} \frac{1}{||\nabla u_h^n||} \phi \;\; = \;\; \int_{\Omega} \frac{\nabla u_h^{n+1}}{||\nabla u_h^n||} \cdot \nabla \phi \tag{8.10}$$

with ϕ test functions from the space of piecewise linear, globally continuous elements. For further numerical details and an extension to anisotropic situations see [16]. Fig. 8.4 shows the evolution of the surface at different time steps.

Again, the evolution quickly smoothes small surface features and afterwards evolves towards its equilibrium shape, while the volume is shrinking. As expected, the time scale is different than for the surface diffusion case in 8.1.2.

8.1.5 Anisotropic surface diffusion by parametric finite elements

As already done in Section 8.1.2, also in this example surface diffusion is simulated using linear parametric finite elements. But in this example an anisotropic surface energy is used. To consider such anisotropies is important e.g. in the simulation of epitaxial crystal growth.
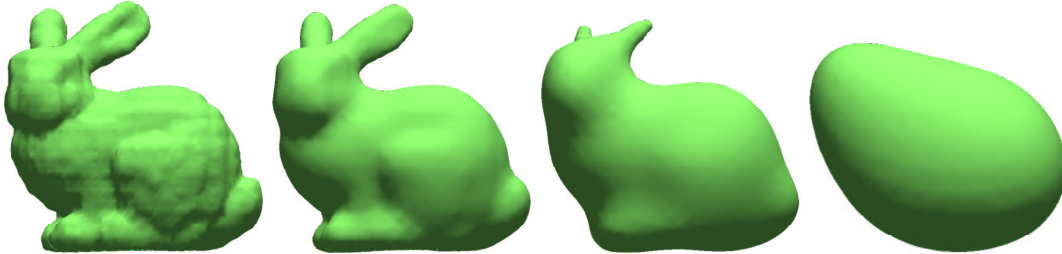
Figure 8.4: Surface evolution due to mean curvature flow. The time steps are $t = 0, 0.006, 0.030, 0.200$. The simulations are performed by C. Stöcker.

The surface evolves according to an anisotropic surface diffusion if the normal velocity v satisfies

$$v = \Delta_S \kappa_\gamma, \tag{8.11}$$

where Δ_S is the surface Laplacian and κ_γ its mean curvature according to the anisotropy γ.

The Wulff shape according to an anisotropy γ describes the state with the least anisotropic surface free energy for a given volume. It is defined by

$$\mathcal{W}_\gamma = \{z \in \mathbb{R}^3 | z \cdot q \le \gamma(z) \text{ for all } q \in \mathbb{R}^3\}. \tag{8.12}$$

In this example γ is the vector valued strong (convex) anisotropy

$$\gamma(z) = \sum_{k=1}^{3} (0.01|z|^2 + z_k^2)^{\frac{1}{2}} \quad \text{(regularized } l^1\text{-anisotropy)}. \tag{8.13}$$

The corresponding Wulff shape is shown in Figure 8.5.

Figure 8.5: Wulff shape for the regularized l^1-anisotropy.

As already described in Section 8.1.2, a second order splitting leads to a system of two second order equations. The anisotropy is incorporated by using an appropriate weak formulation for the vector valued anisotropic mean curvature $\kappa_\gamma \vec{n}$, see [15, 17].

In the first row of Figure 8.6, the surface evolution starting with a sphere is shown, simulated without using time and space adaptivity. If one compares the steady state at $t \approx 10^{-3}$ with the

Wulff shape in Figure 8.5, it reveals that the rounded corners of the Wulff shape are not resolved properly.

$$t = 0(1538) \qquad t = 0.0002(1538) \qquad t = 0.0005(1538) \qquad t = 0.001(1538)$$

$$t = 0(386) \qquad t = 0.0002(338) \qquad t = 0.0005(314) \qquad t = 0.001(494)$$

$$t = 0(386) \qquad t = 0.0002(855) \qquad t = 0.0005(894) \qquad t = 0.001(1085)$$
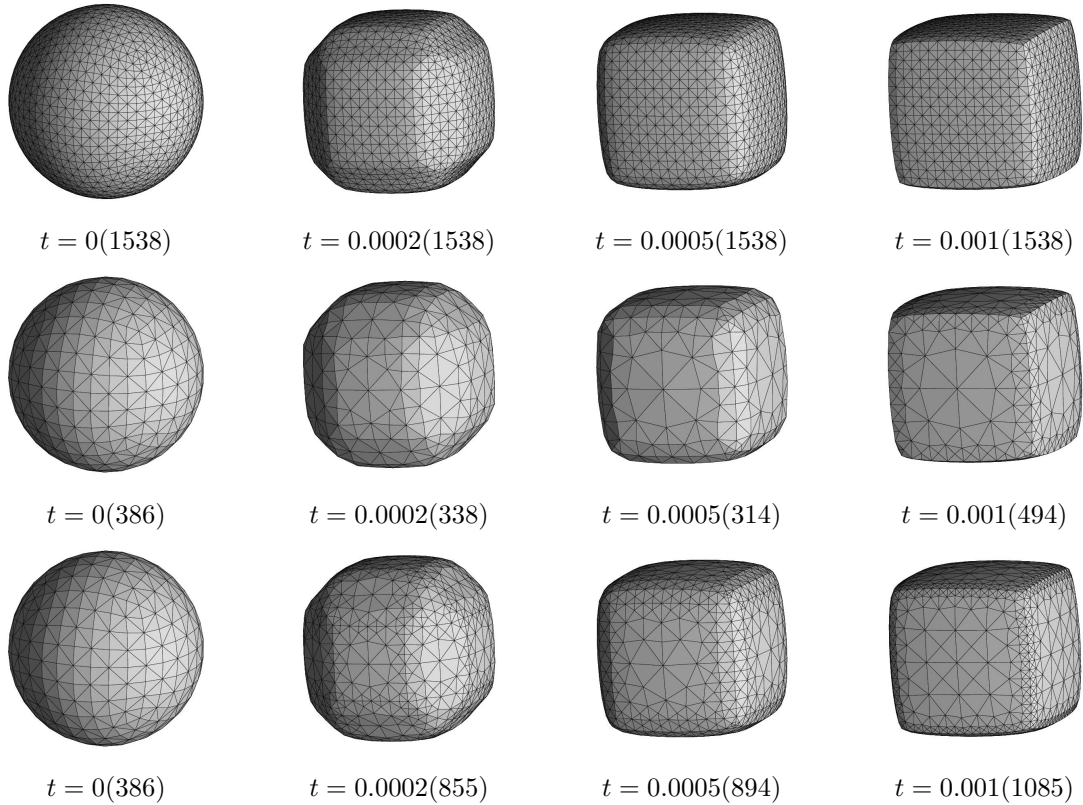
Figure 8.6: Evolution of the sphere with volume 1.0 towards its steady state at different times. The corresponding numbers of grid points are given in parentheses. First row: Fixed globally refined mesh. Second row: Adaptively refined mesh with $\varepsilon = 0.1$. Third row: Adaptively refined mesh with $\varepsilon = 0.05$.

Therefore, space adaptivity based on a geometric criterion as proposed in [7] is used: A local error indicator E_T estimates the accuracy for each triangle T. As long as E_T is larger than an element tolerance ε for at least one triangle T, elements with a large error indicator are refined and elements with a small indicator are coarsened. To prevent mesh distortion local mesh regularization and angle width control are used. Furthermore, a adaptive time step control considers the changing dynamics of the problem.

In the second row of Figure 8.6, the results for $\varepsilon = 0.1$ and in the third row the results for $\varepsilon = 0.05$ are shown. One can see that the mesh is refined in regions with high curvature and it is coarsened in regions with low curvature. If the element tolerance ε is smaller, the rounded corners of the Wulff shape are resolved more properly. In the case of $\varepsilon = 0.05$, the corners are resolved better than in the case of the globally refined mesh, while using less grid points.

For further details about this example see [27].

8.1.6   Geometric evolution by diffuse interface approximation

As mentioned in Section 8.1.2, phenomena like pinch-offs are hard to simulate by an explicit surface description with parametric finite elements, because in such cases topological mesh changes would be necessary. In this example, the geometric evolution of a surface due to surface diffusion

is simulated using a phase field model which implicitly describes the surface by a diffuse interface approximation.

This approach is based on the Cahn-Hilliard equation, which has already been introduced in Section 8.1.1. Thereby, a phase field variable is used to describe the two phases. The basic idea is to smear out the discrete function being 0 in one phase and 1 in the other one on a length scale of $\mathcal{O}(\varepsilon)$.

To get an approximation of surface diffusion, a degenerate mobility restricts the diffusion to the smeared out interface. The evolving surface now is described by the $\frac{1}{2}$ level set of the phase field variable. For $\varepsilon \to 0$, this diffuse interface approximation converges towards the sharp interface solution.

The simulation is done in 3d. To resolve the interface properly an adaptively refined mesh is used at the interface. As local error indicator, the jump residual of the phase field variable is used, which is small in the phases and large at the interface. Thus, an unnecessarily fine mesh is avoided away from the interface and computational costs are drastically reduced.

In addition, adaptive time step control is used in order to resolve critical time intervals (e.g. the time around a pinch-off) properly.

$t = 0$ $\quad\quad\quad\quad$ $t = 0.00015$ $\quad\quad\quad\quad$ $t = 0.00087$ $\quad\quad\quad\quad$ $t = 0.12003$

$t = 0.30029$ $\quad\quad\quad\quad$ $t = 0.39616$ $\quad\quad\quad\quad$ $t = 0.39699$ $\quad\quad\quad\quad$ $t = 0.50372$
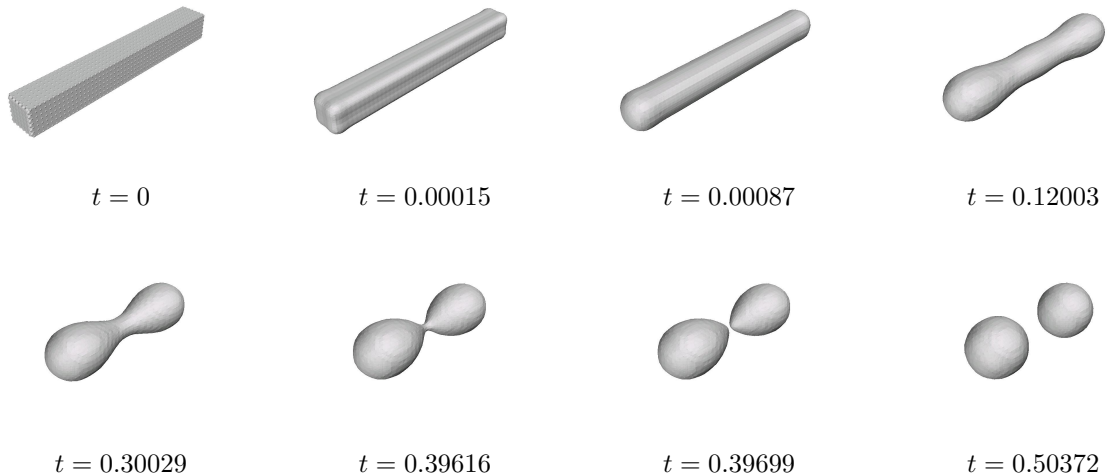
Figure 8.7: Evolution of a $8 \times 1 \times 1$ prism including pinch-off and two spheres of equal size as stationary solution.

In Figure 8.7, the evolution of the surface of a $8 \times 1 \times 1$ prism towards the steady state is shown. Around $t = 0.39699$ a pinch-off happens. After that, the two resulting surfaces evolve to spheres of same size. For more details about this simulation see [36].

8.2 Multigrid examples

In this section, several numerical results concerning the multigrid concepts, described in Chapter 5, are discussed.

8.2.1 Effect of level gap and Galerkin operator

We consider the following Poisson problem with Dirichlet boundary conditions:

$$-\Delta u = f \quad \text{in } \Omega \subset \mathbb{R}^d \tag{8.14}$$
$$u = g \quad \text{on } \partial\Omega \tag{8.15}$$
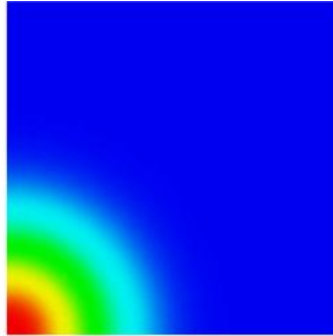
Figure 8.8: Solution of the Poisson equation on the unit square.

with

$$f(x) \quad = \quad -\left(400x^2 - 20d\right)e^{-10x^2} \tag{8.16}$$

$$g(x) \quad = \quad e^{-10x^2}, \tag{8.17}$$

where d is the problem dimension. Here, we set $d = 2$. The domain Ω is the unit square $(0,1)^2$. The Dirichlet boundary function g is set to the analytic solution of the problem which is shown in Figure 8.8.

The problem is discretized by linear finite elements on globally refined meshes. To study the multigrid behavior the size of the linear system of equations is consecutively enlarged by increasing the number of global refinements. The initial solution guess is set to 0 in all cases. Then the times that are needed for the solution of the system of equations are measured, including the multigrid initialization (construction of multigrid levels and level operators). The solver tolerance is set to 10^{-8}.

In the first test, the multigrid levels directly correspond to the mesh levels (no mesh levels are skipped, see Section 5.3.2), and the coarse grid level operators are re-assembled for each multigrid level (see Section 5.3.6). The number of pre- and post-smoothing steps is set to 1 in a V-cycle scheme.

For the second test, the multigrid level gap is set to 1. This means that every multigrid level is constructed as the union of two mesh levels, as described in Section 5.3.2. This reduces the number of multigrid levels and therefore the needed memory demand, too.

In the third test, Galerkin coarse grid operators are used. Instead of re-assembling the system matrix for each multigrid level, the finest level matrix is restricted to coarser levels (see Section 5.3.6).

In Figure 8.9, the solution times are plotted versus the number of unknowns for the different cases. In all cases, the solution time depends linearly on the number of unknowns. If the level gap is set to 1, the performance is slightly improved. In this case, more multigrid cycles are necessary, but each cycle is less expensive. The usage of Galerkin coarse grid operators highly improves the multigrid performance. Here, the construction of the level matrices is less expensive and a better convergence rate is achieved. But as described in Section 5.3.6, Galerkin operators are only useful for linear finite elements. For higher order elements the restriction of fine grid matrices would lead to more and more dense coarse grid matrices.

8.2.2 Comparison with other solvers

Now the multigrid performance is compared to other iterative solvers, namely the Krylov methods CG, BiCGStab and GMRes. The problem is discretized as in the last section and the multigrid solver with Galerkin operator and level gap is used.
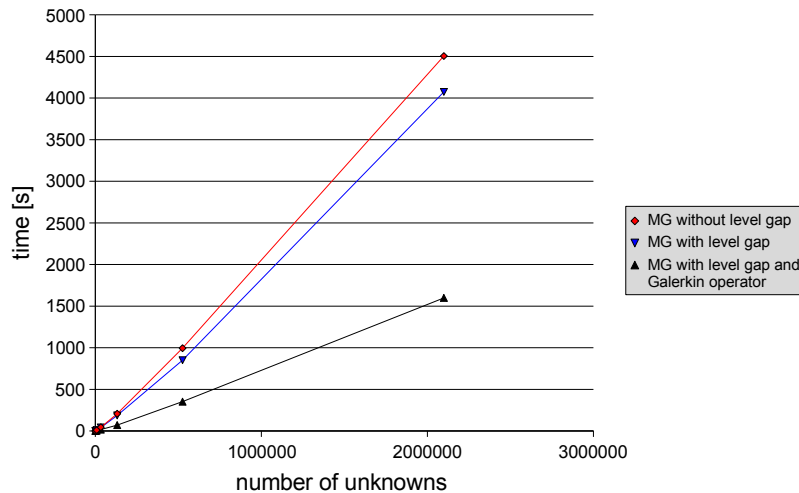
Figure 8.9: Multigrid solution times for the two dimensional Poisson equation with globally refined meshes and linear basis functions. The times are plotted versus the number of unknowns used in the computations. All solvers always start with an initial guess of zero. One can see the effect of the level gap (here, every second level is skipped) and of the usage of Galerkin coarse grid operators.

In Figure 8.10, the corresponding solution times are shown. For large systems of equations the multigrid method is faster than all other solvers. Even the CG solver, which is specialized on symmetric and positive-definite matrices like we have in this example, is slower for large systems. As the other Krylov solvers, it has an over linear complexity. BiCGStab and GMRes can, just like multigrid, be applied to more general systems of equations. But for these solvers the multigrid advance is much larger.
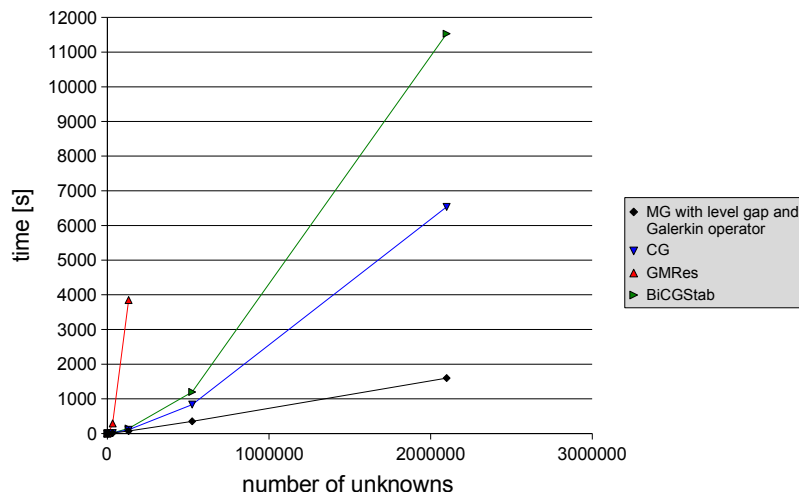
Figure 8.10: Comparison of multigrid with other iterative solvers. For large systems of equations, multigrid provides the best performance. It is also faster than CG which is specialized to symmetric positive definite matrices, in contrast to multigrid.

8.2.3 Multigrid for higher order elements

If finite elements of higher order are used, Galerkin operators can not be applied to coarser levels. In this case, the coarse grid matrices have to be assembled in the same way as the finest level matrix. In this example, basis functions of degree 3 are used. All other settings correspond to the previous sections.

In Figure 8.11, the multigrid method with and without level gap are compared with a CG solver. Again, the level gap leads to a slightly improved multigrid performance and less memory demand. Additionally, in this case of higher order elements, the multigrid solver is faster than CG for large systems of equations.
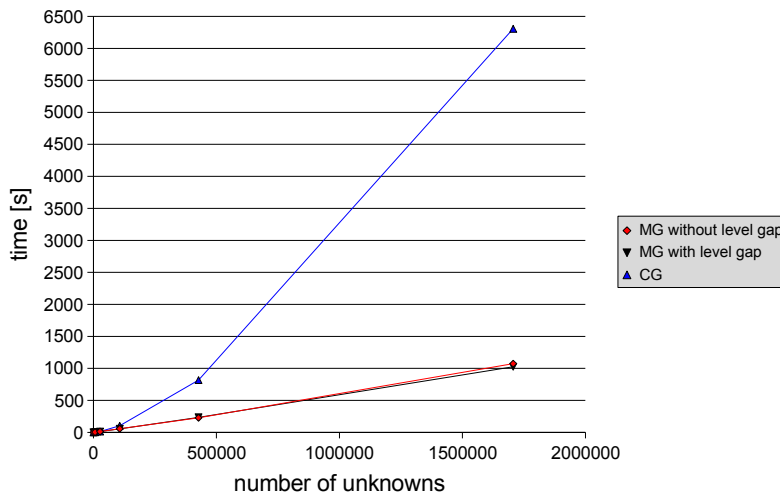
Figure 8.11: Multigrid for finite elements of degree 3 compared to CG. For higher order elements no Galerkin operators can be used. So the coarse grid operators must be re-assembled for every level. The multigrid times with level gap approximately corresponds to the times without level gap. Therefore, with level gap less memory is needed.

8.2.4 Adaptive Multigrid

In the last examples, the multigrid method was applied to globally refined meshes. In this example, we start with a very coarse macro mesh which is adaptively refined. In every iteration of the adaptation loop, the solution of the last iteration, interpolated to the refined mesh, is used as initial guess for the solver. Because we use linear finite elements, we can apply Galerkin coarse grid operators. Again, the level gap is set to 1. The multigrid performance is compared with CG and GMRes.

In Figure 8.12, one can see, that for large systems again multigrid is the fastest solver. However, in this example, CG is not much slower than multigrid. The reason is that CG highly benefits from the good initial guess of the last adaptive iteration. However, also in this example, multigrid is much faster than GMRes.

8.2.5 Multigrid on the sphere

In this example, the multigrid method is applied to a problem defined on the unit sphere S. To the best of my knowledge, I'm not aware of any other multigrid approach which is applied to solve problems defined on a surface. In AMDiS the same algorithm is used as described in Chapter 5.
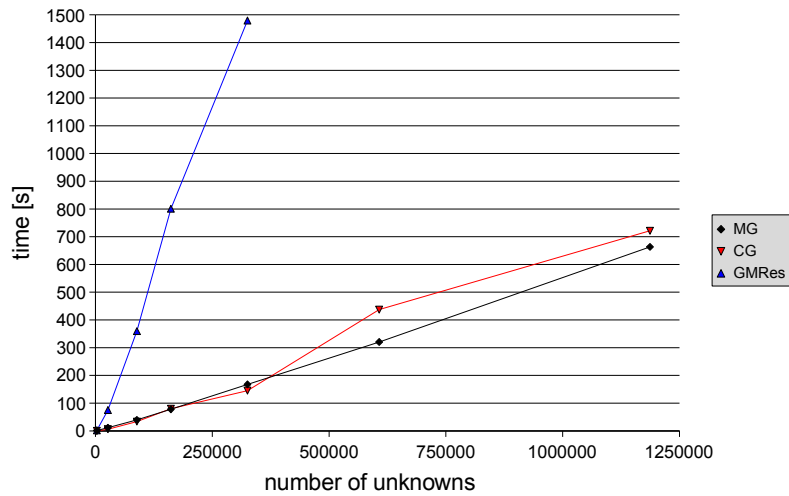
Figure 8.12: Multigrid compared to CG and GMRes for adaptive mesh refinements. Again, for large systems multigrid is faster than the other solvers. But in this case, in every adaptive iteration the interpolated solution of the last iteration is used as initial guess. The CG solver obviously takes a strong benefit from this fact, so it is not much slower than multigrid.

The problem is defined by

$$\Delta u(x) \quad = \quad 2x_0 \quad \text{on } S, \tag{8.18}$$

where x_0 is the first component of x.

The macro mesh is defined by 12 triangles that build the surface of a cube. Every new vertex that is created during refinement is projected to the unit sphere. We use linear finite elements and global refinements. The initial solution for every level is set to 0.

Figure 8.13 shows the solutions at different refinement levels. The meshes at different levels describe different domains. The finer the mesh the better is the approximation of the sphere. Thus, also the different multigrid levels for the solution on a fine mesh describe different domains. So far, in theory it is not clear, which influence this domain adaptivity has on the multigrid method. But in practice, it does not seem to have a bad influence on the multigrid accuracy.

Figure 8.14 shows that the multigrid performance compared to the other solvers is similar as in the pure 2d examples.

8.3 Parallelization examples

This section discusses some numerical results according to the parallelization concepts described in Chapter 6.

8.3.1 Varying local coarse grid level

To analyze the effect of the local coarse grid level, introduced in Section 6.2.3, we look at the following two dimensional Poisson problem:

$$-\Delta u \quad = \quad f \quad \text{in } \Omega \tag{8.19}$$

$$u \quad = \quad g \quad \text{on } \partial\Omega \tag{8.20}$$
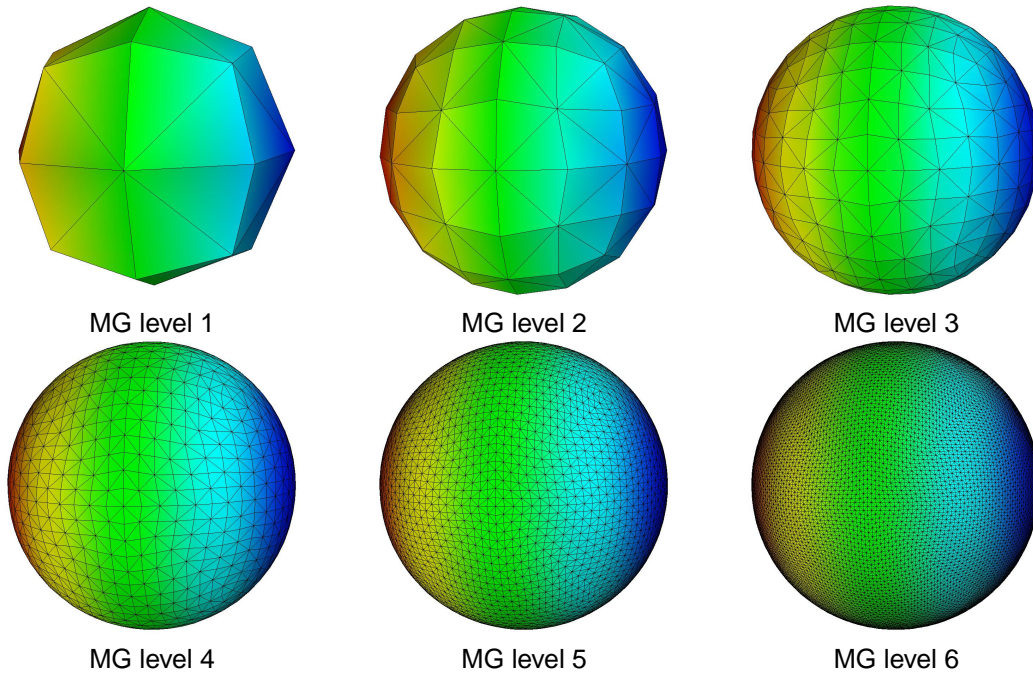
Figure 8.13: Meshes of multigrid levels 1-6 with corresponding solutions. The meshes of different levels describe different domains which is clear to see for coarse levels.
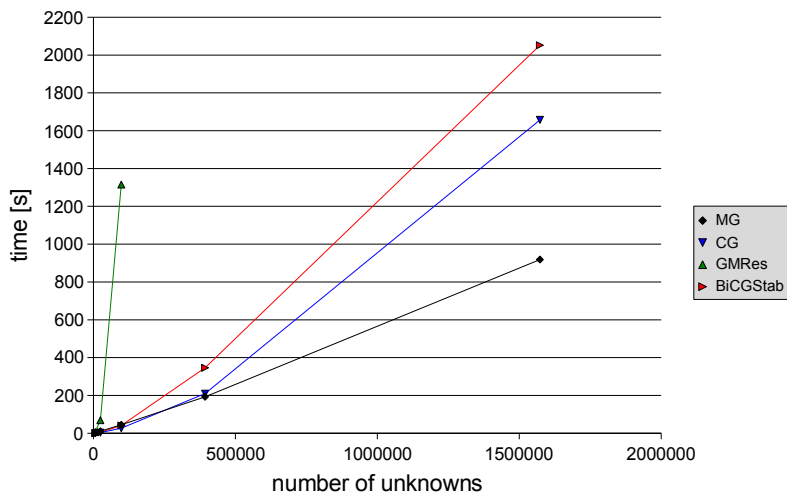
Figure 8.14: Solution times for the Poisson equation solved on the sphere. Also in this example, multigrid is faster than the other solvers for large systems.

with

$$f(x) \quad = \quad n\left(\sin(2\pi n x_0) + \sin(2\pi n x_1)\right) \tag{8.21}$$

$$g(x) \quad = \quad \frac{1}{4\pi^2 n}\left(\sin(2\pi n x_0) + \sin(2\pi n x_1)\right) \tag{8.22}$$

with $x = (x_0, x_1)$. The right-hand side f and boundary function g are constructed such that the problem results in a sine-shaped solution. The scaling parameter $n \in \mathbb{N}$ determines frequency and amplitude of the solution. In this example, we set $n = 4$ and solve on $\Omega = (0,1)^2$. Figure 8.15
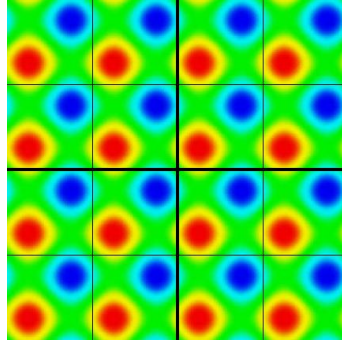
Figure 8.15: Solution of (8.19) and (8.20) with f and g according to (8.21) and (8.22) with $n = 4$.
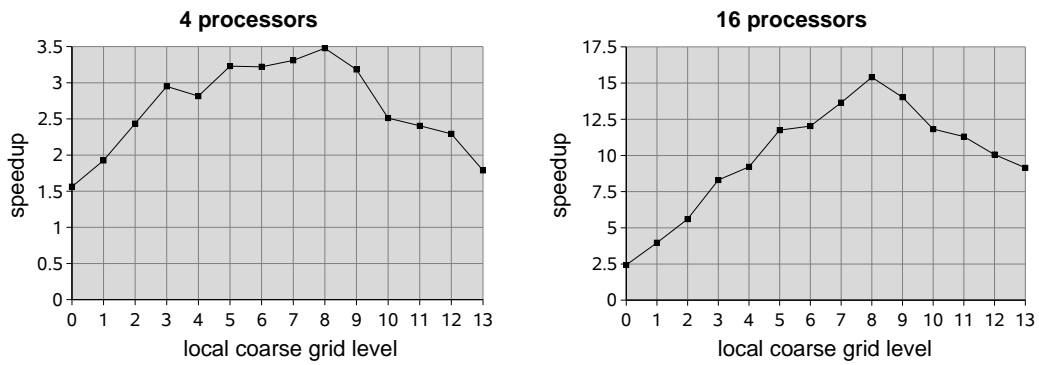
Figure 8.16: Speedup factors for different local coarse grid levels with 4 and 16 processes. The local coarse grid level describes the number of uniform refinements within Ω_i starting from the partitioning level (see Section 6.2.3).
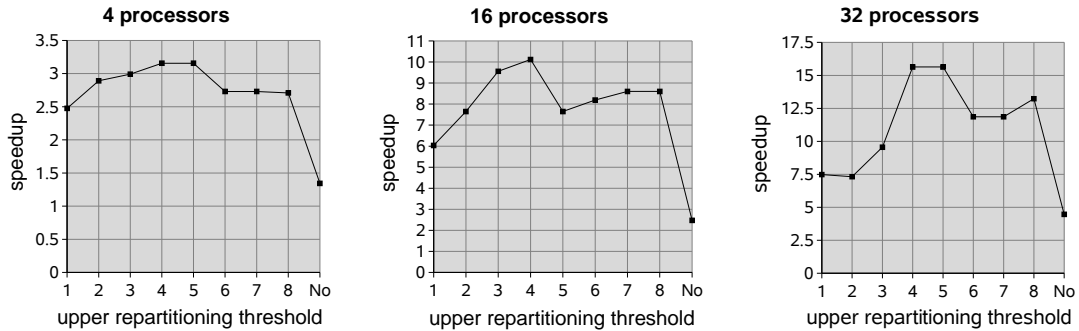
Figure 8.17: Speedup for varying repartitioning values with 4, 16 and 32 processes. The lower repartitioning threshold is always set to the reciprocal of the upper threshold. The last value in each case describes the speedup when no repartitioning was performed at all.

shows the corresponding solution and partition boundaries for 4 partitions (thick lines) and 16 partitions (thin lines). Because of the periodic shape, in each partition nearly the same sub-problem has to be solved (except for different boundary situations). Thus, we obtain an optimal load balance in each iteration, and no repartitionings are necessary. We perform the initial partitioning on a mesh consisting of 64 triangles (4 global refinements of 4 macro elements), and measure the speedup to the corresponding sequential code for different choices of the local coarse grid level. The adaptation tolerance was set to $5 \cdot 10^{-4}$.

Figure 8.16 shows the resulting speedup factors for computations with 4 and with 16 processes. In both cases a local coarse grid level of 0 results in a very bad speedup. The reason is that here the overlap for each partition was computed at the relatively coarse partitioning level and covers a large part of Ω. On those overlap regions, more than one process computes the solution on a fine mesh. In the worst case, where the overlap of each partition covers the whole remaining domain, every process would perform exactly the same computations on the same mesh on the whole domain. Including the overhead for partitioning and building the global solution, this would result in a speedup factor smaller than one. Increasing the local coarse grid level, the size of the resulting overlap will be decreased. For a local coarse grid level of 8 we obtain a speedup of 3.48 with 4 processes and a speedup of 15.41 with 16 processes. For higher values the speedup becomes worse again, because the number of needed adaptive iterations to reach the desired tolerance is getting smaller. Thus, the needed time is getting smaller, too, and the relative parallelization overhead grows.

8.3.2   Varying repartitioning thresholds

While the example in Section 8.3.1 was constructed to avoid any repartitionings, the next example explores the influence of repartitioning thresholds, described in Section 6.2.4. For this purpose, we again use the Poisson problem of Section 8.2.1.

This problem can be seen as a worst case scenario for the parallelization approach, because the irregularity of the solution can never be resolved by all processes equally.

The solution on $\Omega = (0, 1)^2$ is shown in Figure 8.8. The fact that the source is located in a corner of the domain leads to successive refinements towards this corner and to load imbalance between the partitions. We use the 8 times globally refined macro mesh as the partitioning level and set the local coarse grid level to 4. Then we solve the problem with 4, 16 and 32 processes with a tolerance of $5 \cdot 10^{-4}$ and with varying values of upper and lower repartitioning thresholds. Then we compare the computation times with the sequential case. In Figure 8.17, the results are shown for upper repartitioning thresholds between 1 and 8. The lower repartitioning threshold was always set to the reciprocal of the upper one. *No* on the x-axis means that no repartitioning was performed during the whole computation. This was realized by setting the upper threshold

to 1000000 and the lower one to 0. As one can see, the optimal value for the upper threshold in this example is 4 in all cases. The benefit compared to an upper threshold of 1 increases when more processes are used, whereas the overall relative speedup is worse if more processes are used. The worst speedup is achieved in each case when no repartitioning was performed. This is further illustrated by comparing our approach with [5].

The optimal choice for the repartitioning thresholds is probably problem specific. Therefore, an algorithm was added which adapts the thresholds depending on the relationship between the time used for the last repartitioning and the elapsed computation time since the last repartitioning. The resulting speedup was similar to that with the fixed optimal thresholds.

8.3.3   Comparison with the approach of Bank and Holst

Bank and Holst make the assumption that partitions with approximately equal error lead to approximately equal work for each process. They note in [5] that this is a fragile assumption, but show that it works well for several examples.

However, for the problem defined in Section 8.3.2 this assumption does not hold. Using the approach of Bank and Holst, partitioning is done only once at a relatively coarse mesh at the beginning of the computation. Element weights are the local error estimates of this mesh. We use four processes. The partitioning results in four partitions with approximately the same estimation sums. The largest errors were estimated in the lower left corner of the domain. This leads to small partitions 0, 1 and 2 around this corner, compared to the much larger partition 3, see Figure 8.18 (a).

As described in [5], we avoid any communication within the adaptation loop to decouple the iterations of the different processes. Therefore, decisions concerning which elements are marked for refinement and when the total tolerance is reached, can be based only on local process information. We distribute the total tolerance equally between the four processes, which is reasonable if the assumption holds that equal errors lead to equal work loads.

In Figure 8.18 (b), the mesh after the parallel computation is shown. Figure 8.18 (c) shows the mesh after the corresponding sequential computation, which differs from the parallel computation. In this example, the reached speedup factor was 0.6, which actually is a slow down for the parallel case. One reason for this bad result is that it needs much less work to reduce the error on the small partitions 0, 1 and 2 with very few elements, than on the large partition 3, where the same error is distributed on many more elements. The final mesh of partition 3 has over 325,000 elements, whereas the meshes of partition 0 and 1 have about 7,000 elements, and the mesh of partition 2 has 20,000 elements. Thus, nearly the whole work is done by process 3.

This effect is enhanced by distributing the tolerance equally between the processes. We force each process to reduce the error by the same factor, which is not what happens in the sequential case. Bank and Holst stop the computations at each process, when a given target number of elements or degrees of freedom is reached. This, however, does not guarantee a solution, which fulfills the given tolerance criterion. Furthermore, a comparison with the sequential case can not be done. Therefore, we use the tolerance to stop the computation.

For this example our approach provides a far better speedup than the Bank and Holst approach does. In Section 8.3.2, we showed that, with the right repartitioning thresholds, a speedup of over 3 can be reached. Even if no repartitionings are applied, the speedup is still better than with the Bank and Holst approach, see Figure 8.17.

8.3.4   Scaled problem domain

The previous examples showed that the relative speedup (speedup divided by number of used processes) for a given problem gets worse, when the number of processes grows. One way to avoid that, is to adapt the local coarse grid level according to the number of used processes. In this section, we analyze another kind of scalability: We use a higher number of processes to solve an accordingly more complex problem. This is a realistic scenario for many problems in
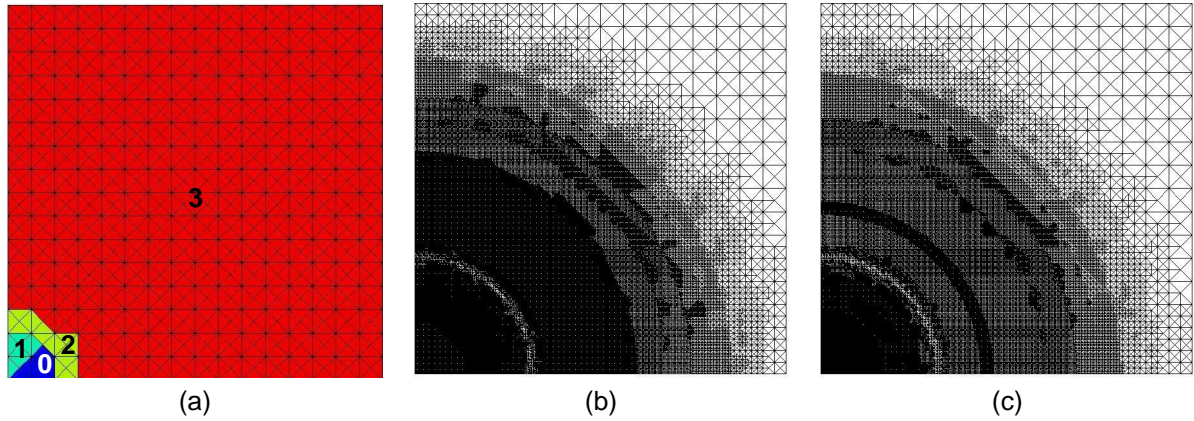
Figure 8.18: (a): The four partitions with approximately equal estimation sums, (b): The composite mesh after the parallel computation, (c): The mesh after sequential computation.

(a) $n = 1$ (b) $n = 2$ (c) $n = 4$ (d) $n = 8$
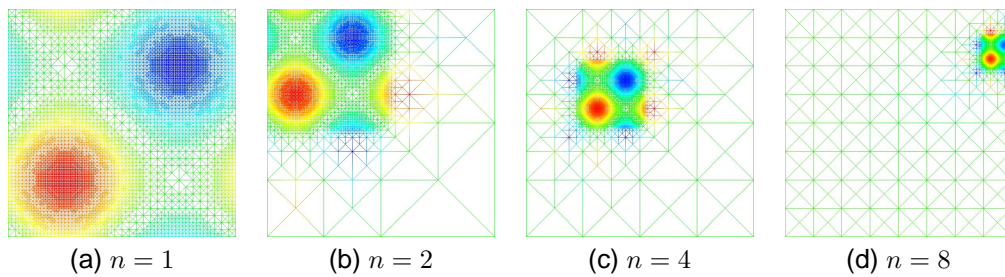
Figure 8.19: Final mesh of partition 0 in the different cases. The value of n corresponds to the square root of the used process number.

p	1	4	16	64
factor	1	1.279	1.496	1.589

Table 8.1: Time factor compared to the sequential case
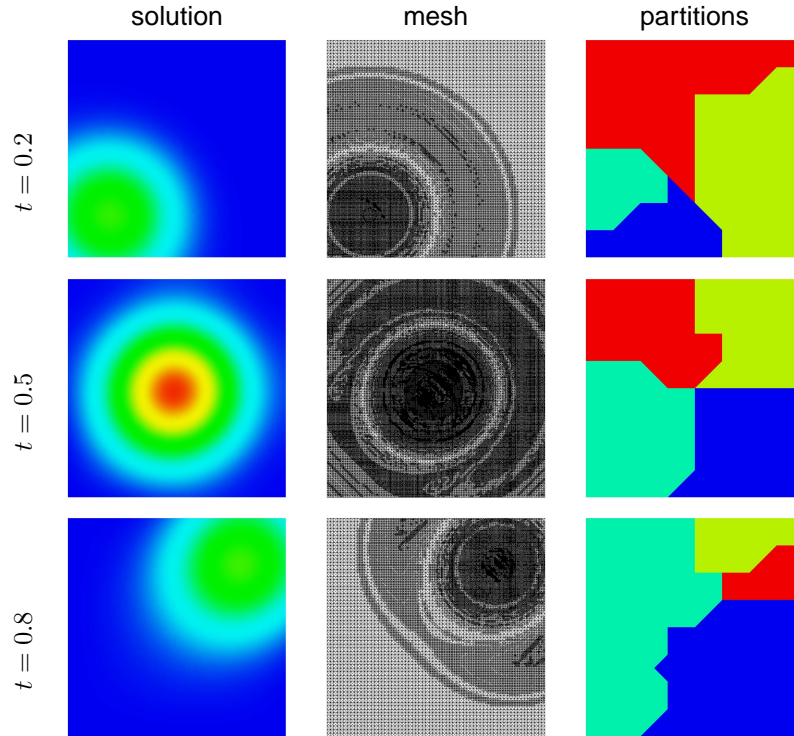
Figure 8.20: Solution, resulting mesh and corresponding partitions at different times.

materials science, where the overall goal is not a reduction of computing time but the increase in domain size, see e.g. [3]. We use the problem defined by equations (8.19) – (8.22) in Section 8.3.1 again, but now we use different values for n. To double the value of n is equivalent to double the expansion of Ω in both directions and continue the problem formulation on the extension. We use $n = \sqrt{p}$ with p the number of processes for $p \in \{1, 4, 16, 64\}$. The partitioning level is set to $\log_2(p)$, the local coarse grid level is set to 8. In Figure 8.19 the final meshes for partition 0 in the different cases are shown.

In Table 8.1, the resulting time factors compared to the sequential case are shown. Solving the problem, which is 64 times more complex than the original one, needs only about 1.6 times the time when using 64 processes.

8.3.5 Moving source

In this section, we give an example of a simple time dependent problem solved in parallel with four processes. The problem is defined by

$$
\begin{aligned}
u_t(x,t) - \Delta u(x,t) &= f(x,t) &&\text{in } \Omega \\
u(x,t) &= g(x,t) &&\text{on } \partial\Omega.
\end{aligned}
$$

The function u_t is the time derivative of u. The functions f and g are chosen such that the source is moving from the lower left corner to the upper right corner of $\Omega = (0,1)^2$ while the time t proceeds from 0 to 1. Furthermore, the source amplitude grows from 0 to 1 until $t = 0.5$ and falls

(a) $t = 0.001187$ (b) $t = 0.271900$ (c) partition 0
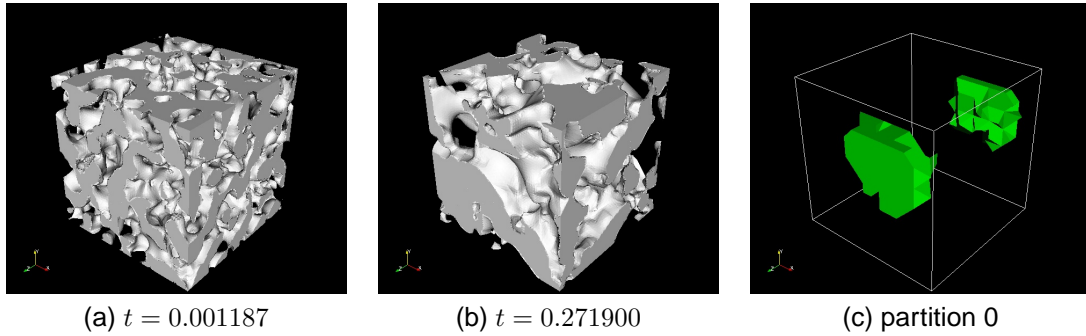
Figure 8.21: (a)–(b): Evolution of u at two times. The value $u = 0$ is denoted by opaque volume, $u = 1$ by transparent volume. (c) Volume of partition 0, connected through periodic boundary conditions. The simulations were performed by A. Rätz.

back to 0 until $t = 1.0$ following a sine function. We use a fixed timestep of 0.1 and an implicit time strategy, which adapts in each timestep until the tolerance of 10^{-4} is reached. Furthermore, element coarsening is allowed if local error indicators are sufficiently small. We use a partitioning level of 6 and a local coarse grid level of 6. Upper and lower repartitioning thresholds are set to $\frac{3}{2}$ and $\frac{2}{3}$, respectively. In Figure 8.20, we can see, how the solution changes over time. The meshes at the different times are adapted to the current solution. On the right part of the figure, we see the partitions at the beginning of the corresponding timesteps. One can see that the partitioning follows the mesh changes to obtain a good load balance in each timestep. The speedup compared to the sequential solution of the same problem was 3.65.

8.3.6 Higher order problem in 3d

This is an example of a time dependent three dimensional parallel computation for a system of PDEs. Here, we look at a classical model for spinodal decomposition of a binary alloy, the Cahn-Hilliard equation. The Cahn-Hilliard equation and its implementation in AMDiS has already been described in Section 8.1.1. The only difference here is that it is solved on $\Omega = [-1, 1]^3$, discretized by a mesh consisting of about 6.3 million tetrahedra, and with periodic boundary conditions applied on $\partial\Omega$. This discretization is too fine to be covered by one processor. In this example, 24 processes are used.

Figure 8.21 (a)–(b) shows the evolution of u at different times. The solution quickly separates Ω into two regions Ω_0 and Ω_1, where u takes the values of 0 and 1, respectively. The remaining part of Ω lies on an interface of width $\mathcal{O}(\epsilon)$ between the two regions. At later stages, Ω_0 and Ω_1 change shape so that the surface of the interface between the two regions decreases while maintaining the volume of Ω_0 and Ω_1. In Figure 8.21 (c), the volume of partition 0, connected through a periodic boundary, is visualized.

Chapter 9

Conclusion and outlook

In this work, the software concepts of the finite element toolbox AMDiS were described and its possibilities have been demonstrated. The problem definition that can be done on a high abstraction level in a dimension independent way allows a fast and intuitive implementation of many different application types. Great emphasis is put on the treatment of systems of PDEs. The implementation of parametric finite elements makes it possible to solve problems on arbitrary manifolds and on moving meshes. Thereby, the parameterization is done automatically within the mesh traversal transparent to the user. Furthermore, the interface oriented implementation of adaptation loops and its loose coupling to concrete problem implementations ensures a high reusability of adaptation strategies and provides an easy way of coupling problems.

A main part of this work was the development of a multigrid solver in the context of adaptive finite elements. The multigrid method implemented in AMDiS allows a fast solution of many problems by utilizing the hierarchical mesh structure to construct the different multigrid levels. In addition, by applying the full approximation scheme, adaptive mesh refinements can be considered in an efficient way. The usage of Galerkin coarse grid operators often leads to an improved multigrid performance in the case of linear finite elements. Furthermore, skipping some mesh levels in the construction of multigrid levels mostly leads to a slightly improved performance together with a clearly reduced memory demand.

To take advantage of modern computing clusters, a new parallelization approach was developed. Using the concept of adaptive full domain covering meshes, sequential codes can be parallelized in an easy way producing little communication needs. Thereby, mesh structure codes are used to exchange mesh information in a very efficient way between the processes. In addition, the parallel speedup can be optimized by varying parallelization parameters like local coarse grid level and repartitioning thresholds. If the problem complexity increases together with the number of used processes, a good scalability of the approach can be observed. Time dependent problems and systems of PDEs are treated in a straightforward way. The number of processes used in the examples was moderate. If the number of processes drastically increases, the need for communication in traditional parallelization concepts becomes an issue. I believe the concept of adaptive full domain covering meshes to be a useful tool to overcome this problem as the need for communication is reduced to a minimum.

The Shared Mesh Interface (SMI) and its client-server implementation over TCP/IP allows the distributed management of meshes that are shared between different applications maybe running on different machines. One application example is the loose coupling of simulation and visualization codes. Besides the pure mesh management, aspects like transaction management, synchronization points, user defined relations and iterators allow an easy an robust usage of SMI.

Up to now, AMDiS is a C++ library without any graphical user interface for the problem definition. But in principle, it is already possible to define problems at runtime. Therefore, one aspect of future work could be the implementation of a user front end that allows the problem definition in an interactive way. In a first step, a XML problem description language could be developed which

117

later could be used as interface between front end and C++ library.

A further aspect of future work could be the independent refinement of different component meshes for a system of PDEs. In the current AMDiS implementation, each component can be discretized by different basis functions but all components must share the same mesh. Independent mesh refinements for each component would demand a deep intervention into the system assemblage routines, but it would allow a better consideration of different component properties.

Appendix A

AMDiS tutorial

A.1   Introduction

The objective of this tutorial is to introduce the user into the main AMDiS features by giving some application examples.

Section A.2 describes the installation of the AMDiS library and the building of user applications step by step. The corresponding application makefile is given in Section A.3.

In Section A.4, for every example the following aspects are described:

- **Abstract problem description**: In the header of each example section, the abstract problem definition is given. Sometimes, some solution strategies on a high abstraction level are mentioned, also.

- **Source code**: In the source code section, the listing of the example source code is explained.

- **Parameter file**: In this section, the parameter file is described. The parameter file contains parameters which are read be the application at runtime. The name of the parameter file is usually passed to the the application as a command line argument.

- **Macro file**: In the macro file section, the definition of the coarse macro mesh is shown, which is the basis for adaptive refinements.

- **Output**: The AMDiS results are written to output files that contain the final mesh and the problem solution on this mesh. The output can be visualized by proper tools (*CrystalClear, ParaView, TecPlot*). In the output section, the visualized problem results are shown and discussed.

To avoid unnecessary repetitions, not every aspect of every example is described, but only those aspects that have not appeared in previous examples.

A.2   Installation

A.2.1   Installation of the AMDiS library

To install the AMDiS library, the following steps must be performed:

1. Create the AMDiS source directory. This can be done in different ways. Here, we show three possibilities:

   - Unpacking the archive file `AMDiS.tar.gz`:

- – > `gunzip AMDiS.tar.gz`
- – > `tar xvf AMDiS.tar`
- • Checking out a CVS project:
  - – > `export CVSROOT=<cvsroot>`
  - – > `cvs checkout AMDiS`
- • Checking out a CVS project:
  - – > `svn checkout file://<SVN-REPOSITORY-PATH>/AMDiS`

2. Change into the AMDiS directory:

   > `cd AMDiS`

3. Create the makefiles for your system using the `configure` script:

   > `./configure <CONFIGURE-OPTIONS>`

   The `configure` script creates the needed makefiles. It can be called with the following options:

   `--prefix=<AMDIS-DIR>`: Installation path of the AMDiS library. The library file will be stored in `<AMDIS-DIR>/lib`. With `--prefix='pwd'` AMDiS will be installed in the current working directory, which mostly is a good choice.

   `--enable-debug`: If this option is used, AMDiS will be compiled with debug support. By default, AMDiS is compiled in an optimized mode without debug support.

   `--with-mpi=<MPI-DIR>`: MPI installation path. Used for parallelization support.

   `--with-parmetis=<PARMETIS-DIR>`: ParMETIS installation path. ParMETIS is a parallel graph partitioning library, see
   `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`. Used for parallelization support.

   If AMDiS should be compiled for parallel usage, the MPI and ParMETIS paths must be set.

4. Make the library:

   > `make install`

If you have added a new source file or you want to change something on the automake-system, you have to rerun the following commands:

```
> libtoolize --force --copy
> aclocal
> autoconf
> automake --add-missing --copy
```

Then repeat steps 3 and 4. For the additional steps `libtool`, `automake` and `autotools` must be installed on your system.

Further information about the installation process can be found in the `README` file in the AMDiS source directory.

A.2.2 Compilation of an example application

For the compilation of the examples, described in this section, the following steps must be executed:

1. Get the sources:

   - Unpack an archive file:
     ```
     > gunzip demo.tar.gz
     > tar xvf demo.tar
     ```
     or
   - CVS checkout:
     ```
     > export CVSROOT=<CVSROOT>
     > cvs checkout demo
     ```
     or
   - SVN checkout:
     ```
     > svn checkout file://<SVN-REPOSITORY-PATH>/demo
     ```

2. Change into the demo directory:
   ```
   > cd demo
   ```

3. Edit the `Makefile`:

   - Set the AMDiS path and paths of other needed libraries.
   - Set user flags.

   The makefile is described in Section A.3 in detail.

4. Make the application example:
   ```
   > make <PROG-NAME>
   ```

   `<PROG-NAME>` is the name of the application example.

To run the example, call:

- In the sequential case:
  ```
  > ./<PROG-NAME> <PARAMETER-FILE>
  ```

- In the parallel case:
  ```
  > mpirun <MPI-OPTIONS> ./<PROG-NAME> <PARAMETER-FILE>
  ```

  The `<MPI-OPTIONS>` at least should contain the number of used processes, which is given by `-np <NUM-PROCS>`. For further MPI options see `http://www-unix.mcs.anl.gov/mpi/`.

A.3 Application makefile

In this section, the organization of the application makefile is described which is used for the examples in this tutorial. The same organization can be used for other user applications, too.

In the first block, user flags and directories are specified.

```
# ============================================================================
# ===== flags and directories (to be modified by the user) ===================
# ============================================================================
```

```
USE_PARALLEL_AMDIS = 0              # 0: sequential AMDiS, 1: parallel AMDiS
DEBUG             = 0              # 0: no debug, 1: debug mode

AMDIS_DIR    = <AMDIS-DIR>        # fill the AMDiS installation path here
MPI_DIR      = <MPI-DIR>          # fill the MPI installation path here
PARMETIS_DIR = <PARMETIS-DIR>     # fill the ParMETIS installation path here
```

If `USE_PARALLEL_AMDIS` is set to 1, parallel applications will be supported. A necessary condition is that the AMDiS library is configured for parallelization, too (see Chapter A.2). The `DEBUG` entry specifies, whether applications should be compiled in debug mode, or not. This entry is independent of the corresponding AMDiS settings, but if the AMDiS library was not compiled in debug mode, only application code can be debugged.

`AMDIS_DIR` stores the AMDiS installation path and must be set by the user. This is the path given to the AMDiS configure script by the `--prefix` option. The values of `MPI_DIR` and `PARMETIS_DIR` only are needed if parallelization should be supported. Here, the installation pathes of MPI and ParMETIS are stored.

In the next block, include pathes are defined.

```
# ============================================================================
# ===== includes pathes ======================================================
# ============================================================================

AMDIS_INCLUDE    = -I$(AMDIS_DIR)/src
MPI_INCLUDE      = -I$(MPI_DIR)/include
PARMETIS_INCLUDE = -I$(PARMETIS_DIR)

INCLUDES = -I. $(AMDIS_INCLUDE) $(MPI_INCLUDE) $(PARMETIS_INCLUDE)
```

Now, we introduce the needed libraries.

```
# ============================================================================
# ===== libraries ============================================================
# ============================================================================

AMDIS_LIB    = -L$(AMDIS_DIR)/lib -lamdis
PARMETIS_LIB = -L$(PARMETIS_DIR) -lparmetis -lmetis

LIBS = $(AMDIS_LIB)
```

By default, `LIBS` contains only the AMDiS library. If `USE_PARALLEL_AMDIS` is 1, `LIBS` is extended by the ParMETIS library. In the same way, other libraries can be added. In the sequential case, we use the GNU C++ compiler `g++`, in the parallel case, the MPI C++ compiler `mpiCC`.

```
# ============================================================================
# ===== parallel or sequential ? =============================================
# ============================================================================

ifeq ($(USE_PARALLEL_AMDIS), 0)
      COMPILE = g++
else
      COMPILE = $(MPI_DIR)/bin/mpiCC
      LIBS += $(PARMETIS_LIB)
endif
```

The next block sets the compile flags. In debug mode, we use no optimization (`-O0`) and add symbolic debug information (`-g`). Otherwise, we compile with optimazation level 2 (`-O2`).

```
# ================================================================================
# ===== compile flags ===========================================================
# ================================================================================

ifeq ($(DEBUG), 0)
        CPPFLAGS = -O2
else
        CPPFLAGS = -g -O0
endif
```

We use the `libtool` in the AMDiS installation path for linking.

```
# ================================================================================
# ===== libtool linking =========================================================
# ================================================================================

LIBTOOL = $(AMDIS_DIR)/libtool
LINK = $(LIBTOOL) --mode=link $(COMPILE)
```

Now, we define rules to create and delete objects files.

```
# ================================================================================
# ===== rules ====================================================================
# ================================================================================

clean:
        -rm -rf *.o

.cc.o: $*.cc
        $(COMPILE) $(INCLUDES) $(CPPFLAGS) -c -o $*.o $^
```

The second rule creates needed object files automatically using the corresponding C++ files.

Finally, we define rules for the linking of user applications. Here, we present only the rule for the `ellipt` application. Other applications can be created in an analog way.

```
# ================================================================================
# ===== user programs ============================================================
# ================================================================================

VPATH = .:../src:$(AMDIS_DIR)/src

# ===== myprog ===================================================================

ELLIPT_OFILES = ellipt.o

ellipt: $(ELLIPT_OFILES)
        $(LINK) $(CPPFLAGS) -o ellipt $(ELLIPT_OFILES) $(LIBS)
```

The `VPATH` variable contains all pathes, where sources can be located. The `ellipt` rule first creates all needed object files defined in `ELLIPT_OFILES`. In this example, only `ellipt.o` is needed. Then all needed object files and libraries are linked together. The `-o` option specifies that the executable will be written to the file `ellipt`.
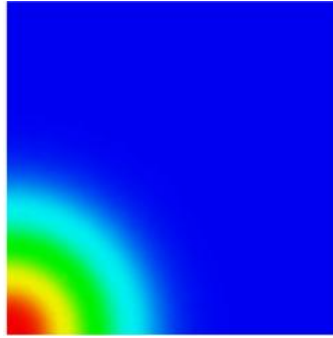
Figure A.1: Solution of the Poisson equation on the unit square.

		1d	2d	3d
source code	`src/`	`ellipt.cc`		
parameter file	`init/`	`ellipt.dat.1d`	`ellipt.dat.2d`	`ellipt.dat.3d`
macro file	`macro/`	`macro.stand.1d`	`macro.stand.2d`	`macro.stand.3d`
output files	`output/`	`ellipt.mesh, ellipt.dat`		

Table A.1: Files of the `ellipt` example.

A.4   Implementation of example problems

A.4.1   Stationary problem with Dirichlet boundary condition

As example for a stationary problem, we choose the Poisson equation

$$-\Delta u = f \quad \text{in } \Omega \subset \mathbb{R}^{dim} \tag{A.1}$$
$$u = g \quad \text{on } \partial\Omega \tag{A.2}$$

with

$$f(x) = -\left(400x^2 - 20dow\right)e^{-10x^2} \tag{A.3}$$
$$g(x) = e^{-10x^2}. \tag{A.4}$$

dim is the problem dimension and thus the dimension of the mesh the problem will be discretized on. dow is the dimension of the world the problem lives in. So world coordinates are always real valued vectors of size dow. Note that the problem is defined in a dimension independent way. Furthermore, dow has not to be equal to dim as long as $1 \leq dim \leq dow \leq 3$ holds.

Although the implementation described in Section A.4.1 is dimension independent, we focus on the case $dim = dow = 2$ for the rest of this section. The analytical solution on $\Omega = [0,1] \times [0,1]$ is shown in Figure A.1.

Source code

For this first example, we give the complete source code. But to avoid loosing the overview, we sometimes interrupt the code to give some explaining comment. The first three lines of the application code are:

```
#include "AMDiS.h"
using namespace std;
using namespace AMDiS;
```

In the first line, the AMDiS header is included. In line 2 and 3, used namespaces are introduced. `std` is the C++ standard library namespace, used e.g. for the STL classes. AMDiS provides its own namespace `AMDiS` to avoid potential naming conflicts with other libraries.

Now, the functions f and g will be defined by the classes `F` and `G`:

```cpp
// ===== function definitions =====
class G : public AbstractFunction<double, WorldVector<double> >
{
public:
  MEMORY_MANAGED(G);

  const double& operator()(const WorldVector<double>& x) const
  {
    static double result;
    result = exp(-10.0*(x*x));
    return result;
  };
};
```

`G` is a sub class of the templated class `AbstractFunction<R, T>` that represents a mapping from type `T` to type `R`. Here, we want to define a mapping from $\mathbb{R}^{dow}$, implemented by the class `WorldVector<double>`, to $\mathbb{R}$, represented by the data type `double`. The actual mapping is defined by overloading the `operator()`. For efficiency reasons not a copy of the result is returned but a reference to a static local variable. `x*x` stands for the scalar product of vector `x` with itself.

Using the macro call `MEMORY_MANAGED(G)`, the class will be managed by the memory management of AMDiS. This memory management provides memory monitoring to locate memory leaks and block memory allocation to accelerate memory access. Objects of a memory managed class can now be allocated through the memory manager by the macro `NEW` and deallocated by the macro `DELETE`.

The class `F` is defined in a similar way:

```cpp
class F : public AbstractFunction<double, WorldVector<double> >
{
public:
  MEMORY_MANAGED(F);

  F(int degree) : AbstractFunction<double, WorldVector<double> >(degree) {};

  const double& operator()(const WorldVector<double>& x) const {
    static double result = 0.0;
    int dow = Global::getGeo(WORLD);
    double r2 = (x*x);
    double ux = exp(-10.0*r2);
    result = -(400.0*r2 - 20.0*dow)*ux;
    return result;
  };
};
```

`F` gets the world dimension from the class `Global` by a call of the static function `getGeo(WORLD)`. The degree handed to the constructor determines the polynomial degree with which the function should be considered in the numerical integration. A higher degree leads to a quadrature of higher order in the assembling process.

Now, we start with the main program:

```cpp
// ===== main program =====
int main(int argc, char* argv[])
{
  FUNCNAME("main");
```

```
// ===== check for init file =====
TEST_EXIT(argc == 2)("usage:␣ellipt␣initfile\n");

// ===== init parameters =====
Parameters::init(true, argv[1]);
```

The macro `FUNCNAME` defines the current function name that is used for command line output, e.g. in error messages. The macro `TEST_EXIT` tests for the condition within the first pair of brackets. If the condition does not hold, an error message given in the second bracket pair is prompted and the program exits. Here the macro is used to check, whether the parameter file was specified by the user as command line argument. If this is the case, the parameters are initialized by `Parameters::init(true, argv[1])`. The first argument specifies, whether the initialized parameters should be printed after initialization for debug reasons. The second argument is the name of the parameter file.

Now, a scalar problem with name `ellipt` is created and initialized:

```
// ===== create and init the scalar problem =====
ProblemScal ellipt("ellipt");
ellipt.initialize(INIT_ALL);
```

The name argument of the problem is used to identify parameters in the parameter file that belong to this problem. In this case, all parameters with prefix `ellipt->` are associated to this problem. The initialization argument `INIT_ALL` means that all problem modules are created in a standard way. Those are: The finite element space including the corresponding mesh, needed system matrices and vectors, an iterative solver, an estimator, a marker, and a file writer for the computed solution. The initialization of these components can be controlled through the parameter file (see Section A.4.1).

The next steps are the creation of the adaptation loop and the corresponding `AdaptInfo`:

```
// === create adapt info ===
AdaptInfo *adaptInfo = NEW AdaptInfo("ellipt->adapt", 1);

// === create adapt ===
AdaptStationary *adapt = NEW AdaptStationary("ellipt->adapt", &ellipt,
                                             adaptInfo);
```

The `AdaptInfo` object contains information about the current state of the adaptation loop as well as user given parameters concerning the adaptation loop, like desired tolerances or maximal iteration numbers. Using `adaptInfo`, the adaptation loop can be inspected and controlled at runtime. Now, a stationary adaptation loop is created, which implements the standard *assemble-solve-estimate-adapt* loop. Arguments are the name, again used as parameter prefix, the problem as implementation of an iteration interface, and the `AdaptInfo` object. The adaptation loop only knows when to perform which part of an iteration. The implementation and execution of the single steps is delegated to an iteration interface, here implemented by the scalar problem `ellipt`.

Now, we define boundary conditions:

```
// ===== add boundary conditions =====
ellipt.addDirichletBC(1, NEW G);
```

We have one Dirichlet boundary condition associated with identifier 1. All nodes belonging to this boundary are set to the value of function `G` at the corresponding coordinates. In the macro file (see Section A.4.1) the Dirichlet boundary is marked with identifier 1, too. So the nodes can be uniquely determined.

The operators now are defined as follows:

```
// ===== create matrix operator =====
```

```
Operator matrixOperator(Operator::MATRIX_OPERATOR, ellipt.getFESpace());
matrixOperator.addSecondOrderTerm(NEW Laplace_SOT);
ellipt.addMatrixOperator(&matrixOperator);

// ===== create rhs operator =====
int degree = ellipt.getFESpace()->getBasisFcts()->getDegree();
Operator rhsOperator(Operator::VECTOR_OPERATOR, ellipt.getFESpace());
rhsOperator.addZeroOrderTerm(NEW CoordsAtQP_ZOT(NEW F(degree)));
ellipt.addVectorOperator(&rhsOperator);
```

First, we define a matrix operator (left hand side operator) on the finite element space of the problem. Now, we add the term $-\Delta u$ to it. Note that the minus sign isn't explicitly given, but implicitly contained in `Laplace_SOT`. With `addMatrixOperator` we add the operator to the problem. The definition of the right hand side is done in a similar way. We choose the degree of our function to be equal to the current basis function degree.

Finally we start the adaptation loop and afterwards write out the results:

```
// ===== start adaption loop =====
adapt->adapt();

// ===== write result =====
ellipt.writeFiles(adaptInfo, true);
}
```

The second argument of `writeFiles` forces the file writer to print out the results. In time dependent problems it can be useful to write the results only every i-th timestep. To allow this behavior the second argument has to be `false`.

Parameter file

The name of the parameter file must be given as command line argument. In the 2d case we call:

```
> ./ellipt init/ellipt.dat.2d
```

In the following, the content of file `init/ellipt.dat.2d` is described:

```
dimension of world:              2

elliptMesh->macro file name:     ./macro/macro.stand.2d
elliptMesh->global refinements:  0
```

The dimension of the world is 2, the macro mesh with name `elliptMesh` is defined in file `./macro/macro.stand.2d` (see Section A.4.1). The mesh is not globally refined before the adaptation loop. A value of n for `elliptMesh->global refinements` means n bisections of every macro element. Global refinements before the adaptation loop can be useful to save computation time by starting adaptive computations with a finer mesh.

```
ellipt->mesh:                    elliptMesh
ellipt->dim:                     2
ellipt->polynomial degree:       3
```

Now, we construct the finite element space for the problem `ellipt` (see Section A.4.1). We use the mesh `elliptMesh`, set the problem dimension to 2, and choose Lagrange basis functions of degree 3.

```
ellipt->solver:                  cg    % no bicgstab cg gmres odir ores
ellipt->solver->max iteration:   1000
ellipt->solver->tolerance:       1.e-8
ellipt->solver->left precon:     diag  % no, diag
```
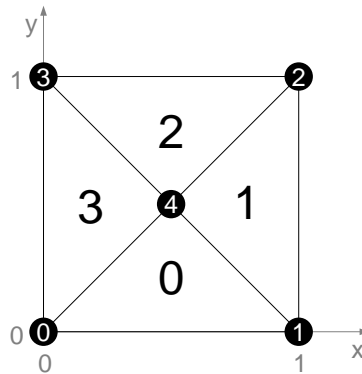
Figure A.2: Two dimensional macro mesh

We use the *conjugate gradient method* as iterative solver. The solving process stops after maximal 1000 iterations or when a tolerance of 10^{-8} is reached. Furthermore, we apply diagonal pre-conditioning.

```
ellipt ->estimator :                   residual % residual , recovery
ellipt ->estimator ->error norm :      1
ellipt ->estimator ->C0 :              0.1
ellipt ->estimator ->C1 :              0.1
```

As error estimator we use the residual method. The used error norm is the H1-norm (instead of the L2-norm: 2). Element residuals (C0) and jump residuals (C1) both are weighted by factor 0.1.

```
ellipt ->marker ->strategy :           2      % 0:  no  1:  GR  2:  MS  3:  ES  4:GERS
ellipt ->marker ->MSGamma :            0.5
```

After error estimation, elements are marked for refinement and coarsening. Here, we use the maximum strategy with $\gamma = 0.5$.

```
ellipt ->adapt ->tolerance :           1e-4
ellipt ->adapt ->max iteration :       100
ellipt ->adapt ->refine bisections :   2
```

The adaptation loop stops, when an error tolerance of 10^{-4} is reached, or after maximal 100 iterations. An element that is marked for refinement, is bisected twice within one iteration. Analog elements that are marked for coarsening are coarsened twice per iteration.

```
ellipt ->output ->filename :           output/ellipt
ellipt ->output ->AMDiS  format :      1
ellipt ->output ->AMDiS  mesh ext :    .mesh
ellipt ->output ->AMDiS  data ext :    .dat
```

The result is written in AMDiS-format to the files `output/ellipt.mesh` and `output/ellipt.dat`. The first contains the final mesh, the second contains the corresponding solution values.

Macro file

In Figure A.2 one can see the macro mesh which is described by the file `macro/macro.stand.2d`. First, the dimension of the mesh and of the world are defined:

```
DIM: 2
DIM_OF_WORLD: 2
```

Then the total number of elements and vertices are given:

```
number of elements: 4
number of vertices: 5
```

The next block describes the two dimensional coordinates of the five vertices:

```
vertex coordinates:
 0.0 0.0
 1.0 0.0
 1.0 1.0
 0.0 1.0
 0.5 0.5
```

The first two numbers are interpreted as the coordinates of vertex 0, and so on.
Corresponding to these vertex indices now the four triangles are given:

```
element vertices:
0 1 4
1 2 4
2 3 4
3 0 4
```

Element 0 consists in the vertices 0, 1 and 4. The numbering is done anticlockwise starting with the vertices of the longest edge.

It follows the definition of boundary conditions:

```
element boundaries:
0 0 1
0 0 1
0 0 1
0 0 1
```

The first number line means that element 0 has no boundaries at edge 0 and 1, and a boundary with identifier 1 at edge 2. The edge with number i is the edge opposite to vertex number i. The boundary identifier 1 corresponds to the identifier 1 we defined within the source code for the Dirichlet boundary. Since all elements of the macro mesh have a Dirichlet boundary at edge 2, the line `0 0 1` is repeated three times.

The next block defines element neighborships. `-1` means there is no neighbor at the corresponding edge. A non-negative number determines the index of the neighbor element.

```
element neighbours:
1 3 -1
2 0 -1
3 1 -1
0 2 -1
```

This block is optional. If it isn't given in the macro file, element neighborships are computed automatically.

Output

Now, the program is started by the call `./ellipt init/ellipt.dat.2d`. After 9 iterations the tolerance is reached and the files `output/ellipt.mesh` and `output/ellipt.dat` are written. In Figure A.3(a) the solution is shown and in A.3(b) the corresponding mesh. The visualizations was done by the VTK based tool **CrystalClear**.

(a)   (b)
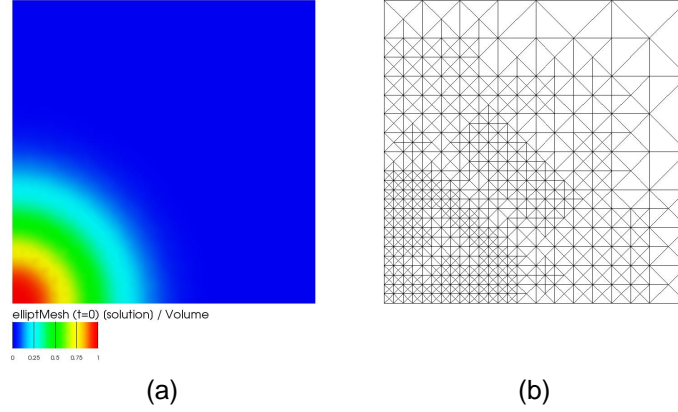
Figure A.3: (a): Solution after 9 iterations, (b): corresponding mesh

A.4.2   Time dependent problem

This is an example for a time dependent scalar problem. The problem is described by the heat equation

$$\partial_t u - \Delta u \;=\; f \qquad \text{in } \Omega \subset \mathbb{R}^{dim} \times \left(t^{begin}, t^{end}\right) \tag{A.5}$$

$$u \;=\; g \qquad \text{on } \partial\Omega \times \left(t^{begin}, t^{end}\right) \tag{A.6}$$

$$u \;=\; u_0 \qquad \text{on } \Omega \times \left(t^{begin}\right). \tag{A.7}$$

We solve the problem in the time interval $\left(t^{begin}, t^{end}\right)$ with Dirichlet boundary conditions on $\partial\Omega$. The problem is constructed, such that the exact solution is $u(x,t) = \sin(\pi t)e^{-10x^2}$. So we set

$$f(x,t) \;=\; \pi\cos(\pi t)e^{-10x^2} - \left(400x^2 - 20dow\right)\sin(\pi t)e^{-10x^2} \tag{A.8}$$

$$g(x,t) \;=\; \sin(\pi t)e^{-10x^2} \tag{A.9}$$

$$u_0(x) \;=\; \sin(\pi t^{begin})e^{-10x^2}. \tag{A.10}$$

We use a variable time discretization scheme. Equation (A.5) is approximated by

$$\frac{u^{new} - u^{old}}{\tau} - \left(\theta\Delta u^{new} + (1-\theta)\Delta u^{old}\right) = f(\cdot, t^{old} + \theta\tau). \tag{A.11}$$

$\tau = t^{new} - t^{old}$ is the timestep size between the old and the new problem time. u^{new} is the (searched) solution at $t = t^{new}$. u^{old} is the solution at $t = t^{old}$, which is already known from the last timestep. The parameter θ determines the implicit and explicit treatment of Δu. For $\theta = 0$ we have the forward explicit Euler scheme, for $\theta = 1$ the backward implicit Euler scheme. $\theta = 0.5$ results in the Crank-Nicholson scheme. If we bring all terms that depend on u^{old} to the right hand side, the equation reads

$$\frac{u^{new}}{\tau} - \theta\Delta u^{new} = \frac{u^{old}}{\tau} + (1-\theta)\Delta u^{old} + f(\cdot, t^{old} + \theta\tau). \tag{A.12}$$

Source code

Now, we describe the crucial parts of the source code. First, the functions f and g are defined. In contrast to the ellipt example, the functions now are time dependent. This is implemented by deriving the function classes also from class `TimedObject`. This class provides a pointer to the

		1d	**2d**	**3d**
source code	`src/`	`heat.cc`		
parameter file	`init/`	`heat.dat.1d`	`heat.dat.2d`	`heat.dat.3d`
macro file	`macro/`	`macro.stand.1d`	`macro.stand.2d`	`macro.stand.3d`
output files	`output/`	`heat_<t>.mesh, heat_<t>.dat`		

Table A.2: Files of the `heat` example. In the output file names, `<t>` is replaced by the time.
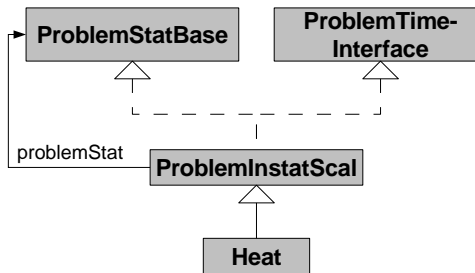
Figure A.4: UML diagram for class Heat.

current time, as well as corresponding setting and getting methods. The usage of a pointer to a real value allows to manage the current time in one location. All objects that deal with the same time, point to the same value. In our example, f is evaluated at $t = t^{old} + \theta\tau$, while g (the Dirichlet boundary function for u^{new}) is evaluated at $t = t^{new}$. Function g is implemented as follows:

```
class G : public AbstractFunction<double, WorldVector<double> >,
          public TimedObject
{
public:
  MEMORY_MANAGED(G);

  const double& operator()(const WorldVector<double>& x) const
  {
    static double result;
    result = sin(M_PI*(*timePtr)) * exp(-10.0*(x*x));
    return result;
  };
};
```

The variable $timePtr$ is a base class member of `TimedObject`. This pointer has to be set once before g is evaluated the first time. Implementation of function f is done in the same way.

Now, we begin with the implementation of class `Heat`, that represents the instationary problem. In Figure A.4, its class diagram is shown. `Heat` is derived from class `ProblemInstatScal` which leads to following properties:

`Heat` implements the `ProblemTimeInterface`, so the adaptation loop can set the current time and schedule timesteps.

`Heat` implements `ProblemStatBase` in the role as initial (stationary) problem. The adaptation loop can compute the initial solution through this interface. The single iteration steps can be overloaded by sub classes of `ProblemInstatScal`. Actually, the initial solution is computed through the method `solveInitialProblem` of `ProblemTimeInterface`. But this method is implemented by `ProblemInstatScal` interpreting itself as initial stationary problem.

`Heat` knows another implementation of `ProblemStatBase`: This other implementation represents
a stationary problem which is solved within each timestep.

The first lines of class `Heat` are:

```
class Heat : public ProblemInstatScal
{
public:
  MEMORY_MANAGED(Heat);

  Heat(ProblemScal *heatSpace)
    : ProblemInstatScal("heat", heatSpace)
  {
    theta = -1.0;
    GET_PARAMETER(0, name + "->theta", "%f", &theta);
    TEST_EXIT(theta >= 0)("theta␣not␣set!\n");
    theta1 = theta - 1;
  };
```

The argument `heatSpace` is a pointer to the stationary problem which is solved each timestep.
It is directly handed to the base class constructor of `ProblemInstatScal`. In the body of the
constructor, θ is read from the parameter file and stored in a member variable. The member
variable `theta1` stores the value of $\theta - 1$. A pointer to this value is used later as factor in the
θ-scheme.

The next lines show the implementation of the time interface.

```
  void setTime(AdaptInfo *adaptInfo) {
    rhsTime =
      adaptInfo->getTime()-
      (1-theta)*adaptInfo->getTimestep();
    boundaryTime = adaptInfo->getTime();
    tau1 = 1.0 / adaptInfo->getTimestep();
  };

  void closeTimestep(AdaptInfo *adaptInfo) {
    ProblemInstatScal::closeTimestep(adaptInfo);
    WAIT;
  };
```

The method `setTime` is called by the adaptation loop to inform the problem about the current
time. The right hand side function f will be evaluated at $t^{old} + \theta\tau = t^{new} - (1 - \theta)\tau$, the Dirichlet
boundary function g at t^{new}. t^{new} is the current time, τ is the current timestep, both set by the
adaptation loop and stored in `adaptInfo`. `tau1` stores the value of $\frac{1}{\tau}$, which is used later as factor
for the zero order time discretization terms.

The method `closeTimestep` is called at the end of each timestep by the adaptation loop. In the
default implementation of `ProblemInstatScal::closeTimestep`, the solution is written to output
files, if specified in the parameter file. Note that the base class implementation of a method must
be explicitly called, if the method is overwritten in a sub class. The macro `WAIT` waits until the
`return` key is pressed by the user, if the corresponding entry in the parameter file is set to 1. The
macro `WAIT_REALLY` would wait, independent of parameter settings. If `closeTimestep` wouldn't
be overloaded here, the default implementation without the `WAIT` statement would be called after
each timestep.

Now, the implementation of the `ProblemStatBase` interface begins. As mentioned above, the
instationary problem plays the role of the initial problem by implementing this interface.

```
  void solve(AdaptInfo *adaptInfo)
  {
    problemStat->getSolution()->interpol(exactSolution);
```

```
  };

  void estimate(AdaptInfo *adaptInfo)
  {
    double errMax, errSum;
    errSum = Error<double>::L2Err(*exactSolution,
                                  *(problemStat->getSolution()),
                                  0, &errMax, false);
    adaptInfo->setEstSum(errSum, 0);
    adaptInfo->setEstMax(errMax, 0);
  };
```

Here, only the solve and the estimate step are overloaded. For the other steps, there are empty default implementations in `ProblemInstatScal`. Since the mesh is not adapted in the initial problem, the initial adaptation loop will stop after one iteration. In the solve step, the exact solution is interpolated on the macro mesh and stored in the solution vector of the stationary problem. In the estimate step, the L2 error is computed. The maximal element error and the sum over all element errors are stored in `adaptInfo`. To make the exact solution known to the problem, we need a setting function:

```
  void setExactSolution(AbstractFunction<double, WorldVector<double> > *fct)
  {
    exactSolution = fct;
  }
```

Now, we define some getting functions and the private member variables:

```
  double *getThetaPtr() { return &theta; };
  double *getTheta1Ptr() { return &theta1; };
  double *getTau1Ptr() { return &tau1; };
  double *getRHSTimePtr() { return &rhsTime; };
  double *getBoundaryTimePtr() { return &boundaryTime; };

private:
  double theta;
  double theta1;
  double tau1;
  double rhsTime;
  double boundaryTime;
  AbstractFunction<double, WorldVector<double> > *exactSolution;
};
```

The definition of class `Heat` is now finished. In the following, the main program is described.

```
int main(int argc, char** argv)
{
  // ===== check for init file =====
  TEST_EXIT(argc == 2)("usage:␣heat␣initfile\n");

  // ===== init parameters =====
  Parameters::init(false, argv[1]);

  // ===== create and init stationary problem =====
  ProblemScal *heatSpace = NEW ProblemScal("heat->space");
  heatSpace->initialize(INIT_ALL);

  // ===== create instationary problem =====
  Heat *heat = new Heat(heatSpace);
  heat->initialize(INIT_ALL);
```

So far, the stationary space problem `heatSpace` and the instationary problem `heat` were created and initialized. `heatSpace` is an instance of `ProblemScal`. `heat` is an instance of the class `Heat` we defined above. `heatSpace` is given to `heat` as its stationary problem.

The next step is the creation of the needed `AdaptInfo` objects and of the instationary adaptation loop:

```
// create adapt info for heat
AdaptInfo *adaptInfo = NEW AdaptInfo("heat->adapt");

// create initial adapt info
AdaptInfo *adaptInfoInitial = NEW AdaptInfo("heat->initial->adapt");

// create instationary adapt
AdaptInstationary *adaptInstat = NEW AdaptInstationary("heat->adapt",
                                                        heatSpace,
                                                        adaptInfo,
                                                        heat,
                                                        adaptInfoInitial);
```

The object `heatSpace` is handed as `ProblemIterationInterface` (implemented by class `ProblemScal`) to the adaptation loop. `heat` is interpreted as `ProblemTimeInterface` (implemented by class `ProblemInstatScal`).

The definitions of functions f and g are:

```
// ===== create boundary functions =====
G *boundaryFct = NEW G;
boundaryFct->setTimePtr(heat->getBoundaryTimePtr());
heat->setExactSolution(boundaryFct);

heatSpace->addDirichletBC(1, boundaryFct);

// ===== create rhs functions =====
int degree = heatSpace->getFESpace()->getBasisFcts()->getDegree();
F *rhsFct = NEW F(degree);
rhsFct->setTimePtr(heat->getRHSTimePtr());
```

The functions interpreted as `TimedObject`s are linked with the corresponding time pointers by `setTimePtr`. The boundary function is handed to `heat` as exact solution and as Dirichlet boundary function with identifier 1 to `heatSpace`.

Now, we define the operators:

```
// ===== create operators =====
double one = 1.0;
double zero = 0.0;

// create laplace
Operator *A = NEW Operator(Operator::MATRIX_OPERATOR |
                           Operator::VECTOR_OPERATOR,
                           heatSpace->getFESpace());

A->addSecondOrderTerm(new Laplace_SOT);

A->setUhOld(heat->getOldSolution());

if((*(heat->getThetaPtr())) != 0.0)
  heatSpace->addMatrixOperator(A, heat->getThetaPtr(), &one);

if((*(heat->getTheta1Ptr())) != 0.0)
  heatSpace->addVectorOperator(A, heat->getTheta1Ptr(), &zero);
```

Operator `A` represents $-\Delta u$. It is used as matrix operator on the left hand side with factor θ and as vector operator on the right hand side with factor $-(1-\theta) = \theta - 1$. These assemble factors are the second arguments of `addMatrixOperator` and `addVectorOperator`. The third argument is the factor used for estimation. In this example, the estimator will consider the operator only on the left hand side with factor 1. On the right hand side the operator is applied to the solution of the last timestep. So the old solution is handed to the operator by `setUhOld`.

```
// create zero order operator
Operator *C = NEW Operator(Operator::MATRIX_OPERATOR |
                           Operator::VECTOR_OPERATOR,
                           heatSpace->getFESpace());

C->addZeroOrderTerm(NEW Simple_ZOT);

C->setUhOld(heat->getOldSolution());

heatSpace->addMatrixOperator(C, heat->getTau1Ptr(), heat->getTau1Ptr());
heatSpace->addVectorOperator(C, heat->getTau1Ptr(), heat->getTau1Ptr());
```

The `Simple_ZOT` of operator `C` represents the zero order terms for the time discretization. On both sides of the equation u is added with $\frac{1}{\tau}$ as assemble factor and as estimate factor.

Finally, the operator for the right hand side function f is added and the adaptation loop is started:

```
// create RHS operator
Operator *F = NEW Operator(Operator::VECTOR_OPERATOR,
                           heatSpace->getFESpace());

F->addZeroOrderTerm(NEW CoordsAtQP_ZOT(rhsFct));

heatSpace->addVectorOperator(F);

// ===== start adaption loop =====
adaptInstat->adapt();
}
```

`CoordsAtQP_ZOT` is a zero order term that evaluates a given function fct at all needed quadrature points. At the left hand side, it would represent the term $fct(x, t) \cdot u$, on the right hand side, just $fct(x, t)$. Note that the old solution isn't given to the operator here. Otherwise the term would represent $fct(x, t) \cdot u^{old}$ on the right hand side.

Parameter file

In this section, we show only the relevant parts of the parameter file `heat.dat.2d`.

First the parameter θ for the time discretization is defined:

```
heat->theta:                          1.0
```

Then we define the initial timestep and the time interval:

```
heat->adapt->timestep:                0.1
heat->adapt->start time:              0.0
heat->adapt->end time:                1.0
```

Now, tolerances are determined:

```
heat->adapt->tolerance:               0.01
heat->adapt->rel space error:         0.5
heat->adapt->rel time error:          0.5
```

```
heat ->adapt ->time theta 1:          1.0
heat ->adapt ->time theta 2:          0.3
```

The total tolerance is divided in a space tolerance and a time tolerance. The space tolerance is the maximal allowed space error, given by the product of `tolerance` and `rel space error`. It is reached by adaptive mesh refinements. The time tolerance is the maximal allowed error, due to the timestep size. It is given by the product of `tolerance` and `rel time error` and `time theta 1`. It is relevant, only if an implicit time strategy with adaptive timestep size is used. The parameter `time theta 2` is used to enlarge the timestep, if the estimated time error falls beneath a given threshold.

```
heat ->adapt ->strategy :             1
heat ->adapt ->time delta 1:          0.7071
heat ->adapt ->time delta 2:          1.4142
```

If `strategy` is 0, an explicit time strategy with fixed timestep size is used. A value of 1 stands for the implicit strategy. The time tolerance is reached by successively multiplying the timestep with `time delta 1`. If the estimated timestep error is smaller than the product of `tolerance` and `rel time error` and `time theta 2` at the end of a timestep, the timestep size is multiplied by `time delta 2`.

The following lines determine, whether coarsening is allowed in regions with sufficient small errors, and how many refinements or coarsenings are performed for marked elements.

```
heat ->adapt ->coarsen allowed :      1
heat ->adapt ->refine bisections :    2
heat ->adapt ->coarsen bisections :   2
```

Now, the output behavior is determined:

```
heat ->space ->output ->filename :        output/heat

heat ->space ->output ->AMDiS format :    1
heat ->space ->output ->AMDiS mesh ext: .mesh
heat ->space ->output ->AMDiS data ext: .dat

heat ->space ->output ->write every i-th timestep: 10

heat ->space ->output ->append index :    1
heat ->space ->output ->index length :    6
heat ->space ->output ->index decimals : 3
```

In this example, all output filenames start with prefix `output/heat` and end with the extensions `.mesh` and `.dat`. Output is written after every 10th timestep. The time of the single solution is added after the filename prefix with 6 letters, three of them are decimals. The solution for $t = 0$ e.g. would be written in the files `output/heat00.000.mesh` and `output/heat00.000.dat`.

Finally, we set parameter `WAIT` to 1. So each call of the macro `WAIT` in the application will lead to an interruption of the program, until the `return` key is pressed.

```
WAIT :                                    1
```

Macro file

We again use the macro file `macro/macro.stand.2d`, which was described in Section A.4.1.

| | $t \approx 0.2$ | $t \approx 0.47$ | $t \approx 0.81$ |
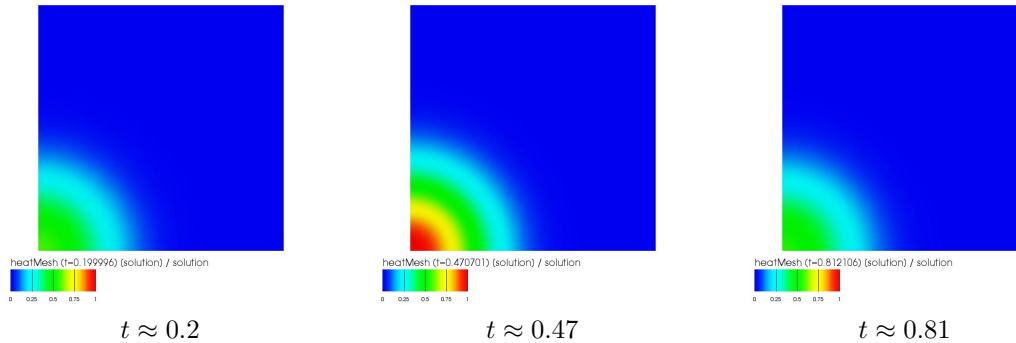
Figure A.5: The solution at three different timesteps.

		1d	**2d**	**3d**
source code	`src/`	`vecellipt.cc`		
parameter file	`init/`	`vecellipt.dat.1d`	`vecellipt.dat.2d`	`vecellipt.dat.3d`
macro file	`macro/`	`macro.stand.1d`	`macro.stand.2d`	`macro.stand.3d`
output files	`output/`	`vecellipt_comp<c>.mesh, vecellipt_comp<c>.dat`		

Table A.3: Files of the `vecellipt` example. In the output file names, `<c>` is replaced by the component number.

Output

As mentioned above, the output files look like `output/heat00.000.mesh` and `output/heat00.000.dat`. Depending on the corresponding value in the parameter file only the solution after every i-th timestep is written. In Figure A.5, the solution at three timesteps is visualized.

A.4.3 Systems of PDEs

In this example, we show how to implement a system of coupled PDEs. We define

$$- \Delta u = f \tag{A.13}$$
$$u - v = 0. \tag{A.14}$$

For the first equation, we use the boundary condition and definition of function f from Section A.4.1. The second equation defines a second solution component v, which is coupled to u, such that $v = u$. For the second equation, no boundary conditions have to be defined. The system can be written in matrix-vector form as

$$\begin{pmatrix} -\Delta & 0 \\ I & -I \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} f \\ 0 \end{pmatrix}, \tag{A.15}$$

where I stands for the identity and 0 for a *zero operator* (or for the absence of any operator). This is a very simple example without practical relevance. But it is appropriate to demonstrate the main principles of implementing vector valued problems.

Source code

Instead of a scalar problem, we now create and initialize the vector valued problem `vecellipt`:

```
ProblemVec vecellipt("vecellipt");
vecellipt.initialize(INIT_ALL);
```

The `AdaptInfo` constructor is called with the number of problem components, which is defined in the parameter file.

```
// === create adapt info ===
AdaptInfo *adaptInfo = NEW AdaptInfo("vecellipt->adapt",
                                     vecellipt.getNumComponents());

// === create adapt ===
AdaptStationary *adapt = NEW AdaptStationary("vecellipt->adapt",
                                             &vecellipt,
                                             adaptInfo);
```

The adaptation loop doesn't care about the component number. It treats `vecellipt` only as implementation of the iteration interface.

The Dirichlet boundary condition for the first equation is defined by

```
// ===== add boundary conditions =====
vecellipt.addDirichletBC(1, 0, NEW G);
```

The first argument is the condition identifier, as in the scalar case. The second argument is the component, the boundary condition belongs to.

The operator definitions for the first equation are:

```
// ===== create operators =====
Operator matrixOperator00(Operator::MATRIX_OPERATOR,
                          vecellipt.getFESpace(0),
                          vecellipt.getFESpace(0));
matrixOperator00.addSecondOrderTerm(NEW Laplace_SOT);
vecellipt.addMatrixOperator(&matrixOperator00, 0, 0);

Operator rhsOperator0(Operator::VECTOR_OPERATOR,
                      vecellipt.getFESpace(0));

int degree = vecellipt.getFESpace(0)->getBasisFcts()->getDegree();

rhsOperator0.addZeroOrderTerm(NEW CoordsAtQP_ZOT(NEW F(degree)));

vecellipt.addVectorOperator(&rhsOperator0, 0);
```

Operator `matrixOperator00` represents the $-\Delta$ operator. Each operator belongs to two finite element spaces, the *row space* and the *column space*. If an operator has the position (i, j) in the operator matrix, the row space is the finite element space of component i and the column space is the finite element space of component j. The finite element spaces can differ in the used basis function degree. The underlying meshes must be the same. After `matrixOperator00` is created, it is handed to the problems operator matrix at position $(0, 0)$. The right hand side operator `rhsOperator0` only needs a row space, which is the finite element space of component 0 (u). It is handed to the operator vector at position 0.

Now, the operators for the second equation are defined:

```
Operator matrixOperator10(Operator::MATRIX_OPERATOR,
                          vecellipt.getFESpace(1),
                          vecellipt.getFESpace(0));

Operator matrixOperator11(Operator::MATRIX_OPERATOR,
                          vecellipt.getFESpace(1),
                          vecellipt.getFESpace(1));

matrixOperator10.addZeroOrderTerm(NEW Simple_ZOT);
```

```
 vecellipt.addMatrixOperator(&matrixOperator10 , 1, 0);

 matrixOperator11.addZeroOrderTerm(NEW Simple_ZOT(-1.0));

 vecellipt.addMatrixOperator(&matrixOperator11 , 1, 1);
```

Note that the operator `matrixOperator10` can have different finite element spaces, if the spaces of the two components differ. The operators I and $-I$ are implemented by `Simple_ZOT`, once with a fixed factor of 1 and once with a factor of -1.

Parameter file

First, the number of components and the basis function degrees are given. We use Lagrange polynomials of degree 1 for the first component and of degree 2 for the second component.

```
vecellipt ->components :                2

vecellipt ->polynomial degree [0]:    1
vecellipt ->polynomial degree [1]:    2
```

In general, the linear system of equations for systems of PDEs is not symmetric. So with the GMRes solver, we use a solver that doesn't assume symmetric matrices.

```
vecellipt ->solver :                    gmres
```

Note that we have only one solver, because the equations of our system are assembled in one linear system of equations.

Each equation can have its own estimator. In this case, adaptivity should be managed only by the first component. So the second equation has no estimator.

```
vecellipt ->estimator [0]:              residual
vecellipt ->estimator [1]:              no
```

Also the marking strategy can differ between the components. Refinement is done, if at least one component has marked an element for refinement. Coarsening only is done, if all components have marked the element for coarsening. In our example, only component 0 will mark elements.

```
vecellipt ->marker [0]->strategy :    2
vecellipt ->marker [1]->strategy :    0
```

We have only one adaptation loop, which does maximal 6 iterations. The tolerance can be determined for each component. The total tolerance criterion is fulfilled, if all criteria of all components are fulfilled.

```
vecellipt ->adapt ->max iteration :    6

vecellipt ->adapt [0]->tolerance :      1e-4
vecellipt ->adapt [1]->tolerance :      1e-4
```

Also the output can be controlled for each component individually:

```
vecellipt ->output [0]->filename :            output/vecellipt_comp0

vecellipt ->output [0]->AMDiS format :      1
vecellipt ->output [0]->AMDiS mesh ext :    .mesh
vecellipt ->output [0]->AMDiS data ext :    .dat

vecellipt ->output [1]->filename :            output/vecellipt_comp1
```

component 0   component 1
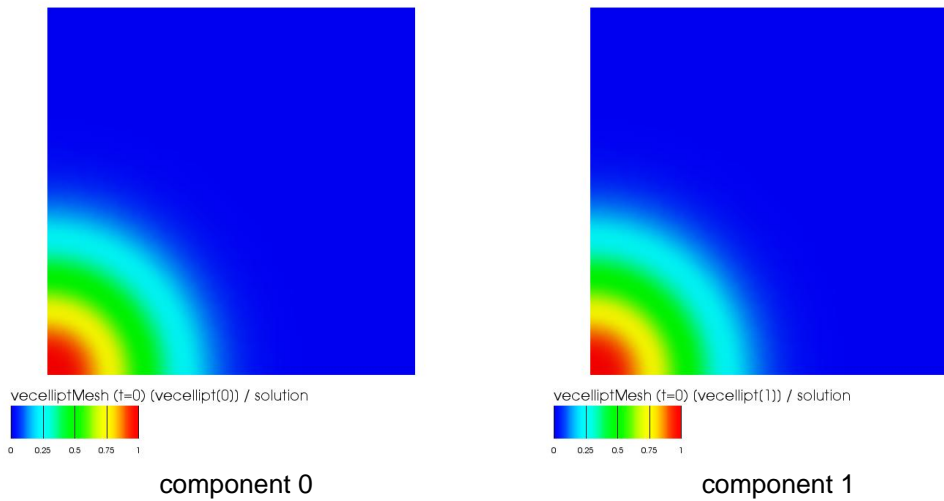
Figure A.6: The two solution components for u and v.

```
vecellipt -> output [1] -> AMDiS  format :     1
vecellipt -> output [1] -> AMDiS  mesh  ext :    . mesh
vecellipt -> output [1] -> AMDiS  data  ext :    . dat
```

Macro file

We again use the macro file `macro/macro.stand.2d`, which was described in Section A.4.1.

Output

Component 0 of the solution (approximation of u) is written to the files `output/vecellipt0.mesh` and `output/vecellipt0.dat`. Component 1 of the solution (approximation of v) is written to the files `output/vecellipt1.mesh` and `output/vecellipt1.dat`. The two components are visualized in Figure A.6.

A.4.4 Coupled problems

In this example, we solve the same problem as in Section A.4.3, but here we treat the two equations as two coupled problems. The main difference is that the equations now aren't assembled into the same large system of equations, but into two separated systems of equations, that have to be solved separately. We define the two problems

$$-\Delta u = f \quad \text{in } \Omega \subset \mathbb{R}^{dim} \tag{A.16}$$
$$u = g \quad \text{on } \partial\Omega \tag{A.17}$$

and

$$v = u. \tag{A.18}$$

We first solve the first problem and then use its solution to solve the second problem. This happens in every iteration of the adaptation loop. Both problems should use the same mesh. Mesh adaptation is done by the first problem. So one iteration now looks like illustrated in Figure A.7.
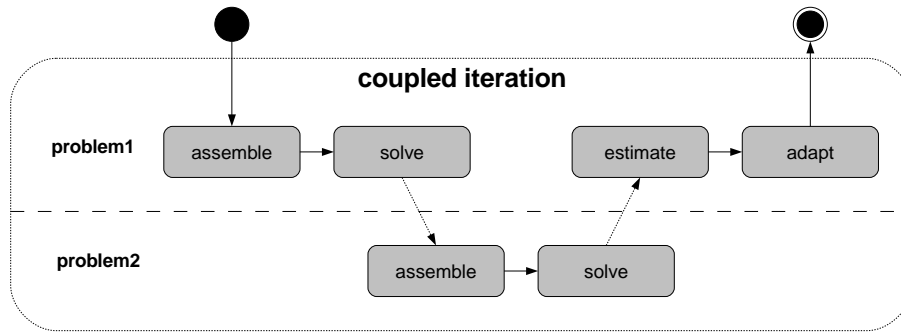
Figure A.7: State diagram of the coupled iteration.

		1d	2d	3d
source code	src/	couple.cc		
parameter file	init/	couple.dat.1d	couple.dat.2d	couple.dat.3d
macro file	macro/	macro.stand.1d	macro.stand.2d	macro.stand.3d
output files	output/	couple.mesh, couple.dat		

Table A.4: Files of the `couple` example.

Source code

In the previous examples, the iteration was implemented always by the corresponding problem class. In this example, we want to couple two problems within one iteration, so the default implementation can't be used. For that reason, we define our own coupled iteration class `MyCoupledIteration` which implements the `ProblemIterationInterface`:

```
class MyCoupledIteration : public ProblemIterationInterface
{
public:
  MyCoupledIteration(ProblemStatBase *prob1,
                     ProblemStatBase *prob2)
    : problem1(prob1),
      problem2(prob2)
  {};
```

In the constructor pointers to the two problems are assigned to the private members `problem1` and `problem2`. Note that the pointers point to the interface `ProblemStatBase` and not to `ProblemScal`. This leads to a more general implementation. If e.g. two vector valued problems should be coupled in the future, we could use our iteration class without modifications.

Now, we implement the needed interface methods:

```
void beginIteration(AdaptInfo *adaptInfo)
{
  FUNCNAME("StandardProblemIteration::beginIteration()");
  MSG("\n");
  MSG("begin␣of␣iteration␣%d\n", adaptInfo->getSpaceIteration()+1);
  MSG("============================\n");
};

void endIteration(AdaptInfo *adaptInfo) {
  FUNCNAME("StandardProblemIteration::endIteration()");
```

```
  MSG("\n");
  MSG("end␣of␣iteration␣%d\n", adaptInfo->getSpaceIteration()+1);
  MSG("============================\n");
};
```

These two functions are called at the beginning and at the end of each iteration. Here, we only prompt some output.

The method `oneIteration` is the crucial part of our implementation:

```
Flag oneIteration(AdaptInfo *adaptInfo, Flag toDo = FULL_ITERATION)
{
  Flag flag, markFlag;
  if(toDo.isSet(MARK)) markFlag = problem1->markElements(adaptInfo);
  if(toDo.isSet(ADAPT) && markFlag.isSet(MESH_REFINED)) {
    flag = problem1->refineMesh(adaptInfo);
  }
  if(toDo.isSet(BUILD)) problem1->buildAfterCoarsen(adaptInfo, markFlag);
  if(toDo.isSet(SOLVE)) problem1->solve(adaptInfo);

  if(toDo.isSet(BUILD)) problem2->buildAfterCoarsen(adaptInfo, markFlag);
  if(toDo.isSet(SOLVE)) problem2->solve(adaptInfo);

  if(toDo.isSet(ESTIMATE)) problem1->estimate(adaptInfo);
  return flag;
};
```

The `toDo` flag is used by the adaptation loop to determine which parts of the iteration should be performed. The first iteration is always an iteration without mesh adaptation (see Figure A.8). So we start our iteration by marking and adapting the mesh. The mesh and its adaptation is managed by the first problem. So we call `markElements` and `refineMesh` of `problem1`. Note that no mesh coarsenings have to be performed in our example. Afterwards, `problem1` assembles its system of equations by `buildAfterCoarsen`. Assemblage and mesh adaptation are nested operations in AMDiS (`buildBeforeRefine`, `refineMesh`, `buildBeforeCoarsen`, `coarsenMesh`,`buildAfterCoarsen`). Here, we implement a simplified version.

After `problem1` has solved its system of equations, `problem2` can assemble and solve its equations system using the solution of the first problem as right hand side. In the method `oneIteration`, only the order of method calls is determined. The dependency to the solution of the first problem is created later when the operator for the right hand side of `problem2` is created.

After also the second problem computed its solution, `problem1` does the error estimation (remember: mesh adaptation is managed by `problem1`).

Now, the access to the coupled problems is implemented and the member variables are defined:

```
int getNumProblems()
{
  return 2;
};

ProblemStatBase *getProblem(int number = 0)
{
  FUNCNAME("CoupledIteration::getProblem()");
  if(number == 0) return problem1;
  if(number == 1) return problem2;
  ERROR_EXIT("invalid␣problem␣number\n");
  return NULL;
};
```
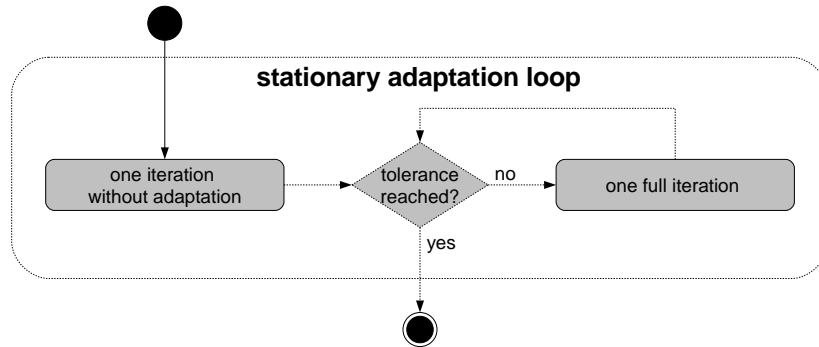
Figure A.8: Stationary adaptation loop.

```
private:
  ProblemStatBase *problem1;
  ProblemStatBase *problem2;
};
```

The class `MyCoupledIteration` is finished now.

The next class, `Identity`, implements the identity $I(x) = x$ for `double` variables. An arbitrary degree can be given to it. The class is used later to determine the quadrature degree used for the right hand side of `problem2`.

```
class Identity : public AbstractFunction<double, double>
{
public:
  MEMORY_MANAGED(Identity);

  Identity(int degree) : AbstractFunction<double, double>(degree) {};

  const double& operator()(const double& x) const {
    static double result;
    result = x;
    return result;
  };
};
```

Now, we start with the main program:

```
int main(int argc, char* argv[])
{
  FUNCNAME("main");
  TEST_EXIT(argc == 2)("usage:␣couple␣initfile\n");
  Parameters::init(true, argv[1]);

  // ===== create and init the first problem =====
  ProblemScal problem1("problem1");
  problem1.initialize(INIT_ALL);

  // ===== add boundary conditions for problem1 =====
  problem1.addDirichletBC(1, NEW G);
```

So far, we created and initialized `problem1` and its boundary conditions.

Now, we create `problem2`. It should have its own finite element space, system, solver and file writer, but the mesh should be adopted from `problem1`.

```
// ===== create and init the second problem =====
Flag initFlag =
  INIT_FE_SPACE |
  INIT_SYSTEM |
  INIT_SOLVER |
  INIT_FILEWRITER;

Flag adoptFlag =
  CREATE_MESH |
  INIT_MESH;

ProblemScal problem2("problem2");
problem2.initialize(initFlag,
                    &problem1,
                    adoptFlag);
```

The operators for the first problem are defined like in Section A.4.1. Here, we only show the operators of `problem2`.

```
// ===== create operators for problem2 =====
Operator matrixOperator2(Operator::MATRIX_OPERATOR,
                         problem2.getFESpace());
matrixOperator2.addZeroOrderTerm(NEW Simple_ZOT);
problem2.addMatrixOperator(&matrixOperator2);

Operator rhsOperator2(Operator::VECTOR_OPERATOR, problem2.getFESpace());
rhsOperator2.addZeroOrderTerm(NEW VecAtQP_ZOT(problem1.getSolution(),
                                             NEW Identity(degree)));
problem2.addVectorOperator(&rhsOperator2);
```

At the left hand side, we have just an ordinary `Simple_ZOT`. At the right hand side, we have a zero order term of the form $f(u)$ with $f = I$ the identity. u is given by the solution DOF vector of `problem1`. I maps the values of the DOF vector evaluated at quadrature points to itself. The function degree is used to determine the needed quadrature degree in the assembler.

Now, the adaptation loop is created:

```
// ===== create adaptation loop and iteration interface =====
AdaptInfo *adaptInfo = NEW AdaptInfo("couple->adapt", 1);

MyCoupledIteration coupledIteration(&problem1, &problem2);

AdaptStationary *adapt = NEW AdaptStationary("couple->adapt",
                                            &coupledIteration,
                                            adaptInfo);
```

Note that not a pointer to one of the problems is passed to the adaptation loop, but a pointer to the `coupledIteration` object, which in turn knows both problems.

The adaptation loop is now started. After it is finished, the solutions of both problems are written.

```
// ===== start adaptation loop =====
adapt->adapt();

// ===== write solution =====
problem1.writeFiles(adaptInfo, true);
problem2.writeFiles(adaptInfo, true);
```

```
}
```

Parameter file

We have one adaptation loop called `couple->adapt`:

```
couple ->adapt ->tolerance:          1e-8
couple ->adapt ->max iteration:       10
couple ->adapt ->refine bisections: 2
```

The coupled problem consits of two sub problems. The first problem creates the mesh, solves its linear system of equations, estimates the error, adapts the mesh, and finally writes its output:

```
coupleMesh ->macro file name:     ./macro/macro.stand.2d
coupleMesh ->global refinements: 0

problem1 ->mesh:                  coupleMesh
problem1 ->dim:                   2
problem1 ->polynomial degree:    1

problem1 ->solver:                cg % no, bicgstab, cg, gmres, odir, ores
problem1 ->solver ->max iteration: 1000
problem1 ->solver ->tolerance:    1.e-8
problem1 ->solver ->left precon:   diag

problem1 ->estimator:             residual
problem1 ->estimator ->C0:        0.1 % constant of element residual
problem1 ->estimator ->C1:        0.1 % constant of jump residual

problem1 ->marker ->strategy:     2 % 0: no 1: GR 2: MS 3: ES 4:GERS
problem1 ->marker ->MSGamma:      0.5

problem1 ->output ->filename:      output/problem1
problem1 ->output ->AMDiS format:  1
problem1 ->output ->AMDiS mesh ext: .mesh
problem1 ->output ->AMDiS data ext: .dat
```

The second problem uses the mesh of `problem1`. So it creates no mesh, no estimator, and no marker. But a solver is needed to solve `problem2`s linear system of equations, and a file writer to write the solution:

```
problem2 ->dim:                   2
problem2 ->polynomial degree:    1

problem2 ->solver:                cg % no, bicgstab, cg, gmres, odir, ores
problem2 ->solver ->max iteration: 1000
problem2 ->solver ->tolerance:    1.e-8
problem2 ->solver ->left precon:   diag

problem2 ->estimator:             no

problem2 ->marker ->strategy:     0

problem2 ->output ->filename:      output/problem2
problem2 ->output ->AMDiS format:  1
problem2 ->output ->AMDiS mesh ext: .mesh
problem2 ->output ->AMDiS data ext: .dat
```

Macro file

We again use the macro file `macro/macro.stand.2d`, which was described in Section A.4.1.

Output

The solution of the first problem is written to the files `output/problem1.mesh` and `output/problem1.dat`. The solution of the second problem is written to the files `output/problem2.mesh` and `output/problem2.dat`. We don't visualize the results here, because they conform with the results showed in Section A.4.3.

A.4.5　Nonlinear problem

We define the nonlinear problem

$$
\begin{aligned}
-\Delta u + u^4 &= f \quad \text{in } \Omega \subset \mathbb{R}^{dim} & \text{(A.19)} \\
u &= g \quad \text{on } \partial\Omega. & \text{(A.20)}
\end{aligned}
$$

We choose the functions f and g so that the exact solution again is $u(x) = e^{-10x^2}$. This leads to

$$
\begin{aligned}
f(x) &= -(400 - 20dow)\, e^{-10x^2} + \left(e^{-10x^2}\right)^4 & \text{(A.21)} \\
g(x) &= e^{-10x^2}, & \text{(A.22)}
\end{aligned}
$$

with dow the world dimension.

　　We linearize the problem using the Newton method. First, we define an initial guess u_0 of the solution which is 0 for the first adaptation loop iteration. In later iterations we can use the solution of the last iteration interpolated to the current mesh as initial guess. In each Newton step, a correction d for the solution of the last step is computed by solving

$$
DF(u_n)(d) = F(u_n) \tag{A.23}
$$

for d, where $F(u) := -\Delta u + u^4 - f$ and

$$
\begin{aligned}
DF(u_n)(d) &= \lim_{h \to 0} \frac{F(u_n + hd) - F(u_n)}{h} & \text{(A.24)} \\
&= \lim_{h \to 0} \frac{-\Delta u_n - h\Delta d + \Delta u_n}{h} + \lim_{h \to 0} \frac{(u_n + hd)^4 - u_n^4}{h} & \text{(A.25)} \\
&= -\Delta d + 4u_n^3 d & \text{(A.26)}
\end{aligned}
$$

the directional derivative of F at u_n along d.

　　Then the solution is updated:

$$
u_{n+1} := u_n - d. \tag{A.27}
$$

We repeat this procedure until $||d||_{L^2} < tol$ with tol a given tolerance for the Newton method.

　　In our example, equation (A.23) reads:

$$
-\Delta d + 4u_n^3 d = -\Delta u_n + u_n^4 - f. \tag{A.28}
$$

　　In Figure A.9, the Newton method is illustrated.
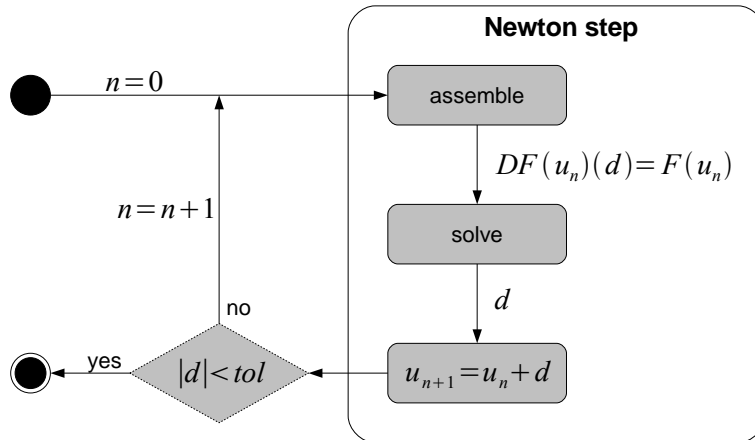
Figure A.9: Solve step of the nonlinear problem.

		1d	**2d**	**3d**
source code	`src/`	`nonlin.cc`		
parameter file	`init/`	`nonlin.dat.1d`	`nonlin.dat.2d`	`nonlin.dat.3d`
macro file	`macro/`	`macro_big.stand.1d`	`macro_big.stand.2d`	`macro_big.stand.3d`
output files	`output/`	`nonlin.mesh, nonlin.dat`		

Table A.5: Files of the `nonlin` example.

Source code

The main idea is to realize the Newton method as implementation of the *ProblemIterationInterface*, which replaces the standard iteration in the adaptation loop. The Newton method has to know the problem which has to be solved, as well as the implementation of one Newton step. Estimation and adaptation is done by the problem object. The assemble step and solve step are delegated to a Newton-step object.

Now, we describe the code step by step. The function g is defined like in the previous examples. In the following, we define a zero-function which is later used to implement the Dirichlet boundary condition for the Newton-step implementation (at domain boundaries no correction has to be done). The function `F` implements the right hand side function f.

```
class Zero : public AbstractFunction<double, WorldVector<double> >
{
public:
  MEMORY_MANAGED(Zero);

  const double& operator()(const WorldVector<double>& x) const {
    static double result = 0.0;
    return result;
  };
};

class F : public AbstractFunction<double, WorldVector<double> >
{
public:
  MEMORY_MANAGED(F);
```

```
  /** \brief
   * Constructor
   */
  F(int degree)
    : AbstractFunction<double, WorldVector<double> >(degree)
  {};

  /** \brief
   * Implementation of AbstractFunction::operator().
   */
  const double& operator()(const WorldVector<double>& x) const {
    static double result;
    int dow = x.getSize();
    double r2 = x*x, ux = exp(-10.0*r2), ux4 = ux*ux*ux*ux;
    result = ux4 -(400.0*r2 - 20.0*dow)*ux;
    return result;
  };
};
```

The class X3 implements the function $u^3(x)$ used within the Newton step.

```
class X3 : public AbstractFunction<double, double>
{
public:
  MEMORY_MANAGED(X3);

  X3() : AbstractFunction<double, double>(3) {};

  /** \brief
   * Implementation of AbstractFunction::operator().
   */
  const double& operator()(const double& x) const {
    static double result = 0.0;
    result = x * x * x;
    return result;
  };
};
```

In the following, we define an interface which has to be implemented by the Newton-step object.

```
class NewtonStepInterface
{
public:
  virtual void initNewtonStep(AdaptInfo *adaptInfo) = 0;
  virtual void exitNewtonStep(AdaptInfo *adaptInfo) = 0;
  virtual void assembleNewtonStep(AdaptInfo *adaptInfo, Flag flag) = 0;
  virtual void solveNewtonStep(AdaptInfo *adaptInfo) = 0;
  virtual DOFVector<double> *getCorrection() = 0;
};
```

The initNewtonStep method is called before each Newton step, the method exitNewtonStep after each Newton step. assembleNewtonStep assembles the linear system of equations needed for the next step, solveNewtonStep solves this system of equations. The solution is the correction d. The method getCorrection returns a pointer to the vector storing the correction.

Now, the Newton method will be implemented. Actually, the class NewtonMethod replaces the whole iteration in the adaptation loop, including mesh adaptation and error estimation. The Newton method, which is a loop over Newton steps, is one part of this iteration.

```
class NewtonMethod : public ProblemIterationInterface
{
public:
  NewtonMethod(const char *name,
               ProblemScal *problem,
               NewtonStepInterface *step)
    : problemNonlin(problem),
      newtonStep(step),
      newtonTolerance(1e-8),
      newtonMaxIter(100)
  {
    GET_PARAMETER(0, std::string(name) + "->tolerance", "%f",
                  &newtonTolerance);
    GET_PARAMETER(0, std::string(name) + "->max iteration", "%d",
                  &newtonMaxIter);
    solution = problemNonlin->getSolution();
    correction = newtonStep->getCorrection();
  };
```

In the constructor, pointers to the nonlinear problem and to the Newton-step object are stored to the class members `problemNonlin` and `newtonStep`. Furthermore, the parameters `newtonTolerance` and `newtonMaxIter` are initialized, and pointers to the nonlinear solution and to the correction vector are stored.

The following methods define one iteration in the adaptation loop.

```
  void beginIteration(AdaptInfo *adaptInfo)
  {
    FUNCNAME("NewtonMethod::beginIteration()");
    MSG("\n");
    MSG("begin of iteration %d\n", adaptInfo->getSpaceIteration()+1);
    MSG("============================\n");
  }

  Flag oneIteration(AdaptInfo *adaptInfo, Flag toDo = FULL_ITERATION)
  {
    Flag flag = 0, markFlag = 0;

    if(toDo.isSet(MARK)) markFlag = problemNonlin->markElements(adaptInfo);
    if(toDo.isSet(ADAPT) && markFlag.isSet(MESH_REFINED))
      flag = problemNonlin->refineMesh(adaptInfo);
    if(toDo.isSet(ADAPT) && markFlag.isSet(MESH_COARSENED))
      flag |= problemNonlin->coarsenMesh(adaptInfo);

    if(toDo.isSet(SOLVE)) {
      newtonStep->initNewtonStep(adaptInfo);
      int newtonIteration = 0;
      double res = 0.0;
      do {
        newtonIteration++;
        newtonStep->assembleNewtonStep(adaptInfo, flag);
        newtonStep->solveNewtonStep(adaptInfo);
        res = correction->L2Norm();
        *solution -= *correction;
        MSG("newton iteration %d: residual %f (tol: %f)\n",
            newtonIteration, res, newtonTolerance);
      } while((res > newtonTolerance) && (newtonIteration < newtonMaxIter));

      newtonStep->exitNewtonStep(adaptInfo);
```

```
    }

    if(toDo.isSet(ESTIMATE)) problemNonlin->estimate(adaptInfo);
    return flag;
  };

  void endIteration(AdaptInfo *adaptInfo)
  {
    FUNCNAME("NewtonMethod::endIteration()");
    MSG("\n");
    MSG("end of iteration number: %d\n", adaptInfo->getSpaceIteration()+1);
    MSG("==============================\n");
  }
```

The methods `beginIteration` and `endIteration` only print some information to the standard output. In `oneIteration`, the iteration, including the loop over the Newton steps, is defined.

Finally, the methods `getNumProblems` and `getProblem` are implemented to complete the `ProblemIterationInterface`, and the private class members are defined.

```
  int getNumProblems() { return 1; };

  ProblemStatBase *getProblem(int number = 0)
  {
    FUNCNAME("NewtonMethod::getProblem()");
    if(number == 0) return problemNonlin;
    ERROR_EXIT("invalid problem number\n");
    return NULL;
  };

private:
  ProblemScal *problemNonlin;
  NewtonStepInterface *newtonStep;
  double newtonTolerance;
  int newtonMaxIter;
  DOFVector<double> *solution;
  DOFVector<double> *correction;
};
```

The class `Nonlin` implements both, the nonlinear problem and the Newton-step. Since the Newton step is accessed over an own interface, it is always clear, in which role a `Nonlin` instance is called by the Newton method.

```
class Nonlin : public ProblemScal,
               public NewtonStepInterface
{
public:
  Nonlin(const char *name)
    : ProblemScal(name)
  {};
```

In the constructor, the base class constructor of `ProblemScal` is called and the name is given to it.

In the initialization, the base class intialization is called, the correction vector is created and initialized, and Dirichlet boundary conditions are created.

```
  void initialize(Flag initFlag,
                  ProblemScal *adoptProblem = NULL,
                  Flag adoptFlag = INIT_NOTHING)
  {
```

```
    ProblemScal::initialize(initFlag, adoptProblem, adoptFlag);
    correction = NEW DOFVector<double>(this->getFESpace(), "old␣solution");
    correction->set(0.0);

    dirichletZero = NEW DirichletBC(1, &zero, feSpace_);
    dirichletG    = NEW DirichletBC(1, &g, feSpace_);

    solution_->getBoundaryManager()->addBoundaryCondition(dirichletG);
    systemMatrix_->getBoundaryManager()->
      addBoundaryCondition(dirichletZero);
    rhs_->getBoundaryManager()->addBoundaryCondition(dirichletZero);
    correction->getBoundaryManager()->addBoundaryCondition(dirichletZero);
  };
```

To the solution of the nonlinear problem the function g is applied as Dirichlet boundary function. The system matrix, the correction vector and the right hand side vector build the system for the Newton step. Since no correction has to be done at the domain boundaries, zero Dirichlet conditions are applied to them.

In the destructor, the allocated memory is freed.

```
~Nonlin()
{
  DELETE correction;
  DELETE dirichletZero;
  DELETE dirichletG;
};
```

Now, we implement the Newton step functionality. First, in `initNewtonStep`, we fill the solution vector with boundary values. This will not be done automatically because we let the `solution_` pointer point to `correction`. The address of `solution_` is stored in `tmp`. After the Newton method is finished, the `solution_` pointer is reset to its original value in `exitNewtonStep`.

```
void initNewtonStep(AdaptInfo *adaptInfo) {
  solution_->getBoundaryManager()->initVector(solution_);
  TraverseStack stack;
  ElInfo *elInfo = stack.traverseFirst(mesh_, -1,
                                       Mesh::CALL_LEAF_EL |
                                       Mesh::FILL_COORDS |
                                       Mesh::FILL_BOUND);
  while(elInfo) {
    solution_->getBoundaryManager()->fillBoundaryConditions(elInfo,
                                                            solution_);
    elInfo = stack.traverseNext(elInfo);
  }
  solution_->getBoundaryManager()->exitVector(solution_);

  tmp = solution_;
  solution_ = correction;
};

void exitNewtonStep(AdaptInfo *adaptInfo) {
  solution_ = tmp;
};
```

The implementation of `assembleNewtonStep` and `solveNewtonStep` just delegates the calls to the base class implementations in `ProblemScal`.

```
void assembleNewtonStep(AdaptInfo *adaptInfo, Flag flag) {
  ProblemScal::buildAfterCoarsen(adaptInfo, flag);
```

```
  };

  void solveNewtonStep(AdaptInfo *adaptInfo) {
    ProblemScal::solve(adaptInfo);
  };
```

Finally, the `getCorrection` method is implemented and private class members are defined.

```
  DOFVector<double> *getCorrection() { return correction; };

private:
  DOFVector<double> *correction;
  DOFVector<double> *tmp;
  Zero zero;
  G g;
  DirichletBC *dirichletZero;
  DirichletBC *dirichletG;
};
```

Now, we start with the main program.

```
int main(int argc, char* argv[])
{
  FUNCNAME("main");

  TEST_EXIT(argc == 2)("usage:␣nonlin␣initfile\n");

  Parameters::init(false, argv[1]);

  Nonlin nonlin("nonlin");
  nonlin.initialize(INIT_ALL);

  AdaptInfo *adaptInfo = NEW AdaptInfo("nonlin->adapt", 1);

  NewtonMethod newtonMethod("nonlin->newton", &nonlin, &nonlin);

  AdaptStationary *adapt = NEW AdaptStationary("nonlin->adapt",
                                               &newtonMethod,
                                               adaptInfo);
```

An instance of class `NewtonMethod` was created with a pointer to a `Nonlin` object as nonlinear problem and as Newton-step implementation. Instead of the nonlinear problem, now, the object `newtonMethod` is handed to the adaptation loop as implementation of `ProblemIterationInterface`.

We have to add operators representing the Newton step equation $-\Delta d + 4u_n^3 d = -\Delta u_n + u_n^4 - f$ as well as operators representing the nonlinear problem $-\Delta u_n + u_n^4 = f$. When the operators are given to the problem, one can determine an assemble factor (second argument of `addMatrixOperator` and `addVectorOperator`) as well as an estimation factor (third argument) for each operator. So, we can manage both equations in one problem instance. Note that in the Newton step we solve for d. u_n there is known from the last Newton step. The term u_n^4 was implemented as $u_n^3 v$, where v for the Newton step equation is equal to d and in the nonlinear problem equation equal to u_n. So, the corresponding operator can be used in both equations just with different factors. In Figure A.10, the operator factors for the assemble and the estimate step are shown.

```
  // ===== create operators =====
  double four = 4.0;
  double one = 1.0;
  double zero = 0.0;
```

	$-\Delta v$		$u_n^{\,3}\,v$		f	
	mat	vec	mat	vec	mat	vec
assemble $(v\!=\!d)$	1	1	4	1	0	-1
estimate $(v\!=\!u_n)$	1	0	1	0	0	1

Figure A.10: Operator factors for the assemble step and for the estimate step.

```
double minusOne = -1.0;

Operator *nonlinOperator0 = NEW Operator(Operator::MATRIX_OPERATOR |
                                         Operator::VECTOR_OPERATOR ,
                                         nonlin.getFESpace());

nonlinOperator0->setUhOld(nonlin.getSolution());
nonlinOperator0->addZeroOrderTerm(NEW VecAtQP_ZOT(nonlin.getSolution(),
                                        NEW X3));

nonlin.addMatrixOperator(nonlinOperator0, &four, &one);
nonlin.addVectorOperator(nonlinOperator0, &one, &zero);

Operator *nonlinOperator2 = NEW Operator(Operator::MATRIX_OPERATOR |
                                         Operator::VECTOR_OPERATOR ,
                                         nonlin.getFESpace());

nonlinOperator2->setUhOld(nonlin.getSolution());
nonlinOperator2->addSecondOrderTerm(NEW Laplace_SOT);

nonlin.addMatrixOperator(nonlinOperator2, &one, &one);
nonlin.addVectorOperator(nonlinOperator2, &one, &zero);

int degree = nonlin.getFESpace()->getBasisFcts()->getDegree();

Operator* rhsFunctionOperator = NEW Operator(Operator::VECTOR_OPERATOR ,
                                         nonlin.getFESpace());
rhsFunctionOperator->addZeroOrderTerm(NEW CoordsAtQP_ZOT(NEW F(degree)));

nonlin.addVectorOperator(rhsFunctionOperator, &minusOne, &one);
```

Finally, the adaptation loop is started and after it is fnished the result is written.

```
adapt->adapt();
nonlin.writeFiles(adaptInfo, true);
}
```

Parameter file

The used parameter file `nonlin.dat.2d` looks like:

```
dimension of world:    2

nonlinMesh->macro file name:      ./macro/macro_big.stand.2d
nonlinMesh->global refinements:   0
```

```
nonlin ->mesh:    nonlinMesh

nonlin ->dim :                2
nonlin ->polynomial degree :   1

nonlin ->newton ->tolerance :     1e-8
nonlin ->newton ->max iteration: 100

nonlin ->solver :                 cg
nonlin ->solver ->max iteration:  1000
nonlin ->solver ->tolerance :     1.e-8
nonlin ->solver ->left precon :   diag

nonlin ->estimator :              residual
nonlin ->estimator ->C0 :         0.1
nonlin ->estimator ->C1 :         0.1

nonlin ->marker ->strategy :      2
nonlin ->marker ->MSGamma :       0.5

nonlin ->adapt ->tolerance :      1e-1
nonlin ->adapt ->max iteration:   100

nonlin ->output ->filename :       output/nonlin
nonlin ->output ->AMDiS format :   1
nonlin ->output ->AMDiS mesh ext: .mesh
nonlin ->output ->AMDiS data ext: .dat
```

Here, as macro file `macro_big.stand.2d` is used, which is described in Section A.4.5. The parameters `nonlin->newton->tolerance` and `nonlin->newton->max iteration` determine the tolerance of the Newton solver and the maximal number of Newton steps within each iteration of the adaptation loop.

Macro file

The used macro file `macro_big.stand.2d` is very similar to the macro file of the first example described in Section A.4.1. Only the domain was changed from $\Omega = [0,1]^2$ to $\Omega = [-1,1]^2$ by adapting the vertex coordinates correspondingly.

Output

In Figure A.11, the solution and the final mesh written after the adaptation loop are visualized. The solution is shown as height field where the values are interpreted as z-coordinates.

A.4.6   Neumann boundary conditions

In this example, we solve the problem defined in Section A.4.1. But now, we set the domain Ω to $[-0.5; 0.5]^2$, so the source f is located in the middle of Ω. Furthermore, we use Neumann boundary conditions on the left and on the right side of Ω. We set $A\nabla u \cdot \nu = 1$ at the Neumann boundary. So, the derivative in direction of the surface normal is set to 1 at these points. The rest of the boundary keeps unchanged (Dirichlet boundary, set to the true solution).
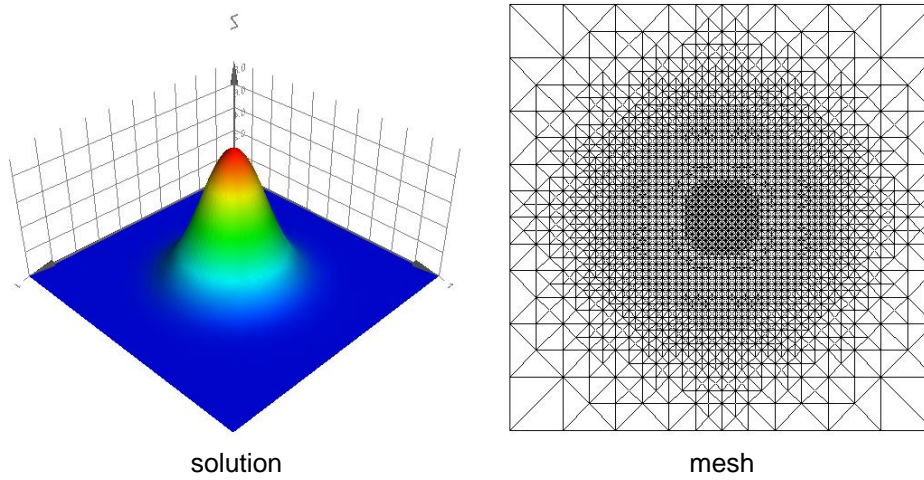
solution   mesh

Figure A.11: Solution and final mesh of the nonlinear problem

		1d	**2d**	**3d**
source code	`src/`	`neumann.cc`		
parameter file	`init/`	`neumann.dat.1d`	`neumann.dat.2d`	`neumann.dat.3d`
macro file	`macro/`	`neumann.macro.1d`	`neumann.macro.2d`	`neumann.macro.3d`
output files	`output/`	`neumann.mesh, neumann.dat`		

Table A.6: Files of the `neumann` example.

Source code

Only a few changes in the source code are necessary to apply Neumann boundary conditions. First, we define the function $N = 1$.

```cpp
class N : public AbstractFunction<double, WorldVector<double> >
{
public:
  MEMORY_MANAGED(N);

  const double& operator()(const WorldVector<double>& x) const
  {
    static double result = 1.0;
    return result;
  };
};
```

In the main program we add the boundary conditions to our problem `neumann`.

```cpp
int main(int argc, char* argv[])
{
  ...
  neumann.addNeumannBC(1, NEW N);
  neumann.addDirichletBC(2, NEW G);
  ...
}
```

Since the Dirichlet condition has a higher ID, it has a higher priority against the Neumann boundary condition. This is important, where different conditions meet each other in some points.

In this example, these are the corner points of Ω. If Dirichlet boundary conditions are used together with boundary conditions of other type, the Dirichlet conditions should always have the higher priority.

Parameter file

In the parameter file, we use the file `./macro/neumann.macro.2d` as macro mesh file, described in the next section.

Macro file

The file `neumann.macro.2d` is listed below:

```
DIM: 2
DIM_OF_WORLD: 2

number of vertices: 5
number of elements: 4

vertex coordinates:
-0.5 -0.5
 0.5 -0.5
 0.5  0.5
-0.5  0.5
 0.0  0.0

element vertices:
0 1 4
1 2 4
2 3 4
3 0 4

element boundaries:
0 0 2
0 0 1
0 0 2
0 0 1
```

In contrast to the standard file `macro.stand.2d`, here the vertex coordinates are shifted to describe the domain $[-0.5; 0.5]^2$. Furthermore, the boundary block changed. The elements 0 and 2 have the Dirichlet boundary with ID 2 at edge 2. Elements 1 and 3 have the Neumann boundary condition with ID 1 applied to their local edge 2.

Output

In Figure A.12, the solution is shown. At the Neumann boundaries, one can see the positive slope. At Dirichlet boundaries, the solution is set to $g(x)$.

A.4.7   Periodic boundary conditions

Periodic boundary conditions allow to simulate an effectively infinite tiled domain, where the finite domain Ω is interpreted as one tile of the infinte problem domain. The solution outside of Ω can be constructed by periodically continue the solution within Ω. In Figure A.13 two examples for periodic boundary conditions on a two dimensional domain are illustrated. On the left hand side example, the upper and the lower part of the boundary as well as the left and the right part of the
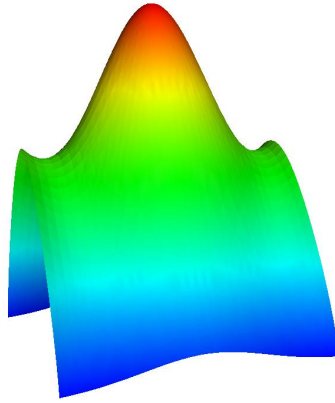
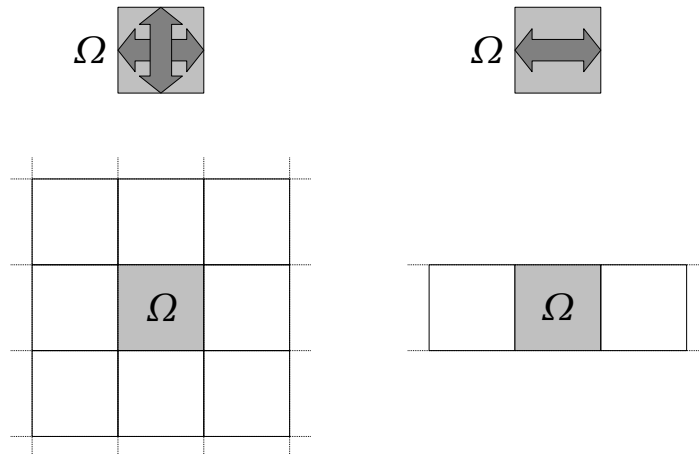Figure A.12: Solution of the problem with Neumann boundary conditions at two sides.

Figure A.13: Two dimensional domain with periodic boundary conditions in both dimensions (left) and in only one dimension (right). In the first case, the solution at Ω can be propagated to the whole plane of $\mathbb{R}^2$. In the second case, the solution only describes a band within $\mathbb{R}^2$
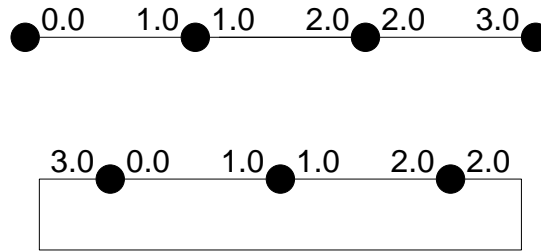
Figure A.14: A one dimensional mesh with vertex coordinates stored at the elements and a corresponding periodic mesh with changed mesh topology. Note that the geometric data are not changed. The coordinates of the first vertex depend on the element it belongs to.

		1d	2d	3d
source code	src/	periodic.cc
parameter file	init/	periodic.dat.1d	periodic.dat.2d	periodic.dat.3d
periodic file	init/	periodic.per.1d	periodic.per.2d	periodic.per..3d
macro file	macro/	periodic.macro.1d	periodic.macro.2d	periodic.macro.3d
output files	output/	periodic.mesh, periodic.dat

Table A.7: Files of the `periodic` example.

boundary are assigned to each other as periodic boundary. This results in a solution, which tiles the infinte plane. On the right hand side example, only the left and the right part of the domain boundary are assigned to each other, which results in a infinte band.

In AMDiS, there are two ways to implement periodic boundary conditions:

1. Changing the mesh topology (*mode 0*): Before the computation is started, the topology of the macro mesh is changed. Two vertices that are assigned to each other by a periodic boundary condition, are replaced by one single vertex, which is now treated as an inner vertex of the mesh (if it is not part of any other boundary). Since geometric data like coordinates are stored at elements and not at vertices, this modification does not change the geometry of the problem. Topological information, like element neighborhood, does change. The method is illustrated in Figure A.14.

2. Modify the linear system of equations in each iteration (*mode 1*): Sometimes it is necessary to store geometric information at vertices. E.g., if moving meshes are implemented with parametric elements, a DOF vector may store the coordinates. In this case, the mesh topology keeps unchanged, and the periodic boundary conditions are applied, like any other boundary condition, after the assemblage of the linear system of equations. In the application source code a boundary condition object has to be created, and in the macro file the periodic boundary must be specified.

In this section, we show how to use both variants of periodic boundary conditions. Again, we use the problem defined in Section A.4.1. We choose $\Omega = [-0.2; 0.8] \times [-0.5; 0.5]$ (we do not use $\Omega = [-0.5; 0.5]^2$, because in this example periodic boundary conditions would then be equal to the trivial zero flux conditions).

We apply a periodic boundary condition which connects the left and the right edge of Ω. Since we do not know the exact solution of this periodic problem, we apply zero Dirichlet conditions at the lower and upper edge of the domain.

Source code

If we use *mode 0*, no modifications in the source code have to be made. For *mode 1*, we have to add a periodic boundary condition object to the problem.

```
periodic.addPeriodicBC(-1);
```

Note that periodic boundary conditions must be described by negative numbers.

Parameter file

In the parameter file, we add an link to the *periodic file*.

```
periodicMesh->periodic file:        ./init/periodic.per.2d
```

The periodic file `periodic.per.2d` contains the needed periodic information for the mesh.

```
associations: 2

mode  bc  el1 - local vertices <->  el2 - local vertices
 1    -1    4        1 2              7        2 1
 1    -1    0        1 2              3        2 1
```

First, the number of edge associations (point associations in 2d, face associations in 3d) is given. Then each association is described in one line. The first entry is the mode which should be used for this periodic association. If the mode is 1, the next entry specifies the identifier of the used boundary condition. This identifier also must be used in the source code and in the macro file. If the mode is 0, the identifier is ignored. The rest of the line describes, which (local) vertices of which elements are associated with each other. The first association in this example is interpreted as follows: The local vertices 1 and 2 of element 4 are associated with the vertices 2 and 1 of element 7. Or more precisely, vertex 1 of element 4 is associated with vertex 2 of element 7, and vertex 2 of element 4 with vertex 1 of element 7.

Macro file

To avoid degenerated elements, one macro element must not contain two vertices which are associated with each other. Therefore, we choose a macro mesh with a few more elements, showed in Figure A.15.

The corresponding file `periodic.macro.2d` looks like:

```
DIM: 2
DIM_OF_WORLD: 2

number of elements: 8
number of vertices: 9

element vertices:
1 3 0
3 1 4
2 4 1
4 2 5
4 6 3
6 4 7
5 7 4
7 5 8

element boundaries:
-1 1 0
```
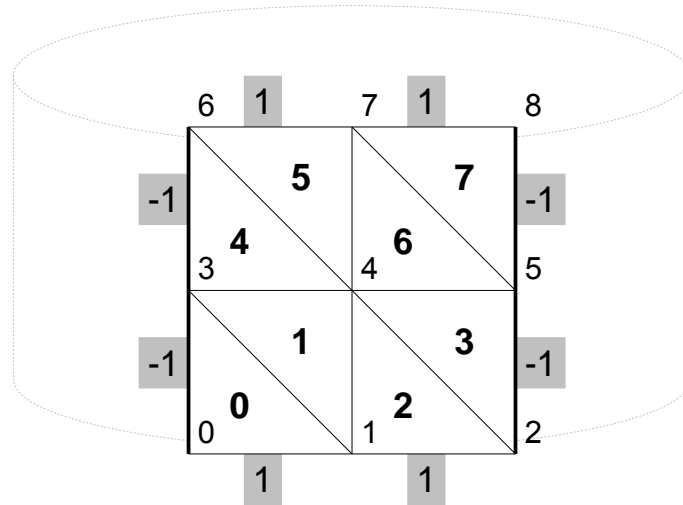
Figure A.15: Macro mesh for the two dimensioanl periodic problem.

```
 0 0 0
 0 1 0
-1 0 0
-1 0 0
 0 1 0
 0 0 0
-1 1 0

vertex coordinates:
-0.2 -0.5
 0.3 -0.5
 0.8 -0.5
-0.2  0.0
 0.3  0.0
 0.8  0.0
-0.2  0.5
 0.3  0.5
 0.8  0.5

element neighbours:
 3 -1  1
 2  4  0
 1 -1  3
 0  6  2
 7  1  5
 6 -1  4
 5  3  7
 4 -1  6
```

Compared to the macro file of Section A.4.1, the vertex coordinates are shifted by -0.2 in x-direction.

In the boundary block -1 specifies the periodic boundary. If we use *mode 0*, this boundaries are ignored (Here, the minus sign becomes important! Only boundary conditions with negative IDs are recognized as periodic boundaries and can be ignored if they are not used).
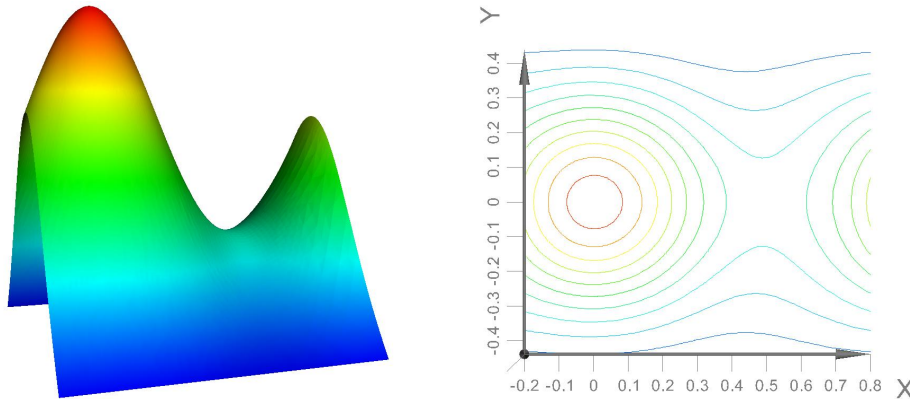
Figure A.16: Solution of the problem with periodic boundary conditions at two sides (left) and iso lines of the solution for the values $0.1, 0.2, \ldots, 1.1$ (right).
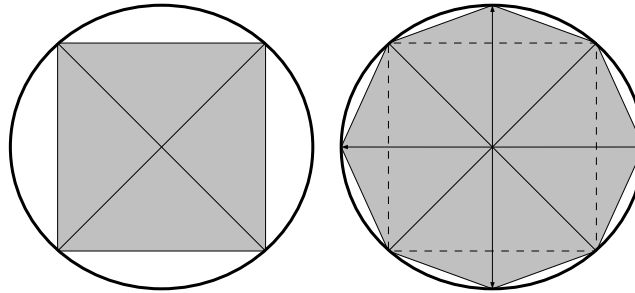
Figure A.17: Boundary projection for a two dimensional mesh. The boundary vertices of the mesh are projected on the circle.

In the neighbors block, neighborships between elements that are connected by a periodic edge (point/face) are added. Note that this must also be done for *mode 1* periodic boundaries.

Output

In Figure A.16, the solution of our periodic problem is shown as height field at the left hand side. At the right hand side, one can see iso lines for the values $0.1, 0.2, \ldots, 1.1$.

A.4.8 Projections

In AMDiS, projections can be applied to the vertex coordinates of a mesh. There are two types of projections:

1. *Boundary projections*: Only vertices at the domain boundary are projected.

2. *Volume projections*: All vertices of the mesh are projected.

Projections are applied to all (boundary) vertices of the macro mesh and to each new (boundary) vertex, created during adaptive refinements. In Figure A.17, this is illustrated for a two dimensional mesh which boundary vertices are successively projected to a circle. In Figure A.18 the vertices of a one dimensional mesh are successively projected on the circle.

Figure A.18: Volume projection for the one dimensional mesh. All vertices of this mesh are projected on the circle.

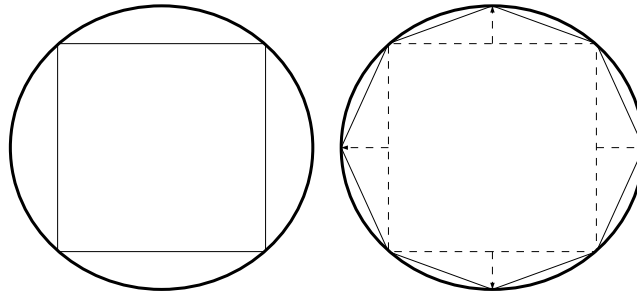|		1d	2d	3d
source code	src/	sphere.cc		
parameter file	init/	–	–	sphere.dat.3d
macro file	macro/	–	–	sphere.macro.3d
output files	output/	sphere.mesh, sphere.dat		

Table A.8: Files of the `sphere` example.

In this section, we give an example for both projection types. As projection we choose the projection to the unit sphere in 3d. In the first example, we start with the three dimensional cube $[-1, 1]^3$ and solve the three dimensional version of problem A.1 in it. Furthermore, we apply a boundary projection to the unit sphere. In the second example, we set the right hand side f of equation (A.1) to $2x_0$ (x_0 is the first component of x), and solve on the two dimensional surface of the sphere. Here, we start with a macro mesh that defines the surface of a cube and apply a volume projection to it.

Source code

First we define the projection by implementing a sub class `BallProject` of the base class `Projection`.

```
class BallProject : public Projection
{
public:
  BallProject(int id,
              ProjectionType type,
              WorldVector<double> &center,
              double radius)
    : Projection(id, type),
      center_(center),
```

		1d	2d	3d
source code	src/	ball.cc		
parameter file	init/	–	–	ball.dat.3d
macro file	macro/	–	–	ball.macro.3d
output files	output/	ball.mesh, ball.dat		

Table A.9: Files of the `ball` example.

```
      radius_(radius)
  {};

  void project(WorldVector<double> &x) {
    x -= center_;
    double norm = sqrt(x*x);
    TEST_EXIT(norm != 0.0)("can't project vector x\n");
    x *= radius_/norm;
    x += center_;
  };

protected:
  WorldVector<double> center_;
  double radius_;
};
```

First, in the constructor, the base class constructor is called with a projection identifier and the projection type which can be `BOUNDARY_PROJECTION` or `VOLUME_PROJECTION`. The projection identifier is used to associated a projection instance to projections defined in the macro file. The method `project` implements the concrete projection of a point x in world coordinates.

If we compute on the surface, we redefine the function f.

```
class F : public AbstractFunction<double, WorldVector<double> >
{
public:
  MEMORY_MANAGED(F);

  F(int degree) : AbstractFunction<double, WorldVector<double> >(degree) {};

  const double& operator()(const WorldVector<double>& x) const
  {
    static double result = 0.0;
    result = -2 * x[0];
    return result;
  };
};
```

In the main program, we create an instance of `BallProject` with ID 1, center 0 and radius 1. If we solve in the three dimensional volume of the sphere, the projection type is `BOUNDARY_PROJECTION`, because we project only boundary vertices to the sphere.

```
// ===== create projection =====
WorldVector<double> ballCenter;
ballCenter.set(0.0);
NEW BallProject(1,
                BOUNDARY_PROJECTION,
                ballCenter,
                1.0);
```

If we solve on the two dimensional surface of the sphere, the projection type is `VOLUME_PROJECTION`, because all vertices of the mesh are projected.

```
// ===== create projection =====
WorldVector<double> ballCenter;
ballCenter.set(0.0);
NEW BallProject(1,
                VOLUME_PROJECTION,
                ballCenter,
                1.0);
```

Parameter file

First, we present the parameter file for the volume projection case (two dimensional mesh).

```
dimension of world:        3

sphereMesh->macro file name:         ./macro/sphere_macro.3d
sphereMesh->global refinements:    10

sphere->mesh:                  sphereMesh
sphere->dim:                   2
sphere->polynomial degree:     1

sphere->solver:                  cg
sphere->solver->max iteration: 100
sphere->solver->tolerance:     1.e-8
sphere->solver->left precon:   diag

sphere->estimator:               no
sphere->marker->strategy:      0

sphere->output->filename:        output/sphere
sphere->output->AMDiS format:    1
sphere->output->AMDiS mesh ext: .mesh
sphere->output->AMDiS data ext: .dat
```

The world dimension is 3, whereas the mesh dimension is set to 2. We use a macro mesh which defines the surface of a cube, defined in `./macro/sphere_macro.3d`, and apply 10 global refinements to it. In this example we do not use adaptivity. Thus, no estimator and no marker is used.

Now, we show the parameter file for the boudary projection case (three dimensional mesh).

```
dimension of world:           3

ballMesh->macro file name:    ./macro/macro.ball.3d
ballMesh->global refinements: 15

ball->mesh:                  ballMesh
ball->dim:                   3
ball->polynomial degree:     1

ball->solver:                  cg
ball->solver->max iteration:  1000
ball->solver->tolerance:      1.e-8
ball->solver->left precon:    diag

ball->estimator:               no
ball->marker->strategy:       0

ball->output->filename:        output/ball
ball->output->AMDiS format:    1
ball->output->AMDiS mesh ext: .mesh
ball->output->AMDiS data ext: .dat
```

The macro mesh is a three dimensional cube, defined in `./macro/macro.ball.3d`, and 15 times globally refined.

Macro file

First, the macro file for the two dimensional mesh.

```
DIM:           2
DIM_OF_WORLD: 3

number of vertices: 8
number of elements: 12

vertex coordinates:
 -1.0     1.0    -1.0
  1.0     1.0    -1.0
  1.0     1.0     1.0
 -1.0     1.0     1.0
 -1.0    -1.0    -1.0
  1.0    -1.0    -1.0
  1.0    -1.0     1.0
 -1.0    -1.0     1.0

element vertices:
 3 1 0
 1 3 2
 2 5 1
 5 2 6
 6 4 5
 4 6 7
 7 0 4
 0 7 3
 2 7 6
 7 2 3
 1 4 0
 4 1 5

element boundaries:
 0 0 0
 0 0 0
 0 0 0
 0 0 0
 0 0 0
 0 0 0
 0 0 0
 0 0 0
 0 0 0
 0 0 0
 0 0 0
 0 0 0

projections:
 1 0 0
 1 0 0
 1 0 0
 1 0 0
 1 0 0
 1 0 0
 1 0 0
 1 0 0
 1 0 0
 1 0 0
```

```
1 0 0
1 0 0
```

In the projections block, the projection IDs for each element are listed. There is one entry for each side of each element. Since we use volume projection, here, only the first entry of a line is used.

Now, we list the macro file for the three dimensional volume mesh.

```
DIM:           3
DIM_OF_WORLD: 3

number of vertices: 8
number of elements: 6

vertex coordinates:
 -1.0 -1.0  0.0
  0.0 -1.0 -1.0
  0.0 -1.0  1.0
  1.0 -1.0  0.0
  0.0  1.0 -1.0
  1.0  1.0  0.0
 -1.0  1.0  0.0
  0.0  1.0  1.0

element vertices:
  0     5     4     1
  0     5     3     1
  0     5     3     2
  0     5     4     6
  0     5     7     6
  0     5     7     2

element boundaries:
  1     1     0     0
  1     1     0     0
  1     1     0     0
  1     1     0     0
  1     1     0     0
  1     1     0     0

element neighbours:
 -1    -1     1     3
 -1    -1     0     2
 -1    -1     5     1
 -1    -1     4     0
 -1    -1     3     5
 -1    -1     2     4

projections:
  1     1     0     0
  1     1     0     0
  1     1     0     0
  1     1     0     0
  1     1     0     0
  1     1     0     0
```

Here, we use boundary projections. In the boundary block for each boundary side of an element the projection ID is given.
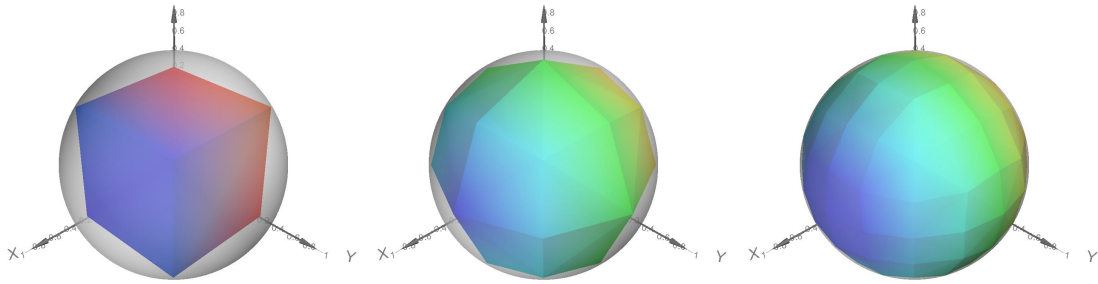
Figure A.19: Surface of a cube successively refinend and projected on the sphere.
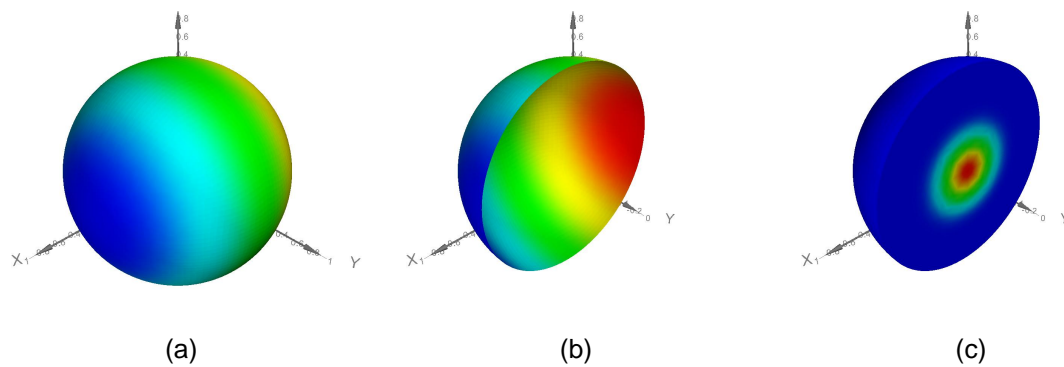

(a)   (b)   (c)

Figure A.20: (a): Solution of the two dimensional problem on the surface of the sphere, (b): Halfed sphere, (c): Solution of the three dimensional problem (halfed ball).

Output

In Figure A.19, the solution of the two dimensional problem is shown on a successively refined mesh whose vertices are projected on the sphere. The finer the mesh, the better is the approximation to the sphere.

In Figure A.20 (a), the final solution of the two dimensional problem is shown, Figure A.20 (b) shows the halfed sphere to demonstrate that the solution is really defined on the sphere. The solution of the three dimensional problem is shown in Figure A.20 (c).

A.4.9 Parametric elements

With parametric elements, problems can be solved on meshes which dimensions are not necessarily equal to the world dimension. Therefore, problems on arbitrary manifolds can be solved. Furthermore, the vertex coordinates of the mesh can be flexible. Hence, moving meshes can be implemented.

In this section, we solve equation (A.1) with $f = 2x_0$ (x_0 is the first component of x) on a torus. Then we rotate the torus about the y-axis and solve the problem again.

The torus can be created by revolving a circle about an axis coplanar with the circle, which does not touch the circle. We call r_1 the radius of the revolved circle and r_2 the radius of the revolution, which is the distance of the center of the tube to the center of the torus. In Figure A.21, a torus with its two radii r_1 and r_2 is shown.

We create a torus with center $(0; 0; 0)$ and the rotation axis in z-direction $(0; 0; 1)$. The projection of a point x_0 on the torus is implemented by the following steps:

1. x_1 is the projection of x_0 on the xy-plane
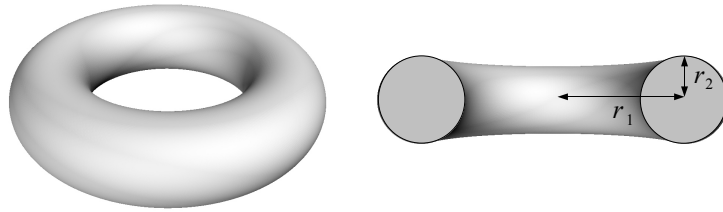
Figure A.21: A torus and a halfed torus with the two radiuses r_1 and r_2.

		1d	2d	3d
source code	`src/`			`torus.cc`
parameter file	`init/`	–	–	`torus.dat.3d`
macro file	`macro/`	–	–	`torus.macro.3d`
output files	`output/`	`torus.mesh/.dat, rotation1.mesh/.dat, rotation2.mesh/dat`		

Table A.10: Files of the `torus` example.

2. $x_2 = x_1 \frac{r_1}{||x_1||}$. Projection of x_1 on the sphere with radius r_1 with center 0. Thereby, x_2 is used as the center of a sphere with radius r_2.

3. $x_3 = x_0 - x_2$. Move coordinate system into the center of the sphere with center x_2. Thereby, x_3 contains the coordinates of x_0 in this new coordinate system.

4. $x_4 = x_3 \frac{r_2}{||x_3||}$. Thereby, x_4 is the projection of x_3 on the sphere with radius r_2.

5. $x_5 = x_4 + x_2$. Thereby, x_5 contains the coordinates of x_4 in the original coordinate system. It is the projection of x_0 on the torus.

Source code

First, we define the rotation about the y-axis, which is used later to rotate the whole torus and the right hand side function.

```
class YRotation
{
public:
  static WorldVector<double>& rotate(WorldVector<double> &x, double angle)
  {
    double x0 = x[0] * cos(angle) + x[2] * sin(angle);
    x[2] = -x[0] * sin(angle) + x[2] * cos(angle);
    x[0] = x0;
    return x;
  };
};
```

The right hand side function f has to follow the rotation of the torus.

```
class F : public AbstractFunction<double, WorldVector<double> >
{
public:
  MEMORY_MANAGED(F);
```

```
  F(int degree)
    : AbstractFunction<double, WorldVector<double> >(degree),
      rotation(0.0)
  {};

  const double& operator()(const WorldVector<double>& x) const {
    static double result = 0.0;
    WorldVector<double> myX = x;
    YRotation::rotate(myX, -rotation);
    result = -2 * myX[0];
    return result;
  };

  void rotate(double r) { rotation += r; };

private:
  double rotation;
};
```

Every time, the mesh is rotated, the right hand side function will be informed over the method rotate.

Now, we implement the projection on the torus.

```
class TorusProject : public Projection
{
public:
  TorusProject(int id,
               ProjectionType type,
               double radius1,
               double radius2)
    : Projection(id, type),
      radius1_(radius1),
      radius2_(radius2)
  {};

  virtual ~TorusProject() {};

  void project(WorldVector<double> &x) {

    WorldVector<double> xPlane = x;
    xPlane[2] = 0.0;

    double norm = std::sqrt(xPlane*xPlane);
    TEST_EXIT(norm != 0.0)("can't project vector x\n");

    WorldVector<double> center = xPlane;
    center *= radius1_ / norm;

    x -= center;

    norm = std::sqrt(x*x);
    TEST_EXIT(norm != 0.0)("can't project vector x\n");
    x *= radius2_/norm;

    x += center;
  };

protected:
```

```
  double radius1_;
  double radius2_;
};
```

In the main program, we create a torus projection as `VOLUME_PROJECTION` with ID 1. The values of r_1 and r_2 are chosen, such that the resulting torus is completely surrounded by the macro mesh that is defined later.

```
int main(int argc, char* argv[])
{
  FUNCNAME("torus␣main");

  // ===== check for init file =====
  TEST_EXIT(argc == 2)("usage:␣torus␣initfile\n");

  // ===== init parameters =====
  Parameters::init(false, argv[1]);

  // ===== create projection =====
  double r2 = (1.5 - 1.0/std::sqrt(2.0)) / 2.0;
  double r1 = 1.0/std::sqrt(2.0) + r2;

  NEW TorusProject(1,
                   VOLUME_PROJECTION,
                   r1,
                   r2);

  ...

  adapt->adapt();

  torus.writeFiles(adaptInfo, true);
```

The problem definition and the creation of the adaptation loop are done in the usual way (here, replaced by . . .) . After the adaptation loop has returned, we write the result.

Before we let the torus rotate, some variables are defined. We set the rotation angle to $\frac{\Pi}{3}$.

```
  double rotation = M_PI/3.0;
  int i, j;
  int dim = torus.getMesh()->getDim();
  int dow = Global::getGeo(WORLD);

  DegreeOfFreedom dof;
  WorldVector<double> x;

  const FiniteElemSpace *feSpace = torus.getFESpace();
  const BasisFunction *basFcts = feSpace->getBasisFcts();
  int numBasFcts = basFcts->getNumber();
  DegreeOfFreedom *localIndices = GET_MEMORY(DegreeOfFreedom, numBasFcts);
  DOFAdmin *admin = feSpace->getAdmin();

  WorldVector<DOFVector<double>*> parametricCoords;
  for(i = 0; i < dow; i++) {
    parametricCoords[i] = NEW DOFVector<double>(feSpace,
                                                "parametric␣coords");
  }
```

In the next step, we store the rotated vertex coordinates of the mesh in `parametricCoords`,a vector of DOF vectors, where the first vector stores the first component of each vertex coordinate,

and so on. In the STL map `visited`, we store which vertices have already been visited, to avoid multiple rotations of the same point.

```
std::map<DegreeOfFreedom, bool> visited;
TraverseStack stack;
ElInfo *elInfo = stack.traverseFirst(torus.getMesh(), -1,
                                     Mesh::CALL_LEAF_EL |
                                     Mesh::FILL_COORDS);
while(elInfo) {
  basFcts->getLocalIndices(elInfo->getElement(), admin, localIndices);
  for(i = 0; i < dim + 1; i++) {
    dof = localIndices[i];
    x = elInfo->getCoord(i);
    YRotation::rotate(x, rotation);
    if(!visited[dof]) {
      for(j = 0; j < dow; j++) {
        (*(parametricCoords[j]))[dof] = x[j];
      }
      visited[dof] = true;
    }
  }
  elInfo = stack.traverseNext(elInfo);
}
```

We create an instance of class `ParametricFirstOrder` which then is handed to the mesh. Now, in all future mesh traverses the vertex coordinates stored in `parametricCoords` are returned, instead of the original coordinates.

```
ParametricFirstOrder parametric(&parametricCoords);
torus.getMesh()->setParametric(&parametric);
```

We rotate the right hand side function, reset `adaptInfo` and start the adaptation loop again. Now, we compute the solution on the rotated torus, which then is written to the files `rotation1.mesh` and `rotation1.dat`.

```
f.rotate(rotation);
adaptInfo->reset();
adapt->adapt();

DataCollector *dc = NEW DataCollector(feSpace, torus.getSolution());
MacroWriter::writeMacro(dc, "output/rotation1.mesh");
ValueWriter::writeValues(dc, "output/rotation1.dat");
DELETE dc;
```

We perform another rotation. All we have to do is to modify the coordinates in `parametricCoords` and to inform f about the rotation.

```
visited.clear();
elInfo = stack.traverseFirst(torus.getMesh(), -1,
                             Mesh::CALL_LEAF_EL | Mesh::FILL_COORDS);
while(elInfo) {
  basFcts->getLocalIndices(elInfo->getElement(), admin, localIndices);
  for(i = 0; i < dim + 1; i++) {
    dof = localIndices[i];
    x = elInfo->getCoord(i);
    YRotation::rotate(x, rotation);
    if(!visited[dof]) {
      for(j = 0; j < dow; j++) {
        (*(parametricCoords[j]))[dof] = x[j];
```

```
      }
      visited[dof] = true;
    }
  }
  elInfo = stack.traverseNext(elInfo);
}

f.rotate(rotation);
adaptInfo->reset();
adapt->adapt();

dc = NEW DataCollector(feSpace, torus.getSolution());
MacroWriter::writeMacro(dc, "output/rotation1.mesh");
ValueWriter::writeValues(dc, "output/rotation1.dat");
DELETE dc;
```

The solution is written to `rotation2.mesh` and `rotation2.dat`.
Finally, we free some memory and finish the main program.

```
  for(i = 0; i < dow; i++)
    DELETE parametricCoords[i];
  FREE_MEMORY(localIndices, DegreeOfFreedom, numBasFcts);
}
```

Parameter file

In the parameter file, we set the macro file to `./macro/torus_macro.3d`. This two dimensional mesh is 8 times globally refined and successively projected on the torus.

```
dimension of world:      3

torusMesh->macro file name:      ./macro/torus_macro.3d
torusMesh->global refinements:   8

torus->mesh:              torusMesh
torus->dim:               2
torus->polynomial degree: 1

torus->solver:            cg
torus->solver->max iteration: 1000
torus->solver->tolerance:  1.e-8
torus->solver->left precon: diag
torus->estimator:          no
torus->marker:             no

torus->output->filename:     output/torus
torus->output->AMDiS format: 1
torus->output->AMDiS mesh ext: .mesh
torus->output->AMDiS data ext: .dat
```

Macro file

The macro mesh defined in `./macro/torus_macro.3d` is shown in Figure A.22.

Output
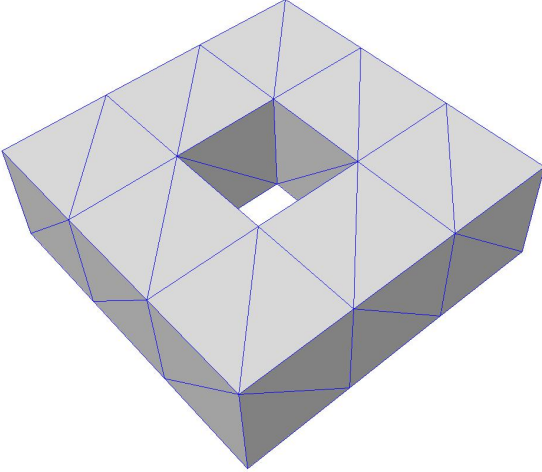
In Figure A.23, the solutions on the three tori are shown.
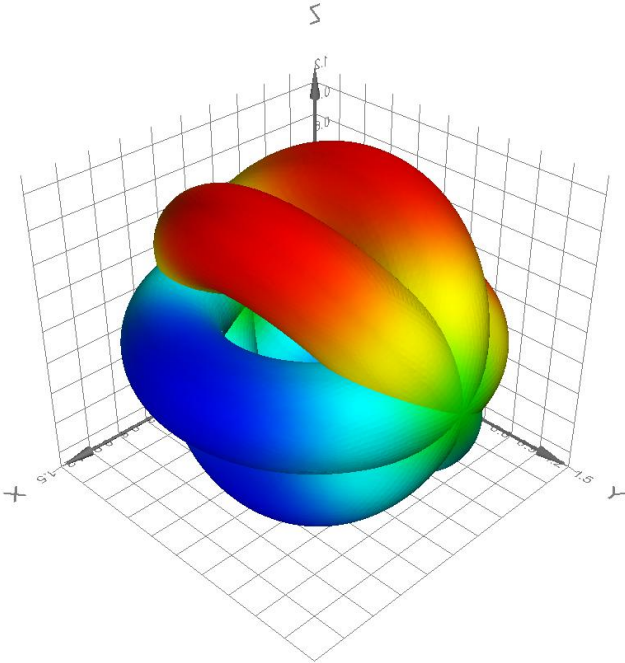
Figure A.22: Macro mesh of the torus problem.

Figure A.23: Solution on the original torus and on the two rotated tori.

		1d	**2d**	**3d**
source code	`src/`	`multigrid.cc`		
parameter file	`init/`	`multigrid.dat.1d`	`multigrid.dat.2d`	`multigrid.dat.3d`
macro file	`macro/`	`macro.stand.1d`	`macro.stand.2d`	`macro.stand.3d`
output files	`output/`	`multigrid.mesh, multigrid.dat`		

Table A.11: Files of the `multigrid` example.

A.4.10 Multigrid

The multigrid method is an effective way to solve large linear systems of equations. Its complexity is proportional to the number of unknowns in the system of equations. This can be achieved by using the hierarchical mesh structure of AMDiS. The multigrid idea is based on the following two principles:

- *Smoothing principle*: Classical iterative methods often have a strong smoothing effect on the error.

- *Coarse grid principle*: A quantity that is smooth on a given grid can be approximated on a coarser grid without any essential loss of information.

To solve $L_h u_h = f_h$ on the fine grid, we start with an arbitrary initial guess of the solution u_h. Then we apply the following steps, until a given tolerance criterion for the solution is fulfilled:

1. Apply some smoothing steps to u_h on the fine grid (pre-smoothing).

2. Compute the fine grid residual $d_h = f_h - L_h u_h$.

3. Restrict the residual d_h to the coarse grid ($d_h \rightarrow d_H$).

4. Solve the defect equation $L_H v_H = d_H$ on the coarse grid.

5. Interpolate v_H to the fine grid ($v_H \rightarrow v_h$).

6. Correct the solution: $u_h = u_h + v_h$.

7. Apply some smoothing steps to u_h on the fine grid (post-smoothing).

To solve the defect equation on the coarse level, we can apply the multigrid method recursively. This can be done once (*V-cycle*) or twice (*W-cycle*). At the coarsest level, the system of equations can be solved by a direct solver or again some smoothing steps are applied to it (in AMDiS, so far, no direct solver is applied at the coarsest level).

To use the multigrid solver in AMDiS, only the parameter file has to be modified. So, we omit the other sections here.

Parameter file

First, we have to avoid, that DOFs at coarse levels are freed, if they are not used at finer levels.

```
multigridMesh ->preserve  coarse  dofs :                   1
```

Now, we choose `mg` as solver, which is the key for the multigrid solver in AMDiS.

```
multigrid ->solver :                                     mg
```

The next three lines are not multigrid specific. They determine the solver tolerance, the maximal number of solver iterations, and the pre-conditioner.
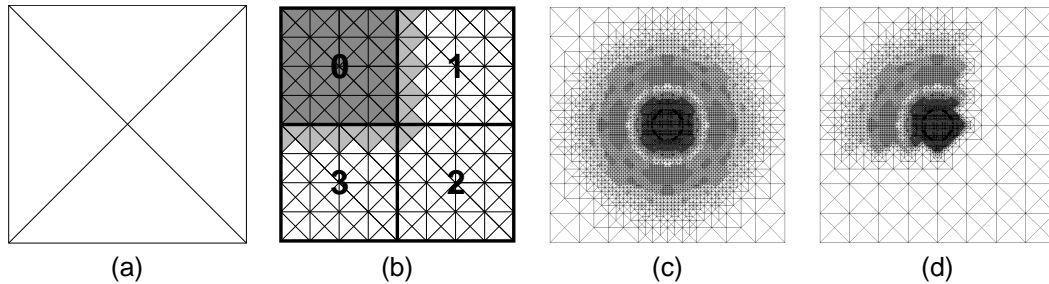
Figure A.24: (a) A triangular macro mesh, (b) domain decomposition after six global refinements and overlap for partition 0, (c) composite mesh after adaptation loop, (d) local mesh of process 0 after adaptation loop

```
multigrid ->solver ->tolerance :                        1.e-12
multigrid ->solver ->max iteration :                    100
multigrid ->solver ->left precon :                      diag
```

Now, multigrid specific parameters follow.

```
multigrid ->solver ->use galerkin operator :            0
multigrid ->solver ->mg cycle index :                   1 % 1: V, 2: W
multigrid ->solver ->smoother :                         gs
multigrid ->solver ->smoother ->omega :                 1.0
multigrid ->solver ->pre smoothing steps :              3
multigrid ->solver ->post smoothing steps :             3
multigrid ->solver ->coarse level smoothing steps:      1
multigrid ->solver ->min level :                        0
multigrid ->solver ->min level gap :                    1
multigrid ->solver ->max mg levels :                    100
```

The entry `use galerkin operator` determines, how the system on coarse levels is assembled. If the value is set to 1, the system of the finest level is successively restricted to coarser levels by the galerkin operator (which is only defined for linear Lagrange elements). If the value is set to 0, the system of coarser levels is assembled using the usual problem operators on the coarser meshes.

The `mg cycle index` determines the number of recursive multigrid calls within each level. A value of 1 results in V-cycle iterations, a value of 2 in W-cycle iterations (in principle, also higher values could be chosen).

Currently, as smoother only a relaxed Gauss-Seidel method is implemented (`gs`). The value `smoother->omega` is the relaxation parameter.

The meaning of `pre smoothing steps`, `post smoothing steps` and `coarse level smoothing steps` is explained above.

`min level` is the coarsest multigrid level. By default it is 0.

`min level gap` describes the number of mesh levels that at least are skipped between two multigrid levels. `max mg levels` sets the maximum number of allowed multigrid levels. If necessary, more than `min level gap` levels are skipped, to fulfill this requirement.

A.4.11 Parallelization

Before we start with tha application example, we give a short overview over the parallelization approach of full domain covering meshes used in AMDiS. The approach can be summerized by the following steps:

• Create a partitioning level by some adaptive or global refinements of the macro mesh.

- Create a partitioning with approximately the same number of elements in each partition.

- Every process computes on the whole domain, but adaptive mesh refinements are only allowed within the local partition including some overlap.

- After each adaptive iteration, a new partitioning can be computed if the parallel load balance becomes to bad. Partitionings are always computed at the same mesh level. The partitioning elements are weighted by the number of corresponding leaf elements (elements with no children).

- At the end of each timestep or at the end of parallel computations, a global solution is constructed by a partititon of unity method (weighted sum over all process solutions).

Figure A.24 illustrates the concept of full domain covering meshes.

The so called three level approach allows to decouple the following levels:

1. Partitioning level: Basis for the partitioning. The partitioning level is built by the current leaf elements of the mesh, when the parallelization is initialized.

2. Global coarse grid level: Number of global refinements starting from the partitioning level for the whole domain. The global coarse grid level builds the coarsest level for all future computations.

3. Local coarse grid level: Number of global refinements starting from the partitioning level within the local domain including neighbor elements. The local coarse grid level reduces the size of partition overlap.

In Figure A.25, an example for the three level approach is given. The dashed line shows the overlap of size 1 for the local partition computed at partitining level.

The parallelization in AMDiS uses the *Message Passing Interface* MPI, see

```
http://www-unix.mcs.anl.gov/mpi/ .
```

For the partitioning the parallel graph partitioning library ParMETIS is used, see

```
http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview .
```

To use AMDiS in parallel, it must be configured like

```
> ./configure --prefix=<AMDIS-DIR>
              --with-mpi=<MPI-DIR>
              --with-parmetis=<PARMETIS-DIR> .
```

The parallel application (in this example `parallelheat`) must be compiled with the MPI C++ compiler `mpiCC` and linked against the ParMETIS library. Then the application can be called with `mpirun`:

```
> mpirun -np <num-procs> ./parallelheat init/parallelheat.dat.2d
```

Now we start with the example. We want to solve a time dependent problem as described in A.4.2. As right hand side function f we choose the *moving source*

$$f(x,t) = \sin(\pi t)e^{-10(x-\vec{t})^2} \tag{A.29}$$

with $0 \le t \le 1$ and $\vec{t}$ a vector with all components set to t. The maximum of this function moves from $(0,0)$ to $(1,1)$ within the specified time intervall. We use 4 parallel processes.

partitioning level **global coarse grid level** **local coarse grid level**

█ local partition   ⌐ ⌐ original overlap   ░ final overlap
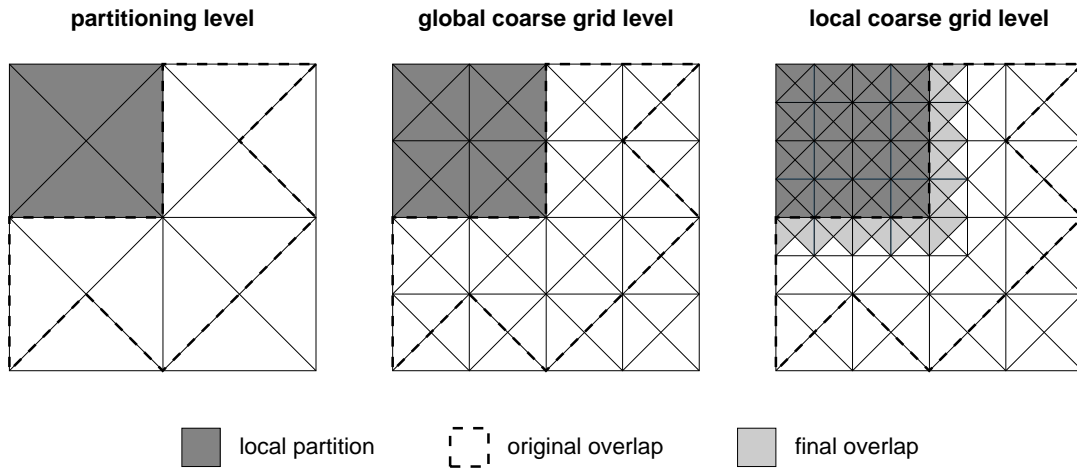
Figure A.25: Example for the three levels of partitioning. The partitioning mesh is globally refined twice to get the global coarse grid level. Then two further global refinement steps applied on Ω_i and its direct neighbor elements (in each step) result in the local coarse grid level.

		1d	2d	3d
source code	src/		parallelheat.cc	
parameter file	init/	–	parallelheat.dat.2d	–
macro file	macro/	–	macro.stand.2d	–
output files	output/	parallelheat_proc<n>_<t>.mesh/.dat		

Table A.12: Files of the parallelheat example. In the output file names, <n> is replaced by the process number and <t> is replaced by the time.

Source code

In this section the interesting aspects of file `parallelheat.cc` are described. First, the moving functions f and g are defined.

```cpp
class G : public AbstractFunction<double, WorldVector<double> >,
         public TimedObject
{
public:
  MEMORY_MANAGED(G);

  /** \brief
   * Implementation of AbstractFunction::operator().
   */
  const double& operator()(const WorldVector<double>& argX) const
  {
    WorldVector<double> x = argX;
    static double result;
    int dim = x.getSize();
    int i;
    for(i = 0; i < dim; i++) {
      x[i] -= *timePtr;
    }
    result = sin(M_PI*(*timePtr)) * exp(-10.0*(x*x));
    return result;
  };
};

class F : public AbstractFunction<double, WorldVector<double> >,
         public TimedObject
{
public:
  MEMORY_MANAGED(F);

  F(int degree) : AbstractFunction<double, WorldVector<double> >(degree) {};

  /** \brief
   * Implementation of AbstractFunction::operator().
   */
  const double& operator()(const WorldVector<double>& argX) const {
    WorldVector<double> x = argX;

    static double result;

    int dim = x.getSize();
    int i;
    for(i = 0; i < dim; i++) {
      x[i] -= *timePtr;
    }

    double r2 = (x*x);
    double ux = sin(M_PI * (*timePtr)) * exp(-10.0*r2);
    double ut = M_PI * cos(M_PI*(*timePtr)) * exp(-10.0*r2);
    result = ut -(400.0*r2 - 20.0*dim)*ux;
    return result;
  };
};
```

The main program starts with `MPI::Init` to initialize MPI, and ends with `MPI::Finalize` to finalize MPI.

```
int main(int argc, char** argv)
{
  MPI::Init(argc, argv);

  ...

  std::vector<DOFVector<double>*> vectors;
  ParallelProblemScal parallelheat("heat->parallel", heatSpace, heat,
                                   vectors);

  AdaptInstationary *adaptInstat =
    NEW AdaptInstationary("heat->adapt",
                          &parallelheat,
                          adaptInfo,
                          &parallelheat,
                          adaptInfoInitial);

  ...

  parallelheat.initParallelization(adaptInfo);
  adaptInstat->adapt();
  parallelheat.exitParallelization(adaptInfo);

  MPI::Finalize();
}
```

The parallel problem `parallelheat` has to know the sequential instationary problem `heat` and the space problem `heatSpace`. The vector `vectors` stores pointers to DOF vectors which are used by the operator terms. The values of these vectors have to be exchanged during repartitioning. The solution of the last time step is considered automatically. Hence, `vectors` here is empty.

The adaptation loop `adaptInstat` uses `parallelheat` as iteration interface and as time interface. Before the adaptation loop starts, the parallelization is initialized by `initParallelization`. After the adaptation loop has finished, the parallelization is finalized by `exitParallelization`.

Parameter file

We use the parameter file described in Section A.4.2 as basis for the file `parallelheat.dat.2d` and describe the relevant changes for the parallel case.

```
heatMesh->global refinements:        6
```

The macro mesh is globally refined 6 times to create the partitioning mesh which is used as basis for the domain decomposition.

In the following, one can see the parameters for the parallel problem `heat->parallel`.

```
heat->parallel->upper part threshold: 1.5
heat->parallel->lower part threshold: 0.66
```

These two parameters determine the upper and lower partitioning thresholds rt_{high} and rt_{low}. If at least one process exceeds the number of $rt_{high} \cdot sum_{av}$ elements or falls below $rt_{low} \cdot sum_{av}$ elements, a repartitioning must be done (sum_{av} is the average number of elements per partition).

```
heat->parallel->global coarse grid level: 0
heat->parallel->local coarse grid level:  4
```
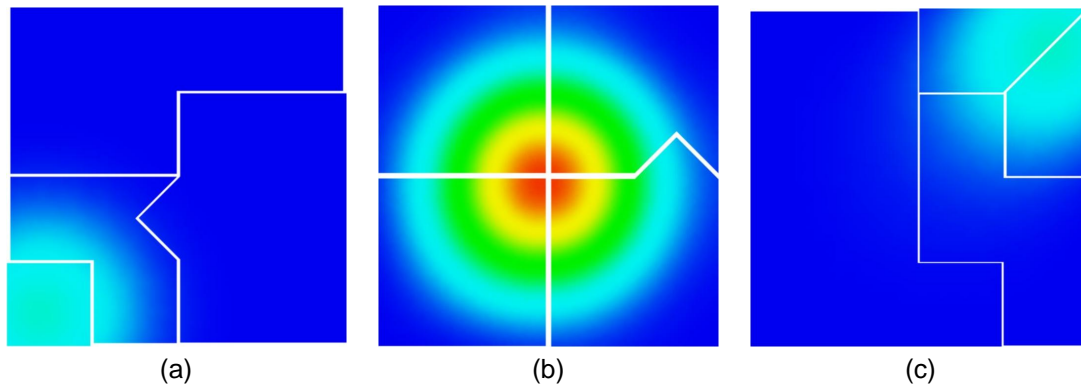
Figure A.26: Local process solutions for $t = 0.1$ (a), $t = 0.5$ (b), and $t = 0.9$ (c).

Here, the parameters used for the three level approach are given. The partitioning level has already been determined by `heatMesh->global refinements`. The value of `global coarse grid level` is set to 0, and `local coarse grid level` is set to 4. This means that 4 global refinements are done within the local partititon at each process (incl. neighoring elements), to reduce the overlap size. No further refinements are done outside of the local partitions.

```
heat -> adapt -> timestep :            0.1
heat -> adapt -> min timestep :        0.1
heat -> adapt -> max timestep :        0.1
heat -> adapt -> start time :          0.0
heat -> adapt -> end time :            1.0
```

We use a fixed timestep of 0.1. The start time is set to 0.0, the end time is set to 1.0. Thus, we have 10 timesteps (without the initial solution at 0.0).

```
heat -> space -> output -> filename :        output / heat
```

For the output the prefix `output/heat` is used. Each process adds its process ID. Then the time is added and finally the file extensions. The result of the initial problem of process 1 will be written to `output/heat_proc0_00.000.mesh` and `output/heat_proc0_00.000.dat`.

Macro file

We again use the macro file `macro/macro.stand.2d`, which was described in Section A.4.1.

Output

For each timestep and for each process, one mesh file and one value file is written (total file number: $11 \cdot 4 \cdot 2 = 88$). In Figure A.26, the local process solutions for $t = 0.1$, $t = 0.5$ and $t = 0.9$ are visualized after the partititon of unity was applied. One can see that the partitioning considers the moving source.

Appendix B

SMI reference

B.1 Installation

B.1.1 Library installation

Change working directory to the SMI directory:

```
> cd <SMI_DIR>
```

Call the configure script to generate the makefiles:

```
> ./configure --prefix=<SMI_DIR> [--enable-debug]
                          [--with-commoncpp=<COMMON_CPP_DIR>]
```

If you add the `--enable-debug` option, debug information will be added and the library will be build without any optimization. The `--with-commoncpp=<COMMON_CPP_DIR>` option is needed, if you want to use SMI in the client-server mode. Common C++ is a GNU project which includes platform independent thread and socket implementation. The Common C++ library can be found at
`http://www.gnu.org/software/commoncpp/`. After creating the makefiles you have to build and install the library:

```
> make install
```

Installations without Common C++

Libraries in `<SMI_DIR>/lib`:

- `libsmm`: Contains the shared mesh manager which does all needed SMI management (non blocking transactions).

- `libsmi`: Delegates all SMI calls directly to the shared mesh manager in `libsmm`.

Installations with Common C++

Libraries in `<SMI_DIR>/lib`:

- `libsmm`: Contains the shared mesh manager which does all needed SMI management (blocking transactions).

- `libsmi`: Delegates all SMI calls directly to the shared mesh manager in `libsmm`.

- `libsmiclient`: Sends all SMI calls per TCP/IP to the SMI server which will execute them.

Executables in `<SMI_DIR>/bin`:

- `smiserver`: Receives the SMI calls from all SMI clients and delegates them to a shared mesh manager.

B.1.2   Compiling and running user programs

You can use SMI in two modes. In the *standard mode*, SMI is directly linked to the user program. Hence, the managed data can not accessed by other programs. In the *client-server mode*, all SMI calls are send by SMI clients to an SMI server. The connection between clients and server is realized by TCP/IP, so several applications at even different machines can connect to the server, and the managed data can be shared between these applications.

The standard mode can be used with or without Common C++. If Common C++ is used, `SMI_Begin_read_transaction()` and `SMI_Begin_write_transaction()` are blocking calls, therefore, they return after the transaction can be started. Otherwise the calls are non-blocking. In this case the functions immediately return with `SMI_OK`, if the transaction could be started, and with `SMI_ERR_TRANSACTION` otherwise. The client-server mode can only be used together with Common C++ and in blocking mode (see Figure B.1).

	with Common C++	**without Common C++**
standard mode	**blocking transactions**	**non blocking transactions**
client–server mode	**blocking transactions**	

Figure B.1: SMI modes with and without Common C++

Standard mode

In the standard mode, SMI is directly linked to the user program as a shared library. In the link step, `-L<SMI_DIR>/lib -lsmi` must be added. If you build your program using *GNU autotools* together with *libtools*, the run path (needed to find the shared library for run time linking) will be set automatically. Otherwise, you must add `<SMI_DIR>/lib` to the `LD_LIBRARY_PATH` variable.

Client-server mode

In the client-server mode the SMI server must be started before any user program. After that user programs can connect to the server by `SMI_Connect_to_server(<hostname>, <portnr>)`. The user programs must be linked against `-L<SMI_DIR>/lib -lsmiclient`. If you build your program using *GNU autotools* together with *libtools* the run path (needed to find the shared library for run time linking) will be set automatically. Otherwise you must add `<SMI_DIR>/lib` to the `LD_LIBRARY_PATH` variable. So if you want to start an SMI client-server session, you must follow these steps:

1. Install Common C++ in `<COMMON_CPP_DIR>`.

2. Call the SMI configure script with the `--with-commoncpp=<COMMON_CPP_DIR>` option.

3. Install SMI calling `make install`.

4. Link your user program against `-L<SMI_DIR>/lib -lsmiclient`.

5. Use `SMI_Connect_to_server(<hostname>, <portnr>)` and `SMI_Disconnect()` in your user program.

6. If you don't use *libtools* add `<SMI_DIR>/lib` to the `LD_LIBRARY_PATH` variable.

7. Start the server: `<SMI_DIR>/bin/smiserver <portnr> [<infoLevel>]`.

8. Start the user programs.

B.2 SMI functions

Every SMI function returns an integer which is used as an error value. If the function returned, successfully `SMI_OK` is returned, otherwise specific error numbers defined in `smi_const.h` are returned.

B.2.1 Client-server functions

The client-server functions are only used in client-server mode. If you call them in standard mode, `SMI_ERR_NOT_IMPLEMENTED` will be returned.

SMI_Connect_to_server

```
int SMI_Connect_to_server(const char    *hostname,
                          unsigned short port)
```

Tries to establish an SMI connection to the SMI server.
Arguments:

- `hostname` (IN): IP address or DNS name of servers host machine

- `port` (IN): Port number the server is listening at.

SMI_Disconnect

```
int SMI_Disconnect();
```

Closes the SMI connection.

B.2.2 Application functions

In SMI, different applications can access the managed data. These applications are internally represented by application ids. The application concept is used to synchronize reading and writing accesses and to log data changes from an applications point of view.

SMI_Add_application

```
int SMI_Add_application(int applicationId,
                        int applicationMode)
```

Adds a new application to SMI.
Arguments:

- `applicationId` (IN): ID of the new application.

- `applicationMode` (IN): Flag for the mode of the new application. `SMI_APP_MODE_DEFAULT` for default mode, `SMI_APP_MODE_CHANGE_LOG` if changes should be logged for this application.

SMI_Remove_application

```
int SMI_Remove_application(int applicationId)
```

Removes an application.
Arguments:

- `applicationId` (IN): ID of the application.

SMI_Get_all_application_ids

```
int SMI_Get_all_application_ids(int    applicationId,
                                int  *numApplicationIds,
                                int **applicationIds)
```

Delivers the IDs of all registered applications.
Arguments:

- `applicationId` (IN): ID of the calling application.

- `numApplicationIds` (OUT): Pointer to an integer where the total number of IDs should be stored.

- `applicationIds` (OUT): Pointer to an integer array where the IDs should be stored. The needed memory is allocated within SMI and it is valid until the next call of this function.

Example:

```
int  numApplicationIds;
int *applicationIds;
int  error = SMI_Get_all_application_ids(1,
                                         &numApplicationIds,
                                         &applicationIds);
```

B.2.3 Mesh functions

Before nodes and elements can be defined on a mesh, the mesh must be registered with a unique mesh ID.

SMI_Add_mesh

Adds a new empty mesh to SMI. The dimension of its node coordinates will be defined by the first call of `SMI_Add_nodes()` for this mesh.

```
int SMI_Add_mesh(int applicationId,
                 int meshId,
                 int meshMode)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the new mesh.

- `meshMode` (IN): `SMI_MESH_MODE_NODE_ELEMS` means that one can access all elements that belong to one node by `SMI_Get_node_elems()` in a fast way. This mode internally leads to additional memory usage. `SMI_MESH_MODE_DEFAULT` means that the mesh is stored in a standard way.

SMI_Get_all_mesh_ids

Delivers the IDs of all managed meshes.

```
int SMI_Get_all_mesh_ids(int    applicationId,
                         int  *numMeshes,
                         int **meshIds)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `numMeshes` (OUT): Pointer to an integer where the number of meshes will be stored.

- `meshIds` (OUT): Pointer to an integer array where the mesh IDs will be stored. The needed memory is allocated within SMI and it is valid until the next call of this function.

SMI_Get_dim_of_coords

Delivers dimension of node coordinates. Before the first call of `SMI_Add_nodes()` for this mesh, the dimension is set to -1 which means that it is still undefined.

```
int SMI_Get_dim_of_coords(int   applicationId,
                          int   meshId,
                          int  *dim)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `dim` (OUT): Pointer to an integer where the dimension will be stored.

SMI_Clear_mesh

Deletes all nodes and elements of the mesh.

```
int SMI_Clear_mesh(int applicationId,
                   int meshId)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

B.2.4 Transaction functions

In SMI, read and write transactions are defined. An application which wants to read or write data for a specific mesh, first has to open a corresponding transaction and close it after the access is finished. When a write transaction is open, no other read or write transaction can be opened. When a read transaction is open, other read transactions can be opened, but no write transaction is allowed.

If Common C++ is used the begin-transaction functions are blocking and they return, after the desired transaction can be established. Otherwise `SMI_ERR_TRANSACTION` will be returned, when a transaction can't be established at the moment (non blocking).

SMI_Begin_write_transaction

Opens a write transaction.

```
int SMI_Begin_write_transaction(int applicationId,
                                int meshId)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

SMI_End_write_transaction

Closes a write transaction.

```
int SMI_End_write_transaction(int applicationId,
                              int meshId)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

SMI_Begin_read_transaction

Opens a read transaction.

```
int SMI_Begin_read_transaction(int applicationId,
                               int meshId)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

SMI_End_read_transaction

Closes a read transaction.

```
int SMI_End_read_transaction(int applicationId,
                             int meshId)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

B.2.5 Synchronization point functions

SMI_Add_sync_point

Adds a synchronization point for the given applications to SMI. If the synchronization point already exists, its application IDs are compared with the here given IDs. If the IDs match, `SMI_OK` is returned.

```
int SMI_Add_sync_point(int   applicationId,
                       int   syncPointId,
                       int   numApplications,
                       int *applicationIds)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `syncPointId` (IN): ID of the synchronization point.

- `numApplications` (IN): Number of applications belonging to the synchronization point.

- `applicationIds` (IN): IDs of the applications.

SMI_Reach_sync_point

Blocks the calling application, until all other needed applications have reached this synchronization point.

```
int SMI_Reach_sync_point(int applicationId,
                         int syncPointId)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `syncPointId` (IN): ID of the synchronization point.

B.2.6 Node functions

An SMI mesh consists of elements which in turn consists of nodes. Thus, before an element can be defined the corresponding nodes must be added. Each node consists of an arbitrary but in this mesh unique node ID and a coordinate vector of given dimension. All nodes within one mesh must be of the same coordinate dimension.

SMI_Add_nodes

Adds new nodes to the mesh.

```
int SMI_Add_nodes(int      applicationId,
                  int      meshId,
                  int      numNodes,
                  int    *indices,
                  double *coords,
                  int      dimOfCoords)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `numNodes` (IN): Number of nodes.

- `indices` (IN): Integer array of size `numNodes` containing the node indices.

- `coords` (IN): Double array of size `numNodes*dimOfCoords` containing all node coordinates. The first `dimOfCoords` entries belong to the first node, and so on.

- `dimOfCoords` (IN): Dimension of node coordinates.

SMI_Remove_nodes

Removes nodes from the mesh. To avoid inconsistent meshes, no nodes should be removed which still belong to at least one element. If `elemCheck` is not zero, this will be checked. If the mesh is stored in node-elem-mode, this check can efficiently be done, otherwise all elements must be searched for the node indices to remove!

```
int SMI_Remove_nodes(int   applicationId,
                     int   meshId,
                     int   numNodes,
                     int  *indices,
                     int   elemCheck)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `numNodes` (IN): Number of nodes.

- `indices` (IN): Integer array of size `numNodes` containing the node indices.

- `elemCheck` (IN): If not zero, only nodes can be removed which don't belong to an element.

SMI_Get_nodes

Delivers coordinates of given nodes.

```
int SMI_Get_nodes(int      applicationId,
                  int      meshId,
                  int      numNodes,
                  int      dimOfCoords,
                  int     *nodeIndices,
                  double  *coords)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `numNodes` (IN): Number of nodes.

- `dimOfCoords` (IN): Dimension of node coordinates.

- `nodeIndices` (IN): Integer array of size `numNodes` containing the node indices.

- `coords` (OUT): Double array of size `numNodes*dimOfCoords` in which the coordinates will be stored. The first `dimOfCoords` entries belong to the first node, and so on. Memory must be allocated by the caller.

SMI_Get_all_nodes

Delivers indices of all nodes of the mesh.

```
int SMI_Get_all_nodes(int    applicationId,
                       int    meshId,
                       int   *numNodes,
                       int  **nodeIndices)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `numNodes` (OUT): Number of nodes.

- `nodeIndices` (OUT): Pointer to an integer array where the node indices will be stored. The needed memory is allocated within SMI and it is valid until the next call of this function.

SMI_Get_changed_nodes

Delivers indices of all nodes which have changed since the last call of this function by this application (only for applications in change-log-mode). Changes which were made before the application was registered to SMI will not occur. Hence, first a call of `SMI_Get_all_nodes()` would be useful.

```
int SMI_Get_changed_nodes(int    applicationId,
                          int    meshId,
                          int   *numNodes,
                          int  **nodeIndices,
                          int  **actions)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `numNodes` (OUT): Number of nodes.

- `nodeIndices` (OUT): Pointer to an integer array where the node indices will be stored. The needed memory is allocated within SMI and it is valid until the next call of this function.

- `actions` (OUT): Pointer to an integer array where the actions will be stored. An action can be of type `SMI_ACTION_ADD` or `SMI_ACTION_REMOVE`. The needed memory is allocated within SMI and it is valid until the next call of this function.

B.2.7 Element functions

Elements consists of nodes. The type of an element specifies how many nodes belong to it. However, no further semantics about an element type are given to SMI.

SMI_Add_elem_type

Adds a new element type for the given mesh.

```
int SMI_Add_elem_type(int applicationId,
                      int meshId,
                      int elemTypeId,
                      int numNodesPerElem)
```

Arguments:

- applicationId (IN): ID of the calling application.

- meshId (IN): ID of the mesh.

- elemTypeId (IN): ID of the element type.

- numNodesPerElem (IN): Number of nodes for this element type.

SMI_Add_elems

Adds elements to the mesh.

```
int SMI_Add_elems(int   applicationId ,
                  int   meshId ,
                  int   numElems ,
                  int  *elemTypes ,
                  int   numNodeIndices ,
                  int  *nodeIndices ,
                  int  *elemIndices )
```

Arguments:

- applicationId (IN): ID of the calling application.

- meshId (IN): ID of the mesh.

- numElems (IN): Number of elements.

- elemTypes (IN): Integer array of size numElems in which the element types are stored.

- numNodeIndices (IN): Size of array nodeIndices.

- nodeIndices (IN): Integer of size nodeIndices storing the node indices for each element.
First, all node indices of the first element are stored, and so on.

- elemIndices (IN): Integer array of size numElems storing the element indices.

SMI_Remove_elems

Removes elements from the mesh.

```
int SMI_Remove_elems(int   applicationId ,
                     int   meshId ,
                     int   numElems ,
                     int  *elemIndices )
```

Arguments:

- applicationId (IN): ID of the calling application.

- meshId (IN): ID of the mesh.

- numElems (IN): Number of elements.

- elemIndices (IN): Integer array of size numElems storing the element indices.

SMI_Get_elems

Delivers node indices and element types of the given elements.

```
int SMI_Get_elems(int    applicationId,
                  int    meshId,
                  int    numElems,
                  int   *elemIndices,
                  int   *numNodeIndices,
                  int  **nodeIndices,
                  int   *elemTypes,
                  int   *numNodesPerElem)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `numElems` (IN): Number of elements.

- `elemIndices` (IN): Integer array of size `numElems` storing the element indices.

- `numNodeIndices` (OUT): Pointer to an integer storing the size of `nodeIndices`. If NULL, no node information will be delivered and `nodeIndices` will be ignored.

- `nodeIndices` (OUT): Pointer to an integer array storing the node indices for each element. First, all node indices of the first element are stored, and so on. The needed memory is allocated within SMI and it is valid until the next call of this function. Can be NULL only if `numNodeIndices` is NULL pointer too.

- `elemTypes` (OUT): Integer array of size `numElems` storing the element type of each element. The needed memory must be allocated by the caller. If NULL, no type information will be delivered.

- `numNodesPerElem` (OUT): Integer array of size `numElems` storing the number of nodes of each element. The needed memory must be allocated by the caller. If NULL, this information won't be delivered.

SMI_Get_node_elems

Delivers all elements containing the given nodes (only for meshes in node-elem-mode).

```
int SMI_Get_node_elems(int    applicationId,
                       int    meshId,
                       int    numNodes,
                       int   *nodeIndices,
                       int   *numElemsPerNode,
                       int  **nodeElems)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `numNodes` (IN): Number of nodes.

- `nodeIndices` (IN): Integer array of size `numNodes` containing the node indices.

- `numElemsPerNode` (OUT): Integer array of size `numNodes` storing the number of elements belonging to each node. The needed memory must be allocated by the caller.

- `nodeElems` (OUT): Pointer to an integer array containing the elements. The first `numElemsPerNode[0]` entries belong to node 0, and so on. The needed memory is allocated within SMI and it is valid until the next call of this function.

SMI_Get_all_elems

Delivers all element indices of this mesh.

```
int SMI_Get_all_elems(int   applicationId,
                      int   meshId,
                      int  *numElems,
                      int **elemIndices)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `numElems` (OUT): Number of elements.

- `elemIndices` (OUT): Pointer to an integer array where the element indices will be stored. The needed memory is allocated within SMI and it is valid until the next call of this function.

SMI_Get_changed_elems

Delivers indices of all elements which have changed since the last call of this function by this application (only for applications in change-log-mode). Changes which were made before the application was registered to SMI will not occur. Thus, first a call of `SMI_Get_all_elems()` would be useful.

```
int SMI_Get_changed_elems(int   applicationId,
                          int   meshId,
                          int  *numElems,
                          int **elemIndices,
                          int **actions)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `numElems` (OUT): Number of elements.

- `elemIndices` (OUT): Pointer to an integer array where the element indices will be stored. The needed memory is allocated within SMI and it is valid until the next call of this function.

- `actions` (OUT): Pointer to an integer array where the actions will be stored. An action can be of type `SMI_ACTION_ADD` or `SMI_ACTION_REMOVE`. The needed memory is allocated within SMI and it is valid until the next call of this function.

SMI_Count_nodes_of_elems

Calculates the sum:

$$numNodes = \sum_{i=0}^{numElems-1} nodesForElementType(elementType(elemIndices[i])) \qquad \text{(B.1)}$$

```
int SMI_Count_nodes_of_elems(int  applicationId,
                             int  meshId,
                             int  numElems,
                             int *elemIndices,
                             int *numNodes)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `numElems` (IN): Number of elements.

- `elemIndices` (IN): array of size `numElems` containing the element indices.

- `numNodes` (OUT): Pointer to an integer where the result will be stored.

B.2.8 Quantity functions

Quantities are used to store mesh related values. Quantities can be located at elements or nodes, or they can be global, which means that they are located at the mesh. Memory for a value is allocated not before this value is explicitly set or retrieved. If a value is retrieved which was not set before, it will be created and initialized with the default value of the corresponding quantity.

SMI_Add_quantity

Adds a new quantity to the mesh.

```
int SMI_Add_quantity(int   applicationId,
                     int   meshId,
                     int   quantityId,
                     int   quantityLoc,
                     int   quantityType,
                     int   quantityDim,
                     void *defaultValue)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `quantityId` (IN): ID of the quantity.

- `quantityLoc` (IN): Quantity location. It can be

  - `SMI_LOCATION_NODE`
  - `SMI_LOCATION_ELEM`
  - `SMI_LOCATION_GLOBAL`

- `quantityType` (IN): Quantity type. It can be

  - `SMI_TYPE_FLOAT`

  - `SMI_TYPE_DOUBLE`

  - `SMI_TYPE_LONG_DOUBLE`

  - `SMI_TYPE_INT`

  - `SMI_TYPE_LONG`

  - `SMI_TYPE_BOOL`

  - `SMI_TYPE_CHAR`

- `quantityDim` (IN): Dimension of the quantity (1 for scalar quantities).

- `defaultValue` (IN): Pointer to the default value. If `quantityDim` is larger than one, it must be an array of the given type of size `quantityDim`. The default value will be copied into SMI, thus, the memory pointed by `defaultValue` can be freed or changed after the call.

SMI_Remove_quantity

Removes a quantity from the mesh.

```
int SMI_Remove_quantity ( int applicationId ,
                          int meshId ,
                          int quantityId )
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `quantityId` (IN): ID of the quantity.

SMI_Get_all_quantity_ids

Delivers all quantity IDs defined on the mesh.

```
int SMI_Get_all_quantity_ids ( int   applicationId ,
                               int   meshId ,
                               int  *numQuantities ,
                               int **quantityIds )
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `numQuantities` (OUT): Number of quantities.

- `quantityIds` (OUT): Pointer to an integer array where the quantity IDs will be stored. The needed memory is allocated within SMI and it is valid until the next call of this function.

SMI_Get_quantity_info

Delivers information about a quantity.

```
int SMI_Get_quantity_info(int   applicationID,
                          int   meshId,
                          int   quantityId,
                          int  *quantityLoc,
                          int  *quantityType,
                          int  *quantityDim)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `quantityId` (IN): ID of the quantity.

- `quantityLoc` (OUT): If not NULL, here the quantity location will be stored.

- `quantityType` (OUT): If not NULL, here the quantity type will be stored.

- `quantityDim` (OUT):If not NULL, here the quantity dimension will be stored.

SMI_Set_quantity_values

Sets values for this quantity at the given indices.

```
int SMI_Set_quantity_values(int    applicationId,
                            int    meshId,
                            int    quantityId,
                            int    quantityType,
                            int    quantityDim,
                            int    numValues,
                            int   *nodeOrElemIndices,
                            void  *values)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `quantityId` (IN): ID of the quantity.

- `quantityType` (IN): Type of the quantity. Must be equal to the type defined in
  `SMI_Add_quantity()`.

- `quantityDim` (IN): Dimension of the quantity. Must be equal to the dimension defined in
  `SMI_Add_quantity()`.

- `numValues` (IN): Number of values.

- `nodeOrElemIndices` (IN): Integer array of size `numValues` containing the node or element
  indices. For global quantities, this argument is ignored.

- `values` (IN): Array of size `numValues*quantityDim` of given type containing the values. The
  first `quantityDim` entries belong to the first value, and so on.

SMI Get quantity values

Delivers values for this quantity at the given indices.

```
int SMI_Get_quantity_values(int   applicationId,
                            int   meshId,
                            int   quantityId,
                            int   quantityType,
                            int   dim,
                            int   numValues,
                            int  *nodeOrElemIndices,
                            void *values)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `quantityId` (IN): ID of the quantity.

- `quantityType` (IN): Type of the quantity. Must be equal to the type defined in `SMI_Add_quantity()`.

- `quantityDim` (IN): Dimension of the quantity. Must be equal to the dimension defined in `SMI_Add_quantity()`.

- `numValues` (IN): Number of values.

- `nodeOrElemIndices` (IN): Integer array of size `numValues` containing the node or element indices. For global quantities, this argument is ignored.

- `values` (OUT): Array of size `numValues*quantityDim` of given type where the values will be stored. The first `quantityDim` entries belong to the first value, and so on. The needed memory must be allocated by the caller.

SMI Get changed quantity values

Provides the node or element indices for all values which have been changed since the last call of this function by this application.

```
int SMI_Get_changed_quantity_values(int    applicationId,
                                    int    meshId,
                                    int    quantityId,
                                    int   *numValues,
                                    int  **nodeOrElemIndices)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `quantityId` (IN): ID of the quantity.

- `numValues` (OUT): Pointer to an integer storing the number of changed values.

- `nodeOrElemIndices` (OUT): Pointer to an array storing the indices of all changed values. The needed memory is allocated within SMI and is valid since the next call of this function.

B.2.9   Relation functions

SMI_Add_relation

Adds a user defined relation to SMI.

```
int SMI_Add_relation(int applicationId,
                     int meshId,
                     int relationId,
                     int numElems,
                     int numNodes)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `relationId` (IN): ID of the relation.

- `numElems` (IN): Number of element indices in each tuple.

- `numNodes` (IN): Number of node indices in each tuple.

SMI_Add_relation_tuples

Adds tuples to the given user defined relation.

```
int SMI_Add_relation_tuples(int  applicationId,
                            int  meshId,
                            int  relationId,
                            int  numTuples,
                            int  numElemsPerTuple,
                            int  numNodesPerTuple,
                            int *elems,
                            int *nodes)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `relationId` (IN): ID of the relation.

- `numTuples` (IN): Number of tuples.

- `numElemsPerTuple` (IN): Number of element indices per tuple.

- `numNodesPerTuple` (IN): Number of node indices per tuple.

- `elems` (IN): Element indices. The first `numElemsPerTuple` indices belong to the first tuple, and so on.

- `nodes` (IN): Node indices. The first `numNodesPerTuple` indices belong to the first tuple, and so on.

SMI_Query_relation_tuples

Retrieves all tuples that match the given query.

```
int SMI_Query_relation_tuples(int    applicationId,
                              int    meshId,
                              int    relationId,
                              int    numQueryTuples,
                              int    numElemsPerTuple,
                              int    numNodesPerTuple,
                              int   *queryElems,
                              int   *queryNodes,
                              int    queryType,
                              int   *numTuples,
                              int  **elems,
                              int  **nodes)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `relationId` (IN): ID of the relation.

- `numQueryTuples` (IN): Number of query tuples.

- `numElemsPerTuple` (IN): Number of element indices per tuple.

- `numNodesPerTuple` (IN): Number of node indices per tuple.

- `queryElems` (IN): Query elements. The first `numElemsPerTuple` indices belong to the first tuple, and so on.

- `queryNodes` (IN): Query nodes. The first `numNodesPerTuple` indices belong to the first tuple, and so on.

- `queryType` (IN): Can be `SMI_QUERY_TYPE_UNION` or `SMI_QUERY_TYPE_INTERSUBSECTION`. This parameter specifies whether the union or the intersubsection of the results of the single query tuples should be built.

- `numTuples` (OUT): Number of result tuples.

- `elems` (OUT): Element indices of the result tuples.

- `nodes` (OUT): Node indices of the result tuples.

B.2.10   Iterator functions

SMI_Add_iterator

Adds a iterator to the mesh.

```
int SMI_Add_iterator(int applicationId,
                     int meshId,
                     int iteratorId,
                     int iteratorType)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `iteratorId` (IN): ID of the iterator.

- `iteratorType` (IN): `SMI_ITERATOR_NODE` for node iterators, `SMI_ITERATOR_ELEM` for element iterators.

SMI_Add_iterator_indices

Adds iterator indices to the iterator.

```
int SMI_Add_iterator_indices(int  applicationId,
                             int  meshId,
                             int  iteratorId,
                             int  numIndices,
                             int *indices)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `iteratorId` (IN): ID of the iterator.

- `numIndices` (IN): Number of indices.

- `indices` (IN): Array of size `numIndices` storing the indices.

SMI_Reset_iterator

Sets the iterator to its first index.

```
int SMI_Reset_iterator(int applicationId,
                       int meshId,
                       int iteratorId)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `iteratorId` (IN): ID of the iterator.

SMI_Clear_iterator

Deletes all indices of this iterator.

```
int SMI_Clear_iterator(int applicationId,
                       int meshId,
                       int iteratorId)
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `iteratorId` (IN): ID of the iterator.

SMI_Get_iterator_indices

Get the next indices of this iterator.

```
int SMI_Get_iterator_indices(int   applicationId ,
                             int   meshId ,
                             int   iteratorId ,
                             int   numIndices ,
                             int  *indices ,
                             int  *numIndicesRead )
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `iteratorId` (IN): ID of the iterator.

- `numIndices` (IN): Number of indices.

- `indices` (OUT): Array of size `numIndices` where the indices will be stored.

- `numIndicesRead` (OUT): Number of indices that was filled into `indices`. If there are less than `numIndices` indices left for iteration, this number is not equal to `numIndices`.

SMI_Get_all_iterator_indices

Retrieves all indices of this iterator at once.

```
int SMI_Get_all_iterator_indices(int    applicationId ,
                                 int    meshId ,
                                 int    iteratorId ,
                                 int   *numIndices ,
                                 int  **indices )
```

Arguments:

- `applicationId` (IN): ID of the calling application.

- `meshId` (IN): ID of the mesh.

- `iteratorId` (IN): ID of the iterator.

- `numIndices` (OUT): Number of indices.

- `indices` (OUT): Array of size `numIndices` where the indices will be stored.

Bibliography

[1] I. Babuska and J. M. Melenk. The partition of unity method. *Internat. J. Numer. Methods Engrg.*, 40:727–758, 1997.

[2] R. Backofen, A. Rätz, and A. Voigt. Nucleation and growth by a phase field crystal (pfc) model. *Phil. Mag. Lett.*, 87(11):813–820, 2007.

[3] R. Backofen, A. Rätz, and A. Voigt. Nucleation and growth in a phase field crystal (PFC) model. *Phil. Mag.*, accepted.

[4] K. Banas. On a modular architecture for finite element systems. i sequential codes. *Comput. Vis. Sci.*, 8:35–47, 2005.

[5] R. E. Bank and M. Holst. A new paradigm for parallel adaptive meshing algorithms. *SIAM Rev.*, 45(2):291–323, 2003.

[6] E. Bänsch, F. Haußer, O. Lakkis, B. Li, and A. Voigt. Finite element method for epitaxial growth with attachment-detachment kinetics. *J. Comput. Phys.*, 194:409–434, 2004.

[7] E. Bänsch, P. Morin, and R. H. Nochetto. A finite element method for surface diffusion: the parametric case. *J. Comput. Phys.*, 203:321–343, 2005.

[8] E. Bänsch, P. Morin, and R.H. Nochetto. A finite element method for surface diffusion: the parametric case. *J. Comput. Phys.*, 203:321–343, 2005.

[9] E. Bänsch and K. G. Siebert. A posteriori error estimation for nonlinear problems by duality techniques. *Preprint*, 1995.

[10] J.W. Barrett and J.F. Blowey. Finite element approximation of the Cahn-Hilliard equation with concentration dependent mobility. *Math. Comp.*, 68:487–517, 1999.

[11] P. Bastian, M. Droske, C. Engwer, R. Klöfkorn, T. Neubauer, M. Ohlberger, and M. Rumpf. Towards a unified framework for scientific computing. In R. Kornhuber, R.H.W. Hoppe, D.E. Keyes, J. Periaux, O. Pironneau, and J. Xu, editors, *Proceedings of the 15th Conference on Domain Decomposition Methods*, LNCSE. Springer-Verlag, 2005. accepted for publication.

[12] F. Bornemann and C. Rasch. Finite-element discretization of static Hamilton-Jacobi equations based on a local variational principle. *Comput. Vis. Sci.*, 2005.

[13] D. Braess. *Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics*. Springer, 1997.

[14] P. G. Ciarlet. *The Finite Element Method for Elliptic Problems*, volume 40 of *Classics in Applied Mathematics*. SIAM, 2002.

[15] U. Clarenz, G. Dziuk, and M. Rumpf. On generalized mean curvature flow in surface processing. In *Geometric analysis and nonlinear partial differential equations*, pages 217–248. Springer, Berlin, 2003.

[16] U. Clarenz, F. Haußer, M. Rumpf, A. Voigt, and U. Weickard. On level set formulations for anisotropic mean curvature flow and surface diffusion. In *Multiscale Modeling in epitaxial growth*, volume 149 of *ISNM*, pages 227–237. Birkhäuser, 2005.

[17] K. Deckelnick, G. Dziuk, and C.M.Elliott. Computation of geometric partial differential equations and mean curvature flow. *Acta Numerica*, 2005.

[18] W. Dörfler. A convergent adaptive algorithm for poisson's equation. *SIAM Journal on Numerical Analysis*, pages 1106–1124, 1996.

[19] X.B. Feng and A. Prohl. Numerical analysis of the Cahn-Hilliard equation and approximation for the Hele-Shaw problem. *Interf. Free Bound.*, 7(1):1–28, 2005.

[20] M. Fowler and K. Scott. *UML distilled. A brief guide to the standard object modeling language.* Addison Wesley, 1999.

[21] E. Fried and M.E. Gurtin. A unified treatment of evolving interfaces accounting for small deformations and atomic transport with emphasis on grain-boundaries and epitaxy. *Adv. Appl. Mech.*, 40:1–177, 2004.

[22] M. Fried. A level set based finite element algorithm for the simulation of dendritic growth. *Comput. Vis. Sci.*, 7(2):97–110, 2004.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[24] W. Hackbusch and S.A. Sauter. Composite finite elements for the approximation of pdes on domains with complicated micro-structures. *Numer. Math.*, 75:447–472, 1997.

[25] D. Halpern, O.E. Jensen, and J.B. Grotberg. A theoretical study of surfactant and liquid delivery into the lung. *J. Appl. Physiology*, 85:333–352, 1998.

[26] F. Haußer and A. Voigt. Anisotropic surface diffusion, a numerical approach by parametric finite elements. *J. Sci. Comput.*, 2006.

[27] F. Haußer and A. Voigt. A discrete scheme for parametric anisotropic surface diffusion. *J. Sci. Comp.*, 30:223–235, 2007.

[28] M. Holst. Applications of domain decomposition and partition of unity methods in physics and geometry. *Proceedings of the Fourteenth International Conference on Domain Decomposition Methods*, pages 63–78, 2002.

[29] J. Kim, K.K. Kang, and J. Lowengrub. Conservative multigrid methods for Cahn-Hilliard fluids. *J. Comput. Phys.*, 193(2):511–543, 2004.

[30] A. Meister. *Numerik linearer Gleichungssysteme.* vieweg, 1999.

[31] F. Memoli, G. Sapiro, and P. Thompson. Implicit brain imaging. *Human Brain Mapping*, 23:179–188, 2004.

[32] W. F. Mitchell. Parallel adaptive multilevel methods with full domain partitions. *App. Num. Anal. and Comp. Math.*, 1:36–48, 2004.

[33] T.G. Myers and J.P.F. Charpin. A mathematical model for atmosheric ice accretion and water flow on a cold surface. *Int. J. Heat Mass Trans.*, 47(25):5483–5500, 2004.

[34] S. Osher and R. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces.* Springer, 2003.

[35] S. Osher and J.A. Sethian. Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulation. *J. Comput. Phys.*, 79:12–49, 1988.

[36] A. Rätz. *Modelling and Numerical Treatment of Diffuse-Interface Models with Applications in Epitaxial Growth*. PhD thesis, Universität Bonn, Mathematisch-Naturwissenschaftliche Fakultät, Bonn, Germany, 2007.

[37] A. Rätz, A. Ribalta, and A. Voigt. Surface evolution of elastically stressed films under deposition. *J. Comput. Phys.*, 2006.

[38] A. Rätz, A. Ribalta, and A. Voigt. Surface evolution of elastically stressed films under deposition by a diffuse interface model. *J. Comput. Phys.*, 214(1):187–208, 2006.

[39] A. Rätz and A. Voigt. A diffuse-interface approximation for surface diffusion including adatoms. *Nonlinearity*, 20(1):177–192, 2007.

[40] A. Ribalta, C. Stöcker, S. Vey, and A.Voigt. Amdis - adaptive multidimensional simulations: parallel concepts. In *Proceedings of the 17th International Conference on Domain Decomposition Methods*, in press, 2006.

[41] A. Ribalta, S. Vey, and A. Voigt. Error reduction in adaptive full domain covering meshes for parallel computing. *Num. Math.*, in review.

[42] G. Sapiro. *Geometric partial differential equations and image analysis*. Cambridge Univ. Press, 2001.

[43] K. Schloegel, G. Karypis, and V. Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14:219–240, 2002.

[44] A. Schmidt and K.G. Siebert. *Design of Adaptive Finite Element Software*, volume 42 of *LNCSE*. Springer, 2005.

[45] J.A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge Uni. Press, 1999.

[46] P. Smereka. A level set based finite element algorithm for the simulation of dendritic growth. *J. Sci. Comput.*, 19:439–456, 2003.

[47] C. Stöcker, S. Vey, and A. Voigt. AMDiS-adaptive multidimensional simulation: composite finite elements and signed distance functions. *WSEAS Trans. Circ. Syst.*, 4:111–116, 2005.

[48] C. Stöcker and A. Voigt. The effect of kinetics in the surface evolution in thin crystalline films. *J. of Crystal Growth*, 303(1):90–94, 2007.

[49] C. Stöcker and A. Voigt. A level set approach to anisotropic surface evolution with free adatoms. *SIAM J. on Applied Mathematics*, accepted, 2007.

[50] C. Stöcker and A. Voigt. Geodesic evolution laws - a level set approach. *SIAM J. on Imaging Sciences*, submitted, 2007.

[51] U. Trottenberg, C. Oosterlee, and A. Schüler. *Multigrid*. Elsevier Academic Press, 2001.

[52] G. Turk and M. Levoy. Zipped polygon meshes from range images. In *SIGGRAPH'94*, pages 311–318, 1994.

[53] R. Verfürth. A posteriori error estimates for nonlinear problems: Finite element discretization of elliptic equations. *Math. Comp.*, pages 445–475, 1994.

[54] S. Vey and A.Voigt. Adaptive full domain covering meshes for parallel finite element computations. *Computing*, 81:53–75, 2007.

[55] S. Vey and A. Voigt. Amdis - adapive multidimensional simulations: Object oriented software concepts for scientific computing. *WSEAS Trans. Circ. Syst.*, 3:1564–1569, 2004.

[56] S. Vey and A. Voigt. Amdis - adaptive multidimensional simulations. *Computing and Visualization in Science*, 10:57–67, 2007.

Acknowledgements

Versicherung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

Diese Arbeit wurde am Forschungszentrum caesar in Bonn unter der wissenschaftlichen Betreuung von Prof. Dr. Axel Voigt, Direktor des Institutes für Wissenschaftliches Rechnen der Technischen Universität Dresden, angefertigt. Ich bestätige, dass ich die Promotionsordnung der Fakultät Mathematik und Naturwissenschaften der Technischen Universität Dresden vom 20. März 2000 anerkenne.

Bonn, 23. November 2007