Design of adaptive finite element software: The finite element toolbox ALBERTA

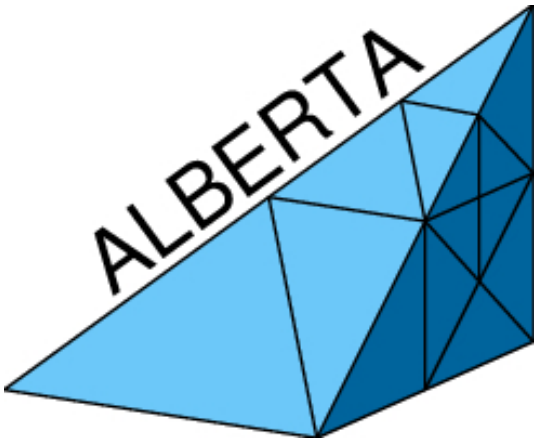
Alfred Schmidt
Zentrum fuer Technomathematik
Universität Bremen
Bibliothekstr. 2
D-28359 Bremen, Germany

Kunibert G. Siebert
Institut für Mathematik
Universität Augsburg
Universitätstr. 14
D-86159 Augsburg, Germany

Daniel Köster
Department of Applied Mathematics
University of Twente
P.O. Box 217
7500AE Enschede, The Netherlands

Claus-Justus Heine
Abteilung für Angewandte Mathematik
Albert-Ludwigs-Universität Freiburg
Hermann-Herder-Str. 10
D-79104 Freiburg im Breisgau, Germany

http://www.alberta-fem.de



ALBERTA is an Adaptive multi-Level finite element toolbox using Bisectioning refinement and Error control by Residual Techniques for scientific Applications.

Version: ALBERTA-2.0, November 12, 2018

Preface

During the last years, scientific computing has become an important research branch located between applied mathematics and applied sciences and engineering. Nowadays, in numerical mathematics not only simple model problems are treated, but modern and well-founded mathematical algorithms are applied to solve complex problems of real life applications. Such applications are demanding for computational realization and need suitable and robust tools for a flexible and efficient implementation. Modularity and abstract concepts allow for an easy transfer of methods to different applications.

Inspired by and parallel to the investigation of real life applications, numerical mathematics has built and improved many modern algorithms which are now standard tools in scientific computing. Examples are adaptive methods, higher order discretizations, fast linear and non-linear iterative solvers, multi-level algorithms, etc. These mathematical tools are able to reduce computing times tremendously and for many applications a simulation can only be realized in a reasonable timeframe using such highly efficient algorithms.

A very flexible software is needed when working in both fields of scientific computing and numerical mathematics. We developed the toolbox ALBERTA[1] for meeting these requirements. Our intention in the design of ALBERTA is threefold: First, it is a toolbox for fast and flexible implementation of efficient software for real life applications, based on the modern algorithms mentioned above. Secondly, in an interplay with mathematical analysis, ALBERTA is an environment for improving existent, or developing new numerical methods. And finally, it allows the direct integration of such new or improved methods in existing simulation software.

Before having ALBERTA, we worked with a variety of solvers, each designed for the solution of one single application. Most of them were based on data structures specifically designed for one single application. A combination of different solvers or exchanging modules between programs was hard to do. Facing these problems, we wanted to develop a general adaptive finite element environment, open for implementing a large class of applications, where an exchange of modules and a coupling of different solvers is easy to realize.

Such a toolbox has to be based on a solid concept which is still open for extensions as science develops. Such a solid concept can be derived from a mathematical abstraction of problem classes, numerical methods, and solvers. Our mathematical view of numerical algorithms, especially finite element methods, is based on our education and scientific research in the departments for applied mathematics at the universities of Bonn and Freiburg. This view point has greatly inspired the abstract concepts of ALBERTA as well as their practical realization, reflected in the main data structures. The robustness and flexible extensibility of our concept was approved in various applications from physics and engineering, like computational fluid dynamics, structural mechanics, industrial crystal growth, etc. as well as by the validation of new mathematical methods.

ALBERTA is a library with data structures and functions for adaptive finite element simulations in one, two, and three space dimension, written in the programming language ANSI-C. Shortly after finishing the implementation of the first version of ALBERTA and using it for first scientific applications, we confronted students with it in a course about finite element methods. The idea was to work on more interesting projects in the course and providing a strong foundation for an upcoming diploma thesis. Using ALBERTA in education then re-

[1] The original name of the toolbox was ALBERT. Due to copyright reasons, we had to rename it and we have chosen ALBERTA.

quired a documentation of data structures and functions. The numerical course tutorials were the basis for a description of the background and concepts of adaptive finite elements.

The combination of the abstract and concrete description resulted in a manual for ALBERTA and made it possible that it is now used world wide in universities and research centers. The interest from other scientists motivated a further polishing of the manual as well as the toolbox itself, and resulted in this book, which is accompanied by a full distribution of ALBERTA on a CD.

Starting first as a two-men-project, ALBERTA is evolving and now there are more people maintaining and extending it. We are grateful for a lot of substantial contributions coming from: Michael Fried, who was the first brave man besides us to use ALBERT, Claus-Justus Heine, Daniel Köster, and Oliver Kriessl. Daniel and Claus in particular set up the GNU configure tools for an easy, platform-independent installation of the software.

We are indebted to the authors of the gltools, especially Jürgen Fuhrmann, and also to the developers of GRAPE, especially Bernard Haasdonk, Robert Klöfkorn, Mario Ohlberger, and Martin Rumpf.

We want to thank the Department of Mathematics at the University of Maryland (USA), in particular Ricardo H. Nochetto, where part of the documentation was written during a visit of the second author. We appreciate the invitation of the Isaac Newton Institute in Cambridge (UK) where we could meet and work intensively on the revision of the manual for three weeks.

We thank our friends, distributed all over the world, who have pointed out a lot of typos in the manual and suggested several improvements for ALBERTA.

Last but not least, ALBERTA would not have come into being without the stimulating atmosphere in the group in Freiburg, which was the perfect environment for working on this project. We want to express our gratitude to all former colleagues, especially Gerhard Dziuk.

Bremen and Augsburg, December 2003

Alfred Schmidt and Kunibert G. Siebert

The book is organized as follows: In Chapter 1 we describe the concepts of adaptive finite element methods and its ingredients like the domain discretization, finite element basis functions and degrees of freedom, numerical integration via quadrature formulas for the assemblage of discrete systems, and adaptive algorithms.

The second chapter is a tutorial for using ALBERTA without giving much details about data structures and functions. The implementation of three model problems is presented and explained. We start with the easy and straight forward implementation of the Poisson problem to learn about the basics of ALBERTA . The examples with the implementation of a nonlinear reaction-diffusion problem and the time dependent heat equation are more involved and show the tools of ALBERTA for attacking more complex problems. The chapter is closed with a short introduction to the installation of the ALBERTA-distribution enclosed to this book in a UNIX/Linux environment.

The realization of data structures and functions in ALBERTA is based on the abstract concepts presented in Chapter 1. A detailed description of all data structures and functions of ALBERTA is given in Chapter 3. The book closes with separate lists of all data types, symbolic constants, functions, and macros.

Tentative: The cover pictures show the ALBERTA logo, cogwheel [16], flute [4], and minimal surface [30], and perhaps also Bridgman crystal growth [21, 36].

Visit the ALBERTA web site, `www.alberta-fem.de`, for more information, FAQ, contributions, updates, pictures from different projects, etc.

Contents

Introduction

Finite element methods provide a widely used tool for the solution of problems with an underlying variational structure. Modern numerical analysis and implementations for finite elements provide more and more tools for the efficient solution of large-scale applications. Efficiency can be increased by using local mesh adaption, by using higher order elements, where applicable, and by fast solvers.

Adaptive procedures for the numerical solution of partial differential equations started in the late 70's and are now standard tools in science and engineering. Adaptive finite element methods are a meaningful approach for handling multi scale phenomena and making realistic computations feasible, specially in 3d.

There exists a vast variety of books about finite elements. Here, we only want to mention the books by Ciarlet [24], and Brenner and Scott [22] as the most prominent ones. The book by Brenner and Scott also contains an introduction to multi-level methods.

The situation is completely different for books about adaptive finite elements. Only few books can be found with introductory material about the mathematics of adaptive finite element methods, like the books by Verfürth [57], and Ainsworth and Oden [2]. Material about more practical issues like adaptive techniques and refinement procedures can for example be found in [3, 5, 7, 35, 37, 38].

Another basic ingredient for an adaptive finite element method is the a posteriori error estimator which is the main object of interest in the analysis of adaptive methods. While a general theory exists for these estimators in the case of linear and mildly nonlinear problems [9, 57], highly nonlinear problems usually still need a special treatment, see [23, 30, 44, 45, 54] for instance. There exist a lot of different approaches to (and a large number of articles about) the derivation of error estimates, by residual techniques, dual techniques, solution of local problems, hierarchical approaches, etc., a fairly incomplete list of references is [1, 3, 6, 12, 20, 31, 43, 56].

Although adaptive finite element methods in practice construct a sequence of discrete solutions which converge to the true solution, this convergence could only be proved recently for linear elliptic problem [41, 42, 43] and for the nonlinear Laplacian [55], based on the fundamental paper [28]. For a modification of the convergent algorithm in [41], quasi-optimality of the adaptive method was proved in [15] and [53].

During the last years there has been a great progress in designing finite element software. It is not possible to mention all freely available packages. Examples are [5, 10, 11, 40, 49], and an continuously updated list of other available finite element codes and resources can for instance be found at

www.engr.usask.ca/~macphed/finite/fe_resources/.

Adaptive finite element methods and basic concepts of ALBERTA

Finite element methods calculate approximations to the true solution in some finite dimensional function space. This space is built from *local function spaces*, usually polynomials of low order, on elements of a partitioning of the domain (the *mesh*). An adaptive method adjusts this mesh (or the local function space, or both) to the solution of the problem. This adaptation is based on information extracted from *a posteriori error estimators*.

The basic iteration of an adaptive finite element code for a stationary problem is

- assemble and solve the discrete system;

- calculate the error estimate;
- adapt the mesh, when needed.

For time dependent problems, such an iteration is used in each time step, and the step size of a time discretization may be subject to adaptivity, too.

The core part of every finite element program is the problem dependent assembly and solution of the discretized problem. This holds for programs that solve the discrete problem on a fixed mesh as well as for adaptive methods that automatically adjust the underlying mesh to the actual problem and solution. In the adaptive iteration, the assemblage and solution of a discrete system is necessary after each mesh change. Additionally, this step is usually the most time consuming part of that iteration.

A general finite element toolbox must provide flexibility in problems and finite element spaces while on the other hand this core part can be performed efficiently. Data structures are needed which allow an easy and efficient implementation of the problem dependent parts and also allow the use of adaptive methods, mesh modification algorithms, and fast solvers for linear and nonlinear discrete problems by calling library routines. On one hand, large flexibility is needed in order to choose various kinds of finite element spaces, with higher order elements or combinations of different spaces for mixed methods or systems. On the other hand, the solution of the resulting discrete systems may profit enormously from a simple vector–oriented storage of coefficient vectors and matrices. This also allows the use of optimized solver and BLAS libraries. Additionally, multilevel preconditioners and solvers may profit from hierarchy information, leading to highly efficient solvers for the linear (sub–) problems.

ALBERTA [46, 47, 49] provides all those tools mentioned above for the efficient implementation and adaptive solution of general nonlinear problems in one, two, or three space dimensions. The design of the ALBERTA data structures allows a dimension independent implementation of problem dependent parts. The mesh adaptation is done by local refinement and coarsening of mesh elements, while the same local function space is used on all mesh elements.

Starting point for the design of ALBERTA data structures is the abstract concept of a finite element space defined (similar to the definition of a single finite element by Ciarlet [24]) as a triple consisting of

- a collection of *mesh elements*;
- a set of local *basis functions* on a single element, usually a restriction of global basis functions to a single element;
- a connection of local and global basis functions giving global *degrees of freedom* for a finite element function.

This directly leads to the definition of three main groups of data structures:

- data structures for geometric information storing the underlying mesh together with element coordinates, boundary type and geometry, etc.;
- data structures for finite element information providing values of local basis functions and their derivatives;
- data structures for algebraic information linking geometric data and finite element data.

Using these data structures, the finite element toolbox ALBERTA provides the whole abstract framework like finite element spaces and adaptive strategies, together with hierarchical meshes, routines for mesh adaptation, and the complete administration of finite element spaces

and the corresponding degrees of freedom (DOFs) during mesh modifications. The underlying data structures allow a flexible handling of such information. Furthermore, tools for numerical quadrature, matrix and load vector assembly as well as solvers for (linear) problems, like conjugate gradient methods, are available.

A specific problem can be implemented and solved by providing just some problem dependent routines for evaluation of the (linearized) differential operator, data, nonlinear solver, and (local) error estimators, using all the tools above mentioned from a library.

Both geometric and finite element information strongly depend on the space dimension. Thus, mesh modification algorithms and basis functions are implemented for one (1d), two (2d), and three (3d) dimensions separately and are provided by the toolbox. Everything besides that can be formulated in such a way that the dimension only enters as a parameter (like size of local coordinate vectors, e.g.). For usual finite element applications this results in a dimension independent programming, where all dimension dependent parts are hidden in a library. This allows a dimension independent programming of applications to the greatest possible extent.

The remaining parts of the introduction give a short overview over the main concepts, details are then given in Chapter 1.

The hierarchical mesh

The underlying mesh is a conforming triangulation of the computational domain into simplices, i.e. intervals (1d), triangles (2d), or tetrahedra (3d). The simplicial mesh is generated by refinement of a given initial triangulation. Refined parts of the mesh can be de–refined, but elements of the initial triangulation (*macro elements*) must not be coarsened. The refinement and coarsening routines construct a sequence of nested meshes with a hierarchical structure. In ALBERTA, the recursive refinement by bisection is implemented, using the notation of Kossaczký [35].

During refinement, new degrees of freedom are created. A single degree of freedom is shared by all elements which belong to the support of the corresponding finite element basis function (compare next paragraph). The mesh refinement routines must create a new DOF only once and give access to this DOF from all elements sharing it. Similarly, DOFs are handled during coarsening. This is done in cooperation with the DOF administration tool, see below.

The bisectioning refinement of elements leads naturally to nested meshes with the hierarchical structure of binary trees, one tree for every element of the initial triangulation. Every interior node of that tree has two pointers to the two children; the leaf elements are part of the actual triangulation, which is used to define the finite element space(s). The whole triangulation is a list of given macro elements together with the associated binary trees. The hierarchical structure allows the generation of most information by the hierarchy, which reduces the amount of data to be stored. Some information is stored on the (leaf) elements explicitly, other information is located at the macro elements and is transferred to the leaf elements while traversing through the binary tree. Element information about vertex coordinates, domain boundaries, and element adjacency can be computed easily and very fast from the hierarchy, when needed. Data stored explicitly at tree elements can be reduced to pointers to the two possible children and information about local DOFs (for leaf elements). Furthermore, the hierarchical mesh structure directly leads to multilevel information which can be used by multilevel preconditioners and solvers.

Access to mesh elements is available solely via routines which traverse the hierarchical trees; no direct access is possible. The traversal routines can give access to all tree elements, only to leaf elements, or to all elements which belong to a single hierarchy level (for a multilevel application, e.g.). In order to perform operations on visited elements, the traversal routines call a subroutine which is given to them as a parameter. Only such element information which is needed by the current operation is generated during the tree traversal.

Finite elements

The values of a finite element function or the values of its derivatives are uniquely defined by the values of its DOFs and the values of the basis functions or the derivatives of the basis functions connected with these DOFs. We follow the concept of finite elements which are given on a single element S in local coordinates: Finite element functions on an element S are defined by a finite dimensional function space $\bar{\mathbb{P}}$ on a reference element $\bar{S}$ and the (one to one) mapping $\lambda^S : \bar{S} \to S$ from the reference element $\bar{S}$ to the element S. In this situation the non vanishing basis functions on an arbitrary element are given by the set of basis functions of $\bar{\mathbb{P}}$ in local coordinates λ^S. Also, derivatives are given by the derivatives of basis functions on $\bar{\mathbb{P}}$ and derivatives of λ^S.

Each local basis function on S is uniquely connected to a global degree of freedom, which can be accessed from S via the DOF administration tool. ALBERTA supports basis functions connected with DOFs, which are located at vertices of elements, at edges, at faces (in 3d), or in the interior of elements. DOFs at a vertex are shared by all elements which meet at this vertex, DOFs at an edge or face are shared by all elements which contain this edge or face, and DOFs inside an element are not shared with any other element. The support of the basis function connected with a DOF is the patch of all elements sharing this DOF.

For a very general approach, we only need a vector of the basis functions (and its derivatives) on $\bar{S}$ and a function for the communication with the DOF administration tool in order to access the degrees of freedom connected to local basis functions. By such information every finite element function (and its derivatives) is uniquely described on every element of the mesh.

During mesh modifications, finite element functions must be transformed to the new finite element space. For example, a discrete solution on the old mesh yields a good initial guess for an iterative solver and a smaller number of iterations for a solution of the discrete problem on the new mesh. Usually, these transformations can be realized by a sequence of local operations. Local interpolations and restrictions during refinement and coarsening of elements depend on the function space $\bar{\mathbb{P}}$ and the refinement of $\bar{S}$ only. Thus, the subroutine for interpolation during an atomic mesh refinement is the efficient implementation of the representation of coarse grid functions by fine grid functions on $\bar{S}$ and its refinement. A restriction during coarsening is implemented using similar information.

Lagrange finite element spaces up to order four are currently implemented in one, two, and three dimensions. This includes the communication with the DOF administration as well as the interpolation and restriction routines.

Degrees of freedom

Degrees of freedom (DOFs) connect finite element data with geometric information of a triangulation. For general applications, it is necessary to handle several different sets of degrees

of freedom on the same triangulation. For example, in mixed finite element methods for the Navier-Stokes problem, different polynomial degrees are used for discrete velocity and pressure functions.

During adaptive refinement and coarsening of a triangulation, not only elements of the mesh are created and deleted, but also degrees of freedom together with them. The geometry is handled dynamically in a hierarchical binary tree structure, using pointers from parent elements to their children. For data corresponding to DOFs, which are usually involved with matrix–vector operations, simpler storage and access methods are more efficient. For that reason every DOF is realized just as an integer index, which can easily be used to access data from a vector or to build matrices that operate on vectors of DOF data. This results in a very efficient access during matrix/vector operations and in the possibility to use libraries for the solution of linear systems with a sparse system matrix ([26], e.g.).

Using this realization of DOFs two major problems arise:

- During refinement of the mesh, new DOFs are added, and additional indices are needed. The total range of used indices has to be enlarged. At the same time, all vectors and matrices that use these DOF indices have to be adjusted in size, too.

- During coarsening of the mesh, DOFs are deleted. In general, the deleted DOF is not the one which corresponds to the largest integer index. Holes with unused indices appear in the total range of used indices and one has to keep track of all used and unused indices.

These problems are solved by a general DOF administration tool. During refinement, it enlarges the ranges of indices, if no unused indices produced by a previous coarsening are available. During coarsening, a book–keeping about used and unused indices is done. In order to reestablish a contiguous range of used indices, a compression of DOFs can be performed; all DOFs are renumbered such that all unused indices are shifted to the end of the index range, thus removing holes of unused indices. Additionally, all vectors and matrices connected to these DOFs are adjusted correspondingly. After this process, vectors do not contain holes anymore and standard operations like BLAS1 routines can be applied and yield optimal performance.

In many cases, information stored in DOF vectors has to be adjusted to the new distribution of DOFs during mesh refinement and coarsening. Each DOF vector can provide pointers to subroutines that implement these operations on data (which usually strongly depend on the corresponding finite element basis). Providing such a pointer, a DOF vector will automatically be transformed during mesh modifications.

All tasks of the DOF administration are performed automatically during refinement and coarsening for every kind and combination of finite elements defined on the mesh.

Adaptive solution of the discrete problem

The aim of adaptive methods is the generation of a mesh which is adapted to the problem such that a given criterion, like a tolerance for the estimated error between exact and discrete solution, is fulfilled by the finite element solution on this mesh. An optimal mesh should be as coarse as possible while meeting the criterion, in order to save computing time and memory requirements. For time dependent problems, such an adaptive method may include mesh changes in each time step and control of time step sizes. The philosophy implemented in ALBERTA is to change meshes successively by local refinement or coarsening, based on error

estimators or error indicators, which are computed a posteriori from the discrete solution and given data on the current mesh.

Several adaptive strategies are proposed in the literature, that give criteria which mesh elements should be marked for refinement. All strategies are based on the idea of an equidistribution of the local error to all mesh elements. Babuška and Rheinboldt [3] motivate that for stationary problems a mesh is almost optimal when the local errors are approximately equal for all elements. So, elements where the error indicator is large will be marked for refinement, while elements with a small estimated indicator are left unchanged or are marked for coarsening. In time dependent problems, the mesh is adapted to the solution in every time step using a posteriori information like in the stationary case. As a first mesh for the new time step we use the adaptive mesh from the previous time step. Usually, only few iterations of the adaptive procedure are then needed for the adaptation of the mesh for the new time step. This may be accompanied by an adaptive control of time step sizes.

Given pointers to the problem dependent routines for assembling and solution of the discrete problems, as well as an error estimator/indicator, the adaptive method for finding a solution on a quasi–optimal mesh can be performed as a black–box algorithm. The problem dependent routines are used for the calculation of discrete solutions on the current mesh and (local) error estimates. Here, the problem dependent routines heavily make use of library tools for assembling system matrices and right hand sides for an arbitrary finite element space, as well as tools for the solution of linear or nonlinear discrete problems. On the other hand, any specialized algorithm may be added if needed. The marking of mesh elements is based on general refinement and coarsening strategies relying on the local error indicators. During the following mesh modification step, DOF vectors are transformed automatically to the new finite element spaces as described in the previous paragraphs.

Dimension independent program development

Using black–box algorithms, the abstract definition of basis functions, quadrature formulas and the DOF administration tool, only few parts of the finite element code depend on the dimension. Usually, all dimension dependent parts are hidden in the library. Hence, program development can be done in 1d or 2d, where execution is usually much faster and debugging is much easier (because of simple 1d and 2d visualization, e.g., which is much more involved in 3d). With no (or maybe few) additional changes, the program will then also work in 3d. This approach leads to a tremendous reduction of program development time for 3d problems.

Notations. For a differentiable function $f\colon \Omega \to \mathbb{R}$ on a domain $\Omega \subset \mathbb{R}^d$, $d = 1, 2, 3$, we set

$$\nabla f(x) = (f_{,x_1}(x), \dots, f_{,x_d}(x)) = \left(\frac{\partial}{\partial x_1} f(x), \dots, \frac{\partial}{\partial x_d} f(x) \right)$$

and

$$D^2 f(x) = (f_{,x_k x_l})_{k,l=1,\dots d} = \left(\frac{\partial^2}{\partial x_k x_l} f(x) \right)_{k,l=1,\dots d}.$$

In the case of a vector valued, differentiable function $f = (f_1, \dots, f_n)\colon \Omega \to \mathbb{R}^n$ we write

$$\nabla f(x) = (f_{i,x_1}(x), \dots, f_{i,x_d}(x))_{i=1,\dots,n} = \left(\frac{\partial}{\partial x_1} f_i(x), \dots, \frac{\partial}{\partial x_d} f_i(x) \right)_{i=1,\dots,n}$$

and

$$D^2 f(x) = (f_{i,x_k x_l})_{\substack{i=1,\dots,n \\ k,l=1,\dots d}} = \left(\frac{\partial^2}{\partial x_k x_l} f_i(x) \right)_{\substack{i=1,\dots,n \\ k,l=1,\dots d}}.$$

By $L^p(\Omega)$, $1 \le p \le \infty$, we denote the usual Lebesgue spaces with norms

$$\|f\|_{L^p(\Omega)} = \left(\int_\Omega |f(x)|^p \, dx \right)^{1/p} \qquad \text{for } p < \infty \qquad \text{and} \qquad \|f\|_{L^\infty(\Omega)} = \operatorname*{ess\,sup}_{x \in \Omega} |f(x)|.$$

The Sobolev space of functions $u \in L^2(\Omega)$ with weak derivatives $\nabla u \in L^2(\Omega)$ is denoted by $H^1(\Omega)$ with semi norm

$$|u|_{H^1(\Omega)} = \left(\int_\Omega |\nabla u(x)|^2 \, dx \right)^{1/2} \qquad \text{and norm} \qquad \|u\|_{H^1(\Omega)} = \left(\|u\|^2_{L^2(\Omega)} + |u|^2_{H^1(\Omega)} \right)^{1/2}.$$

Chapter 1

Concepts and abstract algorithms

1.1 Mesh refinement and coarsening

In this section, we describe the basic algorithms for the local refinement and coarsening of simplicial meshes in two and three dimensions. In 1d the grid is built from intervals, in 2d from triangles, and in 3d from tetrahedra. We restrict ourselves here to simplicial meshes, for several reasons:

1. A simplex is one of the most simple geometric types and complex domains may be approximated by a set of simplices quite easily.

2. Simplicial meshes allow local refinement (see Figure 1.1) without the need of non–conforming meshes (hanging nodes), parametric elements, or mixture of element types (which is the case for quadrilateral meshes, e.g., see Figure 1.2).

3. Polynomials of any degree are easily represented on a simplex using local (barycentric) coordinates.

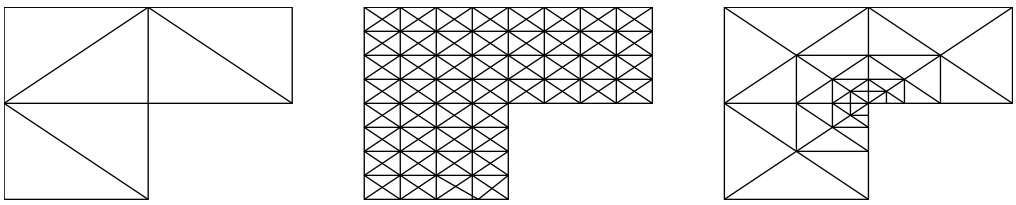


Figure 1.1: Global and local refinement of a triangular mesh.

First of all we start with the definition of a simplex, parametric simplex and triangulation:

1.1.1 Definition (Simplex)**.** a) Let $a_0, \ldots, a_d \in \mathbb{R}^n$ be given such that $a_1 - a_0, \ldots, a_d - a_0$ are linear independent vectors in $\mathbb{R}^n$. The convex set

$$S = \text{conv hull}\{a_0, \ldots, a_d\}$$

is called a *d–simplex* in $\mathbb{R}^n$. For $k < d$ let

$$S' = \text{conv hull}\{a'_0, \ldots, a'_k\} \subset \partial S$$

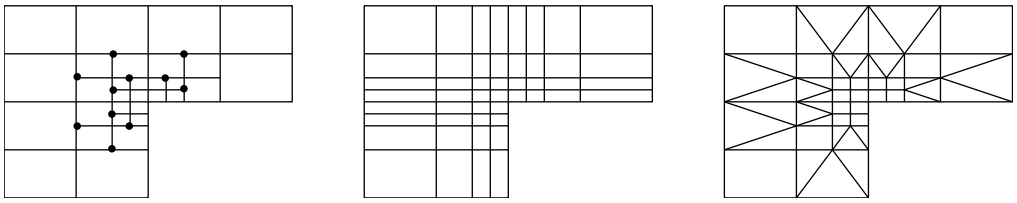


Figure 1.2: Local refinements of a rectangular mesh: with hanging nodes, conforming closure using bisected rectangles, and conforming closure using triangles. Using a conforming closure with rectangles, a local refinement has always global effects up to the boundary.

be a k–simplex with $a'_0, \dots, a'_k \in \{a_0, \dots, a_d\}$. Then S' is called a *k–sub–simplex* of S. A 0–sub–simplex is called *vertex*, a 1–sub–simplex *edge* and a 2–sub–simplex *face*.

b) The *standard simplex* in $\mathbb{R}^d$ is defined by

$$\hat{S} = \text{conv hull}\,\{\hat{a}_0 = 0, \hat{a}_1 = e_1, \dots, \hat{a}_d = e_d\}\,,$$

where e_i are the unit vectors in $\mathbb{R}^d$.

c) Let $F_S\colon \hat{S} \to S \subset \mathbb{R}^n$ be an invertible, differentiable mapping. Then S is called a *parametric d–simplex* in $\mathbb{R}^n$. The k–sub–simplices S' of S are given by the images of the k–sub–simplices $\hat{S}'$ of $\hat{S}$. Thus, the vertices $a_0, \dots, a_d$ of S are the points $F_S(\hat{a}_0), \dots, F_S(\hat{a}_d)$.

d) For a d–simplex S, we define

$$h_S := \text{diam}(S) \qquad \text{and} \qquad \rho_S := \sup\{2r;\, B_r \subset S \text{ is a } d\text{–ball of radius } r\},$$

the diameter and inball–diameter of S.

1.1.2 Remark. Every d–simplex S in $\mathbb{R}^n$ is a parametric simplex. Defining the matrix $A_S \in \mathbb{R}^{n\times d}$ by

$$A_S = \begin{bmatrix} \vdots & & \vdots \\ a_1 - a_0 & \cdots & a_d - a_0 \\ \vdots & & \vdots \end{bmatrix},$$

the parameterization $F_S\colon \hat{S} \to S$ is given by

$$F_S(\hat{x}) = A_S\,\hat{x} + a_0. \tag{1.1}$$

Since F_S is affine linear it is differentiable. It is easy to check that $F_S\colon \hat{S} \to S$ is invertible and that $F_S(\hat{a}_i) = a_i$, $i = 0, \dots, d$ holds.

1.1.3 Definition (Triangulation)**.** a) Let $\mathcal{S}$ be a set of (parametric) d–simplices and define

$$\Omega = \text{interior} \bigcup_{S\in\mathcal{S}} S \subset \mathbb{R}^n.$$

We call $\mathcal{S}$ a *conforming triangulation* of Ω, iff for two simplices $S_1, S_2 \in \mathcal{S}$ with $S_1 \neq S_2$ the intersection $S_1 \cap S_2$ is either empty or a complete k–sub–simplex of both S_1 and S_2 for some $0 \le k < d$.

b) Let $\mathcal{S}_k$, $k \ge 0$, be a sequence of conforming triangulations. This sequence is called *(shape) regular*, iff

$$\sup_{k\in\mathbb{N}_0} \max_{S\in\mathcal{S}_k} \max_{\hat{x}\in\hat{S}} \text{cond}(DF_S^t(\hat{x}) \cdot DF_S(\hat{x})) < \infty \tag{1.2}$$

holds, where DF_S is the Jacobian of F_S and $\mathrm{cond}(A) = \|A\|\|A^{-1}\|$ denotes the condition number.

1.1.4 Remark. For a sequence $\mathcal{S}_k$, $k \geq 0$, of non–parametric triangulations the regularity condition (1.2) is equivalent to the condition

$$\sup_{k\in\mathbb{N}_0} \max_{S\in\mathcal{S}_k} \frac{h_S}{\rho_S} < \infty.$$

In order to construct a sequence of triangulations, we consider the following situation: An initial (coarse) triangulation $\mathcal{S}_0$ of the domain is given. We call it *macro triangulation*. It may be generated by hand or by some mesh generation algorithm ([50, 51]).

Some (or all) of the simplices are marked for refinement, depending on some error estimator or indicator. The marked simplices are then refined, i.e. they are cut into smaller ones. After several refinements, some other simplices may be marked for coarsening. Coarsening tries to unite several simplices marked for coarsening into a bigger simplex. A successive refinement and coarsening will produce a sequence of triangulations $\mathcal{S}_0, \mathcal{S}_1, \ldots$. The refinement of single simplices that we describe in the next section produces for every simplex of the macro triangulation only a finite and small number of similarity classes for the resulting elements. The coarsening is more or less the inverse process of refinement. This leads to a finite number of similarity classes for all simplices in the whole sequence of triangulations.

The refinement of non–parametric and parametric simplices is the same topological operation and can be performed in the same way. The actual children's shape of parametric elements additionally involves the children's parameterization. In the following we describe the refinement and coarsening for triangulations consisting of non–parametric elements. The refinement of parametric triangulations can be done in the same way, additionally using given parameterizations. Regularity for the constructed sequence can be obtained with special properties of the parameterizations for parametric elements and the finite number of similarity classes for simplices.

Marking criteria and marking strategies for refinement and coarsening are subject of Section 1.5.

1.1.1 Refinement algorithms for simplicial meshes

For simplicial elements, several refinement algorithms are widely used. The discussion about and description of these algorithms mainly centers around refinement in 2d and 3d since refinement in 1d is straight forward.

One example is regular refinement ("red refinement"), which divides every triangle into four similar triangles, see Figure 1.3. The corresponding refinement algorithm in three dimensions cuts every tetrahedron into eight tetrahedra, and only a small number of similarity classes occur during successive refinements, see [13, 14]. Unfortunately, hanging nodes arise during local regular refinement. To remove them and create a conforming mesh, in two dimensions some triangles have to be bisected ("green closure"). In three dimensions, several types of irregular refinement are needed for the green closure. This creates more similarity classes, even in two dimensions. Additionally, these bisected elements have to be removed before a further refinement of the mesh, in order to keep the triangulations shape regular.

Another possibility is to use bisection of simplices only. For every element (triangle or tetrahedron) one of its edges is marked as the *refinement edge*, and the element is refined into

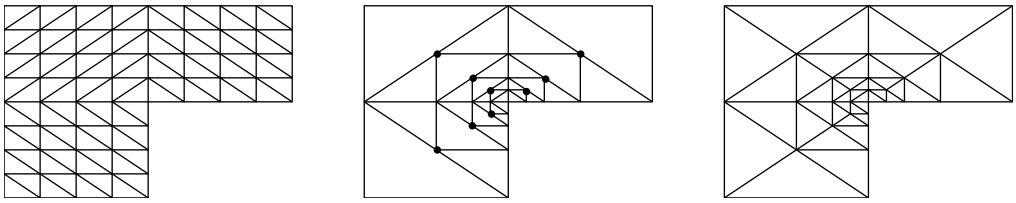


Figure 1.3: Global and local regular refinement of triangles and conforming closure by bisection.

two elements by cutting this edge at its midpoint. There are several possibilities to choose such a refinement edge for a simplex, one example is to use the longest edge; Mitchell [39] compared different approaches. We focus on an algorithm where the choice of refinement edges on the macro triangulation prescribes the refinement edges for all simplices that are created during mesh refinement. This makes sure that shape regularity of the triangulations is conserved.

In two dimensions we use the *newest vertex* bisection (in Mitchell's notation) and in three dimensions the bisection procedure of Kossaczký described in [35]. We use the convention, that all vertices of an element are given fixed *local indices*. Valid indices are 0, 1, for vertices of an interval, 0, 1, and 2 for vertices of a triangle, and 0, 1, 2, and 3 for vertices of a tetrahedron. Now, the refinement edge for an element is fixed to be the edge between the vertices with local indices 0 and 1. Here we use the convention that in 1d the element itself is called "refinement edge".

During refinement, the new vertex numbers, and thereby the refinement edges, for the newly created child simplices are prescribed by the refinement algorithm. For both child elements, the index of the newly generated vertex at the midpoint of this edge has the highest local index (2 resp. 3 for triangles and tetrahedra). These numbers are shown in Figure 1.4 for 1d and 2d, and in Figure 1.5 for 3d. In 1d and 2d this numbering is the same for all refinement levels. In 3d, one has to make some special arrangements: the numbering of the second child's vertices depends on the *type* of the element. There exist three different element types 0, 1, and 2. The type of the elements on the macro triangulation can be prescribed (usually type 0 tetrahedron). The type of the refined tetrahedra is recursively given by the definition that the type of a child element is ((parent's type + 1) modulo 3). In Figure 1.5 we used the following convention: for the index set `{1,2,2}` on `child[1]` of a tetrahedron of type `0` we use the index `1` and for a tetrahedron of type `1` and `2` the index `2`. Figure 1.6 shows successive refinements of a type 0 tetrahedron, producing tetrahedra of types 1, 2, and 0 again.

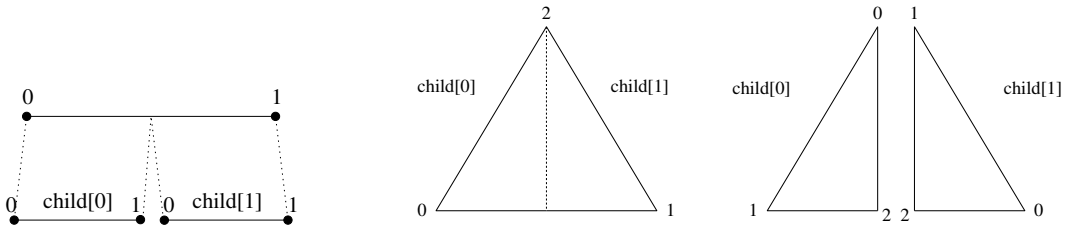


Figure 1.4: Numbering of nodes on parent and children for intervals and triangles.

By the above algorithm the refinements of simplices are totally determined by the local

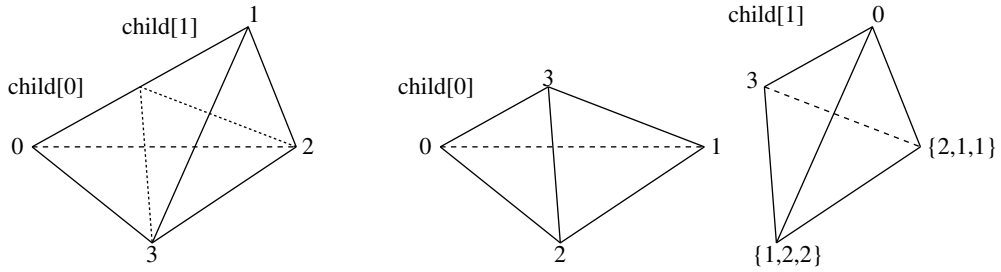


Figure 1.5: Numbering of nodes on parent and children for tetrahedra.

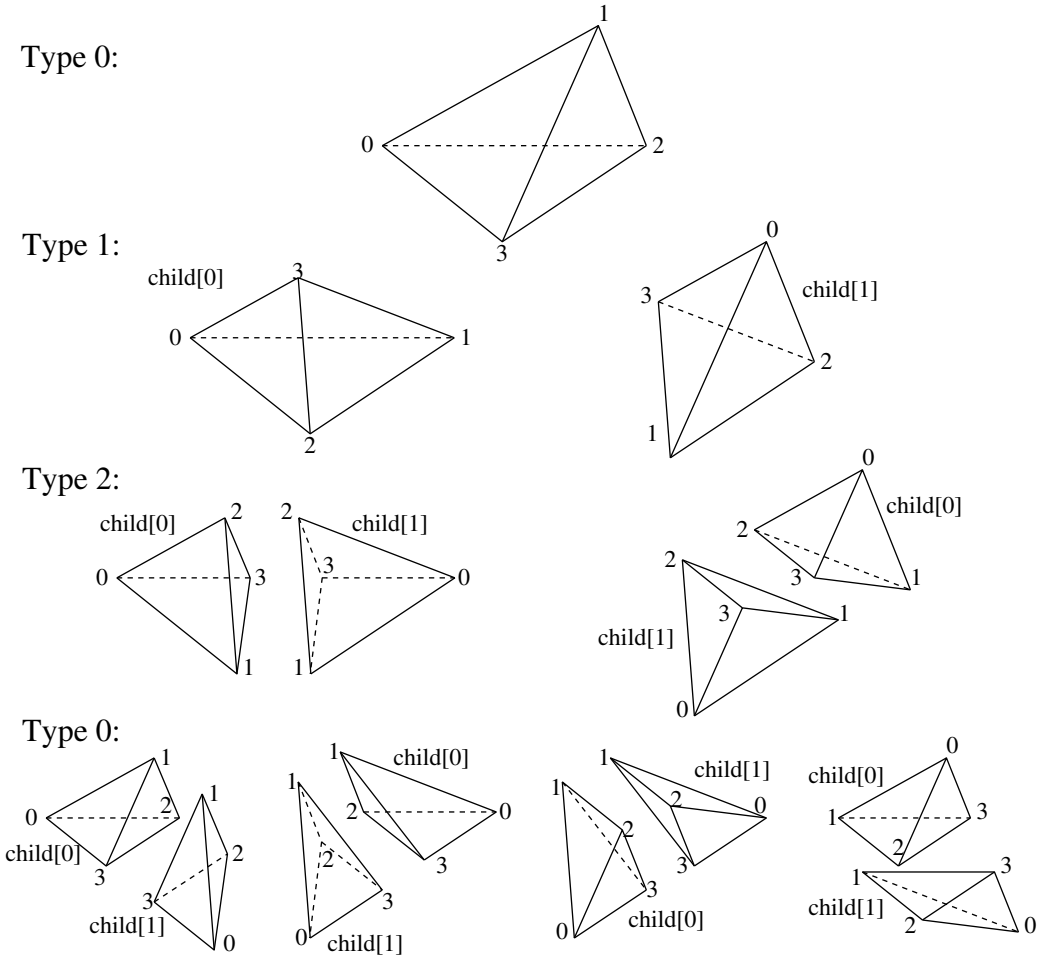


Figure 1.6: Successive refinements of a type 0 tetrahedron.

vertex numbering on the macro triangulation, plus a prescribed type for every macro element in three dimensions. Furthermore, a successive refinement of every macro element only produces a small number of similarity classes. In case of the "generic" triangulation of a (unit) square in 2d and cube in 3d into two triangles resp. six tetrahedra (see Figure 1.7 for a single triangle and tetrahedron from such a triangulation – all other elements are generated by rotation and reflection), the numbering and the definition of the refinement edge during refinement of the elements guarantee that the longest edge will always be the refinement edge,

see Figure 1.8.

The refinement of a given triangulation now uses the bisection of *single* elements and can be performed either *iteratively* or *recursively*. In 1d, bisection only involves the element which is subject to refinement and thus is a completely local operation. Both variants of refining a given triangulation are the same. In 2d and 3d, bisection of a single element usually involves other elements, resulting in two different algorithms.

For tetrahedra, the first description of such a refinement procedure was given by Bänsch using the iterative variant [7]. It abandons the requirement of one to one inter–element adjacencies during the refinement process and thus needs the intermediate handling of hanging nodes. Two recursive algorithms, which do not create such hanging nodes and are therefore easier to implement, are published by Kossaczký [35] and Maubach [37]. For a special class of macro triangulations, they result in exactly the same tetrahedral meshes as the iterative algorithm.

In order to keep the mesh conforming during refinement, the bisection of an edge is allowed only when such an edge is the refinement edge for *all* elements which share this edge. Bisection of an edge and thus of all elements around the edge is the *atomic refinement operation*, and no other refinement operation is allowed. See Figures 1.9 and 1.10 for the two and three–dimensional situations.

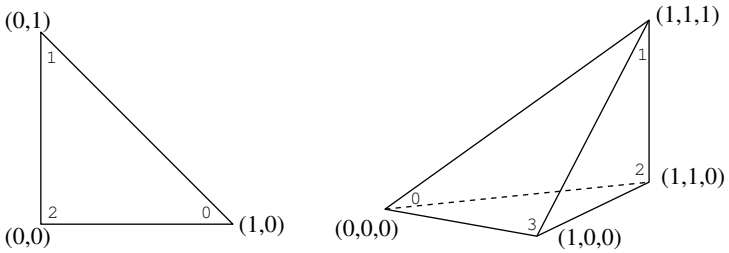


Figure 1.7: Generic elements in two and three dimensions.

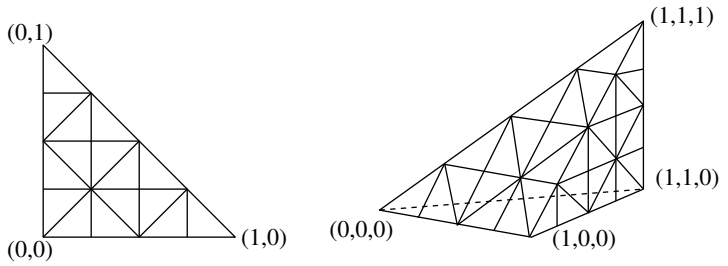


Figure 1.8: Refined generic elements in two and three dimensions.

If an element has to be refined, we have to collect all elements at its refinement edge. In two dimensions this is either the neighbour opposite this edge or there is no other element in the case that the refinement edge belongs to the boundary. In three dimensions we have to loop around the edge and collect all neighbours at this edge. If for all collected neighbours the common edge is the refinement edge too, we can refine the whole patch at the same time by inserting one new vertex in the midpoint of the common refinement edge and bisecting every element of the patch. The resulting triangulation then is a conforming one.

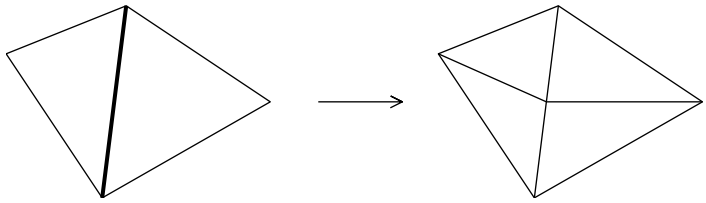


Figure 1.9: Atomic refinement operation in two dimensions. The common edge is the refinement edge for both triangles.

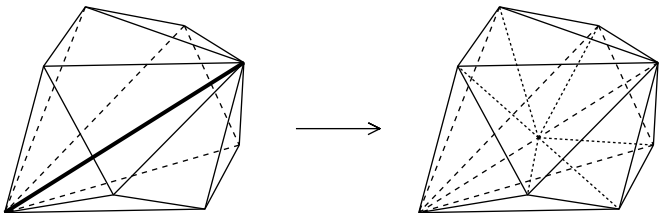


Figure 1.10: Atomic refinement operation in three dimensions. The common edge is the refinement edge for all tetrahedra sharing this edge.

But sometimes the refinement edge of a neighbour is not the common edge. Such a neighbour is *not compatibly divisible* and we have to perform first the atomic refinement operation at the neighbour's refinement edge. In 2d the child of such a neighbour at the common edge is then compatibly divisible; in 3d such a neighbour has to be bisected at most three times and the resulting tetrahedron at the common edge is then compatibly divisible. The recursive refinement algorithm now reads

1.1.5 Algorithm (Recursive refinement of one simplex)**.**

```
subroutine recursive_refine(S, 𝒮)
  do
    𝒜 := {S' ∈ 𝒮; S' is not compatibly divisible with S}
    for all S' ∈ 𝒜 do
      recursive_refine(S', 𝒮);
    end for
  until 𝒜 = ∅

  𝒜 := {S' ∈ 𝒮; S' is element at the refinement edge of S}
  for all S' ∈ 𝒜
    bisect S' into S'_0 and S'_1
    𝒮 := 𝒮\{S'} ∪ {S'_0, S'_1}
  end for
```

In Figure 1.11 we show a two–dimensional situation where recursion is needed. For all triangles, the longest edge is the refinement edge. Let us assume that triangles A and B are marked for refinement. Triangle A can be refined at once, as its refinement edge is a boundary edge. For refinement of triangle B, we have to recursively refine triangles C and D. Again, triangle D can be directly refined, so recursion terminates there. This is shown in the second part of the figure. Back in triangle C, this can now be refined together with its neighbour. After this, also triangle B can be refined together with its neighbour.

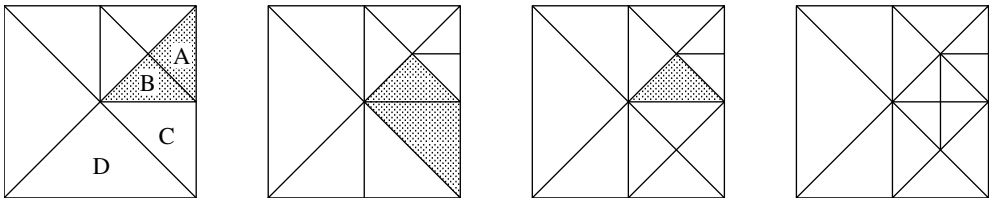


Figure 1.11: Recursive refinement in two dimensions. Triangles A and B are initially marked for refinement.

The refinement of a given triangulation $\mathcal{S}$ where some or all elements are marked for refinement is then performed by

1.1.6 Algorithm (Recursive refinement algorithm)**.**

```
subroutine refine(S)
  for all S ∈ S do
    if S is marked for refinement
      recursive_refine(S, S)
    end if
  end for
```

Since we use recursion, we have to guarantee that recursions terminates. Kossaczký [35] and Mitchell [39] proved

1.1.7 Theorem (Termination and Shape Regularity)**.** *The recursive refinement algorithm using bisection of single elements fulfills*

1. *The recursion terminates if the macro triangulation satisfies certain criteria.*
2. *We obtain shape regularity for all elements at all levels.*

1.1.8 Remark. 1.) A first observation is, that simplices initially not marked for refinement are bisected, enforced by the refinement of a marked simplex. This is a necessity to obtain a conforming triangulation, also for the regular refinement.

2.) It is possible to mark an element for more than one bisection. The natural choice is to mark a d–simplex S for d bisections. After d refinement steps all original edges of S are bisected. A simplex S is refined k times by refining the children S_1 and S_2 $k-1$ times right after the refinement of S.

3.) The recursion does not terminate for an arbitrary choice of refinement edges on the macro triangulation. In two dimensions, such a situation is shown in Figure 1.12. The selected refinement edges of the triangles are shown by dashed lines. One can easily see, that there are no patches for the atomic refinement operation. This triangulation can only be refined if other choices of refinement edges are made, or by a non–recursive algorithm.

4.) In two dimensions, for every macro triangulation it is possible to choose the refinement edges in such a way that the recursion terminates (selecting the 'longest edge'). In three dimensions the situation is more complicated. But there is a maybe refined grid such that refinement edges can be chosen in such a way that recursion terminates [35].

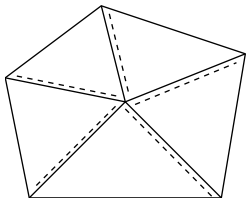


Figure 1.12: A macro triangulation where recursion does not stop.

1.1.2 Coarsening algorithm for simplicial meshes

The coarsening algorithm is more or less the inverse of the refinement algorithm. The basic idea is to collect all those elements that were created during the refinement at same time, i.e. the parents of these elements build a compatible refinement patch. The elements may only be coarsened if *all* involved elements are marked for coarsening and are of finest level locally, i.e. no element is refined further. The actual coarsening again can be performed in an *atomic coarsening operation* without the handling of hanging nodes. Information is passed from all elements onto the parents and the whole patch is coarsened at the same time by removing the vertex in the parent's common refinement edge (see Figures 1.13 and 1.14 for the atomic coarsening operation in 2d and 3d). This coarsening operation is completely local in 1d.

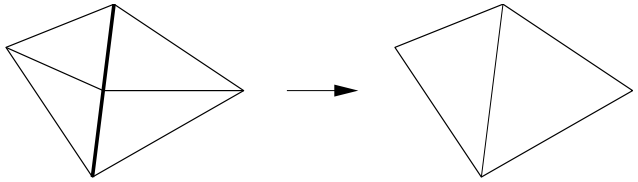


Figure 1.13: Atomic coarsening operation in two dimensions.

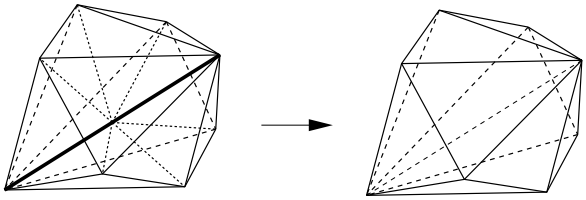


Figure 1.14: Atomic coarsening operation in three dimensions.

During refinement, the bisection of an element can enforce the refinement of an unmarked element in order to keep the mesh conforming. During coarsening, an element may only be coarsened if all elements involved in this operation are marked for coarsening. This is the main difference between refinement and coarsening. In an adaptive method this guarantees that elements with a large local error indicator marked for refinement are refined and no element is coarsened where the local error indicator is not small enough (compare Section 1.5.3).

Since the coarsening process is the inverse of the refinement, refinement edges on parent elements are again at their original position. Thus, further refinement is possible with a terminating recursion and shape regularity for all resulting elements.

1.1.9 Algorithm (Local coarsening)**.**

`subroutine coarsen_element(`S`, `$\mathcal{S}$`)`

```
A := {S' ∈ S; S' must not be coarsened with S}
if A = ∅
  for all child pairs S'_0, S'_1 at common coarsening edge do
    coarsen S'_0 and S'_1 into the parent S'
    S := S\{S'_0, S'_1} ∪ {S'}
  end for
end if
```

The following routine coarsens as many elements as possible of a given triangulation $\mathcal{S}$:

1.1.10 Algorithm (Coarsening algorithm)**.**

```
subroutine coarsen(S)
  for all S ∈ S do
    if S is marked for coarsening
      coarsen_element(S, S)
    end if
  end for
```

1.1.11 Remark. Also in the coarsening procedure an element can be marked for several coarsening steps. Usually, the coarsening markers for all patch elements are cleared if a patch must not be coarsened. If the patch must not be coarsened because one patch element is not of locally finest level but may coarsened more than once, elements stay marked for coarsening. A coarsening of the finer elements can result in a patch which may then be coarsened.

1.1.3 Operations during refinement and coarsening

The refinement and coarsening of elements can be split into four major steps, which are now described in detail.

1.1.3.1 Topological refinement and coarsening

The actual bisection of an element is performed as follows: the simplex is cut into two children by inserting a new vertex at the refinement edge. All objects like this new vertex, or a new edge (in 2d and 3d), or face (in 3d) only have to be created once on the refinement patch. For example, all elements *share* the new vertex and two child triangles share a common edge. The refinement edge is divided into two smaller ones which have to be adjusted to the respective children. In 3d all faces inside the patch are bisected into two smaller ones and this creates an additional edge for each face. All these objects can be shared by several elements and have to be assigned to them. If neighbour information is stored, one has to update such information for elements inside the patch as well as for neighbours at the patch's boundary.

In the coarsening process the vertex which is shared by all elements is removed, edges and faces are rejoined and assigned to the respective parent simplices. Neighbour information has to be reinstalled inside the patch and with patch neighbours.

1.1.3.2 Administration of degrees of freedoms

Single DOFs can be assigned to a vertex, edge, or face and such a DOF is shared by all simplices meeting at the vertex, edge, or face respectively. Finally, there may be DOFs on the

element itself, which are not shared with any other simplex. At each object there may be a single DOF or several DOFs, even for several finite element spaces.

During refinement new DOFs are created. For each newly created object (vertex, edge, face, center) we have to create the exact amount of DOFs, if DOFs are assigned to the object. For example we have to create vertex DOFs at the midpoint of the refinement edge, if DOFs are assigned to a vertex. Again, DOFs must only be created once for each object and have to be assigned to all simplices having this object in common.

Additionally, all vectors and matrices using these DOFs have to be adjusted in size automatically.

1.1.3.3 Transfer of geometric data

Information about the childrens'/parent's shape has to be transformed. During refinement, for a simplex we only have to calculate the coordinates of the midpoint of the refinement edge, coordinates of the other vertices stay the same and can be handed from parent to children. If the refinement edge belongs to a curved boundary, the coordinates of the new vertex are calculated by projecting this midpoint onto the curved boundary. During coarsening, no calculations have to be done. The $d+1$ vertices of the two children which are *not* removed are the vertices of the parent.

For the shape of parametric elements, usually more information has to be calculated. Such information can be stored in a DOF–vector, e.g., and may need DOFs on parent and children. Thus, information has to be assembled after installing the DOFs on the children and before deleting DOFs on the parent during refinement; during coarsening, first DOFs on the parent have to be installed, then information can be assembled, and finally the children's DOFs are removed.

1.1.3.4 Transformation of finite element information

Using iterative solvers for the (non-) linear systems, a good initial guess is needed. Usually, the discrete solution from the old grid, interpolated into the finite element space on the new grid, is a good initial guess. For piecewise linear finite elements we only have to compute the value at the newly created node at the midpoint of the refinement edge and this value is the mean value of the values at the vertices of the refinement edge:

$$u_h(\text{midpoint}) = \frac{1}{2}(u_h(\text{vertex } 0) + u_h(\text{vertex } 1)).$$

For linear elements an interpolation during coarsening is trivial since the values at the vertices of the parents stay the same.

For higher order elements more DOFs are involved, but only DOFs belonging to the refinement/coarsening patch. The interpolation strongly depends on the local basis functions and it is described in detail in Section 1.4.4.

Usually during coarsening information is lost (for example, we lose information about the value of a linear finite element function at the coarsening edge's midpoint). But linear functionals applied to basis functions that are calculated on the fine grid and stored in some coefficient vector can be transformed during coarsening without loss of information, if the finite element spaces are nested. This is also described in detail in Section 1.4.4. One application of this procedure is a time discretization, where L^2 scalar products of the new basis functions

with the solution u_h^{old} from the last time step appear on the right hand side of the discrete problem.

Since DOFs can be shared by several elements, these operations are done on the whole refinement/coarsening patch. This avoids that coefficients of the interpolant are calculated more than once for a shared DOF. During the restriction of a linear functional we have to add contribution(s) from one/several DOF(s) to some other DOF(s). Performing this operation on the whole patch makes it easy to guarantee that the contribution of a shared DOF is only added once.

1.2 The hierarchical mesh

There are basically two ways of storing a finite element grid. One possibility is to store only the elements of the triangulation in a vector or a linked list. All information about elements is located at the elements. In this situation there is no direct information of a hierarchical structure, needed, for example, for multigrid methods. Such information has to be generated and stored separately. During mesh refinement, new elements are added (at the end) to the vector or list of elements. During mesh coarsening, elements are removed. In case of an element vector, 'holes' may appear in the vector that contain no longer a valid element. One has to take care of them, or remove them by compressing the vector.

ALBERTA uses the second way of storing the mesh. It keeps information about the whole hierarchy of grids starting with the macro triangulation up to the actual one. Storing information about the whole hierarchical structure will need an additional amount of computer memory. On the other hand, we can save computer memory because such information which can be produced by the hierarchical structure does not have to be stored explicitly on each element.

The simplicial grid is generated by refinement of a given macro triangulation. Refined parts of the grid can be de–refined, but we can not coarsen elements of the macro triangulation. The refinement and coarsening routines, described in Section 1.1, construct a sequence of nested grids with a hierarchical structure. Every refined simplex is refined into two children. Elements that may be coarsened were created by refining the parent into these two elements and are now just coarsened back into this parent (compare Sections 1.1.1, 1.1.2).

Using this structure of the refinement/coarsening routines, every element of the macro triangulation is the root of a binary tree: every interior node of that tree has two pointers to the two children; the leaf elements are part of the actual triangulation, which is used to define the finite element space. The whole triangulation is a list of given macro elements together with the associated binary trees, compare Figure 1.2.

Some information is stored on the (leaf) elements explicitly, other information is located at the macro elements and is transferred to the leaf elements while traversing through the binary tree. For instance, information about DOFs has to be stored explicitely for all (leaf) elements whereas geometric information can be produced using the hierarchical structure.

Operations on elements can only be performed by using the mesh traversal routines described in Section 3.2.17. These routines need as arguments a flag which indicates which information should be present on the elements, which elements should be called (interior or leaf elements), and a pointer to a function which performs the operation on a single element. The traversal routines always start on the first macro element and go to the indicated elements of the binary tree at this macro element. This is done in a recursive way by first traversing

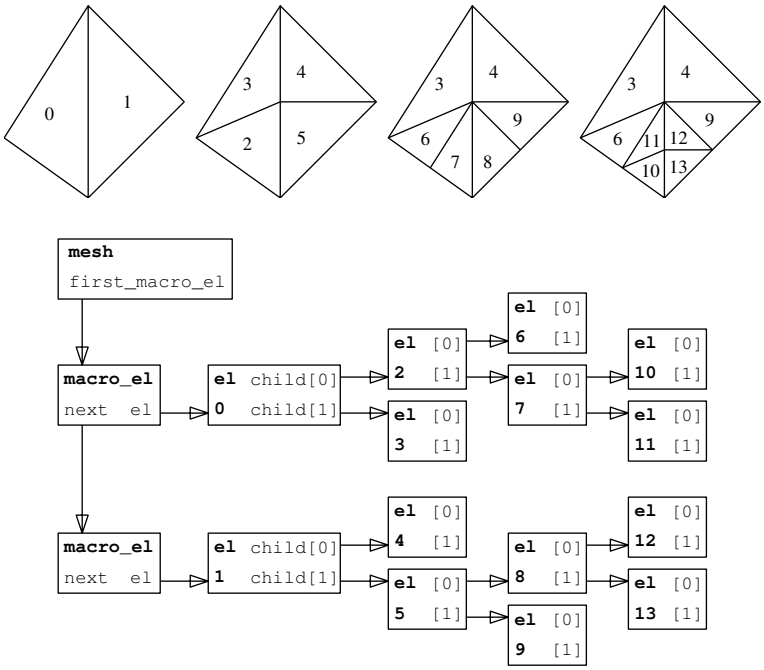


Figure 1.15: Sample mesh refinement and corresponding element trees

through the subtree of the first child and then by traversing through the subtree of the second child. This recursion terminates if a leaf element is reached. After calling all elements of this tree we go to the next macro element, traverse through the binary tree located there, and so on until the end of the list of macro elements.

All information that should be available for mesh elements is stored explicitly for elements of the macro triangulation. Thus, all information is present on the macro level and is transfered to the other tree elements by transforming requested data from one element to its children. This can be done by simple calculations using the hierarchic structure induced by the refinement algorithm, compare Section 1.1.1.

As mentioned above, geometric data like coordinates of the element's vertices can be efficiently computed using the hierarchical structure (in the case of non-parametric elements and polyhedral boundary). Going from parent to child only the coordinates of one vertex change and the new ones are simply the mean value of the coordinates of two vertices at the refinement edge of the parent. The other vertex coordinates stay the same. Another example of such information is information about adjacent elements. Using adjacency information of the macro elements we can compute requested information for all elements of the mesh.

User data on leaf elements Many finite element applications need special information on each element of the actual triangulation, i.e. the leaf elements of the hierarchical mesh. In adaptive codes this may be, for example, error indicators produced by an error estimator. Such information needs to be available only for leaf elements and not for elements inside the binary tree.

The fact that leaf elements do not have children, and thus the pointers to such children in leaf element's data structures are not used, can be exploited by enabling access to special data via these pointers. So, special pointers for such data do not have to be included in an

element data structure. Details about such an implementation are given in

1.3 Degrees of freedom

Degrees of freedom (DOFs) connect finite element data with geometric information of a triangulation. Each finite element function is uniquely determined by the values (coefficients) of all its degrees of freedom.

For example, a continuous and piecewise linear finite element function can be described by the values of this function at all vertices of the triangulation. They build this function's degrees of freedom. A piecewise constant function is determined by its value in each element. In ALBERTA, every abstract DOF is realized as an integer index into vectors, which corresponds to the global index in a vector of coefficients.

For the definition of general finite element spaces DOFs located at vertices of elements, at edges (in 2d and 3d), at faces (in 3d), or in the interior of elements are needed. DOFs at a vertex are shared by all elements which meet at this vertex, DOFs at an edge or face are shared by all elements which contain this edge or face, and DOFs inside an element are not shared with any other element. The location of a DOF and the sharing between elements corresponds directly to the support of basis functions that are connected to them, see Figure 1.16.

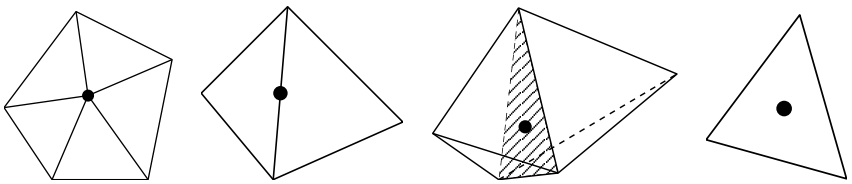


Figure 1.16: Support of basis functions connected with a DOF at a vertex, edge, face (only in 3d), and the interior.

When DOFs and basis functions are used in a hierarchical manner, then the above applies only to a single hierarchical level. Due to the hierarchies, the supports of basis functions which belong to different levels do overlap.

For general applications, it may be necessary to handle several different sets of degrees of freedom on the same triangulation. For example, in mixed finite element methods for the Navier–Stokes problem, different polynomial degrees are used for discrete velocity and pressure functions. In Figure 1.17, three examples of DOF distributions for continuous finite elements in 2d are shown: piecewise quadratic finite elements ⊡ (left), piecewise linear ⊞ and piecewise quadratic ⊡ finite elements (middle, Taylor–Hood element for Navier–Stokes: linear pressure and quadratic velocity), piecewise cubic ⊞ and piecewise quartic ⊡ finite elements (right, Taylor–Hood element for Navier–Stokes: quartic velocity and linear pressure).

Additionally, different finite element spaces may use the same set of degrees of freedom, if appropriate. For example, higher order elements with Lagrange type basis or a hierarchical type basis can share the same abstract set of DOFs.

The DOFs are directly connected to the mesh and its elements, by the connection between local (on each element) and global degrees of freedom. On the other hand, an application uses DOFs only in connection with finite element spaces and basis functions. Thus, while the administration of DOFs is handled by the mesh, definition and access to DOFs is mainly done via finite element spaces.

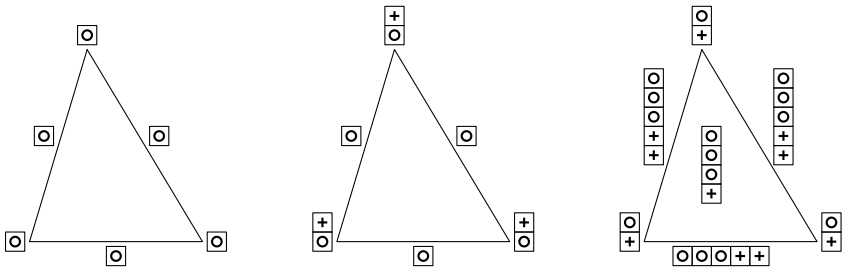


Figure 1.17: Examples of DOF distributions in 2d.

1.4 Finite element spaces and finite element discretization

In the sequel we assume that $\Omega \subset \mathbb{R}^d$ is a bounded domain triangulated by $\mathcal{S}$, i.e.

$$\bar{\Omega} = \bigcup_{S \in \mathcal{S}} S.$$

The following considerations are also valid for a triangulation of an immersed surface (with $n > d$). In this situation one has to exchange derivatives (those with respect to x) by *tangential* derivatives (tangential to the actual element, derivatives are always taken element–wise) and the determinant of the parameterization's Jacobian has to be replaced by Gram's determinant of the parameterization. But for the sake of clearness and simplicity we restrict our considerations to the case $n = d$.

The values of a finite element function or the values of its derivatives are uniquely defined by the values of its DOFs and the values of the basis functions or the derivatives of the basis functions connected with these DOFs. Usually, evaluation of those values is performed element–wise. On a single element the value of a finite element function at a point x in this element is determined by the DOFs associated with this specific element and the values of the non vanishing basis functions at this point.

We follow the concept of finite elements which are given on a single element S in local coordinates. We distinguish two classes of finite elements:

Finite element functions on an element S defined by a finite dimensional function space $\bar{\mathbb{P}}$ on a *reference element* $\bar{S}$ and the (one to one) mapping λ^S from the reference element $\bar{S}$ to S. For this class the dependency on the actual element S is fully described by the mapping λ^S. For example, all Lagrange finite elements belong to this class.

Secondly, finite element functions depending on the actual element S. Hence, the basis functions are not fully described by $\bar{\mathbb{P}}$ and the one to one mapping λ^S. But using an initialization of the actual element (which defines a finite dimensional function space $\bar{\mathbb{P}}$ with information about the actual element), we can implement this class in the same way as the first one. This class is needed for Hermite finite elements which are not affine equivalent to the reference element. Examples in 2d are the *Hsieh–Clough–Tocher* or *HCT element* or the *Argyris element* where only the normal derivative at the midpoint of edges are used in the definition of finite element functions; both elements lead to functions which are globally $C^1(\bar{\Omega})$. The concrete implementation for this class in ALBERTA is future work.

All in all, for a very general situation, we only need a vector of basis functions (and their derivatives) on $\bar{S}$ and a function which connects each of these basis functions with its degree of freedom on the element. For the second class, we additionally need an initialization routine for

the actual element. By such information, every finite element function is uniquely described on every element of the grid.

1.4.1 Barycentric coordinates

For describing finite elements on simplicial grids, it is very convenient to use $d+1$ barycentric coordinates as a local coordinate system on an element of the triangulation. Using $d+1$ local coordinates, the *reference simplex* $\bar{S}$ is a subset of a hyper surface in $\mathbb{R}^{d+1}$:

$$\bar{S} := \{(\lambda_0, \ldots, \lambda_d) \in \mathbb{R}^{d+1}; \lambda_k \geq 0, \sum_{k=0}^{d} \lambda_k = 1\}$$

On the other hand, for numerical integration on an element it is much more convenient to use the standard element $\hat{S} \in \mathbb{R}^d$ defined in Section 1.1 as

$$\hat{S} = \text{conv hull}\,\{\hat{a}_0 = 0, \hat{a}_1 = e_1, \ldots, \hat{a}_d = e_d\}$$

where e_i are the unit vectors in $\mathbb{R}^d$; using $\hat{S}$ for the numerical integration, we only have to compute the determinant of the parameterization's Jacobian and not Gram's determinant.

The relation between a given simplex S, the reference simplex $\bar{S}$, and the standard simplex $\hat{S}$ is now described in detail.

Let S be an element of the triangulation with vertices $\{a_0, \ldots, a_d\}$; let $F_S \colon \hat{S} \to S$ be the diffeomorphic parameterization of S over $\hat{S}$ with regular Jacobian DF_S, such that

$$F_S(\hat{a}_k) = a_k, \qquad k = 0, \ldots, d$$

holds. For a point $x \in S$ we set

$$\hat{x} = F_S^{-1}(x) \in \hat{S}.$$

For a simplex S the easiest choice for F_S is the unique affine mapping (1.1) defined on page 2. For an affine mapping, DF_S is constant. In the following, we assume that the parameterization F_S of a simplex S is affine.

For a simplex S the barycentric coordinates

$$\lambda^S(x) \;=\; (\lambda_0^S, \ldots, \lambda_d^S)(x) \in \mathbb{R}^{d+1}$$

of some point $x \in \mathbb{R}^d$ are (uniquely) determined by the $(d+1)$ equations

$$\sum_{k=0}^{d} \lambda_k^S(x)\, a_k = x \qquad \text{and} \qquad \sum_{k=0}^{d} \lambda_k^S(x) = 1.$$

The following relation holds:

$$x \in S \qquad \text{iff} \qquad \lambda_k^S(x) \in [0,1] \;\text{ for all } k = 0, \ldots, d \qquad \text{iff} \qquad \lambda^S \in \bar{S}.$$

On the other hand, each $\lambda \in \bar{S}$ defines a unique point $x^S \in S$ by

$$x^S(\lambda) = \sum_{k=0}^{d} \lambda_k\, a_k.$$

Thus, $x^S\colon \bar{S} \to S$ is an invertible mapping with inverse $\lambda^S\colon S \to \bar{S}$. The barycentric coordinates of x on S are the same as those of $\hat{x}$ on $\hat{S}$, i.e. $\lambda^S(x) = \lambda^{\hat{S}}(\hat{x})$.

In the general situation, when F_S may not be affine, i.e. we have a parametric element, the barycentric coordinates λ^S are given by the inverse of the parameterization F_S and the barycentric coordinates on $\hat{S}$:

$$\lambda^S(x) = \lambda^{\hat{S}}(\hat{x}) = \lambda^{\hat{S}}\left(F_S^{-1}(x)\right)$$

and the *world coordinates* of a point $x^S \in S$ with barycentric coordinates λ are given by

$$x^S(\lambda) \;=\; F_S\left(\sum_{k=0}^{d} \lambda_k \hat{a}_k\right) \;=\; F_S\left(x^{\hat{S}}(\lambda)\right)$$

(see also Figure 1.18).

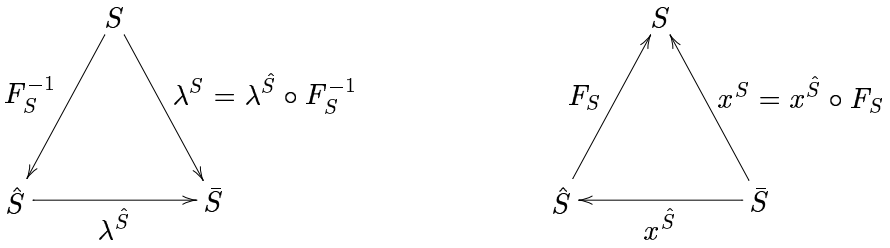


Figure 1.18: Definition of $\lambda^S\colon S \to \bar{S}$ via F_S^{-1} and $\lambda^{\hat{S}}$, and $x^S\colon \bar{S} \to S$ via $x^{\hat{S}}$ and F_S

Every function $f\colon S \to V$ defines (uniquely) two functions

$$\begin{array}{rcl} \bar{f}\colon \bar{S} & \to & V \\ \lambda & \mapsto & f(x^S(\lambda)) \end{array} \qquad \text{and} \qquad \begin{array}{rcl} \hat{f}\colon \hat{S} & \to & V \\ \hat{x} & \mapsto & f(F_S(\hat{x})). \end{array}$$

Accordingly, $\hat{f}\colon \hat{S} \to V$ defines two functions $f\colon S \to V$ and $\bar{f}\colon \bar{S} \to V$, and $\bar{f}\colon \bar{S} \to V$ defines $f\colon S \to V$ and $\hat{f}\colon \hat{S} \to V$.

Assuming that a function space $\bar{\mathbb{P}} \subset C^0(\bar{S})$ is given, it uniquely defines function spaces $\hat{\mathbb{P}}$ and $\mathbb{P}_S$ by

$$\hat{\mathbb{P}} = \left\{\hat{\varphi} \in C^0(\hat{S});\, \bar{\varphi} \in \bar{\mathbb{P}}\right\} \qquad \text{and} \qquad \mathbb{P}_S = \left\{\varphi \in C^0(S);\, \bar{\varphi} \in \bar{\mathbb{P}}\right\}. \tag{1.3}$$

We can also assume that the function space $\hat{\mathbb{P}}$ is given and this space uniquely defines $\bar{\mathbb{P}}$ and $\mathbb{P}_S$ in the same manner. In ALBERTA, we use the function space $\bar{\mathbb{P}}$ on $\bar{S}$; the implementation of a basis $\left\{\bar{\varphi}^1, \ldots, \bar{\varphi}^m\right\}$ of $\bar{\mathbb{P}}$ is much simpler than the implementation of a basis $\left\{\hat{\varphi}^1, \ldots, \hat{\varphi}^m\right\}$ of $\hat{\mathbb{P}}$ as we are able to use symmetry properties of the barycentric coordinates λ.

In the following we shall often drop the superscript S of λ^S and x^S. The mappings $\lambda(x) = \lambda^S(x)$ and $x(\lambda) = x^S(\lambda)$ are always defined with respect to the actual element $S \in \mathcal{S}$.

1.4.2 Finite element spaces

ALBERTA supports scalar and vector valued finite element functions. The basis functions are always real valued; thus, the coefficients of a finite element function in a representation by the basis functions are either scalars or vectors of length n.

For a given function space $\bar{\mathbb{P}}$ and some given real valued function space C on Ω, a finite element space X_h is defined by

$$X_h = X_h(\mathcal{S}, \bar{\mathbb{P}}, C) = \left\{\varphi \in C;\ \varphi_{|S} \in \mathbb{P}_S \text{ for all } S \in \mathcal{S}\right\}$$

for scalar finite elements. For vector valued finite elements, X_h is given by

$$X_h = X_h(\mathcal{S}, \bar{\mathbb{P}}, C) = \left\{\varphi = (\varphi_1, \ldots, \varphi_n) \in C^n;\ \varphi_{i\,|S} \in \mathbb{P}_S \text{ for all } i = 1, \ldots, n,\ S \in \mathcal{S}\right\}.$$

The spaces $\mathbb{P}_S$ are defined by $\bar{\mathbb{P}}$ via (1.3).

For conforming finite element discretizations, C is the continuous space X (for 2nd order problems, $C = X = H^1(\Omega)$). For non–conforming finite element discretizations, C may control the *non conformity* of the ansatz space which has to be controlled in order to obtain optimal error estimates (for example, in the discretization of the incompressible Navier–Stokes equation by the non–conforming Crouzeix–Raviart element, the finite element functions are continuous only in the midpoints of the edges).

The abstract definition of finite element spaces as a triple (mesh, local basis functions, and DOFs) now matches the mathematical definition as $X_h = X_h(\mathcal{S}, \bar{\mathbb{P}}, C)$ in the following way: The mesh is the underlying triangulation $\mathcal{S}$, the local basis functions are given by the function space $\bar{\mathbb{P}}$, and together with the relation between global and local degrees of freedom every finite element function satisfies the global continuity constraint C. This relation between global and local DOFs is now described in detail.

1.4.3 Evaluation of finite element functions

Let $\left\{\bar{\varphi}^1, \ldots, \bar{\varphi}^m\right\}$ be a basis of $\bar{\mathbb{P}}$ and let $\{\varphi_1, \ldots, \varphi_N\}$ be a basis of X_h, $N = \dim X_h$, such that for every $S \in \mathcal{S}$ and for all basis functions φ_j which do not vanish on S

$$\varphi_j|_S(x(\lambda)) = \bar{\varphi}^i(\lambda) \qquad \text{for all } \lambda \in \bar{S}$$

holds with some $i \in \{1, \ldots, m\}$ depending on j and S. Thus, the numbering of the basis functions in $\bar{\mathbb{P}}$ and the mapping x^S induces a local numbering of the non vanishing basis functions on S. Denoting by J_S the index set of all non vanishing basis functions on S, the mapping $i_S : J_S \to \{1, \ldots, m\}$ is one to one and uniquely connects the degrees of freedom of a finite element function on S with the local numbering of basis functions. If $j_S : \{1, \ldots, m\} \to J_S$ denotes the inverse mapping of i_S, the connection between global and local basis functions is uniquely determined on each element S by

$$\varphi_j(x(\lambda)) = \bar{\varphi}^{i_S(j)}(\lambda), \qquad \text{for all } \lambda \in \bar{S},\ j \in J_S, \tag{1.4a}$$

$$\varphi_{j_S(i)}(x(\lambda)) = \bar{\varphi}^i(\lambda), \qquad \text{for all } \lambda \in \bar{S},\ i \in \{1, \ldots, m\}. \tag{1.4b}$$

For $u_h \in X_h$ denote by $(u_1, \ldots, u_N)$ the *global coefficient vector* of the basis $\{\varphi_j\}$ with $u_j \in \mathbb{R}$ for scalar finite elements, $u_j \in \mathbb{R}^n$ for vector valued finite elements, i.e.

$$u_h(x) = \sum_{j=1}^{N} u_j \varphi_j(x) \qquad \text{for all } x \in \Omega,$$

and the *local coefficient vector*

$$\left(u_S^1, \ldots, u_S^m\right) = \left(u_{j_S(1)}, \ldots, u_{j_S(m)}\right)$$

of u_h on S with respect to the local numbering of the non vanishing basis functions (local numbering is denoted by a superscript index and global numbering is denoted by a subscript index). Using the local coefficient vector and the local basis functions we obtain the *local representation* of u_h on S:

$$u_h(x) = \sum_{i=1}^{m} u_S^i \, \bar{\varphi}^i(\lambda(x)) \qquad \text{for all } x \in S.$$

In finite element codes, finite element functions are *not* evaluated at *world coordinates* x as in (1.4.3), but they are evaluated on a single element S at barycentric coordinates λ on S giving the value at the world coordinates $x(\lambda)$:

$$u_h(x(\lambda)) = \sum_{i=1}^{m} u_S^i \, \bar{\varphi}^i(\lambda) \qquad \text{for all } \lambda \in \bar{S}.$$

The mapping j_S, which allows the access to the local coefficient vector from the global one, is the relation between local DOFs and global DOFs. Global DOFs for a finite element space are stored on the mesh elements at positions which are known to the DOF administration of the finite element space. Thus, the corresponding DOF administration will provide information for the implementation of the function j_S and therefore information for reading/writing local coefficients from/to global coefficient vectors (compare Section 3.5.1).

1.4.3.1 Evaluating derivatives of finite element functions

Derivatives of finite element functions are also evaluated on single elements S in barycentric coordinates. In the calculation of derivatives in terms of the basis functions $\bar{\varphi}^i$, the Jacobian $\Lambda = \Lambda_S \in \mathbb{R}^{d \times \texttt{DIM\_OF\_WORLD}}$ of the barycentric coordinates on S is involved (we consider here only the case $d = \texttt{DIM\_OF\_WORLD} = n$):

$$\Lambda(x) = \begin{pmatrix} \lambda_{0,x_1}(x) & \lambda_{0,x_2}(x) & \cdots & \lambda_{0,x_n}(x) \\ \vdots & \vdots & & \vdots \\ \lambda_{d,x_1}(x) & \lambda_{d,x_2}(x) & \cdots & \lambda_{d,x_n}(x) \end{pmatrix} = \begin{pmatrix} - & \nabla\lambda_0(x)^t & - \\ & \vdots & \\ - & \nabla\lambda_d(x)^t & - \end{pmatrix}.$$

Now, using the chain rule we obtain for every function $\varphi \in \mathbb{P}_S$

$$\nabla\varphi(x) = \nabla\left(\bar{\varphi}(\lambda(x))\right) = \sum_{k=0}^{d} \bar{\varphi}_{,\lambda_k}(\lambda(x)) \nabla\lambda_k(x) = \Lambda^t(x) \nabla_\lambda \bar{\varphi}(\lambda(x)), \qquad x \in S$$

and

$$D^2\varphi(x) = \Lambda^t(x) D^2_\lambda \bar{\varphi}(\lambda(x)) \Lambda(x) + \sum_{k=0}^{d} D^2\lambda_k(x) \, \bar{\varphi}_{,\lambda_k}(\lambda(x)), \qquad x \in S.$$

For a simplex S with an affine parameterization F_S, $\nabla\lambda_k$ is constant on S and we get

$$\nabla\varphi(x) = \Lambda^t \nabla_\lambda \bar{\varphi}(\lambda(x)) \quad \text{and} \quad D^2\varphi(x) = \Lambda^t D^2_\lambda \bar{\varphi}(\lambda(x)) \Lambda, \qquad x \in S.$$

Using these equations, we immediately obtain

$$\nabla u_h(x) = \Lambda^t(x) \sum_{i=1}^{m} u_S^i \, \nabla_\lambda \bar{\varphi}^i(\lambda(x)), \qquad x \in S$$

and

$$D^2 u_h(x) = \Lambda^t(x) \sum_{i=1}^{m} u_S^i \, D_\lambda^2 \bar{\varphi}^i(\lambda(x)) \Lambda(x) + \sum_{k=0}^{d} D^2 \lambda_k(x) \sum_{i=1}^{m} u_S^i \, \bar{\varphi}_{,\lambda_k}^i(\lambda(x)), \qquad x \in S.$$

Since the evaluation is actually done in barycentric coordinates, this turns on S into

$$\nabla u_h(x(\lambda)) = \Lambda^t(x(\lambda)) \sum_{i=1}^{m} u_S^i \, \nabla_\lambda \bar{\varphi}^i(\lambda), \qquad \lambda \in \bar{S}$$

and

$$D^2 u_h(x(\lambda)) = \Lambda^t(x(\lambda)) \sum_{i=1}^{m} u_S^i \, D_\lambda^2 \bar{\varphi}^i(\lambda) \Lambda(x(\lambda)) + \sum_{k=0}^{d} D^2 \lambda_k(x(\lambda)) \sum_{i=1}^{m} u_S^i \, \bar{\varphi}_{,\lambda_k}^i(\lambda), \qquad \lambda \in \bar{S}.$$

Once the values of the basis functions, its derivatives, and the local coefficient vector $(u_S^1, \ldots, u_S^m)$ are known, the evaluation of u_h and its derivatives depends only on Λ and can be performed by some general evaluation routine (compare Section 4.3).

1.4.4 Interpolation and restriction during refinement and coarsening

We assume the following situation: Let S be a (non–parametric) simplex with children S_0 and S_1 generated by the bisection of S (compare Algorithm 1.1.5). Let X_S, X_{S_0,S_1} be the finite element spaces restricted to S and $S_0 \cup S_1$ respectively.

Throughout this section we denote by $\{\varphi^i\}_{i=1,\ldots,m}$ the basis of the coarse grid space X_S and by $\{\psi^j\}_{j=1,\ldots,k}$ the basis functions of $X_{S_0 \cup S_1}$. For a function $u_h \in X_S$ we denote by $\boldsymbol{u}_\varphi = (u_\varphi^1, \ldots, u_\varphi^m)^t$ the coefficient vector with respect to the basis $\{\varphi^i\}$ and for a function $v_h \in X_{S_0 \cup S_1}$ by $\boldsymbol{v}_\psi = (v_\psi^1, \ldots, v_\psi^k)^t$ the coefficient vector with respect to $\{\psi^j\}$.

We now derive equations for the transformation of local coefficient vectors for finite element functions that are interpolated during refinement and coarsening, and for vectors storing values of a linear functional applied to the basis functions on the fine grid which are restricted to the coarse functions during coarsening.

Let

$$\mathbb{I}_{S_0,S_1}^S : \; X_S \to X_{S_0 \cup S_1}$$

be an interpolation operator. For nested finite element spaces, i.e. $X_S \subset X_{S_0 \cup S_1}$, every coarse grid function $u_h \in X_S$ belongs also to $X_{S_0 \cup S_1}$, so the natural choice is $\mathbb{I}_{S_0,S_1}^S = id$ on X_S (for example, Lagrange finite elements are nested). The interpolants $\mathbb{I}_{S_0,S_1}^S \varphi^i$ can be written in terms of the fine grid basis functions

$$\mathbb{I}_{S_0,S_1}^S \varphi^i = \sum_{j=1}^{k} a_{ij} \psi^j$$

defining the $(m \times k)$–matrix

$$A = (a_{ij})_{\substack{i=1,\ldots,m \\ j=1,\ldots,k}}. \tag{1.5}$$

This matrix A is involved in the interpolation during refinement and the transformation of a linear functional during coarsening.

For the interpolation of functions during coarsening, we need an interpolation operator $\mathbb{I}_S^{S_0,S_1}\colon X_{S_0\cup S_1} \to X_S$. The interpolants $\mathbb{I}_S^{S_0,S_1}\psi^j$ of the fine grid functions can now be represented by the coarse grid basis

$$\mathbb{I}_S^{S_0,S_1}\psi^j = \sum_{i=1}^{m} b_{ij}\,\varphi^i$$

defining the $(m \times k)$–matrix

$$B = (b_{ij})_{\substack{i=1,\dots,m\\ j=1,\dots,k}}. \tag{1.6}$$

This matrix B is used for the interpolation during coarsening.

Both matrices depend only on the set of local basis functions on parent and children. Thus, they depend on the reference element $\bar{S}$ and one single bisection of the reference element into $\bar{S}_0$, $\bar{S}_1$. The matrices do depend on the local numbering of the basis functions on the children with respect to the parent. Thus, in 3d the matrices depend on the element type of S also (for an element of type `0` the numbering of basis functions on $\bar{S}_1$ differs from the numbering on $\bar{S}_1$ for an element of type `1`, `2`). But all matrices can be calculated by the local set of basis functions on the reference element.

DOFs can be shared by several elements, compare Section 1.3. Every DOF is connected to a basis function which has a support on all elements sharing this DOF. Each DOF refers to one coefficient of a finite element function, and this coefficient has to be calculated only once during interpolation. During the restriction of a linear functional, contributions from several basis functions are added to the coefficient of another basis function. Here we have to control that for two DOFs, both shared by several elements, the contribution of the basis function at one DOF is only added once to the other DOF and vice versa. This can only be done by performing the interpolation and restriction on the whole refinement/coarsening patch at the same time.

1.4.4.1 Interpolation during refinement

Let $\boldsymbol{u}_\varphi = (u_\varphi^1, \dots, u_\varphi^m)^t$ be the coefficient vector of a finite element function $u_h \in X_S$ with respect to $\{\varphi^i\}$, and let $\boldsymbol{u}_\psi = (u_\psi^1, \dots, u_\psi^k)^t$ the coefficient vector of $\mathbb{I}_{S_0,S_1}^S u_h$ with respect to $\{\psi^j\}$. Using matrix A defined in (1.5) we conclude

$$\mathbb{I}_{S_0,S_1}^S u_h = \sum_{i=1}^{m} u_\varphi^i\, \mathbb{I}_{S_0,S_1}^S \varphi^i = \sum_{i=1}^{m} u_\varphi^i \sum_{j=1}^{k} a_{ij}\,\psi^j = \sum_{j=1}^{k} \left(A^t \boldsymbol{u}_\varphi\right)_j \psi^j,$$

or equivalently

$$\boldsymbol{u}_\psi = A^t \boldsymbol{u}_\varphi.$$

A subroutine which interpolates a finite element function during refinement is an efficient implementation of this matrix–vector multiplication.

1.4.4.2 Restriction during coarsening

In an (Euler, e.g.) discretization of a time dependent problem, the term $(u_h^{\text{old}}, \varphi_i)_{L^2(\Omega)}$ appears on the right hand side of the discrete system, where u_h^{old} is the solution from the last time step. Such an expression can be calculated exactly, if the grid does not change from one time step to another. Assuming that the finite element spaces are nested, it is also possible to calculate

this expression exactly when the grid was refined, since u_h^{old} belongs to the fine grid finite element space also. Usually, during coarsening information is lost, since we can not represent u_h^{old} exactly on a coarser grid. But we can calculate $(u_h^{\text{old}}, \psi_i)_{L^2(\Omega)}$ exactly on the fine grid; using the representation of the coarse grid basis functions φ_i by the fine grid functions ψ_i, we can transform data during coarsening such that $(u_h^{\text{old}}, \varphi_i)_{L^2(\Omega)}$ is calculated exactly for the coarse grid functions too.

More general, assume that the finite element spaces are nested and that we can evaluate a linear functional f exactly for all basis functions of the fine grid. Knowing the values $\boldsymbol{f}_\psi = (\langle f, \psi^1\rangle, \ldots, \langle f, \psi^k\rangle)^t$ for the fine grid functions, we obtain with matrix A from (1.5) for the values $\boldsymbol{f}_\varphi = (\langle f, \varphi^1\rangle, \ldots, \langle f, \varphi^m\rangle)^t$ on the coarse grid

$$\boldsymbol{f}_\varphi = A\boldsymbol{f}_\psi$$

since

$$\langle f, \varphi^i\rangle = \langle f, \sum_{j=1}^{k} a_{ij}\psi^j\rangle = \sum_{j=1}^{k} a_{ij}\langle f, \psi^j\rangle$$

holds (here we used the fact, that $\mathbb{I}_{S_0,S_1}^S = id$ on X_S since the spaces are nested).

Thus, given a functional f which we can evaluate exactly for all basis functions of a grid $\tilde{\mathcal{S}}$ and its refinements, we can also calculate $\langle f, \varphi^i\rangle$ exactly for all basis functions φ^i of a grid $\mathcal{S}$ obtained by refinement and coarsening of $\tilde{\mathcal{S}}$ in the following way: First refine all elements of the grid that have to be refined; calculate $\langle f, \varphi\rangle$ for all basis functions φ of this intermediate grid; in the last step coarsen all elements that may be coarsened and restrict this vector during each coarsening step as described above.

In ALBERTA the assemblage of the discrete system inside the adaptive method can be split into three steps: one initialization step before refinement, the second step between refinement and coarsening, and the last, and usually most important, step after coarsening, when all grid modifications are completed, see Section 4.8.1. The second assemblage step can be used for an exact computation of a functional $\langle f, \varphi\rangle$ as described above.

1.4.4.3 Interpolation during coarsening

Finally, we can interpolate a finite element function during coarsening. The matrix for transforming the coefficient vector $\boldsymbol{u}_\psi = (u_\psi^1, \ldots, u_\psi^k)^t$ of a fine grid function u_h to the coefficient vector $\boldsymbol{u}_\varphi = (u_\varphi^1, \ldots, u_\varphi^m)^t$ of the interpolant on the coarse grid is given by matrix B defined in (1.6):

$$\begin{aligned}\mathbb{I}_S^{S_0,S_1} u_h = \mathbb{I}_S^{S_0,S_1} \sum_{j=1}^{k} u_\psi^j\, \psi^j &= \sum_{j=1}^{k} u_\psi^j\, \mathbb{I}_S^{S_0,S_1}\psi^j \\ = \sum_{j=1}^{k} u_\psi^j \sum_{i=1}^{m} b_{ij}\, \varphi^i &= \sum_{i=1}^{m} \left(\sum_{j=1}^{k} b_{ij}\, u_\psi^j\right) \varphi^i.\end{aligned}$$

Thus we have the following equation for the coefficient vector of the interpolant of u_h:

$$\boldsymbol{u}_\varphi = B\,\boldsymbol{u}_\psi.$$

In contrast to the interpolation during refinement and the above described transformation of a linear functional, information is lost during an interpolation to the coarser grid.

1.4.1 Example (Lagrange elements)**.** Lagrange finite elements are connected to Lagrange nodes x^i. For linear elements, these nodes are the vertices of the triangulation, and for quadratic elements, the vertices and the edge–midpoints. The Lagrange basis functions $\{\varphi^i\}$ satisfy

$$\varphi_i(x_j) = \delta_{ij} \qquad \text{for } i,j = 1,\dots,\dim X_h.$$

Consider the situation of a simplex S with children S_0, S_1. Let $\{\varphi^i\}_{i=1,\dots,m}$ be the Lagrange basis functions of X_S with Lagrange nodes $\{x^i_\varphi\}_{i=1,\dots,m}$ on S and $\{\psi^j\}_{j=1,\dots,k}$ be the Lagrange basis functions of $X_{S_0\cup S_1}$ with Lagrange nodes $\{x^j_\psi\}_{j=1,\dots,k}$ on $S_0 \cup S_1$. The Matrix A is then given by

$$a_{ij} = \varphi^i(x^j_\psi), \qquad i = 1,\dots,m,\ j = 1,\dots,k$$

and matrix B is given by

$$b_{ij} = \psi^j(x^i_\varphi), \qquad i = 1,\dots,m,\ j = 1,\dots,k.$$

1.4.5 Discretization of 2nd order problems

In this section we describe the assembling of the discrete system in detail. We consider the following second order differential equation in divergence form:

$$Lu := -\nabla \cdot A\nabla u + b\cdot\nabla u + c\,u = f \qquad \text{in } \Omega, \tag{1.7a}$$

$$u = g \qquad \text{on } \Gamma_D, \tag{1.7b}$$

$$\nu_\Omega \cdot A\nabla u = 0 \qquad \text{on } \Gamma_N, \tag{1.7c}$$

where $A \in L^\infty(\Omega;\mathbb{R}^{n\times n})$, $b \in L^\infty(\Omega;\mathbb{R}^n)$, $c \in L^\infty(\Omega)$, and $f \in L^2(\Omega)$. By $\Gamma_D \subset \partial\Omega$ (with $|\Gamma_D| \neq 0$) we denote the Dirichlet boundary and we assume that the Dirichlet boundary values $g\colon \Gamma_D \to \mathbb{R}$ have an extension to some function $g \in H^1(\Omega)$.

By $\Gamma_N = \partial\Omega\backslash\Gamma_D$ we denote the Neumann boundary, and by ν_Ω we denote the outer unit normal vector on $\partial\Omega$. The boundary condition (1.7c) is a so called *natural* Neumann condition.

Equations (1.7) describe not only a simple model problem. The same kind of equations result from a linearization of nonlinear elliptic problems (for example by a Newton method) as well as from a time discretization scheme for (non–) linear parabolic problems.

Setting

$$X = H^1(\Omega) \qquad \text{and} \qquad \mathring{X} = \{v \in H^1(\Omega);\ v = 0 \text{ on } \Gamma_D\}$$

this equation has the following weak formulation: We are looking for a solution $u \in X$, such that $u \in g + \mathring{X}$ and

$$\int_\Omega (\nabla\varphi(x))\cdot A(x)\nabla u(x) + \varphi(x)\,b(x)\cdot\nabla u(x) + c(x)\,\varphi(x)\,u(x)\,dx = \int_\Omega f(x)\,\varphi(x)\,dx \tag{1.8}$$

for all $\varphi \in \mathring{X}$

Denoting by $\mathring{X}^*$ the dual space of $\mathring{X}$ we identify the differential operator L with the linear operator $L \in \mathcal{L}(X,\mathring{X}^*)$ defined by

$$\langle Lv,\,\varphi\rangle_{\mathring{X}^*\times\mathring{X}} := \int_\Omega \nabla\varphi\cdot A\nabla v + \int_\Omega \varphi\,b\cdot\nabla v + \int_\Omega c\,\varphi\,v \qquad \text{for all } v,\varphi \in \mathring{X}$$

and the right hand side f with the linear functional $f \in \mathring{X}^*$ defined by

$$\langle F, \varphi \rangle_{\mathring{X}^* \times \mathring{X}} := \int_\Omega f\,\varphi \qquad \text{for all } \varphi \in \mathring{X}.$$

With these identifications we use the following reformulation of (1.8): Find $u \in X$ such that

$$u \in g + \mathring{X} : \qquad L\,u = f \qquad \text{in } \mathring{X}^* \tag{1.9}$$

holds.

Suitable assumptions on the coefficients imply that L is elliptic, i.e. there is a constant $C = C_{A,b,c,\Omega}$ such that

$$\langle L\,\varphi, \varphi \rangle_{\mathring{X}^* \times \mathring{X}} \geq C\,\|\varphi\|_X^2 \qquad \text{for all } \varphi \in \mathring{X}.$$

The existence of a unique solution $u \in X$ of (1.9) is then a direct consequence of the Lax–Milgram–Theorem.

We consider a finite dimensional subspace $X_h \subset X$ for the discretization of (1.9) with $N = \dim X_h$. We set $\mathring{X}_h = X_h \cap \mathring{X}$ with $\mathring{N} = \dim \mathring{X}_h$. Let $g_h \in X_h$ be an approximation of $g \in X$. A discrete solution of (1.9) is then given by: Find $u_h \in X_h$ such that

$$u_h \in g_h + \mathring{X}_h : \qquad L\,u_h = f \qquad \text{in } \mathring{X}_h^*, \tag{1.10}$$

i.e.

$$u_h \in g_h + \mathring{X}_h : \qquad \langle L u_h, \varphi_h \rangle_{\mathring{X}_h^* \times \mathring{X}_h} = \langle f, \varphi_h \rangle_{\mathring{X}_h^* \times \mathring{X}_h} \quad \text{for all } \varphi_h \in \mathring{X}_h$$

holds. If L is elliptic, we have a unique discrete solution $u_h \in X_h$ of (1.10), again using the Lax–Milgram–Theorem.

Choose a basis $\{\varphi_1, \ldots, \varphi_N\}$ of X_h such that $\{\varphi_1, \ldots, \varphi_{\mathring{N}}\}$ is a basis of $\mathring{X}_h$. For a function $v_h \in X_h$ we denote by $\boldsymbol{v} = (v_1, \ldots, v_N)$ the coefficient vector of v_h with respect to the basis $\{\varphi_1, \ldots, \varphi_N\}$, i.e.

$$v_h = \sum_{j=1}^{N} v_j \varphi_j.$$

Using (1.10) with test functions φ_i, $i = 1, \ldots, \mathring{N}$, we get the following N equations for the coefficient vector $\boldsymbol{u} = (u_1, \ldots, u_N)$ of u_h:

$$\begin{aligned} \sum_{j=1}^{N} u_j \langle L\varphi_j, \varphi_i \rangle_{\mathring{X}_h^* \times \mathring{X}_h} &= \langle f, \varphi_i \rangle_{\mathring{X}_h^* \times \mathring{X}_h} && \text{for } i = 1, \ldots, \mathring{N}, \\ u_i &= g_i && \text{for } i = \mathring{N}+1, \ldots, N. \end{aligned}$$

Defining the *system matrix* $\boldsymbol{L}$ by

$$\boldsymbol{L} := \begin{bmatrix} \langle L\varphi_1, \varphi_1 \rangle & \ldots & \langle L\varphi_{\mathring{N}}, \varphi_1 \rangle & \langle L\varphi_{\mathring{N}+1}, \varphi_1 \rangle & \ldots & \langle L\varphi_N, \varphi_1 \rangle \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \langle L\varphi_1, \varphi_{\mathring{N}} \rangle & \ldots & \langle L\varphi_{\mathring{N}}, \varphi_{\mathring{N}} \rangle & \langle L\varphi_{\mathring{N}+1}, \varphi_{\mathring{N}} \rangle & \ldots & \langle L\varphi_N, \varphi_{\mathring{N}} \rangle \\ 0 & \ldots & 0 & 1 & 0 \ldots & 0 \\ 0 & \ldots & 0 & 0 & 1 \ldots & 0 \\ \vdots & \ddots & 0 & 0 & 0 & \ddots & \vdots \\ 0 & \ldots & 0 & 0 & 0 & \ldots & 1 \end{bmatrix} \tag{1.11}$$

and the *right hand side vector* or *load vector* $\boldsymbol{f}$ by

$$\boldsymbol{f} := \begin{bmatrix} \langle f, \varphi_1 \rangle \\ \vdots \\ \langle f, \varphi_{\mathring{N}} \rangle \\ g_{\mathring{N}+1} \\ \vdots \\ g_N \end{bmatrix}, \tag{1.12}$$

we can write the discrete equations as the linear $N \times N$ system

$$\boldsymbol{L}\,\boldsymbol{u} = \boldsymbol{f}, \tag{1.13}$$

which has to be assembled and solved numerically.

1.4.6 Discretization of coupled vector valued problems

This section describes the discretization and assembling of coupled vector valued problems. Consider the following artificial coupled Poisson problem:

Let $C \in \mathbb{R}^{n\times n}$ be a regular coupling matrix, $f \in L^2(\Omega;\mathbb{R}^n)$ a given right hand side, and $g\colon \partial\Omega \to \mathbb{R}^n$ suitable boundary values. Find $u : \Omega \to \mathbb{R}^n$ with

$$-\sum_{\nu=1}^{n} C_{\mu\nu}\Delta u_\nu = f_\mu \qquad \text{in } \Omega \text{ for } \mu = 1,\dots,n \tag{1.14a}$$

$$u = g \qquad \text{on } \partial\Omega, \tag{1.14b}$$

By a left multiplication with C^{-1} this problem decouples into a set of independent scalar Poisson problems, for which we could apply the same existence and uniqueness theory as above. However, we will refrain from doing this in order to illustrate the concepts of this section. Generally, the weak form of a coupled system of linear second order equations can be written as follows:

Define vector valued spaces $X = H^1(\Omega;\mathbb{R}^n)$, $\mathring{X} = H^1_0(\Omega;\mathbb{R}^n)$. Find a solution $u \in X$, such that $u \in g + \mathring{X}$ and

$$\langle Lu, \varphi\rangle_{\mathring{X}^*\times\mathring{X}} := \sum_{\mu,\nu=1}^{n} \int_\Omega \nabla\varphi_\mu \cdot A^{\mu\nu}\nabla u_\nu + \varphi_\mu\, b^{\mu\nu}\cdot\nabla u_\nu + c^{\mu\nu}\,\varphi_\mu\, u_\nu\, dx = \int_\Omega f\cdot\varphi\, dx \tag{1.15}$$

for all $\varphi \in \mathring{X}$.

To obtain the weak form of problem (1.14) for example, we would set $b := 0$, $c := 0$ and $A^{\mu\nu}_{ij} := \delta_{ij}C_{\mu\nu}$. The next step is to derive a suitable linear system for a discretization as in the last section.

As mentioned in Section 1.4.2, basis functions are always scalar-valued. To gain a vector valued finite element space X_h, we use vector valued coefficients. Choose a set of scalar basis functions $\{\varphi_1,\dots,\varphi_N\}$ as above. For a function $v_h \in X_h$ we denote by $\boldsymbol{v} = (v_1,\dots,v_N)$ the coefficient vector of v_h. Each v_j is now itself a vector $v_j = (v_{j\mu})_{\mu=1}^n$. Thus, we have the following decomposition:

$$v_h = \sum_{j=1}^{N} v_j\varphi_j.$$

Define $\varphi_j^\mu := (\delta_{\mu\nu}\varphi_j)_{\nu=1}^n$. The discrete problem can now be written as a set of linear equations for the coefficients $u_{j\mu}$:

$$\sum_{j=1}^{N}\sum_{\mu=1}^{n} u_{j\mu}\left\langle L\varphi_j^\mu,\, \varphi_i^\nu\right\rangle_{\mathring{X}_h^*\times\mathring{X}_h} = \left\langle f,\, \varphi_i^\nu\right\rangle_{\mathring{X}_h^*\times\mathring{X}_h} \qquad \text{for } i=1,\dots,\mathring{N};\nu=1,\dots,n,$$

$$u_{i\nu} = g_{i\nu} \qquad \text{for } i=\mathring{N}+1,\dots,N;\nu=1,\dots,n.$$

The corresponding system matrix $\boldsymbol{L}$ is defined by

$$\boldsymbol{L} := \begin{bmatrix} \boldsymbol{L}^{11} & \dots & \boldsymbol{L}^{1\mathring{N}} & \boldsymbol{L}^{1,\mathring{N}+1} & \dots & \boldsymbol{L}^{1N} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{L}^{\mathring{N}1} & \dots & \boldsymbol{L}^{\mathring{N}\mathring{N}} & \boldsymbol{L}^{\mathring{N},\mathring{N}+1} & \dots & \boldsymbol{L}^{\mathring{N}N} \\ 0 & \dots & 0 & \boldsymbol{I} & 0 \dots 0 & \\ 0 & \dots & 0 & 0 & \boldsymbol{I} \dots 0 & \\ \vdots & \ddots & 0 & 0 & 0 \ddots \vdots & \\ 0 & \dots & 0 & 0 & 0 \dots \boldsymbol{I} & \end{bmatrix} \tag{1.16}$$

with $\boldsymbol{I} \in \mathbb{R}^{n\times n}$ an identity matrix and

$$\boldsymbol{L}^{ij} := (\langle L\varphi_j^\mu,\, \varphi_i^\nu\rangle_{\mathring{X}_h^*\times\mathring{X}_h})_{\mu,\nu=1}^n.$$

The load vector $\boldsymbol{f}$ is defined by

$$\boldsymbol{f} := \begin{bmatrix} \langle f,\, \varphi_1^1\rangle \\ \vdots \\ \langle f,\, \varphi_1^n\rangle \\ \vdots \\ \langle f,\, \varphi_{\mathring{N}}^1\rangle \\ \vdots \\ \langle f,\, \varphi_{\mathring{N}}^n\rangle \\ g_{\mathring{N}+1,1} \\ \vdots \\ g_{Nd} \end{bmatrix}. \tag{1.17}$$

The problem can now be written as the linear $Nd \times Nd$ system

$$\boldsymbol{L}\,\boldsymbol{u} = \boldsymbol{f}, \tag{1.18}$$

which has to be assembled and solved. The organization of vectors and matrices using small n-size blocks as components was chosen with the goal of efficient cache usage during matrix-vector multiplication.

1.4.7 Numerical quadrature

For the assemblage of the system matrix and right hand side vector of the linear system (1.13), we have to compute integrals, for example

$$\int_\Omega f(x)\varphi_i(x)\,dx.$$

For general data A, b, c, and f, these integrals can not be calculated exactly. Quadrature formulas have to be used in order to calculate the integrals approximately. Numerical integration in finite element methods is done by looping over all grid elements and using a quadrature formula on each element.

1.4.2 Definition (Numerical quadrature)**.** A *numerical quadrature* $\hat{Q}$ on $\hat{S}$ is a set $\{(w_k, \lambda_k) \in \mathbb{R} \times \mathbb{R}^{d+1}; k = 0, \dots, n_Q - 1\}$ of weights w_k and quadrature points $\lambda_k \in \bar{S}$ (i.e. given in barycentric coordinates) such that

$$\int_{\hat{S}} f(\hat{x})\, d\hat{x} \approx \hat{Q}(f) := \sum_{k=0}^{n_Q-1} w_k f(\hat{x}(\lambda_k)).$$

It is called *exact of degree* p for some $p \in \mathbb{N}$ if

$$\int_{\hat{S}} q(\hat{x})\, d\hat{x} = \hat{Q}(q) \qquad \text{for all } q \in \mathbb{P}_p(\hat{S}).$$

It is called *stable* if

$$w_k > 0 \qquad \text{for all } k = 0, \dots, n_Q - 1.$$

1.4.3 Remark. A given numerical quadrature $\hat{Q}$ on $\hat{S}$ defines for each element S a numerical quadrature Q_S. Using the transformation rule we define Q_S on an element S which is parameterized by $F_S\colon \hat{S} \to S$ and a function $f : S \to \mathbb{R}$:

$$\int_S f(x)\, dx \approx Q_S(f) := \hat{Q}((f \circ F_S)|\det DF_S|) = \sum_{k=0}^{n_Q-1} w_k f(x(\lambda_k))|\det DF_S(\hat{x}(\lambda_k)|.$$

For a simplex S this results in

$$Q_S(f) = d!\,|S| \sum_{k=0}^{n_Q-1} w_k f(x(\lambda_k)).$$

1.4.8 Finite element discretization of 2nd order problems

Let $\bar{\mathbb{P}}$ be a finite dimensional function space on $\bar{S}$ with basis $\{\bar{\varphi}^1, \dots, \bar{\varphi}^m\}$. For a conforming finite element discretization of (1.8) we use the finite element space $X_h = X_h(\mathcal{S}, \bar{\mathbb{P}}, X)$. For this space $\mathring{X}_h$ is given by $X_h(\mathcal{S}, \bar{\mathbb{P}}, \mathring{X})$.

By the relation (1.4a) for global and local basis functions, we obtain for the jth component of the right hand side vector $\boldsymbol{f}$

$$\begin{aligned}\langle f, \varphi_j \rangle &= \int_\Omega f(x)\, \varphi_j(x)\, dx = \sum_{S\in\mathcal{S}} \int_S f(x)\, \varphi_j(x)\, dx = \sum_{\substack{S\in\mathcal{S}\\ S\subset \operatorname{supp}(\varphi_j)}} \int_S f(x)\bar{\varphi}^{i_S(j)}(\lambda(x))\, dx \\ &= \sum_{\substack{S\in\mathcal{S}\\ S\subset \operatorname{supp}(\varphi_j)}} \int_{\hat{S}} f(F_S(\hat{x}))\bar{\varphi}^{i_S(j)}(\lambda(\hat{x}))|\det DF_S(\hat{x})|\, d\hat{x},\end{aligned}$$

where S is parameterized by $F_S\colon \hat{S} \to S$. The above sum is reduced to a sum over all $S \subset \operatorname{supp}(\varphi_j)$ which are only few elements due to the small support of φ_j.

The right hand side vector can be assembled in the following way: First, the right hand side vector is initialized with zeros. For each element S of $\mathcal{S}$ we calculate the *element load vector* $\boldsymbol{f}_S = (f_S^1, \ldots, f_S^m)^t$, where

$$f_S^i = \int_{\hat{S}} f(F_S(\hat{x}))\bar{\varphi}^i(\lambda(\hat{x}))|\det DF_S(\hat{x})|\,d\hat{x}, \qquad i = 1, \ldots, m. \tag{1.19}$$

Denoting again by $j_S : \{1, \ldots, m\} \to J_S$ the function which connects the local DOFs with the global DOFs (defined in (1.4b)), the values f_S^i are then added to the $j_S(i)$th component of the right hand side vector $\boldsymbol{f}$, $i = 1, \ldots, m$.

For general f, the integrals in (1.19) can not be calculated exactly and we have to use a quadrature formula for the approximation of the integrals (compare Section 1.4.7). Given a numerical quadrature $\hat{Q}$ on $\hat{S}$ we approximate

$$f_S^i \approx \hat{Q}\left((f \circ F_S)\,(\bar{\varphi}^i \circ \lambda)|\det DF_S|\right) = \sum_{k=0}^{n_Q-1} w_k f(x(\lambda_k))\,\bar{\varphi}^i(\lambda_k)|\det DF_S(\hat{x}(\lambda_k)|. \tag{1.20}$$

For a simplex S this is simplified to

$$f_S^i \approx d!\,|S| \sum_{k=0}^{n_Q-1} w_k f(x(\lambda_k))\,\bar{\varphi}^i(\lambda_k).$$

In ALBERTA, information about values of basis functions and its derivatives as well as information about the connection of global and local DOFs (i.e. information about j_S) is stored in special data structures for local basis functions (compare Section 3.5). By such information, the element load vector can be assembled by a general routine if a function for the evaluation of the right hand side is supplied. For parametric elements, a function for evaluating $|\det DF_S|$ is additionally needed. The assemblage into the global load vector $\boldsymbol{f}$ can again be performed automatically, using information about the connection of global and local DOFs.

The calculation of the system matrix is also done element–wise. Additionally, we have to handle derivatives of basis functions. Looking first at the second order term we obtain by the chain rule (1.4.3.1) and the relation (1.4) for global and local basis functions

$$\begin{aligned}\int_S \nabla\varphi_i(x)\cdot A(x)\nabla\varphi_j(x)\,dx &= \int_S \nabla(\bar{\varphi}^{i_S(i)}\circ\lambda)(x)\cdot A(x)\nabla(\bar{\varphi}^{i_S(j)}\circ\lambda)(x)\,dx \\ &= \int_S \nabla_\lambda\bar{\varphi}^{i_S(i)}(\lambda(x))\cdot(\Lambda(x)\,A(x)\,\Lambda^t(x))\nabla_\lambda\bar{\varphi}^{i_S(j)}(\lambda(x))\,dx,\end{aligned}$$

where Λ is the Jacobian of the barycentric coordinates λ on S. In the same manner we obtain for the first and zero order terms

$$\int_S \varphi_i(x)\,b(x)\cdot\nabla\varphi_j(x)\,dx = \int_S \bar{\varphi}^{i_S(i)}(\lambda(x))\,(\Lambda(x)\,b(x))\cdot\nabla_\lambda\bar{\varphi}^{i_S(j)}(\lambda(x))\,dx$$

and

$$\int_S c(x)\,\varphi_i(x)\,\varphi_j(x)\,dx = \int_S c(x)\,\bar{\varphi}^{i_S(i)}(\lambda(x))\,\bar{\varphi}^{i_S(j)}(\lambda(x))\,dx.$$

Using on S the abbreviations

$$\begin{aligned}\bar{A}(\lambda) &:= (\bar{a}_{kl}(\lambda))_{k,l=0,\dots,d} := |\det DF_S(\hat{x}(\lambda))|\,\Lambda(x(\lambda))\,A(x(\lambda))\,\Lambda^t(x(\lambda)),\\ \bar{b}(\lambda) &:= \big(\bar{b}_l(\lambda)\big)_{l=0,\dots,d} := |\det DF_S(\hat{x}(\lambda))|\,\Lambda(x(\lambda))\,b(x(\lambda)), \quad \text{and}\\ \bar{c}(\lambda) &:= |\det DF_S(\hat{x}(\lambda))|\,c(x(\lambda))\end{aligned}$$

and transforming the integrals to the reference simplex, we can write the *element matrix* $\boldsymbol{L}_S = (L_S^{ij})_{i,j=1,\dots,m}$ as

$$\begin{aligned}L_S^{ij} = &\int_{\hat{S}} \nabla_\lambda \bar{\varphi}^i(\lambda(\hat{x}))\cdot \bar{A}(\lambda(\hat{x}))\,\nabla_\lambda \bar{\varphi}^j(\lambda(\hat{x}))\,d\hat{x} + \int_{\hat{S}} \bar{\varphi}^i(\lambda(\hat{x}))\,\bar{b}(\lambda(\hat{x}))\cdot\nabla_\lambda\bar{\varphi}^j(\lambda(\hat{x}))\,d\hat{x}\\ &+\int_{\hat{S}} \bar{c}(\lambda(\hat{x}))\,\bar{\varphi}^i(\lambda(\hat{x}))\,\bar{\varphi}^j(\lambda(\hat{x}))\,d\hat{x},\end{aligned} \tag{1.21}$$

or writing the matrix–vector and vector–vector products explicitly

$$\begin{aligned}L_S^{ij} = &\sum_{k,l=0}^{d}\int_{\hat{S}} \bar{a}_{kl}(\lambda(\hat{x}))\,\bar{\varphi}^i_{,\lambda_k}(\lambda(\hat{x}))\,\bar{\varphi}^j_{,\lambda_l}(\lambda(\hat{x}))\,d\hat{x} + \sum_{l=0}^{d}\int_{\hat{S}} \bar{b}_l(\lambda(\hat{x}))\,\bar{\varphi}^i(\lambda(\hat{x}))\,\bar{\varphi}^j_{,\lambda_l}(\lambda(\hat{x}))\,d\hat{x}\\ &+\int_{\hat{S}} \bar{c}(\lambda(\hat{x}))\,\bar{\varphi}^i(\lambda(\hat{x}))\,\bar{\varphi}^j(\lambda(\hat{x}))\,d\hat{x},\end{aligned}$$

$i,j = 1,\dots,m$. Using quadrature formulas $\hat{Q}_2$, $\hat{Q}_1$, and $\hat{Q}_0$ for the second, first and zero order term we approximate the element matrix

$$L_S^{ij} \approx \hat{Q}_2\left(\sum_{k,l=0}^{d}(\bar{a}_{kl}\bar{\varphi}^i_{,\lambda_k}\,\bar{\varphi}^j_{,\lambda_l})\circ\lambda\right) + \hat{Q}_1\left(\sum_{l=0}^{d}(\bar{b}_l\,\bar{\varphi}^i\,\bar{\varphi}^j_{,\lambda_l})\circ\lambda\right) + \hat{Q}_0\Big((\bar{c}\,\bar{\varphi}^i\,\bar{\varphi}^j)\circ\lambda\Big),$$

$i,j = 1,\dots,m$. Having access to functions for the evaluation of

$$\bar{a}_{kl}(\lambda_q), \quad \bar{b}_l(\lambda_q), \quad \bar{c}(\lambda_q)$$

at all quadrature points λ_q on S, $\boldsymbol{L}_S$ can be computed by some general routine. The assemblage into the system matrix can also be done automatically (compare the assemblage of the load vector).

1.4.4 Remark. Due to the small support of the global basis function, the system matrix is a sparse matrix, i.e. the maximal number of entries in all matrix rows is much smaller than the size of the matrix. Special data structures are needed for an efficient storage of sparse matrices and they are described in Section 3.3.4.

1.4.5 Remark. The calculation of the gradient of the barycentric coordinates $\Lambda(x(\lambda))$ usually involves the determinant of the parameterization's Jacobian $|\det DF_S(\hat{x}(\lambda))|$. Thus, a calculation of $|\det DF_S(\hat{x}(\lambda))|\,\Lambda(x(\lambda))\,A(x(\lambda))\,\Lambda^t(x(\lambda))$ may be much faster than the calculation of $\Lambda(x(\lambda))\,A(x(\lambda))\,\Lambda^t(x(\lambda))$ only; the same holds for the first order term.

Assuming that the coefficients A, b, and c are piecewise constant on a non–parametric triangulation, $\bar{A}(\lambda)$, $\bar{b}(\lambda)$, and $\bar{c}(\lambda)$ are constant on each simplex S and thus simplified to

$$\bar{A}_S = (\bar{a}_{kl})_{k,l=0,\dots,d} = d!|S|\,\Lambda\,A_{|S}\,\Lambda^t, \quad \bar{b}_S = \big(\bar{b}_l\big)_{l=0,\dots,d} = d!|S|\,\Lambda\,b_{|S}, \quad \bar{c}_S = d!|S|\,c_{|S}.$$

For the approximation of the element matrix by quadrature we then obtain

$$\boldsymbol{L}_S^{ij} \approx \sum_{k,l=0}^{d} \bar{a}_{kl} \hat{Q}_2\Big((\bar{\varphi}^i_{,\lambda_k}\, \bar{\varphi}^j_{,\lambda_l}) \circ \lambda\Big) + \sum_{l=0}^{d} \bar{b}_l\, \hat{Q}_1\Big((\bar{\varphi}^i\, \bar{\varphi}^j_{,\lambda_l}) \circ \lambda\Big) + \bar{c}_S\, \hat{Q}_0\Big((\bar{\varphi}^i\, \bar{\varphi}^j) \circ \lambda\Big) \tag{1.22}$$

$i, j = 1, \ldots, m$. Here, the numerical quadrature is only applied for approximating integrals of the basis functions on the standard simplex. Theses values can be computed only once, and can then be used on each simplex S. This will speed up the assembling of the system matrix. Additionally, for polynomial basis functions we can use quadrature formulas which integrate these integrals exactly. So far we have only considered the case of scalar problems. The transition to (coupled) vector valued problems is straight forward and simply involves two more indices. The entries of the element matrix are now $d \times d$ matrices themselves:

$$\begin{aligned} L^{ij}_{S,\mu\nu} &:= \int_S \nabla\varphi_i(x) \cdot A^{\mu\nu}(x) \nabla\varphi_j(x) + \varphi_i(x)\, b^{\mu\nu}(x) \cdot \nabla\varphi_j(x) + c^{\mu\nu}(x)\, \varphi_i(x)\, \varphi_j(x)\, dx \\ &= \int_{\hat{S}} \nabla_\lambda \bar{\varphi}^i(\lambda(\hat{x})) \cdot \bar{A}^{\mu\nu}(\lambda(\hat{x}))\, \nabla_\lambda \bar{\varphi}^j(\lambda(\hat{x}))\, d\hat{x} + \int_{\hat{S}} \bar{\varphi}^i(\lambda(\hat{x}))\, \bar{b}^{\mu\nu}(\lambda(\hat{x})) \cdot \nabla_\lambda \bar{\varphi}^j(\lambda(\hat{x}))\, d\hat{x} \\ &\quad + \int_{\hat{S}} \bar{c}^{\mu\nu}(\lambda(\hat{x}))\, \bar{\varphi}^i(\lambda(\hat{x}))\, \bar{\varphi}^j(\lambda(\hat{x}))\, d\hat{x}, \end{aligned}$$

with

$$\begin{aligned} \bar{A}^{\mu\nu}(\lambda) &:= \big(\bar{a}^{\mu\nu}_{kl}(\lambda)\big)_{k,l=0,\ldots,d} := |\det DF_S(\hat{x}(\lambda))|\, \Lambda(x(\lambda))\, A^{\mu\nu}(x(\lambda))\, \Lambda^t(x(\lambda)), \\ \bar{b}^{\mu\nu}(\lambda) &:= \big(\bar{b}^{\mu\nu}_{l}(\lambda)\big)_{l=0,\ldots,d} := |\det DF_S(\hat{x}(\lambda))|\, \Lambda(x(\lambda))\, b^{\mu\nu}(x(\lambda)), \quad \text{and} \\ \bar{c}^{\mu\nu}(\lambda) &:= |\det DF_S(\hat{x}(\lambda))|\, c^{\mu\nu}(x(\lambda)) \end{aligned}$$

for $\mu, \nu = 1, \ldots, d$. The approximation of the integrals using quadratures is done analogously to the scalar case.

As a result, using information about values of basis functions and their derivatives, and information about the connection of global and local DOFs, the linear system can be assembled automatically by some general routines. Only functions for the evaluation of given data have to be provided for special applications. The general assemble routines are described in Section 4.7.

1.5 Adaptive Methods

The aim of adaptive methods is the generation of a mesh which is adapted to the problem such that a given criterion, like a tolerance for the estimated error between exact and discrete solution, if fulfilled by the finite element solution on this mesh. An optimal mesh should be as coarse as possible while meeting the criterion, in order to save computing time and memory requirements. For time dependent problems, such an adaptive method may include mesh changes in each time step and control of time step sizes.

The philosophy implemented in ALBERTA is to change meshes successively by local refinement or coarsening, based on error estimators or error indicators, which are computed a posteriori from the discrete solution and given data on the current mesh.

1.5.1 Adaptive method for stationary problems

Let us assume that a triangulation $\mathcal{S}$ of Ω, a finite element solution $u_h \in X_h$ to an elliptic problem, and an a posteriori error estimate

$$\|u - u_h\| \;\leq\; \eta(u_h) \;=\; \left(\sum_{S \in \mathcal{S}} \eta_S(u_h)^p \right)^{1/p}, \qquad p \in [1, \infty) \tag{1.23}$$

on this mesh are given. If *tol* is a given allowed tolerance for the error, and $\eta(u_h) > tol$, the problem arises

- where to refine the mesh in order to reduce the error,
- while at the same time the number of unknowns should not become too large.

A global refinement of the mesh would lead to the best error reduction, but the amount of new unknowns might be much larger than needed to reduce the error below the given tolerance. Using local refinement, we hope to do much better.

The design of an "optimal" mesh, where the number of unknowns is as small as possible to keep the error below the tolerance, is an open problem and will probably be much too costly. Especially in the case of linear problems, the design of an optimal mesh will be much more expensive than the solution of the original problem, since the mesh optimization is a highly nonlinear problem. Usually, some heuristic arguments are then used in the algorithm. The aim is to produce a result that is "not too far" from an optimal mesh, but with a relatively small amount of additional work to generate it.

Several adaptive strategies are proposed in the literature, that give criteria which mesh elements should be marked for refinement. All strategies are based on the idea of an equidistribution of the local error to all mesh elements. Babuška and Rheinboldt [3] motivate that a mesh is almost optimal when the local errors are approximately equal for all elements. So, elements where the error indicator is large will be marked for refinement, while elements with a small error indicator are left unchanged.

The general outline of the adaptive algorithm for a stationary problem is the following. Starting from an initial triangulation $\mathcal{S}_0$, we produce a sequence of triangulations $\mathcal{S}_k$, for $k = 1, 2, \ldots$, until the estimated error is below the given tolerance:

1.5.1 Algorithm (General adaptive refinement strategy)**.**

```
Start with S_0 and error tolerance tol
  k := 0
  solve the discrete problem on S_k
  compute global error estimate η and local indicators η_S, S ∈ S_k
  while η > tol do
    mark elements for refinement (or coarsening)
    adapt mesh S_k, producing S_{k+1}
    k := k + 1
    solve the discrete problem on S_k
    compute global error estimate η and local indicators η_S, S ∈ S_k
  end while
```

1.5.2 Mesh refinement strategies

Since a discrete problem has to be solved in every iteration of this algorithm, the number of iterations should be as small as possible. Thus, the marking strategy should select not too few mesh elements for refinement in each cycle. On the other hand, not much more elements should be selected than is needed to reduce the error below the given tolerance.

In the sequel, we describe several marking strategies that are commonly used in adaptive finite element methods.

The basic assumption for all marking strategies is the fact that the mesh is "optimal" when the local error is the same for all elements of the mesh. This optimality can be shown under some heuristic assumptions, see [3]. Since the true error is not know we try to equidistribute the local error indicators. This is motivated by the lower bound for error estimators of elliptic problems. This lower bound ensures that the local error is large if the local indicator is large and data of the problem is sufficiently resolved [2, 57]. As a consequence, elements with a large local error indicator should be refined, while elements with a very small local error indicator may be coarsened.

Global refinement: The simplest strategy is not really "adaptive" at all, at least not producing a locally refined mesh. It refines the mesh globally, until the given tolerance is reached.

If an a priori estimate for the error in terms of the maximal size of a mesh element h is known, where the error is bounded by a positive power of h, and if the error estimate tends to zero if the error gets smaller, then this strategy is guaranteed to produce a mesh and a discrete solution which meets the error tolerance.

But, in most cases, global refinement produces far too much mesh elements than are needed to meet the error tolerance.

Maximum strategy: Another very simple strategy is the maximum strategy. A threshold $\gamma \in (0,1)$ is given, and all elements $S \in \mathcal{S}_k$ with

$$\eta_S \;>\; \gamma \max_{S' \in \mathcal{S}_k} \eta_{S'} \tag{1.24}$$

are marked for refinement. A small γ leads to more refinement and maybe non–optimal meshes, while a large γ leads to more cycles until the error tolerance is reached, but usually produces a mesh with less unknowns. Typically, a threshold value $\gamma = 0.5$ is used when the power p in (1.23) is $p = 2$ [56, 58].

1.5.2 Algorithm (Maximum strategy)**.**

```
Given parameter γ ∈ (0,1)
η_max := max(η_S, S ∈ S_k)
for all S in S_k do
   if η_S > γ η_max then mark S for refinement
end for
```

Equidistribution strategy: Let N_k be the number of mesh elements in $\mathcal{S}_k$. If we assume that the error indicators are equidistributed over all elements, i. e. $\eta_S = \eta_{S'}$ for all $S, S' \in \mathcal{S}_k$, then

$$\eta \;=\; \left(\sum_{S \in \mathcal{S}_h} \eta_S^p \right)^{1/p} \;=\; N_k^{1/p}\, \eta_S \;\stackrel{!}{=}\; tol \qquad \text{and} \qquad \eta_S \;=\; \frac{tol}{N_k^{1/p}}\,.$$

We can try to reach this equidistribution by refining all elements where it is violated because the error indicator is larger than $tol/N_k^{1/p}$. To make the procedure more robust, a parameter $\theta \in (0,1)$, $\theta \approx 1$, is included in the method.

1.5.3 Algorithm (Equidistribution strategy[31]).

```
Start with parameter θ ∈ (0,1), θ ≈ 1
for all S in S_k do
   if η_S > θ tol/N_k^{1/p} then mark S for refinement
end for
```

If the error η is already near tol, then the choice $\theta = 1$ leads to the selection of only very few elements for refinement, which results in more iterations of the adaptive process. Thus, θ should be chosen smaller than 1, for example $\theta = 0.9$. Additionally, this accounts for the fact that the number of mesh elements increases, i. e. $N_{k+1} > N_k$, and thus the tolerance for local errors will be smaller after refinement.

Guaranteed error reduction strategy: Usually, it is not clear whether the adaptive refinement strategy Algorithm 1.5.1 using a marking strategy (other than global refinement) will converge and stop. Dörfler [28] describes a strategy with a guaranteed error reduction for the Poisson equation within a given tolerance.

We need the assumptions, that

- given data of the problem (like the right hand side) is sufficiently resolved by the initial mesh $\mathcal{S}_0$ with respect to the given tolerance (such that, for example, errors from the numerical quadrature are negligible),
- all edges of marked mesh elements are at least bisected by the refinement procedure (using regular refinement or two/three iterated bisections of triangles/tetrahedra, for example).

The idea is to refine a subset of the triangulation whose element errors sum up to a fixed amount of the total error η. Given a parameter $\theta_* \in (0,1)$, the procedure is:

$$\text{Mark a set } \mathcal{A} \subseteq \mathcal{S}_k \text{ such that } \quad \sum_{S \in \mathcal{A}} \eta_S^p \geq (1-\theta_*)^p \eta^p\,.$$

It follows from the assumptions that the error will be reduced by at least a factor $\kappa < 1$ depending of θ_* and data of the problem. Selection of the set $\mathcal{A}$ can be done in the following way. The maximum strategy threshold γ is reduced in small steps of size $\nu \in (0,1)$, $\nu << 1$, until the maximum strategy marks a set which is large enough. This inner iteration is not costly in terms of CPU time as no computations are performed.

1.5.4 Algorithm (Guaranteed error reduction strategy[28])**.**

```
Start with given parameters θ* ∈ (0,1), ν ∈ (0,1)
η_max := max(η_S, S ∈ S_k)
sum := 0
γ := 1
while sum < (1 − θ*)^p η^p do
   γ := γ − ν
   for all S in S_k do
      if S is not marked
         if η_S > γ η_max
            mark S for refinement
            sum := sum + η_S^p
         end if
      end if
   end for
end while
```

Using the above algorithm, Dörfler [27] describes a robust adaptive strategy also for the *nonlinear* Poisson equation $-\Delta u = f(u)$. It is based on a posteriori error estimates and a posteriori saturation criteria for the approximation of the nonlinearity.

1.5.5 Remark. Using this GERS strategy and an additional marking of elements due to data approximation, Morin, Nochetto, and Siebert [41, 42, 43] could remove the assumption that data is sufficiently resolved on $\mathcal{S}_0$ in order to prove convergence. The result is a simple and efficient adaptive finite element method for linear elliptic PDEs with a linear rate of convergence without any preliminary mesh adaptation.

Other refinement strategies: Babuška and Rheinboldt [3] describe an extrapolation strategy, which estimates the local error decay. Using this estimate, refinement of elements is done when the actual local error is larger than the biggest expected local error *after refinement*.

Jarausch [33] describes a strategy which generates quasi–optimal meshes. It is based on an optimization procedure involving the increase of a cost function during refinement and the profit while minimizing an energy functional.

For special applications, additional information may be generated by the error estimator and used by the adaptive strategy. This includes (anisotropic) directional refinement of elements [34, 52], or the decision of local h– or p–enrichment of the finite element space [25, 48].

1.5.3 Coarsening strategies

Up to now we presented only refinement strategies. Practical experience indicates that for linear elliptic problems, no more is needed to generate a quasi–optimal mesh with nearly equidistributed local errors.

In time dependent problems, the regions where large local errors are produced can move in time. In stationary nonlinear problems, a bad resolution of the solution on coarse meshes may lead to some local refinement where it is not needed for the final solution, and the mesh

could be coarsened again. Both situations result in the need to coarsen the mesh at some places in order to keep the number of unknowns small.

Coarsening of the mesh can produce additional errors. Assuming that these are bounded by an a posteriori estimate $\eta_{c,S}$, we can take this into account during the marking procedure.

Some of the refinement strategies described above can also be used to mark mesh elements for coarsening. Actually, elements will only be coarsened if all neighbour elements which are affected by the coarsening process are marked for coarsening, too. This makes sure that only elements where the error is small enough are coarsened, and motivates the coarsening algorithm in Section 1.1.2.

The main concept for coarsening is again the equidistribution of local errors mentioned above. Only elements with a very small local error estimate are marked for coarsening. On the other hand, such a coarsening tolerance should be small enough such that the local error *after coarsening* should not be larger than the tolerance used for refinement. If the error after coarsening gets larger than this value, the elements would be directly refined again in the next iteration (which may lead to a sequence of oscillating grid never meeting the desired criterion).

Usually, an upper bound μ for the mesh size power of the local error estimate is known, which can be used to determine the coarsening tolerance: if

$$\eta_S \;\leq\; c h_S^{\mu},$$

then coarsening by undoing b bisections will enlarge the local error by a factor smaller than $2^{\mu b/\mathtt{DIM}}$, such that the local coarsening tolerance tol_c should be smaller than

$$tol_c \;\leq\; \frac{tol_r}{2^{\mu b/\mathtt{DIM}}},$$

where tol_r is the local refinement tolerance.

Maximum strategy: Given two parameters $\gamma > \gamma_c$, refine all elements S with

$$\eta_S^p > \gamma \max_{S' \in \mathcal{S}_k} \eta_{S'}^p$$

and mark all elements S with

$$\eta_S^p + \eta_{c,S}^p \leq \gamma_c \max_{S' \in \mathcal{S}_k} \eta_{S'}^p$$

for coarsening.

Equidistribution strategy: Equidistribution of the tolerated error tol leads to

$$\eta_S \approx \frac{tol}{N_k^{1/p}} \qquad \text{for all } S \in \mathcal{S}.$$

If the local error at an element is considerably smaller than this mean value, we may coarsen the element without producing an error that is too large. All elements with

$$\eta_S > \theta \, \frac{tol}{N_k^{1/p}}$$

are marked for refinement, while all elements with

$$\eta_S + \eta_{c,S} \leq \theta_c \frac{tol}{N_k^{1/p}}$$

are marked for coarsening.

Guaranteed error reduction strategy: Similar to the refinement in Algorithm 1.5.4, Dörfler [29] describes a marking strategy for coarsening. Again, the idea is to coarsen a subset of the triangulation such that the additional error after coarsening is not larger than a fixed amount of the given tolerance tol. Given a parameter $\theta_c \in (0, 1)$, the procedure is:

$$\text{Mark a set } \mathcal{B} \subseteq \mathcal{S}_k \text{ such that } \quad \sum_{S \in \mathcal{B}} \eta_S^p + \eta_{c,S}^p \leq \theta_c^p \eta^p \,.$$

The selection of the set $\mathcal{B}$ can be done similar to Algorithm 1.5.4.

1.5.6 Remark. When local h– and p–enrichment and coarsening of the finite element space is used, then the threshold θ_c depends on the local degree of finite elements. Thus, local thresholds $\theta_{c,S}$ have to be used.

Handling information loss during coarsening. Usually, some information is irreversibly destroyed during coarsening of parts of the mesh, compare Section 3.3.3. If the adaptive procedure iterates several times, it may occur that elements which were marked for coarsening in the beginning are not allowed to coarsen at the end. If the mesh was already coarsened, an error is produced which can not be reduced anymore.

One possibility to circumvent such problems is to delay the mesh coarsening until the final iteration of the adaptive procedure, allowing only refinements before. If the coarsening marking strategy is not too liberal (θ_c not too large), this should keep the error below the given bound.

Dörfler [29] proposes to keep all information until it is clear, after solving and by estimating the error on a (virtually) coarsened mesh, that the coarsening does not lead to an error which is too large.

1.5.4 Adaptive methods for time dependent problems

In time dependent problems, the mesh is adapted to the solution in every time step using a posteriori error estimators or indicators. This may be accompanied by an adaptive control of time step sizes, see below.

Bänsch [8] lists several different adaptive procedures (in space) for time dependent problems:

- **Explicit strategy:** The current time step is solved once on the mesh from the previous time step, giving the solution u_h. Based on a posteriori estimates of u_h, the mesh is locally refined and coarsened. The problem is *not* solved again on the new mesh, and the solve–estimate–adapt process is *not* iterated.
  This strategy is only usable when the solution is nearly stationary and does not change much in time, or when the time step size is very small. Usually, a given tolerance for the error can not be guaranteed with this strategy.

- **Semi–implicit strategy:** The current time step is solved once on the mesh from the previous time step, giving an intermediate solution $\tilde{u}_h$. Based on a posteriori estimates of $\tilde{u}_h$, the mesh is locally refined and coarsened. This produces the final mesh for the current time step, where the discrete solution u_h is computed. The solve–estimate–adapt process is *not* iterated.
  This strategy works quite well, if the time steps are not too large, such that regions of refinement move too fast.

- **Implicit strategy A:** In every time step starting from the previous time step's triangulation, a mesh is generated using local refinement and coarsening based on a posteriori estimates of a solution which is calculated on the current mesh. This solve–estimate–adapt process is iterated until the estimated error is below the given bound.
  This guarantees that the estimated error is below the given bound. Together with an adaptive control of the time step size, this leads to global (in time) error bounds. If the time step size is not too large, the number of iterations of the solve–estimate–adapt process is usually very small.

- **Implicit strategy B:** In every time step starting from the macro triangulation, a mesh is generated using local refinements based on a posteriori estimates of a solution which is calculated on the current (maybe quite coarse) mesh; no mesh coarsening is needed. This solve–estimate–adapt process is iterated until the estimated error is below the given bound.
  Like implicit strategy A, this guarantees error bounds. As the initial mesh for every time step is very coarse, the number of iterations of the solve–estimate–adapt process becomes quite large, and thus the algorithm might become expensive. On the other hand, a solution on a coarse grid is fast and can be used as a good initial guess for finer grids, which is usually better than using the solution from the old time step.
  Implicit strategy B can also be used with anisotropically refined triangular meshes, see [32]. As coarsening of anisotropic meshes and changes of the anisotropy direction are still open problems, this implies that the implicit strategy A can not be used in this context.

The following algorithm implements one time step of the implicit strategy A. The adaptive algorithm ensures that the mesh refinement/coarsening is done at least once in each time step, even if the error estimate is below the limit. Nevertheless, the error might not be equally distributed over all elements; for some simplices the local error estimates might be bigger than allowed.

1.5.7 Algorithm (Implicit strategy A)**.**

```
Start with given parameters tol and time step size τ,
the solution u_n from the previous time step on grid S_n
S_{n+1} := S_n
solve the discrete problem for u_{n+1} on S_{n+1} using data u_n
compute error estimates on S_{n+1}
do
  mark elements for refinement or coarsening
  if elements are marked then
```

```
      adapt mesh S_{n+1} producing a modified S_{n+1}
      solve the discrete problem for u_{n+1} on S_{n+1} using data u_n
      compute error estimates on S_{n+1}
    end if
  while η > tol
```

1.5.4.1 Adaptive control of the time step size

A posteriori error estimates for parabolic problems usually consist of four different types of terms:

- terms estimating the initial error;
- terms estimating the error from discretization in space;
- terms estimating the error from mesh coarsening between time steps;
- terms estimating the error from discretization in time.

Thus, the total estimate can be split into parts

$$\eta_0,\ \eta_h,\ \eta_c,\ \text{and}\ \eta_\tau$$

estimating these four different error parts.

Example: Eriksson and Johnson [31] prove an a posteriori error estimate for the discontinuous Galerkin time discretization of the heat equation

$$u_t - \Delta u = f \ \text{ in } \Omega, \quad u_{|\partial\Omega} = 0, \quad u_{|t=0} = u_0;$$

the error estimate for piecewise constant time discretization and piecewise linear discretization in space is given by

$$\begin{aligned}\|u(t_N) - U_N\| \ \leq\ & \|u_0 - U_0\| + \max_{1\leq n\leq N}\Bigg(C_1\|h_n^2 f\|_{L^\infty(I_n,L^2(S))} + C_2\int_{I_n}\|f\|\,dt \\ & + C_3\Big(\sum_{e\in E_n} h_e^3\Big|\Big[\frac{\partial U_n}{\partial \nu_e}\Big]\Big|^2\Big)^{1/2} + C_4\|U_n - U_{n-1}\| + C_5\Big\|h_n^2\frac{[U_{n-1}]}{\tau_n}\Big\|\Bigg),\end{aligned}$$

where U_n is the discrete solution on $I_n := (t_{n-1}, t_n)$, $\tau_n = t_n - t_{n-1}$ is the n^{th} time step size, $[\cdot]$ denotes jumps over edges or between time intervals, and $\|\cdot\|$ denotes the norm in $L^2(\Omega)$. The last term $C_5\|\ldots\|$ is present only in case of mesh coarsening. The constants C_i depend on the time t_N and the size of the last time step: $C_i = C_i(\log(\frac{t_N}{\tau_N}))$.

This leads to the following error estimator parts:

$$\begin{aligned}\eta_0 &= \|u_0 - U_0\|,\\ \eta_h &= \sum_{S\in\mathcal{S}_n}\Big(\tilde{C}_1\|h_n^2 f\|_{L^\infty(I_n,L^2(S))} + \tilde{C}_3\Big(\frac{1}{2}\sum_{e\subset\partial S} h_e^3\Big|\Big[\frac{\partial U_n}{\partial\nu_e}\Big]\Big|^2\Big)^{1/2}\Big),\\ \eta_c &= \sum_{S\in\mathcal{S}_n}\Big(\tilde{C}_5\Big\|h_n^2\frac{[U_{n-1}]}{\tau_n}\Big\|_{L^2(S)}\Big),\\ \eta_\tau &= \tau_n\Big(C_2\|f\|_{L^\infty(I_n,L^2(S))} + C_4\Big\|\frac{U_n - U_{n-1}}{\tau_n}\Big\|\Big).\end{aligned}$$

When a bound *tol* is given for the total error produced in each time step, the widely used strategy is to allow one fixed portion $\Gamma_0\, tol$ to be produced by the discretization of initial data, a portion $\Gamma_h\, tol$ to be produced by the spatial discretization, and another portion $\Gamma_\tau\, tol$ of the error to be produced by the time discretization, with $\Gamma_0 + \Gamma_h + \Gamma_\tau \leq 1.0$. The adaptive procedure now tries to adjust time step sizes and meshes such that

$$\eta_o \approx \Gamma_0\, tol$$

and in every time step

$$\eta_\tau \approx \Gamma_\tau\, tol \quad \text{and} \quad \eta_h + \eta_c \approx \Gamma_h\, tol\,.$$

The adjustment of the time step size can be done via extrapolation techniques known from numerical methods for ordinary differential equations, or iteratively: The algorithm starts from the previous time step size τ_{old} or from an initial guess. A parameter $\delta_1 \in (0,1)$ is used to reduce the step size until the estimate is below the given bound. If the error is smaller than the bound, the step size is enlarged by a factor $\delta_2 > 1$ (usually depending on the order of the time discretization). In this case, the actual time step is not recalculated, only the initial step size for the next time step is changed. Two additional parameters $\theta_1 \in (0,1)$, $\theta_2 \in (0,\theta_1)$ are used to keep the algorithm robust, just like it is done in the equidistribution strategy for the mesh adaption. The algorithm starts from the previous time step size τ_{old} or from an initial guess.

If $\delta_1 \approx 1$, consecutive time steps may vary only slightly, but the number of iterations for getting the new accepted time step may increase. Again, as each iteration includes the solution of a discrete problem, this value should be chosen not too large. For a first order time discretization scheme, a common choice is $\delta_1 \approx 1/\sqrt{2}$.

1.5.8 Algorithm (Time step size control)**.**

```
Start with parameters δ1 ∈ (0,1), δ2 > 1, θ1 ∈ (0,1), θ2 ∈ (0,θ1)

τ := τ_old
Solve time step problem and estimate the error
while η_τ > θ1 Γ_τ tol do
   τ := δ1 τ
   Solve time step problem and estimate the error
end while
if η_τ ≤ θ2 Γ_τ tol then
   τ := δ2 τ
end if
```

The above algorithm controls only the time step size, but does not show the mesh adaption. There are several possibilities to combine both controls. An inclusion of the grid adaption in every iteration of Algorithm 1.5.8 can result in a large number of discrete problems to solve, especially if the time step size is reduced more than once. A better procedure is first to do the step size control with the old mesh, then adapt the mesh, and after this check the time error again. In combination with the implicit strategy A, this procedure leads to the following algorithm for one single time step

1.5.9 Algorithm (Time and space adaptive algorithm)**.**

```
Start with given parameter tol, δ1 ∈ (0,1), δ2 > 1, θ1 ∈ (0,1), θ2 ∈ (0,θ1),
```

```
the solution u_n from the previous time step on grid S_n at time t_n
with time step size τ_n

S_{n+1} := S_n
τ_{n+1} := τ_n
t_{n+1} := t_n + τ_{n+1}
solve the discrete problem for u_{n+1} on S_{n+1} using data u_n
compute error estimates on S_{n+1}

while η_τ > θ_1 Γ_τ tol
   τ_{n+1} := δ_1 τ_{n+1}
   t_{n+1} := t_n + τ_{n+1}
   solve the discrete problem for u_{n+1} on S_{n+1} using data u_n
   compute error estimates on S_{n+1}
end while

do
   mark elements for refinement or coarsening
   if elements are marked then
      adapt mesh S_{n+1} producing a modified S_{n+1}
      solve the discrete problem for u_{n+1} on S_{n+1} using data u_n
      compute estimates on S_{n+1}
   end if
   while η_τ > θ_1 Γ_τ tol
      τ_{n+1} := δ_1 τ_{n+1}
      t_{n+1} := t_n + τ_{n+1}
      solve the discrete problem for u_{n+1} on S_{n+1} using data u_n
      compute error estimates on S_{n+1}
   end while
while η_h > tol

if η_τ ≤ θ_2 Γ_τ tol then
   τ_{n+1} := δ_2 τ_{n+1}
end if
```

The adaptive a posteriori approach can be extended to the adaptive choice of the order of the time discretization: Bornemann [17, 18, 19] describes an adaptive variable order time discretization method, combined with implicit strategy B using the extrapolation marking strategy for the mesh adaption.

1.6 Submeshes

Probably the most significant new feature introduced in ALBERTA 2.0 are the automatic generation and maintenance of *submeshes* or *slave meshes*. We motivate the idea by extending the model problem (1.7) to include inhomogeneous Neumann boundary conditions:

$$-\nabla \cdot A\nabla u + b \cdot \nabla u + c\,u = f \qquad \text{in } \Omega, \tag{1.25a}$$

$$u = g_D \qquad \text{on } \Gamma_D, \tag{1.25b}$$

$$\nu_\Omega \cdot A\nabla u = g_N \qquad \text{on } \Gamma_N, \tag{1.25c}$$

with a given function $g_N : \Gamma_N \to \mathbb{R}$. Using the notation of Section 1.4.5 our weak formulation now consists of finding a solution $u \in X$, such that $u \in g_D + \mathring{X}$ and

$$\int_\Omega (\nabla\varphi(x)) \cdot A(x)\nabla u(x) + \varphi(x)\, b(x) \cdot \nabla u(x) + c(x)\, \varphi(x)\, u(x)\, dx = \int_\Omega f(x)\, \varphi(x)\, dx + \int_{\Gamma_N} g_N(x)\, \gamma\varphi(x)\, do(x) \quad (1.26)$$

for all $\varphi \in \mathring{X}$. Here we have used γ to denote the weak trace operator from $H^1(\Omega)$ to $H^{\frac{1}{2}}(\Gamma_N)$ REFERENZ EINFUEGEN???. The inhomogeneous Neumann boundary condition therefore leads to an additional term on the right hand side.

Consider again a finite dimensional subspace $X_h \subset X$ with $N = \dim X_h$ based on a triangulation $\mathcal{S}$ of Ω. Assume that $\mathcal{S}$ is constructed in such a way that

$$\overline{\Gamma_N} = \bigcup_{T \in \mathcal{T}} \quad \text{with} \quad \mathcal{T} = \{S \cap \Gamma_N; S \in \mathcal{S}\} \quad (1.27)$$

holds. In other words, the triangulation $\mathcal{S}$ of Ω induces a triangulation $\mathcal{T}$ of Γ_N. We define

$$Y_h = \operatorname{span}\{\gamma\varphi_h; \varphi_h \in X_h\}.$$

Let X_h be chosen as the space of standard Lagrange elements of order p on the triangulation $\mathcal{S}$ of Ω. Thanks to (1.27) we then have that Y_h is exactly the corresponding space of Lagrange elements of order p on the triangulation $\mathcal{T}$ of Γ_N. The position of the DOFs of Y_h in space coincides with corresponding DOFs of X_h.

To be precise, let us again define bases $\{\varphi_1, \ldots, \varphi_N\}$ of X_h and $\{\psi_1, \ldots, \psi_M\}$ of Y_h. Then there exists a unique injective mapping of indices J with the following properties:

$$J : \{1, \ldots, M\} \to \{1, \ldots, N\},$$
$$\psi_i = \gamma\varphi_{J(i)} \qquad \text{for all } i = 1, \ldots, M.$$

The mesh $\mathcal{T}$ satisfying the conformity property (1.27) is known as a submesh of $\mathcal{S}$.

Making use of these properties, we can implement the assemblage of the additional right hand side as follows. Assume that a load vector $\boldsymbol{f}$ as defined in (1.12) is already assembled. Define a temporary quantity $\boldsymbol{g}$ associated with Y_h as

$$\boldsymbol{g} := \begin{bmatrix} \langle g_N, \psi_1 \rangle_{Y_h^* \times Y_h} \\ \vdots \\ \langle g_N, \psi_M \rangle_{Y_h^* \times Y_h} \end{bmatrix}, \quad (1.28)$$

with

$$\langle g_N, \psi \rangle_{Y_h^* \times Y_h} = \int_{\Gamma_N} g_N(x)\, \psi(x)\, do(x).$$

Let $j \in \{1, \ldots, N\}$. We have

$$\int_{\Gamma_N} g_N(x)\, \gamma\varphi_j(x)\, do(x) = \int_{\Gamma_N} g_N(x)\, \gamma\varphi_{J(i)}(x)\, do(x) = \int_{\Gamma_N} g_N(x)\, \psi_i(x)\, do(x)$$

if $j = J(i)$ for some $i \in \{1, \dots, N\}$ and

$$\int_{\Gamma_N} g_N(x)\, \gamma\varphi_j(x)\, do(x) = 0$$

otherwise. We may therefore complete the assemblage of $\boldsymbol{f}$ using the loop

$$\boldsymbol{f}_{J(i)} = \boldsymbol{f}_{J(i)} + \boldsymbol{g}_i \qquad \text{for } i = 1, \dots, M. \tag{1.29}$$

It follows that we may make use of standard ALBERTA routines for the assemblage of right hand sides, here applied to the finite element space Y_h on the submesh $\mathcal{T}$. This enables us to create simple and clear code. Furthermore, we assembled the right hand side contribution due to the Neumann boundary condition by traversing the smaller submesh instead of the bulk mesh, which may reduce computational effort. On the downside, we now need to store the submesh $\mathcal{T}$ as well as the mapping of indices J in memory.

The current version ALBERTA 2.0 provides an automatic mechanism to generate submeshes. The user must only provide a function which will implement the decision about which subsimplices of which macro elements will belong to the newly created submesh. ALBERTA takes care of providing an index mapping J, given that fitting Lagrange elements are used on bulk mesh and submesh.

Most important however is the fact that ALBERTA will automatically refine and coarsen bulk mesh and submeshes *simultaneously*, which means that the property (1.27) is preserved throughout an adaptive simulation. The user may even define submeshes of submeshes, allowing whole hierarchies of dependent meshes, see Figure 1.19.

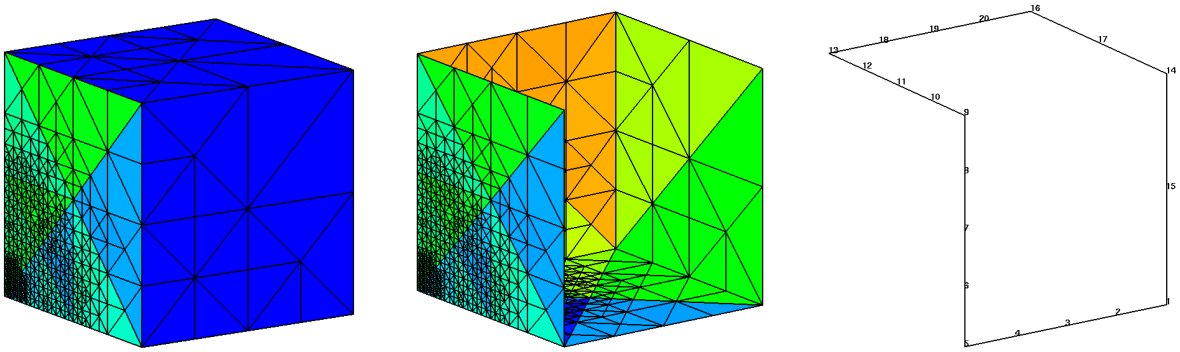


Figure 1.19: Entire hierarchies of submeshes may be defined.

Bibliography

[1] M. Ainsworth and J. T. Oden, *A unified approach to a posteriori error estimation using element residual methods*, Numer. Math., 65 (1993), pp. 23–50.

[2] ——, *A posteriori error estimation in finite element analysis*, Wiley, 2000.

[3] I. Babuška and W. Rheinboldt, *Error estimates for adaptive finite element computations*, SIAM J. Numer. Anal., 15 (1978), pp. 736–754.

[4] A. Bamberger, E. Bänsch, and K. G. Siebert, *Experimental and numerical investigation of edge tones.* Preprint WIAS Berlin no. 681, 2001.

[5] R. E. Bank, *PLTMG: a software package for solving elliptic partial differential equations user's guide 8.0.* Software - Environments - Tools. 5. Philadelphia, PA: SIAM. xii, 1998.

[6] R. E. Bank and A. Weiser, *Some a posteriori error estimators for elliptic partial differential equations.*, Math. Comput., 44 (1985), pp. 283–301.

[7] E. Bänsch, *Local mesh refinement in 2 and 3 dimensions*, IMPACT Comput. Sci. Engrg., 3 (1991), pp. 181–191.

[8] ——, *Adaptive finite element techniques for the Navier–Stokes equations and other transient problems*, in Adaptive Finite and Boundary Elements, C. A. Brebbia and M. H. Aliabadi, eds., Computational Mechanics Publications and Elsevier, 1993, pp. 47–76.

[9] E. Bänsch and K. G. Siebert, *A posteriori error estimation for nonlinear problems by duality techniques.* Preprint 30, Universität Freiburg, 1995.

[10] P. Bastian, K. Birken, K. Johannsen, S. Lang, V. Reichenberger, C. Wieners, G. Wittum, and C. Wrobel, *Parallel solution of partial differential equations with adaptive multigrid methods on unstructured grids*, in High performance computing in science and engineering '99, E. Krause and et al., eds., Berlin: Springer, 2000, pp. 496–508. Transactions of the High Performance Computing Center Stuttgart (HLRS). 2nd workshop, Stuttgart, Germany, October 4-6, 1999.

[11] R. Beck, B. Erdmann, and R. Roitzsch, *An object-oriented adaptive finite element code: Design issues and applications in hyperthermia treatment planning*, in Modern software tools for scientific computing, E. Arge and et al., eds., Boston: Birkhaeuser, 1997, pp. 105–124. International workshop, Oslo, Norway, September 16–18, 1996.

[12] R. Becker and R. Rannacher, *A feed-back approach to error control in finite element methods: Basic analysis and examples.*, East-West J. Numer. Math., 4 (1996), pp. 237–264.

[13] J. Bey, *Tetrahedral grid refinement*, Computing, 55 (1995), pp. 355–378.

[14] ——, *Simplicial grid refinement: On Freudenthal's algorithm and the optimal number of congruence classes*, Numer. Math., 85 (2000), pp. 1–29.

[15] P. Binev, W. Dahmen, and R. DeVore, *Adaptive finite element methods with convergence rates*. IGPM Report No. 219, RWTH Aachen, 2002.

[16] M. Böhm, A. Schmidt, and M. Wolff, *Adaptive finite element simulation of a model for transformation induced plasticity in steel*. Report ZeTeM Bremen, 2003.

[17] F. A. Borneman, *An adaptive multilevel approach to parabolic equations I*, IMPACT Comput. Sci. Engrg., 2 (1990), pp. 279–317.

[18] ——, *An adaptive multilevel approach to parabolic equations II*, IMPACT Comput. Sci. Engrg., 3 (1990), pp. 93–122.

[19] ——, *An adaptive multilevel approach to parabolic equations III*, IMPACT Comput. Sci. Engrg., 4 (1992), pp. 1–45.

[20] F. A. Bornemann, B. Erdmann, and R. Kornhuber, *A posteriori error estimates for elliptic problems in two and three space dimensions.*, SIAM J. Numer. Anal., 33 (1996), pp. 1188–1204.

[21] S. Boschert, A. Schmidt, K. G. Siebert, E. Bänsch, K. W. Benz, G. Dziuk, and T. Kaiser, *Simulation of industrial crystal growth by the vertical Bridgman method*, in Mathematics - Key Technology for the Future Joint Projects Between Universities and Industry, W. Jäger and H.-J. Krebs, eds., Springer, 2003, pp. 315–330.

[22] S. C. Brenner and L. Scott, *The mathematical theory of finite element methods. 2nd ed.*, Springer, 2002.

[23] Z. Chen and R. H. Nochetto, *Residual type a posteriori error estimates for elliptic obstacle problems.*, Numer. Math., 84 (2000), pp. 527–548.

[24] P. G. Ciarlet, *The finite element methods for elliptic problems. Repr., unabridged republ. of the orig. 1978.*, SIAM, 2002.

[25] L. Demkowicz, J. T. Oden, W. Rachowicz, and O. Hardy, *Toward a universal h–p adaptive finite element strategy, Part 1 – Part 3*, Comp. Methods Appl. Mech. Engrg., 77 (1989), pp. 79–212.

[26] W. Dörfler, *FORTRAN–Bibliothek der Orthogonalen Fehler–Methoden*, Manual, Mathematische Fakultät Freiburg, 1995.

[27] ——, *A robust adaptive strategy for the nonlinear poisson equation*, Computing, 55 (1995), pp. 289–304.

[28] ——, *A convergent adaptive algorithm for Poisson's equation*, SIAM J. Numer. Anal., 33 (1996), pp. 1106–1124.

[29] ——, *A time- and spaceadaptive algorithm for the linear time-dependent Schrödinger equation*, Numer. Math., 73 (1996), pp. 419–448.

[30] W. Dörfler and K. G. Siebert, *An adaptive finite element method for minimal surfaces*, in Geometric Analysis and Nonlinear Partial Differential Equations, S. Hildebrandt and H. Karcher, eds., Springer, 2003, pp. 146–175.

[31] K. Eriksson and C. Johnson, *Adaptive finite element methods for parabolic problems I: A linear model problem*, SIAM J. Numer. Anal., 28 (1991), pp. 43–77.

[32] J. Fröhlich, J. Lang, and R. Roitzsch, *Selfadaptive finite element computations with smooth time controller and anisotropic refinement*, in Numerical Methods in Engineering, J. Desideri, P. Tallec, E. Onate, J. Periaux, and E. Stein, eds., John Wiley & Sons, New York, 1996, pp. 523–527.

[33] H. Jarausch, *On an adaptive grid refining technique for finite element approximations*, SIAM J. Sci. Stat. Comput., 7 (1986), pp. 1105–1120.

[34] R. Kornhuber and R. Roitzsch, *On adaptive grid refinement in the presence of internal or boundary layers*, IMPACT Comput. Sci. Engrg., 2 (1990), pp. 40–72.

[35] I. Kossaczký, *A recursive approach to local mesh refinement in two and three dimensions*, J. Comput. Appl. Math., 55 (1994), pp. 275–288.

[36] K. Lin, S. Boschert, P. Dold, K. W. Benz, O. Kriessl, A. Schmidt, K. G. Siebert, and G. Dziuk, *Numerical methods for industrial vertical Bridgman growth of (Cd,Zn)Te*, J. Crystal Growth, 237-239 (2002), pp. 1736–1740.

[37] J. M. Maubach, *Local bisection refinement for n-simplicial grids generated by reflection*, SIAM J. Sci. Comput., 16 (1995), pp. 210–227.

[38] W. F. Mitchell, *Unified Multilevel Adaptive Finite Element Methods for Elliptic Problems*, PhD thesis, Department of Computer Science, University of Illinois, Urbana, 1988.

[39] ——, *A comparison of adaptive refinement techniques for elliptic problems*, ACM Trans. Math. Softw., 15 (1989), pp. 326–347.

[40] ——, *MGGHAT: Elliptic PDE software with adaptive refinement, multigrid and high order finite elements*, in Sixth Copper Mountain Conference on Multigrid Methods, N. D. Melson, T. A. Manteuffel, and S. F. McCormick, eds., NASA, 1993, pp. 439–448.

[41] P. Morin, R. H. Nochetto, and K. G. Siebert, *Data oscillation and convergence of adaptive FEM*, SIAM J. Numer. Anal., 38 (2000), pp. 466–488.

[42] ——, *Convergence of adaptive finite element methods*, SIAM Review, 44 (2002), pp. 631–658.

[43] ——, *Local problems on stars: A posteriori error estimators, convergence, and performance*, Math. Comp., 72 (2003), pp. 1067–1097.

[44] R. H. Nochetto, A. Schmidt, and C. Verdi, *A posteriori error estimation and adaptivity for degenerate parabolic problems*, Math. Comput., 69 (2000), pp. 1–24.

[45] R. H. Nochetto, K. G. Siebert, and A. Veeser, *Pointwise a posteriori error control for elliptic obstacle problems*, Numer. Math., 95 (2003), pp. 163–195.

[46] A. Schmidt and K. G. Siebert, *Concepts of the finite element toolbox ALBERT*. Preprint 17/98 Freiburg, 1998. To appear in Notes on Numerical Fluid Mechanics.

[47] ——, *Abstract data structures for a finite element package: Design principles of ALBERT.*, Z. Angew. Math. Mech., 79 (1999), pp. S49–S52.

[48] ——, *A posteriori estimators for the h–p version of the finite element method in 1d*, Applied Numerical Mathematics, 35 (2000), pp. 43–46.

[49] ——, *ALBERT – Software for scientific computations and applications*, Acta Math. Univ. Comenianae, 70 (2001), pp. 105–122.

[50] J. Schoeberl, *NETGEN: An advancing front 2D/3D-mesh generator based on abstract rules.*, Comput. Vis. Sci., 1 (1997), pp. 41–52.

[51] J. R. Shewchuk, *Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator*, in Applied Computational Geometry: Towards Geometric Engineering, M. C. Lin and D. Manocha, eds., vol. 1148 of Lecture Notes in Computer Science, Springer-Verlag, May 1996, pp. 203–222. From the First ACM Workshop on Applied Computational Geometry.

[52] K. G. Siebert, *A posteriori error estimator for anisotropic refinement*, Numer. Math., 73 (1996), pp. 373–398.

[53] R. Stevenson, *An optimal adaptive finite element method.* Preprint No. 1271, Department of Mathematics, University of Utrecht, 2003.

[54] A. Veeser, *Efficient and reliable a posteriori error estimators for elliptic obstacle problems.*, SIAM J. Numer. Anal., 39 (2001), pp. 146–167.

[55] ——, *Convergent adaptive finite elements for the nonlinear Laplacian*, Numer. Math., 92 (2002), pp. 743–770.

[56] R. Verfürth, *A posteriori error estimation and adaptive mesh–refinement techniques*, J. Comp. Appl. Math., 50 (1994), pp. 67–83.

[57] ——, *A Review of A Posteriori Error Estimation and Adaptive Mesh-Refinement Techniques*, Wiley-Teubner, 1996.

[58] O. C. Zienkiewicz, D. W. Kelly, J. Gago, and I. Babuška, *Hierarchical finite element approaches, error estimates and adaptive refinement*, in The mathematics of finite elements and applications IV, J. Whiteman, ed., Academic Press, 1982, pp. 313–346.