

Lecture Notes
in Computational Science
and Engineering

42

Editors

Timothy J. Barth, Moffett Field, CA

Michael Griebel, Bonn

David E. Keyes, New York

Risto M. Nieminen, Espoo

Dirk Roose, Leuven

Tamar Schlick, New York

Alfred Schmidt
Kunibert G. Siebert

Design of Adaptive Finite Element Software

The Finite Element Toolbox ALBERTA

With 30 Figures

 Springer

Alfred Schmidt
Zentrum für Technomathematik
Fachbereich Mathematik/Informatik
Universität Bremen
Bibliothekstr. 2
28359 Bremen, Germany
e-mail: schmidt@math.uni-bremen.de

Kunibert G. Siebert
Institut für Mathematik
Universität Augsburg
Universitätsstraße 14
86159 Augsburg, Germany
e-mail: siebert@math.uni-augsburg.de

Library of Congress Control Number: 2004113298

Mathematics Subject Classification (2000):
65M50, 65M60, 65N22, 65N30, 65N50, 65Y99, 68U20

ISSN 1439-7358
ISBN 3-540-22842-X Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable for prosecution under the German Copyright Law.

The publisher and the authors accept no legal responsibility for any damage caused by improper use of the instructions and programs contained in this book and the CD-ROM. Although the software has been tested with extreme care, errors in the software cannot be excluded.

Springer is a part of Springer Science+Business Media
springeronline.com

© Springer-Verlag Berlin Heidelberg 2005
Printed in Germany

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Cover design: Friedhelm Steinen-Broo, Estudio Calamar, Spain
Cover production: *design & production*, Heidelberg
Typeset by the authors using a Springer \TeX macro package
Production: LE- \TeX Jelonek, Schmidt & Vöckler GbR, Leipzig

Printed on acid-free paper 46/3142/YL - 5 4 3 2 1 0

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 time frame 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 ALBERTA¹ 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.

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

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 required 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.

These notes are 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. Visit the ALBERTA web site

<http://www.alberta-fem.de/>

for updates, more information, FAQ, contributions, pictures from different projects, etc.

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.

The cover picture of this book shows the ALBERTA logo, combined with a locally refined cogwheel mesh [17], and the norm of the velocity from a calculation of edge tones in a flute [4].

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 gtools, 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, October 2004

Alfred Schmidt and Kunibert G. Siebert

Contents

| | |
|---|----|
| Introduction | 1 |
| 1 Concepts and abstract algorithms | 9 |
| 1.1 Mesh refinement and coarsening | 9 |
| 1.1.1 Refinement algorithms for simplicial meshes | 12 |
| 1.1.2 Coarsening algorithm for simplicial meshes | 18 |
| 1.1.3 Operations during refinement and coarsening | 20 |
| 1.2 The hierarchical mesh | 22 |
| 1.3 Degrees of freedom | 24 |
| 1.4 Finite element spaces and finite element discretization..... | 25 |
| 1.4.1 Barycentric coordinates | 26 |
| 1.4.2 Finite element spaces | 29 |
| 1.4.3 Evaluation of finite element functions | 29 |
| 1.4.4 Interpolation and restriction during refinement and coarsening | 32 |
| 1.4.5 Discretization of 2nd order problems | 35 |
| 1.4.6 Numerical quadrature | 38 |
| 1.4.7 Finite element discretization of 2nd order problems ... | 39 |
| 1.5 Adaptive Methods | 42 |
| 1.5.1 Adaptive method for stationary problems..... | 42 |
| 1.5.2 Mesh refinement strategies..... | 43 |
| 1.5.3 Coarsening strategies | 47 |
| 1.5.4 Adaptive methods for time dependent problems | 49 |
| 2 Implementation of model problems | 55 |
| 2.1 Poisson equation | 56 |
| 2.1.1 Include file and global variables | 56 |
| 2.1.2 The main program for the Poisson equation..... | 57 |
| 2.1.3 The parameter file for the Poisson equation | 59 |
| 2.1.4 Initialization of the finite element space | 60 |
| 2.1.5 Functions for leaf data | 60 |

| | | |
|----------|---|------------|
| 2.1.6 | Data of the differential equation | 62 |
| 2.1.7 | The assemblage of the discrete system | 63 |
| 2.1.8 | The solution of the discrete system | 65 |
| 2.1.9 | Error estimation | 66 |
| 2.2 | Nonlinear reaction–diffusion equation | 68 |
| 2.2.1 | Program organization and header file | 69 |
| 2.2.2 | Global variables | 71 |
| 2.2.3 | The main program for the nonlinear reaction–diffusion equation | 71 |
| 2.2.4 | Initialization of the finite element space and leaf data | 72 |
| 2.2.5 | The build routine | 72 |
| 2.2.6 | The solve routine | 73 |
| 2.2.7 | The estimator for the nonlinear problem | 73 |
| 2.2.8 | Initialization of problem dependent data | 75 |
| 2.2.9 | The parameter file for the nonlinear reaction–diffusion equation | 79 |
| 2.2.10 | Implementation of the nonlinear solver | 80 |
| 2.3 | Heat equation | 93 |
| 2.3.1 | Global variables | 93 |
| 2.3.2 | The main program for the heat equation | 94 |
| 2.3.3 | The parameter file for the heat equation | 96 |
| 2.3.4 | Functions for leaf data | 98 |
| 2.3.5 | Data of the differential equation | 99 |
| 2.3.6 | Time discretization | 100 |
| 2.3.7 | Initial data interpolation | 100 |
| 2.3.8 | The assemblage of the discrete system | 101 |
| 2.3.9 | Error estimation | 105 |
| 2.3.10 | Time steps | 107 |
| 2.4 | Installation of ALBERTA and file organization | 111 |
| 2.4.1 | Installation | 111 |
| 2.4.2 | File organization | 111 |
| 3 | Data structures and implementation | 113 |
| 3.1 | Basic types, utilities, and parameter handling | 113 |
| 3.1.1 | Basic types | 113 |
| 3.1.2 | Message macros | 114 |
| 3.1.3 | Memory allocation and deallocation | 118 |
| 3.1.4 | Parameters and parameter files | 122 |
| 3.1.5 | Parameters used by the utilities | 127 |
| 3.1.6 | Generating filenames for meshes and finite element data | 127 |
| 3.2 | Data structures for the hierarchical mesh | 128 |
| 3.2.1 | Constants describing the dimension of the mesh | 128 |
| 3.2.2 | Constants describing the elements of the mesh | 129 |
| 3.2.3 | Neighbour information | 129 |

| | | |
|--------|---|-----|
| 3.2.4 | Element indices | 130 |
| 3.2.5 | The BOUNDARY data structure | 130 |
| 3.2.6 | The local indexing on elements | 132 |
| 3.2.7 | The MACRO_EL data structure | 132 |
| 3.2.8 | The EL data structure | 134 |
| 3.2.9 | The EL_INFO data structure | 135 |
| 3.2.10 | The NEIGH, OPP_VERTEX and EL_TYPE macros | 137 |
| 3.2.11 | The INDEX macro | 137 |
| 3.2.12 | The LEAF_DATA_INFO data structure | 138 |
| 3.2.13 | The RC_LIST_EL data structure | 140 |
| 3.2.14 | The MESH data structure | 141 |
| 3.2.15 | Initialization of meshes | 143 |
| 3.2.16 | Reading macro triangulations | 144 |
| 3.2.17 | Writing macro triangulations | 150 |
| 3.2.18 | Import and export of macro triangulations from/to other formats | 151 |
| 3.2.19 | Mesh traversal routines | 153 |
| 3.3 | Administration of degrees of freedom | 161 |
| 3.3.1 | The DOF_ADMIN data structure | 162 |
| 3.3.2 | Vectors indexed by DOFs: The DOF*_VEC data structures | 164 |
| 3.3.3 | Interpolation and restriction of DOF vectors during mesh refinement and coarsening | 167 |
| 3.3.4 | The DOF_MATRIX data structure | 168 |
| 3.3.5 | Access to global DOFs: Macros for iterations using DOF indices | 170 |
| 3.3.6 | Access to local DOFs on elements | 171 |
| 3.3.7 | BLAS routines for DOF vectors and matrices | 173 |
| 3.3.8 | Reading and writing of meshes and vectors | 173 |
| 3.4 | The refinement and coarsening implementation | 176 |
| 3.4.1 | The refinement routines | 176 |
| 3.4.2 | The coarsening routines | 182 |
| 3.5 | Implementation of basis functions | 183 |
| 3.5.1 | Data structures for basis functions | 184 |
| 3.5.2 | Lagrange finite elements | 190 |
| 3.5.3 | Piecewise constant finite elements | 191 |
| 3.5.4 | Piecewise linear finite elements | 191 |
| 3.5.5 | Piecewise quadratic finite elements | 195 |
| 3.5.6 | Piecewise cubic finite elements | 200 |
| 3.5.7 | Piecewise quartic finite elements | 204 |
| 3.5.8 | Access to Lagrange elements | 206 |
| 3.6 | Implementation of finite element spaces | 206 |
| 3.6.1 | The finite element space data structure | 206 |
| 3.6.2 | Access to finite element spaces | 207 |
| 3.7 | Routines for barycentric coordinates | 208 |

| | | |
|--------|--|-----|
| 3.8 | Data structures for numerical quadrature | 210 |
| 3.8.1 | The <code>QUAD</code> data structure | 210 |
| 3.8.2 | The <code>QUAD_FAST</code> data structure | 212 |
| 3.8.3 | Integration over sub-simplices (edges/faces) | 215 |
| 3.9 | Functions for the evaluation of finite elements | 216 |
| 3.10 | Calculation of norms for finite element functions | 221 |
| 3.11 | Calculation of errors of finite element approximations | 222 |
| 3.12 | Tools for the assemblage of linear systems | 224 |
| 3.12.1 | Assembling matrices and right hand sides | 224 |
| 3.12.2 | Data structures and function for matrix assemblage | 227 |
| 3.12.3 | Data structures for storing pre-computed integrals of basis functions | 236 |
| 3.12.4 | Data structures and functions for vector update | 243 |
| 3.12.5 | Dirichlet boundary conditions | 247 |
| 3.12.6 | Interpolation into finite element spaces | 248 |
| 3.13 | Data structures and procedures for adaptive methods | 249 |
| 3.13.1 | ALBERTA adaptive method for stationary problems | 249 |
| 3.13.2 | Standard ALBERTA marking routine | 255 |
| 3.13.3 | ALBERTA adaptive method for time dependent problems | 256 |
| 3.13.4 | Initialization of data structures for adaptive methods | 260 |
| 3.14 | Implementation of error estimators | 263 |
| 3.14.1 | Error estimator for elliptic problems | 263 |
| 3.14.2 | Error estimator for parabolic problems | 266 |
| 3.15 | Solver for linear and nonlinear systems | 268 |
| 3.15.1 | General linear solvers | 268 |
| 3.15.2 | Linear solvers for DOF matrices and vectors | 272 |
| 3.15.3 | Access of functions for matrix-vector multiplication | 274 |
| 3.15.4 | Access of functions for preconditioning | 275 |
| 3.15.5 | Multigrid solvers | 277 |
| 3.15.6 | Nonlinear solvers | 282 |
| 3.16 | Graphics output | 285 |
| 3.16.1 | One and two dimensional graphics subroutines | 286 |
| 3.16.2 | gltools interface | 290 |
| 3.16.3 | GRAPE interface | 293 |
| | References | 295 |
| | Index | 301 |
| | Data types, symbolic constants, functions, and macros | 311 |
| | Data types | 311 |
| | Symbolic constants | 311 |
| | Functions | 312 |
| | Macros | 315 |

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 adaptation, 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 [25], and Brenner and Scott [23] 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 [73], and Ainsworth and Oden [2]. Material about more practical issues like adaptive techniques and refinement procedures can for example be found in [3, 5, 8, 44, 46].

Another basic ingredient for an adaptive finite element method is the a posteriori error estimator which is 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 [10, 73], highly nonlinear problems usually still need a special treatment, see [24, 33, 54, 55, 69] 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, 7, 13, 21, 36, 52, 72].

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 [50, 51, 52] and for the nonlinear Laplacian [70], based on the fundamental paper [31]. For a modification of the convergent algorithm in [50], quasi-optimality of the adaptive method was proved in [16] and [67].

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, 11, 12, 49, 62], and an continuously updated list of other available finite element codes and resources can for instance be found at

http://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 to use 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 [59, 60, 62] provides all those tools mentioned above for the efficient implementation and adaptive solution of general nonlinear problems in two and 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 [25]) 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 Kossaczky [44].

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 $\lambda^S : \bar{S} \rightarrow S$ 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 ([29], 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 implements 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 $f: \Omega \rightarrow \mathbb{R}$ 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

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

For a vector valued, differentiable function $f = (f_1, \dots, f_n): \Omega \rightarrow \mathbb{R}^n$ 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

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

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

$$\|f\|_{L^p(\Omega)} = \left(\int_{\Omega} |f(x)|^p dx \right)^{1/p} \quad \text{for } p < \infty$$

and

$$\|f\|_{L^\infty(\Omega)} = \operatorname{ess\,sup}_{x \in \Omega} |f(x)|.$$

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

$$|u|_{H^1(\Omega)} = \left(\int_{\Omega} |\nabla u(x)|^2 dx \right)^{1/2}$$

and norm

$$\|u\|_{H^1(\Omega)} = \left(\|u\|_{L^2(\Omega)}^2 + |u|_{H^1(\Omega)}^2 \right)^{1/2}.$$

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 Fig. 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 Fig. 1.2).
3. Polynomials of any degree are easily represented on a simplex using local (barycentric) coordinates.

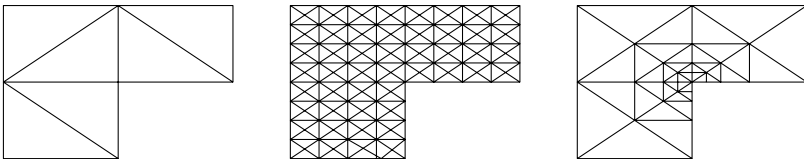


Fig. 1.1. Global and local refinement of a triangular mesh.

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

Definition 1.1 (Simplex).

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

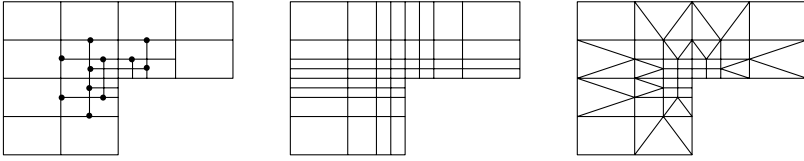


Fig. 1.2. Local refinements of a rectangular mesh: with hanging nodes, conforming closure using bisected rectangles, and conforming closure using triangles. Using a conforming closure with rectangles, a local refinement has always global effects up to the boundary.

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

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

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

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 $F_S: \hat{S} \rightarrow S \subset \mathbb{R}^n$ 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

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

the diameter and inball-diameter of S .

Remark 1.2. 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 $F_S: \hat{S} \rightarrow S$ 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 $F_S: \hat{S} \rightarrow S$ is invertible and that $F_S(\hat{a}_i) = a_i, i = 0, \dots, d$ holds.

Definition 1.3 (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 $0 \leq k < d$.

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

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

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

Remark 1.4. 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 ([63, 65]).

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 $\mathcal{S}_0, \mathcal{S}_1, \dots$. 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 Fig. 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 [14, 15]. 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.

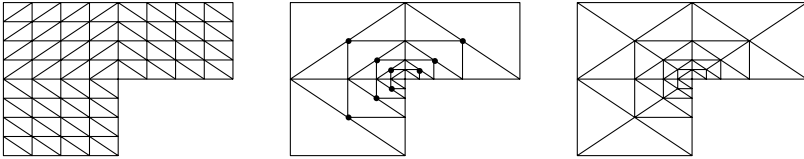


Fig. 1.3. Global and local regular refinement of triangles and conforming closure by bisection.

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 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 [48] 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 Kossaczky described in [44]. 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 children 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 Fig. 1.4 for 1d and 2d, and in Fig. 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 does depend 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 $((\text{parent's type} + 1) \bmod 3)$. In Fig. 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. Fig. 1.6 shows successive refinements of a type 0 tetrahedron, producing tetrahedra of types 1, 2, and 0 again.

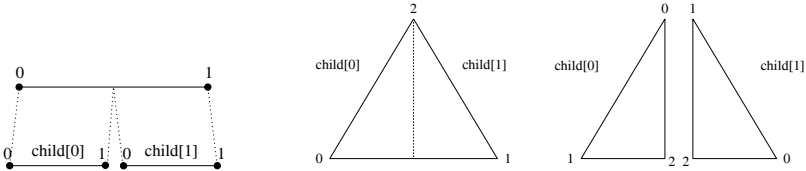


Fig. 1.4. Numbering of nodes on parent and children for intervals and triangles.

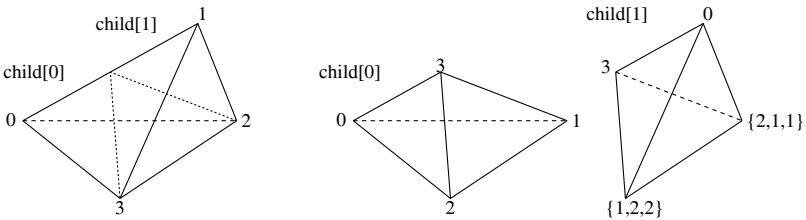


Fig. 1.5. Numbering of nodes on parent and children for tetrahedra.

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

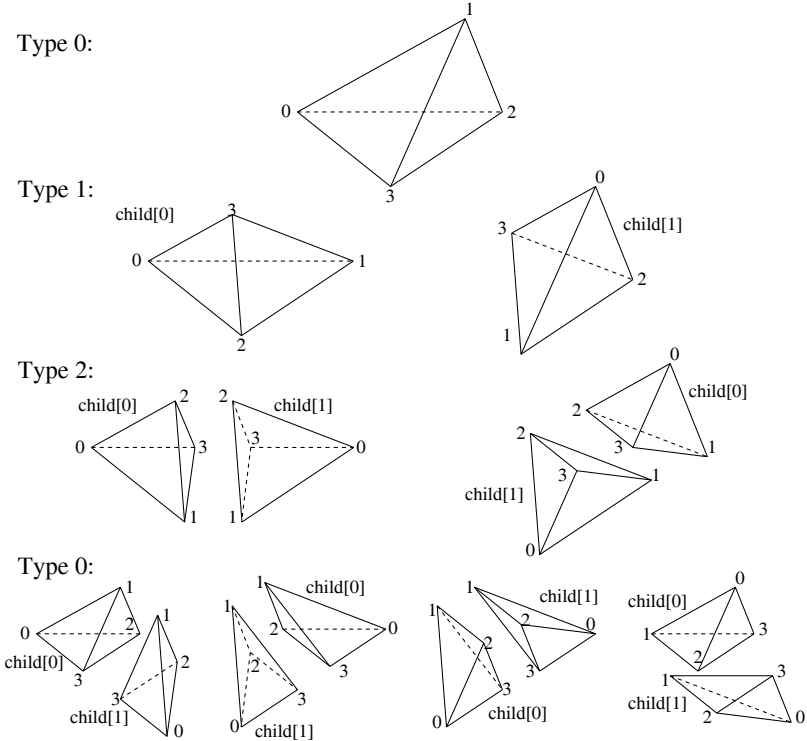


Fig. 1.6. Successive refinements of a type 0 tetrahedron.

classes. In case of the “generic” triangulation of a (unit) square in 2d and cube in 3d into two triangles resp. six tetrahedra (see Fig. 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 always the longest edge will be the refinement edge and will be bisected, see Fig. 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 [8]. 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 Kossaczky [44] and Maubach [46]. 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 Figs. 1.9 and 1.10 for the two and three-dimensional situations.

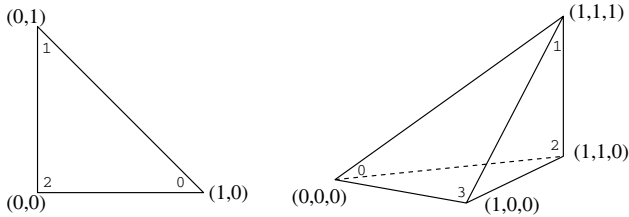


Fig. 1.7. Generic elements in two and three dimensions.

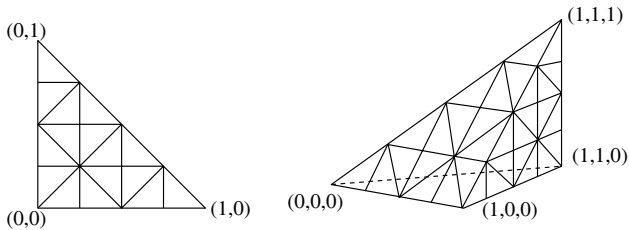


Fig. 1.8. Refined generic elements in two and three dimensions.

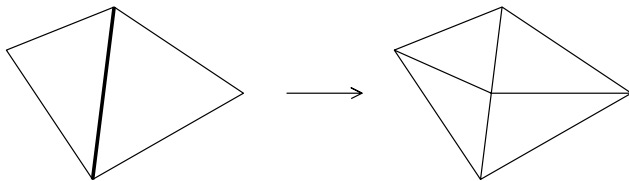


Fig. 1.9. Atomic refinement operation in two dimensions. The common edge is the refinement edge for both triangles.

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

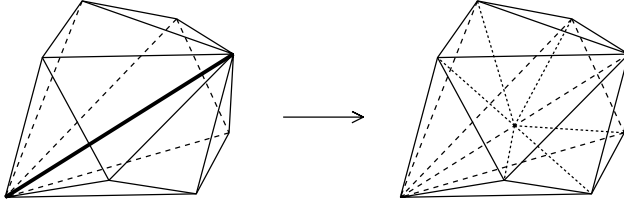


Fig. 1.10. Atomic refinement operation in three dimensions. The common edge is the refinement edge for all tetrahedra sharing this edge.

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.

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

Algorithm 1.5 (Recursive refinement of one simplex).

```

subroutine recursive_refine( $S$ ,  $S$ )
  do
     $\mathcal{A} := \{S' \in S; S' \text{ is not compatibly divisible with } S\}$ 
    for all  $S' \in \mathcal{A}$  do
      recursive_refine( $S'$ ,  $S$ );
    end for
     $\mathcal{A} := \{S' \in S; S' \text{ is not compatibly divisible with } S\}$ 
  until  $\mathcal{A} = \emptyset$ 

   $\mathcal{A} := \{S' \in S; S' \text{ is element at the refinement edge of } S\}$ 
  for all  $S' \in \mathcal{A}$ 
    bisect  $S'$  into  $S'_0$  and  $S'_1$ 
     $S := S \setminus \{S'\} \cup \{S'_0, S'_1\}$ 
  end for

```

In Fig. 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.

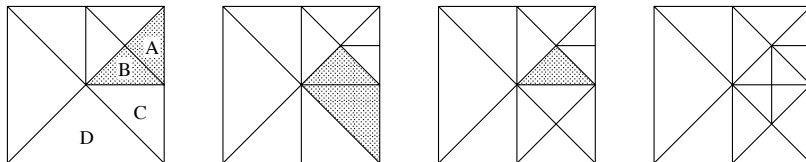


Fig. 1.11. Recursive refinement in two dimensions. Triangles A and B are initially marked for refinement.

The refinement of a given triangulation S where some or all elements are marked for refinement is then performed by

Algorithm 1.6 (Recursive refinement algorithm).

```

subroutine refine( $S$ )
  for all  $S \in \mathcal{S}$  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. Kossaczky [44] and Mitchell [48] proved

Theorem 1.7 (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.*

Remark 1.8.

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 Fig. 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.

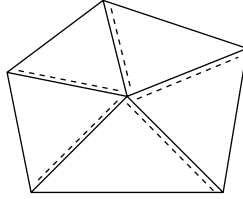


Fig. 1.12. A macro triangulation where recursion does not stop.

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 [44].

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 must 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 Figs. 1.13 and 1.14 for the atomic coarsening operation in 2d and 3d). This coarsening operation is completely local in 1d.

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 must 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).

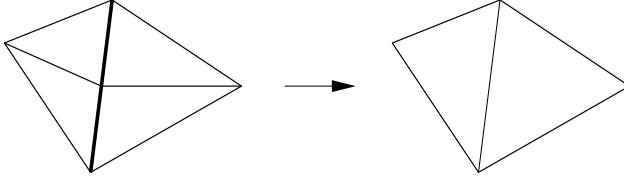


Fig. 1.13. Atomic coarsening operation in two dimensions.

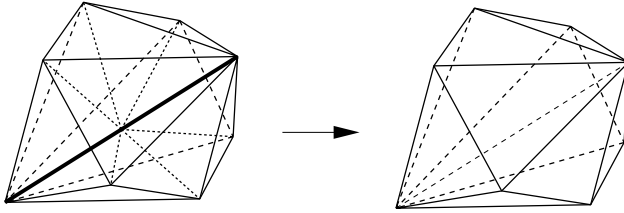


Fig. 1.14. Atomic coarsening operation in three dimensions.

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.

Algorithm 1.9 (Local coarsening).

```

subroutine coarsen_element( $S, \mathcal{S}$ )
   $\mathcal{A} := \{S' \in \mathcal{S}; S' \text{ must not be coarsened with } S\}$ 
  if  $\mathcal{A} = \emptyset$ 
    for all child pairs  $S'_0, S'_1$  at common coarsening edge do
      coarsen  $S'_0$  and  $S'_1$  into the parent  $S'$ 
       $\mathcal{S} := \mathcal{S} \setminus \{S'_0, S'_1\} \cup \{S'\}$ 
    end for
  end if

```

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

Algorithm 1.10 (Coarsening algorithm).

```

subroutine coarsen( $\mathcal{S}$ )
  for all  $S \in \mathcal{S}$  do
    if  $S$  is marked for coarsening
      coarsen_element( $S, \mathcal{S}$ )
    end if
  end for

```

Remark 1.11. 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.

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) have only to be created once on the refinement patch. For example, all elements *share* the new vertex and two children 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.

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 automatically to be adjusted in size.