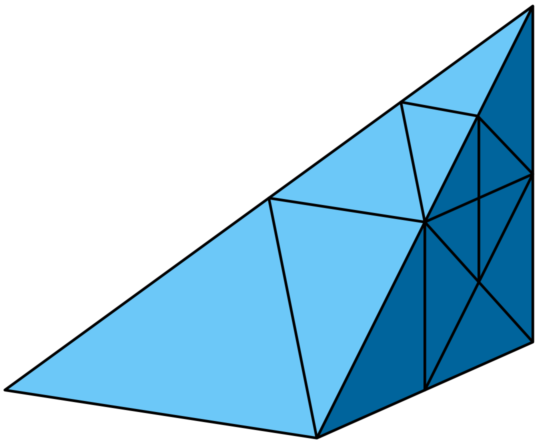
ALBERT: An adaptive hierarchical finite element toolbox

Alfred Schmidt and Kunibert G. Siebert

Institut für Angewandte Mathematik
Albert–Ludwigs–Universität Freiburg i. Br.
Hermann–Herder Straße 10
D–79 104 Freiburg

http://www.mathematik.uni-freiburg.de/IAM/Research/projectsdz/albert



Version: ALBERT-1.0, February 2000

Abstract

ALBERT is an Adaptive multi-Level finite element toolbox using Bisectioning refinement and Error control by Residual Techniques. Its design is based on appropriate data structures holding geometric, finite element, and algebraic information. Using such data structures, abstract adaptive methods for stationary and time dependent problems, assembly tools for discrete systems, and dimension dependent tasks like mesh modifications can be provided in a library. This allows dimension–independent development and programming of a general class of applications. In ALBERT, hierarchical 2d and 3d meshes are stored in binary trees. Several sets of finite elements can be used on the same mesh, either using predefined ones, or by adding new sets for special applications. Depending on the currently used finite element spaces, all degrees of freedom are automatically managed during mesh modifications. Tools for assembling system matrices and right hand sides as well as (linear) system solvers are included.

Contents

Introduction

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.

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 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.

ALBERT 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 ALBERT 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 ALBERT data structures is the abstract concept of a finite element space defined (similar to the definition of a single finite element by Ciarlet [9]) 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* of 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 ALBERT 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 above mentioned tools from the library.

Both geometric and finite element information strongly depend on the space dimension. Thus, mesh modification algorithms and basis functions are implemented for 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 hierarchical mesh

The underlying mesh is a conforming triangulation of the computational domain into simplices, i.e. 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 ALBERT, the recursive refinement by bisection is implemented, using the notation of Kossaczký [26].

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 section). 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. 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 transfered 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. ALBERT 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 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 ([13], 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 we have 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 (BLAS, e.g.) can be applied.

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 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 ALBERT 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 [1] 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 sections.

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 2d, where execution is usually much faster and debugging is much easier (because of simple 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 domain $\Omega \subset \mathbb{R}^d$, $d = 2, 3$ we denote by $L^p(\Omega)$, $1 \le p \le \infty$, the usual Lebesgue spaces with norms

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

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

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

For a differentiable function $f\colon\ \Omega\subset\mathbb{R}^d\to\mathbb{R}$ 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 $f=(f_1,\dots,f_n)\colon\ \Omega\subset\mathbb{R}^d\to\mathbb{R}^n$ 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

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

This report is organized as follows: In Chapter 1 we describe the mathematical concepts of the underlying mesh, DOFs, local basis functions, assemblage of the discrete system, and adaptive methods. In Chapter 2 we describe the ALBERT data structures derived from the concepts of Chapter 1 and all functions implemented in ALBERT. In the last chapter we finally discuss the implementation of three model problems.

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 2d the grid is built 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.

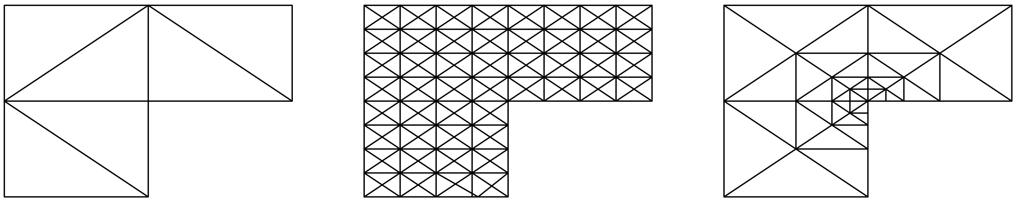


Figure 1.1: Global and local refinement of a triangular mesh.

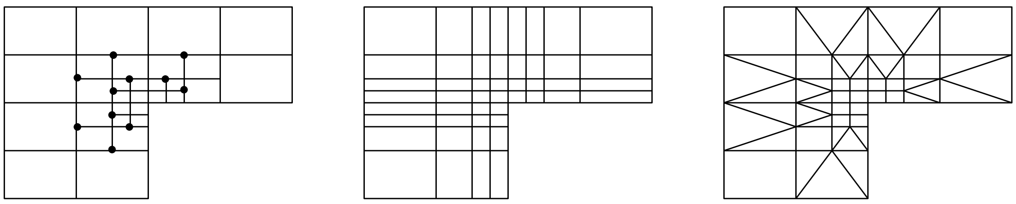


Figure 1.2: Local refinements of a rectangular mesh: with hanging nodes, conforming closure using bisected rectangles, and conforming closure using triangles. Using a conforming closure with rectangles, a local refinement has always global effects up to the boundary.

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

1.1.1. Definition (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

$$S = \text{conv hull}\{a_0, \dots, a_d\} \tag{1.1}$$

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

$$S' = \text{conv hull}\{a'_0, \dots, a'_k\} \subset \partial S \tag{1.2}$$

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

$$\hat{S} = \text{conv hull}\{\hat{a}_0 = 0, \hat{a}_1 = e_1, \dots, \hat{a}_d = e_d\}, \tag{1.3}$$

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

$$h_S := \text{diam}(S) \qquad \text{and} \qquad \rho_S := \sup\{2r;\, B_r \subset S \text{ is a } d\text{–ball of radius } r\}, \tag{1.4}$$

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

$$F_S(\hat{x}) = A_S\,\hat{x} + a_0. \tag{1.5}$$

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

$$\sup_{k\in\mathbb{N}_0} \max_{S\in\mathcal{S}_k} \max_{\hat{x}\in\hat{S}} \text{cond}(DF_S^t(\hat{x}) \cdot DF_S(\hat{x})) < \infty \tag{1.6}$$

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

1.1.4. Remark. For a sequence $\mathcal{S}_k$, $k \geq 0$, of non–parametric triangulations the regularity condition (1.6) 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 ([33, 35]).

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. 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 [5]. 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 regular.

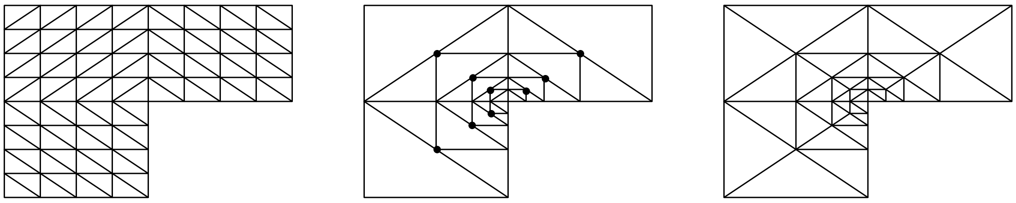


Figure 1.3: Global and local regular refinement of triangles and conforming closure by bisection.

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 [29] 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 [26]. We use the convention, that all vertices of an element are given fixed *local indices.* Valid indices are 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.

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 Figure 1.4 for 2d and in 1.5 for 3d. In 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 types 0, 1, and 2, and 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.

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 classes. In case of the "generic" triangulation of a (unit) square and cube 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 always the longest edge will be the refinement edge and will be bisected, see Figure 1.8.

The refinement of a given triangulation now uses the bisectioning of *single* elements and can be performed either *iteratively* or *recursively.* For tetrahedra, the first description of such refinements was done in the iterative way by Bänsch [2]. It needs the intermediate handling of hanging nodes and not one to one inter–element adjacencies during the refinement process. Two recursive algorithms, which do not create such hanging nodes and are therefore easier to implement, are published by Kossaczký [26] and Maubach [28]. For special 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.

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.

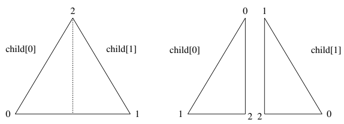


Figure 1.4: Numbering of nodes on parent and children for triangles.

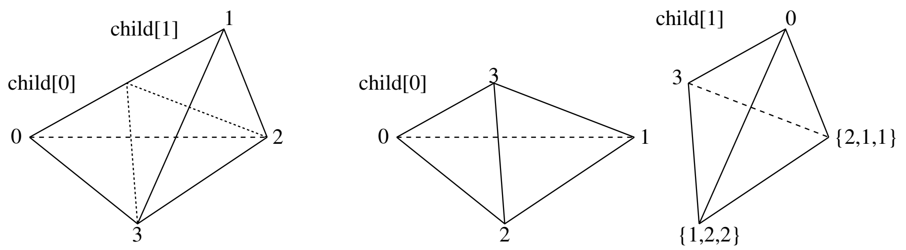


Figure 1.5: Numbering of nodes on parent and children for tetrahedra.

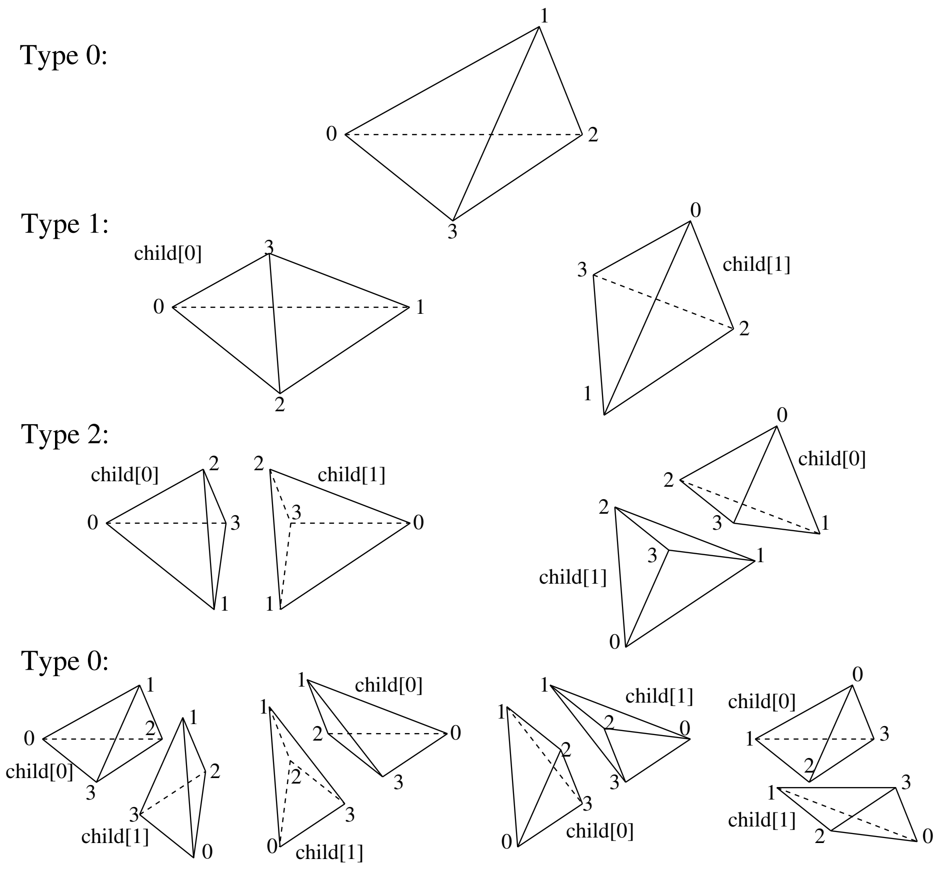


Figure 1.6: Successive refinements of a type 0 tetrahedron.

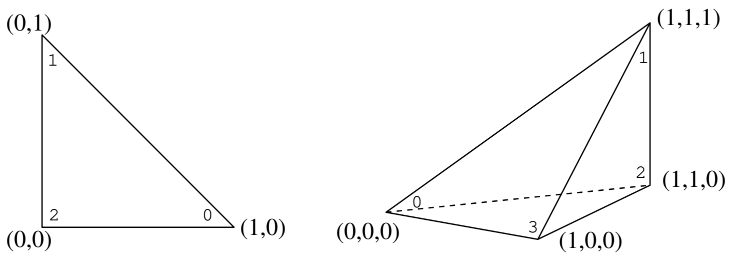


Figure 1.7: Generic elements in two and three dimensions.

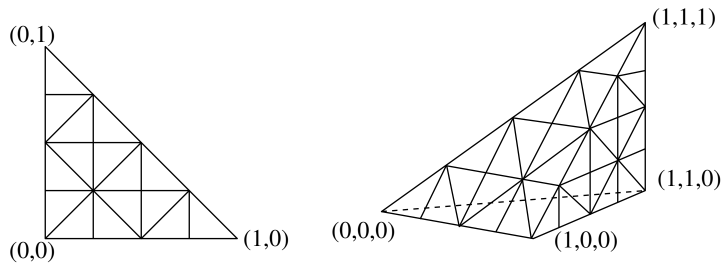


Figure 1.8: Refined generic elements in two and three dimensions.

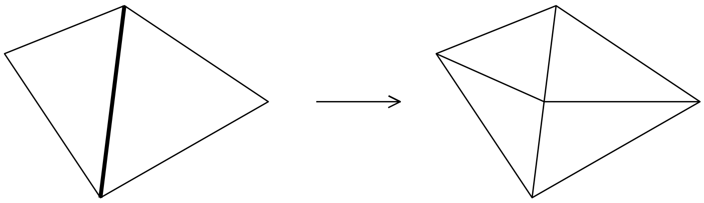


Figure 1.9: Atomic refinement operation in two dimensions. The common edge is the refinement edge for both triangles.

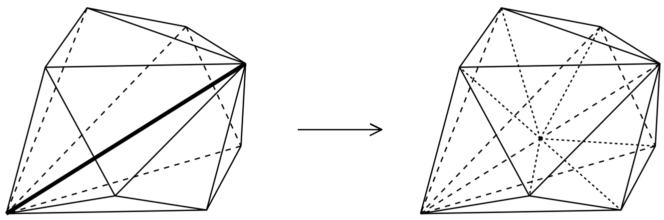


Figure 1.10: Atomic refinement operation in three dimensions. The common edge is the refinement edge for all tetrahedra sharing this edge.

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.

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
    𝒜 := {S' ∈ 𝒮; S' is not compatibly divisible with S}
  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 stops 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.

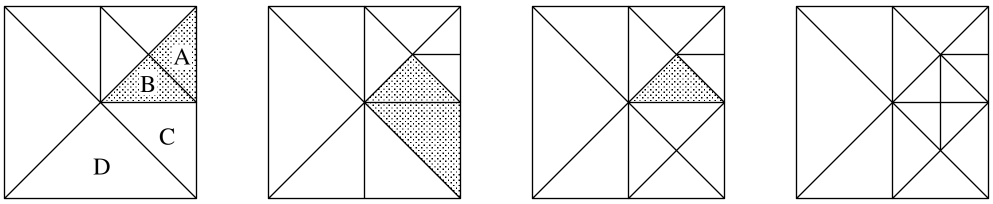


Figure 1.11: Recursive refinement in two dimensions. Triangles A and B are initially marked for refinement.

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 [26] and Mitchell [29] proved

1.1.7. Theorem.

1. *The recursion terminates if the macro triangulation fulfills 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.

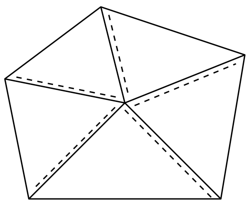


Figure 1.12: A macro triangulation where recursion does not stop.

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 [26].

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 on 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).

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

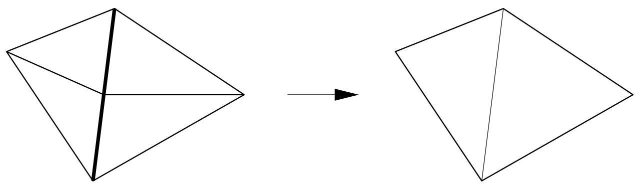


Figure 1.13: Atomic coarsening operation in two dimensions.

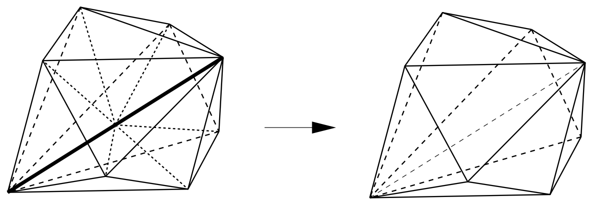


Figure 1.14: Atomic coarsening operation in three dimensions.

if all elements involved in this operation are marked for coarsening. This is the main difference between refinement and coarsening. In an adaptive method it 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, $\mathcal{S}$)
  $\mathcal{A} := \{S' \in \mathcal{S};\ S'$ must not be coarsened with $S\}$
  if $\mathcal{A} = \emptyset$
    for all child pairs S'_0, S'_1 at common coarsening edge do
      coarse S'_0 and S'_1 into the parent S'
      $\mathcal{S} := \mathcal{S} \backslash \{S'_0, S'_1\} \cup \{S'\}$
    end for
  end if

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

1.1.10. Algorithm (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

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.

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, 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.

Transfer of geometric data

Information about the children's/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 nothing has to be done. The $d+1$ vertices of the two children which are 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.

Transformation of finite element information

Using iterative solvers for the linear systems, we need a good initial guess. Usually, the discrete solution from the old grid is such a good guess when it is interpolated into the finite element space on the new grid. 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 loose 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 scalar products 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

In this section we describe the concept of the hierarchical mesh.

There are basically two kinds 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.

ALBERT 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 (not implemented at the moment). The refinement and coarsening routines as 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 1.15.

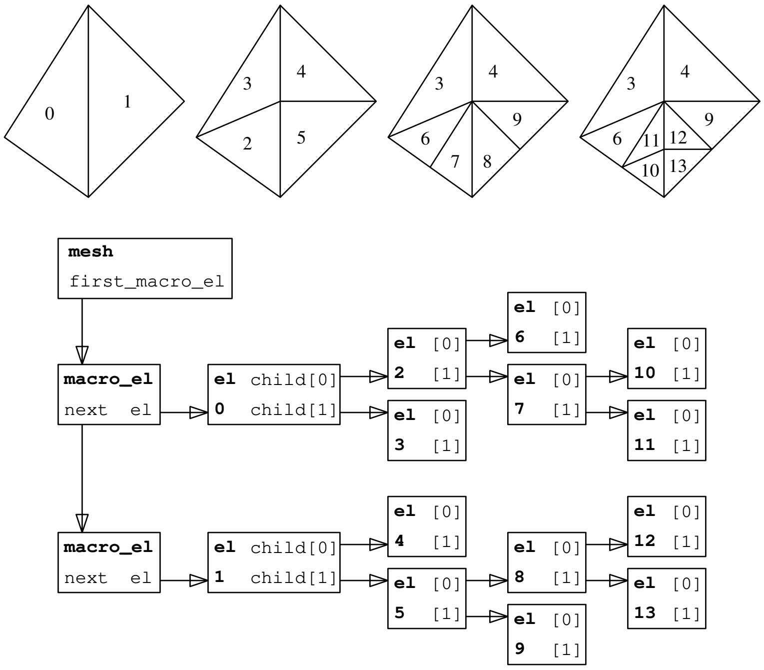


Figure 1.15: Sample mesh refinement and corresponding element trees

Some information is stored on the (leaf) elements explicitly, other information is located at the macro elements and is transfered to the leaf elements while traversing through the binary tree.

Operations on elements can only be performed by using the mesh traversal routines described in Section 2.2.18. 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 through the subtree of the first child and then by traversing through the subtree of the second child. This recursion stops 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. These can be done by simple calculations using the hierarchic structure induced by the refinement algorithm, compare Section 1.1.1.

An example of information which is not stored for each element are the coordinates of the element's vertices (in the case of non-parametric elements and polyhedral boundary). Going from parent to child only the coordinates of one vertex changes 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. But especially in the recursive refinement we have to traverse from an element to some other elements of the mesh by going to a neighbour. This traversing from one neighbour to another takes about 50% of time which is needed for the refinement. If this is too time consuming for special applications, it is also possible to store neighbour information explicitly (which requires more storage but less computational time), compare Section 2.2.3.

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 has only to be available on leaf elements and not for elements inside the binary tree.

Information for leaf elements depends strongly on the application and so it seems not to be appropriate to define a fixed data type for storing this information. Thus, we implemented the following general concept: The user can define his own type for data that should be present on leaf elements. ALBERT only needs the size of memory that is required to store leaf data (one entry in a structure `leaf_data_info` described in Section 2.2.12 in detail). During refinement and coarsening ALBERT automatically allocates and deallocates memory for user data on leaf elements if the data size is bigger than zero. Thus, after grid modifications each leaf element possesses a memory area which is big enough to take leaf data.

To access leaf data we must have for each leaf element a pointer to the provided memory area. This would need an additional pointer on leaf elements. To make the element data structure as small as possible and in order to avoid different element types for leaf and interior elements we "hide" leaf data at the pointer of the second child on leaf elements:

By definition, a leaf element is an element without children. For a leaf element the pointers to the first *and* second child are pointers to `nil`, but since we use a binary tree the pointer to the second child must be `nil` if the pointer to the first child is a `nil` pointer and vice versa. Thus, only testing the first child will give correct information whether an element is a leaf element or not, and we do not have to use the pointer of the second child for this test. As consequence we can use the pointer of the second child as a pointer to the allocated area for leaf data and the user can write or read leaf data via this pointer (using casting to a data type defined by himself).

The consequence is that a pointer to the second child is only a pointer to an element if the pointer to the first child is not a `nil` pointer. Thus testing whether an element is a leaf element or not must only be done using the pointer to the first child.

If the leaf data size equals zero then the pointer to the second child is also a `nil` pointer for leaf elements.

Finally, the user may supply routines for transforming user data from parent to children during refinement and for transforming user data from children to parent during coarsening. If these routines are not supplied, information stored for the parent or the children respectively is lost.

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 ALBERT, every abstract DOF is realized as an integer index into vectors, which corresponds to the global index in a vector of coefficients.

ALBERT supports 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 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.

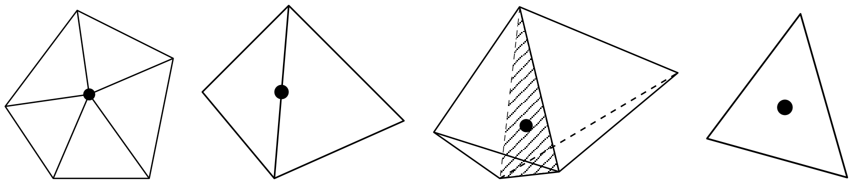


Figure 1.16: Support of basis functions connected with a DOF at a vertex, edge, face (only in 3d), and the interior.

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 $\boxed{\circ}$ (left), piecewise linear $\boxed{+}$ and piecewise quadratic $\boxed{\circ}$ finite elements (middle, Taylor–Hood element for Navier–Stokes: linear pressure and quadratic velocity), piecewise cubic $\boxed{+}$ and piecewise quartic $\boxed{\circ}$ finite elements (right, Taylor–Hood element for Navier–Stokes: quartic velocity and linear pressure).

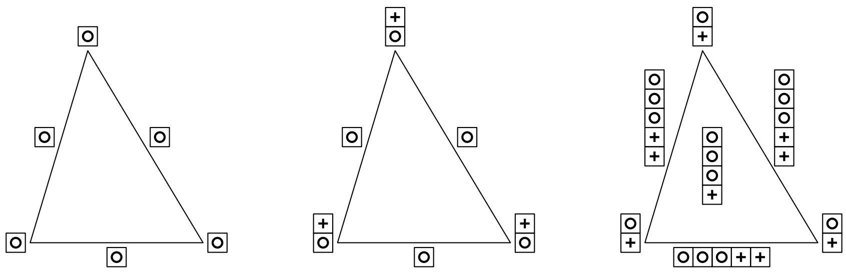


Figure 1.17: Examples of DOF distributions in 2d.

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 of DOFs is mainly done via finite element spaces.

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 ALBERT is future work.

All in all, for a very general situation, we only need a vector of basis functions (and its 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 $\mathbb{R}^{d+1}$:

$$\bar{S} := \big\{(\lambda_0, \ldots, \lambda_d) \in \mathbb{R}^{d+1}; \lambda_k \geq 0, \sum_{k=0}^{d} \lambda_k = 1\big\} \tag{1.7}$$

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 (1.3) 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

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

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

$$\hat{x} = F_S^{-1}(x) \in \hat{S}. \tag{1.8}$$

For a simplex S the easiest choice for F_S is the unique affine mapping (1.5). 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 $\hat{S}$:

$$\lambda^S(x) = \lambda^{\hat{S}}(\hat{x}) = \lambda^{\hat{S}}\left(F_S^{-1}(x)\right) \tag{1.9}$$

$$
\begin{tikzcd}
 & S \arrow[dl, "F_S^{-1}"'] \arrow[dr, "\lambda^S = \lambda^{\hat S} \circ F_S^{-1}"] & \\
\hat S \arrow[rr, "\lambda^{\hat S}"'] & & \bar S
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
 & S & \\
\hat S \arrow[ur, "F_S"] & & \bar S \arrow[ul, "x^S = x^{\hat S} \circ F_S"'] \arrow[ll, "x^{\hat S}"]
\end{tikzcd}
$$

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

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

$$x^S(\lambda) \;=\; F_S\left(\sum_{k=0}^{d} \lambda_k \hat a_k\right) \;=\; F_S\left(x^{\hat S}(\lambda)\right) \tag{1.10}$$

(see also Figure 1.18).

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

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

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

$$\hat{\mathbb{P}} = \left\{\hat\varphi \in C^0(\hat S);\ \bar\varphi \in \bar{\mathbb{P}}\right\} \qquad \text{and} \qquad \mathbb{P}_S = \left\{\varphi \in C^0(S);\ \bar\varphi \in \bar{\mathbb{P}}\right\}. \tag{1.11}$$

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 ALBERT, we use the function space $\bar{\mathbb{P}}$ on $\bar S$; the implementation of a basis $\{\bar\varphi^1, \dots, \bar\varphi^m\}$ of $\bar{\mathbb{P}}$ is much simpler than the implementation of a basis $\{\hat\varphi^1, \dots, \hat\varphi^m\}$ 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

ALBERT 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

$$X_h = X_h(\mathcal{S}, \bar{\mathbb{P}}, C) = \left\{\varphi \in C;\ \varphi_{|S} \in \mathbb{P}_S \text{ for all } S \in \mathcal{S}\right\} \tag{1.12}$$

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

$$\begin{aligned} X_h &= X_h(\mathcal{S}, \bar{\mathbb{P}}, C) \\ &= \left\{\varphi = (\varphi_1, \dots, \varphi_n) \in (C \times C \times \cdots \times C);\ \varphi_{i\,|S} \in \mathbb{P}_S \text{ for all } i = 1, \dots, n,\ S \in \mathcal{S}\right\}. \end{aligned} \tag{1.13}$$

for vector valued finite elements. The spaces $\mathbb{P}_S$ are defined by $\bar{\mathbb{P}}$ via (1.11).

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 $\bar{\mathbb{P}}$, but together with the relation between global and local degrees of freedom every finite element function has globally to satisfy the continuity constraint C. The relation between global and local DOFs is now described in detail.

1.4.3 Evaluation of finite element functions

Let $\{\bar{\varphi}^1, \ldots, \bar{\varphi}^m\}$ 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 $j_S : [1, m] \to J_S$ denotes the inverse mapping of i_S, the connection between global and local basis functions is uniquely determined on each element S by

$$\varphi_j(x(\lambda)) = \bar{\varphi}^{i_S(j)}(\lambda), \qquad \text{for all } \lambda \in \bar{S},\ j \in J_S, \tag{1.14a}$$

$$\varphi_{j_S(i)}(x(\lambda)) = \bar{\varphi}^i(\lambda), \qquad \text{for all } \lambda \in \bar{S},\ i \in \{1, \ldots, m\}. \tag{1.14b}$$

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*

$$\left(u_S^1, \ldots, u_S^m\right) = \left(u_{j_S(1)}, \ldots, u_{j_S(m)}\right) \tag{1.15}$$

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:

$$u_h(x) = \sum_{i=1}^{m} u_S^i \, \bar{\varphi}^i(\lambda(x)) \qquad \text{for all } x \in S. \tag{1.16}$$

In finite element codes, finite element functions are *not* evaluated at *world coordinates* x as in (1.16), but they are evaluated on a single element S at barycentric coordinates λ on S giving the value at the world coordinates $x(\lambda)$:

$$u_h(x(\lambda)) = \sum_{i=1}^{m} u_S^i\, \bar{\varphi}^i(\lambda) \qquad \text{for all } \lambda \in \bar{S}. \tag{1.17}$$

The mapping j_S, which allows the access of 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 2.5.1).

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 $\Lambda = \Lambda_S \in \mathbb{R}^{(\texttt{DIM+1})\times\texttt{DIM\_OF\_WORLD}}$ of the barycentric coordinates on S is involved (we consider here only the case $\texttt{DIM} = \texttt{DIM\_OF\_WORLD} = d$):

$$\Lambda(x) = \begin{pmatrix} \lambda_{0,x_1}(x) & \lambda_{0,x_2}(x) & \cdots & \lambda_{0,x_d}(x) \\ \vdots & \vdots & & \vdots \\ \lambda_{d,x_1}(x) & \lambda_{d,x_2}(x) & \cdots & \lambda_{d,x_d}(x) \end{pmatrix} = \begin{pmatrix} - & \nabla\lambda_0(x) & - \\ & \vdots & \\ - & \nabla\lambda_d(x) & - \end{pmatrix}. \tag{1.18}$$

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

$$\nabla\varphi(x) = \nabla\left(\bar{\varphi}(\lambda(x))\right) = \sum_{k=0}^{d} \bar{\varphi}_{,\lambda_k}(\lambda(x))\nabla\lambda_k(x) = \Lambda^t(x)\nabla_\lambda\bar{\varphi}(\lambda(x)), \qquad x \in S \tag{1.19}$$

and

$$D^2\varphi(x) = \Lambda^t(x)D_\lambda^2\bar{\varphi}(\lambda(x))\Lambda(x) + \sum_{k=0}^{d} D^2\lambda_k(x)\,\bar{\varphi}_{,\lambda_k}(\lambda(x)), \qquad x \in S. \tag{1.20}$$

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

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

Using these equations, we immediately obtain

$$\nabla u_h(x) = \Lambda^t(x)\sum_{i=1}^{m} u_S^i\, \nabla_\lambda\bar{\varphi}^i(\lambda(x)), \qquad x \in S \tag{1.21}$$

and

$$D^2 u_h(x) = \Lambda^t(x)\sum_{i=1}^{m} u_S^i\, D_\lambda^2\bar{\varphi}^i(\lambda(x))\Lambda(x) + \sum_{k=0}^{d} D^2\lambda_k(x)\sum_{i=1}^{m} u_S^i\,\bar{\varphi}^i_{,\lambda_k}(\lambda(x)), \qquad x \in S. \tag{1.22}$$

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

$$\nabla u_h(x(\lambda)) = \Lambda^t(x(\lambda)) \sum_{i=1}^{m} u_S^i \, \nabla_\lambda \bar\varphi^i(\lambda), \qquad \lambda \in \bar S \tag{1.23}$$

and

$$D^2 u_h(x(\lambda)) = \Lambda^t(x(\lambda)) \sum_{i=1}^{m} u_S^i \, D_\lambda^2 \bar\varphi^i(\lambda) \Lambda(x(\lambda)) + \sum_{k=0}^{d} D^2\lambda_k(x(\lambda)) \sum_{i=1}^{m} u_S^i \, \bar\varphi^i_{,\lambda_k}(\lambda), \qquad \lambda \in \bar S. \tag{1.24}$$

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 does only depend on Λ and can be performed by some general evaluation routine (compare Section 2.9).

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

$$A = (a_{ij})_{\substack{i=1,\ldots,m \\ j=1,\ldots,k}}. \tag{1.25}$$

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

$$B = (b_{ij})_{\substack{i=1,\dots,m\\ j=1,\dots,k}}. \tag{1.26}$$

This matrix B is used for the interpolation during coarsening.

Both matrices do only depend 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. Such a DOF is connected to a basis function which has a support on all elements sharing this DOF. Every DOF refers to one coefficient of a finite element function, and this coefficient must only be calculated 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.

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 $\mathbb{I}^S_{S_0,S_1}u_h$ with respect to $\{\psi^j\}$. Using matrix A defined in (1.25) we conclude

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

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.

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 (1.25) 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.

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 (1.26):

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

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

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

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 $\{x_\varphi^i\}_{i=1,\dots,m}$ on S and $\{\psi^j\}_{j=1,\dots,k}$ be the Lagrange basis functions of $X_{S_0\cup S_1}$ with Lagrange nodes $\{x_\psi^j\}_{j=1,\dots,k}$ on $S_0 \cup S_1$. The Matrix A is then given by

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

and matrix B is given by

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

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:

$$\begin{aligned} Lu := -\nabla\cdot A\nabla u + b\cdot\nabla u + c\,u &= f && \text{in } \Omega, && (1.27\text{a})\\ u &= g && \text{on } \Gamma_D, && (1.27\text{b})\\ \nu_\Omega\cdot A\nabla u &= 0 && \text{on } \Gamma_N, && (1.27\text{c})\end{aligned}$$

where $A \in L^\infty(\Omega;\mathbb{R}^{d\times d})$, $b \in L^\infty(\Omega;\mathbb{R}^d)$, $c \in L^\infty(\Omega)$, 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 normal vector on $\partial\Omega$. The boundary condition (1.27c) is a so called *natural* Neumann condition.

Equations (1.27) 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

$$X = H^1(\Omega) \qquad \text{and} \qquad \mathring{X} = \{v \in H^1(\Omega);\, v = 0 \text{ on } \Gamma_D\} \tag{1.28}$$

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

$$\int_\Omega (\nabla\varphi(x))\cdot A(x)\nabla u(x) + \varphi(x)\,b(x)\cdot\nabla u(x) + c(x)\,\varphi(x)\,u(x)\,dx = \int_\Omega f(x)\,\varphi(x)\,dx \tag{1.29}$$

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

$$\langle Lv,\, \varphi\rangle_{\mathring{X}^*\times\mathring{X}} := \int_\Omega \nabla\varphi\cdot A\nabla v + \int_\Omega \varphi\, b\cdot\nabla v + \int_\Omega c\,\varphi\, v \qquad \text{for all } v, \varphi \in \mathring{X} \tag{1.30}$$

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

$$\langle F,\, \varphi\rangle_{\mathring{X}^*\times\mathring{X}} := \int_\Omega f\,\varphi \qquad \text{for all } \varphi \in \mathring{X}. \tag{1.31}$$

With these identifications we use the following reformulation of (1.29): Find $u \in X$ such that

$$u \in g + \mathring{X} : \qquad L\,u = f \qquad \text{in } \mathring{X}^* \tag{1.32}$$

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 (1.32) is then a direct consequence of the Lax–Milgram–Theorem.

We consider a finite dimensional subspace $X_h \subset X$ for the discretization of (1.32) 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 (1.32) is then given by: Find $u_h \in X_h$ such that

$$u_h \in g_h + \mathring{X}_h : \qquad L\,u_h = f \qquad \text{in } \mathring{X}_h^*, \tag{1.33}$$

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 (1.33), again using the Lax–Milgram–Theorem.

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

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

Using (1.33) with test functions φ_i, $i = 1, \dots, \mathring{N}$, we get the following N equations for the coefficient vector $\boldsymbol{u} = (u_1, \dots, u_N)$ of u_h:

$$\sum_{j=1}^{N} u_j \langle L\varphi_j,\, \varphi_i\rangle_{\mathring{X}_h^*\times\mathring{X}_h} = \langle f,\, \varphi_i\rangle_{\mathring{X}_h^*\times\mathring{X}_h} \qquad \text{for } i = 1, \dots, \mathring{N}, \tag{1.34a}$$

$$u_i = g_i \qquad \text{for } i = \mathring{N}+1, \dots, N. \tag{1.34b}$$

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

$$\boldsymbol{L} := \begin{bmatrix} \langle L\varphi_1, \varphi_1\rangle & \dots & \langle L\varphi_{\mathring{N}}, \varphi_1\rangle & \langle L\varphi_{\mathring{N}+1}, \varphi_1\rangle & \dots & & \langle L\varphi_N, \varphi_1\rangle \\ \vdots & \ddots & \vdots & \vdots & & \ddots & \vdots \\ \langle L\varphi_1, \varphi_{\mathring{N}}\rangle & \dots & \langle L\varphi_{\mathring{N}}, \varphi_{\mathring{N}}\rangle & \langle L\varphi_{\mathring{N}+1}, \varphi_{\mathring{N}}\rangle & \dots & & \langle L\varphi_N, \varphi_{\mathring{N}}\rangle \\ 0 & \dots & 0 & 1 & 0 & \dots & 0 \\ 0 & \dots & 0 & 0 & 1 & \dots & 0 \\ \vdots & \ddots & 0 & 0 & 0 & \ddots & \vdots \\ 0 & \dots & 0 & 0 & 0 & \dots & 1 \end{bmatrix} \tag{1.35}$$

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

$$\boldsymbol{f} := \begin{bmatrix} \langle f, \varphi_1 \rangle \\ \vdots \\ \langle f, \varphi_{\mathring{N}} \rangle \\ g_{\mathring{N}+1} \\ \vdots \\ g_N \end{bmatrix}, \tag{1.36}$$

we can write (1.34) as the linear $N \times N$ system

$$\boldsymbol{L}\,\boldsymbol{u} = \boldsymbol{f}, \tag{1.37}$$

which has to be assembled and solved numerically.

1.4.6 Numerical quadrature

For the assemblage of the system matrix and right hand side vector of the linear system (1.37), 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 $\{(w_k, \lambda_k) \in \mathbb{R} \times \mathbb{R}^{d+1}; k = 0, \ldots, n_Q - 1\}$ 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

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

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 $f : S \to \mathbb{R}$:

$$\int_S f(x)\,dx \approx Q_S(f) := \hat{Q}((f \circ F_S)|\det DF_S|) = \sum_{k=0}^{n_Q-1} w_k f(x(\lambda_k))|\det DF_S(\hat{x}(\lambda_k))|. \tag{1.38}$$

For a simplex S this results in

$$Q_S(f) = d!\,|S| \sum_{k=0}^{n_Q-1} w_k f(x(\lambda_k)). \tag{1.39}$$

1.4.7 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 (1.29) 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 (1.14a) 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* $\boldsymbol{f}_S = (f_S^1, \dots, f_S^m)^t$, where

$$f_S^i = \int_{\hat{S}} f(F_S(\hat{x}))\bar{\varphi}^i(\lambda(\hat{x}))|\det DF_S(\hat{x})|\,d\hat{x}, \qquad i = 1, \dots, m. \tag{1.40}$$

Denoting again by $j_S : [1, m] \to J_S$ the function which connects the local DOFs with the global DOFs (defined in (1.14b)), the values f_S^i are then added to the $j_S(i)$th component of the right hand side vector $\boldsymbol{f}$, $i = 1, \dots, m$.

For general f, the integrals in (1.40) can not be calculated exactly and we have to use a quadrature formula for the approximation of the integrals (compare Section 1.4.6). Given a numerical quadrature $\hat{Q}$ on $\hat{S}$ we approximate

$$f_S^i \approx \hat{Q}\left((f \circ F_S)\,(\bar{\varphi}^i \circ \lambda)|\det DF_S|\right) = \sum_{k=0}^{n_Q-1} w_k f(x(\lambda_k))\,\bar{\varphi}^i(\lambda_k)|\det DF_S(\hat{x}(\lambda_k))|. \tag{1.41}$$

For a simplex S this is simplified to

$$f_S^i \approx d!\,|S| \sum_{k=0}^{n_Q-1} w_k f(x(\lambda_k))\,\bar{\varphi}^i(\lambda_k). \tag{1.42}$$

In ALBERT, 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 2.5). 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

(1.19) and the relation (1.14) 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

$$\bar A(\lambda) := (\bar a_{kl}(\lambda))_{k,l=0,\dots,d} := |\det DF_S(\hat x(\lambda))|\,\Lambda(x(\lambda))\,A(x(\lambda))\,\Lambda^t(x(\lambda)), \tag{1.43a}$$

$$\bar b(\lambda) := \big(\bar b_l(\lambda)\big)_{l=0,\dots,d} := |\det DF_S(\hat x(\lambda))|\,\Lambda(x(\lambda))\,b(x(\lambda)), \quad \text{and} \tag{1.43b}$$

$$\bar c(\lambda) := |\det DF_S(\hat x(\lambda))|\,c(x(\lambda)) \tag{1.43c}$$

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

$$\begin{aligned}L_S^{ij} = \int_{\hat S} \nabla_\lambda\bar\varphi^i(\lambda(\hat x))\cdot\bar A(\lambda(\hat x))\,\nabla_\lambda\bar\varphi^j(\lambda(\hat x))\,d\hat x + \int_{\hat S}\bar\varphi^i(\lambda(\hat x))\,\bar b(\lambda(\hat x))\cdot\nabla_\lambda\bar\varphi^j(\lambda(\hat x))\,d\hat x\\ + \int_{\hat S}\bar c(\lambda(\hat x))\,\bar\varphi^i(\lambda(\hat x))\,\bar\varphi^j(\lambda(\hat x))\,d\hat x,\end{aligned} \tag{1.44}$$

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

$$\begin{aligned}L_S^{ij} = \sum_{k,l=0}^{d}\int_{\hat S}\bar a_{kl}(\lambda(\hat x))\,\bar\varphi^i_{,\lambda_k}(\lambda(\hat x))\,\bar\varphi^j_{,\lambda_l}(\lambda(\hat x))\,d\hat x + \sum_{l=0}^{d}\int_{\hat S}\bar b_l(\lambda(\hat x))\,\bar\varphi^i(\lambda(\hat x))\,\bar\varphi^j_{,\lambda_l}(\lambda(\hat x))\,d\hat x\\ + \int_{\hat S}\bar c(\lambda(\hat x))\,\bar\varphi^i(\lambda(\hat x))\,\bar\varphi^j(\lambda(\hat x))\,d\hat x,\end{aligned} \tag{1.44'}$$

$i,j = 1,\dots,m$. 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

$$\boldsymbol{L}_S^{ij} \approx \hat Q_2\left(\sum_{k,l=0}^{d}(\bar a_{kl}\bar\varphi^i_{,\lambda_k}\,\bar\varphi^j_{,\lambda_l})\circ\lambda\right) + \hat Q_1\left(\sum_{l=0}^{d}(\bar b_l\,\bar\varphi^i\,\bar\varphi^j_{,\lambda_l})\circ\lambda\right) + \hat Q_0\Big((\bar c\,\bar\varphi^i\,\bar\varphi^j)\circ\lambda\Big), \tag{1.45}$$

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

$$\bar a_{kl}(\lambda_q), \quad \bar b_l(\lambda_q), \quad \bar c(\lambda_q) \tag{1.46}$$

at all quadrature points λ_q on S, (1.45) 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 2.3.4.

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, the abbreviations (1.43) are on each simplex S simplified to

$$\bar{A}_S = (\bar{a}_{kl})_{k,l=0,\dots,d} = d!|S|\,\Lambda\,A_{|S}\,\Lambda^t, \quad \bar{b}_S = \left(\bar{b}_l\right)_{l=0,\dots,d} = d!|S|\,\Lambda\,b_{|S}, \quad \bar{c}_S = d!|S|\,c_{|S}. \tag{1.47}$$

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

$$\boldsymbol{L}_S^{ij} \approx \sum_{k,l=0}^{d} \bar{a}_{kl}\hat{Q}_2\Big((\bar{\varphi}^i_{,\lambda_k}\,\bar{\varphi}^j_{,\lambda_l})\circ\lambda\Big) + \sum_{l=0}^{d} \bar{b}_l\,\hat{Q}_1\Big((\bar{\varphi}^i\,\bar{\varphi}^j_{,\lambda_l})\circ\lambda\Big) + \bar{c}_S\,\hat{Q}_0\Big((\bar{\varphi}^i\,\bar{\varphi}^j)\circ\lambda\Big) \tag{1.48}$$

$i,j = 1,\dots,m$. Here, the numerical quadrature only involves basis functions on the standard simplex. Theses values have to 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.

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 2.12.

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 ALBERT 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

$$\|u - u_h\| \;\leq\; \eta(u_h) \;=\; \left(\sum_{S\in\mathcal{S}} \eta_S(u_h)^p\right)^{1/p}, \qquad p \in [1,\infty) \tag{1.49}$$

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 reduction of the error, 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 [1] 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 $\mathcal{S}_k$, $k = 1, 2, \dots$, until the estimated error is below the given tolerance:

1.5.1. Algorithm (General adaptive refinement strategy).

Start with $\mathcal{S}_0$ and error tolerance tol

> $k := 0$
> solve the discrete problem on $\mathcal{S}_k$
> compute global error estimate η and local indicators η_S, $S \in \mathcal{S}_k$
> while $\eta > tol$ do
>   mark elements for refinement (or coarsening)
>   adapt mesh $\mathcal{S}_k$, producing $\mathcal{S}_{k+1}$
> k := $k+1$
>   solve the discrete problem on $\mathcal{S}_k$
>   compute global error estimate η and local indicators η_S, $S \in \mathcal{S}_k$
> 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 [1]. 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 simple strategy is the maximum strategy. A threshold $\gamma \in (0,1)$ is given, and all elements $S \in \mathcal{S}_k$ with

$$\eta_S \;>\; \gamma \max_{S' \in \mathcal{S}_k} \eta_{S'} \tag{1.50}$$

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 (1.49) is $p = 2$ [39, 40].

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 is 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[19]).

```
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 $N_{k+1} > N_k$, 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 [15] describes a strategy with a guaranteed relative error reduction for the Poisson equation.

We need the assumptions, that

- given data of the problem (like the right hand side) is sufficiently resolved by the current mesh $\mathcal{S}_k$ (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[15]).

```
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 [14] 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 [30] could remove the assumption that data is sufficiently resolved on $\mathcal{S}_k$ 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 [1] 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 [23] 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 [25, 36], or the decision of local h– or p–enrichment of the finite element space [11, 32].

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

$$\eta_S \;\le\; c h_S^\mu,$$

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

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

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 [16] 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 2.3.3. 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 [16] 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 [3] 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 [20]. 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 be not equally distributed between 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
```

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 [19] 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

$$\begin{aligned} \|u(t_N) - U_N\| \ \leq \ & \|u_0 - U_0\| + \max_{1\le n\le N} \Bigg(C_1 \|h_n^2 f\|_{L^\infty(I_n, L^2(S))} + C_2 \int_{I_n} \|f\|\,dt \\ & + C_3 \Big(\sum_{e\in E_n} h_e^4 \Big| \Big[\frac{\partial U_n}{\partial \nu_e} \Big] \Big|^2 \Big)^{1/2} + C_4 \|U_n - U_{n-1}\| + C_5 \Big\| h_n^2 \frac{[U_{n-1}]}{\tau_n} \Big\| \Bigg), \end{aligned}$$

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:

$$\begin{aligned}
\eta_0 &= \|u_0 - U_0\|, \\
\eta_h &= \sum_{S\in\mathcal{S}_n} \left(\tilde{C}_1 \|h_n^2 f\|_{L^\infty(I_n,L^2(S))} + \tilde{C}_3 \Big(\frac{1}{2} \sum_{e\subset\partial S} h_e^4 \Big| \Big[\frac{\partial U_n}{\partial \nu_e} \Big] \Big|^2 \Big)^{1/2} \right), \\
\eta_c &= \sum_{S\in\mathcal{S}_n} \left(\tilde{C}_5 \Big\| h_n^2 \frac{[U_{n-1}]}{\tau_n} \Big\|_{L^2(S)} \right), \\
\eta_\tau &= \tau_n \left(C_2 \|f\|_{L^\infty(I_n,L^2(S))} + C_4 \Big\| \frac{U_n - U_{n-1}}{\tau_n} \Big\| \right).
\end{aligned}$$

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 $\Gamma_h\, tol$ to be produced by the spatial discretization, and another portion $\Gamma_\tau\, tol$ of the error to be produced by the time discretization, with $\Gamma_h + \Gamma_\tau \leq 1.0$. The adaptive procedure now tries to adjust time step sizes and meshes such that 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 $\delta_1 \approx 0.5$, for example.

1.5.8. Algorithm (Time step size control).

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

τ := τold
Solve time step problem and estimate the error
while ητ > θ1 Γτ tol do
  τ := δ1 τ
  Solve time step problem and estimate the error
end while
if ητ ≤ θ2 Γτ 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 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, δ_1 ∈ (0,1), δ_2 > 1, θ_1 ∈ (0,1), θ_2 ∈ (0,θ_1),
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 [6, 7, 8] describes an adaptive variable order time discretization method, combined with implicit strategy B using the extrapolation marking strategy for the mesh adaption.

Chapter 2

Data structures and implementation

The ALBERT toolbox provides two header files albert_util.h and albert.h, which contain the definitions of all data structures, macros, and subroutine prototypes. The file albert_util.h is included in the header file albert.h.

2.1 Basic types, utilities, and parameter handling

The file albert_util.h contains some type definitions and macro definitions for memory (de-) allocation and messages, which we describe briefly in this section. The following system header files are included in albert_util.h

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <float.h>
```

2.1.1 Basic types

ALBERT uses the following basis symbolic constants and macro definition:

```
#define true         1
#define false        0
#define nil          NULL
#define MAX(a, b)    ((a) > (b) ? (a) : (b))
#define MIN(a, b)    ((a) < (b) ? (a) : (b))
#define ABS(a)       ((a) >= 0 ? (a) : -(a))
#define SQR(a)       ((a)*(a))
```

In order to store information in a compact way, we define two bit fields U_CHAR and S_CHAR:

```
typedef unsigned char      U_CHAR;
typedef signed char        S_CHAR;
```

The mesh traversal routines need flags which are stored in the data type FLAGS:

```
typedef unsigned long      FLAGS;
```

By the data type REAL the user can specify to store floating point values in the type float or double. All pointers to variables or vectors of floating point values have to be defined as REAL!

```
typedef double              REAL;
```

The use of `float` is also possible, but it is not guaranteed to work and may lead to problems when using other libraries (like libs for linear solver or graphics, e.g.).

2.1.2 Message macros

There are several macros to write messages and error messages. Especially for error messages the exact location of the error is of interest. Thus, error messages are preceded by the name of the source file and the line number at that this error was detected. Such information is produced by the `C`-preprocessor. Additionally, the name of the function is printed. Since there is no symbolic constant defined by the `C`-preprocessor holding the function name, in each function a string `funcName` containing the name of the function has to be defined. This is usually done by the macro `FUNCNAME`

```
#define FUNCNAME(nn)  const char *funcName = nn
```

as the first declaration:

2.1.1. Example (FUNCNAME).

```
static void refine_element(EL *el)
{
   FUNCNAME("refine_element");

   ...
}
```

All message macros use this local variable `funcName` and it has to be defined in each function. Usual output to `stdout` is done by the macro `MSG()` which has the same arguments as `printf()`:

```
MSG(char *format, ...);
```

The format string should be ended with the newline character '`\n`'. `MSG()` usually precedes the message by the function's name. If the last message was produced by the same function, the function's name is omitted and the space of the name is filled with blanks.

If the format string of `MSG()` does not end with the newline character, and one wants to print more information to the same line, this can be done by `print_msg()` which again has the same arguments as `printf()`:

```
print_msg(char *format, ...);
```

2.1.2. Example (MSG(), print_msg()).

```
static void refine_element(EL *el)
{
   FUNCNAME("refine_element");

   ...

   MSG("refining element %d\n", INDEX(el));
   MSG("neighbours of element: ");
   for (i = 0; i < N_VERTICES-1; i++)
      print_msg("%d, ", INDEX(NEIGH(el)[i]));
```

```
    print_msg("%d\n", INDEX(NEIGH(el)[N_VERTICES-1]));
}
```

produces for instance output

```
refine_element:    refining element 10
                   neighbours of element: 0, 14, 42
```

Often it is useful to suppress messages or to give only information up to a suitable level. There are two ways for defining such a level of information. The first one is a local level, which is determined by some variable in a function; the other one is a global restriction for information. For this global restriction a global variable

```
int      msg_info = 10;
```

is defined with an default value of `10`. Using one of the macros

```
#define INFO(info,noinfo)\
   if (!(msg_info&&(MIN(msg_info,(info))>=(noinfo))));else MSG
#define PRINT_INFO(info,noinfo)\
   if (!(msg_info&&(MIN(msg_info,(info))>=(noinfo))));else print_msg
```

only messages are produced by `INFO()` or `PRINT_INFO()` if `msg_info` is non zero and the value `MIN(msg_info, info)` is greater or equal `noinfo`, where `noinfo` denotes some local level of information. Thus after setting `msg_info = 0`, no further messages are produced. Changing the value of this variable via a parameter file is described below in Section 2.1.5.

2.1.3. Example (`INFO()`, `PRINT_INFO()`).

```
static void refine_element(EL *el)
{
   FUNCNAME("refine_element");

   ...

   INFO(info,4)("refining element %d\n", INDEX(el));
   INFO(info,6)("neighbours of element: ");
   for (i = 0; i < N_VERTICES-1; i++)
      PRINT_INFO(info,6)("%d, ", INDEX(NEIGH(el)[i]));
    PRINT_INFO(info,6)("%d\n", INDEX(NEIGH(el)[N_VERTICES-1]));
}
```

will print the element index, if the value of the global variable `info` $\geq$ `4` and additionally the indices of neighbours if `info` $\geq$ `6`.

For error messages macros `ERROR` and `ERROR_EXIT` are defined. `ERROR` has the same functionality as the `MSG` macro but the output is piped to `stderr`. `ERROR_EXIT` exits the program after using the `ERROR` macro with return value 1:

```
ERROR(char *format, ...);
ERROR_EXIT(char *format, ...);
```

Furthermore, two macros for testing boolean values are available:

```
#define TEST(test)      if(test); else ERROR
#define TEST_EXIT(test) if(test); else ERROR_EXIT
```

If `test` is not true both macros will print an error message. `TEST` will continue the program afterwards, meanwhile `TEST_EXIT` will exit the program with return value 1.

Error messages can not be suppressed and the information variable `msg_info` does not influence these error functions.

2.1.4. Example (`TEST()`, `TEST_EXIT()`).

```
static void refine_element(EL *el)
{
   FUNCNAME("refine_element");

   TEST_EXIT(el)("no element for refinement\n");
   ...
}
```

Finally, there exists a macro `WARNING` for writing warnings to the same stream as for messages. Each warning is preceeded by the word `WARNING`. Warnings can not be suppressed by the information variable `msg_info`.

```
WARNING(char *format, ...);
```

Sometimes it may be very useful to write messages to file, or write parts of messages to file. By the functions

```
void change_msg_out(FILE *fp);
void open_msg_file(const char *filename, const char *type);
```

the user can select a new stream or file for message output. Using the first routine, the user directly specifies the new stream `fp`. If this stream is non nil, all messages are flushed to this stream, otherwise ALBERT will use the old stream furthermore. The function `open_msg_file()` acts like `change_msg_out(fopen(filename, type))`.

Similar functions are available for error messages and they act in the same manner on the output stream for error messages:

```
void change_error_out(FILE *fp);
void open_error_file(const char *filename, const char *type);
```

For setting breakpoints in the program two macros

```
WAIT
WAIT_REALLY
```

are defined.

`WAIT`: this macro uses a global variable `msg_wait` and if the value of this variable is not zero the statement `WAIT;` will produce the message

```
wait for <enter> ...
```

and will continue after pressing the `enter` or `return` key. If the value of `msg_wait` is zero, no message is produced and the program continues. The value of `msg_wait` can be modified by the parameter tools (see Section 2.1.5);

`WAIT_REALLY`: the statement `WAIT_REALLY` will always produce the above message and will wait for pressing the `enter` or `return` key.

2.1.3 Memory allocation and deallocation

Various functions and macros for memory allocation and deallocation are implemented. The basic ones are

```
void *albert_alloc(size_t size, const char *name, const char *file, int line);
void *albert_realloc(void *ptr, size_t o_size, size_t n_size,
                     const char *name, const char *file, int line);
void *albert_calloc(size_t n_el, size_t el_size,
                    const char *name, const char *file, int line);
void albert_free(void *ptr, size_t size);
```

In the following `name` is a pointer to the string holding the function name of the calling function (defined by the `FUNCNAME` macro, e.g.), `file` a pointer to the string holding the name of the source file (generated by the `__FILE__` CPP macro) and `line` is the line number of the call (generated by the `__LINE__` CPP macro). All functions will exit the running program with an error message, if the size to be allocated is 0 or the memory allocation by the system functions fails.

`albert_alloc(size, name, file, line)`: returns a pointer to a block of memory of at least the number of bytes specified by `size`;

`albert_realloc(ptr, o_size, n_size, name, file, line)`: changes the size of the block of memory pointed to by the pointer `ptr` to the number of bytes specified by `n_size`, and returns a pointer to the block. The contents of the block remain unchanged up to the lesser of the `o_size` and `n_size`; if necessary, a new block is allocated, and data is copied to it; if the `ptr` is a `nil` pointer, the `albert_realloc()` function allocates a new block of the requested size;

`albert_calloc(n_el, el_size, name, file, line)`: returns a pointer to a vector with the `n_el` number of elements, where each element is of the size `el_size`; the space is initialized to zeros;

`albert_free(ptr, size)`: frees the block of memory pointed to by the argument `ptr` for further allocation; `ptr` must have been previously allocated by either `albert_alloc()`, `albert_realloc()`, or `albert_calloc()`.

A more comfortable way to use these functions, is the use of the following macros:

```
TYPE* MEM_ALLOC(size_t n, TYPE);
TYPE* MEM_REALLOC(TYPE *ptr, size_t n_old, size_t n_new, TYPE);
TYPE* MEM_CALLOC(size_t n, TYPE);
TYPE* MEM_FREE(TYPE *ptr, size_t n, TYPE);
```

They supply the above described functions with the function name, file name and line number automatically. For an allocation by these macros, the number of elements and the data type have to be specified; the actual size in bytes is computed automatically. Additionally, casting to the correct type is performed.

`MEM_ALLOC(n, TYPE)`: returns a pointer to a vector of type `TYPE` with the `n` number of elements;

`MEM_REALLOC(ptr, n_old, n_new, TYPE)`: reallocates the vector of type `TYPE` at pointer `ptr` with `n_old` elements for `n_new` elements; values of the vector are not changed for all elements up to the minimum of `n_old` and `n_new`; returns a pointer to the new vector;

`MEM_CALLOC(n, TYPE)`: returns a pointer to a vector of type `TYPE` with the `n` number of elements; the elements are initialized to zeros;

`MEM_FREE(ptr, n, TYPE)`: frees a vector of type `TYPE` with `n` number of elements at `ptr`, allocated previously by either `MEM_ALLOC()`, `MEM_REALLOC()`, or `MEM_CALLOC()`.

2.1.5. Example (MEM_ALLOC(), MEM_FREE()).

```
REAL  *u = MEM_ALLOC(10, REAL);
...
MEM_FREE(u, 10, REAL);
```

allocates a vector of 10 REALs and frees this vector again.

For some applications matrices are needed too. Matrices can be allocated and freed by the functions

```
void **albert_matrix(size_t nr, size_t nc, size_t el_size,
                     const char *name, const char *file, int line);
void free_albert_matrix(void **ptr, int nc int nr, size_t el_size);
```

albert_matrix(nr, nc, el_size, name, file, line): returns a pointer `**ptr` to a matrix with `nr` number of rows and `nc` number of columns, where each element is of size `el_size`; `name` is a string holding the name of the calling function, `file` a string holding the name of the source file and `line` the line number of the call;

free_albert_matrix(ptr, nr, nc, el_size): frees the matrix pointed to by `ptr`, previously allocated by `albert_matrix()`.

Again, the following macros simplify the use of the above functions:

```
TYPE** MAT_ALLOC(size_t nr, size_t nc, TYPE);
void   MAT_FREE(TYPE **ptr, size_t nr, size_t nc, TYPE);
```

They supply the above described functions with the function name, file name and line number automatically. These macros need the type of the matrix elements instead of the size. Casting to the correct type is performed.

MAT_ALLOC(nr,nc,type): returns a pointer `**ptr` to a matrix with elements `ptr[i][j]` of type `TYPE` and indices in the range $0 \le i < nr$ and $0 \le j < nc$;

MAT_FREE(ptr,nr,nc,type): frees a matrix allocated by `MAT_ALLOC()`;

Many subroutines need additional workspace for storing vectors, etc. (linear solvers like conjugate gradient methods, e.g.). Many applications need such kinds of workspaces for several functions. In order to make handling of such workspaces easy, a data structure `WORKSPACE` is available. In this data structure a pointer to the workspace and the actual size of the workspace is stored.

```
typedef struct workspace    WORKSPACE;

struct workspace
{
  size_t  size;
  void    *work;
};
```

The members yield following information:

size: actual size of the workspace in bytes;

work: pointer to the workspace;

A workspace can be accessed by the function

```
WORKSPACE *realloc_workspace(WORKSPACE *work_space, size_t size,
                             const char *name, const char *file, int line);
```

Description:

`realloc_workspace(work_space, size, name, file, line)`: return value is a pointer to a `WORKSPACE` structure holding a vector of at least length `size` bytes; the member `size` holds the true length of the vector `work`; if `work_space` is a `nil` pointer, a new `WORKSPACE` structure is allocated; `name` is a string holding the name of the calling function, `file` a string holding the name of the source file and `line` the line number of the call;

The macro

```
WORKSPACE* REALLOC_WORKSPACE(WORKSPACE*ws, size_t size)
```

simplifies the use of `realloc_workspace()` by supplying the function with `name`, `file`, and `line`.

`REALLOC_WORKSPACE(ws, size)`: returns a pointer to `WORKSPACE` structure holding a vector of at least length `size` bytes; the member `size` holds the true length of the vector `work`; if `ws` is a `nil` pointer, a new `WORKSPACE` structure is allocated;

The functions

```
void clear_workspace(WORKSPACE *ws);
void free_workspace(WORKSPACE *ws);
```

are used for `WORKSPACE` deallocation. Description:

`clear_workspace(ws)`: frees the vector `ws->work` and sets `ws->work` to `nil` and `ws->size` to `0`; the structure `ws` is not freed;

`free_workspace(ws)`: frees the vector `ws->work` and then the structure `ws`.

2.1.4 Parameters and parameter files

Many procedures need parameters, for example the maximal number of iterations for an iterative solver, the tolerance for the error in the adaptive procedure, etc. It is often very helpful to change the values of these parameters without recompiling the program by initializing them from a parameter file.

In order to avoid a fixed list of parameters, we use the following concept: Every parameter consists of two strings: a key string by which the parameter is identified, and a second string containing the parameter values. These values are stored as `ASCII`-characters and can be converted to `int`, `REAL`, etc. according to a format specified by the user (see below). Using this concept, parameters can be accessed at any point of the program.

Usually parameters are initialized from parameter files. Each line of the file describes either a single parameter: the key definition terminated by a `':'` character followed by the parameter values, or specifies another parameter file to be included at that point (this can also be done recursively). The syntax of these files is described below and an example is given at the end of this section.

Parameter files

The definition of a parameter has the following syntax:

```
key: parameter values % optional comment
```

Lines are only read up to the first occurrence of the comment sign `'%'`. All characters behind this sign in the same line are ignored. The comment sign may be a character of the specified filename in an include statement (see below). In this case, `'%'` is treated as a usual character.

The definition of a new parameter consists out of a key string and a string containing the parameter values. The definition of the key for a new parameter has to be placed in one line before the first comment sign. For the parameter values a continuation line can be used (see below). The key string is a sequence of arbitrary characters except `':'` and the comment character. It is terminated by `':'`, which does not belong to the key string. A key may contain blanks. Optional white space characters as blanks, tabs, etc. in front of a key and in between `':'` and the first character of the parameter value are discarded.

Each parameter definition must have at least one parameter value, but it can have more than one. If there are no parameter values specified, i.e. the rest of the line (and all continuation lines) contain(s) only white-space characters (and the continuation character(s)). Such a parameter definition is ignored and the line(s) is (are) skipped.

One parameter value is a sequence of non white-space characters. We will call such a sequence of non white-space characters a word. Two parameter values are separated by at least one white-space character. A string as a parameter value may not contain white-space characters. Strings enclosed in single or double quotes are not supported at the moment. These quotes are treated as usual characters.

Parameter values are stored as a sequence of words in one string. The words are separated by exactly one blank, although parameter values in the parameter file may be separated by more than one white-space character.

The key definition must be placed in one line. Parameter values can also be specified in so called continuation lines. A line is a continuation line if the last two characters in the preceding line are a `'\'` followed directly by the newline character. The `'\'` and the newline character are removed and the line is continued at the beginning of the next line. No additional blank is inserted.

Lines containing only white-space characters (if they are not continuation lines!) are skipped.

Besides a parameter definition we can include another parameter file with name `filename`:

```
#include "filename"
```

The effect of an include statement is the same as an include statement in a `C`-program. Using the function `init_parameters()` (see below) for reading the parameter file, the named file is not inserted virtually into the actual parameter file like the `C`-preprocessor does, but it is read by a recursive call of the function which reads the parameters. Thus, the included parameter file may also contain an include statement. The rest of line behind the closing `"` is skipped. Initialization then is continued from the next line on. An include statement must not have a continuation line.

Using the function `init_parameters_cpp()`, the file is included using the `C`-preprocessor (see below).

If a parameter file can not be opened for reading, an error message is produced and the reading of the file is skipped.

Errors occur and are reported if a key definition is not terminated in the same line by `':'`, no parameter values are specified, filename for include files are not specified correctly in between `" "`. The corresponding lines are ignored. No parameter is defined, or no file is included.

A parameter can be defined more than once but only the latest definition is valid. All previous definitions are ignored.

Reading of parameter files

Initializing parameters from such files is done by

```
void init_parameters(int print, const char *filename);
void init_parameters_cpp(int print, const char *filename, const char *cpp_flags);
```

Description:

`init_parameters(print, filename)`: initializes parameters from a file; `filename` is a string holding the name of the file and if values of the argument `print` and the global variable `msg_info` are not zero, a list of all defined parameters is printed to the message stream; if `filename` is a `nil` pointer, parameters are read from the file `"albert.dat"` located in the current directory; if `init_parameters()` can not open the input file, no parameter is defined.

One call of this function should be the first executable statement in the main program. Several calls of `init_parameters()` are possible. If a key is defined more than once, parameter values from the latest definition are valid. Parameter values from previous definition(s) are ignored.

`init_parameters_cpp(print, filename, cpp_flags)`: the behavior of this function is the same as of `init_parameters()` but the file is first processed by the C–preprocessor with argument `cpp_flags`, replacing macros by their definitions in the parameter file and including files specified by `#include"..."`.

```
init_parameters_cpp(0, "albert.dat", "-DNEIGH_IN_EL=0");
```

The C–preprocessor will first replace each occurrence of `NEIGH_IN_EL` in `"albert.dat"` by `0`. Also preprocessor conditionals depending on such macro definitions are allowed in the parameter file. Default macro definitions to the preprocessor are

```
-DDIM=d -DDIM_OF_WORLD=dow
```

where `d` and `dow` are replaced by the actually used values, i.e `d` equals `2` for 2d and `3` for 3d simulations. After this preprocessing the the parameters are initialized.

This function is only available, if the ALBERT library was compiled with the `CPP` macro holding the command for the C–preprocessor.

Adding of parameters or changing of parameter values

Several calls of `init_parameters()` or `init_parameters_cpp()` are possible. This may add new parameters or change the value of an existing parameter since only the values from the latest definition are valid.

Parameters can also be defined or modified by the function or the macro

```
void add_parameter(int flag, const char *key, const char *value);
ADD_PARAMETER(int flag, const char *key, const char *value)
```

Description:

`add_parameter(flag, key, value)`: initializes a parameter identified by `key` with values `value`; if the parameter already exists, the old values are replaced by the new one; if `flag` is not zero information about the initialization is printed; This message can be suppressed by a global level of parameter information (see the parameter `parameter information` in Section 2.1.5);

`ADD_PARAMETER(flag, key, value)`: acts as `add_parameter(flag, key, value)` but the function is additionally supplied with the name of the calling function, source file and line, which results in more detailed messages during parameter definition.

Saving of parameters and parameter values

For long time simulations it is important to write all parameters and their values to file; using this file the simulation can be re–done with exactly the same parameter values although the original parameter file was changed. Thus, after the initialization of parameters in a long time simulation, they should be written to a file by the following function:

```
void save_parameters(const char *file, int info);
```

Description:

`save_parameters(file, info)`: writes all up to now initialized parameters to `file` according to the above described parameter file format; if the value of `info` is different from zero, the location of the initialization is supplied for each parameter as a comment; no original comment is written, since these are not stored.

Getting parameter values

After initializing parameters by `init_parameters()` or `init_parameters_cpp()` we can access the values of a parameter by a call of

```
int get_parameter(int flag, char *key, char *format, ...);
int GET_PARAMETER(int flag, char *key, char *format, ...)
```

Description:

`get_parameter(flag, key, format, ...)`: looks for a parameter which matches the identifying key string `key` and converts the values of the corresponding string containing the parameter values according to the control string `format`. Pointers to variable(s) of suitable types are placed in the unnamed argument list (compare the syntax of `scanf()`). The first argument `flag` defines the level of information during the initialization of parameters with a range of `0` to `4`: no to full information. The return value is the number of successfully matched and assigned input items.

If there is no parameter key matching `key`, `get_parameter()` returns without an initialization. The return value is zero. It will also return without an initialization and return value zero if no parameter has been defined by `init_parameters()`.

In the case that a parameter matching the key is found, `get_parameter()` acts like a simplified version of `sscanf()`. The input string is the string containing the parameter values. The function reads characters from this string, interprets them according to a format, and stores the results in its arguments. It expects, as arguments, a control string, `format` (described below) and a set of pointer arguments indicating where the converted input should be stored. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are simply ignored. The return value is the number of converted arguments.

The control string must only contain the following characters used as conversion specification: `%s`, `%c`, `%d`, `%e`, `%f`, `%g`, `%U`, `%S`, or `%*`. All other characters are ignored. In contrast to `scanf()`, a numerical value for a field width is not allowed. For each element of the control string the next word of the parameter string is converted as follows:

`%s`: a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating `'\0'`, which will be added automatically; the string is one single word of the parameter string;

as mentioned above strings enclosed in single or double quotes are not supported at the moment;

`%c`: matches a single character; the corresponding argument should be a pointer to a `char` variable; if the corresponding word of the parameter string consists of more than one character, the rest of the word is ignored; no space character is possible as argument;

`%d`: matches an decimal integer, whose format is the same as expected for the subject sequence of the `atoi()` function; the corresponding argument should be a pointer to an `int` variable;

`%e,%f,%g`: matches an optionally signed floating point number, whose format is the same as expected for the subject string of the `atof()` function; the corresponding argument should be a pointer to a `REAL` variable;

`%U`: matches an unsigned decimal integer in the range [0,255], whose format is the same as expected for the subject sequence of the `atoi()` function; the corresponding argument should be a pointer to an `U_CHAR` variable;

`%S`: matches an optionally signed decimal integer in the range [-128,127], whose format is the same as expected for the subject sequence of the `atoi()` function; the corresponding argument should be a pointer to an `S_CHAR` variable;

`%*`: next word of parameter string should be skipped; there must not be a corresponding argument.

`get_parameter()` will always finish its work, successfully or not. It may fail if a misspelled key is handed over or there are not so many parameter values as format specifiers (the remaining variables are not initialized!). If `flag` is zero, `get_parameter()` works silently; no error message is produced. Otherwise the key and the initialized values and error messages are printed. The second way to influence messages produced by `get_parameter()` namely by is a parameter `parameter information` specified in a parameter file, see Section 2.1.5.

`GET_PARAMETER(flag, key, format, ...)`: is a macro and acts in the same way as the function `get_parameter(flag, key, format, ...)` but the function is additionally supplied with the name of the calling function, source file and line, which results in more detailed messages during parameter definition.

In order to prevent the program from working with uninitialized variables, all parameters should be initialized before! By the return value the number of converted arguments can be checked.

2.1.6. Example (`init_parameters()`, `GET_PARAMETER()`). Consider the following parameter file `init.dat`:

```
adapt info:  3             % level of information of the adaptive method
adapt tolerance:  0.001    % tolerance for the error
```

Then

```
init_parameters(0, "init.dat");
...
tolerance = 0.1;
GET_PARAMETER(0, "adapt tolerance", "%e", &tolerance);
```

initializes the `REAL` variable `tolerance` with the value `0.001`.

2.1.5 Parameters used by the utilities

The utility tools use the following parameters initialized with default values given in `()`:

`level of information (10)`: the global level of information; can restrict the local level of information (compare Section 2.1.2);

`parameter information (1)`: enforces more/less information than specified by the argument `flag` of the routine `get_parameter(flag, ...)`:

- `0`: no message at all is produced, although the value `flag` may be one;
- `1`: gives only messages if the value of `flag` is non zero;
- `2`: all error messages are printed, although the value of `flag` may be zero;
- `4`: all messages are printed, although the value of `flag` may be zero.

`WAIT (1)`: sets the value of the global variable `msg_wait` and changes by that the behaviour of the macro `WAIT` (see Section 2.1.2).

2.2 Data structures for the hierarchical mesh

2.2.1 Constants describing the dimension of the mesh

The symbolic constant `DIM` defines the dimension of the triangulation:

```
#define DIM  2
```

for two dimensional grids and

```
#define DIM  3
```

for three dimensional ones. Most parts of the finite element code do not depend on the particular choice of `DIM`. But especially the refinement/coarsening and hierarchical information processing routines strongly depend on `DIM`.

For finite element methods on two or three dimensional surfaces embedded in $\mathbb{R}^n$ (like mean curvature flow [18]) the vertex coordinates of the simplices have n components. Thus, we need another symbolic constant `DIM_OF_WORLD` to define the number of these components. For applications in domains in $\mathbb{R}^{\mathtt{DIM}}$, `DIM_OF_WORLD` equals `DIM` otherwise the value of `DIM_OF_WORLD` is the dimension of the $\mathbb{R}^n$ where the `DIM`-dimensional surface is embedded.

As a convention all `typedef` definitions with the ending `_D` define a new data type for vectors of length `DIM_OF_WORLD`, definitions with the ending `_DD` a matrix of size `DIM_OF_WORLD` $\times$ `DIM_OF_WORLD`, for example

```
typedef REAL          REAL_D[DIM_OF_WORLD];
typedef REAL          REAL_DD[DIM_OF_WORLD][DIM_OF_WORLD];
```

Several macros are defined for `REAL_D` vectors:

```
void SET_VAL_DOW(REAL a, REAL_D x)
REAL SCP_DOW(REAL_D x, REAL_D y)
REAL DIST_DOW(REAL_D x, REAL_D y)
REAL NORM_DOW(REAL_D x)
```

Description:

`SET_VAL_DOW(a, x)`: set all elements of vector `x` to `a`;

`SCP_DOW(x, y)`: returns the Euclidean scalar product of the two vectors `x`, `y`;

`DIST_DOW(x, y)`: returns the Euclidean distance of the two vectors `x`, `y`;
`NORM_DOW(x)`: returns the Euclidean norm of the vector `x`.

2.2.2 Constants describing the elements of the mesh

For a simplex we have to define the number of vertices, edges, adjacent simplices, and in three dimensions the number of boundary faces:

```
#define N_VERTICES      3
#define N_EDGES         3
#define N_NEIGH         3
```

for two dimensions and

```
#define N_VERTICES      4
#define N_EDGES         6
#define N_FACES         4
#define N_NEIGH         4
```

for three dimensions.

2.2.3 Neighbour information

As mentioned in Section 1.2 neighbour information is usually not stored explicitly, but such an explicit storing may speed up special applications. ALBERT supports both kinds of managing neighbour information. Via the symbolic constant `NEIGH_IN_EL` the user can enforce to store neighbour information explicitly in the elements. Setting

```
#define NEIGH_IN_EL   0
```

will result in calculating neighbour information (if requested) by the traversal routines, while

```
#define NEIGH_IN_EL   1
```

enforces that neighbour information is stored explicitly on each element and has to be set by the refinement/coarsening routines. This consumes additional computer memory, but will speed up the traversal routines that usually have to compute neighbour information. Macros

```
NEIGH(el, el_info),   OPP_VERTEX(el, el_info),   EL_TYPE(el,el_info)
```

are defined to access information independently of the value of `NEIGH_IN_EL`, compare Section 2.2.10.

2.2.4 Element indices

It is often very helpfully — especially during program development — that every element has a unique global index. This requires an entry in the element data structure which adds to the needed computer memory.

On the other hand this additional amount of computer memory may be a disadvantage in real applications where a big number of elements is needed, and — after program development — element index information is no longer of interest.

After setting the value of the symbolic constant `EL_INDEX` to be `1` an element index is available and adds to the amount of computer memory, and setting it to be `0` no element index information is available which results in less computer memory usage. The macro

```
INDEX(el)
```

is defined to access element indices independently of the value of `EL_INDEX`. If no indices are available, the macro returns `-1`, compare Section 2.2.11.

2.2.5 The BOUNDARY data structure

Degrees of freedom (DOFs) which describe the finite element space are located at a node which is either a vertex, an edge, (a face,) or the barycenter of an element (see Section 1.3). For each node we have to specify if (Dirichlet) boundary conditions are prescribed for DOFs at this node or if the DOFs are true degrees of freedom. At the moment we support three kinds of nodes:

- interior node where DOFs are handled as real degrees of freedom;
- nodes on the Dirichlet boundary where Dirichlet boundary conditions are prescribed;
- nodes on the Neumann boundary which are treated like interior nodes.

The boundary type is specified by a `S_CHAR`. We use the following symbolic constants and convention:

```
#define INTERIOR      0
#define DIRICHLET     1
#define NEUMANN      -1
```

An interior node has the boundary type `INTERIOR`, a node on a Dirichlet boundary has positive boundary type ($\geq$ `DIRICHLET`), and a node on the Neumann boundary has negative type ($\leq$ `NEUMANN`).

For the approximation of curved boundaries, the new nodes at the midpoint of the refinement edge are projected onto the curved boundary. Most likely, the boundary of the domain is split in several parts, where different functions for these projections are needed. For applying such a function to a node we have to know which part of the domain's boundary the refinement edge belongs.

In order to handle curved boundaries and to classify the type of nodes located at edges and faces we define the following structure:

```
typedef struct boundary   BOUNDARY;

struct boundary
{
  void           (*param_bound)(REAL_D  *coord);
  S_CHAR         bound;
};
```

The members yield following information:

`param_bound`: pointer to a function which projects a newly generated vertex onto a curved boundary; the input of this function are the world coordinates of the midpoint of the refinement edge; after the call, `coord` contains the coordinates of the projection to the curved boundary, compare Section 2.4.1, Figure 2.3; if `param_bound` is a pointer to `nil`, the edge or face does not belong to a curved boundary;

`bound`: boundary type of a node at the edge or face; the boundary type is uniquely defined by the sign of `bound`; by the value of `bound`, the part of the domain's boundary to which this object belongs can be stored (assuming, that these parts are enumerated).

There are some useful macros for the `BOUNDARY` structure defined in `albert.h`. They can be applied to any `BOUNDARY` pointer and give a default return value if the value of that pointer is `nil`.

```
#define GET_BOUND(boundary) ((boundary) ? (boundary)->bound : INTERIOR)
#define IS_INTERIOR(boundary)\
   ((boundary) ? (boundary)->bound == INTERIOR : true)
#define IS_DIRICHLET(boundary)\
   ((boundary) ? (boundary)->bound >= DIRICHLET : false)
#define IS_NEUMANN(boundary)\
   ((boundary) ? (boundary)->bound <= NEUMANN : false)
```

2.2.6 The local indexing on elements

For the handling of higher order discretizations where besides vertices DOFs can be located at edges, faces, or center, we also need a local numbering for edges, and faces. We also need a local numbering of neighbours for handling neighbour information which is used in the refinement algorithm itself and for error estimator calculation, e.g.

In 2d the `i`-th edge/neighbour is the edge/neighbour opposite the `i`-th vertex; in 3d the `i`-th face/neighbour is the face/neighbour opposite the `i`-th vertex; edges in 3d are numbered in the following way (compare Figure 2.1):

edge 0: between vertex `0` and `1`, **edge 3:** between vertex `1` and `2`,
edge 1: between vertex `0` and `2`, **edge 4:** between vertex `1` and `3`,
edge 2: between vertex `0` and `3`, **edge 5:** between vertex `2` and `3`.

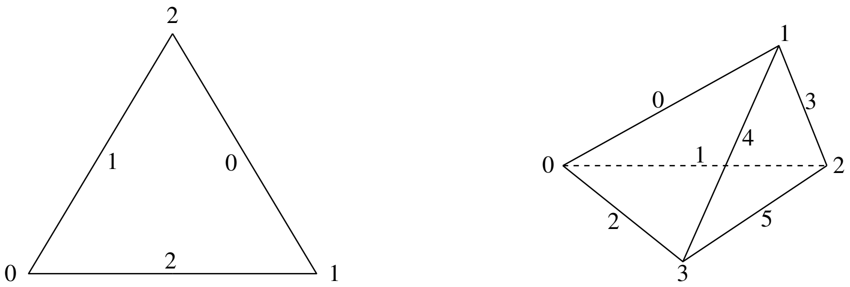


Figure 2.1: Local indices of edges/neighbours in 2d and local indices of edges in 3d.

2.2.7 The `MACRO_EL` data structure

In this section, we describe the macro triangulation and data type for an element of the macro triangulation.

The macro triangulation is stored in a doubly linked list of macro elements:

```
typedef struct macro_el   MACRO_EL;

struct macro_el
{
  MACRO_EL        *next, *last;
  EL              *el;
  REAL            *coord[N_VERTICES];
  int             index;

  MACRO_EL        *neigh[N_NEIGH];
  U_CHAR          opp_vertex[N_NEIGH];
```

```
  S_CHAR          bound[N_VERTICES];
#if DIM == 2
  const BOUNDARY  *boundary[N_EDGES];
#else
  const BOUNDARY  *boundary[N_FACES+N_EDGES];
  U_CHAR          el_type;
#endif
};
```

The members yield following information (alternatives for two and three dimensions are separated by '/'):

next: pointer to next macro element; if next is nil then the macro element is the last macro element;

last: pointer to previous macro element; if last is nil the macro element is the first macro element;

el: root of the binary tree located at this macro element;

coord: pointer to the world coordinates of the element's vertices;

index: a unique index of all macro elements;

neigh: neigh[i] points to the macro element at the edge/face with local index i; it is a pointer to nil for boundary edges/faces;

opp_vertex: opp_vertex[i] is undefined if neigh[i] is a pointer to nil; otherwise it is the local index of the neighbour's vertex opposite the common edge/face.

bound: bound[i] boundary type of vertex i;

boundary: boundary[i] is a pointer to a boundary structure of the i-th edge/face for $0 \leq$ i $<$ N_NEIGH; in 3d boundary[N_FACES+i] is a pointer to a boundary structure of the i-th edge for $0 \leq$ i $<$ N_EDGES; for an interior edge/face, it is a pointer to nil;

el_type: type of the element $\in [0, 1, 2]$ used for refinement and coarsening (for the definition of the element type see Section 1.1.1, only 3d).

2.2.8 The EL data structure

Now we describe the elements of the binary trees and information that should be present for tree elements:

```
typedef struct el    EL;

struct el
{
  EL              *child[2];
  DOF             **dof;

  S_CHAR          mark;
  REAL            *new_coord;

#if EL_INDEX
  int             index;
#endif

#if NEIGH_IN_EL
  EL              *neigh[N_NEIGH];
```

```
  U_CHAR          opp_vertex[N_NEIGH];
#if DIM == 3
  U_CHAR          el_type;
#endif
#endif
};
```

The members yield following information:

`child`: pointers to the two children for interior elements of the tree; `child[0]` is a pointer to `nil` for leaf elements; `child[1]` is a pointer to user data on leaf elements if `leaf_data_size` is bigger than zero, otherwise `child[1]` is also a pointer to `nil` for leaf elements (see Section 2.2.12);

`dof`: vector of pointers to DOFs; these pointers must be available for the elements vertices (for the geometric description of the mesh); there may be pointers at the midpoints of the edges, at the barycenter and in three dimensions at the midpoints of the faces; they are ordered in the following way: the first `N_VERTICES` entries correspond to the DOFs at the vertices; the next one are those at the edges, if present, then those at the faces, if present (only for 3d), and then those at the barycenter; the offsets are defined in the `MESH` structure (see Sections 2.2.14, 2.4.1, 2.4.2);

`mark`: marker for refinement and coarsening: if `mark` is positive for a leaf element this element is refined `mark` times; if it is negative for a leaf element the element may be coarsened `-mark` times; (see Sections 2.4.1, 2.4.2);

`new_coord`: if the element has a boundary edge on a curved boundary this is a pointer to the coordinates of the new vertex that is created due to the refinement of the element, otherwise it is a `nil` pointer; thus, coordinate information can also be produced by the traversal routines in the case of a curved boundary;

`index`: unique global index of the element; these indices are not strictly ordered and may be larger than the number of elements in the binary tree (the list of indices may have holes after coarsening); the index is available only if `EL_INDEX` is `true`.

If neighbour information should be present in the element structure (if `NEIGH_IN_EL` is `true`) then we have the additional entries:

`neigh`: `neigh[i]` pointer to the element across the edge/face with local index `i`; it is a pointer to `nil` for boundary edges/faces;

`opp_vertex`: `opp_vertex[i]` is undefined if `neigh[i]` is a pointer to nil; otherwise it is the local index of the neighbour's vertex opposite the common edge/face;

`el_type`: the element's type (see Section 2.4.1); has to be available on the element if neighbour information is located at the element, since such information then can not be produced by the traversal routines going from one element to another by the neighbour pointers.

2.2.9 The `EL_INFO` data structure

The `EL_INFO` data structure has entries for all information which is not stored on the element itself, but may be generated. It is filled by the mesh traversal routines, most entries only if specially requested (see Section 2.2.18).

```
typedef struct el_info   EL_INFO;

struct el_info
{
  MESH            *mesh;
  REAL_D          coord[N_VERTICES];
  EL              *el;
  EL              *parent;
  const MACRO_EL  *macro_el;

  FLAGS           fill_flag;
  U_CHAR          level;

#if ! NEIGH_IN_EL
  EL              *neigh[N_NEIGH];
  U_CHAR          opp_vertex[N_NEIGH];

#if DIM == 3
  U_CHAR          el_type;
#endif
#endif

  REAL_D          opp_coord[N_NEIGH];

  S_CHAR          bound[N_VERTICES];
#if DIM == 2
  const BOUNDARY  *boundary[N_EDGES];
#else
  const BOUNDARY  *boundary[N_FACES+N_EDGES];
  S_CHAR          orientation;
#endif
};
```

The members yield following information:

`mesh`: a pointer to the current mesh;

`coord`: `coord[i]` is a `DIM_OF_WORLD` vector storing world coordinates of the `i`-th vertex;

`el`: pointer to the current element;

`parent`: `el` is child of element `parent`;

`macro_el`: `el` is an element of the binary tree located at macro element `macro_el`;

`fill_flag`: a flag which indicates which elements are called and which information should be present (see Section 2.2.18);

`level`: level of the element; the level is zero for macro elements and the level of the children is (level of the parent + 1); the level is filled always by the traversal routines;

`opp_coord`: `opp_coord[i]` coordinates of the `i-th` neighbour's vertex opposite the common edge/face;

`bound`: `bound[i]` boundary type of vertex `i`;

`boundary`: `boundary[i]` is a pointer to a boundary structure of the `i`-th edge/face for `i` $=0,\dots$ `N_NEIGH` -1; in 3d `boundary[N_FACES+i]` is a pointer to a boundary structure of the `i`-th edge for `i` $=0,\dots$ `N_EDGES` -1; it is a pointer to `nil` for an interior edge/face;

orientation: ± 1: sign of the determinant of the transformation to the reference element with vertices $(0,0,0)$, $(1,1,1)$, $(1,1,0)$, $(1,0,0)$ (see Figure 1.7).

If neighbour information is not present in the element structure (`NEIGH_IN_EL` is `false`), then we have the additional entries:

neigh: `neigh[i]` pointer to the elements at the edge/face with local index `i`; it is a pointer to `nil` for boundary edges/faces;

opp_vertex: `opp_vertex[i]` is undefined if `neigh[i]` is a pointer to `nil`; otherwise it is the local index of the neighbour's vertex opposite the common edge/face;

el_type: the element's type (see Section 2.4.1); is filled automatically by the traversal routines (only 3d).

2.2.10 The NEIGH, OPP_VERTEX and EL_TYPE macros

If neighbour information is stored explicitly at each element (`NEIGH_IN_EL == 1`) we get a pointer to the `i`-th neighbour of an element `el` and the corresponding `opp_vertex` by

```
neigh = el->neigh[i];
opp_v = el->opp_vertex[i];
```

If such information is produced by the traversal routines we have to access neighbour information by the corresponding `el_info` structure:

```
neigh = el_info->neigh[i];
opp_v = el_info->opp_vertex[i];
```

To have same access for both situations there are two macros `NEIGH` and `OPP_VERTEX` defined in `albert.h`:

```
#if NEIGH_IN_EL
#define NEIGH(el,el_info) el->neigh
#define OPP_VERTEX(el,el_info) el->opp_vertex
#else
#define NEIGH(el,el_info) el_info->neigh
#define OPP_VERTEX(el,el_info) el_info->opp_vertex
#endif
```

Similarly, the element type (only in 3d) is stored either in the `EL` structure or is generated during traversal in `EL_INFO`. A macro `EL_TYPE` is defined to access such information:

```
#if NEIGH_IN_EL
#define EL_TYPE(el,el_info) el->el_type
#else
#define EL_TYPE(el,el_info) el_info->el_type
#endif
```

Using these macros we always get information about the `i`-th neighbour or the element type by

```
neigh = NEIGH(el,el_info)[i];
opp_v = OPP_VERTEX(el,el_info)[i];
type  = EL_TYPE(el,el_info);
```

independently of the value of `NEIGH_IN_EL`.

2.2.11 The INDEX macro

In order to avoid to eliminate all lines where the element index is accessed after recompiling the source with `EL_INDEX == 0`, a macro `INDEX` is defined:

```
#if EL_INDEX
#define INDEX(el)   ((el) ? (el)->index : -1)
#else
#define INDEX(el)   -1
#endif
```

If element indices are stored at the elements `INDEX(el)` gives the index of the element, if the `el` is not a pointer to `nil`. If `el` is a pointer to `nil`, the value of `INDEX(el)` is `-1`. This allows a construction like `INDEX(NEIGH(el,el_info)[i])` without testing whether this neighbour exists or not (in the case of a boundary edge/face). If element indices are not stored, the value of `INDEX(el)` is always `-1`.

2.2.12 The LEAF_DATA_INFO data structure

As described in Section 1.2, we can "hide" user data on leaf elements at the pointer to the second child. In order to handle an arbitrary kind of user data on leaf elements we define a data structure `LEAF_DATA_INFO` which gives information about size of user data and may give access to functions transforming user data during refinement and coarsening. A pointer to such a structure is an entry in the `MESH` data structure.

```
typedef struct leaf_data_info   LEAF_DATA_INFO;

struct leaf_data_info
{
  char            *name;
  unsigned        leaf_data_size;
  void            (*refine_leaf_data)(EL *parent, EL *child[2]);
  void            (*coarsen_leaf_data)(EL *parent, EL *child[2]);

  const void      *leaf_data_block_info; /* only used by ALBERT!!!       */
};
```

Following information is provided via this structure:

`name`: textual description of leaf data

`leaf_data_size`: size of memory area for storing leaf data; if `leaf_data_size == 0` no memory for leaf data is allocated and `child[1]` is also pointer to `nil` for leaf elements; if `leaf_data_size > 0` then the value of `leaf_data_size` should be bigger than `sizeof(void *)` since ALBERT allocates memory for leaf data block wise and builds a linked list of single leaf data elements; if `leaf_data_size > 0` and `leaf_data_size < sizeof(void *)` ALBERT changes this value to `sizeof(void *)`;

`refine_leaf_data`: pointer to a function which transforms leaf data from parent to child during refinement; after such a transformation leaf data of the parent is destroyed; if `refine_leaf_data` is a pointer to `nil` no transformation takes place, parent information is lost; transformation only takes place if `leaf_data_size > 0`;

`coarse_leaf_data`: pointer to a function which transforms leaf data from children to parent during coarsening; after such a transformation leaf data of the children is destroyed; if coarse_leaf_data is a pointer to nil no transformation takes place, children information is lost; transformation only takes place if leaf_data_size > 0;

leaf_data_block_info: pointer used by ALBERT for a block wise memory allocation and deallocation of leaf data; must not be changed in an application.

Following macros for testing leaf elements and accessing leaf data are provided:

```
#define IS_LEAF_EL(el) (!(el)->child[0])
#define LEAF_DATA(el)  ((void *)(el)->child[1])
```

2.2.13 The RC_LIST_EL data structure

For refining and coarsening we need information of the elements at the refinement and coarsening edge (compare Sections 1.1.1 and 1.1.2). Thus, we have to collect all elements at this edge. In 2d we have at most the current element and its neighbour across this edge, if the edge is not part of the boundary. In 3d we have to loop around this edge to collect all the elements. Every element at the edge has at most two neighbours sharing the same edge. Defining an orientation for this edge, we can define the *right* and *left* neighbour.

For every element at the refinement/coarsening edge we have an entry in a vector. The elements of this vector build the refinement/coarsening patch. In 2d the vector has length 2 and in 3d length mesh->max_no_edge_neigh since this is the maximal number of elements sharing the same edge in the mesh mesh.

```
typedef struct rc_list_el  RC_LIST_EL;

struct rc_list_el
{
  EL              *el;
  int             no;
  int             flag;
#if DIM == 3
  RC_LIST_EL      *neigh[2];
  int             opp_vertex[2];
  U_CHAR          el_type;
#endif
};
```

Information that is provided for every element in this RC_LIST_EL vector:

el: pointer to the element itself;

no: this is the no–th entry in the vector;

flag: only used in the coarsening module: flag is true if the coarsening edge of the element is the coarsening edge of the patch, otherwise flag is false;

neigh: neigh[0/1] neighbour of element to the right/left in the orientation of the edge, or a nil pointer in the case of a boundary face (only 3d);

opp_vertex: opp_vertex[0/1] the opposite vertex of neigh[0/1] (only 3d);

el_type: the element type; is set during looping around the refinement/coarsening edge; if neighbour information is produced by the traversal routines, information about the type of an element can not be accessed via el->el_type and thus has to be stored in the RC_LIST_EL vector (only 3d).

This RC_LIST_EL vector is one argument to the interpolation and restriction routines for DOF vectors (see Section 2.3.3).

2.2.14 The MESH data structure

All information about a triangulation is available via the MESH data structure:

```
typedef struct mesh  MESH;

struct mesh
{
  const char      *name;
  int             n_vertices;
  int             n_edges;
#if DIM == 3
  int             n_faces;
#endif
  int             n_elements;
  int             n_hier_elements;

#if DIM == 3
  int             max_edge_neigh;
#endif

  int             n_macro_el;
  MACRO_EL        *first_macro_el;
  REAL            diam[DIM_OF_WORLD];

  PARAMETRIC      *parametric;

  LEAF_DATA_INFO  leaf_data_info[1];
  U_CHAR          preserve_coarse_dofs;

  DOF_ADMIN       **dof_admin;
  int             n_dof_admin;
  int             n_node_el;
  int             n_dof_el;
  int             n_dof[DIM+1];
  int             node[DIM+1];
};
```

The members yield following information:

name: string with a textual description for the mesh, or **nil**;

n_vertices: number of vertices of the mesh;

n_edges: number of edges of the mesh;

n_faces: number of faces of the mesh (only 3d);

n_elements: number of leaf elements of the mesh;

n_hier_elements: number of all elements of the mesh;

max_edge_neigh: maximal number of elements that share one edge; used to allocate memory to store pointers to the neighbour at the refinement/coarsening edge (only 3d);

n_macro_el: number of macro elements;

first_macro_el: pointer to the first macro element;

diam: diameter of the mesh in the DIM_OF_WORLD directions;

parametric: is pointer to nil if mesh contains no parametric elements else pointer to a PARAMETRIC structure containing coefficients of the parameterization and related information;

leaf_data_info: a structure with information about data on the leaf elements;

preserve_coarse_dofs: if the value is non zero then preserve all DOFs on all levels (can be used for multigrid, e.g.); otherwise all DOFs on the parent that are not handed over to a child are removed during refinement and added again on the parent during coarsening.

The last entries are used for the administration of DOFs and are explained in Section 2.3 in detail.

dof_admin: vector of dof_admins;

n_dof_admin: number of dof_admins;

n_node_el: number of nodes on a single element where DOFs are located; needed for the (de-) allocation of the dof-vector on the element;

n_dof_el: number of all DOFs on a single element;

n_dof: number of DOFs at the different positions VERTEX, EDGE, (FACE,) CENTER on an element:

- **n_dof[VERTEX]:** number of DOFs at a vertex (≥ 1);
- **n_dof[EDGE]:** number of DOFs at an edge; if no DOFs are associated to edges, then this value is 0;
- **n_dof[FACE]:** number of DOFs at a face; if no DOFs are associated to faces, then this value is 0 (only 3d);
- **n_dof[CENTER]:** number of DOFs at the barycenter; if no DOFs are associated to the barycenter, then this value is 0;

node: gives the index of the first node at vertex, edge, face (only 3d), and barycenter:

- **node[VERTEX]:** has always value 0; dof[0],...,dof[N_VERTICES-1] are always DOFs at the vertices;
- **node[EDGE]:** dof[node[EDGE]],..., dof[node[EDGE]+N_EDGES-1] are the DOFs at the N_EDGES edges, if DOFs are located at edges;
- **node[FACE]:** dof[node[FACE]],..., dof[node[FACE]+N_FACES-1] are the DOFs at the N_FACES faces, if DOFs are located at faces (only 3d);
- **node[CENTER]:** dof[node[CENTER]] are the DOFs at the barycenter, if DOFs are located at the barycenter;

2.2.15 Initialization of meshes

It is possible to handle more than one mesh at the same time. A mesh must be accessed by one of the following functions or macro

```
MESH *get_mesh(const char *name, void (*init_dof_admins)(MESH *),
               void (*init_leaf_data)(LEAF_DATA_INFO *));
MESH *check_and_get_mesh(int dim, int dow, int neigh, int index,
                         const char *version, const char *name,
                         void (*init_dof_admins)(MESH *),
                         void (*init_leaf_data)(LEAF_DATA_INFO *));
MESH *GET_MESH(const char *name, void (*init_dof_admins)(MESH *),
               void (*init_leaf_data)(LEAF_DATA_INFO *))
```

Description:

get_mesh(name, init_dof_admin, init_leaf_data): returns a pointer to a filled mesh structure; `name` is a string holding a textual description of mesh and is duplicated at the member `name` of the mesh;

`init_dof_admin` is a pointer to a user defined function for the initialization of the required DOFs on the mesh (see Section 2.6.2 for an example of such a function); if this argument is a `nil` pointer a warning is given and a `DOF_ADMIN` structure for DOFs at the vertices of the mesh is generated. This is also done if none of the users `DOF_ADMIN`s uses DOFs at vertices (as mentioned above, DOFs at vertices have to be present for the geometric description of the mesh);

`init_leaf_data` is a pointer to a function for the initialization of the `leaf_data_info` structure, i.e. the size of leaf data can be set and pointers to transformation functions for leaf data can be adjusted. If this argument is a `nil` pointer, the member `leaf_data_info` structure is initialized with zero.

The return value of the function is a pointer to the filled `mesh` data structure. The user must not change any entry in this structure aside from the `max_edge_neigh`, `n_macro_el`, `first_macro_el`, `diam`, and `parametric` entries.

At the moment there exists no function to add further DOFs after the call of `get_mesh()`. Such a function would have to perform something like a so called p-refinement, which is not implemented yet.

There is no other possibility to define new meshes inside ALBERT.

check_and_get_mesh(d, dow, n, i, v, name, ida, ild): returns also a pointer to a filled mesh structure; the last three arguments are the same as of `get_mesh()`; in addition several checks about the used ALBERT library are performed: `d` is `DIM`, `dow` is `DIM_OF_WORLD`, `n` is `NEIGH_IN_EL`, `i` is `EL_INDEX`, and `v` is `VERSION` in the user program; these values are checked against the constants in the used library; if these values are identical, the mesh is accessed by `get_mesh()`; otherwise an error message is produced and the program stops.

GET_MESH(name, ida, ild): returns also pointer to a filled mesh structure; this macro calls `check_and_get_mesh()` and automatically supplies this function with the first five (missing) arguments; this macro should always be used for accessing a mesh.

After this initialization data for macro elements can be specified for example by reading it from file (see Section 2.2.16).

A mesh that is not needed any more can be freed by a call of the function

```
void free_mesh(MESH *mesh);
```

2.2.16 Reading macro triangulations

Data for macro triangulations can easily be stored in an `ASCII`–file. For the macro triangulation file we use a similar key–data format like the parameter initialization (see Section 2.1.4):

```
DIM:          dim
DIM_OF_WORLD: dow

number of vertices: nv
number of elements: ne

vertex coordinates:
```

```
DIM_OF_WORLD coordinates of vertex[0]
...
DIM_OF_WORLD coordinates of vertex[nv-1]

element vertices:
N_VERTICES indices of vertices of simplex[0]
...
N_VERTICES indices of vertices of simplex[ne-1]

element boundaries:
N_NEIGH boundary descriptions of simplex[0]
...
N_NEIGH boundary descriptions of simplex[ne-1]

element type:
element type of simplex[0]
...
element type of simplex[ne-1]
```

All lines closed by the `':'` character are keys for following data (more or less self explaining). Data for elements and vertices are read and stored in vectors for the macro triangulation. Index information given in the file correspond to this vector oriented storage of data. Thus, index information must be in the range `0,...,ne-1` for elements and `0,...,nv-1` for vertices. Although a vertex may be a common vertex of several macro elements the coordinates are only stored once.

An important point is that the refinement edges are determined by the local numbering of the vertices on each element! This is always the edge in between the vertices with local index `0` and `1`. In 2d the local ordering of vertex indices must be in counterclockwise sense for each triangle; in 3d by this numbering and by the element type the distribution of the refinement edges for the children is determined.

Information about the element type must only be given in 3d and such information is optional. Values must be in the range `0,1,2`. If such information is not given in the file, all elements are assumed to be of type `0`.

There are only some restrictions on the ordering of data in the file:

On the first two lines `DIM` and `DIM_OF_WORLD` have to be specified (in an arbitrary order); for the 2d version of ALBERT, `DIM` must equal `2` and `DIM_OF_WORLD` must be $\geq$`2`; for the 3d version of ALBERT, `DIM` must equal `3` and `DIM_OF_WORLD` must be $\geq$`3`. By these values it is checked whether data matches to the actual version of ALBERT.

The last two restrictions are that `number of elements` has to be specified before `element vertices`, `element boundaries`, and `element type` (only 3d, optional). The `number of vertices` has to specified before `vertex coordinates` and `element vertices`.

Besides these restrictions, ordering of data is left to the user.

Information about neighbours may be supplied in the file by

```
element neighbours:
N_NEIGH neighbour indices of simplex[0]
...
N_NEIGH neighbour indices of of simplex[ne-1]
```

where the index is `-1` for a non existing neighbour at a boundary edge/face. If information about neighbours is not present or not correctly specified in the file, it is, as the local indices of opposite vertices, generated automatically.

Reading data of the macro grid from these files can be done by

```
void read_macro(MESH *mesh, const char *name,
                const BOUNDARY *(*ibdry)(MESH *, int ));
```

Description:

`read_macro(mesh, name, ibdry)`: reads data of the macro triangulation for `mesh` from the `ASCII`–file `name`; adjustment of `BOUNDARY` structures can be done via the function pointer `ibdry`; the `mesh` structure must have been initialized by a call of `get_mesh()` or `check_and_get_mesh()`; this is important for setting the correct DOF pointers on the macro triangulation.

Using index information from the file, all information concerning element vertices, neighbour relations can be calculated directly.

In the macro file only information about boundary types of edges in 2d and faces in 3d is given. Zero values correspond to an interior edge/face, positive values to an edge/face on the Dirichlet boundary and negative values to an edge/face on the Neumann boundary. In the macro file only boundary information for edges/faces is given. Thus information for vertices in 2d, and for vertices and edges in 3d is computed by the routine; we use the conventions that the Dirichlet boundary is a closed subset of the boundary. This leads to the following assignments:

1. a vertex belongs to the Dirichlet boundary if it is a vertex of one edge/face belonging to the Dirichlet boundary;
2. a vertex belongs to the Neumann boundary if it is a vertex of an edge/face belonging to the Neumann boundary and it is not a vertex of any edge/face on the Dirichlet boundary;
3. all other vertices belong to the interior.

The same holds accordingly for edges in 3d.

For boundary edges/faces we additionally have to fill a boundary structure; this is done by the third argument `ibdry` which is a pointer to a user defined function; it returns a pointer to a filled boundary structure, compare Section 2.2.5. The function is called for each boundary edge/face

```
(*ibdry)(mesh, bound)
```

where `mesh` is the structure to be filled and `bound` the boundary value read from the file for this edge/face; using this value, the user can choose inside `ibdry` the correct function for projecting nodes that will be created in this edge/face; if `ibdry` is a `nil` pointer, it is assumed that the domain is polygonal, no projection has to be done; the corresponding boundary pointers are adjusted to two default structures

```
const BOUNDARY dirichlet_boundary = {nil, DIRICHLET};
const BOUNDARY neumann_boundary = {nil, NEUMANN};
```

for positive resp. negative values of `bound`.

In 3d we have to adjust a pointer to such a `BOUNDARY` structure also for edges. This is done by setting this pointer to the `BOUNDARY` structure of *one* of the meeting faces at that edge; if a `BOUNDARY` structure of a face supplies a function for the projection of a node onto a curved boundary during refinement, this function must be able to project any point in the closure of that face (because it may be used for any edge of that face); if the boundary type of the edge is `DIRICHLET`, then at least one the boundary types of the meeting faces is `DIRICHLET` also, and we will use the `BOUNDARY` structure of a `DIRICHLET` face.

During the initialization of the macro triangulation, other entries like `n_edges`, `n_faces`, and `max_edge_neigh` in the mesh data structure are calculated.

2.2.1. Example (The standard triangulation of the unit square in $\mathbb{R}^2$).

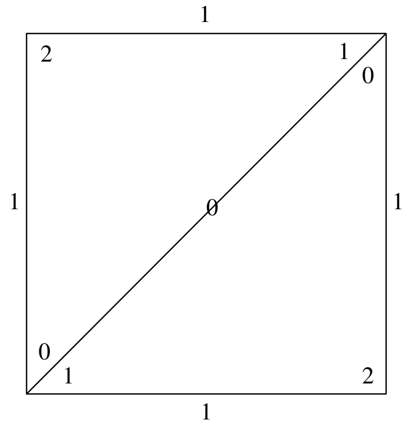
```
DIM: 2
DIM_OF_WORLD: 2

number of elements: 2
number of vertices: 4

element vertices:
2 0 1
0 2 3

element boundaries:
1 1 0
1 1 0

vertex coordinates:
 0.0 0.0
 1.0 0.0
 1.0 1.0
 0.0 1.0
```



Macro triangulation of the unit square.

2.2.2. Example (The standard triangulation of the unit cube in $\mathbb{R}^3$).

```
DIM:          3
DIM_OF_WORLD: 3

number of vertices: 8
vertex coordinates:
 -1.0 -1.0 -1.0
  1.0 -1.0 -1.0
 -1.0 -1.0  1.0
  1.0 -1.0  1.0
  1.0  1.0 -1.0
  1.0  1.0  1.0
 -1.0  1.0 -1.0
 -1.0  1.0  1.0

number of elements: 6

element vertices:
  0    5    4    1
  0    5    3    1
  0    5    3    2
  0    5    4    6
  0    5    7    6
  0    5    7    2

element boundaries:
  1    1    0    0
  1    1    0    0
  1    1    0    0
  1    1    0    0
  1    1    0    0
```

```
 1   1   0   0
```

2.2.3. Example (A triangulation of three quarters of the unit disc). Here, we have now to deal with a curved boundary. Thus, we have to provide a function for the projection during refinement in the boundary data structure. The actual projection is easy to implement, since we only have normalize the coordinates for nodes belonging to the curved boundary. A pointer to this function is accessible inside `read_mesh()` by the the `ibdry` function. Furthermore, we assume that the two straight edges belong to the Neumann boundary, and the curved boundary is the Dirichlet boundary.

The macro triangulation file:

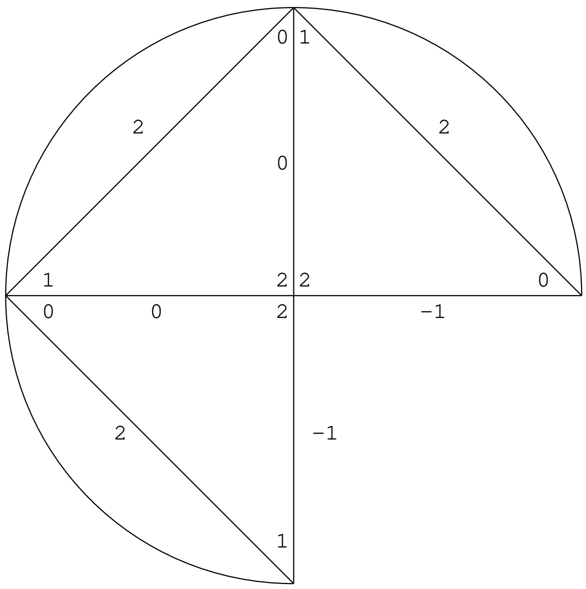
```
DIM: 2
DIM_OF_WORLD: 2

number of vertices: 5
number of elements: 3

vertex coordinates:
 0.0  0.0
 1.0  0.0
 0.0  1.0
-1.0  0.0
 0.0 -1.0

element vertices:
1 2 0
2 3 0
3 4 0

element boundaries:
 0 -1  2
 0  0  2
-1  0  2
```



Macro triangulation of a 3/4 disc.

and the functions `ball_proj()` for the projection and `ibdry()` for the initialization:

```
static void ball_project(REAL_D p)
{
  FUNCNAME("ball_project");
  REAL     norm;
  int      k;

  TEST_EXIT(p)("p = nil\n");

  norm = NORM_DOW(p);
  norm = 1.0/MAX(1.0E-10, norm);

  for (k = 0; k < DIM_OF_WORLD; k++)
    p[k] *= norm;

  return;
}

const BOUNDARY *ibdry(MESH *mesh, int bound)
```

```
{
  FUNCNAME("ibdry");
  static const BOUNDARY curved_dirichlet = {ball_project, DIRICHLET};
  static const BOUNDARY straight_neumann = {nil, NEUMANN};

  switch(bound)
  {
  case  2:  return(&curved_dirichlet);
  case -1:  return(&straight_neumann);
  default: ERROR_EXIT("no boundary %d\n", bound);
  }
}
```

2.2.17 Writing macro triangulations

The counterpart of the function `read_macro()` is the function

```
int write_macro(MESH *mesh, const char *filename);
```

Description:

`write_macro(mesh, name)`: writes all the leaf elements of `mesh` as a macro triangulation to the file specified by `name` in the above described format; if the file could be written, the return value is `1`, otherwise an error message is produced and the return value is `0`.

For writing meshes including the whole hierarchy, see Section 2.3.8

2.2.18 Mesh traversal routines

As described before, the mesh is organized in a binary tree, and most local information is not stored at leaf element level, but is generated from hierarchical information and macro element data. The generation of such local information is done during tree traversal routines.

When some work has to be done at each tree element or leaf element, such a tree traversal is most easily done in a recursive way, calling some special subroutine at each (leaf) element which implements the operation that currently has to be done. For some other applications, it is necessary to operate on the (leaf) elements in another fashion, where a recursive traversal is not possible. To provide access for both situations, there exist recursive and non-recursive mesh traversal routines.

For both styles, selection criteria are available to indicate on which elements the operation should take place. Following constants are defined:

```
CALL_EVERY_EL_PREORDER
CALL_EVERY_EL_INORDER
CALL_EVERY_EL_POSTORDER
CALL_LEAF_EL
CALL_LEAF_EL_LEVEL
CALL_EL_LEVEL
CALL_MG_LEVEL
```

`CALL_EVERY_EL_PREORDER`, `CALL_EVERY_EL_INORDER`, and `CALL_EVERY_EL_POSTORDER` all three operate on *all* hierarchical elements of the mesh. These three differ in the sequence of operation on elements: `..._PREORDER` operates first on a parent element before operating on both children, `..._POSTORDER` operates first on both children before operating on their parent, and `..._INORDER` first operates on `child[0]`, then on the parent element, and last on `child[1]`.

CALL_LEAF_EL operates on *all* leaf elements of the tree, whereas CALL_LEAF_EL_LEVEL operates only on leaf elements which are exactly at a specified tree depth. CALL_EL_LEVEL operates on all tree elements at a specified tree depth. The option CALL_MG_LEVEL is special for multigrid operations. It provides the operation on all hierarchy elements on a specified multigrid level (which is usually `el->level / DIM`).

Additional flags are defined that specify which local information in EL_INFO has to be generated during the hierarchical mesh traversal. A bitwise OR of some of these constants is given as a parameter to the traversal routines. These flags are more or less self explaining:

`FILL_NOTHING:` no information needed at all;

`FILL_COORDS:` the vertex coordinates `EL_INFO.coord` are filled;

`FILL_BOUND:` the boundary descriptions `EL_INFO.bound`, `EL_INFO.boundary`, and (in 3d) `EL_INFO.edge_boundary` are filled;

`FILL_NEIGH:` neighbour element information `EL_INFO.neigh` and `EL_INFO.opp_vertex` is generated (when not NEIGH_IN_EL);

`FILL_OPP_COORDS:` neighbour element information and the opposite vertex coordinates `EL_INFO.opp_coords` are filled;

`FILL_ORIENTATION:` the element orientation info `EL_INFO.orientation` is generated (3d only).

During mesh traversal, such information is generated hierarchically using the two subroutines

```
void fill_macro_info(MACRO_EL *mel, EL_INFO *elinfo);
void fill_elinfo(int ichild, EL_INFO *parent_info, EL_INFO *elinfo);
```

Description:

`fill_macro_info(mel, elinfo):` fills `elinfo` with macro element information of `mel`;

`fill_elinfo(ichild, parent_info, elinfo):` fills `elinfo` for the child `ichild` using hierarchy information and parent data `parent_info`.

Sequence of visited elements

The sequence of elements which are visited during the traversal is given by the following rules:

- All elements in the binary mesh tree of one `MACRO_EL mel` are visited prior to any element in the tree of `mel->next`.
- For every `EL el`, all elements in the subtree `el->child[0]` are visited before any element in the subtree `el->child[1]`.
- The PREORDER, INORDER, and POSTORDER flags determine the traversal order of an element and its two child trees, as defined above in Section 2.2.18.

Only during non-recursive traversal, this order may be changed by calling explicitly the `traverse_neighbour()` routine, see below.

Recursive mesh traversal routines

Recursive traversal of mesh elements is done by the routine

```
void mesh_traverse(MESH *mesh, int level, FLAGS fill_flag,
                   void (*el_fct)(const EL_INFO *));
```

Description:

`mesh_traverse(mesh, level, fill_flag, el_fct)`: traverses the mesh `mesh`; the argument `level` specifies the element level if `CALL_EL_LEVEL` or `CALL_LEAF_EL_LEVEL`, or the multigrid level if `CALL_MG_LEVEL` is set; otherwise this variable is ignored; by the argument `fill_flag` the elements to be traversed and data to be filled into `EL_INFO` is selected, using bitwise *or* of one `CALL_...` flag and several `FILL_...` flags; the argument `el_fct` is a pointer to a function which is called on every element selected by the `CALL_...` part of `fill_flag`.

It is possible to use the recursive mesh traversal recursively, by calling `mesh_traverse()` from `el_fct`.

2.2.4. Example. An example of a mesh traversal where the volume of each leaf element should be calculated is:

```
...
extern void el_volume(const EL_INFO *);

mesh_traverse(mesh, -1, CALL_LEAF_EL|FILL_COORDS, el_volume);
```

where `el_volume()` is a function which calculates the volume of an element and thus needs information of the elements vertex coordinates.

2.2.5. Example. We give an implementation of the `CALL_EVERY_EL_...` routines to show the simple structure of all recursive traversal routines.

```
/* a TRAVERSE_INFO holds the traversal flag and a pointer to el_fct */

static void recursive_traverse(EL_INFO *elinfo, TRAVERSE_INFO *trinfo)
{
  EL      *el = elinfo->el;
  EL_INFO elinfo_new;

  if (el->child[0])
  {
    if (trinfo->flag & CALL_EVERY_EL_PREORDER)  (trinfo->el_fct)(elinfo);

    fill_elinfo(0, elinfo, &elinfo_new);
    recursive_traverse(&elinfo_new, trinfo);

    if (trinfo->flag & CALL_EVERY_EL_INORDER)   (trinfo->el_fct)(elinfo);

    fill_elinfo(1, elinfo, &elinfo_new);
    recursive_traverse(&elinfo_new, trinfo);

    if (trinfo->flag & CALL_EVERY_EL_POSTORDER) (trinfo->el_fct)(elinfo);
  }
```

```
  else
  {
    (trinfo->el_fct)(elinfo);
  }

  return;
}


static void mesh_traverse_every_el(MESH *mesh, FLAGS fill_flag,
                                   void (*el_fct)(const EL_INFO *));
{
  MACRO_EL *mel;
  EL_INFO elinfo;
  TRAVERSE_INFO traverse_info;

  elinfo.fill_flag = (flag & FILL_ANY);

  traverse_info.mesh   = mesh;
  traverse_info.el_fct = el_fct;
  traverse_info.flag = flag;
  elinfo.mesh = mesh;

  for (mel = mesh->first_macro_el; mel; mel = mel->next)
  {
    fill_macro_info(mel, &elinfo);
    recursive_traverse(&elinfo, &traverse_info);
  }
}
```

Non–recursive mesh traversal routines

Mesh refinement and coarsening routines (in the case where `NEIGH_IN_EL == 0`, see Sections 2.4.1 and 2.4.2) and the GRAPE graphic interface (see Section 2.16.3) are examples of functions which need a non–recursive access to the mesh elements.

The implementation of the non–recursive mesh traversal routines uses a stack to save the tree path from a macro element to the current element. A data structure **`TRAVERSE_STACK`** holds such information. Before calling the non–recursive mesh traversal routines, such a stack must be allocated (and passed to the traversal routines). The stack is initialized by each call to **`traverse_first()`**.

```
typedef struct traverse_stack   TRAVERSE_STACK;
```

By allocating a new stack, it is even possible to recursively call the non–recursive mesh traversal during another mesh traversal without destroying the stack which is already in use. Allocation and deallocation of traverse stacks is done by routines:

```
TRAVERSE_STACK *get_traverse_stack(void);
void free_traverse_stack(TRAVERSE_STACK *stack);
```

For non–recursive mesh traversal, the following routines are provided:

```
EL_INFO *traverse_first(TRAVERSE_STACK *stack, MESH *mesh,
                        int level, FLAGS fill_flag);
```

```
EL_INFO *traverse_next(TRAVERSE_STACK *stack, EL_INFO *);
```

Besides the stack, parameters are identical to those of `mesh_traverse()`, but there is no `el_fct()` to call on elements, because all operations are done by the routines which call these traversal functions. A traversal is initiated by calling `traverse_first()`, giving the first element of the mesh which fulfills the selection criterion specified by `level` and `fill_flag`. After doing all necessary operations on an element, a call to `traverse_next()` delivers the next element. After calling `traverse_next()`, all `EL_INFO` information about a previous element is invalid, the structure may be overwritten with new data.

The GRAPE MESH/HMESH interface implementation (see Section 2.16.3) and the gltools interface (see Section 2.16.2) uses the non–recursive mesh traversal.

2.2.6. Example. An application of the non–recursive mesh traversal routines is shown in the following code segment:

```
TRAVERSE_STACK *stack;
EL_INFO        *el_info;
FLAGS          fill_flag = CALL_LEAF_EL | FILL_COORDS;
int            level;

stack = get_traverse_stack();
for (el_info = traverse_first(stack, mesh, level, fill_flag);
     el_info;
     el_info = traverse_next(stack, el_info))
{
  /* perform action on element given by el_info */
}
free_traverse_stack(stack);
```

Neighbour traversal

Some applications, like the recursive refinement algorithm, need the possibility to go from an element to element using neighbour relations. Such a traversal can not be performed by the recursive traversal routines and thus needs the non–recursive mesh traversal.

The traversal routine for going from element to a neighbour is

```
EL_INFO *traverse_neighbour(TRAVERSE_STACK *stack, EL_INFO *, int neighbour);
```

and can be called at any time during the non–recursive mesh traversal after initializing the first element by `traverse_first(...)`.

`traverse_neighbour(stack, el_info, neighbour)` returns a pointer to an `EL_INFO` structure with information about the neighbour of `el_info->el` across edge/face `neighbour`. After a call to `traverse_neighbour()`, all `EL_INFO` information about a previous element is invalid, and can be regenerated only by calling `traverse_neighbour()` again with the *old* `OPP_VERTEX` value. If called at the boundary, when no adjacent element is available, then the routine returns `nil`; nevertheless, information from the old `EL_INFO` may be overwritten and lost. To avoid such behavior, one should check for boundary edges/faces before calling `traverse_neighbour()`.

Access to an element at world coordinates xy

Some applications need the access to elements at a special location in world coordinates. Examples are characteristic methods for convection problems, or the implementation of a special right hand side like point evaluations or curve integrals. For such purposes, a routine is available which returns an element pointer and corresponding barycentric coordinates:

```
int find_el_at_pt(const MESH *mesh, const REAL_D xy, EL **elp, REAL bary[DIM+1],
                  const MACRO_EL *start_mel, const REAL_D xy0, REAL *sp)
```

The parameters are:

`mesh`: mesh to be traversed;

`xy`: world coordinates of point (should be in the domain occupied by `mesh`);

`elp`: return address for a pointer to the element at `xy`;

`bary`: returns barycentric coordinates of `xy`;

`start_mel`: initial guess for the macro element containing `xy`, or `nil`;

`xy0`: start point from a characteristic method, see below, or `nil`;

`sp`: return address for relative distance to domain boundary in a characteristic method, see below, or `nil`.

The return value is `true` if `xy` is inside the domain, or `false` otherwise.

For a characteristic method, where $xy = xy_0 - \boldsymbol{V}\tau$, it may be convenient to know the point on the domain's boundary which lies on the line segment between the old point `xy0` and the new point `xy`, in case that `xy` is outside the domain. Such information is returned when `xy0` and a pointer `sp`$\neq$`nil` are supplied: `*sp` is set to the value s such that $xy_0 + s\,(xy - xy_0) \in \partial\text{Domain}$, and the element and local coordinates corresponding to that boundary point will be returned via `elp` and `bary`.

The implementation of `find_el_at_pt()` is based on the transformation from world to local coordinates, available via the routine `world_to_coord()`, compare Section 2.7. At the moment, `find_el_at_pt()` works correctly only for domains with non–curved boundary. This is due to the fact that the implementation first looks for the macro–element containing `xy` and then finds its path through the corresponding element tree based on the macro barycentric coordinates. For non–convex domains, it is possible that in some cases a point inside the domain is considered as external.

2.3 Administration of degrees of freedom

Degrees of freedom (DOFs) give connection between local and global finite element functions, compare Sections 1.4.2 and 1.3. We want to be able to have several finite element spaces and corresponding sets of DOFs at the same time. One set of DOFs may be shared between different finite element spaces, when appropriate.

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.

During coarsening of the mesh, when DOFs are deleted, all remaining DOFs have to be kept the same. In general, the deleted DOFs are not the ones which correspond to the largest integer index. Holes with unused indices appear in the total range of used indices. One of the main aspects of the DOF administration is to keep track of all used and unused indices. One possibility to remove holes form vectors is the compression of DOFs, i.e. the renumbering of all DOFs such that all unused indices are shifted to the end of the index range, thus removing holes of unused indices. While the global index corresponding to a DOF may change, the *relative* order of DOF indices remains unchanged during compression.

During refinement of the mesh, new DOFs are added, and additional indices are needed. If a deletion of DOFs created some unused indices before, some of these can be reused for the new DOFs. Otherwise, the total range of used indices has to be enlarged, and the new indices are taken from this new range. At the same time, all vectors and matrices which are supposed to use these DOF indices have to be adjusted in size, too. This is the next major aspect of the DOF administration. To be able to do this, lists of vectors and matrices are included in the `DOF_ADMIN` data structure. Entries are added to or removed from these lists via special subroutines, see Section 2.3.2.

In ALBERT, every abstract DOF is realized as an integer index into vectors:

```
typedef signed int   DOF;
```

These indices are administrated via the `DOF_ADMIN` data structure (see 2.3.1) and some subroutines. For each set of DOFs, one `DOF_ADMIN` structure is created. Degrees of freedom are directly connected with the mesh. The `MESH` data structure contains a reference to all sets of DOFs which are used on a mesh, compare Section 2.2.14. The `FE_SPACE` structure describing a finite element space references the corresponding set of DOFs, compare Sections 1.4.2, 2.5.1. Several `FE_SPACE`s may share the same set of DOFs, thus reference the same `DOF_ADMIN` structure. Usually, a `DOF_ADMIN` structure is created during definition of a finite element space by `get_fe_space()`, see Section 2.6.2. For special applications, additional DOF sets, that are not connected to any finite element space may also be defined (compare Section 2.6.2).

In Sections 2.3.5 and 2.3.6, we describe storage and access methods for global DOFs and local DOFs on single mesh elements.

As already mentioned above, special data types for data vectors and matrices are defined, see Sections 2.3.2 and 2.3.4. Several BLAS routines are available for such data, see Section 2.3.7.

2.3.1 The `DOF_ADMIN` data structure

The following data structure holds all data about one set of DOFs. It includes information about used and unused DOF indices, as well as linked lists of matrices and vectors of different data types, that are automatically resized and resorted during mesh changes. Currently, only an automatic *enlargement* of vectors is implemented, but no automatic shrinking. The actual implementation of used and unused DOFs is not described here in detail — it uses only one bit of storage for every integer index.

```
typedef struct dof_admin  DOF_ADMIN;
typedef unsigned int      DOF_FREE_UNIT;

struct dof_admin
{
  MESH         *mesh;
  const char   *name;

  DOF_FREE_UNIT *dof_free;    /* flag bit vector                          */
  unsigned int  dof_free_size;/* flag bit vector size                     */
  unsigned int  first_hole;   /* index of first non-zero dof_free entry   */
```

```
  int  size;              /* allocated size of dof_list vector        */
  int  used_count;        /* number of used dof indices               */
  int  hole_count;        /* number of FREED dof indices (NOT size-used)*/
  int  size_used;         /* > max. index of a used entry             */

  int  n_dof[DIM+1];      /* dofs from THIS dof_admin                 */
  int  n0_dof[DIM+1];     /* dofs from previous dof_admins            */

  DOF_INT_VEC     *dof_int_vec;        /* linked list of int vectors    */
  DOF_INT_VEC     *dof_dof_vec;        /* linked list of dof vectors    */
  DOF_INT_VEC     *int_dof_vec;        /* linked list of dof vectors    */
  DOF_UCHAR_VEC   *dof_uchar_vec;      /* linked list of u_char vectors */
  DOF_SCHAR_VEC   *dof_schar_vec;      /* linked list of s_char vectors */
  DOF_REAL_VEC    *dof_real_vec;       /* linked list of real vectors   */
  DOF_REAL_D_VEC  *dof_real_d_vec;     /* linked list of real_d vectors */
  DOF_MATRIX      *dof_matrix;         /* linked list of matrices       */
};
```

The entries yield following information:

mesh: this is a dof_admin on mesh;

name: a string holding a textual description of this dof_admin;

dof_free, dof_free_size, first_hole: internally used variables for administration of used and free DOF indices;

size: current size of vectors in dof_?_vec and dof_matrix lists;

used_count: number of used dof indices;

hole_count: number of *freed* dof indices (*not* size−used_count);

size_used: $\geq$ largest used DOF index;

n_dof: numbers of degrees of freedom defined by this dof_admin structure; n_dof[VERTEX], n_dof[EDGE], n_dof[FACE], and n_dof[CENTER] are the DOF counts at vertices, edges, faces (only in 3d) and element interiors, compare Section 2.3.6. These values are usually set by get_fe_space() as a copy from bas_fcts->n_dof (compare Section 2.5.1).

n0_dof: start indices n0_dof[VERTEX], ..., of the first dofs defined by this dof_admin in the element's dof[VERTEX], ..., vectors. These are the sums of degrees of freedom defined by previous dof_admin structures that were already added to the same mesh; n0_dof[VERTEX], n0_dof[EDGE], n0_dof[FACE], and n0_dof[CENTER] are all set automatically by get_fe_space(). See Section 2.3.6 for details and usage;

dof_?_vec, dof_matrix: pointers to linked lists of DOF_?_VEC and DOF_MATRIX structures which are associated with the DOFs administrated by this DOF_ADMIN and whose size is automatically adjusted during mesh refinements, compare Section 2.3.2.

Deletion of DOFs occurs not only when the mesh is (locally) coarsened, but also during refinement of a mesh with higher order elements. This is due to the fact, that during local interpolation operations, both coarse–grid and fine–grid DOFs must be present, so deletion of coarse–grid DOFs that are no longer used is done after allocation of new fine–grid DOFs. Usually, all operations concerning DOFs are done automatically by routines doing mesh adaption or handling finite element spaces. The removal of "holes" in the range of used DOF indices is not done automatically. It is actually not *needed* to be done, but may speed up the access in loops over global DOFs; When there are no holes, then a simple for–loop can be used without checking for each index, whether it is currently in use or not. The FOR_ALL_DOFS()–macro described in Section 2.3.5 checks this case. Hole removal is done for all DOF_ADMINs of a mesh by the function

```
void dof_compress(MESH *mesh);
```

Description:

dof_compress(mesh): remove all holes of unused DOF indices by compressing the used range of indices (it does *not* resize the vectors). While the global index corresponding to a DOF may change, the *relative* order of DOF indices remains unchanged during compression.

This routine is usually called after a mesh adaption involving higher order elements or coarsening.

2.3.1. Remark. In the moment, the use of DOF matrices which combine two different sets of DOFs may produce errors during dof_compress(). Such matrices should be cleared by calling clear_dof_matrix() before a call to dof_compress().

Usually, the range of DOF indices is enlarged in fixed increments given by the symbolic constant SIZE_INCREMENT, defined in dof_admin.c. If an estimate of the finally needed number of DOFs is available, then a direct enlargement of the DOF range to that number can be forced by calling:

```
void enlarge_dof_lists(DOF_ADMIN *admin, int minsize);
```

Description:

enlarge_dof_lists(admin, minsize): enlarges the range of the indices of admin to minsize.

2.3.2 Vectors indexed by DOFs: The DOF_VEC data structures

The DOFs described above are just integers that can be used as indices into vectors and matrices. During refinement and coarsening of the mesh, the number of used DOFs, the meaning of one integer index, and even the total range of DOFs change. To be able to handle these changes automatically for all vectors, which are indexed by the DOFs, special data structures are used which contain such vector data. Lists of these structures are kept in the DOF_ADMIN structure, so that all vectors in the lists can be resized together with the range of DOFs. During refinement and coarsening of elements, values can be interpolated automatically to new DOFs, and restricted from old DOFs, see Section 2.3.3.

ALBERT includes data types for vectors of type REAL, REAL_D, S_CHAR, U_CHAR, and int. Below, the DOF_REAL_VEC structure is described in detail. Structures DOF_REAL_D_VEC, DOF_SCHAR_VEC, DOF_UCHAR_VEC, and DOF_INT_VEC are declared similarly, the only difference between them is the type of the structure entry vec.

Although the administration of such vectors is done completely by the DOF administration which needs DOF_ADMIN data, the following data structures include a reference to a FE_SPACE, which includes additionally the MESH and BAS_FCTS. In this way, complete information about a finite element function given by a REAL– and REAL_D–valued vector is directly accessible.

```
typedef struct dof_real_vec  DOF_REAL_VEC;

struct dof_real_vec
{
  DOF_REAL_VEC *next;
  const FE_SPACE     *fe_space;

  const char  *name;
```

```
  int   size;
  REAL  *vec;  /* different type in DOF_INT_VEC, ... */

  void  (*refine_interpol)(DOF_REAL_VEC *, RC_LIST_EL *, int n);
  void  (*coarse_restrict)(DOF_REAL_VEC *, RC_LIST_EL *, int n);
};
```

The members yield following information:

`next`: linked list of `DOF_REAL_VEC` structures in `fe_space->admin`;

`fe_space`: `FE_SPACE` structure with information about DOFs and basis functions;

`name`: string with a textual description of vector values, or `nil`;

`size`: current size of `vec`;

`vec`: pointer to `REAL` vector of size `size`;

`refine_interpol, coarse_restrict`: interpolation and restriction routines, see Section 2.3.3. For `REAL` and `REAL_D` vectors, these usually point to the corresponding routines from `fe_space->bas_fcts`, compare Section 2.5.1. While we distinguish there between *restriction* and *interpolation* during coarsening, only one such operation is appropriate for a given vector, as it either represents a finite element function or values of a functional applied to basis functions.

Usually, all DOF vectors linked in the corresponding `dof_admin->dof_?_vec` list are automatically adjusted in size and reordered during mesh changes. Values are transformed during local mesh changes, if the `refine_interpol` and/or `coarse_restrict` entries are not `nil`, compare Section 2.3.3.

Integer DOF vectors can be used in several ways: They may either hold an `int` value for each DOF, or reference a `DOF` value for each DOF. In both cases, the vectors should be automatically resized and rearranged during mesh changes. Additionally, values should be automatically changed in the second case. Such vectors are referenced in the `dof_admin->dof_int_vec` and `dof_admin->dof_dof_vec` lists.

On the other hand, `DOF_INT_VEC`s provide a way to implement for special applications a vector of `DOF` values, which is *not indexed* by DOFs. For such vectors, only the values are automatically changed during mesh changes, but not the size or order. The user program is responsible for allocating memory for the `vec` vector. Such DOF vectors are referenced in the `dof_admin->int_dof_vec` list.

A macro `GET_DOF_VEC` is defined to simplify the secure access to a `DOF_VEC`'s data. It assigns `dof_vec->vec` to `ptr`, if both `dof_vec` and `dof_vec->vec` are not `nil`, and generates an error in other cases:

```
#define GET_DOF_VEC(ptr, dof_vec) TEST_EXIT((dof_vec)&&(ptr = (dof_vec)->vec))\
     ("%s == nil", (dof_vec) ? (dof_vec)->name : #dof_vec)
```

The following subroutines are provided to handle DOF vectors. Allocation of a new `DOF_VEC` and freeing of a `DOF_VEC` (together with its `vec`) are done with:

```
DOF_REAL_VEC    *get_dof_real_vec(const char *name, const FE_SPACE *fe_space);
DOF_REAL_D_VEC  *get_dof_real_d_vec(const char *name, const FE_SPACE *fe_space);
DOF_INT_VEC     *get_dof_int_vec(const char *name, const FE_SPACE *fe_space);
DOF_INT_VEC     *get_dof_dof_vec(const char *name, const FE_SPACE *fe_space);
DOF_INT_VEC     *get_int_dof_vec(const char *name, const FE_SPACE *fe_space);
DOF_SCHAR_VEC   *get_dof_schar_vec(const char *name, const FE_SPACE *fe_space);
DOF_UCHAR_VEC   *get_dof_uchar_vec(const char *name, const FE_SPACE *fe_space);
void            free_dof_real_vec(DOF_REAL_VEC *vec);
```

```
void            free_dof_real_d_vec(DOF_REAL_D_VEC *vec);
void            free_dof_int_vec(DOF_INT_VEC *vec);
void            free_dof_dof_vec(DOF_INT_VEC *vec);
void            free_int_dof_vec(DOF_INT_VEC *vec);
void            free_dof_schar_vec(DOF_SCHAR_VEC *vec);
void            free_dof_uchar_vec(DOF_UCHAR_VEC *vec);
```

By specifying a finite element space for a DOF_VEC, the corresponding set of DOFs is implicitly specified by `fe_space->admin`. The DOF_VEC is linked into `DOF_ADMIN`'s appropriate `dof_?_vec` list for automatic handling during mesh changes. The DOF_VEC structure entries `next` and `admin` are set during creation and must not be changed otherwise! The size of the `dof_vec->vec` vector is automatically adjusted to the range of DOF indices controlled by `fe_space->admin`.

There is a special list for each type of DOF vectors in the `DOF_ADMIN` structure. All `DOF_REAL_VEC`s, `DOF_REAL_D_VEC`s, `DOF_UCHAR_VEC`s, and `DOF_SCHAR_VEC`s are added to the respective lists, whereas a `DOF_INT_VEC` may be added to one of three lists in `DOF_ADMIN`: `dof_int_vec`, `dof_dof_vec`, and `int_dof_vec`. The difference between these three lists is their handling during a resize or compress of the DOF range. In contrast to all other cases, for a vector in `admin`'s `int_dof_vec` list, the `size` is NOT changed with `admin->size`. But the values `vec[i]`, $i = 1, \dots,$ `size` are adjusted when `admin` is compressed, for example. For vectors in the `dof_dof_vec` list, both adjustments in `size` and adjustment of values is done.

The `get_?_vec()` routines automatically allocate enough memory for the data vector `vec` as indicated by `fe_space->admin->size`. Pointers to the routines `refine_interpol` and `coarse_restrict` are set to `nil`. They must be set explicitly after the call to `get_?_vec()` for an interpolation during refinement and/or interpolation/restriction during coarsening. The `free_?_vec()` routines remove the vector from a `vec->fe_space->admin->dof_?_vec` list and free the memory used by `vec->vec` and `*vec`.

A printed output of the vector entries together with its `name` (mainly for test purposes) is produced by the routines:

```
void print_dof_int_vec(const DOF_INT_VEC *div);
void print_dof_real_vec(const DOF_REAL_VEC *drv);
void print_dof_real_d_vec(const DOF_REAL_D_VEC *drdv);
void print_dof_schar_vec(const DOF_SCHAR_VEC *dsv);
void print_dof_uchar_vec(const DOF_UCHAR_VEC *duv);
```

2.3.3 Interpolation and restriction of DOF vectors during mesh refinement and coarsening

During mesh refinement and coarsening, new DOFs are produced, or old ones are deleted. In many cases, information stored in `DOF_VEC`s has to be adjusted to the new distribution of DOFs. To do this automatically during the refinement and coarsening process, each `DOF_VEC` can provide pointers to subroutines `refine_interpol` and `coarse_restrict`, that implements these operations on data. During refinement and coarsening of a `mesh`, these routines are called for all `DOF_VEC`s with non-`nil` pointers in all `DOF_ADMIN`s in `mesh->dof_admin`.

Before doing the mesh operations, it is checked whether any automatic interpolations or restrictions during refinement or coarsening are requested. If yes, then the corresponding operations will be performed during local mesh changes.

As described in Sections 2.4.1 and 2.4.2, interpolation resp. restriction of values is done during the mesh refinement and coarsening locally on every refined resp. coarsened patch of elements. Which of the local DOFs are created new, and which ones are kept from parent/children elements, is described in these other sections, too. All necessary interpolations or restrictions are done by looping through all `DOF_ADMIN`s in `mesh` and calling the `DOF_VEC`'s routines

```
struct dof_real_vec
{
  ...

  void  (*refine_interpol)(DOF_REAL_VEC *, RC_LIST_EL *, int );
  void  (*coarse_restrict)(DOF_REAL_VEC *, RC_LIST_EL *, int );
}
```

Those implement interpolation and restriction on one patch for this `DOF_VEC`. Only these have to know about the actual meaning of the DOFs. Here, `RC_LIST_EL` is a vector holding pointers to all `n` parent elements which build the patch (and thus have a common refinement edge). Usually, the interpolation and restriction routines for `REAL` or `REAL_D` vectors are defined in the corresponding `dof_vec->fe_space->bas_fcts` structures. Interpolation or restriction of non–real values (`int` or `CHAR`) is usually application dependent and is not provided in the `BAS_FCTS` structure.

Examples of these routines are shown in Section 2.5.2.

2.3.4 The `DOF_MATRIX` data structure

Not only vectors indexed by DOFs are available in ALBERT, but also matrices which operate on these `DOF_VEC`s. For finite element calculations, these matrices are usually sparse, and should be stored in a way that reflects this sparseness. We use a storage method which is similar to the one used in [31]. Every row of a matrix is realized as a linked list of `MATRIX_ROW` structures, each holding a maximum of `ROW_LENGTH` matrix entries from that row. Each entry consists of a column DOF index and the corresponding `REAL` matrix entry. Unused entries in a `MATRIX_ROW` are marked with a negative column index. The `ROW_LENGTH` is a symbolic preprocessor constant defined in `albert.h`. For `DIM=2` meshes built from triangles, the refinement by bisection generates usually at most eight elements meeting at a common vertex, more elements may meet only at macro vertices. Thus, for piecewise linear (Lagrange) elements on triangles, up to nine entries are non–zero in most rows of a mass or stiffness matrix. This motivates the choice `ROW_LENGTH = 9`. For higher order elements or tetrahedra, there are much more non–zero entries in each row. Thus, a split of rows into short `MATRIX_ROW` parts should not produce too much overhead.

```
typedef struct matrix_row  MATRIX_ROW;

struct matrix_row
{
   MATRIX_ROW *next;
   int         col[ROW_LENGTH];
   REAL        entry[ROW_LENGTH];
};

#define ENTRY_USED(col)         (col >= 0)
#define ENTRY_NOT_USED(col)     (col < 0)
#define UNUSED_ENTRY    -1
#define NO_MORE_ENTRIES -2
```

Such a `DOF_MATRIX` structure is usually filled by local operations on single elements, using the `update_matrix()` routine, compare Section 2.12.1, which automatically generates space for new matrix entries by adding new `MATRIX_ROW`s, if needed.

Similar to DOF vectors, the DOF_MATRIX structure contains pointers to routines for interpolation and restriction during mesh refinement and coarsening. Providing such routines, an existing DOF_MATRIX can be updated by local operations, and a complete recalculation is not necessary. For DOF vectors describing finite element functions, such an interpolation can be necessary even from a mathematical point of view. For matrices, this is more mandatory. For implicit discretizations, where a (non–) linear system involving the DOF_MATRIX has to be solved, this solution is usually much more expensive than a complete new matrix recalculation. Thus, local matrix updates will not save much time. But for explicit discretizations or for expensive matrices, such a local matrix update may save a noticeable amount of computing time.

A DOF_MATRIX may also be used to combine two different sets of DOFs, compare Section 2.12.1. In the moment, the use of such matrices may produce errors during dof_compress(). Such a matrix should be cleared by calling clear_dof_matrix() before a call to dof_compress().

```
typedef struct dof_matrix  DOF_MATRIX;

struct dof_matrix
{
  DOF_MATRIX     *next;
  const FE_SPACE *fe_space;
  const char     *name;

  MATRIX_ROW     **matrix_row;
  DOF            size;

  void           (*refine_interpol)(DOF_MATRIX *, RC_LIST_EL *, int n);
  void           (*coarse_restrict)(DOF_MATRIX *, RC_LIST_EL *, int n);
};
```

The entries yield following information:

next: linked list of DOF_MATRIX structures in fe_space->admin;

fe_space: FE_SPACE structure with information about corresponding row DOFs and basis functions;

name: a textual description for the matrix, or nil;

matrix_row: vector of pointers to MATRIX_ROWs, one for each row;

size: current size of the matrix_row vector;

refine_interpol, coarse_restrict: interpolation and restriction routines, see Section 2.3.3.

Allocation and freeing of DOF–matrices is done via routines

```
DOF_MATRIX *get_dof_matrix(const char *name, const FE_SPACE *fe_space);
void       free_dof_matrix(DOF_MATRIX *matrix);
```

As described above for DOF vectors, get_dof_matrix() does automatically link the new matrix into the fe_space->admin->dof_matrix list. A matrix_row vector of length fe_space->admin->size is allocated and all entries are set to nil. The free_dof_matrix() routine frees first all MATRIX_ROWs in matrix->matrix_row, then matrix->matrix_row, and finally the structure *matrix.

Subroutines

```
void       clear_dof_matrix(DOF_MATRIX *matrix);
void       print_dof_matrix(const DOF_MATRIX *matrix);
```

are available to clear a matrix or (mainly for test purposes) to print its contents. `clear_dof_matrix()` does not reset all entries of a matrix to zero: it *removes* all entries from the matrix. This is done by freeing all `MATRIX_ROW`s in `matrix->matrix_row` and setting all entries in `matrix->matrix_row` to `nil`.

2.3.5 Access to global DOFs: Macros for iterations using DOF indices

For loops over all used (or free) DOFs, the following macros are defined:

```
FOR_ALL_DOFS(admin, todo);
FOR_ALL_FREE_DOFS(admin, todo);
```

Here, `ADMIN *admin` is the pointer to an admin structure, and `todo` is a list of C-statements which are to be executed on every used resp. unused DOF index. During `todo`, the local variable `int dof` holds the current index of an used resp. unused entry. It must not be altered by `todo`.

Two examples illustrate the usage of these macros.

2.3.2. Example (Initialization of vectors). This BLAS–1 routine `dset()` initializes all elements of a vector with a given value; for `DOF_REAL_VEC`s we have to set this value for all *used* DOFs. All used entries of the `DOF_REAL_VEC *drv` are set to a value α by:

```
FOR_ALL_DOFS(drv->fe_space->admin, drv->vec[dof] = alpha);
```

The BLAS–1 routine `dof_set()` is written this way, compare Section 2.3.7.

2.3.3. Example (Matrix–vector multiplication). As a more complex example we give an implementation of the matrix–vector product in `dof_mv()`, compare Sections 2.3.4 and 2.3.7:

```
FOR_ALL_DOFS(admin,
               sum = 0.0;
               for (row = a->matrix_row[dof]; row; row = row->next) {
                 for (j=0; j<ROW_LENGTH; j++) {
                   jcol = row->col[j];
                   if (ENTRY_USED(jcol)) {
                     sum += row->entry[j] * xvec[jcol];
                   }
                   else {
                     if (jcol == NO_MORE_ENTRIES)
                       break;
                   }
                 }
               }
               yvec[dof] = sum;
             );
```

2.3.6 Access to local DOFs on elements

As shown by the examples in Figure 1.17, the DOF administration is able to handle different sets of DOFs, defined by different `DOF_ADMIN` structures, at the same time. All operations with finite element functions, like evaluation or integration, are done locally on the level of single elements. Thus, access on element level to DOFs from a single `DOF_ADMIN` has to be provided in a way that is independent from all other finite element spaces which might be defined on the mesh.

As described in Section 2.2.8, the `EL` data structure holds a vector of pointers to DOF vectors, that holds data for *all* DOFs on the element from all `DOF_ADMIN`s:

```
struct el
{
  ...
  DOF              **dof;
  ...
};
```

During initialization of a mesh, the lengths of these vectors is computed by collecting data from all `DOF_ADMIN`s associated with the mesh; details are given below. Information about all DOFs associated with a mesh is collected and accessible in the `MESH` data structure (compare Section 2.2.14):

```
struct mesh
{
  ...

  DOF_ADMIN **dof_admin;
  int       n_dof_admin;

  int       n_dof_el;
  int       n_dof[DIM+1];
  int       n_node_el;
  int       node[DIM+1];
};
```

The meaning of these variables is:

dof_admin: a vector of pointers to all `DOF_ADMIN` structures for the mesh;

n_dof_admin: number of all `DOF_ADMIN` structures for the mesh;

n_dof_el: total number of DOFs on one element from all `DOF_ADMIN` structures;

n_dof: total number of `VERTEX`, `EDGE`, `FACE`, and `CENTER` DOFs from all `DOF_ADMIN` structures;

n_node_el: number of used nodes on each element (vertices, edges, faces, and center), this gives the dimension of `el->dof`;

node: The entry `node[i]`, $i \in \{\texttt{VERTEX}, \texttt{EDGE}, \texttt{FACE}, \texttt{CENTER}\}$ gives the index of the first `i`-node in `el->dof`.

All these variables must not be changed by a user routine — they are set during the `init_dof_admins()` routine given as a parameter to `GET_MESH()`, compare Section 2.2.15. Actually, only a subroutine `get_fe_space()` is allowed to change such information (compare Section 2.6.2).

We denote the different locations of DOFs on an element by *nodes*. As there are DOFs connected with different–dimensional (sub–) simplices, there are *vertex*, *edge*, *face*, and *center* nodes. Using the symbolic constants from Section 2.2.2, there may be `N_VERTICES` vertex nodes, `N_EDGES` edge nodes, `N_FACES` face nodes (in 3d), and one center node. Depending on the finite element spaces in use, not all possible nodes must be associated with DOFs, but some nodes may be associated with DOFs from several different finite element spaces (and several `DOF_ADMIN`s). In order to minimize the memory usage for pointers and DOF vectors, the elements store data only for such nodes where DOFs are used. Thus, the number of nodes on an element is determined during mesh initialization, when all finite element spaces and DOFs are defined. The total number of nodes is stored in `mesh->n_node_el`, which will be the length of the `el->dof` vector for all elements.

In order to access the DOFs for one node, `mesh->node[l]` contains the index of the first `l`–node in `el->dof`, where `l` is either `VERTEX`, `EDGE`, `FACE`, or `CENTER` (compare Figure 2.2). So, a pointer to

DOFs from the i–th edge node is stored at `el->dof[mesh->node[EDGE]+i]` ($0 \leq$ i $<$ N_EDGES), and these DOFs (and the vector holding them) are shared by all elements meeting at this edge.

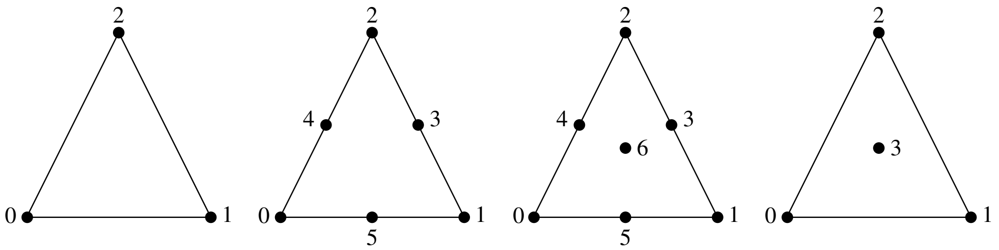


Figure 2.2: DOF vector indices in `el->dof` for DOFs at vertices, vertices and edges, vertices, edges and center, and vertices and center (in 2d). Corresponding `mesh->node` values are {0,0,0}, {0,3,0}, {0,3,6}, and {0,0,3}.

The total number of DOFs at an l–node is available in `mesh->n_dof[l]`. This number is larger than zero, iff the node is in use. All DOFs from different `DOF_ADMIN`s are stored together in one vector. In order to access DOFs from a given finite element space (and its associated `DOF_ADMIN`), the start index for DOFs from this `DOF_ADMIN` must be known. This start index is generated during mesh initialization and stored in `admin->n0_dof[l]`. The number of DOFs from this `DOF_ADMIN` is given in `admin->n_dof[l]`. Thus, a loop over all DOFs associated with the i–th edge node can be done by:

```
DOF *dof_ptr = el->dof[mesh->node[EDGE]+i] + admin->n0_dof[EDGE];
for (j = 0 ; j < admin->n_dof[EDGE]; j++)
{
  dof = dof_ptr[j];
  ...
}
```

In order to simplify the access of DOFs for a finite element space on an element, the `BAS_FCTS` structure must provide a routine

```
const DOF *(*get_dof_indices)(const EL *,const DOF_ADMIN *,int *);
```

which returns a vector containing all global DOFs associated with basis functions, in the correct order: the k–th DOF is associated with the k–th local basis function (compare Section 2.5.1).

2.3.7 BLAS routines for DOF vectors and matrices

Several basic linear algebra subroutines (BLAS [27, 12]) are implemented for DOF vectors and DOF matrices, see Table 2.1. Some non–standard routines are added: `dof_xpay()` is a variant of `dof_axpy()`, `dof_min()` and `dof_max()` calculate minimum and maximum values, and `dof_mv()` is a simplified version of the general `dof_gemv()` matrix-vector multiplication routine. The BLAS–2 routines `dof_gemv()` and `dof_mv()` use a `MatrixTranspose` flag: `transpose`= 0 indicates the original matrix, while `transpose`= 1 indicates the transposed matrix.

Similar routines are provided for `DOF_REAL_D` vectors, see Table 2.2.

2.3.8 Reading and writing of meshes and vectors

Section 2.2.16 described the input and output of ASCII files for macro triangulations. Locally refined triangulations including the mesh hierarchy and corresponding DOFs are saved in binary

`REAL dof_nrm2(const DOF_REAL_VEC *x)`	$nrm2 = (\sum X_i^2)^{1/2}$
`REAL dof_asum(const DOF_REAL_VEC *x)`	$asum = \sum \lvert X_i \rvert$
`REAL dof_min(const DOF_REAL_VEC *x)`	$min = \min X_i$
`REAL dof_max(const DOF_REAL_VEC *x)`	$max = \max X_i$
`void dof_set(REAL alpha, DOF_REAL_VEC *x)`	$X = (\alpha, \ldots, \alpha)$
`void dof_scal(REAL alpha, DOF_REAL_VEC *x)`	$X = \alpha * X$
`REAL dof_dot(const DOF_REAL_VEC *x, const DOF_REAL_VEC *y)`	$dot = \sum X_i Y_i$
`void dof_copy(const DOF_REAL_VEC *x, DOF_REAL_VEC *y)`	$Y = X$
`void dof_axpy(REAL alpha,` `const DOF_REAL_VEC *x, DOF_REAL_VEC *y)`	$Y = \alpha * X + Y$
`void dof_xpay(REAL alpha,` `const DOF_REAL_VEC *x, DOF_REAL_VEC *y)`	$Y = X + \alpha * Y$

`void dof_gemv(MatrixTranspose transpose, REAL alpha,`
`const DOF_MATRIX *a, const DOF_REAL_VEC *x,`
`REAL beta, DOF_REAL_VEC *y)`
	$Y = \alpha * A * X + \beta * Y$
	or
	$Y = \alpha * A^t * X + \beta * Y$

`void dof_mv(MatrixTranspose transpose,`
`const DOF_MATRIX *a,`
`const DOF_REAL_VEC *x, DOF_REAL_VEC *y)`
	$Y = A * X$
	or
	$Y = A^t * X$

Table 2.1: Implemented BLAS routines for DOF vectors and matrices

`REAL dof_nrm2_d(const DOF_REAL_D_VEC *x)`	$nrm2 = (\sum \lvert X_i \rvert^2)^{1/2}$
`void dof_set_d(REAL alpha, DOF_REAL_D_VEC *x)`	$X = ((\alpha, \ldots, \alpha), \ldots)$
`void dof_scal_d(REAL alpha, DOF_REAL_D_VEC *x)`	$X = \alpha * X$
`REAL dof_dot_d(const DOF_REAL_D_VEC *x,` `const DOF_REAL_D_VEC *y)`	$dot = \sum (X_i, Y_i)$
`void dof_copy_d(const DOF_REAL_D_VEC *x, DOF_REAL_D_VEC *y)`	$Y = X$
`void dof_axpy_d(REAL alpha,` `const DOF_REAL_D_VEC *x, DOF_REAL_D_VEC *y)`	$Y = \alpha * X + Y$
`void dof_xpay_d(REAL alpha,` `const DOF_REAL_D_VEC *x, DOF_REAL_D_VEC *y)`	$Y = X + \alpha * Y$

`void dof_mv_d(MatrixTranspose transpose,`
`const DOF_MATRIX *a,`
`const DOF_REAL_D_VEC *x, DOF_REAL_D_VEC *y)`
	$Y = A * X$
	or
	$Y = A^t * X$

Table 2.2: Implemented BLAS routines for `DOF_REAL_D` vectors

formats. Finite element data is saved (and restored) in binary format, too, in order to keep the full data precision. As the binary data and file format does usually depend on hardware and operating system, the interchange of data between different platforms needs a machine independent format. The XDR (External Data Representation) library provides a widely used interface for such a format. The `_xdr` routines should be used whenever data must be transfered between different computer platforms.

```
int write_mesh(MESH *mesh, const char *file, REAL time);

MESH *read_mesh(const char *file, REAL *timeptr,
                void (*init_leaf_data)(LEAF_DATA_INFO *),
                const BOUNDARY *init_boundary(MESH *mesh, int ibdry));
```

The routine `write_mesh` stores information about the mesh in a file named `file`. Written data includes the corresponding `time` (only important for time dependent problems), macro elements, mesh elements including the parent/child hierarchy information, DOF administration and element DOFs. The return value is 1 if an error occurs, otherwise 0.

Routine `read_mesh` reads a complete mesh from file `file`, which was created by `write_mesh`. The corresponding time, if any, is stored at `*timeptr`. The arguments `init_leaf_data` and `init_boundary` are used in the same way as in `read_macro`, compare Section 2.2.16.

For input and output of finite element data, the following routines are provided which read of write files containing binary DOF vectors.

```
int write_dof_int_vec(const DOF_INT_VEC *div, const char *file);
int write_dof_real_vec(const DOF_REAL_VEC *drv, const char *file);
int write_dof_real_d_vec(const DOF_REAL_D_VEC *drdv, const char *file);
int write_dof_schar_vec(const DOF_SCHAR_VEC *dsv, const char *file);
int write_dof_uchar_vec(const DOF_UCHAR_VEC *duv, const char *file);

DOF_INT_VEC    *read_dof_int_vec(const char *file, MESH *, FE_SPACE*);
DOF_REAL_VEC   *read_dof_real_vec(const char *file, MESH *, FE_SPACE *);
DOF_REAL_D_VEC *read_dof_real_d_vec(const char *file, MESH *, FE_SPACE *);
DOF_SCHAR_VEC  *read_dof_schar_vec(const char *file, MESH *, FE_SPACE *);
DOF_UCHAR_VEC  *read_dof_uchar_vec(const char *file, MESH *, FE_SPACE *);
```

For the output and input of machine independent data files, similar routines are provided. The XDR library is used, and thus all routine names end with `_xdr`:

```
int write_mesh_xdr(MESH *mesh, const char *file, REAL time);

MESH *read_mesh_xdr(const char *file, REAL *timeptr,
                    void (*init_leaf_data)(LEAF_DATA_INFO *),
                    const BOUNDARY *init_boundary(MESH *mesh, int ibdry));
```

```
int write_dof_int_vec_xdr(const DOF_INT_VEC *div, const char *file);
int write_dof_real_vec_xdr(const DOF_REAL_VEC *drv, const char *file);
int write_dof_real_d_vec_xdr(const DOF_REAL_D_VEC *drdv, const char *file);
int write_dof_schar_vec_xdr(const DOF_SCHAR_VEC *dsv, const char *file);
int write_dof_uchar_vec_xdr(const DOF_UCHAR_VEC *duv, const char *file);
```

```
DOF_INT_VEC    *read_dof_int_vec_xdr(const char *file, MESH *, FE_SPACE*);
DOF_REAL_VEC   *read_dof_real_vec_xdr(const char *file, MESH *, FE_SPACE *);
DOF_REAL_D_VEC *read_dof_real_d_vec_xdr(const char *file, MESH *, FE_SPACE *);
DOF_SCHAR_VEC  *read_dof_schar_vec_xdr(const char *file, MESH *, FE_SPACE *);
DOF_UCHAR_VEC  *read_dof_uchar_vec_xdr(const char *file, MESH *, FE_SPACE *);
```

2.3.4. Remark. In the moment, the `read_mesh(_xdr)()` and `write_mesh(_xdr)()` are only available if `NEIGH_IN_EL==0`.

2.4 The refinement and coarsening implementation

2.4.1 The refinement routines

The refinement of a mesh is done by the function

```
U_CHAR refine(MESH *mesh);
```

Description:

`refine(mesh)`: refines all leaf element with a *positive* element marker `mark` times (this mark is usually set by some adaptive procedure); the routine loops over all leaf elements and refines the elements with a positive marker until there is no element left with a positive marker; the return value is `MESH_REFINED`, if an element was refined, and `0` otherwise. Every refinement has to be done via this routine. The basic steps of this routine are described below.

A second function for refinement is

```
U_CHAR global_refine(MESH *mesh, int mark);
```

Description:

`global_refine(mesh, mark)`: sets all element markers for leaf elements of `mesh` to `mark`; the mesh is then refined by `refine()` which results in a `mark` global refinement of the mesh; the return value is `MESH_REFINED`, if `mark` is positive, and `0` otherwise.

Basic steps of the refinement algorithm

The refinement of a mesh is principally done in two steps. In the first step no coordinate information is available on the elements. In the case that neighbour information is stored at the elements such coordinate information can not be produced when going from one neighbour to another by the neighbour pointers. Thus, only a topological refinement is performed. If new nodes are created on the boundary these can be projected onto a curved boundary in the second step when coordinate information is available.

Thus, the algorithm looks as follows:

1. The whole mesh is refined only topologically. This part consists out of
   - the collection of a compatible refinement patch; this includes the recursive refinement of adjacent elements with an incompatible refinement edge;
   - the topological bisection of the patch elements;
   - the transformation of leaf data from parent to child, if such a function is available in the `leaf_data_info` structure;
   - allocation of new DOFs;

- handing on of DOFs from parent to the children;
- interpolation of DOF vectors and matrices from the coarse grid to the fine one on the whole refinement patch, if the function `refine_interpol()` is available for these DOF vectors (compare Section 2.3.3); these routines must not use coordinate information;
- a deallocation of DOFs on the parent when `preserve_coarse_dofs == 0`.

This process is described in detail below.

2. New nodes which belong to the curved part of the boundary are now projected onto the curved boundary via the `param_bound()` function in the `BOUNDARY` structure of the refinement edge. The coordinates of the projected node are stored in a `REAL_D`–vector and the pointers `el->new_coord` of all parents `el` which belong to the refinement patch are set to this vector. Figure 2.3 shows some refinements of a triangle with one edge on the curved boundary. The projections of refinement edge midpoints (small circles) to the curved boundary are shown by the black dots.

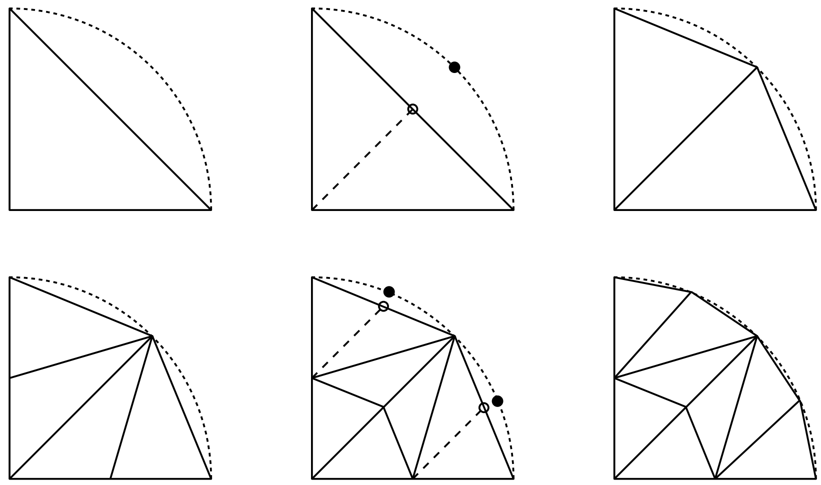


Figure 2.3: Refinement at curved boundary: refinement edge midpoints ∘ are projected by `param_bound()` to the curved boundary •

The topological refinement is done by the recursive refinement Algorithm 1.1.5. First, all all elements at the refinement edge of an marked element are collected. If a neighbour with an incompatible refinement edge is found, this neighbour is refined first by a recursive call of the refinement function. Thus, after looping around the refinement edge, the patch of simplices at this edge is always a compatible divisible refinement patch. The elements of this patch are stored in a vector `ref_list` with elements of type `RC_LIST_EL`, compare Section 2.2.13. This vector is an argument for the functions for interpolation of DOF vectors/matrices during refinement, compare Section 2.3.3.

In 2d the vector has length `2` and only the `el` entry of the components is set and used. If the refinement edge is an interior edge the number of elements at the edge is `2`, otherwise only the element itself has to be refined.

In 3d this vector is allocated with length `mesh->max_edge_neigh`. As mentioned in Section 2.2.13 we can define an orientation of the edge and by this orientation we can define the right and left neighbours (inside the patch) of an element at this edge.

Then the patch is bisected by first inserting a new vertex at the midpoint of the refinement edge. Then all elements of the refinement patch are bisected. This includes the allocation of new

DOFs, the adjustment of DOF pointers, and the memory allocation for leaf data (if the leaf data size is positive) and transformation of leaf data from parent to child (if a pointer to a function `refine_leaf_data()` is provided by the user in the `leaf_data_info` structure). Then memory for parents' leaf data is freed and stored information is definitely lost.

In the case of higher order elements we also have to add new DOFs on the patch and if we do not need information about the higher order DOFs on coarser levels they are removed from the parents. There are some basic rules for adding and removing DOFs which are important for the prolongation and restriction of data (see Section 2.3.3):

1. Only DOFs of the same kind (i.e. `VERTEX`, `EDGE`, or `FACE`) and whose nodes have the same geometrical position on parent and child are handed on to this child from the parent;
2. DOFs at a vertex, an edge or a face belong to all elements sharing this vertex, edge, face, respectively;
3. DOFs on the parent are only removed if the entry `preserve_coarse_dofs` in the mesh data structure is `false`; in that case only DOFs which are not handed on to a child are removed on the parent.

A direct consequence of 1. is that only DOFs inside the patch are added or removed; DOFs on the patch boundary stay untouched. `CENTER` DOFs can not be handed from parent to child since the centers of the parent and the children are always at different positions.

Using standard Lagrange finite elements, only DOFs that are not handed from parent to child have to be set while interpolating a finite element function to the finer grid; all values of the other DOFs stay the same (the same holds during coarsening and interpolating to the coarser grid).

Due to 2. it is clear that DOFs shared by more than one element have to be allocated only once and pointers to these DOFs are set correctly for all elements sharing it.

Now, we take a closer look at DOFs that are handed on by the parents and those that have to be allocated: In 2d we have

```
child[0]->dof[0] = el->dof[2];
child[0]->dof[1] = el->dof[0];
child[1]->dof[0] = el->dof[1];
child[1]->dof[1] = el->dof[2];
```

and in 3d for `child[0]` we always have

```
child[0]->dof[0] = el->dof[0];
child[0]->dof[1] = el->dof[2];
child[0]->dof[2] = el->dof[3];
```

For `child[1]` this passing of DOFs depends on the type of the parent. For a parent of type `0` we have

```
child[1]->dof[0] = el->dof[1];
child[1]->dof[1] = el->dof[3];
child[1]->dof[2] = el->dof[2];
```

and for a parent of type `1` or `2`

```
child[1]->dof[0] = el->dof[1];
child[1]->dof[1] = el->dof[2];
child[1]->dof[2] = el->dof[3];
```

For both dimensions

```
child[0]->dof[DIM] = child[1]->dof[DIM]
```

is the newly allocated DOF at the midpoint of the refinement edge (compare Figures 1.4 and 1.5).

In the case that we have DOFs at the midpoint of edges the following DOFs are passed on (let `enode = mesh->node[EDGE]` be the offset for DOFs at edges): for 2d

```
child[0]->dof[enode+2] = el->dof[enode+1];
child[1]->dof[enode+2] = el->dof[enode+0];
```

and for 3d

```
child[0]->dof[enode+0] = el->dof[enode+1];
child[0]->dof[enode+1] = el->dof[enode+2];
child[0]->dof[enode+3] = el->dof[enode+5];
```

for `child[0]` a for `child[1]` of a parent of type `0`

```
child[1]->dof[enode+0] = el->dof[enode+4];
child[1]->dof[enode+1] = el->dof[enode+3];
child[1]->dof[enode+3] = el->dof[enode+5];
```

and finally for `child[1]` of a parent of type `1` or `2`

```
child[1]->dof[enode+0] = el->dof[enode+3];
child[1]->dof[enode+1] = el->dof[enode+4];
child[1]->dof[enode+3] = el->dof[enode+5];
```

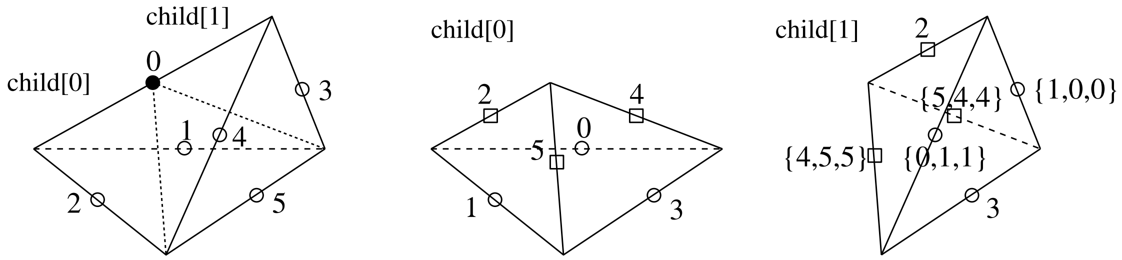


Figure 2.4: Edge DOFs that are freed $\bullet$, passed on $\circ$, and newly allocated $\Box$

We also have to create new DOFs (compare Figure 2.4). Two additional DOFs are created in the refinement edge which are shared by all patch elements. Pointers to these DOFs are adjusted for

```
child[0]->dof[enode+0],
child[1]->dof[enode+1]
```

in 2d and

```
child[0]->dof[enode+2],
child[1]->dof[enode+2]
```

in 3d for all patch elements.

In 3d for each interior face of the refinement patch there is a new edge where we have to add a new DOF vector. These DOFs are shared by two children in the case of a boundary face; otherwise it is shared by four children and pointers of

```
child[0]->dof[enode+4] = child[1]->dof[enode+{5,4,4}],
child[0]->dof[enode+5] = child[1]->dof[enode+{4,4,5}]
```

are adjusted for those elements.

In 3d there may be also DOFs at faces; the face DOFs in the boundary of the patch are passed on (let `fnode = mesh->node[FACE]` be the offset for DOFs at faces):

```
child[0]->dof[fnode+3] = el->dof[fnode+1];
child[1]->dof[fnode+3] = el->dof[fnode+0];
```

For the common face of `child[0]` and `child[1]` we have to allocate a new face DOF vector which is located at

```
child[0]->dof[fnode+0] = child[1]->dof[fnode+0]
```

and finally for each interior face of the patch two new face DOF vectors are created and pointers for adjacent children are adjusted:

```
child[0]->dof[fnode+1],
child[0]->dof[fnode+2],
child[1]->dof[fnode+1],
child[1]->dof[fnode+2]
```

Each of these DOF vectors may be shared with another child of a patch element.

If DOFs are located at the barycenter they have to be allocated for both children in 2d and 3d (let `cnode = mesh->node[CENTER]` be the offset for DOFs at the center)

```
child[0]->dof[cnode],
child[1]->dof[cnode].
```

After adding and passing on of DOFs on the patch we can interpolate data from the coarse to the fine grid on the whole patch. This is an operation on the whole patch since new DOFs can be shared by more than one patch element and usually the value(s) of such an DOF should only be calculated once.

All DOF vectors where a pointer to a function `refine_interpol()` in the corresponding `DOF_*_VEC` data structure is provided are interpolated to the fine grid. Such a function does essentially depend on the described passing on and new allocation of DOFs. An abstract description of such functions can be found in Section 1.4.4 and a more detailed one for Lagrange elements in Section 2.5.2.

After such an interpolation, DOFs of higher degree on parent elements may be no longer of interest (if we do not use a multigrid method e.g.) In such a case the entry `preserve_coarse_dofs` in the mesh data structure has to be `0` and all DOFs on the parent that are not handed over to the children will be removed. Following DOFs are removed on the parent for all patch elements (some DOFs are shared by several elements):

```
el->dof[mesh->node[EDGE]+2],
el->dof[mesh->node[CENTER]]
```

in 2d and

```
el->dof[mesh->node[EDGE]+0],
el->dof[mesh->node[FACE]+2],
el->dof[mesh->node[FACE]+3],
el->dof[mesh->node[CENTER]]
```

in 3d. All pointers to DOFs at edges, faces and centers are set to `nil`. No information about these DOFs is available on interior tree elements in this case. Again we want to point out that for the geometrical description of the mesh we do not free vertex DOFs and all pointers to vertex DOFs stay untouched on the parent elements.

This setting of DOF pointers, pointers to children and adjustment of adjacency relations when `NEIGH_IN_EL == 1` is the main part of the refinement module.

If neighbour information is produced by the traversal routines, then it is valid for all tree elements. If it is stored explicitly, then neighbour information is valid for all leaf elements with all neighbours; but it is only valid for interior elements of the tree with those neighbours that belong to the common refinement patch!

2.4.2 The coarsening routines

The coarsening of a mesh is done by the function

```
U_CHAR coarsen(MESH *mesh);
```

Description:

`coarsen(mesh)`: tries to coarsen all leaf element with a *negative* element marker `-mark` times (again, this mark is usually set by an adaptive procedure); the return value is `MESH_COARSENED`, if any element was coarsened, and `0` otherwise.

A second function for coarsening is

```
U_CHAR global_coarsen(MESH *mesh, int mark);
```

Description:

`global_coarsen(mesh, mark)`: sets all element markers for leaf elements of `mesh` to `-mark`; the mesh is then coarsened by `coarsen()`; depending on the actual distribution of coarsening edges on the mesh, this may not result in a `mark` global coarsening; the return value is `coarsen(mesh)` if `mark` is positive, and `0` otherwise.

The function `coarsen()` implements Algorithm 1.1.10. For a marked element, the coarsening patch is collected first. This is done in the same manner as in the refinement procedure. If such a patch can *definitely* not be coarsened (if one element of the patch may not be coarsened, e.g.) all coarsening markers for all patch elements are reseted. If we can not coarsen the patch immediately, because one of the elements has not a common coarsening edge but is allowed to be coarsened more than once, then nothing is done in the moment and we try to coarsen this patch later on (compare Remark 1.1.11).

The coarsening of a patch is the "inverse" of the refinement of a compatible patch. If DOFs of the parents were removed during refinement, these are now added on the parents. Pointers to those that have been passed on to the children are now adjusted back on the parents (see Section 2.4.1 which DOFs of children are now assigned to the parent, just swap the left and right hand sides of the assignments). DOFs that have been freed have to be newly allocated.

If user leaf data is stored at the pointer of `child[1]`, then memory for parent's leaf data is allocated and the children's leaf data is freed. If a function `coarsen_leaf_data` is provided in the `leaf_data_info` structure then leaf data is transformed from children to parent before freeing the children's leaf data.

Like the interpolation of data during refinement, we now can restrict/interpolate data from children to parent. This is done by the `coarse_restrict()` functions for all those DOF vectors where such a function is available in the corresponding `DOF_*_VEC` data structure. Since it does not make sense to both interpolate and restrict data, `coarse_restrict()` may be a pointer to a function either for interpolation or restriction. An abstract description of those functions can be found in Section 1.4.4 and a more detailed one for Lagrange elements in Section 2.5.2.

After these preliminaries the main part of the coarsening can be performed. DOFs that have been created in the refinement step are now freed again, and the children of all patch elements are freed and the pointer to the first child is set to `nil` and the pointer to the second child is adjusted to the `leaf_data` of the parent, or also set to `nil`. Thus, all fine grid information is lost at that moment and it is clear that a restriction of data has to be done in advance.

2.5 Implementation of basis functions

In order to build up a finite element space, we have to specify a set of local basis functions. Together with the corresponding DOF administration and the underlying mesh, the finite element space is given. In the following section we describe the basic data structures for storing basis function information.

In ALBERT Lagrange finite elements up to order four are implemented; they are presented in the subsequent sections.

2.5.1 Data structures for basis functions

For the handling of local basis functions, i.e. a basis of the function space $\bar{\mathbb{P}}$ (compare Section 1.4.2) we use the following data structures:

```
typedef REAL        BAS_FCT(const REAL[DIM+1]);
typedef const REAL  *GRD_BAS_FCT(const REAL[DIM+1]);
typedef const REAL  (*D2_BAS_FCT(const REAL[DIM+1]))[DIM+1];
```

Description:

`BAS_FCT`: the data type for a local finite element function, i.e. a function $\bar{\varphi} \in \bar{\mathbb{P}}$, evaluated at barycentric coordinates $\lambda \in \mathbb{R}^{\mathtt{DIM}+1}$ and its return value $\bar{\varphi}(\lambda)$ is of type `REAL`;

`GRD_BAS_FCT`: the data type for the gradient (with respect to λ) of a local finite element function, i.e. a function returning a pointer to $\nabla_\lambda\bar{\varphi}$ for some function $\bar{\varphi} \in \bar{\mathbb{P}}$:

$$\nabla_\lambda\bar{\varphi}(\lambda) = \left(\frac{\partial\bar{\varphi}(\lambda)}{\partial\lambda_0}, \dots, \frac{\partial\bar{\varphi}(\lambda)}{\partial\lambda_{\mathtt{DIM}}}\right);$$

the argument of such a function are barycentric coordinates and the return value is a pointer to a `const REAL` vector of length `[DIM+1]` storing $\nabla_\lambda\bar{\varphi}(\lambda)$; this vector is overwritten by the next call of the function.

`D2_BAS_FCT`: the data type for the second derivatives (with respect to λ) of a local finite element function, i.e. a function returning a pointer to the matrix $D^2_\lambda\bar{\varphi}$ for some function $\bar{\varphi} \in \bar{\mathbb{P}}$:

$$D^2_\lambda\bar{\varphi} = \begin{pmatrix} \frac{\partial^2\bar{\varphi}(\lambda)}{\partial\lambda_0\partial\lambda_0} & \cdots & \frac{\partial^2\bar{\varphi}(\lambda)}{\partial\lambda_0\partial\lambda_{\mathtt{DIM}}} \\ \vdots & & \vdots \\ \frac{\partial^2\bar{\varphi}(\lambda)}{\partial\lambda_{\mathtt{DIM}}\partial\lambda_0} & \cdots & \frac{\partial^2\bar{\varphi}(\lambda)}{\partial\lambda_{\mathtt{DIM}}\partial\lambda_{\mathtt{DIM}}} \end{pmatrix};$$

The argument of such a function are barycentric coordinates and the return value is a pointer to a $(\mathtt{DIM}+1) \times (\mathtt{DIM}+1)$ matrix storing $D^2_\lambda\bar{\varphi}$; this matrix is overwritten during the next call of the function.

For the implementation of a finite element space, we need a basis of the function space $\bar{\mathbb{P}}$. For such a basis we need the connection of *local* and *global* DOFs on each element (compare Section 1.4.3), information about the interpolation of a given function on an element, and information about interpolation/restriction of finite element functions during refinement/coarsening (compare Section 1.4.4). Such information is stored in the `BAS_FCTS` data structure:

```
typedef struct bas_fcts BAS_FCTS;

struct bas_fcts
```

```
{
  char         *name;       /*  textual description                        */
  int          n_bas_fcts;  /*  number of basis functions on one el        */
  int          degree;      /*  maximal degree of the basis functions      */
  int          n_dof[DIM+1];/*  dofs from these bas_fcts                   */

  void         (*init_element)(const EL_INFO *, const FE_SPACE *, U_CHAR);
  BAS_FCT      **phi;
  GRD_BAS_FCT  **grd_phi;
  D2_BAS_FCT   **D2_phi;

  const DOF    *(*get_dof_indices)(const EL *,const DOF_ADMIN *,int *);
  const S_CHAR *(*get_bound)(const EL_INFO *, S_CHAR *);

/*************** entries must be set for interpolation   ********************/

  const REAL   *(*interpol)(const EL_INFO *, int, const int *,
                            REAL (*)(const REAL_D), REAL *);
  const REAL_D *(*interpol_d)(const EL_INFO *, int, const int *b_no,
                              const REAL *(*)(const REAL_D, REAL_D), REAL_D *);

/********************   optional entries  **********************************/

  const int    *(*get_int_vec)(const EL *, const DOF_INT_VEC *, int *);
  const REAL   *(*get_real_vec)(const EL *, const DOF_REAL_VEC *, REAL *);
  const REAL_D *(*get_real_d_vec)(const EL *, const DOF_REAL_D_VEC *, REAL_D *);
  const U_CHAR *(*get_uchar_vec)(const EL *, const DOF_UCHAR_VEC *, U_CHAR *);
  const S_CHAR *(*get_schar_vec)(const EL *, const DOF_SCHAR_VEC *, S_CHAR *);

  void  (*real_refine_inter)(DOF_REAL_VEC *, RC_LIST_EL *, int);
  void  (*real_coarse_inter)(DOF_REAL_VEC *, RC_LIST_EL *, int);
  void  (*real_coarse_restr)(DOF_REAL_VEC *, RC_LIST_EL *, int);

  void  (*real_d_refine_inter)(DOF_REAL_D_VEC *, RC_LIST_EL *, int);
  void  (*real_d_coarse_inter)(DOF_REAL_D_VEC *, RC_LIST_EL *, int);
  void  (*real_d_coarse_restr)(DOF_REAL_D_VEC *, RC_LIST_EL *, int);
};
```

The entries yield following information:

`name`: string containing a textual description or `nil`;

`n_bas_fcts`: number of local basis functions;

`degree`: maximal polynomial degree of the basis functions; this entry is only used by routines using numerical quadrature where no `QUAD` structure is provided; in such a case via `degree` some default numerical quadrature is chosen (see Section 2.8.1); additionally, `degree` is used by some graphics routines (see Section 2.16.1);

`n_dof`: vector with the count of DOFs for this set of basis functions;

`n_dof[VERTEX,EDGE(,FACE),CENTER]` count of DOFs at the vertices, edges, faces (only 3d), and the center of an element; the corresponding DOF administration of the finite element space uses such information;

`init_element`: is used for the initialization of element dependent local finite element spaces; is not supported by ALBERT in this version;

phi: vector of function pointers for the evaluation of local basis functions in barycentric coordinates;

`(*phi[i])(lambda)` returns the value $\bar\varphi^{\mathtt{i}}(\lambda)$ of the `i`–th basis function at `lambda` for $0 \le \mathtt{i} < \mathtt{n\_bas\_fcts}$;

grd_phi: vector of function pointers for the evaluation of gradients of the basis functions in barycentric coordinates;

`(*grd_phi[i])(lambda)` returns a pointer to a vector of length `DIM+1` containing all first derivatives (with respect to the barycentric coordinates) of the `i`–th basis function at `lambda`, i.e. `(*grd_phi[i])(lambda)[k]`$= \bar\varphi^{\mathtt{i}}_{,\lambda_{\mathtt{k}}}(\lambda)$ for $0 \le \mathtt{k} \le \mathtt{DIM}$, $0 \le \mathtt{i} < \mathtt{n\_bas\_fcts}$; this vector is overwritten on the next call of `(*grd_phi[i])()`;

D2_phi: vector of function pointers for the evaluation of second derivatives of the basis functions in barycentric coordinates;

`(*D2_phi[i])(lambda)` returns a pointer to a `(DIM+1)` $\times$ `(DIM+1)` matrix containing all second derivatives (with respect to the barycentric coordinates) of the `i`–th basis function at `lambda`, i.e. `(*D2_phi[i])(lambda)[k][l]`$= \bar\varphi^{\mathtt{i}}_{,\lambda_{\mathtt{k}}\lambda_{\mathtt{l}}}(\lambda)$ $0 \le \mathtt{k},\mathtt{l} \le \mathtt{DIM}$, $0 \le \mathtt{i} < \mathtt{n\_bas\_fcts}$; this matrix is overwritten on the next call of `(*D2_phi[i])()`;

get_dof_indices: pointer to a function which connects the set of local basis functions with its global DOFs (an implementation of the function j_S in Section 1.4.3);

`get_dof_indices(el, admin, dof)` returns a pointer to a `const DOF` vector of length `n_bas_fcts` where the `i`–th entry is the index of the DOF associated to the `i`–th basis function; arguments are the actual element `el` and the DOF admin `admin` of the corresponding finite element space `fe_space` (these indices depend on all defined DOF admins and thus on the corresponding `admin`); if the last argument `dof` is `nil`, `get_dof_indices()` has to provide memory for storing this vector, which is overwritten on the next call of `get_dof_indices()`; if `dof` is not `nil`, `dof` is a pointer to a vector which has to be filled;

get_bound: pointer to a function which fills a vector with the boundary types of the basis functions;

`get_bound(el_info, bound)` returns a pointer to this vector of length `n_bas_fcts` where the `i`–th entry is the boundary type of the `i`–th basis function; `bound` may be a pointer to a vector which has to be filled (compare the `dof` argument of `get_dof_indices()`);

such a function needs boundary information; thus, all routines using this function on the elements need the `FILL_BOUND` flag during mesh traversal;

When using ALBERT routines for the interpolation of `REAL(_D)` valued functions the `interpol(_d)` function pointer must be set (for example the calculation of Dirichlet boundary values by `dirichlet_bound()` described in Section 2.12.5):

interpol(_d): pointer to a function which performs the local interpolation of a `REAL(_D)` valued function on an element;

`interpol(_d)(el_info, n, indices, f, coeff)` returns a pointer to a `const REAL(_D)` vector with interpolation coefficients of the `REAL(_D)` valued function `f`; if `indices` is a pointer to `nil`, the coefficient for all basis functions are calculated and the `i`–th entry in the vector is the coefficient of the `i`–th basis function; if `indices` is non `nil`, only the coefficients for a subset of the local basis functions has to be calculated; `n` is the number of those basis functions, `indices[0]`,…,`indices[n-1]` are the local indices of the basis functions where the coefficients have to be calculated, and the `i`–th entry in the return vector is then the coefficient of the `indices[i]`–th basis function; `coeff` may be a pointer to a vector which has to be filled (compare the `dof` argument of `get_dof_indices()`);

such a function usually needs vertex coordinate information; thus, all routines using this function on the elements need the `FILL_COORDS` flag during mesh traversal.

Optional entries:

`get_*_vec`: pointer to a function which fills a local vector with values of a `DOF_*_VEC` at the DOFs of the basis functions;

`get_*_vec(el, dof_*_vec, * values)` returns a pointer to a `const *` vector where the `i`–th entry is the value of `dof_*_vec` at the DOF of the `i`–th basis function on element `el`; `values` may be a pointer to a vector which has to be filled (compare the `dof` argument of `get_dof_indices()`);

Since the interpolation of finite element functions during refinement and coarsening, and the restriction of functionals during coarsening strongly depend on the basis functions and its DOFs (compare Section 1.4.4), pointers for functions which perform those operations can be stored at the following entries:

`real(_d)_refine_inter`: pointer to a function for interpolating a `REAL(_D)` valued function during refinement;

`real(_d)_refine_inter(vec, refine_list, n)` interpolates the `DOF_REAL(_D)_VEC` vector `vec` on the refinement patch onto the finer grid; information about all parents of the refinement patch is accessible in the vector `refine_list` of length `n`;

`real(_d)_coarse_inter`: pointer to a function for interpolating a `REAL(_D)` valued function during coarsening;

`real(_d)_coarse_inter(vec, refine_list, n)` interpolates the `DOF_REAL(_D)_VEC` vector `vec` on the coarsening patch onto the coarser grid; information about all parents of the refinement patch is accessible in the vector `refine_list` of length `n`;

`real(_d)_coarse_restr`: pointer to a function for restriction of `REAL(_D)` valued linear functionals during coarsening;

`real(_d)_coarse_restrict(vec, refine_list, n)` restricts the `DOF_REAL(_D)_VEC` vector `vec` on the coarsening patch onto the coarser grid; information about all parents of the refinement patch is accessible in the vector `refine_list` of length `n`;

In Section 2.5.3 and 2.5.4 examples for the implementation of those functions are given.

2.5.1. Remark. The access of local element vectors via the `get_*_vec()` routines can also be done in a standard way by using the `get_dof_indices()` function which must be supplied; if some of the `get_*_vec()` are pointer to `nil`, ALBERT fills in pointers to some standard functions using `get_dof_indices()`. But a specialized function may be faster. An example of such a standard routine is:

```
static const int *get_int_vec(const EL *el, const DOF_INT_VEC *vec, int *ivec)
{
  FUNCNAME("get_int_vec");
  int             i, n_bas_fcts;
  const DOF       *index;
  int             *v = nil, *rvec;
  const FE_SPACE  *fe_space = vec->fe_space;
  static int      *local_vec = nil;
  static int      max_size = 0;

  GET_DOF_VEC(v, vec);
  n_bas_fcts = fe_space->bas_fcts->n_bas_fcts;
```

```
    if (ivec)
    {
      rvec = ivec;
    }
    else
    {
      if (max_size < n_bas_fcts)
      {
        local_vec = MEM_REALLOC(local_vec, max_size, n_bas_fcts, int);
        max_size = n_bas_fcts;
      }
      rvec = local_vec;
    }
    index = fe_space->bas_fcts->get_dof_indices(el, fe_space->admin, nil);

    for (i = 0; i < n_bas_fcts; i++)
      rvec[i] = v[index[i]];

    return((const int *) rvec);
  }
```

A specialized implementation for linear finite elements e.g. is more efficient:

```
const int *get_int_vec(const EL *el, const DOF_INT_VEC *vec, int *ivec)
{
  FUNCNAME("get_int_vec");
  int             i, n0;
  static int      local_vec[N_VERTICES];
  int             *v = vec->vec, *rvec = ivec ? ivec : local_vec;
  DOF             **dof = el->dof;

  n0 = vec->fe_space->admin->n0_dof[VERTEX];

  for (i = 0; i < N_VERTICES; i++)
    rvec[i] = v[dof[i][n0]];

  return((const int *) rvec);
}
```

Any kind of basis functions can be implemented by filling the above described structure for basis functions. All non optional entries have to be defined. Since in the functions for reading and writing of meshes, the basis functions are identified by their names, all used basis functions have to be registered before using these functions (compare Section 2.3.8). All Lagrange finite elements described below are already registered, with names "lagrange1" to "lagrange4"; newly defined basis functions must use different names.

```
int new_bas_fcts(const BAS_FCTS * bas_fcts);
```

Description:

new_bas_fcts(bas_fcts): puts the new set of basis functions `bas_fcts` to an internal list of all used basis functions; different sets of basis functions are identified by their `name`; thus, the member `name` of `bas_fcts` must be a string with positive length holding a description; if an existing set of basis functions with the same name is found, the program stops with an error;

if the entries `phi`, `grd_phi`, `get_dof_indices`, and `get_bound` are not set, this also result in an error and program stops.

Basis functions can be accessed from that list by

```
const BAS_FCTS *get_bas_fcts(const char *name)
```

Description:

`get_bas_fcts(name)`: looks for a set of basis functions with name `name` in the internal list of all registered basis functions; if such a set is found, the return value is a pointer to the corresponding `BAS_FCTS` structure, otherwise the return value is `nil`.

Lagrange elements can be accessed more efficiently by a call of `get_lagrange()`, see Section 2.5.7.

2.5.2 Lagrange finite elements

ALBERT supports Lagrange finite elements up to order four which are described in the following sections. Lagrange finite elements are given by $\bar{\mathbb{P}} = \mathbb{P}_p(\bar{S})$ (polynomials of degree $p \in \mathbb{N}$ on $\bar{S}$) and they are globally continuous. They are uniquely determined by the values at the associated Lagrange nodes $\{x_i\}$. The Lagrange basis functions $\{\phi_i\}$ satisfy

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

Now, consider the basis functions $\{\bar{\varphi}^i\}_{i=1}^m$ of $\bar{\mathbb{P}}$ with the associated Lagrange nodes $\{\lambda_i\}_{i=1}^m$ given in barycentric coordinates:

$$\bar{\varphi}^i(\lambda_j) = \delta_{ij} \qquad \text{for } i, j = 1, \dots, m.$$

Basis functions are located at the vertices, edges, (faces,) or at the center of an element. The corresponding DOF is a vertex, edge, (face,) or center DOF, respectively. The boundary type of a basis function is the boundary type of the associated vertex or edge (or face). Basis functions at the center are always `INTERIOR`. Such boundary information is filled by the `get_bound()` function in the `BAS_FCTS` structure and is straight forward.

The interpolation coefficient for a function f for basis function $\bar{\varphi}^i$ on element S is the value of f at the Lagrange node: $f(x(\lambda_i))$. These coefficients are calculated by the `interpol(_d)()` function in the `BAS_FCTS` structure. Examples for both functions are given below for linear finite elements.

2.5.3 Linear finite elements

Linear finite elements are uniquely defined by their values at the vertices of the triangulation. On each element we have `N_VERTICES` basis functions which are the barycentric coordinates of the element. Thus, in 2d we have three and in 3d four basis functions for Lagrange elements of first order; the basis functions and the corresponding Lagrange nodes in barycentric coordinates are shown in Table 2.3 and Table 2.4. The calculation of derivatives is straight forward. The global

function	position	Lagrange node
$\bar{\varphi}^0(\lambda) = \lambda_0$	vertex 0	$\lambda^0 = (1,0,0)$
$\bar{\varphi}^1(\lambda) = \lambda_1$	vertex 1	$\lambda^1 = (0,1,0)$
$\bar{\varphi}^2(\lambda) = \lambda_2$	vertex 2	$\lambda^2 = (0,0,1)$

Table 2.3: Local basis functions for linear finite elements in 2d.

DOF index of the `i`–th basis functions on element `el` is stored for linear finite elements at

```
el->dof[i][admin->n0_dof[VERTEX]]
```

function	position	Lagrange node
$\bar\varphi^0(\lambda) = \lambda_0$	vertex 0	$\lambda^0 = (1,0,0,0)$
$\bar\varphi^1(\lambda) = \lambda_1$	vertex 1	$\lambda^1 = (0,1,0,0)$
$\bar\varphi^2(\lambda) = \lambda_2$	vertex 2	$\lambda^2 = (0,0,1,0)$
$\bar\varphi^3(\lambda) = \lambda_3$	vertex 3	$\lambda^3 = (0,0,0,1)$

Table 2.4: Local basis functions for linear finite elements in 3d.

Setting `nv = admin->n0_dof[VERTEX]` the associated DOFs are shown in Figure 2.5.

For linear finite elements we want to give examples for the implementation of some routines in the corresponding `BAS_FCTS` structure.

2.5.2. Example. The implementation of `get_dof_indices()` can be done in the following way (compare Figure 2.5):

```
const DOF *get_dof_indices(const EL *el, const DOF_ADMIN *admin, DOF *idof)
{
  FUNCNAME("get_dof_indices");
  static DOF  index_vec[N_VERTICES];
  DOF         *rvec = idof ? idof : index_vec;
  int         i, n0 = admin->n0_dof[VERTEX];
  DOF         **dof = el->dof;

  for (i = 0; i < N_VERTICES; i++)
    rvec[i] = dof[i][n0];

  return((const DOF *) rvec);
}
```

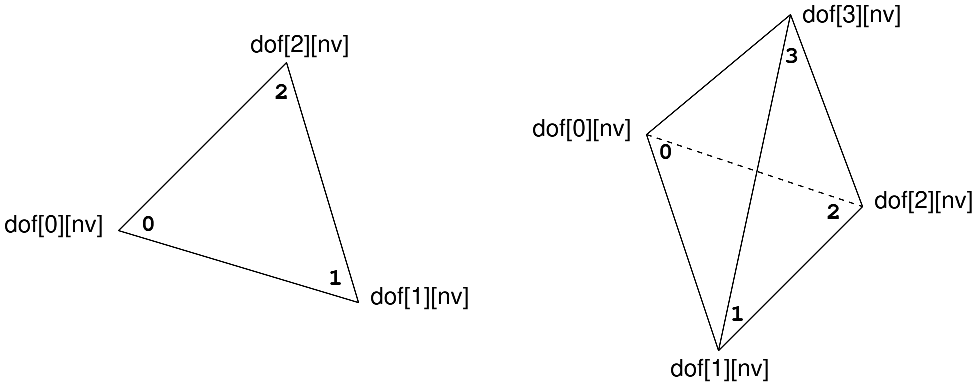


Figure 2.5: DOFs and local numbering of the basis functions for linear elements.

2.5.3. Example (Accessing boundary types for linear f.e.). The `get_bound()` function fills the `bound` vector with the boundary type of the vertices:

```
const S_CHAR *get_bound(const EL_INFO *el_info, S_CHAR *bound)
{
  FUNCNAME("get_bound");
  static S_CHAR  bound_vec[N_VERTICES];
  S_CHAR         *rvec = bound ? bound : bound_vec;
  int            i;

  TEST_FLAG(FILL_BOUND, el_info);

  for (i = 0; i < N_VERTICES; i++)
    rvec[i] = el_info->bound[i];

  return((const S_CHAR *) rvec);
}
```

2.5.4. Example (Interpolation for linear f.e.). For the interpolation `interpol()` routine we have to evaluate the given function at the vertices. Thus, interpolation can be implemented as follows:

```
const REAL *interpol(const EL_INFO *el_info, int n, const int *indices,
                     REAL (*f)(const REAL_D), REAL *vec)
{
  FUNCNAME("interpol");
  static REAL  inter[N_VERTICES];
  REAL         *rvec = vec ? vec : inter;
  int          i;

  TEST_FLAG(FILL_COORDS, el_info);

  if (indices)
  {
    if (n <= 0 || n > N_VERTICES)
    {
      MSG("something is wrong, doing nothing\n");
      rvec[0] = 0.0;
      return((const REAL *) rvec);
    }
    for (i = 0; i < n; i++)
      rvec[i] = f(el_info->coord[indices[i]]);
  }
  else
  {
    for (i = 0; i < N_VERTICES; i++)
      rvec[i] = f(el_info->coord[i]);
  }
  return((const REAL *) rvec);
}
```

An example of the optional `get_*_vec()` functions is shown above for `get_int_vec()` in Section 2.5.1.

2.5.5. Example. The implementation of functions for interpolation during refinement and restriction of linear functionals during coarsening is very simple for linear elements; we do not have to loop over the refinement patch since only the vertices at the refinement/coarsening edge and the new DOF at the midpoint are involved in this process. No interpolation during coarsening has to be done since all values at the remaining vertices stay the same; no function has to be defined.

```
void real_refine_inter(DOF_REAL_VEC *drv, RC_LIST_EL *list, int n)
{
  FUNCNAME("real_refine_inter");
  EL      *el;
  REAL    *vec = nil;
  DOF     dof_new, dof0, dof1;
  int     n0;

  if (n < 1) return;
  GET_DOF_VEC(vec, drv);
  n0 = drv->fe_space->admin->n0_dof[VERTEX];
  el = list->el;
  dof0 = el->dof[0][n0];           /* 1st endpoint of refinement edge */
  dof1 = el->dof[1][n0];           /* 2nd endpoint of refinement edge */
  dof_new = el->child[0]->dof[DIM][n0];  /*      newest vertex is DIM */
  vec[dof_new] = 0.5*(vec[dof0] + vec[dof1]);

  return;
}


void real_coarse_restr(DOF_REAL_VEC *drv, RC_LIST_EL *list, int n)
{
  FUNCNAME("real_coarse_restr");
  EL      *el;
  REAL    *vec = nil;
  DOF     dof_new, dof0, dof1;
  int     n0;

  if (n < 1) return;
  GET_DOF_VEC(vec, drv);
  n0 = drv->fe_space->admin->n0_dof[VERTEX];
  el = list->el;
  dof0 = el->dof[0][n0];           /* 1st endpoint of refinement edge */
  dof1 = el->dof[1][n0];           /* 2nd endpoint of refinement edge */
  dof_new = el->child[0]->dof[DIM][n0];  /*      newest vertex is DIM */
  vec[dof0] += 0.5*vec[dof_new];
  vec[dof1] += 0.5*vec[dof_new];

  return;
}
```

2.5.4 Quadratic finite elements

Quadratic finite elements are uniquely defined by their values at the vertices and the edges' midpoints of the triangulation. In 2d we have six and in 3d we have ten basis functions for Lagrange elements of second order; the basis functions and the corresponding Lagrange nodes in barycentric coordinates are shown in Table 2.5 and Table 2.6.

function	position	Lagrange node
$\bar\varphi^0(\lambda) = \lambda_0(2\lambda_0 - 1)$	vertex 0	$\lambda^0 = (1,0,0)$
$\bar\varphi^1(\lambda) = \lambda_1(2\lambda_1 - 1)$	vertex 1	$\lambda^1 = (0,1,0)$
$\bar\varphi^2(\lambda) = \lambda_2(2\lambda_2 - 1)$	vertex 2	$\lambda^2 = (0,0,1)$
$\bar\varphi^3(\lambda) = 4\lambda_1\,\lambda_2$	edge 0	$\lambda^3 = (0,\frac{1}{2},\frac{1}{2})$
$\bar\varphi^4(\lambda) = 4\lambda_2\,\lambda_0$	edge 1	$\lambda^4 = (\frac{1}{2},0,\frac{1}{2})$
$\bar\varphi^5(\lambda) = 4\lambda_0\,\lambda_1$	edge 2	$\lambda^5 = (\frac{1}{2},\frac{1}{2},0)$

Table 2.5: Local basis functions for quadratic finite elements in 2d.

function	position	Lagrange node
$\bar\varphi^0(\lambda) = \lambda_0(2\lambda_0 - 1)$	vertex 0	$\lambda^0 = (1,0,0,0)$
$\bar\varphi^1(\lambda) = \lambda_1(2\lambda_1 - 1)$	vertex 1	$\lambda^1 = (0,1,0,0)$
$\bar\varphi^2(\lambda) = \lambda_2(2\lambda_2 - 1)$	vertex 2	$\lambda^2 = (0,0,1,0)$
$\bar\varphi^3(\lambda) = \lambda_3(2\lambda_3 - 1)$	vertex 3	$\lambda^3 = (0,0,0,1)$
$\bar\varphi^4(\lambda) = 4\lambda_0\,\lambda_1$	edge 0	$\lambda^4 = (\frac{1}{2},\frac{1}{2},0,0)$
$\bar\varphi^5(\lambda) = 4\lambda_0\,\lambda_2$	edge 1	$\lambda^5 = (\frac{1}{2},0,\frac{1}{2},0)$
$\bar\varphi^6(\lambda) = 4\lambda_0\,\lambda_3$	edge 2	$\lambda^6 = (\frac{1}{2},0,0,\frac{1}{2})$
$\bar\varphi^7(\lambda) = 4\lambda_1\,\lambda_2$	edge 3	$\lambda^7 = (0,\frac{1}{2},\frac{1}{2},0)$
$\bar\varphi^8(\lambda) = 4\lambda_1\,\lambda_3$	edge 4	$\lambda^8 = (0,\frac{1}{2},0,\frac{1}{2})$
$\bar\varphi^9(\lambda) = 4\lambda_2\,\lambda_3$	edge 5	$\lambda^9 = (0,0,\frac{1}{2},\frac{1}{2})$

Table 2.6: Local basis functions for quadratic finite elements in 3d.

The associated DOFs for basis functions at vertices/edges are located at the vertices/edges of the element; the entry in the vector of DOF indices at the vertices/edges is determined by the vertex/edge offset in the corresponding `admin` of the finite element space: the DOF index of the `i`–th basis functions on element `el` is

```
el->dof[i][admin->n0_dof[VERTEX]]
```

for `i = 0, ... ,N_VERTICES-1` and

```
el->dof[i][admin->n0_dof[EDGE]]
```

for `i = N_VERTICES, ... ,N_VERTICES+N_EDGES-1`. Here we used the fact, that for quadratic elements DOFs are located at the vertices and the edges on the mesh. Thus, regardless of any other set of DOFs, the offset `mesh->node[VERTEX]` is zero and `mesh->node[EDGE]` is `N_VERTICES`.

Setting `nv = admin->n0_dof[VERTEX]` and `ne = admin->n0_dof[EDGE]`, the associated DOFs are shown in Figure 2.6.

2.5.6. Example. The function `get_dof_indices()` for quadratic finite elements can be implemented by (compare Figure 2.6):

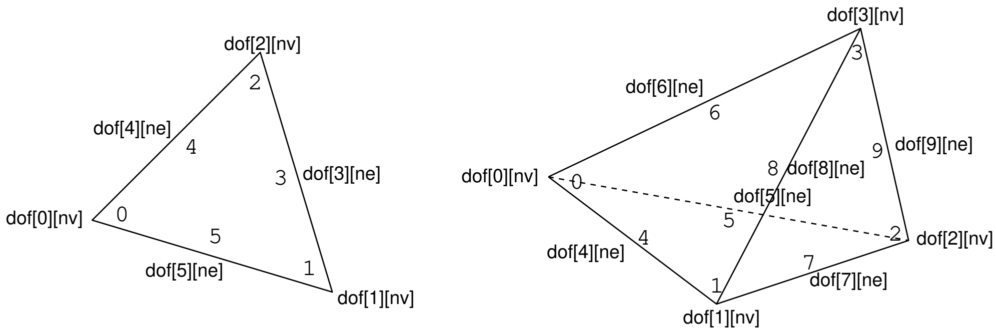


Figure 2.6: DOFs and local numbering of the basis functions for quadratic elements.

```
const DOF *get_dof_indices(const EL *el, const DOF_ADMIN *admin, DOF *idof)
{
  FUNCNAME("get_dof_indices");
  static DOF  index_vec[N_VERTICES+N_EDGES];
  int         i, n0;
  DOF         *rvec = idof ? idof : index_vec;
  DOF         **dof = el->dof;

  n0 = admin->n0_dof[VERTEX];
  for (i = 0; i < N_VERTICES; i++)
    rvec[i] = dof[i][n0];
  n0 = admin->n0_dof[EDGE];
  for (i = N_VERTICES; i < N_VERTICES+N_EDGES; i++)
    rvec[i] = dof[i][n0];

  return((const DOF *) rvec);
}
```

The boundary type of a basis functions at a vertex is the the boundary type of the vertex, and the boundary type of a basis function at an edge is the boundary type of the edge. The i–th interpolation coefficient of a function f on element S is just $f(x(\lambda_{\mathtt{i}}))$. The implementation is similar to that for linear finite elements and is not shown here.

The implementation of functions for interpolation during refinement and coarsening and the restriction during coarsening becomes more complicated and differs between the dimensions. Here we have to set values for all elements of the refinement patch. The interpolation during coarsening in not trivial anymore. As an example of such implementations we present the interpolation during refinement for 2d and 3d.

2.5.7. Example. In 2d we have to set values for the new vertex in the refinement edge, and for the two midpoints of the bisected edge. Then we have to set the value for the midpoint of the common edge of the two children of the bisected triangle and we have to set the corresponding value on the neighbor in the case that the refinement edge is not a boundary edge:

```
void real_refine_inter(DOF_REAL_VEC *drv, RC_LIST_EL *list, int n)
{
  FUNCNAME("real_refine_inter");
  EL              *el;
  REAL            *vec = nil;
  int             node, n0;
```

```
  DOF             cdof;
  const DOF       *pdof;
  const DOF       *(*get_dof_indices)(const EL *, const DOF_ADMIN *, DOF *);
  const DOF_ADMIN *admin;

  if (n < 1) return;
  el = list->el;
  GET_DOF_VEC(vec, drv);
  if (!drv->fe_space)
  {
    ERROR("no fe_space in dof_real_vec %s\n", NAME(drv));
    return;
  }
  else if (!drv->fe_space->bas_fcts)
  {
    ERROR("no basis functions in fe_space %s\n", NAME(drv->fe_space));
    return;
  }
  get_dof_indices = drv->fe_space->bas_fcts->get_dof_indices;
  admin = drv->fe_space->admin;

  pdof = get_dof_indices(el, admin, nil);
  node = drv->fe_space->mesh->node[VERTEX];
  n0 = admin->n0_dof[VERTEX];
/****************************************************************************/
/*  newest vertex of child[0] and child[1]                                  */
/****************************************************************************/
  cdof = el->child[0]->dof[node+DIM][n0];  /*      newest vertex is DIM */
  vec[cdof] = vec[pdof[5]];

  node = drv->fe_space->mesh->node[EDGE];
  n0 = admin->n0_dof[EDGE];
/****************************************************************************/
/*  midpoint of edge on child[0] at the refinement edge                     */
/****************************************************************************/
  cdof = el->child[0]->dof[node][n0];
  vec[cdof] = 0.375*vec[pdof[0]] - 0.125*vec[pdof[1]]  + 0.75*vec[pdof[5]];
/****************************************************************************/
/* node in the common edge of child[0] and child[1]                         */
/****************************************************************************/
  cdof = el->child[0]->dof[node+1][n0];
  vec[cdof] = -0.125*(vec[pdof[0]] + vec[pdof[1]]) + 0.25*vec[pdof[5]]
              + 0.5*(vec[pdof[3]] + vec[pdof[4]]);
/****************************************************************************/
/*  midpoint of edge on child[1] at the refinement edge                     */
/****************************************************************************/
  cdof = el->child[1]->dof[node+1][n0];
  vec[cdof] = -0.125*vec[pdof[0]] + 0.375*vec[pdof[1]]  + 0.75*vec[pdof[5]];

  if (n > 1)
  {
/****************************************************************************/
/*  adjust the value at the midpoint of the common edge of neigh's children */
```

```
/****************************************************************************/
    el = list[1].el;
    pdof = get_dof_indices(el, admin, nil);
    cdof = el->child[0]->dof[node+1][n0];
    vec[cdof] = -0.125*(vec[pdof[0]] + vec[pdof[1]]) + 0.25*vec[pdof[5]]
                + 0.5*(vec[pdof[3]] + vec[pdof[4]]);
  }
  return;
}
```

2.5.8. Example. In 3d we first have to set values for all DOFs that belong to the first element of the refinement patch. Then we have to loop over the refinement patch and set all DOFs that have not previously been set on another patch element. In order to set values only once, by the variable `lr_set` we check, if a common DOFs with a left or right neighbor is set by the neighbor. Such values are already set if the neighbor is a prior element in the list. Since all values are set on the first element for all subsequent elements there must be DOFs which have been set by another element.

```
void real_refine_inter(DOF_REAL_VEC *drv, RC_LIST_EL *list, int n)
{
  FUNCNAME("real_refine_inter");
  EL              *el;
  REAL            *v = nil;
  const DOF       *cdof;
  DOF             pdof[10], cdofi;
  int             i, lr_set;
  int             node0, n0;
  const DOF       *(*get_dof_indices)(const EL *, const DOF_ADMIN *, DOF *);
  const DOF_ADMIN *admin;

  if (n < 1) return;
  el = list->el;

  GET_DOF_VEC(v, drv);
  if (!drv->fe_space)
  {
    ERROR("no fe_space in dof_real_vec %s\n", NAME(drv));
    return;
  }
  else if (!drv->fe_space->bas_fcts)
  {
    ERROR("no basis functions in fe_space %s\n", NAME(drv->fe_space));
    return;
  }
  get_dof_indices = drv->fe_space->bas_fcts->get_dof_indices;
  admin = drv->fe_space->admin;

  get_dof_indices(el, admin, pdof);
  node0 = drv->fe_space->mesh->node[EDGE];
  n0 = admin->n0_dof[EDGE];
/****************************************************************************/
/*  values on child[0]                                                      */
/****************************************************************************/
```

```
  cdof = get_dof_indices(el->child[0], admin, nil);
  v[cdof[3]] = (v[pdof[4]]);
  v[cdof[6]] = (0.375*v[pdof[0]] - 0.125*v[pdof[1]] + 0.75*v[pdof[4]]);
  v[cdof[8]] = (0.125*(-v[pdof[0]] - v[pdof[1]]) + 0.25*v[pdof[4]]
              + 0.5*(v[pdof[5]] + v[pdof[7]]));
  v[cdof[9]] = (0.125*(-v[pdof[0]] - v[pdof[1]]) + 0.25*v[pdof[4]]
              + 0.5*(v[pdof[6]] + v[pdof[8]]));
/****************************************************************************/
/*  values on child[1]                                                      */
/****************************************************************************/
  cdofi = el->child[1]->dof[node0+2][n0];
  v[cdofi] = (-0.125*v[pdof[0]] + 0.375*v[pdof[1]]
            + 0.75*v[pdof[4]]);
/****************************************************************************/
/*   adjust neighbour values                                                */
/****************************************************************************/
  for (i = 1; i < n; i++)
  {
    el = list[i].el;
    get_dof_indices(el, admin, pdof);

    lr_set = 0;
    if (list[i].neigh[0]  &&  list[i].neigh[0]->no < i)  lr_set = 1;
    if (list[i].neigh[1]  &&  list[i].neigh[1]->no < i)  lr_set += 2;
    TEST_EXIT(lr_set)("no values set on both neighbours\n");
/****************************************************************************/
/*  values on child[0]                                                      */
/****************************************************************************/
    switch (lr_set)
    {
    case 1:
      cdofi = el->child[0]->dof[node0+4][n0];
      v[cdofi] = (0.125*(-v[pdof[0]] - v[pdof[1]]) + 0.25*v[pdof[4]]
                + 0.5*(v[pdof[5]] + v[pdof[7]]));
      break;
    case 2:
      cdofi = el->child[0]->dof[node0+5][n0];
      v[cdofi] = (0.125*(-v[pdof[0]] - v[pdof[1]]) + 0.25*v[pdof[4]]
                + 0.5*(v[pdof[6]] + v[pdof[8]]));
    }
  }
  return;
}
```

2.5.5 Cubic finite elements

For Lagrange elements of third order we have ten basis functions in 2d and 20 in 3d; the basis functions and the corresponding Lagrange nodes in barycentric coordinates are shown in Table 2.7 and Table 2.8.

For cubic elements we have to face a further difficulty. At each edge two basis functions are located. The two DOFs of the i–th edge are subsequent entries in the vector `el->dof[i]`. For two neighboring triangles the common edge has a different orientation with respect to the local

function	position	Lagrange node
$\bar\varphi^0(\lambda) = \frac{1}{2}(3\lambda_0-1)(3\lambda_0-2)\lambda_0$	vertex 0	$\lambda^0 = (1,0,0)$
$\bar\varphi^1(\lambda) = \frac{1}{2}(3\lambda_1-1)(3\lambda_1-2)\lambda_1$	vertex 1	$\lambda^1 = (0,1,0)$
$\bar\varphi^2(\lambda) = \frac{1}{2}(3\lambda_2-1)(3\lambda_2-2)\lambda_2$	vertex 2	$\lambda^2 = (0,0,1)$
$\bar\varphi^3(\lambda) = \frac{9}{2}(3\lambda_1-1)\lambda_1\lambda_2$	edge 0	$\lambda^3 = (0,\frac{2}{3},\frac{1}{3})$
$\bar\varphi^4(\lambda) = \frac{9}{2}(3\lambda_2-1)\lambda_2\lambda_1$	edge 0	$\lambda^4 = (0,\frac{1}{3},\frac{2}{3})$
$\bar\varphi^5(\lambda) = \frac{9}{2}(3\lambda_2-1)\lambda_2\lambda_0$	edge 1	$\lambda^5 = (\frac{1}{3},0,\frac{2}{3})$
$\bar\varphi^6(\lambda) = \frac{9}{2}(3\lambda_0-1)\lambda_0\lambda_2$	edge 1	$\lambda^6 = (\frac{2}{3},0,\frac{1}{3})$
$\bar\varphi^7(\lambda) = \frac{9}{2}(3\lambda_0-1)\lambda_0\lambda_1$	edge 2	$\lambda^7 = (\frac{2}{3},\frac{1}{3},0)$
$\bar\varphi^8(\lambda) = \frac{9}{2}(3\lambda_1-1)\lambda_1\lambda_0$	edge 2	$\lambda^8 = (\frac{1}{3},\frac{2}{3},0)$
$\bar\varphi^9(\lambda) = 27\lambda_0\lambda_1\lambda_2$	center	$\lambda^9 = (\frac{1}{3},\frac{1}{3},\frac{1}{3})$

Table 2.7: Local basis functions for cubic finite elements in 2d.

function	position	Lagrange node
$\bar\varphi^0(\lambda) = \frac{1}{2}(3\lambda^0-1)(3\lambda^0-2)\lambda^0$	vertex 0	$\lambda^0 = (1,0,0,0)$
$\bar\varphi^1(\lambda) = \frac{1}{2}(3\lambda^1-1)(3\lambda^1-2)\lambda^1$	vertex 1	$\lambda^1 = (0,1,0,0)$
$\bar\varphi^2(\lambda) = \frac{1}{2}(3\lambda^2-1)(3\lambda^2-2)\lambda^2$	vertex 2	$\lambda^2 = (0,0,1,0)$
$\bar\varphi^3(\lambda) = \frac{1}{2}(3\lambda^3-1)(3\lambda^3-2)\lambda^3$	vertex 3	$\lambda^3 = (0,0,0,1)$
$\bar\varphi^4(\lambda) = \frac{9}{2}(3\lambda^0-1)\lambda^0\lambda^1$	edge 0	$\lambda^4 = (\frac{2}{3},\frac{1}{3},0,0)$
$\bar\varphi^5(\lambda) = \frac{9}{2}(3\lambda^1-1)\lambda^1\lambda^0$	edge 0	$\lambda^5 = (\frac{1}{3},\frac{2}{3},0,0)$
$\bar\varphi^6(\lambda) = \frac{9}{2}(3\lambda^0-1)\lambda^0\lambda^2$	edge 1	$\lambda^6 = (\frac{2}{3},0,\frac{1}{3},0)$
$\bar\varphi^7(\lambda) = \frac{9}{2}(3\lambda^2-1)\lambda^2\lambda^0$	edge 1	$\lambda^7 = (\frac{1}{3},0,\frac{2}{3},0)$
$\bar\varphi^8(\lambda) = \frac{9}{2}(3\lambda^0-1)\lambda^0\lambda^3$	edge 2	$\lambda^8 = (\frac{2}{3},0,0,\frac{1}{3})$
$\bar\varphi^9(\lambda) = \frac{9}{2}(3\lambda^3-1)\lambda^3\lambda^0$	edge 2	$\lambda^9 = (\frac{1}{3},0,0,\frac{2}{3})$
$\bar\varphi^{10}(\lambda) = \frac{9}{2}(3\lambda^1-1)\lambda^1\lambda^2$	edge 3	$\lambda^{10} = (0,\frac{2}{3},\frac{1}{3},0)$
$\bar\varphi^{11}(\lambda) = \frac{9}{2}(3\lambda^2-1)\lambda^2\lambda^1$	edge 3	$\lambda^{11} = (0,\frac{1}{3},\frac{2}{3},0)$
$\bar\varphi^{12}(\lambda) = \frac{9}{2}(3\lambda^1-1)\lambda^1\lambda^3$	edge 4	$\lambda^{12} = (0,\frac{2}{3},0,\frac{1}{3})$
$\bar\varphi^{13}(\lambda) = \frac{9}{2}(3\lambda^3-1)\lambda^3\lambda^1$	edge 4	$\lambda^{13} = (0,\frac{1}{3},0,\frac{2}{3})$
$\bar\varphi^{14}(\lambda) = \frac{9}{2}(3\lambda^2-1)\lambda^2\lambda^3$	edge 5	$\lambda^{14} = (0,0,\frac{2}{3},\frac{1}{3})$
$\bar\varphi^{15}(\lambda) = \frac{9}{2}(3\lambda^3-1)\lambda^3\lambda^2$	edge 5	$\lambda^{15} = (0,0,\frac{1}{3},\frac{2}{3})$
$\bar\varphi^{16}(\lambda) = 27\lambda^1\lambda^2\lambda^3$	face 0	$\lambda^{16} = (0,\frac{1}{3},\frac{1}{3},\frac{1}{3})$
$\bar\varphi^{17}(\lambda) = 27\lambda^2\lambda^3\lambda^0$	face 1	$\lambda^{17} = (\frac{1}{3},0,\frac{1}{3},\frac{1}{3})$
$\bar\varphi^{18}(\lambda) = 27\lambda^3\lambda^0\lambda^1$	face 2	$\lambda^{18} = (\frac{1}{3},\frac{1}{3},0,\frac{1}{3})$
$\bar\varphi^{19}(\lambda) = 27\lambda^0\lambda^1\lambda^2$	face 3	$\lambda^{19} = (\frac{1}{3},\frac{1}{3},\frac{1}{3},0)$

Table 2.8: Local basis functions for cubic finite elements in 3d.

numbering of vertices on the two triangles. In Figure 2.7 the 3rd local basis function on the left and the 4th on the right triangle built up the global basis function, e.g.; thus, both local basis function must have access to the same global DOF.

In order to combine the global DOF with the local basis function in the implementation of the `get_dof_indices()` function, we have to give every edge a *global orientation*, i.e, every edge has a unique beginning and end point. Using the orientation of an edge we are able to order the DOFs stored at this edge. Let for example the common edge in Figure 2.7 be oriented from bottom to top. The global DOF corresponding to 3rd local DOF on the left and the 4th local DOF on the right is then

```
el->dof[N_VERTICES+0][admin->n0_dof[EDGE]]
```

and for the 4th local DOF on the left and 3rd local DOF on the right

```
el->dof[N_VERTICES+0][admin->n0_dof[EDGE]+1]
```

The global orientation gives a unique access of local DOFs from global ones.

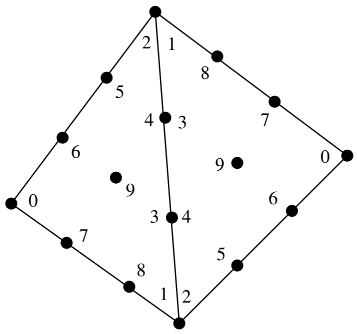


Figure 2.7: Cubic DOFs on a patch of two triangles.

2.5.9. Example. For the implementation for cubics, we use in 2d as well as in 3d an orientation defined by the DOF indices at the edges' vertices. The vertex with the smaller (global) DOF index is the beginning point, the vertex with the higher index the end point. For cubics the implementation differs between 2d and 3d. In 2d we have one degree of freedom at the center and in 3d on degree of freedom at each face and no one at the center.

We only show the implementation for 2d; the implementation for 3d is similar. Accessing the DOFs at an edge, we order them according to the orientation of the edge.

```
const DOF *get_dof_indices(const EL *el, const DOF_ADMIN *admin, DOF *idof)
{
  FUNCNAME("get_dof_indices");
  static DOF  index_vec[N_VERTICES+2*N_EDGES+1];
  DOF         *rvec = idof ? idof : index_vec;
  int         i, j, n0;
  DOF         **dof = el->dof;

  n0 = admin->n0_dof[VERTEX];
  for (i = 0; i < N_VERTICES; i++)
    rvec[i] = dof[i][n0];
  n0 = admin->n0_dof[EDGE];
  for (i = 0, j = N_VERTICES; i < N_EDGES; i++)
  {
    if (dof[vertex_of_edge[i][0]][0] < dof[vertex_of_edge[i][1]][0])
```

```
      {
        rvec[j++] = dof[N_VERTICES+i][n0];
        rvec[j++] = dof[N_VERTICES+i][n0+1];
      }
      else
      {
        rvec[j++] = dof[N_VERTICES+i][n0+1];
        rvec[j++] = dof[N_VERTICES+i][n0];
      }
    }
    n0 = admin->n0_dof[CENTER];
    rvec[9] = dof[6][n0];

    return((const DOF *) rvec);
  }
```

2.5.6 Quartic finite elements

For Lagrange elements of fourth order we have 15 basis functions in 2d and 35 in 3d; the basis functions and the corresponding Lagrange nodes in barycentric coordinates are shown in Table 2.9 and Table 2.10.

function	position	Lagrange node
$\bar\varphi^0(\lambda) = \frac{1}{3}(4\lambda_0-1)(2\lambda_0-1)(4\lambda_0-3)\lambda_0$	vertex 0	$\lambda^0 = (1,0,0)$
$\bar\varphi^1(\lambda) = \frac{1}{3}(4\lambda_1-1)(2\lambda_1-1)(4\lambda_1-3)\lambda_1$	vertex 1	$\lambda^1 = (0,1,0)$
$\bar\varphi^2(\lambda) = \frac{1}{3}(4\lambda_2-1)(2\lambda_2-1)(4\lambda_2-3)\lambda_2$	vertex 2	$\lambda^2 = (0,0,1)$
$\bar\varphi^3(\lambda) = \frac{16}{3}(4\lambda_1-1)(2\lambda_1-1)\lambda_1\lambda_2$	edge 0	$\lambda^3 = (0,\frac{3}{4},\frac{1}{4})$
$\bar\varphi^4(\lambda) = 4(4\lambda_1-1)(4\lambda_2-1)\lambda_1\lambda_2$	edge 0	$\lambda^4 = (0,\frac{1}{2},\frac{1}{2})$
$\bar\varphi^5(\lambda) = \frac{16}{3}(4\lambda_2-1)(2\lambda_2-1)\lambda_1\lambda_2$	edge 0	$\lambda^5 = (0,\frac{1}{4},\frac{3}{4})$
$\bar\varphi^6(\lambda) = \frac{16}{3}(4\lambda_2-1)(2\lambda_2-1)\lambda_0\lambda_2$	edge 1	$\lambda^6 = (\frac{1}{4},0,\frac{3}{4})$
$\bar\varphi^7(\lambda) = 4(4\lambda_2-1)(4\lambda_0-1)\lambda_0\lambda_2$	edge 1	$\lambda^7 = (\frac{1}{2},0,\frac{1}{2})$
$\bar\varphi^8(\lambda) = \frac{16}{3}(4\lambda_0-1)(2\lambda_0-1)\lambda_0\lambda_2$	edge 1	$\lambda^8 = (\frac{3}{4},0,\frac{1}{4})$
$\bar\varphi^9(\lambda) = \frac{16}{3}(4\lambda_0-1)(2\lambda_0-1)\lambda_0\lambda_1$	edge 2	$\lambda^9 = (\frac{3}{4},\frac{1}{4},0)$
$\bar\varphi^{10}(\lambda) = 4(4\lambda_0-1)(4\lambda_1-1)\lambda_0\lambda_1$	edge 2	$\lambda^{10} = (\frac{1}{2},\frac{1}{2},0)$
$\bar\varphi^{11}(\lambda) = \frac{16}{3}(4\lambda_1-1)(2\lambda_1-1)\lambda_0\lambda_1$	edge 2	$\lambda^{11} = (\frac{1}{4},\frac{3}{4},0)$
$\bar\varphi^{12}(\lambda) = 32(4\lambda_0-1)\lambda_0\lambda_1\lambda_2$	center	$\lambda^{12} = (\frac{1}{2},\frac{1}{4},\frac{1}{4})$
$\bar\varphi^{13}(\lambda) = 32(4\lambda_1-1)\lambda_0\lambda_1\lambda_2$	center	$\lambda^{13} = (\frac{1}{4},\frac{1}{2},\frac{1}{4})$
$\bar\varphi^{14}(\lambda) = 32(4\lambda_2-1)\lambda_0\lambda_1\lambda_2$	center	$\lambda^{14} = (\frac{1}{4},\frac{1}{4},\frac{1}{2})$

Table 2.9: Local basis functions for quartic finite elements in 2d.

For the implementation of `get_dof_indices()` for quartics, we again need a global orientation of the edges on the mesh. At every edge three DOFs are located, which then can be ordered with respect to the orientation of the corresponding edge. In 3d, we also need a global orientation of faces for a one to one mapping of global DOFs located at a face to local DOFs on an element at that face. Such an orientation can again be defined by DOF indices at the face's vertices.

function	position	Lagrange node
$\bar\varphi^0(\lambda) = \frac{1}{3}(4\lambda_0-1)(2\lambda_0-1)(4\lambda_0-3)\lambda_0$	vertex 0	$\lambda^0 = (1,0,0,0)$
$\bar\varphi^1(\lambda) = \frac{1}{3}(4\lambda_1-1)(2\lambda_1-1)(4\lambda_1-3)\lambda_1$	vertex 1	$\lambda^1 = (0,1,0,0)$
$\bar\varphi^2(\lambda) = \frac{1}{3}(4\lambda_2-1)(2\lambda_2-1)(4\lambda_2-3)\lambda_2$	vertex 2	$\lambda^2 = (0,0,1,0)$
$\bar\varphi^3(\lambda) = \frac{1}{3}(4\lambda_3-1)(2\lambda_3-1)(4\lambda_3-3)\lambda_3$	vertex 3	$\lambda^3 = (0,0,0,1)$
$\bar\varphi^4(\lambda) = \frac{16}{3}(4\lambda_0-1)(2\lambda_0-1)\lambda_0\lambda_1$	edge 0	$\lambda^4 = (\frac{3}{4},\frac{1}{4},0,0)$
$\bar\varphi^5(\lambda) = 4(4\lambda_0-1)(4\lambda_1-1)\lambda_0\lambda_1$	edge 0	$\lambda^5 = (\frac{1}{2},\frac{1}{2},0,0)$
$\bar\varphi^6(\lambda) = \frac{16}{3}(4\lambda_1-1)(2\lambda_1-1)\lambda_0\lambda_1$	edge 0	$\lambda^6 = (\frac{1}{4},\frac{3}{4},0,0)$
$\bar\varphi^7(\lambda) = \frac{16}{3}(4\lambda_0-1)(2\lambda_0-1)\lambda_0\lambda_2$	edge 1	$\lambda^7 = (\frac{3}{4},0,\frac{1}{4},0)$
$\bar\varphi^8(\lambda) = 4(4\lambda_0-1)(4\lambda_2-1)\lambda_0\lambda_2$	edge 1	$\lambda^8 = (\frac{1}{2},0,\frac{1}{2},0)$
$\bar\varphi^9(\lambda) = \frac{16}{3}(4\lambda_2-1)(2\lambda_2-1)\lambda_0\lambda_2$	edge 1	$\lambda^9 = (\frac{1}{4},0,\frac{3}{4},0)$
$\bar\varphi^{10}(\lambda) = \frac{16}{3}(4\lambda_0-1)(2\lambda_0-1)\lambda_0\lambda_3$	edge 2	$\lambda^{10} = (\frac{3}{4},0,0,\frac{1}{4})$
$\bar\varphi^{11}(\lambda) = 4(4\lambda_0-1)(4\lambda_3-1)\lambda_0\lambda_3$	edge 2	$\lambda^{11} = (\frac{1}{2},0,0,\frac{1}{2})$
$\bar\varphi^{12}(\lambda) = \frac{16}{3}(4\lambda_3-1)(2\lambda_3-1)\lambda_0\lambda_3$	edge 2	$\lambda^{12} = (\frac{1}{4},0,0,\frac{3}{4})$
$\bar\varphi^{13}(\lambda) = \frac{16}{3}(4\lambda_1-1)(2\lambda_1-1)\lambda_1\lambda_2$	edge 3	$\lambda^{13} = (0,\frac{3}{4},\frac{1}{4},0)$
$\bar\varphi^{14}(\lambda) = 4(4\lambda_1-1)(4\lambda_2-1)\lambda_1\lambda_2$	edge 3	$\lambda^{14} = (0,\frac{1}{2},\frac{1}{2},0)$
$\bar\varphi^{15}(\lambda) = \frac{16}{3}(4\lambda_2-1)(2\lambda_2-1)\lambda_1\lambda_2$	edge 3	$\lambda^{15} = (0,\frac{1}{4},\frac{3}{4},0)$
$\bar\varphi^{16}(\lambda) = \frac{16}{3}(4\lambda_1-1)(2\lambda_1-1)\lambda_1\lambda_3$	edge 4	$\lambda^{16} = (0,\frac{3}{4},0,\frac{1}{4})$
$\bar\varphi^{17}(\lambda) = 4(4\lambda_1-1)(4\lambda_3-1)\lambda_1\lambda_3$	edge 4	$\lambda^{17} = (0,\frac{1}{2},0,\frac{1}{2})$
$\bar\varphi^{18}(\lambda) = \frac{16}{3}(4\lambda_3-1)(2\lambda_3-1)\lambda_1\lambda_3$	edge 4	$\lambda^{18} = (0,\frac{1}{4},0,\frac{3}{4})$
$\bar\varphi^{19}(\lambda) = \frac{16}{3}(4\lambda_2-1)(2\lambda_2-1)\lambda_2\lambda_3$	edge 5	$\lambda^{19} = (0,0,\frac{3}{4},\frac{1}{4})$
$\bar\varphi^{20}(\lambda) = 4(4\lambda_2-1)(4\lambda_3-1)\lambda_2\lambda_3$	edge 5	$\lambda^{20} = (0,0,\frac{1}{2},\frac{1}{2})$
$\bar\varphi^{21}(\lambda) = \frac{16}{3}(4\lambda_3-1)(2\lambda_3-1)\lambda_2\lambda_3$	edge 5	$\lambda^{21} = (0,0,\frac{1}{4},\frac{3}{4})$
$\bar\varphi^{22}(\lambda) = 32(4\lambda_1-1)\lambda_1\lambda_2\lambda_3$	face 0	$\lambda^{22} = (0,\frac{1}{2},\frac{1}{4},\frac{1}{4})$
$\bar\varphi^{23}(\lambda) = 32(4\lambda_2-1)\lambda_1\lambda_2\lambda_3$	face 0	$\lambda^{23} = (0,\frac{1}{4},\frac{1}{2},\frac{1}{4})$
$\bar\varphi^{24}(\lambda) = 32(4\lambda_3-1)\lambda_1\lambda_2\lambda_3$	face 0	$\lambda^{24} = (0,\frac{1}{4},\frac{1}{4},\frac{1}{2})$
$\bar\varphi^{25}(\lambda) = 32(4\lambda_0-1)\lambda_0\lambda_2\lambda_3$	face 1	$\lambda^{25} = (\frac{1}{2},0,\frac{1}{4},\frac{1}{4})$
$\bar\varphi^{26}(\lambda) = 32(4\lambda_2-1)\lambda_0\lambda_2\lambda_3$	face 1	$\lambda^{26} = (\frac{1}{4},0,\frac{1}{2},\frac{1}{4})$
$\bar\varphi^{27}(\lambda) = 32(4\lambda_3-1)\lambda_0\lambda_2\lambda_3$	face 1	$\lambda^{27} = (\frac{1}{4},0,\frac{1}{4},\frac{1}{2})$
$\bar\varphi^{28}(\lambda) = 32(4\lambda_0-1)\lambda_0\lambda_1\lambda_3$	face 2	$\lambda^{28} = (\frac{1}{2},\frac{1}{4},0,\frac{1}{4})$
$\bar\varphi^{29}(\lambda) = 32(4\lambda_1-1)\lambda_0\lambda_1\lambda_3$	face 2	$\lambda^{29} = (\frac{1}{4},\frac{1}{2},0,\frac{1}{4})$
$\bar\varphi^{30}(\lambda) = 32(4\lambda_3-1)\lambda_0\lambda_1\lambda_3$	face 2	$\lambda^{30} = (\frac{1}{4},\frac{1}{4},0,\frac{1}{2})$
$\bar\varphi^{31}(\lambda) = 32(4\lambda_0-1)\lambda_0\lambda_1\lambda_2$	face 3	$\lambda^{31} = (\frac{1}{2},\frac{1}{4},\frac{1}{4},0)$
$\bar\varphi^{32}(\lambda) = 32(4\lambda_1-1)\lambda_0\lambda_1\lambda_2$	face 3	$\lambda^{32} = (\frac{1}{4},\frac{1}{2},\frac{1}{4},0)$
$\bar\varphi^{33}(\lambda) = 32(4\lambda_2-1)\lambda_0\lambda_1\lambda_2$	face 3	$\lambda^{33} = (\frac{1}{4},\frac{1}{4},\frac{1}{2},0)$
$\bar\varphi^{34}(\lambda) = 256\lambda_0\lambda_1\lambda_2\lambda_3$	center	$\lambda^{34} = (\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{4})$

Table 2.10: Local basis functions for quartic finite elements in 3d.

2.5.7 Access to Lagrange elements

The Lagrange elements described above are already implemented in ALBERT; a filled BAS_FCTS structure for Lagrange elements of order degree can be accessed by a call of

```
const BAS_FCTS  *get_lagrange(int degree);
```

No call of new_bas_fcts() is needed.

2.6 Implementation of finite element spaces

2.6.1 The finite element space data structure

All information about the underlying mesh, the local basis functions, and the DOFs are collected in the following data structure which defines one single finite element space:

```
typedef struct fe_space         FE_SPACE;

struct fe_space
{
  const char       *name;
  const DOF_ADMIN  *admin;
  const BAS_FCTS   *bas_fcts;
  MESH             *mesh;
};
```

Description:

name: holds a textual description of the finite element space;

admin: pointer to the DOF administration for the DOFs of this finite element space, see Section 2.3.1;

bas_fcts: pointer to the local basis functions, see Section 2.5.1;

mesh: pointer to the underlying mesh, see Section 2.2.14.

Several finite element spaces can be handled on the same mesh. Different finite element spaces can use the same DOF administration, if they share exactly the same DOFs.

2.6.2 Access to finite element spaces

A finite element space can only be accessed by the function

```
const FE_SPACE *get_fe_space(MESH *mesh, const char *name, const int ndof[DIM+1],
                             const BAS_FCTS *bas_fcts);
```

Description:

get_fe_space(mesh, name, ndof, bas_fcts): defines a new finite element space on mesh; if bas_fcts is not nil, the function looks for an existing dof_admin administering the DOFs of bas_fcts, the argument ndof is ignored; otherwise it looks for an existing dof_admin which manages DOFs uniquely defined by ndof (compare Section 2.3.1); if such a dof_admin is not found, a new dof_admin is created; a new FE_SPACE structure is created, the name is duplicated, and all members of the structure are correctly adjusted; the return value is the initialized structure.

The selection of finite element spaces fixes the DOFs that must be present on the mesh elements. For each finite element space there must be a corresponding DOF administration, having information about the used DOFs. For the access of a mesh, via the `GET_MESH()` routine, all used DOFs, i.e. all DOF administrations, have to be specified in advance. After the access of a mesh, no further `DOF_ADMIN` can be defined (this would be correspond to a so called p–refinement of the mesh, which is not implemented). The specification of the used DOFs is done via a user defined routine

```
void init_dof_admin(MESH *mesh);
```

which is the second argument of the `GET_MESH()` routine and the function is called during the initialization of the mesh. Inside the function `init_dof_admin()`, the finite element spaces are accessed, which thereby fixes the used DOFs. Since a mesh gives only access to the `DOF_ADMIN`s defined on it, the user has to store pointers to the `FE_SPACE` structures in some global variable; no access to `FE_SPACE`s is possible via the underlying mesh.

2.6.1. Example. Now, as an example we look at a user defined function `init_dof_admin()`. In the example we want to define two finite element spaces on the mesh, for a mixed finite element formulation of the Navier–Stokes equations with the Taylor–Hood element, e.g. We want to use Lagrange finite elements of order `degree` (for the velocity) and `degree` $-$ 1 (for the pressure). Pointers to the corresponding `FE_SPACE` structures are stored in the global variables `u_fe` and `p_fe`.

```
static FE_SPACE *u_fe, *p_fe;
static int      degree;

void init_dof_admin(MESH *mesh)
{
  FUNCNAME("init_dof_admin");
  const BAS_FCTS  *lagrange;

  TEST_EXIT(mesh)("no MESH\n");
  TEST_EXIT(degree > 1)("degree must be greater than 1\n");

  lagrange = get_lagrange(degree);
  u_fe = get_fe_space(mesh, lagrange->name, nil, lagrange);

  lagrange = get_lagrange(degree-1);
  p_fe = get_fe_space(mesh, lagrange->name, nil, lagrange);

  return;
}
```

An initialization of a mesh with this `init_dof_admin()` function, will provide all DOFs for the two finite element spaces on the elements and the corresponding `DOF_ADMIN`s will have information about the access of local DOFs for both finite element spaces.

It is also possible to define only one or even more finite element spaces; the use of special user defined basis functions is possible too. These should be added to the list of all used basis functions by a call of `new_bas_fcts()` inside `init_dof_admin()`.

2.6.2. Remark. For special applications, additional DOF sets, that are not connected to any finite element space may also be defined `get_fe_space(mesh, name, n_dof, nil)`. Here, `n_dof` is a vector storing the number of DOFs at a `VERTEX`, `EDGE`, (`FACE`,) and `CENTER` that should be present. An additional DOF set with 1 DOF at each edge in 2d and face in 3d can be defined in the following way inside the `init_dof_admin()` function:

```
#if DIM == 2
   int n_dof[DIM+1] = {0,1,0};
#else
   int n_dof[DIM+1] = {0,0,1,0};
#endif
   ...

   face_dof = get_fe_space(mesh, "face dofs", n_dof, nil);

   ...
```

2.7 Routines for barycentric coordinates

Operations on single elements are performed using barycentric coordinates. In many applications, the world coordinates x of the local barycentric coordinates λ have to be calculated (see Section 2.12, e.g.). Some other applications will need the calculation of barycentric coordinates for given world coordinates (see Section 2.2.18, e.g.). Finally, derivatives of finite element functions on elements involve the Jacobian of the barycentric coordinates (see Section 2.9, e.g.).

These operations strongly depend on the element parameterization and no general routines can be supplied. For (non–parametric) simplices, ALBERT supplies functions to perform these basic tasks. Available are

```
const REAL *coord_to_world(const EL_INFO *, const REAL *, REAL_D);
const REAL *world_to_coord(const EL_INFO *, const REAL *, REAL [DIM+1]);
REAL el_grd_lambda(const EL_INFO *, REAL [DIM+1][DIM_OF_WORLD]);
REAL el_det(const EL_INFO *);
REAL el_volume(const EL_INFO *);
```

Description:

`coord_to_world(el_info, lambda, world)`: returns a pointer to a vector, which contains the world coordinates of a point in barycentric coordinates `lambda` with respect to the element `el_info->el`;

if `world` is not `nil` the world coordinates are stored in this vector; otherwise the function itself provides memory for this vector; in this case the vector is overwritten during the next call of `coord_to_world()`;

`world_to_coord(el_info, world, lambda)`: returns a pointer to a vector, which contains the barycentric coordinates with respect to the element `el_info->el` of a point with world coordinates `world`;

if `lambda` is not `nil` the barycentric coordinates are stored in this vector; otherwise the function itself provides memory for this vector; in this case this vector is overwritten during the next call of `world_to_coord()`;

`el_grd_lambda(el_info, Lambda)`: calculates the Jacobian of the barycentric coordinates on `el_info->el` and stores the matrix in `Lambda`; the return value of the function is the absolute value of the determinant of the affine linear parameterization's Jacobian;

`el_det(el_info)`: returns the the absolute value of the determinant of the affine linear parameterization's Jacobian;

el_volume(el_info): returns the the volume of the simplex.

All functions need vertex coordinate information of the element. Thus, it must only be called on elements during a mesh traversal with the `FILL_COORDS` fill flag.

2.8 Data structures for numerical quadrature

For the numerical integration of general integrals

$$\int_S f(x)\,dx$$

we use quadrature formulas described in 1.4.6. ALBERT supports numerical integration in one, two, and three dimensions on the standard simplex $\hat{S}$ in barycentric coordinates.

2.8.1 The QUAD data structure

A quadrature formula is described by the following structure:

```
typedef struct quad   QUAD;

struct quad
{
  char          *name;
  int           degree;

  int           dim;
  int           n_points;
  const double  **lambda;
  const double  *w;
};
```

Description:

name: textual description of the quadrature;

degree: quadrature is exact of degree `degree`;

dim: quadrature for dimension `dim`;

n_points: number of quadrature points;

lambda: vector `lambda[0]`, ..., `lambda[n_points-1]` of quadrature points given in barycentric coordinates (thus having `DIM+1` components);

w: vector `w[0]`, ..., `w[n_points-1]` of quadrature weights.

Numerical quadrature formulas exact up to order 19 in one (Gauss formulas), up to order 17 in two, up to order 7 in three dimensions are implemented. We only use stable formulas; this results in more quadrature points for some formulas (for example in 3d the formula which is exact of degree 3). A compilation of quadrature formulas on triangles and tetrahedra is given in [10] The implemented quadrature formulas are taken from [17, 22, 24, 37].

Functions for numerical quadrature are

```
const QUAD *get_quadrature(int dim, int degree);
REAL integrate_std_simp(const QUAD *quad, REAL (*f)(const REAL *));
```

Description:

`get_quadrature(dim, degree)`: returns a pointer to a `QUAD` structure for numerical integration in `dim` dimensions which is exact of degree min(19, `degree`) for `dim==1`, min(17, `degree`) for `dim==2`, and min(7, `degree`) for `dim==3`;

`integrate_std_simp(quad, f)`: approximates an integral by the numerical quadrature described by `quad`;
`f` is a pointer to a function to be integrated, evaluated in barycentric coordinates; the return value is

$$\sum_{\texttt{k = 0}}^{\texttt{quad->n\_points-1}} \texttt{quad->w[k] * (*f)(quad->lambda[k]);}$$

for the approximation of $\int_S f$ we have to multiply this value with $d!|S|$ for a simplex S; for a parametric simplex `f` should be a pointer to a function which calculates $f(\lambda)|\det DF_S(\hat{x}(\lambda))|$.

The following functions initializes values and gradients of functions at the quadrature nodes:

```
const REAL *f_at_qp(const QUAD *, REAL (*)(const REAL [DIM+1]), REAL *);
const REAL_D *grd_f_at_qp(const QUAD *, const REAL *(*)(const REAL [DIM+1]),
                          REAL_D *vec);
const REAL_D *f_d_at_qp(const QUAD *, const REAL *(*)(const REAL[DIM+1]),
                        REAL_D *);
const REAL_DD *grd_f_d_at_qp(const QUAD *,
                             const REAL_D *(*)(const REAL [DIM+1]), REAL_DD *);
```

Description:

`f_at_qp(quad, f, vec)`: returns a pointer `ptr` to a vector `quad->n_points` storing the values of a `REAL` valued function at all quadrature points of `quad`; `f` is a pointer to that function, evaluated in barycentric coordinates; if `vec` is not `nil`, the values are stored in this vector, otherwise the values are stored in some static local vector, which is overwritten on the next call;
`ptr[i]=(*f)(quad->lambda[i])` for $0 \le$ `i` $<$ `quad->n_points`;

`grd_f_at_qp(quad, grd_f, vec)`: returns a pointer `ptr` to a vector `quad->n_points` storing the gradient (with respect to world coordinates) of a `REAL` valued function at all quadrature points of `quad`;
`grd_f` is a pointer to a function, evaluated in barycentric coordinates and returning a pointer to a vector of length `DIM_OF_WORLD` storing the gradient;
if `vec` is not `nil`, the values are stored in this vector, otherwise the values are stored in some local static vector, which is overwritten on the next call;
`ptr[i][j]=(*grd_f)(quad->lambda[i])[j]`, for $0 \le$ `j` $<$ `DIM_OF_WORLD` and $0 \le$ `i` $<$ `quad->n_points`,

`f_d_at_qp(quad, fd, vec)`: returns a pointer `ptr` to a vector `quad->n_points` storing the values of a `REAL_D` valued function at all quadrature points of `quad`;
`fd` is a pointer to that function, evaluated in barycentric coordinates and returning a pointer to a vector of length `DIM_OF_WORLD` storing all components; if the second argument `val` of `(*fd)(lambda, val)` is not `nil`, the values have to be stored at `val`, otherwise `fd` has to provide memory for the vector which may be overwritten on the next call;
if `vec` is not `nil`, the values are stored in this vector, otherwise the values are stored in some static local vector, which is overwritten on the next call;
`ptr[i][j]=(*fd)(quad->lambda[i],val)[j]`, for $0 \le$ `j` $<$ `DIM_OF_WORLD` and $0 \le$ `i` $<$ `quad->n_points`;

`grd_f_d_at_qp(quad, grd_fd, vec)`: returns a pointer `ptr` to a vector `quad->n_points` storing the Jacobian (with respect to world coordinates) of a `REAL_D` valued function at all quadrature points of `quad`;
`grd_fd` is a pointer to a function, evaluated in barycentric coordinates and returning a pointer to a matrix of size `DIM_OF_WORLD` $\times$`DIM_OF_WORLD` storing the Jacobian; if the second argument `val` of `(*grd_fd)(x, val)` is not `nil`, the Jacobian has to be stored at `val`, otherwise `grd_fd` has to provide memory for the matrix which may be overwritten on the next call
if `vec` is not `nil`, the values are stored in this vector, otherwise the values are stored in some static local vector, which is overwritten on the next call;
`ptr[i][j][k]=(*grd_fd)(quad->lambda[i],val)[j][k]`, for $0 \le$ `j,k` $<$ `DIM_OF_WORLD` and $0 \le$ `i` $<$ `quad->n_points`,

2.8.2 The `QUAD_FAST` data structure

Often numerical integration involves basis functions, such as the assembling of the system matrix and right hand side, or the integration of finite element functions. Since numerical quadrature involves only the values at the quadrature points and the values of basis functions and its derivatives are the same at these points for all elements of the grid, such routines can be much more efficient, if they can use pre–computed values of the basis functions at the quadrature points. In this case the basis functions do not have to be evaluated for each quadrature point on every element newly.

Information that should be pre–computed can be specified by the following symbolic constants:

```
INIT_PHI
INIT_GRD_PHI
INIT_D2_PHI
```

Description:

`INIT_PHI`: pre–compute the values of all basis functions at all quadrature nodes;

`INIT_GRD_PHI`: pre–compute the gradients (with respect to the barycentric coordinates) of all basis functions at all quadrature nodes;

`INIT_D2_PHI`: pre–compute all 2nd derivatives (with respect to the barycentric coordinates) of all basis functions at all quadrature nodes;

In order to store such information for one set of basis functions we define the data structure

```
typedef struct quad_fast        QUAD_FAST;

struct quad_fast
{
  const QUAD        *quad;
  const BAS_FCTS    *bas_fcts;

  int               n_points;
  int               n_bas_fcts;
  const double      *w;

  U_CHAR            init_flag;

  REAL              **phi;
  REAL              (**grd_phi)[DIM+1];
  REAL              (**D2_phi)[DIM+1][DIM+1];
```

```
};
```

Description:

`quad`: values stored for numerical quadrature `quad`;

`bas_fcts`: values stored for basis functions `bas_fcts`;

`n_points`: number of quadrature points; equals `quad->n_points`;

`n_bas_fcts`: number of basis functions; equals `bas_fcts->n_bas_fcts`;

`w`: vector of quadrature weights; `w = quad->w`;

`init_flag`: indicates which information is initialized; may be one of, or a bitwise or of several of `INIT_PHI`, `INIT_GRD_PHI`, `INIT_D2_PHI`;

`phi`: matrix storing function values if the flag `INIT_PHI` is set;
`phi[i][j]` stores the value `bas_fcts->phi[j](quad->lambda[i])`, $0 \leq$ `j` $<$ `n_bas_fcts` and $0 \leq$ `i` $<$ `n_points`;

`grd_phi`: matrix storing all gradients (with respect to the barycentric coordinates) if the flag `INIT_GRD_PHI` is set;
`grd_phi[i][j][k]` stores the value `bas_fcts->grd_phi[j](quad->lambda[i])[k]` for $0 \leq$ `j` $<$ `n_bas_fcts`, $0 \leq$ `i` $< \ldots,$ `n_points`, and $0 \leq$ `k` $<$ `DIM`;

`D2_phi`: matrix storing all second derivatives (with respect to the barycentric coordinates) if the flag `INIT_D2_PHI` is set;
`D2_phi[i][j][k][l]` stores the value `bas_fcts->D2_phi[j](quad->lambda[i])[k][l]` for $0 \leq$ `j` $<$ `n_bas_fcts`, $0 \leq$ `i` $<$ `n_points`, and $0 \leq$ `k,l` $<$ `DIM`;

A filled structure can be accessed by a call of

```
const QUAD_FAST *get_quad_fast(const BAS_FCTS *bas_fcts, const QUAD *quad,
                               U_CHAR init_flag);
```

Description:

`get_quad_fast(bas_fcts, quad, init_flag)`: `bas_fcts` is a pointer to a filled `BAS_FCTS` structure, `quad` a pointer to some quadrature (accessed by `get_quadrature()`, e.g.) and `init_flag` indicates which information should be filled into the `QUAD_FAST` structure; it may be one of, or a bitwise or of several of `INIT_PHI`, `INIT_GRD_PHI`, `INIT_D2_PHI`; the function returns a pointer to a filled `QUAD_FAST` structure where all demanded information is computed and stored.

All used `QUAD_FAST` structures are stored in a linked list and are identified uniquely by the members `quad` and `bas_fcts`; first, `get_quad_fast()` looks for a matching structure in the linked list; if no structure is found, a new structure is generated and linked to the list; thus for one combination `bas_fcts` and `quad` only one `QUAD_FAST` structure is created.

Then `get_quad_fast()` allocates memory for all information demanded by `init_flag` and which is not yet initialized for this structure; only such information is then computed and stored; on the first call for `bas_fcts` and `quad`, all information demanded `init_flag` is generated, on a subsequent call only missing information is generated.

`get_quad_fast()` will return a `nil` pointer, if `INIT_PHI` flag is set and `bas_fcts->phi` is `nil`, `INIT_GRD_PHI` flag is set and `bas_fcts->grd_phi` is `nil`, and `INIT_D2_PHI` flag is set and `bas_fcts->D2_phi` is `nil`.

There may be several `QUAD_FAST` structures in the list for the same set of basis functions for different quadratures, and there may be several `QUAD_FAST` structures for one quadrature for different sets of basis functions.

The function get_quad_fast() should not be called an each element during mesh traversal, because it has to look in a list for an existing entry for a set of basis functions and a quadrature; a pointer to the QUAD_FAST structure should be accessed before mesh traversal and stored in some global variable.

Many functions using the QUAD_FAST structure, need vectors for storing values at all quadrature points; for these functions it can be of interest to get the count of the maximal number of quadrature nodes used by the all initialized quad_fast structures in order to avoid several memory reallocations. This count can be accessed by the function

```
int max_quad_points(void);
```

Description:

max_quad_points(): returns the maximal number of quadrature points for all yet initialized quad_fast structures; this value may change after a new initialization of a quad_fast structures; this count is *not* the maximal number of quadrature points of all used QUAD structures, since new quadratures can be used at any time without an initialization.

2.9 Functions for the evaluation of finite elements

Finite element functions are evaluated locally on single elements using barycentric coordinates (compare Section 1.4.3). ALBERT supplies several functions for calculating values and first and second derivatives of finite element functions on single elements. Functions for the calculation of derivatives are at the moment only implemented for (non–parametric) simplices.

Recalling (1.17) we obtain for the value of a finite element function u_h on an element S

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

where $\left(\bar\varphi^1, \dots, \bar\varphi^m\right)$ is a basis of $\bar{\mathbb{P}}$ and $\left(u_S^1, \dots, u_S^m\right)$ the local local coefficient vector of u_h on S. On S derivatives are evaluated by

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

and

$$D^2 u_h(x(\lambda)) = \Lambda^t \sum_{i=1}^{m} u_S^i\, D_\lambda^2 \bar\varphi^i(\lambda) \Lambda, \qquad \lambda \in \bar S,$$

where Λ is the Jacobian of the barycentric coordinates (compare (1.22) and (1.24)).

These formulas are used for all evaluation routines. Information about values of basis functions and its derivatives can be calculated via function pointers in the BAS_FCTS structure. Additionally, the local coefficient vector and the Jacobian of the barycentric coordinates are needed (for the calculation of derivatives).

Following routines calculate values of a finite element function at a single point, given in barycentric coordinates:

```
REAL eval_uh(const REAL [DIM+1], const REAL *, const BAS_FCTS *);
const REAL *eval_grd_uh(const REAL [DIM+1], const REAL_D [DIM+1], const REAL *,
```

```
                         const BAS_FCTS *, REAL_D);
const REAL_D *eval_D2_uh(const REAL [DIM+1], const REAL_D [DIM+1], const REAL *,
                          const BAS_FCTS *, REAL_DD);
const REAL *eval_uh_d(const REAL [DIM+1], const REAL_D *, const BAS_FCTS *,
                      REAL_D);
const REAL_D *eval_grd_uh_d(const REAL [DIM+1], const REAL_D [DIM+1],
                            const REAL_D *, const BAS_FCTS *, REAL_DD);
REAL eval_div_uh_d(const REAL [DIM+1], const REAL_D [DIM+1], const REAL_D *,
                   const BAS_FCTS *);
const REAL_DD *eval_D2_uh_d(const REAL [DIM+1], const REAL_D [DIM+1],
                            const REAL_D *, const BAS_FCTS *, REAL_DD *);
```

Description:

In the following `lambda` $= \lambda$ are the barycentric coordinates at which the function is evaluated, `Lambda` $= \Lambda$ is the Jacobian of the barycentric coordinates, `uh` the local coefficient vector $\left(u_S^0, \ldots, u_S^{\mathtt{m}-1}\right)$ (where u_S^i is a `REAL` or a `REAL_D`), and `bas_fcts` is a pointer to a `BAS_FCTS` structure, storing information about the set of local basis functions $\left(\bar\varphi^0, \ldots, \bar\varphi^{\mathtt{m}-1}\right)$.

All functions returning a pointer to a vector or matrix, provide memory for the vector or matrix in a static local variable. This area is overwritten during the next call. If the last argument of such a function is non `nil`, memory is provided by the calling function, where values must be stored (optional memory pointer). These values are not overwritten. The memory area must be of correct size, no check is performed.

`eval_uh(lambda, uh, bas_fcts)`: the function returns $u_h(\lambda)$;

`eval_grd_uh(lambda, Lambda, uh, bas_fcts, grd)`: the function returns a pointer `ptr` to a vector of length `DIM_OF_WORLD` storing $\nabla u_h(\lambda)$, i.e.

$$\mathtt{ptr[i]} = u_{h,x_{\mathtt{i}}}(\lambda), \qquad \mathtt{i} = 0, \ldots, \mathtt{DIM\_OF\_WORLD} - 1;$$

`grd` is an optional memory pointer;

`eval_D2_uh(lambda, Lambda, uh, bas_fcts, D2)`: the function returns a pointer `ptr` to a matrix of size (`DIM_OF_WORLD` $\times$ `DIM_OF_WORLD`) storing $D^2 u_h(\lambda)$, i.e.

$$\mathtt{ptr[i][j]} = u_{h,x_{\mathtt{i}}x_{\mathtt{j}}}(\lambda), \qquad \mathtt{i},\mathtt{j} = 0, \ldots, \mathtt{DIM\_OF\_WORLD} - 1;$$

`D2` is an optional memory pointer;

`eval_uh_d(lambda, uh, bas_fcts, val)`: the function returns a pointer `ptr` to a vector of length `DIM_OF_WORLD` storing $u_h(\lambda)$, i.e.

$$\mathtt{ptr[k]} = u_{hk}(\lambda), \qquad \mathtt{k} = 0, \ldots, \mathtt{DIM\_OF\_WORLD} - 1;$$

`val` is an optional memory pointer;

`eval_grd_uh_d(lambda, Lambda, uh, bas_fcts, grd)`: the function returns a pointer `ptr` to a vector of `DIM_OF_WORLD` vectors of length `DIM_OF_WORLD` storing $\nabla u_h(\lambda)$, i.e.

$$\mathtt{ptr[k][i]} = u_{hk,x_{\mathtt{i}}}(\lambda), \qquad \mathtt{k},\mathtt{i} = 0, \ldots, \mathtt{DIM\_OF\_WORLD} - 1;$$

`grd` is an optional memory pointer;

`eval_div_uh_d(lambda, Lambda, uh, bas_fcts)`: the function returns $\operatorname{div} u_h(\lambda)$;

eval_D2_uh(lambda, Lambda, uh, bas_fcts, D2): the function returns a pointer ptr to a vector of (DIM_OF_WORLD $\times$ DIM_OF_WORLD) matrices of length DIM_OF_WORLD storing $D^2 u_h(\lambda)$, i.e.

$$\texttt{ptr[k][i][j]} = u_{hk,x_{\mathtt{i}}x_{\mathtt{j}}}(\lambda), \qquad \mathtt{k},\mathtt{i},\mathtt{j} = 0,\ldots,\mathtt{DIM\_OF\_WORLD}-1;$$

D2 is an optional memory pointer;

Using pre–computed values of basis functions at the evaluation point, these routines can be implemented more efficiently.

```
REAL eval_uh_fast(const REAL *, const REAL *, int);
const REAL *eval_grd_uh_fast(const REAL_D [DIM+1], const REAL *,
                             const REAL (*)[DIM+1], int , REAL_D);
const REAL_D *eval_D2_uh_fast(const REAL_D [DIM+1], const REAL *,
                              const REAL (*)[DIM+1][DIM+1], int , REAL_DD);
const REAL *eval_uh_d_fast(const REAL_D *, const REAL *, int, REAL_D);
const REAL_D *eval_grd_uh_d_fast(const REAL_D [DIM+1], const REAL_D *,
                                 const REAL (*)[DIM+1], int, REAL_DD);
REAL eval_div_uh_d_fast(const REAL_D [DIM+1], const REAL_D *,
                        const REAL (*grd_phi)[DIM+1], int);
const REAL_DD *eval_D2_uh_d_fast(const REAL_D [DIM+1], const REAL_D *,
                                 const REAL (*)[DIM+1][DIM+1], int,REAL_DD *);
```

Description:

In the following Lambda $= \Lambda$ denotes the Jacobian of the barycentric coordinates, uh the local coefficient vector $\left(u_S^0,\ldots,u_S^{\mathtt{m}-1}\right)$ (where u_S^i is a REAL or a REAL_D), and m the number of local basis functions on an element.

eval_uh_fast(uh, phi, m): the function returns $u_h(\lambda)$;
phi is a vector storing the values $\bar\varphi^0(\lambda),\ldots,\bar\varphi^{\mathtt{m}-1}(\lambda)$;

eval_grd_uh_fast(Lambda, uh, grd_phi, m, grd): the function returns a pointer ptr to a vector of length DIM_OF_WORLD storing $\nabla u_h(\lambda)$, i.e.

$$\texttt{ptr[i]} = u_{h,x_{\mathtt{i}}}(\lambda), \qquad \mathtt{i} = 0,\ldots,\mathtt{DIM\_OF\_WORLD}-1;$$

grd_phi is a vector of $(\mathtt{DIM}+1)$ vectors storing $\nabla_\lambda\bar\varphi^0(\lambda),\ldots,\nabla_\lambda\bar\varphi^{\mathtt{m}-1}(\lambda)$;
grd is an optional memory pointer;

eval_D2_uh_fast(Lambda, uh, D2_phi, m, D2): the function returns a pointer ptr to a matrix of size (DIM_OF_WORLD $\times$ DIM_OF_WORLD) storing $D^2 u_h(\lambda)$, i.e.

$$\texttt{ptr[i][j]} = u_{h,x_{\mathtt{i}}x_{\mathtt{j}}}(\lambda), \qquad \mathtt{i},\mathtt{j} = 0,\ldots,\mathtt{DIM\_OF\_WORLD}-1;$$

D2_phi is a vector of $(\mathtt{DIM}+1\times\mathtt{DIM+1})$ matrices storing $D^2_\lambda\bar\varphi^0(\lambda),\ldots,D2_\lambda\bar\varphi^{\mathtt{m}-1}(\lambda)$;
D2 is an optional memory pointer;

eval_uh_d_fast(uh, phi, m, val): the function returns a pointer ptr to a vector of DIM_OF_WORLD vectors of length DIM_OF_WORLD storing $\nabla u_h(\lambda)$, i.e.

$$\texttt{ptr[k][i]} = u_{hk,x_{\mathtt{i}}}(\lambda), \qquad \mathtt{k},\mathtt{i} = 0,\ldots,\mathtt{DIM\_OF\_WORLD}-1;$$

phi is a vector storing the values $\bar\varphi^0(\lambda),\ldots,\bar\varphi^{\mathtt{m}-1}(\lambda)$;
val is an optional memory pointer;

eval_grd_uh_d_fast(Lambda, uh, grd_phi, m, grd): the function returns a pointer ptr to a vector of DIM_OF_WORLD vectors of length DIM_OF_WORLD storing $\nabla u_h(\lambda)$, i.e.

$$\texttt{ptr[k][i]} = u_{hk,x_{\texttt{i}}}(\lambda), \qquad \texttt{k}, \texttt{i} = 0, \ldots, \texttt{DIM\_OF\_WORLD} - 1;$$

grd_phi is a vector of $(\texttt{DIM}+1)$ vectors storing $\nabla_\lambda \bar\varphi^0(\lambda), \ldots, \nabla_\lambda \bar\varphi^{\texttt{m}-1}(\lambda)$;
grd is an optional memory pointer;

eval_div_uh_d_fast(Lambda, uh, grd_phi, m): the function returns $\operatorname{div} u_h(\lambda)$;
grd_phi is a vector of $(\texttt{DIM}+1)$ vectors storing $\nabla_\lambda \bar\varphi^0(\lambda), \ldots, \nabla_\lambda \bar\varphi^{\texttt{m}-1}(\lambda)$;

eval_D2_uh_d_fast(Lambda, uh, D2_phi, m, D2): the function returns a pointer ptr to a vector of $(\texttt{DIM\_OF\_WORLD} \times \texttt{DIM\_OF\_WORLD})$ matrices of length DIM_OF_WORLD storing $D^2 u_h(\lambda)$, i.e.

$$\texttt{ptr[k][i][j]} = u_{hk,x_{\texttt{i}}x_{\texttt{j}}}(\lambda), \qquad \texttt{k}, \texttt{i}, \texttt{j} = 0, \ldots, \texttt{DIM\_OF\_WORLD} - 1;$$

D2_phi is a vector of $(\texttt{DIM} + 1 \times \texttt{DIM+1})$ matrices storing $D^2_\lambda \bar\varphi^0(\lambda), \ldots, D2_\lambda \bar\varphi^{\texttt{m}-1}(\lambda)$;
D2 is an optional memory pointer.

One important task is the evaluation of finite element functions at all quadrature nodes for a given quadrature formula. Using the QUAD_FAST data structures, the values of the basis functions are known at the quadrature nodes which results in an efficient calculation of values and derivatives of finite element functions at the quadrature points.

```
const REAL *uh_at_qp(const QUAD_FAST *, const REAL *, REAL *);
const REAL_D *grd_uh_at_qp(const QUAD_FAST *, const REAL_D [DIM+1],
                           const REAL *, REAL_D *);
const REAL_DD *D2_uh_at_qp(const QUAD_FAST *, const REAL_D [DIM+1],
                           const REAL *, REAL_DD *);
const REAL_D *uh_d_at_qp(const QUAD_FAST *, const REAL_D *, REAL_D *);
const REAL_DD *grd_uh_d_at_qp(const QUAD_FAST *, const REAL_D [DIM+1],
                              const REAL_D *, REAL_DD *);
const REAL *div_uh_d_at_qp(const QUAD_FAST *, const REAL_D [DIM+1],
                           const REAL_D *, REAL *);
const REAL_DD (*D2_uh_d_at_qp(const QUAD_FAST *, const REAL_D [DIM+1],
               const REAL_D *, REAL_DD (*)[DIM_OF_WORLD]))[DIM_OF_WORLD];
```

Description: In the following uh denotes the local coefficient vector $\left(u_S^0, \ldots, u_S^{\texttt{m}-1}\right)$ (where u_S^i is a REAL or a REAL_D) on an element.

uh_at_qp(qfast, uh, val): the function returns a pointer ptr to a vector of length qfast->n_points storing the values of u_h at all quadrature points of qfast->quad, i.e.

$$\texttt{ptr[l]} = u_h(\texttt{qfast->quad->lambda[l]})$$

where $\texttt{l} = 0, \ldots, \texttt{qfast->quad->n\_points} - 1$;
the INIT_PHI flag must be set in qfast->init_flag;
val is an optional memory pointer;

grd_uh_at_qp(qfast, Lambda, uh, grd): the function returns a pointer ptr to a vector of length qfast->n_points of DIM_OF_WORLD vectors storing ∇u_h at all quadrature points of qfast->quad, i.e.

$$\texttt{ptr[l][i]} = u_{h,x_{\texttt{i}}}(\texttt{qfast->quad->lambda[l]})$$

where $\texttt{l} = 0, \ldots, \texttt{qfast->quad->n\_points} - 1$, and $\texttt{i} = 0, \ldots, \texttt{DIM\_OF\_WORLD} - 1$;
the INIT_GRD_PHI flag must be set in qfast->init_flag;
grd is an optional memory pointer;

`D2_uh_at_qp(qfast, Lambda, uh, D2)`: the function returns a pointer `ptr` to a vector of length `qfast->n_points` of (`DIM_OF_WORLD` $\times$ `DIM_OF_WORLD`) matrices storing $D^2 u_h$ at all quadrature points of `qfast->quad`, i.e.

$$\texttt{ptr[l][i][j]} = u_{h,x_{\mathtt{i}}x_{\mathtt{j}}}(\texttt{qfast->quad->lambda[l]})$$

where $\mathtt{l} = 0, \dots,$ `qfast->quad->n_points` -1, and $\mathtt{i}, \mathtt{j} = 0, \dots,$ `DIM_OF_WORLD` -1;
the `INIT_D2_PHI` flag must be set in `qfast->init_flag`;
`D2` is an optional memory pointer;

`uh_d_at_qp(qfast, uh, val)`: the function returns a pointer `ptr` to a vector of length `qfast->n_points` of `DIM_OF_WORLD` vectors storing the values of u_h at all quadrature points of `qfast->quad`, i.e.

$$\texttt{ptr[l][k]} = u_{hk}(\texttt{qfast->quad->lambda[l]})$$

where $\mathtt{l} = 0, \dots,$ `qfast->quad->n_points` -1, and $\mathtt{k} = 0, \dots,$ `DIM_OF_WORLD` -1;
the `INIT_PHI` flag must be set in `qfast->init_flag`;
`val` is an optional memory pointer;

`grd_uh_d_at_qp(qfast, Lambda, uh, grd)`: the function returns a pointer `ptr` to a vector of length `qfast->n_points` of (`DIM_OF_WORLD`$\times$`DIM_OF_WORLD`) matrices storing ∇u_h at all quadrature points of `qfast->quad`, i.e.

$$\texttt{ptr[l][k][i]} = u_{h\mathtt{k},x_{\mathtt{i}}}(\texttt{qfast->quad->lambda[l]})$$

where $\mathtt{l} = 0, \dots,$ `qfast->quad->n_points` -1, and $\mathtt{k}, \mathtt{i} = 0, \dots,$ `DIM_OF_WORLD` -1;
the `INIT_GRD_PHI` flag must be set in `qfast->init_flag`;
`grd` is an optional memory pointer;

`D2_uh_d_at_qp(qfast, Lambda, uh, D2)`: the function returns a pointer `ptr` to a vector of length `qfast->n_points` of (`DIM_OF_WORLD` $\times$ `DIM_OF_WORLD` $\times$ `DIM_OF_WORLD`) tensors storing $D^2 u_h$ at all quadrature points `qfast->quad`, i.e.

$$\texttt{ptr[l][k][i][j]} = u_{hk,x_{\mathtt{i}}x_{\mathtt{j}}}(\texttt{qfast->quad->lambda[l]})$$

where $\mathtt{l} = 0, \dots,$ `qfast->quad->n_points` -1, and $\mathtt{k}, \mathtt{i}, \mathtt{j} = 0, \dots,$ `DIM_OF_WORLD` -1;
the `INIT_D2_PHI` flag must be set in `qfast->init_flag`;
`D2` is an optional memory pointer;

2.10 Calculation of norms for finite element functions

ALBERT supplies functions for the calculation of the L^2 norm and H^1 semi–norm of a given scalar or vector valued finite element function on a non–parametric mesh.

```
REAL H1_norm_uh(const QUAD *, const DOF_REAL_VEC *);
REAL L2_norm_uh(const QUAD *, const DOF_REAL_VEC *);
REAL H1_norm_uh_d(const QUAD *, const DOF_REAL_D_VEC *);
REAL L2_norm_uh_d(const QUAD *, const DOF_REAL_D_VEC *);
```

Description:

`H1_norm_uh(quad, uh)`: returns an approximation to the H^1 semi norm of a finite element function; the coefficient vector of the vector is stored in `uh`; the domain is given by `uh->fe_space->mesh`; the element integrals are approximated by the numerical quadrature `quad`, if `quad` is not `nil`; otherwise a quadrature which is exact of degree `2*uh->fe_space->bas_fcts->degree-2` is used;

`L2_norm_uh(quad, uh)`: returns an approximation to the L^2 norm of a finite element function; the coefficient vector of the vector is stored in `uh`; the domain is given by `uh->fe_space->mesh`; the element integrals are approximated by the numerical quadrature `quad`, if `quad` is not `nil`; otherwise a quadrature which is exact of degree `2*uh->fe_space->bas_fcts->degree` is used;

`H1_norm_uh_d(quad, uh_d)`: returns an approximation to the H^1 semi norm of a finite element function; the coefficient vector of the vector is stored in `uh_d`; the domain is given by `uh_d->fe_space->mesh`; the element integrals are approximated by the numerical quadrature `quad`, if `quad` is not `nil`; otherwise a quadrature which is exact of degree `2*uh_d->fe_space->bas_fcts->degree-2` is used;

`L2_norm_uh_d(quad, uh_d)`: returns an approximation to the L^2 norm of a finite element function; the coefficient vector of the vector is stored in `uh_d`; the domain is given by `uh_d->fe_space->mesh`; the element integrals are approximated by the numerical quadrature `quad`, if `quad` is not `nil`; otherwise a quadrature which is exact of degree `2*uh_d->fe_space->bas_fcts->degree` is used;

2.11 Calculation of errors of finite element approximations

For test purposes it is convenient to calculate the exact error between a finite element approximation and the exact solution. ALBERT supplies functions to calculate the error in several norms.

```
REAL max_err_at_qp(REAL (*)(const REAL_D), const DOF_REAL_VEC *, const QUAD *);
REAL H1_err(const REAL *(*)(const REAL_D), const DOF_REAL_VEC *,
            const QUAD *, int, REAL *(*)(EL *), REAL *);
REAL L2_err(REAL (*)(const REAL_D), const DOF_REAL_VEC *,
            const QUAD *, int, REAL *(*)(EL *), REAL *);
REAL max_err_d_at_qp(const REAL *(*)(const REAL_D, REAL_D),
                     const DOF_REAL_D_VEC *, const QUAD *);
REAL H1_err_d(const REAL_D *(*)(const REAL_D, REAL_DD),
              const DOF_REAL_D_VEC *, const QUAD *, int , REAL *(*)(EL *),
              REAL *);
REAL L2_err_d(const REAL *(*)(const REAL_D, REAL_D),
              const DOF_REAL_D_VEC *, const QUAD *, int, REAL *(*)(EL *),
              REAL *);
```

Description:

`max_err_at_qp(u, uh, quad)`: the function returns the maximal error between the true solution and the approximation at all quadrature nodes on all elements of a mesh; `u` is a pointer to a function for the evaluation of the true solution, `uh` stores the coefficients of the approximation, `uh->fe_space->mesh` is the underlying mesh, and `quad` is the quadrature which gives the quadrature nodes; if `quad` is `nil`, a quadrature which is exact of degree `2*uh->fe_space->bas_fcts->degree-2` is used;

`H1_err(grd_u, uh, quad, rel_err, rw_el_err, max)`: the function returns an approximation to the error between the true solution and the approximation in the H^1 semi norm;

grd_u is a pointer to a function for the evaluation of the gradient of the true solution returning a DIM_OF_WORLD vector storing this gradient, uh stores the coefficients of the approximation, uh->fe_space->mesh is the underlying mesh, and quad is the quadrature for the approximation of the element integrals; if quad is nil, a quadrature which is exact of degree 2*uh->fe_space->bas_fcts->degree-2 is used;

L2_err(u, uh, quad, rel_err, rw_el_err, max): the function returns an approximation to the error between the true solution and the approximation in the L^2 norm; u is a pointer to a function for the evaluation of the true solution, uh stores the coefficients of the approximation, uh->fe_space->mesh is the underlying mesh, and quad is the quadrature for the approximation of the element integrals; if quad is nil, a quadrature which is exact of degree 2*uh->fe_space->bas_fcts->degree-2 is used;

max_err_at_qp_d(u_d, uh_d, quad): the function returns the maximal error between the true solution and the approximation at all quadrature nodes on all elements of a mesh; u_d is a pointer to a function for the evaluation of the true solution returning a DIM_OF_WORLD vector storing the value of the function, uh_d stores the coefficients of the approximation, uh_d->fe_space->mesh is the underlying mesh, and quad is the quadrature which gives the quadrature nodes; if quad is nil, a quadrature which is exact of degree 2*uh_d->fe_space->bas_fcts->degree-2 is used;

H1_err2_d(grd_u_d, uh_d, quad, rel_err, rw_el_err, max): the function returns an approximation to the error between the true solution and the approximation in the H^1 semi norm; grd_u_d is a pointer to a function for the evaluation of the Jacobian of the true solution returning a DIM_OF_WORLD $\times$ DIM_OF_WORLD matrix storing this Jacobian, uh_d stores the coefficients of the approximation, uh_d->fe_space->mesh is the underlying mesh, and quad is the quadrature for the approximation of the element integrals; if quad is nil, a quadrature which is exact of degree 2*uh_d->fe_space->bas_fcts->degree-2 is used;

L2_err2_d(u_d, uh_d, quad, rel_err, rw_el_err, max): the function returns an approximation to the error between the true solution and the approximation in the L^2 norm; u_d is a pointer to a function for the evaluation of the true solution returning a DIM_OF_WORLD vector storing the value of the function, uh stores the coefficients of the approximation, uh_d->fe_space->mesh is the underlying mesh, and quad is the quadrature for the approximation of the element integrals; if quad is nil, a quadrature which is exact of degree 2*uh_d->fe_space->bas_fcts->degree-2 is used;

2.12 Tools for the assemblage of linear systems

This section describes data structures and subroutines for matrix and vector assembly. Section 2.12.1 presents basic routines for the update of global matrices and vectors by adding contributions from one single element. Data structures and routines for global matrix assembly are described in Section 2.12.2. This includes library routines for the efficient implementation of a general second order linear elliptic operator. Section 2.12.3 presents data structures and routines for the handling of pre–computed integrals, which are used to speed up calculations in the case of problems with constant coefficients. The assembly of (right hand side) vectors is described in Section 2.12.4. The incorporation of Dirichlet boundary values into the right hand side is presented in Section 2.12.5. Finally, routines for generation of interpolation coefficients are described in Section 2.12.6.

2.12.1 Updating of matrices and right hand sides

The usual way to assemble the system matrix and the load vector is to loop over all (leaf) elements, calculate the local element contributions and add these to the global system matrix and the global load vector. The updating of the load vector is rather easy. The contribution of a local degree of freedom is added to the value of the corresponding global degree of freedom. Here we have to use the function j_S defined in (1.14) of Section 1.4.3 on each element S, which combines uniquely the local DOFs with the global ones. The basis functions provide in the `BAS_FCTS` structure the entry `get_dof_indices()` which is an implementation of j_S, see Section 2.5.1.

The updating of the system matrix is not that easy. As mentioned in Section 1.4.7, the system matrix is usually sparse and we use special data structures for storing these matrices, compare Section 2.3.4. For sparse matrices we do not have for each DOF a matrix row storing values for all other DOFs; only the values for pairs of DOFs are stored, where the corresponding *global* basis functions have a common support. Usually, this number of entries in one row of a sparse matrix is not know a priori and can change during grid modifications.

Thus, we use the following concept: A call of `clear_matrix()` will not set all matrix entries to zero, but will remove all matrix rows from the matrix, compare the description of this function on Page 85. During the updating of a matrix for the value of a pair of local DOFs (i, j), we look in the $j_S(i)$th row of the matrix for a column $j_S(j)$ (the `col` member of `matrix_row`); if such an entry exists, we add the current contribution; if this entry does not yet exist we will create a new entry, set the current value and column number. This creation may include an enlargement of the row, by linking a new matrix row to the list of matrix rows, if no space for a new entry is left. After the assemblage we then have a sparse matrix, storing all values for pairs of global basis functions with common support.

The function which we describe now allows also to handle matrices where the DOFs indexing the rows can differ from the DOFs indexing the columns; this makes the combination of DOFs from different finite element spaces possible. But, in the moment, only one reference to a `FE_SPACE` structure is included in the `DOF_MATRIX` structure. Thus, the handling of two different DOF sets is not fully implemented, especially the handling of matrices during `dof_compress()` may produce wrong results. Such matrices should be cleared by calling `clear_dof_matrix()` before a call to `dof_compress()`.

The following functions can be used on elements for updating matrices and vectors.

```
void add_element_matrix(DOF_MATRIX *, REAL, int row, int col, const DOF *,
                        const DOF *, const REAL **, const S_CHAR *);
void add_element_vec(DOF_REAL_VEC *, REAL, int, const DOF *, const REAL *,
                     const S_CHAR *);
void add_element_d_vec(DOF_REAL_D_VEC *, REAL, int, const DOF *, const REAL_D *,
                       const S_CHAR *);
```

Description:

`add_element_matrix(mat, fac, n_row, n_col, row_dof, col_dof, el_mat, bound)`:
updates the `DOF_MATRIX mat` by adding element contributions;

`fac` is a multiplier for the element contributions; usually `fac` is `1` or `-1`;

`n_row` is the number of rows of the element matrix;

`n_col` is the number of columns of the element matrix; `n_col` may be less or equal zero if the DOFs indexing the columns are the same as the DOFs indexing the rows; in this case `n_col = n_row` is used;

`row_dof` is a vector of length `n_row` storing the global row indices;

`col_dof` is a vector of length `n_col` storing the global column indices, `col_dof` may be a `nil` pointer if the DOFs indexing the columns are the same as the DOFs indexing the rows; in this case `col_dof = row_dof` is used;

`el_mat` is a matrix of size `n_row` $\times$ `n_col` storing the element contributions;

`bound` is an optional `S_CHAR` vector of length `n_row`; `bound` *must* be a `nil` pointer if row indices and column indices are not the same; if row indices and column indices are the same and `bound` is not `nil` it holds the boundary type of the global DOFs; contributions are not added in rows corresponding to a Dirichlet DOF; the diagonal matrix entry of a Dirichlet DOF is set to `1.0`.

If row indices and column indices differ or if `bound` is a `nil` pointer, then for all `i` the values `fac*el_mat[i][j]` are added to the entries (`row_dof[i],col_dof[j]`) in the global matrix `mat` ($0 \leq$ `i` $<$ `n_row`, $0 \leq$ `j` $<$ `n_col`); if such an entry exists in the `row_dof[i]`th row of the matrix the value is added; otherwise a new entry is created in the row, the value is set and the column number is set to `col_dof[j]`; this may include an enlargement of the row by adding a new `MATRIX_ROW` structure to the list of matrix rows.

If row DOFs and column DOFs are the same and `bound` is not `nil` these values are only added for row indices `i` with `bound[i] < DIRICHLET`; for row indices `i` with `bound[i] >= DIRICHLET` only the diagonal element is set to `1.0`; the values in the `i`th row of `el_mat` are ignored.

If row indices and column indices are the same, the diagonal element is *always* the first entry in a matrix row; this makes the access to the diagonal element easy for a diagonal preconditioner, e.g.;

`add_element_vec(drv, fac, n_dof, dof, el_vec, bound)`: updates the `DOF_REAL_VEC drv`;

`fac` is a multiplier for the element contributions; usually `fac` is `1` or `-1`;

`n_dof` is the number of local degrees of freedom;

`dof` is a vector of length `n_dof` storing the global DOFs, i.e. `dof[i]` is the global DOF of the `i`th contribution;

`el_vec` is a `REAL` vector of length `n_dof` storing the element contributions;

`bound` is an optional vector of length `n_dof`; if `bound` is not `nil` it holds the boundary type of the global DOFs; contributions are only added for non–Dirichlet DOFs.

For all `i` or, if `bound` is not `nil`, for all `i` with `bound[i] < DIRICHLET` ($0 \leq$ `i` $<$ `n_dof`) the value `fac*el_vec[i]` is added to `drv->vec[dof[i]]`; values in `drv->vec` stay untouched for Dirichlet DOFs.

`add_element_d_vec(drdv, fac, n_dof, dof, el_vec, bound)`: performs the update of the `DOF_REAL_D_VEC drdv`;

`fac` is a multiplier for the element contributions; usually `fac` is `1` or `-1`;

`n_dof` is the number of local degrees of freedom;

`dof` is a vector of length `n_dof` storing the global DOFs, i.e. `dof[i]` is the global DOF of the `i`th contribution;

`el_vec` is a `REAL_D` vector of length `n_dof` storing the element contributions;

`bound` is an optional vector of length `n_dof`; if `bound` is not `nil` it holds the boundary type of the global DOFs; contributions are only added for non–Dirichlet DOFs.

For all `i` or, if `bound` is not `nil`, for all `i` with `bound[i] < DIRICHLET` ($0 \leq$ `i` $<$ `n_dof`) the value `fac*el_vec[i]` is added to `drdv->vec[dof[i]]`; values in the `drdv->vec` stay untouched for Dirichlet DOFs.

2.12.2 Data structures and function for matrix assemblage

The following structure holds full information for the assembling of element matrices. This structure is used by the function `update_matrix()` described below.

```
typedef struct el_matrix_info  EL_MATRIX_INFO;

struct el_matrix_info
{
  int              n_row;
  const DOF_ADMIN  *row_admin;
  const DOF        *(*get_row_dof)(const EL *,const DOF_ADMIN *, DOF *);

  int              n_col;
  const DOF_ADMIN  *col_admin;
  const DOF        *(*get_col_dof)(const EL *,const DOF_ADMIN *, DOF *);

  const S_CHAR     *(*get_bound)(const EL_INFO *, S_CHAR *);

  REAL             factor;

  const REAL       **(*el_matrix_fct)(const EL_INFO *, void *);
  void             *fill_info;

  FLAGS            fill_flag;
};
```

Description:

`n_row`: number of rows of the element matrix;

`row_admin`: pointer to a `DOF_ADMIN` structure for the administration of DOFs indexing the rows of the matrix;

`get_row_dof`: pointer to a function for the access of the global row DOFs on a single element; `get_row_dof(el, row_admin, row_dof)` returns a pointer to a vector of length `n_row` storing the global DOFs indexing the rows of the matrix; if `row_dof` is a `nil` pointer, `get_row_dof()` has to provide memory for storing this vector, which may be overwritten on the next call; otherwise the DOFs have to be stored at `row_dof`; usually, `get_row_dof()` is the `get_dof_indices()` function from a `BAS_FCTS` structure (compare Section 2.5.1);

`n_col`: number of columns of the element matrix; `n_col` may be less or equal zero if the DOFs indexing the columns are the same as the DOFs indexing the rows; in this case `n_col = n_row` is used;

`col_admin`: pointer to a `DOF_ADMIN` structure for the administration of DOFs indexing the columns of the matrix; `col_admin` may be a `nil` pointer if the column DOFs are the same as the row DOFs;

`get_col_dof`: pointer to a function for the access of the global row DOFs on a single element; `get_col_dof` may be a `nil` pointer if the column DOFs are the same as the row DOFs; if row and column DOFs differ then the column DOFs are accessed by `get_col_dof()` otherwise the vector accessed by `get_row_dof()` is used for the columns also; `get_col_dof(el, col_admin, col_dof)` returns a pointer to a vector of length `n_col` storing the global DOFs indexing the rows of the matrix; if `col_dof` is a `nil` pointer, `get_col_dof()` has to provide memory for storing this vector, which may be overwritten on the next call; otherwise the DOFs have to be stored at col_dof; usually, get_col_dof() is the get_dof_indices() function from a BAS_FCTS structure (compare Section 2.5.1);

get_bound: is an optional pointer to a function providing information about the boundary type of DOFs; get_bound *must* be a nil pointer if row indices and column indices are not the same; otherwise get_bound(el_info, bound) returns a pointer to a vector of length n_row storing the boundary type of the local DOFs; this pointer is the optional pointer to a vector holding boundary information of the function add_element_matrix() described above; if bound is a nil pointer, get_bound() has to provide memory for storing this vector, which may be overwritten on the next call; otherwise the DOFs have to be stored at bound; usually, get_bound() is the get_bound() function from a BAS_FCTS structure (compare Section 2.5.1);

factor: is a multiplier for the element contributions; usually factor is 1 or -1;

el_matrix_fct: is a pointer to a function for the computation of the element matrix; el_matrix_fct(el_info, fill_info) returns a pointer to a matrix of size n_row $\times$ n_col storing the element matrix on element el_info->el; fill_info is a pointer to data needed by el_matrix_fct(); the function has to provide memory for storing the element matrix, which can be overwritten on the next call;

fill_info: pointer to data needed by el_matrix_fct(); will be given as second argument to this function;

fill_flag: the flag for the mesh traversal for assembling the matrix.

The following function updates a matrix by assembling element contributions during mesh traversal; information for computing the element matrices is provided in an EL_MATRIX_INFO structure:

```
void update_matrix(DOF_MATRIX *matrix, const EL_MATRIX_INFO *);
```

Description:

update_matrix(matrix, info): updates the matrix matrix by traversing the underlying mesh and assembling the element contributions into the matrix; information about the computation of element matrices and connection of local and global DOFs is stored in info;

the flags for the mesh traversal of the mesh matrix->fe_space->mesh are stored at info->fill_flags which specifies the elements to be visited and information that should be present on the elements for the calculation of the element matrices and boundary information (if info->get_bound is not nil).

On the elements, information about the row DOFs is accessed by info->get_row_dof using info->row_admin; this vector is also used for the column DOFs if info->n_col is less or equal zero, or info->get_col_admin or info->get_col_dof is a nil pointer; if row and column DOFs are the same, the boundary type of the DOFs is accessed by info->get_bound if info->get_bound is not a nil pointer; then the element matrix is computed by info->el_matrix_fct(el_info, info->fill_info); these contributions, multiplied by info->factor, are eventually added to matrix by a call of add_element_matrix() with all information about row and column DOFs, the element matrix, and boundary types, if available;

update_matrix() only adds element contributions; this makes several calls for the assemblage of one matrix possible; before the first call, the matrix should be cleared by calling clear_dof_matrix().

Now we want to describe some tools which enable an easy assemblage of the system matrix. For this we have to provide a function for the calculation of the element matrix. For a general elliptic problem the element matrix $\boldsymbol{L}_S = (L_S^{ij})_{i,j=1,\dots,m}$ is given by (recall (1.44) in Section 1.4.5)

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

where $\bar A$, $\bar b$, and $\bar c$ are functions depending on given data and on the actual element, namely

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

Having access to functions for the evaluation of $\bar A$, $\bar b$, and $\bar c$ at given quadrature nodes, the above integrals can be computed by some general routine for any set of local basis functions using quadrature. Additionally, if a coefficient is piecewise constant on the mesh, only an integration of basis functions has to be done (compare (1.48)) for this term. Here we can use pre–computed integrals of the basis functions on the standard element and transform them to the actual element. Such a computation is usually much faster than using quadrature on each single element. Data structures for storing such pre–computed values are described in Section 2.12.3.

For the assemblage routines which we will describe now, we use the following slight generalization: In the discretization of the first order term, sometimes integration by parts is used too. For a divergence free vector field b and purely Dirichlet boundary values this leads for example to

$$\int_\Omega \varphi(x)\, b(x) \cdot \nabla u(x)\, dx = \frac{1}{2}\left(\int_\Omega \varphi(x)\, b(x) \cdot \nabla u(x)\, dx - \int_\Omega \nabla\varphi(x) \cdot b(x)\, u(x)\, dx\right)$$

yielding a modified first order term for the element matrix

$$\int_{\hat S} \bar\varphi^i(\lambda(\hat x))\, \frac{1}{2}\bar b(\lambda(\hat x)) \cdot \nabla_\lambda \bar\varphi^j(\lambda(\hat x))\, d\hat x - \int_{\hat S} \nabla_\lambda \bar\varphi^i(\lambda(\hat x)) \cdot \frac{1}{2}\bar b(\lambda(\hat x))\, \bar\varphi^j(\lambda(\hat x))\, d\hat x.$$

Secondly, we allow that we have two finite element spaces with local basis functions $\{\bar\psi_i\}_{i=1,\dots,n}$ and $\{\bar\varphi_i\}_{i=1,\dots,m}$.

In general the following contributions of the element matrix $\boldsymbol{L}_S = (L_S^{ij})_{\substack{i=1,\dots,n\\ j=1,\dots,m}}$ have to be computed:

$$\begin{aligned} &\int_{\hat S} \nabla_\lambda \bar\psi^i(\lambda(\hat x)) \cdot \bar A(\lambda(\hat x))\, \nabla_\lambda \bar\varphi^j(\lambda(\hat x))\, d\hat x && \text{second order term,} \\ &\int_{\hat S} \bar\psi^i(\lambda(\hat x))\, \bar b^0(\lambda(\hat x)) \cdot \nabla_\lambda \bar\varphi^j(\lambda(\hat x))\, d\hat x && \\ &\int_{\hat S} \nabla_\lambda \bar\psi^i(\lambda(\hat x)) \cdot\ \bar b^1(\lambda(\hat x))\, \bar\varphi^j(\lambda(\hat x))\, d\hat x && \text{first order terms,} \\ &\int_{\hat S} \bar c(\lambda(\hat x))\, \bar\psi^i(\lambda(\hat x))\, \bar\varphi^j(\lambda(\hat x))\, d\hat x && \text{zero order term,} \end{aligned}$$

where for example $\bar b^0 = \bar b$ and $\bar b^1 = 0$, or using integration by parts $\bar b^0 = \frac{1}{2}\bar b$ and $\bar b^1 = -\frac{1}{2}\bar b$.

In order to store information about the finite element spaces, the problem dependent functions $\bar A$, $\bar b^0$, $\bar b^1$, $\bar c$ and the quadrature that should be used for the numerical integration of the element matrix, we define the following data structure:

```
typedef struct operator_info  OPERATOR_INFO;

struct operator_info
{
  const FE_SPACE *row_fe_space;
  const FE_SPACE *col_fe_space;

  const QUAD *quad[3];

  void       (*init_element)(const EL_INFO *, const QUAD *[3], void *);
  const REAL (*(*LALt)(const EL_INFO *, const QUAD *, int, void *))[DIM+1];
  int        LALt_pw_const;
  int        LALt_symmetric;
  const REAL *(*Lb0)(const EL_INFO *, const QUAD *, int, void *);
  int        Lb0_pw_const;
  const REAL *(*Lb1)(const EL_INFO *, const QUAD *, int, void *);
  int        Lb1_pw_const;
  int        Lb0_Lb1_anti_symmetric;
  REAL       (*c)(const EL_INFO *, const QUAD *, int, void *);
  int        c_pw_const;

  int        use_get_bound;
  void       *user_data;
  FLAGS      fill_flag;
};
```

Description:

`row_fe_space`: pointer to a finite element space connected to the row DOFs of the resulting matrix;

`col_fe_space`: pointer to a finite element space connected to the column DOFs of the resulting matrix;

`quad`: vector with pointers to quadratures; `quad[0]` is used for the integration of the zero order term, `quad[1]` for the first order term(s), and `quad[2]` for the second order term;

`init_element`: pointer to a function for doing an initialization step on each element; `init_element` may be a `nil` pointer;
if `init_element` is not `nil`, `init_element(el_info, quad, user_data)` is the first statement executed on each element `el_info->el` and may initialize data which is used by the functions `LALt()`, `Lb0()`, `Lb1()`, and/or `c()` (the Jacobian of the barycentric coordinates in the 1st and 2nd order terms or the element volume for all order terms, e.g.); `quad` is a pointer to a vector of quadratures which is actually used for the integration of the various order terms and `user_data` may hold a pointer to user data, filled by `init_element()`, e.g.;

`LALt`: is a pointer to a function for the evaluation of $\bar{A}$ at quadrature nodes on the element; `LALt` may be a `nil` pointer, if no second order term has to be integrated;
if `LALt` is not `nil`, `LALt(el_info, quad, iq, user_data)` returns a pointer to a matrix of size `DIM+1` $\times$ `DIM+1` storing the value of $\bar{A}$ at `quad->lambda[iq]`; `quad` is the quadrature for the second order term and `user_data` is a pointer to user data;

`LALt_pw_const`: is not zero if $\bar{A}$ is piecewise constant on the mesh (constant matrix A on a non–parametric mesh, e.g.); thus integration of the second order term can use pre–computed integrals of the basis functions on the standard element; otherwise integration is done by using quadrature on each element;

`LALt_symmetric`: is not zero if $\bar{A}$ is a symmetric matrix; if the finite element spaces for rows and columns are the same, only the diagonal and the upper part of the element matrix for the second order term have to be computed; elements of the lower part can then be set using the symmetry; otherwise the complete element matrix has to be calculated;

`Lb0`: is a pointer to a function for the evaluation of $\bar{b}^0$, at quadrature nodes on the element; `Lb0` may be a `nil` pointer, if this first order term has not to be integrated;

if `Lb0` is not `nil`, `Lb0(el_info, quad, iq, user_data)` returns a pointer to a vector of length `DIM+1` storing the value of $\bar{b}^0$ at `quad->lambda[iq]`; `quad` is the quadrature for the first order term and `user_data` is a pointer to user data;

`Lb0_pw_const`: is not zero if $\bar{b}^0$ is piecewise constant on the mesh (constant vector b on a non–parametric mesh, e.g.); thus integration of the first order term can use pre–computed integrals of the basis functions on the standard element; otherwise integration is done by using quadrature on each element;

`Lb1`: is a pointer to a function for the evaluation of $\bar{b}^1$, at quadrature nodes on the element; `Lb1` may be a `nil` pointer, if this first order term has not to be integrated;

if `Lb1` is not `nil`, `Lb1(el_info, quad, iq, user_data)` returns a pointer to a vector of length `DIM+1` storing the value of $\bar{b}^1$ at `quad->lambda[iq]`; `quad` is the quadrature for the first order term and `user_data` is a pointer to user data;

`Lb1_pw_const`: is not zero if $\bar{b}^1$ is piecewise constant on the mesh (constant vector b on a non–parametric mesh, e.g.); thus integration of the first order term can use pre–computed integrals of the basis functions on the standard element; otherwise integration is done by using quadrature on each element;

`Lb0_Lb1_anti_symmetric`: is not zero if the contributions of the complete first order term to the *local* element matrix are anti symmetric (only possible if both `Lb0` and `Lb1` are not `nil`, $\bar{b}^0 = -\bar{b}^1$, e.g.); if the finite element spaces for rows and columns are the same then only the upper part of the element matrix for the first order term has to be computed; elements of the lower part can then be set using the anti symmetry; otherwise the complete element matrix has to be calculated;

`c`: is a pointer to a function for the evaluation of $\bar{c}$ at quadrature nodes on the element; `c` may be a `nil` pointer, if no zero order term has to be integrated;

if `c` is not `nil`, `c(el_info, quad, iq, user_data)` returns the value of the function $\bar{c}$ at `quad->lambda[iq]`; `quad` is the quadrature for the zero order term and `user_data` is a pointer to user data;

`c_pw_const`: is not zero if the zero order term $\bar{c}$ is piecewise constant on the mesh (constant function c on a non–parametric mesh, e.g.); thus integration of the zero order term can use pre–computed integrals of the basis functions on the standard element; otherwise integration is done by using quadrature on each element;

`use_get_bound`: if non–zero, then the `get_bound` entry in `EL_MATRIX_INFO` is set to `row_fe_space->bas_fcts->get_bound()`, and Dirichlet boundary DOFs are handled accordingly; `use_get_bound` must be zero if two different finite element spaces are used;

`user_data`: optional pointer to memory segment for user data used by `init_element()`, `LALt()`, `Lb0()`, `Lb1()`, and/or `c()` and is the last argument to these functions;

`fill_flag`: the flag for the mesh traversal routine indicating which elements should be visited and which information should be present in the `EL_INFO` structure for `init_element()`, `LALt()`, `Lb0()`, `Lb1()`, and/or `c()` on the visited elements.

Information stored in such a structure is used by the following function which returns a pointer to a filled EL_MATRIX_INFO structure; this structure can be used as an argument to the update_matrix() function which will then assemble the discrete matrix corresponding to the operator defined in the OPERATOR_INFO:

```
const EL_MATRIX_INFO *fill_matrix_info(const OPERATOR_INFO *, EL_MATRIX_INFO *);
```

Description:

fill_matrix_info(op_info, mat_info): returns a pointer to a filled EL_MATRIX_INFO structure for the assemblage of the system matrix for the operator defined in op_info;

If the second argument mat_info is a nil pointer, a new structure mat_info is allocated and filled; otherwise the structure mat_info is filled; all members are newly assigned!

op_info->row_fe_space and op_info->col_fe_space are pointers to the finite element spaces (and by this to the basis functions and DOFs) connected to the row DOFs and the column DOFs of the matrix to be assembled; if both pointers are nil pointers, an error message is given, and the program stops; if one of these pointers is nil, rows and column DOFs are connected with the same finite element space (i.e. op_info->row_fe_space = op_info->col_fe_space, or op_info->col_fe_space = op_info->row_fe_space is used).

The numbers of basis functions in both structures define the members mat_info->n_row and mat_info->n_col, the corresponding dof_admins the mat_info->row_admin and mat_info->col_admin, and the get_dof_indices() of the basis functions are used for mat_info->get_row_dof and mat_info->get_col_dof; the entries for the columns are only set if the finite element spaces are not the same; if the spaces are the same and the use_get_bound entry is not zero, then the get_bound() function of the basis functions is used for the member mat_info->get_bound().

The most important member in the structure, namely mat_info->el_matrix_fct, is adjusted to some general routine for the integration of the element matrix for any set of local basis functions; fill_matrix_info() tries to use the fastest function for the element integration for the operator defined in op_info;

Denote by row_degree and col_degree the degree of the basis functions connected to the rows and columns; the following vector quad of quadratures is used for the element integration, if not specified by op_info->quad using the following rule: pre–computed integrals of basis functions should be evaluated exactly, and all terms calculated by quadrature on the elements should use the same quadrature formula (this is more efficient than to use different quadratures). To be more specific:

If the 2nd order term has to be integrated and op_info->quad[2] is not nil, quad[2] = op_info->quad[2] is used, otherwise quad[2] is a quadrature which is exact of degree row_degree+col_degree-2. If the 2nd order term is not integrated then quad[2] is set to nil.

If the 1st order term has to be integrated and op_info->quad[1] is not nil, quad[1] = op_info->quad[1] is used; otherwise: if op_info->Lb_pw_const is zero and quad[2] is not nil, quad[1] = quad[2] is used, otherwise quad[1] is a quadrature which is exact of degree row_degree+col_degree-1. If the 1st order term is not integrated then quad[1] is set to nil.

If the zero order term has to be integrated and op_info->quad[0] is not nil, quad[0] = op_info->quad[0] is used; otherwise: if op_info->c_pw_const is zero and quad[2] is not nil, quad[0] = quad[2] is used, if quad[2] is nil and quad[1] is not nil, quad[0] = quad[1] is used, or if both quadratures are nil, quad[0] is a quadrature which is exact of degree row_degree+col_degree. If the zero order term is not integrated then quad[0] is set to nil.

If `op_info->init_element` is not `nil` then a call of `op_info->init_element(el_info, quad, op_info->user_data)` is the first statement of `mat_info->el_matrix_fct()` on each element; `el_info` is a pointer to the `EL_INFO` structure of the actual element, `quad` is the quadrature vector described above (now giving information about the actually used quadratures), and a pointer to user data.

If `op_info->LALt` is not `nil`, the 2nd order term is integrated using the quadrature `quad[2]`; if `op_info->LALt_pw_const` is not zero, the integrals of the product of gradients of the basis functions on the standard simplex are initialized (using the quadrature `quad[2]` for the integration) and used for the computation on the elements; `op_info->LALt()` is only called once with arguments `op_info->LALt(el_info, quad[2], 0, op_info->user_data)`, i.e. the matrix of the 2nd order term is evaluated only at the first quadrature node; otherwise the integrals are approximated by quadrature and `op_info->LALt()` is called for each quadrature node of `quad[2]`; if `op_info->LALt_symmetric` is not zero, the symmetry of the element matrix is used, if the finite element spaces are the same and this term is not integrated by the same quadrature as the first order term.

If `op_info->Lb0` is not `nil`, this 1st order term is integrated using the quadrature `quad[1]`; if `op_info->Lb0_pw_const` is not zero, the integrals of the product of basis functions with gradients of basis functions on the standard simplex are initialized (using the quadrature `quad[1]` for the integration) and used for the computation on the elements; `op_info->Lb0()` is only called once with arguments `op_info->Lb0(el_info, quad[1], 0, op_info->user_data)`, i.e. the vector of this 1st order term is evaluated only at the first quadrature node; otherwise the integrals are approximated by quadrature and `op_info->Lb0()` is called for each quadrature node of `quad[1]`;

If `op_info->Lb1` is not `nil`, this 1st order term is integrated also using the quadrature `quad[1]`; if `op_info->Lb1_pw_const` is not zero, the integrals of the product of gradients of basis functions with basis functions on the standard simplex are initialized (using the quadrature `quad[1]` for the integration) and used for the computation on the elements; `op_info->Lb1()` is only called once with arguments `op_info->Lb1(el_info, quad[1], 0, op_info->user_data)`, i.e. the vector of this 1st order term is evaluated only at the first quadrature node; otherwise the integrals are approximated by quadrature and `op_info->Lb1()` is called for each quadrature node of `quad[1]`.

If both function pointers `op_info->Lb0` and `op_info->Lb1` are not `nil`, the finite element spaces for rows and columns are the same and `Lb0_Lb1_anti_symmetric` is non–zero, then the contributions of the 1st order term are computed using this anti symmetry property.

If `op_info->c` is not `nil`, the zero order term is integrated using the quadrature `quad[0]`; if `op_info->c_pw_const` is not zero, the integrals of the product of basis functions on the standard simplex are initialized (using the quadrature `quad[0]` for the integration) and used for the computation on the elements; `op_info->c()` is only called once with arguments `op_info->c(el_info, quad[0], 0, op_info->user_data)`, i.e. the zero order term is evaluated only at the first quadrature node; otherwise the integrals are approximated by quadrature and `op_info->c()` is called for each quadrature node of `quad[0]`.

The functions `LALt()`, `Lb0()`, `Lb1()`, and `c()`, can be called in any order on the elements, if not `nil` (this depends on the type of integration, using pre–computed values, using same/different quadrature for the second, first, and/or zero order term, e.g.) but these functions are always initialized by `op_info->init_element()`, if this function pointer is not `nil`.

Using all information about the operator and quadrature an "optimal" routine for the assemblage is chosen. Information for this routine is stored at `mat_info` which includes the pointer to user data `op_info->user_data` (the last argument to `init_element()`, `LALt()`, `Lb0()`, `Lb1()`, and/or `c()`).

Finally, the flag for the mesh traversal used by the function `update_matrix()` is set in `mat_info->fill_flag` to `op_info->fill_flag`; it indicates which elements should be visited and which information should be present in the `EL_INFO` structure for `init_element()`, `LALt()`, `Lb0/1()`, and/or `c()` on the visited elements. If the finite element spaces are the same, and the entry `op_info->use_get_bound` is set, then the `FILL_BOUND` flag is added to `mat_info->fill_flag`.

2.12.1. Example. The following source fragment gives an example of the implementation for the operator $-\Delta u$ and the access of a `MATRIX_INFO` structure for the automatic assemblage of the system matrix for this problem for any set of used basis functions.

Here we only have to implement a constant second order term. For passing information about the gradient of the barycentric coordinates from `init_element()` to the function `LALt` we define the following structure

```
struct op_info
{
  REAL_D  Lambda[DIM+1];    /*  the gradient of the barycentric coordinates */
  REAL    det;              /*  |det D F_S|                                 */
};
```

The function `init_element()` calculates the Jacobian Λ of the barycentric coordinates on the simplex and stores these in the above defined structure:

```
static void init_element(const EL_INFO *el_info, const QUAD *quad[3], void *ud)
{
  FUNCNAME("init_element");
  struct op_info *info = ud;

  info->det = el_grd_lambda(el_info, info->Lambda);
  return;
}
```

The function `LALt` now has to calculate the scaled matrix product $|\det DF_S|\Lambda\Lambda^t$:

```
const REAL (*LALt(const EL_INFO *el_info, const QUAD *quad, int iq,
                  void *ud))[DIM+1]
{
  FUNCNAME("LALt");
  struct op_info *info = ud;
  int            i, j, k;
  static REAL    LALt[DIM+1][DIM+1];

  for (i = 0; i <= DIM; i++)
  {
    for (j = i; j <= DIM; j++)
    {
      for (LALt[i][j] = k = 0; k < DIM_OF_WORLD; k++)
        LALt[i][j] += info->Lambda[i][k]*info->Lambda[j][k];
      LALt[i][j] *= info->det;
      LALt[j][i] = LALt[i][j];
    }
  }
```

```
    return((const REAL (*)[DIM+1]) LALt);
  }
```

To access now a `MATRIX_INFO` structure we additionally need the actually used finite element space `fe_space`:

```
const EL_MATRIX_INFO *matrix_info = nil;
OPERATOR_INFO         o_info = {nil};

o_info.row_fe_space   = o_info.col_fe_space = fe_space;
o_info.init_element   = init_element;
o_info.LALt           = LALt;
o_info.LALt_pw_const  = true;        /* pw const. assemblage is faster */
o_info.LALt_symmetric = true;        /* symmetric assemblage is faster */
o_info.use_get_bound  = true;        /* Dirichlet boundary conditions! */
o_info.user_data = MEM_ALLOC(1, struct op_info);         /* user data! */
o_info.fill_flag = CALL_LEAF_EL|FILL_COORDS;

matrix_info = fill_matrix_info(&o_info, nil);
```

The system matrix `matrix` can then be assembled by

```
clear_dof_matrix(matrix);
update_matrix(matrix, matrix_info);
```

2.12.3 Data structures for storing pre–computed integrals of basis functions

Assume a non–parametric triangulation and constant coefficient functions A, b, and c. Since the Jacobian of the barycentric coordinates is constant on S, the functions $\bar{A}_S$, $\bar{b}^0_S$, $\bar{b}^1_S$, and $\bar{c}_S$ are constant on S also. Now, looking at the element matrix approximated by some quadrature $\hat{Q}$, we observe

$$
\begin{aligned}
\hat{Q}\Big(\sum_{k,l=0}^{d} (\bar{a}_{S,kl}\bar{\psi}^i_{,\lambda_k}\, \bar{\varphi}^j_{,\lambda_l})\Big) &= \sum_{k,l=0}^{d} \bar{a}_{S,kl}\hat{Q}\Big(\bar{\psi}^i_{,\lambda_k}\, \bar{\varphi}^j_{,\lambda_l}\Big),\\
\hat{Q}\Big(\sum_{l=0}^{d} (\bar{b}^0_{S,l}\, \bar{\psi}^i\, \bar{\varphi}^j_{,\lambda_l})\Big) &= \sum_{l=0}^{d} \bar{b}^0_{S,l}\, \hat{Q}\Big(\bar{\psi}^i\, \bar{\varphi}^j_{,\lambda_l}\Big),\\
\hat{Q}\Big(\sum_{k=0}^{d} (\bar{b}^1_{S,k}\, \bar{\psi}^i_{,\lambda_k}\, \bar{\varphi}^j)\Big) &= \sum_{k=0}^{d} \bar{b}^1_{S,k}\, \hat{Q}\Big(\bar{\psi}^i_{,\lambda_k}\, \bar{\varphi}^j)\Big), \qquad \text{and}\\
\hat{Q}\Big((\bar{c}_S\, \bar{\psi}^i\, \bar{\varphi}^j)\Big) &= \bar{c}_S\, \hat{Q}\Big(\bar{\psi}^i\, \bar{\varphi}^j\Big).
\end{aligned}
$$

The values of the quadrature applied to the basis functions does only depend on the basis functions and the standard element but not on the actual simplex S. All information about S is given by $\bar{A}_S$, $\bar{b}^0_S$, $\bar{b}^1_S$, and $\bar{c}_S$. Thus, these quadratures have only to be calculated once, and can then be used on each element during the assembling.

For this we have to store for the basis functions $\{\bar{\psi}_i\}_{i=1,\ldots,n}$ and $\{\bar{\varphi}_j\}_{j=1,\ldots,m}$ the values

$$\hat{Q}^{11}_{ij,kl} := \hat{Q}\Big(\bar{\psi}^i_{,\lambda_k}\, \bar{\varphi}^j_{,\lambda_l}\Big) \qquad \text{for } 1 \le i \le n,\ 1 \le j \le m,\ 0 \le k,l \le \texttt{DIM},$$

if $A \neq 0$,

$$\hat{Q}^{01}_{ij,l} := \hat{Q}\Big(\bar{\psi}^i\, \bar{\varphi}^j_{,\lambda_l}\Big) \qquad \text{for } 1 \le i \le n,\ 1 \le j \le m,\ 0 \le l \le \texttt{DIM},$$

if $b^0 \neq 0$,

$$\hat{Q}^{10}_{ij,k} := \hat{Q}\Big(\bar{\psi}^i_{,\lambda_k}\, \bar{\varphi}^j\Big) \qquad \text{for } 1 \le i \le n,\ 1 \le j \le m,\ 0 \le k \le \texttt{DIM}$$

if $b^1 \neq 0$, and

$$\hat{Q}^{00}_{ij} := \hat{Q}\Big(\bar{\psi}^i\, \bar{\varphi}^j\Big) \qquad \text{for } 1 \le i \le n,\ 1 \le j \le m,$$

if $c \neq 0$. Many of these values are zero, especially for the first and second order terms (if $\bar{\psi}^i$ and $\bar{\varphi}^j$ are the linear nodal basis functions $\hat{Q}^{11}_{ij,kl} = \delta_{ij}\delta_{kl}$). Thus, we use special data structures for a sparse storage of the non zero values for these terms. These are described now.

In order to "define" zero entries we use

```
static const REAL TOO_SMALL = 1.e-15;
```

and all computed values `val` with $|\texttt{val}| \le$ `TOO_SMALL` are treated as zeros.

The following data structure is used for storing values $\hat{Q}^{11}$ for two sets of basis functions integrated with a given quadrature

```
typedef struct q11_psi_phi   Q11_PSI_PHI;

struct q11_psi_phi
{
  const BAS_FCTS     *psi;
  const BAS_FCTS     *phi;
  const QUAD         *quad;

  const int   **n_entries;
  const REAL  ***values;
  const int   ***k;
  const int   ***l;
};
```

Description:

psi: pointer to the first set of basis functions;

phi: pointer to the second set of basis functions;

quad: pointer to the quadrature which is used for the integration;

n_entries: matrix of size `psi->n_bas_fcts` $\times$ `phi->n_bas_fcts` storing the count of non zero integrals;

`n_entries[i][j]` is the count of non zero values of $\hat{Q}^{11}_{\texttt{ij,kl}}$ ($0 \le$ `k,l` $\le$ `DIM`) for the pair (`psi[i]`,`phi[j]`), $0 \le$ `i` $<$ `psi->n_bas_fcts`, $0 \le$ `j` $<$ `phi->n_bas_fcts`;

values: tensor storing the non zero integrals;

`values[i][j]` is a vector of length `n_entries[i][j]` storing the non zero values for the pair (`psi[i]`,`phi[j]`);

k, l: tensor storing the indices k, l of the non zero integrals;
k[i][j] and l[i][j] are vectors of length n_entries[i][j] storing at k[i][j][r] and l[i][j][r] the indices k and l of the value stored at values[i][j][r], i.e.

$$\texttt{values[i][j][r]} = \hat{Q}^{11}_{\texttt{ij},\texttt{k[i][j][r]},\texttt{l[i][j][r]}} = \hat{Q}\left(\bar{\psi}^{\texttt{i}}_{,\lambda_{\texttt{k[i][j][r]}}}\ \bar{\varphi}^{\texttt{j}}_{,\lambda_{\texttt{l[i][j][r]}}}\right),$$

for $0 \le \texttt{r} < \texttt{n\_entries[i][j]}$. Using these pre–computed values we have for all elements S

$$\sum_{k,l=0}^{d} \bar{a}_{S,kl}\hat{Q}\left(\bar{\psi}^{i}_{,\lambda_k}\ \bar{\varphi}^{j}_{,\lambda_l}\right) = \sum_{\texttt{r=0}}^{\texttt{n\_entries[i][j]-1}} \bar{a}_{S,\texttt{k[i][j][r]},\texttt{l[i][j][r]}} \texttt{*values[i][j][r]}.$$

The following function initializes a Q11_PSI_PHI structure:

```
const Q11_PSI_PHI *get_q11_psi_phi(const BAS_FCTS *, const BAS_FCTS *,
                                   const QUAD *);
```

Description:

get_q11_psi_phi(psi, phi, quad): returns a pointer to a filled Q11_PSI_PHI structure.
psi is a pointer to the first set of basis functions, phi is a pointer to the second set of basis functions; if both are nil pointers, nothing is done and the return value is nil; if one of the pointers is a nil pointer, the structure is initialized using the same set of basis functions for the first and second set, i.e. phi = psi or psi = phi is used.
quad is a pointer to a quadrature for the approximation of the integrals; if quad is nil, then a quadrature which is exact of degree psi->degree+phi->degree-2 is used.
All used Q11_PSI_PHI structures are stored in a linked list and are identified uniquely by the members psi, phi, and quad, and for such a combination only one Q11_PSI_PHI structure is created during runtime;
First, get_q11_psi_phi() looks for a matching structure in the linked list; if such a structure is found a pointer to this structure is returned; the values are not computed a second time.
Otherwise a new structure is generated, linked to the list, and the values are computed using the quadrature quad; all values val with $|\texttt{val}| \le$ TOO_SMALL are treated as zeros.

2.12.2. Example. The following example shows how to use these pre–computed values for the integration of the 2nd order term

$$\int_{\hat{S}} \nabla_\lambda \bar{\psi}^i(\lambda(\hat{x})) \cdot \bar{A}(\lambda(\hat{x}))\, \nabla_\lambda \bar{\varphi}^j(\lambda(\hat{x}))\, d\hat{x}$$

for all i, j. We only show the body of a function for the integration and assume that LALt_fct returns a matrix storing $\bar{A}$ (compare the member LALt in the OPERATOR_INFO structure):

```
if (!q11_psi_phi)  q11_psi_phi = get_q11_psi_phi(psi, phi, quad[2]);

LALt = LALt_fct(el_info, quad, 0, user_data);
n_entries = q11_psi_phi->n_entries;

for (i = 0; i < psi->n_bas_fcts; i++)
{
  for (j = 0; j < phi->n__bas_fcts; j++)
  {
    k      = q11_psi_phi->k[i][j];
```

```
        l      = q11_psi_phi->l[i][j];
        values = q11_psi_phi->values[i][j];
        for (val = m = 0; m < n_entries[i][j]; m++)
          val += values[m]*LALt[k[m]][l[m]];
        mat[i][j] += val;
      }
    }
```

The values $\hat{Q}^{01}$ for the set of basis functions `psi` and `phi` are stored in

```
typedef struct q01_psi_phi   Q01_PSI_PHI;

struct q01_psi_phi
{
  const BAS_FCTS     *psi;
  const BAS_FCTS     *phi;
  const QUAD         *quad;

  const int   **n_entries;
  const REAL  ***values;
  const int   ***l;
};
```

Description:

`psi`: pointer to the first set of basis functions;

`phi`: pointer to the second set of basis functions;

`quad`: pointer to the quadrature which is used for the integration;

`n_entries`: matrix of size `psi->n_bas_fcts` $\times$ `phi->n_bas_fcts` storing the count of non zero integrals;

`n_entries[i][j]` is the count of non zero values of $\hat{Q}^{01}_{\mathtt{ij},\mathtt{l}}$ ($0 \le \mathtt{l} \le \mathtt{DIM}$) for the pair (`psi[i]`,`phi[j]`), $0 \le \mathtt{i} <$ `psi->n_bas_fcts`, $0 \le \mathtt{j} <$ `phi->n_bas_fcts`;

`values`: tensor storing the non zero integrals;

`values[i][j]` is a vector of length `n_entries[i][j]` storing the non zero values for the pair (`psi[i]`,`phi[j]`);

`l`: tensor storing the indices l of the non zero integrals;

`l[i][j]` is a vector of length `n_entries[i][j]` storing at `l[i][j][r]` the index `l` of the value stored at `values[i][j][r]`, i.e.

$$\mathtt{values[i][j][r]} = \hat{Q}^{01}_{\mathtt{ij},\mathtt{l[i][j][r]}} = \hat{Q}\left(\bar{\psi}^{\mathtt{i}}\,\bar{\varphi}^{\mathtt{j}}_{,\lambda_{\mathtt{l[i][j][r]}}}\right),$$

for $0 \le \mathtt{r} <$ `n_entries[i][j]`. Using these pre–computed values we have for all elements S

$$\sum_{l=0}^{d} \bar{b}^0_{S,l}\hat{Q}\left(\bar{\psi}^i\,\bar{\varphi}^j_{,\lambda_l}\right) = \sum_{\mathtt{r}=0}^{\mathtt{n\_entries[i][j]-1}} \bar{b}^0_{S,\mathtt{l[i][j][r]}}\,\mathtt{*values[i][j][r]}.$$

The following function initializes a `Q01_PSI_PHI` structure:

```
const Q01_PSI_PHI *get_q01_psi_phi(const BAS_FCTS *, const BAS_FCTS *,
                                   const QUAD *);
```

Description:

get_q01_psi_phi(psi, phi, quad): returns a pointer to a filled Q01_PSI_PHI structure.

psi is a pointer to the first set of basis functions phi is a pointer to the second set of basis functions; if both are nil pointers, nothing is done and the return value is nil; if one of the pointers is a nil pointer, the structure is initialized using the same set of basis functions for the first and second set, i.e. phi = psi or psi = phi is used.

quad is a pointer to a quadrature for the approximation of the integrals; is quad is nil, a quadrature which is exact of degree psi->degree+phi->degree-1 is used.

All used Q01_PSI_PHI structures are stored in a linked list and are identified uniquely by the members psi, phi, and quad, and for such a combination only one Q01_PSI_PHI structure is created during runtime;

First, get_q01_psi_phi() looks for a matching structure in the linked list; if such a structure is found a pointer to this structure is returned; the values are not computed a second time.

Otherwise a new structure is generated, linked to the list, and the values are computed using the quadrature quad; all values val with $|\texttt{val}| \leq$ TOO_SMALL are treated as zeros.

The values $\hat{Q}^{10}$ for the set of basis functions psi and phi are stored in

```
typedef struct q10_psi_phi   Q10_PSI_PHI;

struct q10_psi_phi
{
  const BAS_FCTS     *psi;
  const BAS_FCTS     *phi;
  const QUAD         *quad;

  const int   **n_entries;
  const REAL  ***values;
  const int   ***k;
};
```

Description:

psi: pointer to the first set of basis functions;

phi: pointer to the second set of basis functions;

quad: pointer to the quadrature which is used for the integration;

n_entries: matrix of size psi->n_bas_fcts $\times$ phi->n_bas_fcts storing the count of non zero integrals;

n_entries[i][j] is the count of non zero values of $\hat{Q}^{10}_{\mathtt{ij,k}}$ $(0 \leq \mathtt{k} \leq \mathtt{DIM})$ for the pair (psi[i],phi[j]), $0 \leq \mathtt{i} <$ psi->n_bas_fcts, $0 \leq \mathtt{j} <$ phi->n_bas_fcts;

values: tensor storing the non zero integrals;

values[i][j] is a vector of length n_entries[i][j] storing the non zero values for the pair (psi[i],phi[j]);

k: tensor storing the indices k of the non zero integrals;

k[i][j] is a vector of length n_entries[i][j] storing at k[i][j][r] the index k of the value stored at values[i][j][r], i.e.

$$\mathtt{values[i][j][r]} = \hat{Q}^{10}_{\mathtt{ij,k[i][j][r]}} = \hat{Q}\Big(\bar{\psi}^{\mathtt{i}}_{,\lambda_{\mathtt{k[i][j][r]}}}\,\bar{\varphi}^{\mathtt{j}}\Big),$$

for $0 \le \texttt{r} < \texttt{n\_entries[i][j]}$. Using these pre–computed values we have for all elements S

$$\sum_{k=0}^{d} \bar{b}^1_{S,k} \hat{Q}\left(\bar{\psi}^i_{,\lambda_k}\, \bar{\varphi}^j\right) = \sum_{\texttt{r}=0}^{\texttt{n\_entries[i][j]-1}} \bar{b}^1_{S,\texttt{k[i][j][r]}}\, \texttt{*values[i][j][r]}.$$

The following function initializes a Q10_PSI_PHI structure:

```
const Q10_PSI_PHI *get_q10_psi_phi(const BAS_FCTS *, const BAS_FCTS *,
                                   const QUAD *);
```

Description:

get_q10_psi_phi(psi, phi, quad): returns a pointer to a filled Q10_PSI_PHI structure.

psi is a pointer to the first set of basis functions phi is a pointer to the second set of basis functions; if both are nil pointers, nothing is done and the return value is nil; if one of the pointers is a nil pointer, the structure is initialized using the same set of basis functions for the first and second set, i.e. phi = psi or psi = phi is used.

quad is a pointer to a quadrature for the approximation of the integrals; is quad is nil, a quadrature which is exact of degree psi->degree+phi->degree-1 is used.

All used Q10_PSI_PHI structures are stored in a linked list and are identified uniquely by the members psi, phi, and quad, and for such a combination only one Q10_PSI_PHI structure is created during runtime;

First, get_q10_psi_phi() looks for a matching structure in the linked list; if such a structure is found a pointer to this structure is returned; the values are not computed a second time.

Otherwise a new structure is generated, linked to the list, and the values are computed using the quadrature quad; all values val with $|\texttt{val}| \le$ TOO_SMALL are treated as zeros.

Finally, the values $\hat{Q}^{00}$ for the set of basis functions psi and phi are stored in

```
typedef struct q00_psi_phi   Q00_PSI_PHI;

struct q00_psi_phi
{
  const BAS_FCTS     *psi;
  const BAS_FCTS     *phi;
  const QUAD         *quad;

  const REAL  **values;
};
```

Description:

psi: pointer to the first set of basis functions;

phi: pointer to the second set of basis functions;

quad: pointer to the quadrature which is used for the integration;

values: matrix storing the integrals;

$$\texttt{values[i][j]} = \hat{Q}^{00}_{\texttt{ij}} = \hat{Q}\left(\bar{\psi}^{\texttt{i}}\, \bar{\varphi}^{\texttt{j}}\right),$$

for the pair (psi[i],phi[j]), $0 \le \texttt{i} <$ psi->n_bas_fcts, $0 \le \texttt{j} <$ phi->n_bas_fcts.

The following function initializes a Q00_PSI_PHI structure:

```
const Q00_PSI_PHI *get_q00_psi_phi(const BAS_FCTS *, const BAS_FCTS *,
                                   const QUAD *);
```

Description:

`get_q00_psi_phi(psi, phi, quad)`: returns a pointer to a filled `Q00_PSI_PHI` structure.
`psi` is a pointer to the first set of basis functions `phi` is a pointer to the second set of basis functions; if both are `nil` pointers, nothing is done and the return value is `nil`; if one of the pointers is a `nil` pointer, the structure is initialized using the same set of basis functions for the first and second set, i.e. `phi = psi` or `psi = phi` is used.
`quad` is a pointer to a quadrature for the approximation of the integrals; is `quad` is `nil`, a quadrature which is exact of degree `psi->degree+phi->degree` is used.
All used `Q00_PSI_PHI` structures are stored in a linked list and are identified uniquely by the members `psi`, `phi`, and `quad`, and for such a combination only one `Q00_PSI_PHI` structure is created during runtime;
First, `get_q00_psi_phi()` looks for a matching structure in the linked list; if such a structure is found a pointer to this structure is returned; the values are not computed a second time.
Otherwise a new structure is generated, linked to the list, and the values are computed using the quadrature `quad`.

2.12.4 Data structures and functions for vector update

Besides the general routines `update_real_vec()` and `update_real_d_vec()`, this section presents also easy to use routines for calculation of L^2 scalar products between a given function and all basis functions of a finite element space.

The following structures hold full information for the assembling of element vectors. They are used by the functions `update_real_vec()` and `update_real_d_vec()` described below.

```
typedef struct el_vec_info    EL_VEC_INFO;
typedef struct el_vec_d_info  EL_VEC_D_INFO;

struct el_vec_info
{
  int              n_dof;
  const DOF_ADMIN  *admin;
  const DOF        *(*get_dof)(const EL *,const DOF_ADMIN *, DOF *);
  const S_CHAR     *(*get_bound)(const EL_INFO *, S_CHAR *);

  REAL             factor;

  const REAL       *(*el_vec_fct)(const EL_INFO *, void *);
  void             *fill_info;

  FLAGS            fill_flag;
};

struct el_vec_d_info
{
  int              n_dof;
  const DOF_ADMIN  *admin;
  const DOF        *(*get_dof)(const EL *,const DOF_ADMIN *, DOF *);
  const S_CHAR     *(*get_bound)(const EL_INFO *, S_CHAR *);
```

```
    REAL            factor;

    const REAL_D    *(*el_vec_fct)(const EL_INFO *, void *);
    void            *fill_info;

    FLAGS           fill_flag;
  };
```

Description:

n_dof: size of the element vector;

admin: pointer to a DOF_ADMIN structure for the administration of DOFs of the vector to be filled;

get_dof: pointer to a function for the access of the global DOFs on a single element; get_row_dof(el, admin, dof) returns a pointer to a vector of length n_dof storing the global DOFs; if dof is a nil pointer, get_dof() has to provide memory for storing this vector, which may be overwritten on the next call; otherwise the DOFs have to be stored at dof; (usually, get_dof() is the get_dof_indices() function inside a BAS_FCTS structure (compare Section 2.5.1);

get_bound: is an optional pointer to a function providing information about the boundary type of DOFs; if get_bound() is not nil, get_bound(el_info, bound) returns a pointer to a vector of length n_dof storing the boundary type of the local DOFs; this pointer is the optional pointer to a vector holding boundary information of the function add_element[_d]_vec() described above; if bound is a nil pointer, get_bound() has to provide memory for storing this vector, which may be overwritten on the next call; otherwise the DOFs have to be stored at bound; (usually, get_bound() is the get_bound() function inside a BAS_FCTS structure (compare Section 2.5.1);

factor: is a multiplier for the element contributions; usually factor is 1 or -1;

el_vec_fct: is a pointer to a function for the computation of the element vector; el_vec_fct(el_info, fill_info) returns a pointer to a REAL resp. REAL_D vector of length n_dof storing the element vector on element el_info->el; fill_info is a pointer to data needed by el_matrix_fct(); the function has to provide memory for storing the element vector, which can be overwritten on the next call;

fill_info: pointer to data needed by el_vec_fct(); is the second argument of this function;

fill_flag: the flag for the mesh traversal for assembling the vector.

The following function does the update of vectors by assembling element contributions during mesh traversal; information for computing the element vectors is hold in a EL_VEC[_D]_INFO structure:

```
void update_real_vec(DOF_REAL_VEC *, const EL_VEC_INFO *);
void update_real_d_vec(DOF_REAL_D_VEC *, const EL_VEC_D_INFO *);
```

update_real[_d]_vec(dr[d]v, info): updates the vector drv resp. drdv by traversing the underlying mesh and assembling the element contributions into the vector; information about the computation of element vectors and connection of local and global DOFs is stored in info; the flags for the mesh traversal of the mesh dr[d]v->fe_space->mesh are stored at info->fill_flags which specifies the elements to be visited and information that should be present on the elements for the calculation of the element vectors and boundary information (if `info->get_bound` is not `nil`);

on the elements, information about the global DOFs is accessed by `info->get_dof` using `info->admin`; the boundary type of the DOFs is accessed by `info->get_bound` if `info->get_bound` is not a `nil` pointer; then the element vector is computed by `info->el_vec_fct(el_info, info->fill_info)`; these contributions are finally added to `dr[d]v` multiplied by `info->factor` by a call of `add_element[_d]_vec()` with all information about global DOFs, the element vector, and boundary types, if available;

`update_real[_d]_vec()` only adds element contributions; this makes several calls for the assemblage of one vector possible; before the first call, the vector should be set to zero by a call of `dof_set[_d](0.0, dr[d]v)`.

L^2 scalar products In many applications, the load vector is just the L^2 scalar product of a given function with all basis functions of the finite element space or this scalar product is a part of the right hand side; such a scalar product can be directly assembled by the functions

```
void L2scp_fct_bas(REAL (*)(REAL_D), const QUAD *, DOF_REAL_VEC *);
void L2scp_fct_bas_d(const REAL *(*)(const REAL_D, REAL_D), const QUAD *,
                     DOF_REAL_D_VEC *);
```

Description:

`L2scp_fct_bas(f, quad, fh)`: approximates the L^2 scalar products of a given function with all basis functions by numerical quadrature and adds the corresponding values to a DOF vector;

`f` is a pointer for the evaluation of the given function in world coordinates x and returns the value of that function at x; if `f` is a `nil` pointer, nothing is done;

`fh` is the DOF vector where at the `i`th entry the approximation of the L^2 scalar product of the given function with the `i`th global basis function of `fh->fe_space` is stored;

`quad` is the quadrature for the approximation of the integral on each leaf element of `fh->fe_space->mesh`; if `quad` is a `nil` pointer, a default quadrature which is exact of degree `2*fh->fe_space->bas_fcts->degree-2` is used.

The integrals are approximated by looping over all leaf elements, computing the approximations to the element contributions and adding these values to the vector `fh` by `add_element_vec()`.

The vector `fh` is *not* initialized with `0.0`; only the new contributions are added.

`L2scp_fct_bas_d(fd, quad, fhd)`: approximates the L^2 scalar products of a given vector valued function with all scalar valued basis functions by numerical quadrature and adds the corresponding values to a vector valued DOF vector;

`fd` is a pointer for the evaluation of the given function in world coordinates x; `fd(x, fx)` returns a pointer to a vector storing the value at `x`; if `fx` is not `nil`, the value is stored at `fx` otherwise the function has to provide memory for storing this vector, which can be overwritten on the next call; if `fd` is a `nil` pointer, nothing is done;

`fhd` is the DOF vector where at the `i`th entry (a `REAL_D` vector) the approximation of the L^2 scalar product of the given vector valued function with the `i`th global (scalar valued) basis function of `fhd->fe_space` is stored;

`quad` is the quadrature for the approximation of the integral on each leaf element of `fhd->fe_space->mesh`; if `quad` is a `nil` pointer, a default quadrature which is exact of degree `2*fhd->fe_space->bas_fcts->degree-2` is used.

The integrals are approximated by looping over all leaf elements, computing the approximations to the element contributions and adding these values to the vector `fhd` by `add_element_d_vec()`. The vector `fhd` is *not* initialized with `(0.0,... ,0.0)`; only the new contributions are added.

2.12.5 Dirichlet boundary conditions

For the solution of the discrete system (1.37) derived in Section 1.4.5 we have to set the Dirichlet boundary values for all Dirichlet DOFs. Usually, we take for the approximation g_h of g the interpolant of g, i.e. $g_h = I_h g$ and we have to copy the coefficients of g_h at the Dirichlet DOFs to the load vector (compare (1.36)). Additionally we know that the discrete solution has also the same coefficients at Dirichlet DOFs. Using an iterative solver we should use such information for the initial guess. Copying the coefficients of g_h at the Dirichlet DOFs to the initial guess will result in an initial residual (and then for all subsequent residuals) which is (are) zero at all Dirichlet DOFs.

Furthermore, the matrix we have derived in (1.35) (and which is assembled in this way by the assemblage tools) is not symmetric even for the discretization of symmetric and elliptic operators. Applying directly the conjugate gradient method for solving (1.37) will not work, because the matrix is not symmetric. But setting the Dirichlet boundary values also in the initial guess at Dirichlet DOFs, all residuals are zero at Dirichlet DOFs and thus the conjugate gradient method will only apply to the non Dirichlet DOFs, which means the conjugate gradient method will only "see" the symmetric and positive definite part of the matrix.

The following functions will set Dirichlet boundary values for all DOFs on the Dirichlet boundary, using an interpolation of the boundary values g:

```
void dirichlet_bound(REAL (*)(REAL_D), DOF_REAL_VEC *, DOF_REAL_VEC *,
                     DOF_SCHAR_VEC *);
void dirichlet_bound_d(const REAL *(*)(const REAL_D, REAL_D), DOF_REAL_D_VEC *,
                       DOF_REAL_D_VEC *, DOF_SCHAR_VEC *);
```

Description:

`dirichlet_bound(g, fh, uh, bound)`: sets Dirichlet boundary values for all DOFs on all leaf elements of `fh->fe_space->mesh` or `uh->fe_space->mesh`; values at DOFs not belonging to the Dirichlet boundary are not changed by this function.

`g` is a pointer to a function for the evaluation of the boundary data; if `g` is a `nil` pointer, all coefficients at Dirichlet DOFs are set to `0.0`.

`fh` and `uh` are vectors where Dirichlet boundary values should be set (usually, `fh` stores the load vector and `uh` an initial guess for an iterative solver); one of `fh` and `uh` may be a `nil` pointer; if both arguments are `nil` pointers, nothing is done; if both arguments are not `nil`, `fh->fe_space` must equal `uh->fe_space`.

Boundary values are set element–wise on the leaf elements. The boundary type is accessed on an element by the function `fe_space->bas_fcts->get_bound()`, where `fe_space` is the finite element space of `fh` or `uh`; this function extracts information about the boundary type of a DOF from the `EL_INFO` structure on single elements; interpolation is then done by the function `fe_space->bas_fcts->interpol()` for all Dirichlet DOFs on the element; the function `g` must fit to the needs of this function; for Lagrange elements, `(*g)()` is evaluated for all Lagrange nodes on the Dirichlet boundary and has to return the values at these nodes (compare Section 2.5.1); the flag of the mesh traversal is `CALL_LEAF_EL|FILL_BOUND|FILL_COORDS`.

`bound` is a vector for storing the boundary type for each used DOF; `bound` may be `nil`; if it is not `nil`, the `i`th entry of the vector is filled with the boundary type of the `i`th DOF. `bound->fe_space` must be the same as `fh`'s or `uh`'s `fe_space`.

dirichlet_bound_d(gd, fhd, uhd, bound): sets vector valued Dirichlet boundary values for all DOFs on all leaf elements of fhd->fe_space->mesh or uhd->fe_space->mesh; values at DOFs not belonging to the Dirichlet boundary are not changed by this function.

gd is a pointer to a function for the evaluation of boundary data; the return value is a pointer to a vector storing the values; if the second argument val of (*gd)(x, val) is not nil, the values have to be stored at val, otherwise gd has to provide memory for the vector which may be overwritten on the next call; if gd is a nil pointer, all coefficients at Dirichlet DOFs are set to (0.0, ..., 0.0).

fhd and uhd are DOF vectors where Dirichlet boundary values should be set (usually, fhd stores the load vector and uhd an initial guess for an iterative solver); one of fhd and uhd may be a nil pointer; if both arguments are nil pointers, nothing has is done; if both arguments are not nil pointers, fhd->fe_space must equal uhd->fe_space.

Boundary values are set element–wise on the leaf element. The boundary type is accessed on an element by the function fe_space->bas_fcts->get_bound(), where fe_space is the finite element space of fhd or uhd; this function extracts information about the boundary type of a DOF from the EL_INFO structure on single elements; interpolation is then done by the function fe_space->bas_fcts->interpol() for all Dirichlet DOFs on the element; the function gd must fit to the needs of this function; for Lagrange elements, (*gd)() is evaluated for all Lagrange nodes on the Dirichlet boundary and has to return the values at these nodes (compare Section 2.5.1); the flag of the mesh traversal is CALL_LEAF_EL|FILL_BOUND|FILL_COORDS.

bound is a vector for the storing boundary type for each used DOF; bound may be nil; if it is not nil, the ith entry of the vector is filled with the boundary type of the ith DOF. bound->fe_space must be the same as fh's or uh's fe_space.

2.12.6 Interpolation into finite element spaces

In time dependent problems, usually the "solve" step in the adaptive method for the adaptation of the initial grid is an interpolation of initial data to the finite element space, i.e. a DOF vector is filled with the coefficient of the interpolant. The following functions are implemented for this task:

```
void interpol(REAL (*)(REAL_D), DOF_REAL_VEC *);
void interpol_d(const REAL *(*)(const REAL_D, REAL_D), DOF_REAL_D_VEC *);
```

Description:

interpol(f, fh): computes the coefficients of the interpolant of a function and stores these in a DOF vector;

f is a pointer to a function for the evaluation of the function to be interpolated; if f is a nil pointer, all coefficients are set to 0.0.

fh is a DOF vector for storing the coefficients; if fh is a nil pointer, nothing is done.

Interpolation is done element–wise on the leaf elements of fh->fe_space->mesh; the element interpolation is done by the function fh->fe_space->bas_fcts->interpol(); the fill_flag of the mesh traversal is CALL_LEAF_EL|FILL_COORDS and the function f must fit to the needs of fh->fe_space->bas_fcts->interpol(); for Lagrange elements, (*f)() is evaluated for all Lagrange nodes on the element and has to return the value at these nodes (compare Section 2.5.1).

interpol_d(fd, fhd): computes the coefficients of the interpolant of a vector valued function and stores these in a DOF vector;

fd is a pointer to a function for the evaluation of the function to be interpolated; the return value is a pointer to a vector storing the values; if the second argument fx of (*fd)(x, fx) is not nil, the values have to be stored at fx, otherwise fd has to provide memory for the vector which may be overwritten on the next call; if fd is a nil pointer, all coefficients are set to (0.0,...,0.0).

fhd is a DOF vector for storing the coefficients; if fhd is a nil pointer, nothing is done.

Interpolation is done element–wise on the leaf elements of fhd->fe_space->mesh; the element interpolation is done by the function fhd->fe_space->bas_fcts->interpol_d(); the fill_flag of the mesh traversal is CALL_LEAF_EL|FILL_COORDS and the function fd must fit to the needs of fhd->fe_space->bas_fcts->interpol_d(); for Lagrange elements, (*fd)() is evaluated for all Lagrange nodes on the element and has to return the value at these nodes (compare Section 2.5.1).

2.13 Data structures and procedures for adaptive methods

2.13.1 ALBERT adaptive method for stationary problems

ALBERT provides a subroutine

```
void adapt_method_stat(MESH *mesh, ADAPT_STAT *adapt);
```

which implements the adaptive procedure described in Section 1.5.1. In order to provide flexibility in applications, it uses a structure ADAPT_STAT to pass several parameters, which are described below. Some of the parameters are optional or used only when a special marking strategy is selected.

Additionally, the ADAPT_STAT structure contains pointers to problem dependent routines to build the linear or nonlinear system(s) of equations corresponding to an adapted mesh, and to a routine which solves the discrete problem and computes the new discrete solution. For flexibility and efficiency reasons, building and solution of the system(s) may be split into several parts, which are called at various stages of the mesh adaption process.

```
typedef struct adapt_stat      ADAPT_STAT;

struct adapt_stat
{
  const char  *name;
  REAL        tolerance;
  REAL        p;                      /* power in estimator norm        */
  int         max_iteration;
  int         info;

  REAL   (*estimate)(MESH *mesh, ADAPT_STAT *adapt);
  REAL   (*get_el_est)(EL *el);       /* local error indicator          */
  REAL   (*get_el_estc)(EL *el);      /* local coarsening error         */
  U_CHAR (*marking)(MESH *mesh, ADAPT_STAT *adapt);

  void   *est_info;                   /* estimator parameters           */
  REAL   err_sum, err_max;            /* sum and max of el_est          */

  void   (*build_before_refine)(MESH *mesh, U_CHAR flag);
```

```
  void   (*build_before_coarsen)(MESH *mesh, U_CHAR flag);
  void   (*build_after_coarsen)(MESH *mesh, U_CHAR flag);
  void   (*solve)(MESH *mesh);

  int    refine_bisections;
  int    coarsen_allowed;              /* 0 : 1                           */
  int    coarse_bisections;

  int    strategy;                     /* 1=GR, 2=MS, 3=ES, 4=GERS        */
  REAL   MS_gamma, MS_gamma_c;         /*  maximum strategy               */
  REAL   ES_theta, ES_theta_c;         /*  equidistribution strategy      */
  REAL   GERS_theta_star, GERS_nu, GERS_theta_c;  /* GERS strategy        */
};
```

The entries yield following information:

tolerance: given tolerance for the (absolute or relative) error,

p: power p used in estimate (1.49), $1 \leq p < \infty$;

max_iteration: maximal allowed number of iterations of the adaptive procedure; if `max_iteration <= 0`, no iteration bound is used;

info: level of information printed during the adaptive procedure; if `info >= 2`, the iteration count and final error estimate are printed; if `info >= 4`, then information is printed after each iteration of the adaptive procedure; if `info >= 6`, additional information about the CPU time used for mesh adaption and building the linear systems is printed;

estimate: pointer to a function which calculates the local error indicators and sets the values for the entries `err_sum` and `err_max`; return value is the sum of local error estimates

$$\left(\sum_{S\in\mathcal{S}} \eta_S(u_h)^p\right)^{1/p};$$

get_el_est: pointer to a subroutine which returns the local error estimate $\eta_S(u_h)^p$ on one element; will be called by `marking()`;

get_el_estc: pointer to a subroutine which returns the local coarsening error $\eta_{c,S}(u_h)^p$ on one element; will be called by `marking()`; this pointer may be `nil`, which means $\eta_{c,S}(u_h) = 0$;

marking: pointer to a subroutine that marks elements for refinement or coarsening, depending on local error indicators; if `marking==nil`, the standard ALBERT marking routine is used, which uses the `strategy` entry to select a marking strategy and pay attention to the `refine_bisections`, `coarsen_allowed`, and `coarse_bisections` entries; return value is a flag which tells if any elements are marked for refinement/coarsening (using the symbolic bit flags `MESH_REFINED` and `MESH_COARSENED`);

est_info: pointer to (problem dependent) parameters for the `estimate()` routine; via this pointer the user can pass information to the estimate routine; this pointer may be `nil`;

err_sum: variable to hold the sum of local error indicators $\left(\sum_{S\in\mathcal{S}} \eta_S(u_h)^p\right)^{1/p}$; the value for this entry is set by the user–defined function `estimate()`;

err_max: variable to hold the maximal local error indicators $\max_{S\in\mathcal{S}} \eta_S(u_h)^p$; the value for this entry is set by the user–defined function `estimate()`;

`build_before_refine`: pointer to a subroutine that may build parts of the (non-)linear system(s) before adapting the mesh, the second parameter tells if refinement or coarsening will be done; this pointer may be `nil`;

`build_before_coarsen`: pointer to a subroutine that may build parts of the (non-)linear system(s) after refining, before coarsening the mesh, the second parameter tells if refinement was done and coarsening will be done; this pointer may be `nil`;

`build_after_coarsen`: pointer to a subroutine that may build parts of the (non-)linear system(s) after adapting the mesh, the second parameter tells if refinement or coarsening were done; this pointer may be `nil`;

`solve`: pointer to a subroutine which solves the (non-)linear system(s) and calculates a new solution on the adapted mesh; this pointer may be `nil`;

`refine_bisections`: parameter to tell the marking strategy how many bisections should be performed when an element is marked for refinement; usually the value is 1 or `DIM`;

`coarsen_allowed`: flag to allow or forbid mesh coarsening;

`coarse_bisections`: parameter to tell the marking strategy how many bisections should be undone when an element is marked for coarsening; usually the value is 1 or `DIM`;

`strategy`: parameter to select a marking strategy; for the standard ALBERT marking routine, the possible values are:

- `0`: no mesh adaption,
- `1`: global refinement (GR),
- `2`: maximum strategy (MS),
- `3`: equidistribution strategy (ES),
- `4`: guaranteed error reduction strategy (GERS),

see Section 2.13.2;

`MS_gamma, MS_gamma_c`: parameters for the marking *maximum strategy*, see Sections 1.5.2 and 1.5.3;

`ES_theta, ES_theta_c`: parameters for the marking *equidistribution strategy*, see Sections 1.5.2 and 1.5.3;

`GERS_theta_star, GERS_nu, GERS_theta_c`: parameters for the marking *guaranteed error reduction strategy*, see Sections 1.5.2 and 1.5.3.

Besides parameters given directly in the `ADAPT_STAT` structure, the adaptive procedure gets main information from problem dependent subroutines

```
REAL  estimate(MESH *mesh, ADAPT_STAT *adapt);
```

and

```
REAL  get_el_est(EL *el);
REAL  get_el_estc(EL *el);
```

Pointers to these routines are passed via the `ADAPT_STAT` structure. The `estimate()` routine has to compute error estimates, fill the `ADAPT_STAT` entries `err_sum` and `err_max` with

$$\texttt{err\_sum} = \left(\sum_{S\in\mathcal{S}_h} \eta_S(u_h)^p\right)^{1/p}, \quad \texttt{err\_max} = \max_{S\in\mathcal{S}_h} \eta_S(u_h)^p,$$

and return the value $\eta(u_h)$ = **err_sum**, the total error estimate. Additional parameters for the **estimate()** routine can be passed via an **est_info** pointer to a (problem dependent) parameter structure, which may contain a switch for absolute or relative error, or selection of an error norm to estimate (like L^2– or H^1–norm). Usually, the local error indicators are stored in **LEAF_DATA(el)**, which is problem dependent and thus not directly accessible by general–purpose routines. The routine **get_el_est()** gives access to the local error indicators $\eta_S(u_h)^p$, which are needed by the marking strategies. **get_el_estc()** returns the *coarsening error* estimate $\eta_{c,S}(u_h)^p$, see Section 1.5.3 below.

Several marking strategies are already implemented in ALBERT, see Sections 1.5.2, 1.5.3, and 2.13.2. Additional strategies may be implemented and used by using the **marking** pointer of **ADAPT_STAT**.

The routine **adapt_method_stat()** now implements the whole adaptive procedure for a stationary problem, using the parameters given in **ADAPT_STAT**:

```
void adapt_method_stat(MESH *mesh, ADAPT_STAT *adapt)
{
  FUNCNAME("adapt_method_stat");
  int      iter;
  REAL     est;

  ...

  /* get solution on initial mesh */
  if (adapt->build_before_refine)  adapt->build_before_refine(mesh, 0);
  if (adapt->build_before_coarsen) adapt->build_before_coarsen(mesh, 0);
  if (adapt->build_after_coarsen)  adapt->build_after_coarsen(mesh, 0);
  if (adapt->solve) adapt->solve(mesh);
  est = adapt->estimate(mesh, adapt);

  for (iter = 0;
       (est > adapt->tolerance) &&
         ((adapt->max_iteration <= 0) || (iter < adapt->max_iteration));
       iter++)
  {
    if (adapt_mesh(mesh, adapt))
    {
      if (adapt->solve) adapt->solve(mesh);
      est = adapt->estimate(mesh, adapt);
    }

    ...
  }
}
```

The actual mesh adaption is done in a (static) subroutine **adapt_mesh()**, which combines marking, refinement, coarsening and the linear system building routines:

```
static U_CHAR adapt_mesh(MESH *mesh, ADAPT_STAT *adapt)
{
  FUNCNAME("adapt_mesh");
  U_CHAR   flag = 0;
  U_CHAR   mark_flag;
```

```
  ...

  if (adapt->marking)
    mark_flag = adapt->marking(mesh, adapt);
  else
    mark_flag = marking(mesh, adapt);            /* use standard marking() */

  if (!adapt->coarsen_allowed)
    mark_flag &= MESH_REFINED;                   /* use refine mark only   */

  if (adapt->build_before_refine)  adapt->build_before_refine(mesh, mark_flag);

  if (mark_flag & MESH_REFINED)    flag = refine(mesh);

  if (adapt->build_before_coarsen) adapt->build_before_coarsen(mesh, mark_flag);

  if (mark_flag & MESH_COARSENED)  flag |= coarsen(mesh);

  if (adapt->build_after_coarsen)  adapt->build_after_coarsen(mesh, flag);

  ...
  return(flag);
}
```

As the same procedure is used for time dependent problems, the different pointers to routines for building parts of the (non-)linear systems make it possible, for example, to compute the right hand side including a functional involving the solution from the old time step *before* coarsening the mesh, and then using the DOF_VEC restriction during coarsening to compute exactly the projection to the coarsened finite element space, without loosing any information.

2.13.1. Remark. For time dependent problems, usually the system matrices depend on the current time step size. Thus, matrices may have to be rebuilt even if meshes are not changed, but when the time step size was changed. Such changes can be detected in the set_time() routine, for example.

2.13.2 Standard ALBERT marking routine

When the marking procedure pointer in the ADAPT_STAT structure is nil, the standard ALBERT marking routine is called. By the strategy entry, five different marking strategies may be selected (compare Sections 1.5.2 and 1.5.3):

0: No refinement or coarsening is performed;

1: Global refinement (GR):
the mesh is refined globally, no coarsening is performed;

2: Maximum strategy (MS):
the entries MS_gamma, MS_gamma_c are used as refinement and coarsening parameters;

3: Equidistribution strategy (ES):
the entries ES_theta, ES_theta_c are used as refinement and coarsening parameters;

4: Guaranteed error reduction strategy (GERS):
   the entries `GERS_theta_star`, `GERS_nu`, and `GERS_theta_c` are used as refinement and coarsening parameters.

Coarsening is performed only when `coarsen_allowed` is true.

As `get_el_est()` and `get_el_estc()` return the p–th power of the local estimates, all algorithms are implemented to use the values η_S^p instead of η_S. This results in a small change to the coarsening tolerances, which use the inequality $\eta_S^p + \eta_{c,S}^p \leq c^p\, tol^p/N_k$ instead of $\eta_S + \eta_{c,S} \leq c\, tol/N_k^{1/p}$.

2.13.3 ALBERT adaptive method for time dependent problems

The space and time adaptive procedure described in Section 1.5.4 is implemented in ALBERT in the subroutine

```
void adapt_method_instat(MESH *mesh, ADAPT_INSTAT *adapt);
```

In order to provide flexibility in applications, it uses a structure `ADAPT_INSTAT` to pass several parameters, which are described below. Some of the parameters are optional. As adaption in space is based on the `adapt_method_stat()` routine described in Section 2.13.1, the `ADAPT_INSTAT` structure includes some `ADAPT_STAT` parameter structures.

```
typedef struct adapt_instat     ADAPT_INSTAT;
struct adapt_instat
{
  const char  *name;

  ADAPT_STAT adapt_initial[1];
  ADAPT_STAT adapt_space[1];

  REAL   time;
  REAL   start_time, end_time;
  REAL   timestep;

  void   (*init_timestep)(MESH *mesh, ADAPT_INSTAT *adapt);
  void   (*set_time)(MESH *mesh, ADAPT_INSTAT *adapt);
  void   (*one_timestep)(MESH *mesh, ADAPT_INSTAT *adapt);
  REAL   (*get_time_est)(MESH *mesh, ADAPT_INSTAT *adapt);
  void   (*close_timestep)(MESH *mesh, ADAPT_INSTAT *adapt);

  int    strategy;
  int    max_iteration;

  REAL   tolerance;
  REAL   rel_initial_error;
  REAL   rel_space_error;
  REAL   rel_time_error;
  REAL   time_theta_1;
  REAL   time_theta_2;
  REAL   time_delta_1;
  REAL   time_delta_2;
  int    info;
};
```

The entries yield following information:

adapt_initial: mesh adaption parameters for initial mesh, compare Section 2.13.1;

adapt_space: mesh adaption parameters during time steps, compare Section 2.13.1;

time: actual time, end of time interval for current time step;

start_time: initial time;

end_time: final time T;

timestep: current time step size;

init_timestep: pointer to a routine called at beginning of each time step, may be `nil`;

set_time: pointer to a routine called after changes of entry `time`, may be `nil`;

one_timestep: pointer to a routine which implements one (maybe adaptive) time step, if `nil`, a default routine will be called;

get_time_est: pointer to a routine which returns the estimate η_τ, may be `nil`;

close_timestep: pointer to a routine called after finishing each time step, may be `nil`;

strategy: parameter for the default `one_timestep` routine;

max_iteration: parameter for the default `one_timestep` routine;

tolerance: given total error tolerance *tol*;

rel_initial_error: portion Γ_0 of *tol* allowed for initial error;

rel_space_error: portion Γ_h of *tol* allowed for error from spatial discretization in each time step;

rel_time_error: portion Γ_h of *tol* allowed for error from time discretization in each time step;

time_theta_1: parameter θ_1 used in time step control;

time_theta_2: parameter θ_2 used in time step control;

time_delta_1: parameter δ_1 used in time step reduction;

time_delta_2: parameter δ_2 used in time step enlargement;

info: level of information produced by the adaption routine.

Implementation of the routine is very simple. All essential work is done by calling `adapt_method_stat()` for the generation of the initial mesh, based on parameters given in `adapt->adapt_initial` with tolerance `adapt->tolerance*adapt->rel_space_error`, and in `one_timestep()` which solves the discrete problem and does mesh adaption and time step adjustment for one single time step.

```
void adapt_method_instat(MESH *mesh, ADAPT_INSTAT *adapt)
{
  FUNCNAME("adapt_method_instat");

  TEST_EXIT(adapt)("no ADAPT_INSTAT\n");

  /****************************************************************************/
  /*  adaptation of the initial grid: done by adapt_method_stat()            */
  /****************************************************************************/

  adapt->time = adapt->start_time;
  if (adapt->set_time) adapt->set_time(mesh, adapt);

  adapt->adapt_initial->tolerance
```

```
      = adapt->tolerance * adapt->rel_initial_error;
    adapt_method_stat(mesh, adapt->adapt_initial);

    if (adapt->close_timestep)
      adapt->close_timestep(mesh, adapt);

    /*********************************************************************/
    /*  adaptation of timestepsize and mesh: done by one_timestep()     */
    /*********************************************************************/

    while (adapt->time < adapt->end_time)
    {
      if (adapt->init_timestep)
        adapt->init_timestep(mesh, adapt);

      if (adapt->one_timestep)
        adapt->one_timestep(mesh, adapt);
      else
        one_timestep(mesh, adapt);

      if (adapt->close_timestep)
        adapt->close_timestep(mesh, adapt);
    }
  }
```

Default `one_timestep()` routine

The default `one_timestep()` routine provided by ALBERT implements both the explicit and the implicit time strategy A. The semi–implicit strategy described in Section 1.5.4 is only a special case of the implicit strategy with a limited number of iterations (exactly one).

The routine uses the parameter `adapt->strategy` to select the strategy:

`strategy 0`: Explicit strategy, `strategy 1`: Implicit strategy.

Explicit strategy. The explicit strategy does one adaption of the mesh based on the error estimate computed from the last time step's discrete solution by using parameters given in `adapt->adapt_space`, with `tolerance` set to `adapt->tolerance*adapt->rel_space_error`. Then the current time step's discrete problem is solved, and the error estimators are computed. No time step size adjustment is done.

Implicit strategy. The implicit strategy starts with the old mesh from last time step. Using parameters given in `adapt->adapt_space`, the discrete problem is solved on the current mesh. Error estimates are computed, and time step size and mesh are adjusted, as shown in Algorithm 1.5.9, with tolerances given by `adapt->tolerance*adapt->rel_time_error` and `adapt->tolerance*adapt->rel_space_error`, respectively. This is iterated until the given error bounds are reached, or until `adapt->max_iteration` is reached.

With parameter `adapt->max_iteration==0`, this is equivalent to the semi–implicit strategy described in Section 1.5.4.

2.13.4 Initialization of data structures for adaptive methods

ALBERT provides functions for the initialization of structures ADAPT_STAT and ADAPT_INSTAT. Both functions do *not* fill any function pointer which is a member of the structure! These function pointers have to be adjusted by the user.

```
ADAPT_STAT *get_adapt_stat(const char *, const char *, int, ADAPT_STAT *);
ADAPT_INSTAT *get_adapt_instat(const char *, const char *, int, ADAPT_INSTAT *);
```

Description:

get_adapt_stat(name, prefix, info, adapt): returns a pointer to a partly initialized ADAPT_STAT structure; if the argument adapt is nil, a new structure is created, the name name is duplicated at the name entry of the structure, if name is not nil; if name is nil, and prefix is not nil, this string is duplicated at the name entry; for a newly created structure, all function pointers of the structure are initialized with nil; all other members are initialized with some default value; if the argument adapt is not nil, no default initialization takes place this part is skipped if adapt is not nil;

if prefix is not nil, get_adapt_stat() tries then to initialize members by a call of GET_PARAMETER(), where the key for each member is value(prefix)->member name; the argument info is the first argument of GET_PARAMETER() giving the level of information for the initialization;

only the parameters for the actually chosen strategy are initialized by a call of GET_PARAMETER(): for strategy == 2 only MS_gamma and MS_gamma_c, for strategy == 3 only ES_theta and ES_theta_c, and for strategy == 3 only GERS_theta_star, GERS_nu, and GERS_theta_c;

since the parameter tools are used for the initialization, get_adapt_stat() should be called *after* the initialization of all parameters; there may be no initializer in the parameter file(s) for some member, if the default value should be used; if info is not zero and there is no initializer for some member this will result in an error message by GET_PARAMETER() which can be ignored;

Table 2.11 shows the initialized members, the default values and the key for the initialization by GET_PARAMETER();

member	default	parameter key
tolerance	1.0	prefix->tolerance
p	2	prefix->p
max_iteration	30	prefix->max_iteration
info	2	prefix->info
refine_bisections	DIM	prefix->refine_bisections
coarsen_allowed	0	prefix->coarsen_allowed
coarse_bisections	DIM	prefix->coarse_bisections
strategy	1	prefix->strategy
MS_gamma	0.5	prefix->MS_gamma
MS_gamma_c	0.1	prefix->MS_gamma_c
ES_theta	0.9	prefix->ES_theta
ES_theta_c	0.2	prefix->ES_theta_c
GERS_theta_star	0.6	prefix->GERS_theta_star
GERS_nu	0.1	prefix->GERS_nu
GERS_theta_c	0.1	prefix->GERS_theta_c

Table 2.11: Initialized members of an ADAPT_STAT structure, the default values and the key for the initialization by GET_PARAMETER().

`get_adapt_instat(name, prefix, info, adapt)`: returns a pointer to a partly initialized `ADAPT_INSTAT` structure; if the argument `adapt` is `nil`, a new structure is created, the name `name` is duplicated at the name entry of the structure, if `name` is not `nil`; if `name` is `nil`, and `prefix` is not `nil`, this string is duplicated at the name entry; for a newly created structure, all function pointers of the structure are initialized with `nil`; all other members are initialized with some default value; if the argument `adapt` is not `nil`, no default initialization takes place this part is skipped if `adapt` is not `nil`;

if `prefix` is not `nil`, `get_adapt_instat()` tries then to initialize members by a call of `GET_PARAMETER()`, where the key for each member is `value(prefix)->member name`; the argument `info` is the first argument of `GET_PARAMETER()` giving the level of information for the initialization;

Table 2.12 shows the initialized members, the default values and the key for the initialization by `GET_PARAMETER()`;

2.14 Implementation of error estimators

2.14.1 Error estimator for elliptic problems

ALBERT provides a residual type error estimator for quasi–linear elliptic problems of the type

$$\begin{aligned} -\nabla \cdot A\nabla u(x) + f\Big(x, u(x), \nabla u(x)\Big) &= 0 && x \in \Omega, \\ u(x) &= 0 && x \in \Gamma_D, \\ \nu \cdot A\nabla u(x) &= 0 && x \in \Gamma_N, \end{aligned}$$

where $A \in \mathbb{R}^{d\times d}$ is a positive definite matrix and $\partial\Omega = \Gamma_D \cup \Gamma_N$.

Verfürth [38] proved for this kind of equation under suitable assumptions on f, u and u_h (in the non–linear case) the following estimate

$$\begin{aligned} \|u - u_h\|^2_{H^1(\Omega)} \le \sum_{S\in\mathcal{S}} & C_0^2\, h_S^2\, \| -\nabla \cdot A\nabla u_h + f(.,u_h,\nabla u_h)\|^2_{L^2(S)} \\ & + C_1^2 \sum_{\Gamma\subset\partial S\cap\Omega} h_S\, \|\, [\![A\nabla u_h]\!]\, \|^2_{L^2(\Gamma)}, + C_1^2 \sum_{\Gamma\subset\partial S\cap\Gamma_N} h_S\, \|\nu \cdot A\nabla u_h\|^2_{L^2(\Gamma)}, \end{aligned}$$

where $[\![.]\!]$ denotes the jump of the normal component across an interior edge/face $\Gamma \subset \partial S$.

Bänsch and Siebert [4] proved a similar L^2 error estimate for semi–linear problems, i.e. $f = f(x,u)$, namely

$$\begin{aligned} \|u - u_h\|^2_{L^2(\Omega)} \le \sum_{S\in\mathcal{S}} & C_0^2\, h_S^4\, \| -\nabla \cdot A\nabla u_h + f(.,u_h)\|^2_{L^2(S)} \\ & + C_1^2 \sum_{\Gamma\subset\partial S\cap\Omega} h_S^3\, \|\, [\![A\nabla u_h]\!]\, \|^2_{L^2(\Gamma)} + C_1^2 \sum_{\Gamma\subset\partial S\cap\Gamma_N} h_S^3\, \|\nu \cdot A\nabla u_h\|^2_{L^2(\Gamma)}. \end{aligned}$$

The following function is an implementation of the above estimators:

```
REAL ellipt_est(const DOF_REAL_VEC *, ADAPT_STAT *,
          REAL *(*)(EL *), REAL *(*)(EL *), int, int, REAL[3],const REAL_DD,
          REAL (*f)(const EL_INFO *, const QUAD *, int, REAL,const REAL_D),
          FLAGS);
```

member	default	parameter key
adapt_initial->tolerance	1.0	prefix->initial->tolerance
adapt_initial->p	2	prefix->initial->p
adapt_initial->max_iteration	30	prefix->initial->max_iteration
adapt_initial->info	2	prefix->initial->info
adapt_initial->refine_bisections	DIM	prefix->initial->refine_bisections
adapt_initial->coarsen_allowed	0	prefix->initial->coarsen_allowed
adapt_initial->coarse_bisections	DIM	prefix->initial->coarse_bisections
adapt_initial->strategy	1	prefix->initial->strategy
adapt_initial->MS_gamma	0.5	prefix->initial->MS_gamma
adapt_initial->MS_gamma_c	0.1	prefix->initial->MS_gamma_c
adapt_initial->ES_theta	0.9	prefix->initial->ES_theta
adapt_initial->ES_theta_c	0.2	prefix->initial->ES_theta_c
adapt_initial->GERS_theta_star	0.6	prefix->initial->GERS_theta_star
adapt_initial->GERS_nu	0.1	prefix->initial->GERS_nu
adapt_initial->GERS_theta_c	0.1	prefix->initial->GERS_theta_c
adapt_space->tolerance	1.0	prefix->space->tolerance
adapt_space->p	2	prefix->space->p
adapt_space->max_iteration	30	prefix->space->max_iteration
adapt_space->info	2	prefix->space->info
adapt_space->refine_bisections	DIM	prefix->space->refine_bisections
adapt_space->coarsen_allowed	1	prefix->space->coarsen_allowed
adapt_space->coarse_bisections	DIM	prefix->space->coarse_bisections
adapt_space->strategy	1	prefix->space->strategy
adapt_space->MS_gamma	0.5	prefix->space->MS_gamma
adapt_space->MS_gamma_c	0.1	prefix->space->MS_gamma_c
adapt_space->ES_theta	0.9	prefix->space->ES_theta
adapt_space->ES_theta_c	0.2	prefix->space->ES_theta_c
adapt_space->GERS_theta_star	0.6	prefix->space->GERS_theta_star
adapt_space->GERS_nu	0.1	prefix->space->GERS_nu
adapt_space->GERS_theta_c	0.1	prefix->space->GERS_theta_c
start_time	0.0	prefix->start_time
end_time	1.0	prefix->end_time
timestep	0.01	prefix->timestep
strategy	0	prefix->strategy
max_iteration	0	prefix->max_iteration
tolerance	1.0	prefix->tolerance
rel_initial_error	0.1	prefix->rel_initial_error
rel_space_error	0.4	prefix->rel_space_error
rel_time_error	0.4	prefix->rel_time_error
time_theta_1	1.0	prefix->time_theta_1
time_theta_2	0.3	prefix->time_theta_2
time_delta_1	0.7071	prefix->time_delta_1
time_delta_2	1.4142	prefix->time_delta_2
info	8	prefix->info

Table 2.12: Initialized members of an ADAPT_INSTAT structure, the default values and the key for the initialization by GET_PARAMETER().

Description:

`ellipt_est(uh, adapt, rw_el_est, rw_el_estc, degree, norm, C, A, f, f_flag)`: computes an error estimate of the above type for the H^1 or L^2 norm; the return value is an approximation of the estimate $\|u - u_h\|$ by quadrature.

`uh` is a vector storing the coefficients of the discrete solution; if `uh` is a `nil` pointer, nothing is done, the return value is `0.0`.

`adapt` is a pointer to an `ADAPT_STAT` structure; if it is not `nil`, the entries `err_sum` and `err_max` of `adapt` are set by `ellipt_est()` (compare Section 2.13.1).

`rw_el_est` is a function for writing the local error indicator for a single element (usually to some location inside `leaf_data`, compare Section 2.2.12); if this function is `nil`, only the global estimate is computed, no local indicators are stored. `rw_el_est(el)` returns for each leaf element `el` a pointer to a `REAL` for storing the square of the element indicator, which can directly be used in the adaptive method, compare the `get_el_est()` function pointer in the `ADAPT_STAT` structure (compare Section 2.13.1).

`rw_el_estc` is a function for writing the local coarsening error indicator for a single element (usually to some location inside `leaf_data`, compare Section 2.2.12); if this function is `nil`, no coarsening error indicators are computed and stored; `rw_el_estc(el)` returns for each leaf element `el` a pointer to a `REAL` for storing the square of the element coarsening error indicator.

`degree` is the degree of the quadrature that should be used for the approximation of the norms on the elements and edges/faces; if `degree` is less than zero a quadrature which is exact of degree `2*uh->fe_space->bas_fcts->degree` is used.

`norm` can be either `H1_NORM` or `L2_NORM` (which are defined as symbolic constants in `albert.h`) to indicate that the H^1 or L^2 error estimate has to be calculated.

`C[0]`, `C[1]`, `C[2]` are the constants in front of the element residual, edge/face residual, and coarsening term respectively. If `C` is `nil`, then all constants are set to 1.0.

`A` is the constant matrix of the second order term.

`f` is a pointer to a function for the evaluation of the lower order terms at all quadrature nodes, i.e. $f(x(\lambda), u(\lambda), \nabla u(\lambda))$; if `f` is a `nil` pointer, $f \equiv 0$ is assumed;

`f(el_info, quad, iq, uh_iq, grd_uh_iq)` returns the value of the lower oder terms on element `el_info->el` at the quadrature node `quad->lambda[iq]`, where `uh_iq` is the value and `grd_uh_iq` the gradient (with respect to the world coordinates) of the discrete solution at that quadrature node.

`f_flag` specifies whether the function `f()` actually needs values of `uh_iq` or `grd_uh_iq`. This flag may hold zero, the predefined values `INIT_UH` or `INIT_GRD_UH`, or their composition `INIT_UH|INIT_GRD_UH`; the arguments `uh_iq` and `grd_uh_iq` of `f()` only hold valid information, if the flags `INIT_UH` respectively `INIT_GRD_UH` are set.

The estimate is computed by traversing all leaf elements of `uh->fe_space->mesh`, using the quadrature for the approximation of the residuals and storing the square of the element indicators on the elements (if `rw_el_est` and `rw_el_estc` are not `nil`).

2.14.1. Example (Linear problem). Consider the linear model problem (1.27) with constant coefficients A, b, and c:

$$\begin{aligned} -\nabla \cdot A\nabla u + b \cdot \nabla u + c\,u &= r && \text{in } \Omega, \\ u &= 0 && \text{on } \partial\Omega. \end{aligned}$$

Let `A` be a `REAL_DD` matrix storing A, which is then the eighth argument of `ellipt_est()`. Assume that `const REAL *b(const REAL_D)` is a function returning a pointer to a vector storing b, `REAL c(REAL_D)` returns the value of c and `REAL r(const REAL_D)` returns the value of the right hand side r of (1.27) at some point in world coordinates. The implementation of the function `f` is:

```
static REAL f(const EL_INFO *el_info, const QUAD *quad, int iq, REAL uh_iq,
              const REAL_D grd_uh_iq)
{
  FUNCNAME("f");
  const REAL  *bx, *x;
  extern const REAL b(const REAL_D);
  extern REAL       c(const REAL_D), r(const REAL_D);

  x = coord_to_world(el_info, quad->lambda[iq], nil);
  bx = b(x);

  return(SCP_DOW(bx, grd_uh_iq) + c(x)*uh_iq - r(x));
}
```

As both `uh_iq` and `grd_uh_iq` are used, the estimator parameter `f_flag` must be given as `INIT_UH|INIT_GRD_UH`.

2.14.2 Error estimator for parabolic problems

Similar to the stationary case, the ALBERT library provides an error estimator for the quasi–linear parabolic problem

$$
\begin{aligned}
\partial_t u - \nabla \cdot A\nabla u(x) + f\Big(x, t, u(x), \nabla u(x)\Big) &= 0 \qquad && x \in \Omega, t > 0,\\
u(x,t) &= 0 && x \in \Gamma_D, t > 0,\\
\nu \cdot A\nabla u(x,t) &= 0 && x \in \Gamma_N, t > 0,\\
u(x,0) &= u_0 && x \in \Omega,
\end{aligned}
$$

where $A \in \mathbb{R}^{d\times d}$ is a positive definite matrix and $\partial\Omega = \Gamma_D \cup \Gamma_N$. The estimator is split in several parts, where the initial error

$$\eta_0 = \|u_0 - U_0\|_{L^2(\Omega)}$$

can be approximated by the function `L2_err()`, e.g. (compare Section 2.11).

For the estimation of the spacial discretization error, the coarsening error, and the time discretization error, the ALBERT estimator implements the following (local) indicators

$$
\begin{aligned}
\eta_S^2 = C_0^2\, h_S^4 &\left\| \frac{U_{n+1} - I_{n+1}U_n}{\tau_{n+1}} - \nabla \cdot A\nabla U_{n+1} + f(., t_{n+1}, U_{n+1}, \nabla U_{n+1}) \right\|_{L^2(S)}^2\\
&+ C_1^2 \sum_{\Gamma\subset\partial S\cap\Omega} h_S^3\, \|\, [\![A\nabla U_{n+1}]\!]\, \|_{L^2(\Gamma)}^2 + C_1^2 \sum_{\Gamma\subset\partial S\cap\Gamma_N} h_S^3\, \|\nu \cdot A\nabla U_{n+1}\|_{L^2(\Gamma)}^2,\\
\eta_{S,c}^2 = C_2^2\, &\|U_n - I_{n+1}U_n\|_{L^2(S)}^2\\
\eta_\tau = C_3 &\|U_{n+1} - I_{n+1}U_n\|_{L^2(\Omega)}.
\end{aligned}
$$

The implementation is done by the function

```
REAL heat_est(const DOF_REAL_VEC *, ADAPT_INSTAT *,
        REAL *(*)(EL *), REAL *(*)(EL *), int, REAL[4],
        const DOF_REAL_VEC *, const REAL_DD,
        REAL (*)(const EL_INFO *, const QUAD *, int, REAL, REAL, const REAL_D),
        FLAGS);
```

Description:

`heat_est(uh, adapt, rw_el_est, rw_el_estc, degree, C, uh_old, A, f, f_flag)`: computes an error estimate of the above type, the local and global space discretization estimators are stored in `adapt->adapt_space` and via the `rw_...` pointers; the return value is the time discretization indicator η_τ.

`uh` is a vector storing the coefficients of the discrete solution U_{n+1}; if `uh` is a `nil` pointer, nothing is done, the return value is `0.0`.

`adapt` is a pointer to an `ADAPT_INSTAT` structure; if it is not `nil`, then the entries `adapt_space->err_sum` and `adapt_space->err_max` of `adapt` are set by `heat_est()` (compare Section 2.13.1).

`rw_el_est` is a function for writing the local error indicator η_S^2 for a single element (usually to some location inside `leaf_data`, compare Section 2.2.12); if this function is `nil`, only the global estimate is computed, no local indicators are stored. `rw_el_est(el)` returns for each leaf element `el` a pointer to a `REAL` for storing the square of the element indicator, which can directly be used in the adaptive method, compare the `get_el_est()` function pointer in the `ADAPT_STAT` structure (compare Section 2.13.1).

`rw_el_estc` is a function for writing the local coarsening error indicator $\eta_{S,c}^2$ for a single element (usually to some location inside `leaf_data`, compare Section 2.2.12); if this function is `nil`, no coarsening error indicators are computed and stored; `rw_el_estc(el)` returns for each leaf element `el` a pointer to a `REAL` for storing the square of the element coarsening error indicator.

`degree` is the degree of the quadrature that should be used for the approximation of the norms on the elements and edges/faces; if `degree` is less than zero a quadrature which is exact of degree `2*uh->fe_space->bas_fcts->degree` is used.

`C[0]`, `C[1]`, `C[2]`, `C[3]` are the constants in front of the element residual, edge/face residual, coarsening term, and time residual, respectively. If `C` is `nil`, then all constants are set to 1.0.

`uh_old` is a vector storing the coefficients of the discrete solution U_n from previous time step; if `uh_old` is a `nil` pointer, nothing is done, the return value is `0.0`.

`A` is the constant matrix of the second order term.

`f` is a pointer to a function for the evaluation of the lower order terms at all quadrature nodes, i.e. $f(x(\lambda), t, u(\lambda), \nabla u(\lambda))$; if `f` is a `nil` pointer, $f \equiv 0$ is assumed;

`f(el_info, quad, iq, t, uh_iq, grd_uh_iq)` returns the value of the lower oder terms on element `el_info->el` at the quadrature node `quad->lambda[iq]`, where `uh_iq` is the value and `grd_uh_iq` the gradient (with respect to the world coordinates) of the discrete solution at that quadrature node.

`f_flag` specifies whether the function `f()` actually needs values of `uh_iq` or `grd_uh_iq`. This flag may hold zero, the predefined values `INIT_UH` or `INIT_GRD_UH`, or their composition `INIT_UH|INIT_GRD_UH`; the arguments `uh_iq` and `grd_uh_iq` of `f()` only hold valid information, if the flags `INIT_UH` respectively `INIT_GRD_UH` are set.

The estimate is computed by traversing all leaf elements of `uh->fe_space->mesh`, using the quadrature for the approximation of the residuals and storing the square of the element indicators on the elements (if `rw_el_est` and `rw_el_estc` are not `nil`).

2.15 Solver for linear and nonlinear systems

ALBERT uses a library for solving general linear and nonlinear systems. The solvers use vectors for storing coefficients. The linear solvers need a subroutine for the matrix–vector multiplication, and in the case that a preconditioner is used, a function for preconditioning. The nonlinear solvers need subroutines for assemblage and solution of a linearized system.

In the subsequent sections we describe the basic data structures for the OEM library, an ALBERT interface for solving systems involving a `DOF_MATRIX` and `DOF_REAL[_D]` vectors, and the access of functions for matrix–vector multiplication and preconditioning for a direct use of the OEM solvers. Then we describe the basic data structures for multigrid solvers and for the available solvers of nonlinear equations.

2.15.1 General linear solvers

Very efficient solvers for linear systems are Krylov-space (or Orthogonal Error Methods) solvers.

The OEM library provides such solvers for the solution of general linear systems

$$A\,x = b$$

with $A \in \mathbb{R}^{N\times N}$ and $x, b \in \mathbb{R}^N$. The solvers in that library are working on vectors and do not know the storage of the system matrix, or the matrix used for preconditioning. Matrix–vector multiplication and preconditioning is done by application dependent routines.

The OEM solvers are *not* part of ALBERT. For the access of the basic data structure and prototypes of solvers, the header file

```
#include <oem.h>
```

has to be included in each file using a solver from the library. Most of the implemented solvers are a C-translation from the solvers of FORTRAN OFM-library, by Dörfler [13]. All solvers allow a *left* preconditioning and some also a *right* preconditioning.

The data structure (defined in `oem.h`) for passing information about matrix–vector multiplication, preconditioning and tolerances, etc. to the solvers is

```
typedef struct oem_data OEM_DATA;
struct oem_data
{
  int        (*mat_vec)(void *, int, const REAL *, REAL *);
  void       *mat_vec_data;
  int        (*mat_vec_T)(void *, int, const REAL *, REAL *);
  void       *mat_vec_T_data;
  void       (*left_precon)(void *, int, REAL *);
  void       *left_precon_data;
  void       (*right_precon)(void *, int, REAL *);
  void       *right_precon_data;

  WORKSPACE  *ws;

  REAL       tolerance;
  int        restart;
  int        max_iter;
  int        info;

  REAL       initial_residual;
```

```
  REAL        residual;
};
```

Description:

`mat_vec`: pointer to a function for the matrix–vector multiplication with the system matrix; `mat_vec(mat_vec_data, dim, u, b)` applies A to the input vector `u` and stores the product in `b`; `dim` is the dimension of the linear system, `mat_vec_data` a pointer to user data;

`mat_vec_data`: pointer to user data for the matrix–vector multiplication, first argument to `mat_vec()`;

`mat_vec_T`: pointer to a function for the matrix–vector multiplication with the transposed system matrix;
`mat_vec_T(mat_vec_data, dim, u, b)` applies A^t to the input vector `u` and stores the product in `b`; `dim` is the dimension of the linear system, `mat_vec_T_data` a pointer to user data;

`mat_vec_T_data`: pointer to user data for the matrix–vector multiplication with the transposed matrix, first argument to `mat_vec_T()`;

`left_precon`: pointer to function for left preconditioning; it may be a `nil` pointer; in this case no left preconditioning is done;
`left_precon(left_precon_data, dim, r)` is the implementation of the left preconditioner; `r` is input and output vector of length `dim` and `left_precon_data` a pointer to user data;

`left_precon_data`: pointer to user data for the left preconditioning, first argument to `left_precon()`;

`right_precon`: pointer to function for right preconditioning; it may be a `nil` pointer; in this case no right preconditioning is done;
`right_precon(right_precon_data, dim, r)` is the implementation of the right preconditioner; `r` is input and output vector of length `dim` and `right_precon_data` a pointer to user data;

`right_precon_data`: pointer to user data for the right preconditioning, first argument to `right_precon()`;

`ws`: a pointer to a `WORKSPACE` structure for storing additional vectors used by a solver; if the space is not sufficient, the used solver will enlarge this workspace; if `ws` is `nil`, then the used solver allocates memory, which is freed before exit;

`tolerance`: tolerance for the residual; if the norm of the residual is less or equal `tolerance`, the solver returns the actual iterate as the solution of the system;

`restart`: restart for the linear solver; used only by `oem_gmres()` at the moment;

`max_iter`: maximal number of iterations to be performed although the tolerance may not be reached;

`info`: the level of information produced by the solver; `0` is the lowest level of information (no information is printed) and `10` the highest level;

`initial_residual`: stores the norm of the initial residual on exit;

`residual`: stores the norm of the last residual on exit.

The following linear solvers are currently implemented:

```
int oem_bicgstab(OEM_DATA *, int, const REAL *, REAL *);
int oem_cg(OEM_DATA *, int, const REAL *, REAL *);
int oem_gmres(OEM_DATA *, int, const REAL *, REAL *);
int oem_odir(OEM_DATA *, int, const REAL *, REAL *);
int oem_ores(OEM_DATA *, int, const REAL *, REAL *);
```

Description:

`oem_bicgstab(oem_data, dim, b, x0)`: solves a linear system by a stabilized BiCG method and can be used for symmetric system matrices; `oem_data` stores information about matrix vector multiplication, preconditioning, tolerances, etc. `dim` is the dimension of the linear system, `b` the right hand side vector, and `x0` the initial guess on input and the solution on output; `oem_bicgstab()` needs a workspace for storing 5*`dim` additional `REAL`s; the return value is the number of iterations; `oem_bicgstab()` only uses left preconditioning;

`oem_cg(oem_data, dim, b, x0)`: solves a linear system by the conjugate gradient method and can be used for symmetric positive definite system matrices; `oem_data` stores information about matrix vector multiplication, preconditioning, tolerances, etc. `dim` is the dimension of the linear system, `b` the right hand side vector, and `x0` the initial guess on input and the solution on output; `oem_cg()` needs a workspace for storing 3*`dim` additional `REAL`s; the return value is the number of iterations; `oem_cg()` only uses left preconditioning;

`oem_gmres(oem_data, dim, b, x0)`: solves a linear system by the GMRes method with restart and can be used for regular system matrices; `oem_data` stores information about matrix vector multiplication, preconditioning, tolerances, etc. `dim` is the dimension of the linear system, `b` the right hand side vector, and `x0` the initial guess on input and the solution on output; `oem_data->restart` is the dimension of the Krylov–space for the minimizing procedure; `oem_data->restart` must be bigger than `0` and less or equal `dim`; `oem_gmres()` needs a workspace for storing `(oem_data->restart+2)*dim + oem_data->restart*(oem_data->restart+4)` additional `REAL`s;

`oem_odir(oem_data, dim, b, x0)`: solves a linear system by the method of orthogonal directions and can be used for symmetric, positive system matrices; `oem_data` stores information about matrix vector multiplication, preconditioning, tolerances, etc. `dim` is the dimension of the linear system, `b` the right hand side vector, and `x0` the initial guess on input and the solution on output; `oem_dir()` needs a workspace for storing 5*`dim` additional `REAL`s; the return value is the number of iterations; `oem_odir()` only uses left preconditioning;

`oem_ores(oem_data, dim, b, x0)`: solves a linear system by the method of orthogonal residuals and can be used for symmetric system matrices; `oem_data` stores information about matrix vector multiplication, preconditioning, tolerances, etc. `dim` is the dimension of the linear system, `b` the right hand side vector, and `x0` the initial guess on input and the solution on output; `oem_res()` needs a workspace for storing 7*`dim` additional `REAL`s; the return value is the number of iterations; `oem_ores()` only uses left preconditioning;

2.15.2 Linear solvers for DOF matrices and vectors

OEM solvers

The following functions are an interface to the above described solvers for DOF matrices and vectors. The function `oem_solve_s` is used for scalar valued problems, i.e.

$$A\,x = b$$

with $A \in \mathbb{R}^{N\times N}$ and $x, b \in \mathbb{R}^N$, and oem_solve_d for decoupled vector valued problems of the form

$$\begin{bmatrix} A & 0 & \dots & 0 \\ 0 & A & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & A \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_d \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_d \end{bmatrix}$$

with $A \in \mathbb{R}^{N\times N}$ and $u_i, f_i \in \mathbb{R}^N$, $i = 1, \dots, d$, where $d =$ DIM_OF_WORLD. The vectors $(u_1, \dots, u_d)$ and $(f_1, \dots, f_d)$ are stored in DOF_REAL_D_VECs, whereas the matrix is stored as a single DOF_MATRIX.

For the solver identification we use the following type

```
typedef enum {NoSolver, BiCGStab, CG, GMRes, ODir, ORes} OEM_SOLVER;
```

This type will be changed at the time when additional solvers will be available.

The following functions can directly be used for solving a linear system involving a DOF_MATRIX and DOF_REAL[_D]_VECs.

```
int oem_solve_s(const DOF_MATRIX *, const DOF_REAL_VEC *, const DOF_SCHAR_VEC *,
                DOF_REAL_VEC *, OEM_SOLVER, REAL, int, int, int, int);
int oem_solve_d(const DOF_MATRIX *,const DOF_REAL_D_VEC *,const DOF_SCHAR_VEC *,
                DOF_REAL_D_VEC *, OEM_SOLVER, REAL, int, int, int, int);
```

Description:

oem_solve_s[d](A, f, bound, u, solver, tol, icon, restart, max_iter, info):
solves a linear system by an OEM solver; the return value is the number of performed iterations;

A: pointer to a DOF_MATRIX storing the system matrix;

f: pointer to a DOF_REAL[_D] vector storing the right hand side of the linear system;

bound: pointer to a DOF_SCHAR vector storing information about the boundary type of DOFs; if bound is nil, then boundary information is build in the system matrix; otherwise, all DOFs with bound->vec[dof] >= DIRICHLET are treated as Dirichlet boundary values;

u: pointer to a DOF_REAL[_D] storing the initial guess on input and the calculated solution on output;

the values for u and f have to be the same at all Dirichlet DOFs (compare Section 2.12.5);

solver: use solver solver from the OEM library for solving the linear system; may be one of BiCGStab, CG, GMRes, ODir, or ORes; the meaning of solver is more or less self explaining;

tol: tolerance for the residual; if the norm of the residual is less or equal tol, oem_solve_s[d]() returns the actual iterate as the solution of the system;

icon: parameter for preconditioning:
- 0: no preconditioning
- 1: diagonal preconditioning
- 2: hierarchical basis preconditioning (only for linear f.e. at the moment)

max_iter: maximal number of iterations to be performed by the linear solver although the tolerance may not be reached;

info: is the level of information of the linear solver; 0 is the lowest level of information (no information is printed) and 10 the highest level.

The function initializes a data structure oem_data with information about the matrix–vector multiplication, preconditioning, tolerances, etc. and the additional memory needed by the linear solver is allocated automatically. The linear system is then solved by the chosen OEM solver.

SOR solvers

The SOR and SSOR methods are implemented directly for a linear system involving a `DOF_MATRIX` and `DOF_REAL_[D_]VEC`s.

```
int sor_s(DOF_MATRIX *, const DOF_REAL_VEC *, const DOF_SCHAR_VEC *,
          DOF_REAL_VEC *, REAL, REAL, int, int);
int sor_d(DOF_MATRIX *, const DOF_REAL_D_VEC *, const DOF_SCHAR_VEC *,
          DOF_REAL_D_VEC *, REAL, REAL, int, int);
int ssor_s(DOF_MATRIX *, const DOF_REAL_VEC *, const DOF_SCHAR_VEC *,
          DOF_REAL_VEC *, REAL, REAL, int, int);
int ssor_d(DOF_MATRIX *, const DOF_REAL_D_VEC *, const DOF_SCHAR_VEC *,
          DOF_REAL_D_VEC *, REAL, REAL, int, int);
```

`[s]sor_s[d](matrix, f, bound, u, omega, tol, max_iter, info)`: solves a linear system by the [Symmetric] Successive OverRelaxation method; the return value is the number of used iterations to reach the prescribed tolerance;

`matrix`: pointer to a DOF matrix storing the system matrix;

`f`: pointer to a DOF vector storing the right hand side of the system;

`bound`: optional pointer to a DOF vector giving Dirichlet boundary information;

`u`: pointer to a DOF vector storing the initial guess on input and the calculated solution on output;

`omega`: the relaxation parameter and must be in the interval $(0, 2]$; if it is not in this interval `omega=1.0` is used;

`tol`: tolerance for the maximum norm of the correction; if this norm is less or equal `tol`, `sor_s[d]()` returns the actual iterate as the solution of the system;

`max_iter`: maximal number of iterations to be performed by `sor_s[d]()` although the tolerance may not be reached;

`info`: level of information of `sor_s[d]()`; `0` is the lowest level of information (no information is printed) and `6` the highest level.

2.15.3 Access of functions for matrix–vector multiplication

The general `oem_...()` solvers all need pointers to matrix–vector multiplication routines which do not work with `DOF_REAL_VEC`s and a `DOF_MATRIX` but directly on `REAL` vectors. For the application to a scalar or vector–valued linear system described by a `DOF_MATRIX` (and an optional `DOF_SCHAR_VEC` holding boundary information), the following routines are provided.

```
void *init_mat_vec_s(const DOF_MATRIX *, const DOF_SCHAR_VEC *);
int mat_vec_s(void *, int, const REAL *, REAL *);

void *init_mat_vec_d(const DOF_MATRIX *, const DOF_SCHAR_VEC *);
int mat_vec_d(void *, int, const REAL *, REAL *);
```

After a call to the initialization routine `init_mat_vec_s()`, the routine `mat_vec_s()` implements the (scalar) matrix–vector multiplication. Thus, `mat_vec_s()` and the pointer returned by the `init_...()` routine can be passed as `mat_vec` and `mat_vec_data` entries in the `OEM_DATA` structure to an `oem_...()` solver.

2.15.4 Access of functions for preconditioning

While a preconditioner can be selected in `oem_solve_s[d]()` just by an integer parameter, a direct access to the functions is needed for a more general application, which directly calls one of the `oem_...()` routines.

A preconditioner may need some initialization phase, which depends on the matrix of the linear system, but is independent of the actual application of the preconditioner to a vector. Thus, a preconditioner is described by three functions for initialization, application, and a final exit routine which may free some memory which was allocated during initialization, e.g. All three functions are collected in the structure

```
typedef struct precon PRECON;
struct precon
{
  void    *(*init_precon)(const DOF_MATRIX *, const DOF_SCHAR_VEC *, int);
  void    (*precon)(void *, int, REAL *);
  void    (*exit_precon)(void *);
};
```

The pointer returned by `init_precon()` must be given as first parameter to `precon()` and `exit_precon()`.

Currently, a diagonal preconditioner and a hierarchical basis preconditioner are implemented. Access to the corresponding routines for scalar (`..._s()`) and vector valued problems (`..._d()`) are given via the following functions, which return a pointer to such a `PRECON` structure:

```
const PRECON *get_diag_precon_s(void);
const PRECON *get_diag_precon_d(void);
const PRECON *get_HB_precon_s(void);
const PRECON *get_HB_precon_d(void);
```

As already indicated above, the hierarchical basis preconditioner is currently implemented only for piecewise linear Lagrange finite elements.

2.15.5 Multigrid solvers

A abstract framework for multigrid solvers is available. Based upon information given in a data structure `MULTI_GRID_INFO`, the subroutine `MG()` iterates until a prescribed tolerance is met or a prescribed number of multigrid cycles is performed.

```
typedef struct multi_grid_info MULTI_GRID_INFO;
struct multi_grid_info
{
  REAL            tolerance;                   /* tol. for resid         */
  REAL            exact_tolerance;             /* tol. for exact_solver  */

  int             cycle;                       /* 1=V-cycle, 2=W-cycle   */
  int             n_pre_smooth, n_in_smooth;   /* no of smoothing loops  */
  int             n_post_smooth;               /* no of smoothing loops  */
  int             mg_levels;                   /* current no. of levels  */
  int             exact_level;                 /* level for exact_solver */
  int             max_iter;                    /* max. no of MG iter's   */
  int             info;

  int             (*init_multi_grid)(MULTI_GRID_INFO *mg_info);
```

```
  void              (*pre_smooth)(MULTI_GRID_INFO *mg_info, int level, int n);
  void              (*in_smooth)(MULTI_GRID_INFO *mg_info, int level, int n);
  void              (*post_smooth)(MULTI_GRID_INFO *mg_info, int level, int n);
  void              (*restrict)(MULTI_GRID_INFO *mg_info, int level);
  void              (*prolongate)(MULTI_GRID_INFO *mg_info, int level);
  void              (*exact_solver)(MULTI_GRID_INFO *mg_info, int level);
  REAL              (*resid)(MULTI_GRID_INFO *mg_info, int level);
  void              (*exit_multi_grid)(MULTI_GRID_INFO *mg_info);

  void              *data;                        /* application dep. data */
};
```

The entries yield following information:

`tolerance`: tolerance for norm of residual;

`exact_tolerance`: tolerance for "exact solver" on coarsest level;

`cycle`: selection of multigrid cycle type: 1 =V-cycle, 2 =W-cycle, ...;

`n_pre_smooth`: number of smoothing steps on each level before (first) coarse level correction;

`n_in_smooth`: number of smoothing steps on each level between coarse level corrections (for cycle≥ 2);

`n_post_smooth`: number of smoothing steps on each level after (last) coarse level correction;

`mg_levels`: number of levels;

`exact_level`: selection of grid level where the "exact" solver is used (and no further coarse grid correction), usually `exact_level=0`;

`max_iter`: maximal number of multigrid iterations;

`info`: selection of information message output;

`init_multi_grid(mg_info)`: initialization routine, returns `true` if an error occurs, may be `nil`;

`pre_smooth(mg_info, level, n)`: perform `n` smoothing iterations (before coarse grid corrections) on grid `level`;

`in_smooth(mg_info, level, n)`: perform `n` smoothing iterations (between coarse grid corrections) on grid `level`;

`post_smooth(mg_info, level, n)`: perform `n` smoothing iterations (after coarse grid corrections) on grid `level`;

`restrict(mg_info, level)`: calculate and restrict residual from grid `level` to next coarser grid (`level-1`);

`prolongate(mg_info, level)`: prolongate and add coarse grid (`level-1`) correction to fine grid solution on grid `level`;

`exact_solver(mg_info, level)`: "exact" solution of the problem on grid `level` (with tolerance `mg_info->exact_tolerance`);

`resid(mg_info, level)`: returns norm of residual on grid `level`;

`exit_multi_grid(mg_info)`: cleanup routine, may be `nil`;

`data`: pointer to application dependent data, holding information on or about different grid levels, e.g.

The abstract multigrid solution is done by the subroutine

```
int MG(MULTI_GRID_INFO *mg_info)
```

Main parts of this routine are:

```
{
  int iter;
  REAL resid;

  if (mg_info->init_multi_grid)
    if (mg_info->init_multi_grid(mg_info))
      return(-1);

  resid = mg_info->resid(mg_info, mg_info->mg_levels-1);
  if (resid <= mg_info->tolerance)
    return(0);

  for (iter = 0; iter < mg_info->max_iter; iter++)
  {
    recursive_MG_iteration(mg_info, mg_info->mg_levels-1);
    resid = mg_info->resid(mg_info, mg_info->mg_levels-1);
    if (resid <= mg_info->tolerance)
      break;
  }
  if (mg_info->exit_multi_grid)
    mg_info->exit_multi_grid(mg_info);

  return(iter+1);
}
```

The subroutine `recursive_MG_iteration()` performs smoothing, restriction of the residual and prolongation of the coarse grid correction:

```
static void recursive_MG_iteration(MULTI_GRID_INFO *mg_info, int level)
{
  int  cycle;

  if (level <= mg_info->exact_level) {
    mg_info->exact_solver(mg_info, level);
  }
  else {
    if (mg_info->pre_smooth)
      mg_info->pre_smooth(mg_info, level, mg_info->n_pre_smooth);

    for (cycle = 0; cycle < mg_info->cycle; cycle++) {
      if ((cycle > 0) && mg_info->in_smooth)
        mg_info->in_smooth(mg_info, level, mg_info->n_in_smooth);

      mg_info->restrict(mg_info, level);
      recursive_MG_iteration(mg_info, level-1);
      mg_info->prolongate(mg_info, level);
    }

    if (mg_info->post_smooth)
      mg_info->post_smooth(mg_info, level, mg_info->n_post_smooth);
  }
}
```

member	default	key
mg_info->cycle	1	prefix->cycle
mg_info->n_pre_smooth	1	prefix->n_pre_smooth
mg_info->n_in_smooth	1	prefix->n_in_smooth
mg_info->n_post_smooth	1	prefix->n_post_smooth
mg_info->exact_level	0	prefix->exact_level
mg_info->info		prefix->info
mg_s_info->smoother	1	prefix->smoother
mg_s_info->smooth_omega	1.0	prefix->smooth_omega
mg_s_info->exact_solver	1	prefix->exact_solver
mg_s_info->exact_omega	1.0	prefix->exact_omega

Table 2.13: Parameters read by `mg_s()` and `mg_s_init()`

For multigrid solution of a scalar linear system

$$Au = f$$

given by a `DOF_MATRIX A` and a `DOF_REAL_VEC f`, the following subroutine is available:

```
int mg_s(DOF_MATRIX *matrix, DOF_REAL_VEC *u, const DOF_REAL_VEC *f,
         const DOF_SCHAR_VEC *bound,
         REAL tol, int max_iter, int info, char *prefix);
```

Parameters include:

`matrix`: the system matrix;

`u`: the solution vector;

`f`: the right hand side vector;

`bound`: boundary type vector for correct handling of Dirichlet DOFs;

`tol`: tolerance for multigrid solver;

`max_iter`: maximal number of multigrid iterations;

`info`: selection of information messages from solver;

`prefix`: parameter key prefix for initialization of additional data via `GET_PARAMETER`, see Table 2.13, may be `nil`.

An SOR smoother (`mg_s_info->smoother=1`) and an SSOR smoother (`smoother=2`) are available. An under- or over relaxation parameter can be specified by `mg_s_info->smooth_omega`. These SOR/SSOR smoothers are used for `exact_solver`, too.

For applications, where several systems with the same matrix have to be solved, computing time can be saved by doing all initializations like setup of grid levels and restriction of matrices only once. For such cases, three subroutines are available:

```
MG_S_INFO *mg_s_init(DOF_MATRIX *matrix, const DOF_SCHAR_VEC *bound,
                     int info, char *prefix);
int mg_s_solve(MG_S_INFO *mg_s_info,
               DOF_REAL_VEC *u, const DOF_REAL_VEC *f, REAL tol, int max_iter);
void mg_s_exit(MG_S_INFO *mg_s_info);
```

Parameters are the same as described above; the structure `MG_S_INFO` contains matrices and vectors for linear problems on all different grid levels. After initialization of the multigrid solver via `mg_s_init()`, the routine `mg_s_solve()` can be called several times (with different right hand sides, e.g.). Finally, `mg_s_exit()` frees all memory that was previously allocated by `mg_s_init()`.

2.15.1. Remark. The multigrid solver is currently available only for Lagrange finite elements of first order (`lagrange1`). An implementation for higher order elements is in progress.

2.15.6 Nonlinear solvers

For the solution of a nonlinear equation

$$u \in \mathbb{R}^N : \qquad F(u) = 0 \qquad \text{in } \mathbb{R}^N \tag{2.1}$$

several Newton methods are provided. For testing the convergence a (problem dependent) norm of either the correction d_k in the kth step, i.e.

$$\|d_k\| = \|u_{k+1} - u_k\|,$$

or the residual, i.e.

$$\|F(u_{k+1})\|,$$

is used.

These solvers are *not* part of ALBERT and need for the access of the basic data structure and prototypes of solvers the header file

```
#include <nls.h>
```

The data structure (defined in `nls.h`) for passing information about assembling and solving a linearized equation, tolerances, etc. to the solvers is

```
typedef struct nls_data NLS_DATA;
struct nls_data
{
  void       (*update)(void *, int, const REAL *, int, REAL *);
  void       *update_data;
  int        (*solve)(void *, int, const REAL *, REAL *);
  void       *solve_data;
  REAL       (*norm)(void *, int, const REAL *);
  void       *norm_data;

  WORKSPACE  *ws;

  REAL       tolerance;
  int        restart;
  int        max_iter;
  int        info;

  REAL       initial_residual;
  REAL       residual;
};
```

Description:

`update`: subroutine for computing a linearized system;

`update(update_data, dim, uk, update_matrix, F)` computes a linearization of the system matrix, if `update_matrix` is not zero, and the right hand side `F`, if `F` is not `nil`, around the actual iterate `uk`; `dim` is the dimension of the nonlinear system, and `update_data` a pointer to user data;

`update_data`: pointer to user data for the update of a linearized equation, first argument to `update()`;

`solve`: function for solving a linearized system for the new correction; the return value is the number of iterations used by an iterative solver or zero; this number is printed, if information about the solution process should be produced;

`solve(solve_data, dim, F, d)` solves the linearized equation of dimension `dim` with right hand side `F` for a correction `d` of the actual iterate; `d` is initialized with zeros and `update_data` is a pointer to user data;

`solve_data`: pointer to user data for solution of the linearized equation, first argument to `solve()`;

the nonlinear solver does not know how the sytem matrix is stored; such information can be passed from `update()` to `solve()` by using pointers to the same `DOF` matrix in both `update_data` and `solve_data`, e.g.

`norm`: function for computing a problem dependent norm $\|.\|$; if `norm` is `nil`, the Euclidian norm is used;

`norm(norm_data, dim, x)` returns the norm of the vector `x`; `dim` is the dimension of the nonlinear system, and `norm_data` pointer to user data;

`norm_data`: pointer to user data for the calculation of the problem dependent norm, first argument to `norm()`;

`ws`: a pointer to a `WORKSPACE` structure for storing additional vectors used by a solver; if the space is not sufficient, the used solver will enlarge this workspace; if `ws` is `nil`, then the used solver allocates memory, which is freed before exit;

`tolerance`: tolerance for the nonlinear solver; if the norm of the correction/residual is less or equal `tolerance`, the solver returns the actual iterate as the solution of the nonlinear system;

`restart`: restart for the nonlinear solver;

`max_iter`: is a maximal number of iterations to be performed, even if the tolerance may not be reached;

`info`: the level of information produced by the solver; `0` is the lowest level of information (no information is printed) and `4` the highest level;

`initial_residual`: stores the norm of the initial correction/residual on exit;

`residual`: stores the norm of the last correction/residual on exit.

The following Newton methods for solving (2.1) are currently implemented:

```
int nls_newton(NLS_DATA *, int, REAL *);
int nls_newton_ds(NLS_DATA *, int, REAL *);
int nls_newton_fs(NLS_DATA *, int, REAL *);
```

Description:

`nls_newton(nls_data, dim, u0)`: solves a nonlinear system by the classical Newton method; the return value is the number of iterations;

`nls_data` stores information about functions for the assemblage and solution of $DF(u_k)$, $F(u_k)$, calculation of a norm, tolerances, etc. `dim` is the dimension of the nonlinear system, and `u0` the initial guess on input and the solution on output; `nls_newton()` stops if the norm of the **correction** is less or equal `nls_data->tolerance`; it needs a workspace for storing 2*`dim` additional `REAL`s;

`nls_newton_ds(nls_data, dim, u0)`: solves a nonlinear system by a Newton method with step size control; the return value is the number of iterations;

`nls_data` stores information about functions for the assembling and solving of $DF(u_k)$, $F(u_k)$, calculation of a norm, tolerances, etc. `dim` is the dimension of the nonlinear system, and `u0` the initial guess on input and the solution on output; `nls_newton()` stops if the norm of the **correction** is less or equal `nls_data->tolerance`; in each iteration at most `nls_data->restart` steps for controlling the step size τ are performed; the aim is to choose τ such that

$$\|DF(u_k)^{-1}F(u_k + \tau d_k)\| \le (1 - \tfrac{1}{2}\tau)\|d_k\|$$

holds, where $\|.\|$ is the problem dependent norm, if `nls_data->norm` is not `nil`, otherwise the Euclidian norm; each step needs the update of F, the solution of one linearized problem (the system matrix for the linearized system does not change during step size control) and the calculation of a norm;

`nls_newton()` needs a workspace for storing 4*`dim` additional `REAL`s;

`nls_newton_fs(nls_data, dim, u0)`: solves a nonlinear system by a Newton method with step size control; the return value is the number of iterations;

`nls_data` stores information about functions for the assembling and solving of $DF(u_k)$, $F(u_k)$, calculation of a norm, tolerances, etc. `dim` is the dimension of the nonlinear system, and `u0` the initial guess on input and the solution on output; `nls_newton()` stops if the norm of the **residual** is less or equal `nls_data->tolerance`; in each iteration at most `nls_data->restart` steps for controlling the step size τ are performed; the aim is to choose τ such that

$$\|F(u_k + \tau d_k)\| \le (1 - \tfrac{1}{2}\tau)\|F(u_k)\|$$

holds, where $\|.\|$ is the problem dependent norm, if `nls_data->norm` is not `nil`, otherwise the Euclidian norm; the step size control is not expensive, since in each step only an update of F and the calculation of $\|F\|$ are involved;

`nls_newton()` needs a workspace for storing 3*`dim` additional `REAL`s; the return value is the number of iterations;

2.16 Graphics output

ALBERT provides two dimensional graphic subroutines built on the X-Windows and GL/OpenGL interfaces, and two and three dimensional graphics via the gltools [21] and GRAPE [34].

2.16.1 Two dimensional graphics subroutines

A set of subroutines for opening (and closing) of graphic output windows, drawing of the triangulation, drawing isolines of scalar functions, and arrow based display of vector valued functions are provided.

The routines use the following type definitions for window identification and color specification in [red, green, blue] coordinates, with $0 \leq$red, green, blue≤ 1.

```
typedef void * GRAPH_WINDOW;
typedef float  GRAPH_RGBCOLOR[3];
```

The following "standard" colors are provided by ALBERT:

```
extern const GRAPH_RGBCOLOR rgb_black;
extern const GRAPH_RGBCOLOR rgb_white;
extern const GRAPH_RGBCOLOR rgb_red;
extern const GRAPH_RGBCOLOR rgb_green;
extern const GRAPH_RGBCOLOR rgb_blue;
extern const GRAPH_RGBCOLOR rgb_yellow;
extern const GRAPH_RGBCOLOR rgb_magenta;
extern const GRAPH_RGBCOLOR rgb_cyan;
extern const GRAPH_RGBCOLOR rgb_grey50;
```

Opening graphic output windows

Graphic output windows are created and destroyed with

```
GRAPH_WINDOW graph_open_window(char *title, char *geometry,
                               REAL *world, MESH *mesh);
void         graph_close_window(GRAPH_WINDOW win);
```

Parameters are:

`title`: optional window title, if `nil`, a default title is used;

`geometry`: optional window geometry in X11 format "WxH" or "WxH+X+Y", if `nil`, a default geometry is used;

`world`: optional array of *world coordinates* (xmin, xmax, ymin, ymax) to specify which part of a triangulation will be displayed in this window, if `world` is `nil`, either `mesh` is used or the square $[0,1] \times [0,1]$ will be displayed;

`mesh`: optional triangulation, is used (if `world` is `nil`) to select a range of world coordinates which will be displayed in the window; if both `world` and `mesh` are `nil`, the unit square $[0,1] \times [0,1]$ will be displayed.

`graph_open_window()` returns a `GRAPH_WINDOW`, and `graph_close_window()` needs a previously opened `GRAPH_WINDOW` as parameter. The valid range of `GRAPH_WINDOW` values depends on the underlying graphics library which is used to implement the routines. Because of that, the value `nil` is not necessarily an invalid `GRAPH_WINDOW` value which could be used to check if a window was already opened. A symbolic constant

```
const GRAPH_WINDOW  NO_WINDOW;
```

is defined as an invalid `GRAPH_WINDOW` value.

Clearing of a graphics output window `win` is done by

```
void  graph_clear_window(GRAPH_WINDOW win, GRAPH_RGBCOLOR c);
```

The optional parameter `c` selects a color which is used to fill the window. If `c==nil` then a default color (black) is used.

Drawing triangulations

Triangulations are displayed in a graphics output window by the routine

```
void  graph_mesh(GRAPH_WINDOW win, MESH *mesh, GRAPH_RGBCOLOR c, FLAGS flag);
```

Parameters are:

`win`: graphics window;

`mesh`: triangulation;

`c`: optional color used for drawing lines, if `nil`, a default color (white) will be used;

`flag`: selection of a special or additional display, may be `0` or the bitwise OR of some of the following flags.

Flags used by `graph_mesh()`:

`GRAPH_MESH_BOUNDARY`: if set, only boundary edges are drawn, otherwise all edges of the triangulation will be drawn. If not otherwise specified by `c`, the color of Dirichlet boundary edges is blue, while Neumann edges are drawn in red;

`GRAPH_MESH_ELEMENT_MARK`: if set, then triangles which are marked for refinement are filled red, and triangles which are marked for coarsening are filled blue, unmarked triangles are filled white;

`GRAPH_MESH_VERTEX_DOF`: if set, the *first DOF* at each vertex will be written near the vertex;

`GRAPH_MESH_ELEMENT_INDEX`: if set, element indices will be written (only available if `EL_INDEX==1`).

Printing of indices is only possible at the moment if the library was compiled with the flag `USE_OPENGL=0`.

Isoline display of functions

Isolines of a function which is given by a `DOF_REAL_VEC` are drawn by the routines

```
void  graph_levels(GRAPH_WINDOW win, DOF_REAL_VEC *v,
                   int n, REAL *levels, GRAPH_RGBCOLOR *c, int refine);
void  graph_level(GRAPH_WINDOW win, DOF_REAL_VEC *v, REAL level,
                   GRAPH_RGBCOLOR c, int refine);
```

The routine `graph_levels()` can be used to draw multiple isolines. Parameters are

`win`: graphics window;

`v`: `DOF_REAL_VEC` structure containing data of the function;

`n`: number of isolines to be drawn;

`levels`: optional `REAL` vector of size `n` which contains the different values where isolines will be drawn, if `nil`, then `n` equally spaced isolines are drawn between the minimal and maximal value of the function;

`c`: optional vector of color values for the `n` isolines, if `nil`, then default color values will be used,

refine: optional refinement level selector. Isolines are always computed from the piecewise linear interpolation on a triangle. If `refine`≥ 0, each triangle will be recursively refined into $4^{\texttt{refine}}$ smaller triangles, and the function will be evaluated at all vertices of these small triangles. If `refine`< 0, then the default value `v->admin->bas_fcts->degree-1` will be used.

The routine `graph_level()` draws a single selected isoline at value `level` with color `c` and refinement `refine`; again, if `c=nil`, then a default color (white) will be used, and if `refine`< 0 then the default refinement will be done.

Color display of functions

Function values may be visualized using colored triangles by the routine

```
void  graph_values(GRAPH_WINDOW win, DOF_REAL_VEC *v, int refine);
```

The color values are selected by default; color is distributed between minimum and maximum of the v. If such display is not available, this routine will call the `graph_levels()` routine.

Vector valued functions

Similar to the display routines for scalar functions given above, equivalent routines are available for vector valued functions, which use the modulus $|v(x)|$ of the function as scalar value. Available routines are:

```
void  graph_levels_d(GRAPH_WINDOW win, DOF_REAL_D_VEC *v,
                       int n, REAL *levels, GRAPH_RGBCOLOR *c, int refine);
void  graph_level_d(GRAPH_WINDOW win, DOF_REAL_D_VEC *v,
                      REAL level, GRAPH_RGBCOLOR c, int refine);
void  graph_values_d(GRAPH_WINDOW win, DOF_REAL_D_VEC *v, int refine);
```

(Not yet implemented.)

Additionally, vector valued functions may be visualized by using arrows. Such display will be generated by the routine

```
void  graph_vec(GRAPH_WINDOW win, DOF_REAL_D_VEC *v, REAL scale,
                  GRAPH_RGBCOLOR c, int refine);
```

(Not yet implemented; parameters may change.)

Displaying piecewise constant values

Piecewise constant values, like local error estimates calculated by an error estimator, can be visualized by the routine

```
void graph_el_est(GRAPH_WINDOW win, MESH *mesh, REAL (*get_el_est)(EL *el));
```

Parameters are

win: graphics window;

mesh: triangulation;

get_el_est: pointer to a function which returns the value on a single element; this function defines the piecewise constant function which should be displayed.

2.16.2 gltools interface

The following interface for using the interactive gltools graphics is implemented. For obtaining the gltools package please see

`http://www.wias-berlin.de/~gltools`

This interface can be used for 2d and 3d triangulations, but only when `DIM` equals `DIM_OF_WORLD`. For window identification we use the data type

```
typedef void*    GLTOOLS_WINDOW;
```

The interface provides the following functions:

```
GLTOOLS_WINDOW open_gltools_window(const char *, const char *, const REAL *,
                                   MESH *, int);
void close_gltools_window(GLTOOLS_WINDOW);
void gltools_mesh(GLTOOLS_WINDOW, MESH *, int);
void gltools_drv(GLTOOLS_WINDOW, const DOF_REAL_VEC *, REAL, REAL);
void gltools_drv_d(GLTOOLS_WINDOW, const DOF_REAL_D_VEC *, REAL, REAL);
void gltools_est(GLTOOLS_WINDOW, MESH *, REAL (*)(EL *), REAL, REAL);
```

Description:

`open_gltools_window(title, geometry, world, mesh, dialog)`: the function returns a `GLTOOLS_WINDOW` which is opened for display; if the window could not be opened, the return value is `nil`; `title` is an optional string holding a title for the window; if `title` is `nil`, a default is used; `geometry` is an optional string holding the window geometry in X11 format ("WxH" or "WxH+X+Y"), if `nil`, a default geometry is used; the optional argument `world` is a pointer to an array of *world coordinates* (xmin, xmax, ymin, ymax) for 2d and (xmin, xmax, ymin, ymax, zmin, zmax) for 3d can be used to specify which part of the mesh will be displayed in the window; if `world` is `nil`, either `mesh` or the default domain $[0,1]^{\mathtt{DIM}}$ is used; `mesh` is an optional pointer to a mesh to select a range of world coordinates which will be displayed in the window; if both `world` and `mesh` are `nil`, the default domain $[0,1]^{\mathtt{DIM}}$ is used; display is not done or is done in an interactive mode depending on whether `dialog` equals `0` or not; in interactive mode type `'h'` to get a list of all key bindings;

`close_gltools_window(win)`: closes the window `win` which has been previously opened by `open_gltools_window()`;

`gltools_mesh(win, mesh, mark)`: displays the elements of `mesh` in the graphic window `win`; if `mark` is not zero the piecewise constant function sign(`el->mark`) is shown;

`gltools_drv(win, u, min, max)`: displays the `DOF_REAL_VEC u` in the graphic window `win`; only available for Lagrange elements up to order four; if the Lagrange elements are of higher order, it is possible to display the vector on a refined grid; the key `'P'` toggles between refined and not refined mode (not for cubics in 3d at the moment); `min` and `max` define the range of the discrete function for display; if `min` $\geq$ `max` this range is adjusted automatically;

`gltools_drv_d(win, ud, min, max)`: displays the modulus of the `DOF_REAL_D_VEC ud` in the graphic window `win`; only available for Lagrange elements up to order four; the modulus of `ud` is stored in a `DOF_REAL_VEC` which is displayed by `gltools_drv()`;

`gltools_est(win, mesh, get_el_est, min, max)`: displays the estimated error on `mesh` as a piecewise constant function in the graphic window `win`; the local indicators are accessed by `get_el_est()` on each element; `min` and `max` define the range for display; if `min` $\geq$ `max` this range is adjusted automatically;

gltools_est() can also be used to display any piecewise constant function on the mesh, where local values are accessed by get_el_est().

2.16.3 GRAPE interface

Visualization using the GRAPE library is usually done during postprocessing. Data of the actual geometry and finite element functions is written to file by write_mesh[_xdr]() and write_dof_real[_d]_vec[_xdr]() and then read by some programs, using GRAPE mesh interface for the visualization. The use of the GRAPE H–mesh is work in progress and the description of these programs will be done in the near future.

For obtaining the GRAPE library please see

http://www.iam.uni-bonn.de/sfb256/grape/

Chapter 3

Implementation of model problems

In this chapter we describe the implementation of two stationary model problems (the linear Poisson equation and a nonlinear Poisson equation) and of one instationary model problem (the heat equation). Removing all LaTeX descriptions of functions and variables results in the source code for the adaptive solvers.

3.1 Poisson equation

In this section we describe a model implementation for the Poisson equation

$$\begin{aligned} -\Delta u &= f \qquad \text{in } \Omega \subset \mathbb{R}^d, \\ u &= g \qquad \text{on } \partial\Omega. \end{aligned}$$

The implementation is split into nine major steps which are now described in detail.

3.1.1 Graphical output of the discrete solution and error estimate

All three programs use the following graphic routine

```
graphics(MESH *mesh, DOF_REAL_VEC *u_h, REAL (*get_est)(EL *el))
```

for displaying the discrete solution, the error estimate (, and mesh). The function is implemented in a separate file. By redefining the symbolic constant

```
#define USE_GLTOOLS true
```

to `false` one can easily switch from the gltools graphics (compare Section 2.16.2) to the two dimensional graphics routines (compare Section 2.16.1). The 2d graphics routines are not available in 3d!

The routine uses the parameter `graphic windows`, which should be initialized by the parameter tools, for deciding, which windows should be opened for display. The value of this parameter are two integers (gltools) or three integers (2d graphics) holding the window sizes for displaying the discrete solution, error estimate, and the underlying mesh (only for 2d graphics, the mesh can be displayed using gltools graphics, by toggling to wire frame mode (key `'w'`) in one of the other graphic windows). If the window size is 0 for one of these windows, this window will not be opened for display. All windows are opened on the first call of `graphics()` and values are displayed on each call.

The DOF vector `uh` (the second argument) is displayed in the first window, if this window was opened and `uh` is not `nil`. The error indicators are displayed in the second window, if this window was opened and if the third argument `get_est()` is not `nil`. The function `get_est(el)` will return for each element `el` of the grid the local error indicator.

If the 2d graphics routines are used, then the grid is displayed in the third window, if this window was opened and `mesh` is not `nil`.

```
#include <albert.h>

/****************************************************************************/
/* graphical output of mesh, discrete solution, and estimate               */
/****************************************************************************/
#define USE_GLTOOLS true

#if USE_GLTOOLS
void graphics(MESH *mesh, DOF_REAL_VEC *u_h, REAL (*get_est)(EL *el))
{
  FUNCNAME("graphics");
  static int first = true;
  static GLTOOLS_WINDOW   win_est = nil, win_val = nil;

  if (first)
  {
    int size[2] = {0, 0};
    char   geom[128] = "500x500+0+0";
    GET_PARAMETER(1, "graphic windows", "%d %d", size, size+1);

    if (size[0] > 0)
    {
      sprintf(geom, "%dx%d+%d+%d", size[0], size[0], 0, 0);
      win_val = open_gltools_window("ALBERT values", geom, nil, mesh, true);
    }

    if (size[1] > 0)
    {
      sprintf(geom, "%dx%d+%d+%d", size[1], size[1], size[0], 0);
      win_est = open_gltools_window("ALBERT estimate", geom, nil, mesh, true);
    }
    first = false;
  }

  if (u_h && win_val)
    gltools_drv(win_val, u_h, 0.0, 0.0);
  if (get_est && win_est)
    gltools_est(win_est, mesh, get_est, 0.0, -1.0);
  if (mesh && win_val && !u_h  && !get_est)
    gltools_mesh(win_val, mesh, 1);  /*  graph mesh with element marker   */

  if (!win_val && !win_est)  WAIT;

  return;
}
```

```
#else
void graphics(MESH *mesh, DOF_REAL_VEC *u_h, REAL (*get_est)(EL *el))
{
  FUNCNAME("graphics");
#if DIM == 2
  static int first = true;
  static GRAPH_WINDOW win_est = nil, win_val = nil, win_mesh = nil;

  if (first)
  {
    int size[3] = {0, 0, 0};
    char   geom[128] = "500x500+0+0";
    GET_PARAMETER(1, "graphic windows", "%d %d %d", size, size+1, size+2);

    if (size[0] > 0)
    {
      sprintf(geom, "%dx%d+%d+%d", size[0], size[0], 0, 0);
      win_val = graph_open_window("ALBERT values", "300x300+305+0",nil,mesh);
    }
    if (size[1] > 0)
    {
      sprintf(geom, "%dx%d+%d+%d", size[0], size[0], size[0], 0);
      win_est = graph_open_window("ALBERT estimate", "300x300+610+0",nil,mesh);
    }
    if (size[2] > 0)
    {
      sprintf(geom, "%dx%d+%d+%d", size[2], size[2], size[1], 0);
      win_mesh = graph_open_window("ALBERT mesh", "300x300+0+0",nil,mesh);
    }
    first = false;
  }

  if (mesh && win_mesh)
  {
    graph_clear_window(win_mesh, rgb_black);
    graph_mesh(win_mesh, mesh, rgb_white, 0);
  }
  if (u_h && win_val)
  {
    graph_clear_window(win_val, rgb_black);
    graph_values(win_val, u_h, 0);
  }
  if (get_est && win_est)
  {
    graph_clear_window(win_est, rgb_black);
    graph_el_est(win_est, mesh, get_est);
  }
#else
  MSG("only in 2d available, please use gltools graphics\n");
#endif
  WAIT;

  return;
```

```
}
#endif
```

3.1.2 Include file and global variables

All ALBERT source files must include the header file `albert.h` with all ALBERT type definitions, function prototypes and macro definitions:

```
#include <albert.h>
```

For a linear elliptic problem we need 4 global variables:

`fe_space`: a pointer to the actually used finite element space; it is initialized by the function `init_dof_admin()` which is called by `GET_MESH()`, see Section 3.1.3;

`u_h`: a pointer to a DOF vector storing the coefficients of the discrete solution; it is initialized on the first call of `build()` which is called by `adapt_method_stat()`, see Section 3.1.6;

`f_h`: a pointer to a DOF vector storing the load vector; it is initialized on the first call of `build()`;

`matrix`: a pointer to a DOF matrix storing the system matrix; it is initialized on the first call of `build()`;

```
static const FE_SPACE *fe_space;         /* initialized by init_dof_admin() */
static DOF_REAL_VEC   *u_h = nil;        /* initialized by build()          */
static DOF_REAL_VEC   *f_h = nil;        /* initialized by build()          */
static DOF_MATRIX     *matrix = nil;     /* initialized by build()          */
```

3.1.3 Initialization of the finite element space

During the initialization of the mesh by `GET_MESH()` in the main program, we have to specify all DOFs which we want to use during the simulation on the mesh. The initialization of the DOFs is implemented in the function `init_dof_admin()` which is called by `GET_MESH()`.

For an elliptic problem we need one finite element space for the discretization. In this example, we use Lagrange elements and we initialize the degree of the elements via a parameter. The corresponding `fe_space` is accessed by `get_fe_space()` which automatically gives the mesh information about the DOFs used by this finite element space.

It is possible to access several finite element spaces inside this function, compare Section 2.6.2.

```
void init_dof_admin(MESH *mesh)
{
  FUNCNAME("init_dof_admin");
  int             degree = 1;
  const BAS_FCTS  *lagrange;

  GET_PARAMETER(1, "polynomial degree", "%d", &degree);
  lagrange = get_lagrange(degree);
  TEST_EXIT(lagrange)("no lagrange BAS_FCTS\n");
  fe_space = get_fe_space(mesh, lagrange->name, nil, lagrange);
  return;
}
```

3.1.4 Functions for leaf data

As mentioned in Section 2.2.12 we can "hide" information which is only needed on leaf element at the pointer to the second child. Such information is the local error indicator on an element. For this elliptic problem we need one `REAL` for storing this element indicator.

During mesh initialization by `GET_MESH()` in the main program, we have to give information about the size of leaf data to be stored and how to transform leaf data from parent to children during refinement and vice versa during coarsening. The function `init_leaf_data()` initializes the leaf data used for this problem and is called by `GET_MESH()`. Here, leaf data is one structure `struct ellipt_leaf_data` and no transformation during mesh modifications takes places.

The error estimation is done by the library function `ellipt_est()`, see Section 2.14.1. For `ellipt_est()`, we need a function which gives read and write access to the local element error and for the marking function of the adaptive procedure we need a function which returns the local error indicator, see Section 2.13.1. The indicator is stored as the `REAL` member `estimate` of `struct ellipt_leaf_data` and the function `rw_el_est()` returns for each element a pointer to this member. The function `get_el_est()` returns the value stored at that member for each element.

```
struct ellipt_leaf_data
{
  REAL estimate;            /*  one real for the element indicator           */
};

void init_leaf_data(LEAF_DATA_INFO *leaf_data_info)
{
  FUNCNAME("init_leaf_data");
  leaf_data_info->leaf_data_size = sizeof(struct ellipt_leaf_data);
  leaf_data_info->coarsen_leaf_data = nil;
  leaf_data_info->refine_leaf_data = nil;
  return;
}

static REAL *rw_el_est(EL *el)
{
  FUNCNAME("rw_el_est");

  if (IS_LEAF_EL(el))
    return(&((struct ellipt_leaf_data *)LEAF_DATA(el))->estimate);
  else
    return(nil);
}

static REAL get_el_est(EL *el)
{
  FUNCNAME("get_el_est");

  if (IS_LEAF_EL(el))
    return(((struct ellipt_leaf_data *)LEAF_DATA(el))->estimate);
  else
    return(0.0);
}
```

3.1.5 Data of the differential equation

Data for the Poisson equation are the right hand side f and boundary values g. For test purposes it is convenient to have access to an exact solution of the problem. In this example we use the function

$$u(x) = \mathrm{e}^{-10\,|x|^2}$$

as exact solution, resulting in

$$\nabla u(x) = -20\,x\,\mathrm{e}^{-10\,|x|^2}$$

and

$$f(x) = -\Delta u(x) = -(400\,|x|^2 - 20\,d)\,\mathrm{e}^{-10\,|x|^2}.$$

The functions `u` and `grd_u` are the implementation of u and ∇u and are optional. The functions `g` and `f` are implementations of the boundary values and the right hand side and are not optional.

```
static REAL u(const REAL_D x)
{
  FUNCNAME("u");
  return(exp(-10.0*SCP_DOW(x,x)));
}

static const REAL *grd_u(const REAL_D x)
{
  FUNCNAME("grd_u");
  static REAL_D grd;
  REAL          ux = exp(-10.0*SCP_DOW(x,x));

  grd[0] = -20.0*x[0]*ux;
  grd[1] = -20.0*x[1]*ux;
#if DIM_OF_WORLD == 3
  grd[2] = -20.0*x[2]*ux;
#endif
  return(grd);
}

static REAL g(const REAL_D x)              /* boundary values, not optional */
{
  FUNCNAME("g");
  return(u(x));
}

static REAL f(const REAL_D x)              /* -Delta u, not optional        */
{
  FUNCNAME("f");
  REAL  r2 = SCP_DOW(x,x), ux  = exp(-10.0*r2);
  return(-(400.0*r2 - 20.0*DIM)*ux);
}
```

3.1.6 The assemblage of the discrete system

For the assemblage of the discrete system we use the tools described in Sections 2.12.2, 2.12.4, and 2.12.5. For the matrix assemblage we have to provide the functions `init_element()` and `LALt()` for the evaluation of $\det |DF_S| \Lambda A \Lambda^t$ on single elements, where Λ is the Jacobian of the barycentric coordinates (for $-\Delta$ we have $A = id$).

For passing information about the Jacobian Λ of the barycentric coordinates and $\det |DF_S|$ from the function `init_element()` to the function `LALt` we use the data structure `struct op_info` which stores the Jacobian and the determinant. The function `init_element()` calculates the Jacobian and the determinant by the library function `el_grd_lambda()` and the function `LALt()` uses these values in order to compute $\det |DF_S| \Lambda \Lambda^t$.

Pointers to these functions and to one structure `struct op_info` are members of a structure `OPERATOR_INFO` which is used for the initialization of a function for the automatic assemblage of the system matrix (see also Example 2.12.1 in Section 2.12.2 for the access of a structure `matrix_info`). For more general equations, additional functions `Lb0`, `Lb1`, and/or `c` have to be defined at that point. The functions `init_element()` and `LALt()` only work properly in this version for `DIM_OF_WORLD == DIM`.

This initialization is done on the first call of the function `build()` which is called by `adapt_method_stat()` (compare Section 2.13.1). On the first call, `build()` also accesses the DOF vectors `u_h` and `f_h`, and the DOF matrix `matrix`. The vector `u_h` additionally is initialized with zeros and the function pointers for an automatic interpolation during refinement and coarsening are adjusted to the predefined functions in `fe_space->bas_fcts`. The load vector `f_h` and the system matrix `matrix` are newly assembled on each call of `build()`. Thus, they may not be interpolated during mesh modifications and may not be initialized.

On each call of `build()` the matrix is assembled by first clearing the matrix using the function `clear_dof_matrix()` and then adding element contributions by `update_matrix()`. This function will call `init_element()` and `LALt()` on each element.

The load vector `f_h` is then initialized with zeros and the right hand side is added by `L2scp_fct_bas()`. Finally, Dirichlet boundary values are set for all Dirichlet DOFs in the load vector and the discrete solution `u_h` by `dirichlet_bound()` (compare Section 2.12.4 and 2.12.5).

```
struct op_info
{
  REAL_D  Lambda[DIM+1];    /*  the gradient of the barycentric coordinates */
  REAL    det;              /*  det |D F_S|                                 */
};

static void init_element(const EL_INFO *el_info, const QUAD *quad[3], void *ud)
{
  FUNCNAME("init_element");
  struct op_info *info = ud;

  info->det = el_grd_lambda(el_info, info->Lambda);
  return;
}

const REAL (*LALt(const EL_INFO *el_info, const QUAD *quad,
                  int iq, void *ud))[DIM+1]
{
  FUNCNAME("LALt");
```

```
  struct op_info *info = ud;
  int            i, j, k;
  static REAL    LALt[DIM+1][DIM+1];

  for (i = 0; i <= DIM; i++)
    for (j = i; j <= DIM; j++)
    {
      for (LALt[i][j] = k = 0; k < DIM_OF_WORLD; k++)
        LALt[i][j] += info->Lambda[i][k]*info->Lambda[j][k];
      LALt[i][j] *= info->det;
      LALt[j][i] = LALt[i][j];
    }

  return((const REAL (*)[DIM+1]) LALt);
}

static void build(MESH *mesh, U_CHAR flag)
{
  FUNCNAME("build");
  static const EL_MATRIX_INFO *matrix_info = nil;
  const QUAD                  *quad;

  dof_compress(mesh);
  MSG("%d DOFs for %s\n", fe_space->admin->size_used, fe_space->name);

  if (!u_h)                 /*  access matrix and vector for linear system */
  {
    matrix = get_dof_matrix("A", fe_space);
    f_h    = get_dof_real_vec("f_h", fe_space);
    u_h    = get_dof_real_vec("u_h", fe_space);
    u_h->refine_interpol = fe_space->bas_fcts->real_refine_inter;
    u_h->coarse_restrict = fe_space->bas_fcts->real_coarse_inter;
    dof_set(0.0, u_h);      /*  initialize u_h  !                          */
  }

  if (!matrix_info)           /* information for matrix assembling         */
  {
    OPERATOR_INFO  o_info = {nil};

    o_info.row_fe_space   = o_info.col_fe_space = fe_space;
    o_info.init_element   = init_element;
    o_info.LALt           = LALt;
    o_info.LALt_pw_const  = true;        /* pw const. assemblage is faster */
    o_info.LALt_symmetric = true;        /* symmetric assemblage is faster */
    o_info.use_get_bound  = true;        /* Dirichlet boundary conditions! */
    o_info.user_data = MEM_ALLOC(1, struct op_info);        /* user data! */
    o_info.fill_flag = CALL_LEAF_EL|FILL_COORDS;

    matrix_info = fill_matrix_info(&o_info, nil);
  }

  clear_dof_matrix(matrix);                /* assembling of matrix         */
  update_matrix(matrix, matrix_info);
```

```
  dof_set(0.0, f_h);                         /* assembling of load vector    */
  quad = get_quadrature(DIM, 2*fe_space->bas_fcts->degree - 2);
  L2scp_fct_bas(f, quad, f_h);

  dirichlet_bound(g, f_h, u_h, nil);           /*  boundary values         */
  return;
}
```

3.1.7 The solution of the discrete system

The solution of the resulting linear system is rather easy and we can use a preconditioned Krylov-space solver (`oem_solve_s()`), compare Section 2.15.2. On the first call of `solve()`, the parameters for the linear solver are initialized and stored in `static` variables. For the OEM solver we have to initialize the `solver`, the tolerance `tol` for the residual, a maximal number of iterations `miter`, the use of a preconditioner by the parameter `icon`, which may be `0` (no preconditioning) or `1` (diagonal preconditioning), the dimension of the Krylov-space for the minimizing procedure, if GMRes is used, and the level of information of the linear solver.

After solving the discrete system, the discrete solution (and mesh) is displayed.

The function `solve()` is called by `adapt_method_stat()` (compare Section 2.13.1).

```
static void solve(MESH *mesh)
{
  FUNCNAME("solve");
  static REAL       tol = 1.e-8;
  static int        miter = 1000, info = 2, icon = 1, restart = 0;
  static OEM_SOLVER solver = NoSolver;

  if (solver == NoSolver)
  {
    tol = 1.e-8;
    GET_PARAMETER(1, "solver", "%d", &solver);
    GET_PARAMETER(1, "solver tolerance", "%f", &tol);
    GET_PARAMETER(1, "solver precon", "%d", &icon);
    GET_PARAMETER(1, "solver max iteration", "%d", &miter);
    GET_PARAMETER(1, "solver info", "%d", &info);
    if (solver == GMRes)
      GET_PARAMETER(1, "solver restart", "%d", &restart);
  }
  oem_solve_s(matrix, f_h, nil, u_h, solver, tol, icon, restart, miter, info);

  graphics(mesh, u_h, nil);
  return;
}
```

3.1.8 Error estimation

The estimation of the error for an elliptic problem with constant coefficients in the second order term is done by the library function `ellipt_est()` (compare Section 2.14.1). This function needs a pointer to a function for writing the local error indicators (the function `rw_el_est()` described above in Section 3.1.4) and a function for the evaluation of the lower order terms of the element residuals at quadrature nodes. For the Poisson equation, this function has to return the negative value of the right hand side f at that node (which is implemented in `r()`). Since we only have to evaluate the right hand side f, the init flag `r_flag` is zero. For an equation with lower order term this flag has to be `INIT_UH` and/or `INIT_GRD_UH`, if needed by `r()` (compare Example 2.14.1).

The function `estimate()` which is called by `adapt_method_stat()` first initializes parameters for the error estimator, like the estimated norm and constants in front of the residuals. On each call the error estimate is computed by `ellipt_est()`. Additionally, the true error is calculated by the function `H1_err()` or `L2_err()`, the ratio of the true and estimated error is printed (which should be approximately constant). The experimental order of convergence is calculated, which should be `fe_space->bas_fcts->degree` for the H^1 and `fe_space->bas_fcts->degree+1` for the L^2 norm when using global refinement with `DIM` bisection refinements. Finally, the error indicators are displayed.

```
static REAL r(const EL_INFO *el_info, const QUAD *quad, int iq, REAL uh_iq,
              const REAL_D grd_uh_iq)
{
  FUNCNAME("r");
  REAL_D      x;
  coord_to_world(el_info, quad->lambda[iq], x);
  return(-f(x));
}

#define EOC(e,eo) log(eo/MAX(e,1.0e-15))/M_LN2

static REAL estimate(MESH *mesh, ADAPT_STAT *adapt)
{
  FUNCNAME("estimate");
  static int     degree, norm = -1;
  static REAL    C[3] = {1.0, 1.0, 0.0};
  static REAL    est, est_old = -1.0, err, err_old = -1.0, max_e;
  static const REAL_DD A = {{1.0,0.0}, {0.0,1.0}};
  static FLAGS r_flag = 0;  /* = (INIT_UH | INIT_GRD_UH),  if needed by r() */

  if (norm < 0)
  {
    norm = H1_NORM;
    GET_PARAMETER(1, "error norm", "%d", &norm);
    GET_PARAMETER(1, "estimator C0", "%f", &C[0]);
    GET_PARAMETER(1, "estimator C1", "%f", &C[1]);
    GET_PARAMETER(1, "estimator C2", "%f", &C[2]);
  }
  degree = 2*u_h->fe_space->bas_fcts->degree;
  est = ellipt_est(u_h, adapt, rw_el_est, nil, degree, norm, C, A, r, r_flag);

  MSG("estimate   = %.8le", est);
  if (est_old >= 0)
    print_msg(", EOC: %.2lf\n", EOC(est,est_old));
  else
    print_msg("\n");
  est_old = est;

  if (norm == L2_NORM)
    err = L2_err(u, u_h, nil, 0, nil, &max_e);
```

```
  else
    err = H1_err(grd_u, u_h, nil, 0, nil, &max_e);

  MSG("||u-uh||%s = %.8le", norm == L2_NORM ? "L2" : "H1", err);
  if (err_old >= 0)
    print_msg(", EOC: %.2lf\n", EOC(err,err_old));
  else
    print_msg("\n");
  err_old = err;
  MSG("||u-uh||%s/estimate = %.2lf\n", norm == L2_NORM ? "L2" : "H1",
      err/MAX(est,1.e-15));

  graphics(mesh, nil, get_el_est);
  return(adapt->err_sum);
}
```

3.1.9 The main program for the Poisson equation

The main program now becomes very simple: We first read a parameter file, initialize the mesh, and read the macro triangulation (including an initial global refinement, if necessary).

For the adaptive procedure a pre–initialized data structure `ADAPT_STAT` is accessed by the function `get_adapt_stat()`; the members `adapt->strategy`, `adapt->tolerance`, `adapt->max_iteration`, and `adapt->info` are automatically initialized with values from the parameter file described below in Section 3.1.10; other members can be also initialized by adding similar lines for these members to the parameter file (compare Section 2.13.4).

After the access of the structure and the initialization of parameters, function pointers for the problem dependent routines have to be set (`adapt->estimate`, `adapt->get_el_est`, `adapt->build_after_coarsen`, `adapt->solve`). Since the assemblage is done in one step, only the function pointer `adapt->build_after_coarsen` is adjusted, no assemblage is done before refinement and before coarsening. The adaptive procedure is finally started by a call of `adapt_method_stat()`.

```
int main(int argc, char **argv)
{
  FUNCNAME("main");
  MESH   *mesh;
  int    n_refine = 0;
  static ADAPT_STAT *adapt;
  char   filename[100];

/****************************************************************************/
/*  first of all, init parameters of the init file                          */
/****************************************************************************/

  init_parameters(0, "INIT/ellipt.dat");

/****************************************************************************/
/*  get a mesh, and read the macro triangulation from file                  */
/****************************************************************************/

  mesh = GET_MESH("ALBERT mesh", init_dof_admin, init_leaf_data);
  mesh->preserve_coarse_dofs = 0;
```

```
  GET_PARAMETER(1, "macro file name", "%s", filename);
  read_macro(mesh, filename, nil);
  GET_PARAMETER(1, "global refinements", "%d", &n_refine);
  global_refine(mesh, n_refine*DIM);
  graphics(mesh, nil, nil);

/****************************************************************************/
/*  init adapt structure and start adaptive method                          */
/****************************************************************************/

  adapt = get_adapt_stat("ellipt", "adapt", 1, nil);
  adapt->estimate = estimate;
  adapt->get_el_est = get_el_est;
  adapt->build_after_coarsen = build;
  adapt->solve = solve;

  adapt_method_stat(mesh, adapt);
  WAIT_REALLY;
  return(0);
}
```

3.1.10 The parameter file for the Poisson equation

The following parameter file `INIT/ellipt.dat` is used for the above described program:

```
macro file name:      Macro/macro.stand
global refinements:   0
polynomial degree:    3

graphic windows:      400 400 0

solver:                2 % 1: BICGSTAB 2: CG 3: GMRES 4: ODIR 5: ORES
solver max iteration:  1000
solver restart:        10  %  only used for GMRES
solver tolerance:      1.e-8
solver info:           2
solver precon:         1   % 0: no precon 1: diag precon

error norm:           1   % 1: H1_NORM, 2: L2_NORM
estimator C0:         0.1 % constant of element residual
estimator C1:         0.1 % constant of jump residual
estimator C2:         0.0 % constant of coarsening estimate

adapt->strategy:      2   % 0: no adaption 1: GR 2: MS 3: ES 4:GERS
adapt->tolerance:     1.e-4
adapt->MS_gamma:      0.5
adapt->max_iteration: 10
adapt->info:          8

WAIT: 1
```

The file `Macro/macro.stand` storing data about the macro triangulation for $\Omega = (0,1)^d$ can be found in Section 2.2.16 for 2d and 3d.

The solution and corresponding mesh for the above parameters are shown in Figure 3.1.

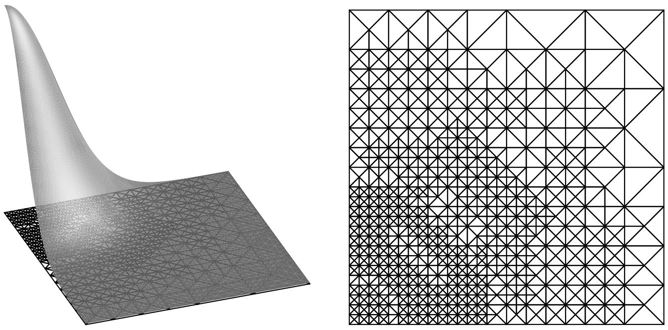


Figure 3.1: Solution of the linear Poisson problem and corresponding mesh. The pictures were produced by GRAPE.

3.2 Nonlinear Poisson equation

In this section we discuss the implementation of the following nonlinear reaction–diffusion equation:

$$-k\Delta u + \sigma\, u^4 = f + \sigma\, u_{ext}^4 \qquad \text{in } \Omega, \tag{3.1a}$$

$$u = g \qquad \text{on } \partial\Omega. \tag{3.1b}$$

For $\Omega \subset \mathbb{R}^2$, this equation models the heat transport in a thin plate Ω, heated by an external heat source f and which radiates heat. Here, k is the constant heat conductivity, σ the Stefan–Bolzmann constant, g the temperature at the edges of the plate and u_{ext} the temperature of the surrounding space (absolute temperature in $°K$).

The solver is applied to following data:

- For testing the solver in 2d and 3d we again use the 'exponential peak'

$$u(x) = e^{-10\,|x|^2}, \qquad x \in \Omega = (-1,1)^d,\ k = 1,\ \sigma = 1,\ u_{ext} = 0.$$

- In general (due to the nonlinearity), the problem is not uniquely solvable; depending on the initial guess for the nonlinear solver at least two discrete solutions can be obtained by using data

$$\Omega = (0,1)^d,\ k = 1,\ \sigma = 1,\ f \equiv 1,\ g \equiv 0,\ u_{ext} = 0.$$

  and the interpolant of

$$u_0(x) = 4^d\, U_0 \prod_{i=1}^{d} x_i(1 - x_i) \qquad \text{with } U_0 \in [-5.0, 1.0].$$

  as initial guess for the discrete solution on the coarsest grid.

- The last application now addresses a physical problem in 2d with following data:

$$\Omega = (-1,1)^2,\ k = 2,\ \sigma = 5.67\text{e-}8,\ g \equiv 300,\ u_{ext} = 273,\ f(x) = \begin{cases} 150, & if x \in (-\frac{1}{2}, \frac{1}{2})^2, \\ 0, & \text{otherwise.} \end{cases}$$

3.2.1 The Newton method

We use a Newton method for the solution of the nonlinear equation

$$u_h \in g_h + \mathring{X}_h : \qquad F(u_h) = 0 \qquad \text{in } \mathring{X}_h^*,$$

where $g_h \in X_h$ is an approximation to boundary data g. The classical Newton method reads: Start with an initial guess $u_0 \in g_h + \mathring{X}_h$. For $m \geq 0$ compute

$$d_m \in \mathring{X}_h : \qquad DF(u_m)d_m = F(u_m) \qquad \text{in } \mathring{X}_h^*$$

and set

$$u_{m+1} = u_m - d_m$$

until some suitable norm $\|d_m\|$ or $\|F(u_{m+1})\|$ is sufficiently small. Since the correction d_m satisfies $d_m \in \mathring{X}_h$, all Newton iterates u_m satisfy $u_m \in g_h + \mathring{X}_h$, $m \geq 0$.

The solution of such a nonlinear problem, needs several assemblage and solve steps (in each Newton step we have to assemble and solve a linearized system). Thus, we do not split the assemble and solve routines in `build()` and `solve()` as in the solver for the Poisson equation (compare Sections 3.1.6 and 3.1.7), but only set Dirichlet boundary values for the initial guess in `build()` and solve the nonlinear equation (including the assemble of linearized systems) in `solve()` (compare Sections 3.2.5 and 3.2.6).

3.2.2 Program organization and header file

The implementation is split into three source files:

- `nonlin.c`: main program with all subroutines for the adaptive procedure; initializes DOFs, leaf data and problem dependent data in `main()` and the `solve()` routine calls the nonlinear solver;
- `nlprob.c`: definition of problem dependent data;
- `nlsolve.c`: implementation of the nonlinear solver.

Data structures used in all source files, and prototypes of functions are defined in the header file `nonlin.h`, which includes the `albert.h` header file on the first line. This file is included by all source files.

```
#include <albert.h>

typedef struct prob_data PROB_DATA;
struct prob_data
{
  REAL          k, sigma;

  REAL          (*g)(const REAL_D x);
  REAL          (*f)(const REAL_D x);

  REAL          (*u0)(const REAL_D x);

  REAL          (*u)(const REAL_D x);
  const REAL    *(*grd_u)(const REAL_D x);
};
```

```
/*--- file nlprob.c -------------------------------------------------------*/
const PROB_DATA *init_problem(MESH *mesh);

/*--- file nlsolve.c ------------------------------------------------------*/
int nlsolve(DOF_REAL_VEC *, REAL, REAL, REAL (*)(const REAL_D));

/*--- file graphics.c -----------------------------------------------------*/
void graphics(MESH *, DOF_REAL_VEC *, REAL (*)(EL *));
```

The data structure `PROB_DATA` yields following information:

`k`: diffusion coefficient (constant heat conductivity);

`sigma`: reaction coefficient (Stefan–Bolzmann constant);

`g`: pointer to a function for evaluating boundary values;

`f`: pointer to a function for evaluating the right-hand side ($f + \sigma\, u_{ext}^4$);

`u0`: pointer to a function for evaluating an initial guess for the discrete solution on the macro triangulation, if not `nil`;

`u`: pointer to a function for evaluating the true solution, if not `nil` (only for test purpose);

`grd_u`: pointer to a function for evaluating the gradient of the true solution, if not `nil` (only for test purpose).

3.2.3 Global variables

In the main source file for the nonlinear solver `nonlin.c` we use the following global variables:

```
#include "nonlin.h"

static const FE_SPACE  *fe_space;        /* initialized by init_dof_admin() */
static DOF_REAL_VEC    *u_h = nil;       /* initialized by build()          */
static const PROB_DATA *prob_data = nil; /* initialized by main()           */
```

As in the solver for the linear Poisson equation, we have a pointer to the used `fe_space` and the discrete solution `u_h`. In this file, we do not need a pointer to a `DOF_MATRIX` for storing the system matrix and a pointer to a `DOF_REAL_VEC` for storing the right hand side. The system matrix and right hand side are handled by the nonlinear solver `nlsolve()`, implemented in `nlsolve.c`. Data about the problem is handled via the `prob_data` pointer.

3.2.4 Initialization of the finite element space and leaf data

The functions for initializing DOFs (`init_dof_admin()`), leaf data (`init_leaf_data()`), and for accessing leaf data (`rw_el_est()`, `get_el_est()`) are exactly the same as in the solver for the linear Poisson equation, compare Sections 3.1.3 and 3.1.4.

3.2.5 The build routine

As mentioned above, inside the build routine we only access one vector for storing the discrete solution. On the coarsest grid, the discrete solution is initialized with zeros, or by interpolating the function `prob_data->u0`, which implements an initial guess for the discrete solution. On a refined grid we do not initialize the discrete solution again. Here, we use the discrete solution from the previous step, which is interpolated during mesh modifications, as an initial guess.

In each adaptive cycle, Dirichlet boundary values are set for the discrete solution. This ensures $u_0 \in g_h + \mathring{X}_h$ for the initial guess of the Newton method.

```
static void build(MESH *mesh, U_CHAR flag)
{
  FUNCNAME("build");

  dof_compress(mesh);
  MSG("%d DOFs for %s\n", fe_space->admin->size_used, fe_space->name);

  if (!u_h)                 /*  access and initialize discrete solution    */
  {
    u_h    = get_dof_real_vec("u_h", fe_space);
    u_h->refine_interpol = fe_space->bas_fcts->real_refine_inter;
    u_h->coarse_restrict = fe_space->bas_fcts->real_coarse_inter;
    if (prob_data->u0)
      interpol(prob_data->u0, u_h);
    else
      dof_set(0.0, u_h);
  }
  dirichlet_bound(prob_data->g, u_h, nil, nil); /* set boundary values     */

  return;
}
```

3.2.6 The solve routine

The `solve()` routine solves the nonlinear equation by calling the function `nlsolve()` which is implemented in `nlsolve.c` and described below. After solving the discrete problem, the new discrete solution is displayed via the `graphics()` routine.

```
static void solve(MESH *mesh)
{
  nlsolve(u_h, prob_data->k, prob_data->sigma, prob_data->f);
  graphics(mesh, u_h, nil);
  return;
}
```

3.2.7 The estimator for the nonlinear problem

The function `r()` which implements the lower order term changes due to the term σu^4 in the differential operator. The right hand side $f + \sigma u_{ext}^4$ is implemented in the function `prob_data->f()`.

In the function `estimate` we have to initialize the diagonal of `A` with the heat conductivity `prob_data->k` and for the function `r` we need the values of u_h at the quadrature node, thus `r_flag = INIT_UH` is used. The true error is only computed for the first problem, where the true solution in known (`prob_data->u()` and `prob_data->grd_u()` are not `nil`). Finally, the error indicator is displayed by `graphics()`.

```
static REAL r(const EL_INFO *el_info, const QUAD *quad, int iq, REAL uh_iq,
              const REAL_D grd_uh_iq)
{
  REAL_D      x;
  REAL        uhx2 = SQR(uh_iq);
```

```
  coord_to_world(el_info, quad->lambda[iq], x);
  return(prob_data->sigma*uhx2*uhx2 - (*prob_data->f)(x));
}

#define EOC(e,eo) log(eo/MAX(e,1.0e-15))/M_LN2

static REAL estimate(MESH *mesh, ADAPT_STAT *adapt)
{
  FUNCNAME("estimate");
  static int     degree, norm = -1;
  static REAL    C[3] = {1.0, 1.0, 0.0};
  static REAL    est, est_old = -1.0, err = -1.0, err_old = -1.0;
  static REAL_DD A = {{1.0,0.0},{0.0,1.0}};
  static REAL    r_flag = INIT_UH;

  A[0][0] = A[1][1] = prob_data->k;
#if DIM_OF_WORLD == 3
  A[2][2] = prob_data->k;
#endif

  if (norm < 0)
  {
    norm = H1_NORM;
    GET_PARAMETER(1, "error norm", "%d", &norm);
    GET_PARAMETER(1, "estimator C0", "%f", C);
    GET_PARAMETER(1, "estimator C1", "%f", C+1);
    GET_PARAMETER(1, "estimator C2", "%f", C+2);
  }
  degree = 2*u_h->fe_space->bas_fcts->degree;
  est = ellipt_est(u_h, adapt, rw_el_est, nil, degree, norm, C, A, r, r_flag);

  MSG("estimate   = %.8le", est);
  if (est_old >= 0)
    print_msg(", EOC: %.2lf\n", EOC(est,est_old));
  else
    print_msg("\n");
  est_old = est;

  if (norm == L2_NORM  &&  prob_data->u)
    err = L2_err(prob_data->u, u_h, nil, 0, nil, nil);
  else if (norm == H1_NORM  &&  prob_data->grd_u)
    err = H1_err(prob_data->grd_u, u_h, nil, 0, nil, nil);

  if (err >= 0)
  {
    MSG("||u-uh||%s = %.8le", norm == L2_NORM ? "L2" : "H1", err);
    if (err_old >= 0)
      print_msg(", EOC: %.2lf\n", EOC(err,err_old));
    else
      print_msg("\n");
    err_old = err;
    MSG("||u-uh||%s/estimate = %.2lf\n", norm == L2_NORM ? "L2" : "H1",
```

```
        err/MAX(est,1.e-15));
  }
  graphics(mesh, nil, get_el_est);
  return(adapt->err_sum);
}
```

3.2.8 The main program for the nonlinear reaction–diffusion equation

In the main program parameters are initialized from the file `INIT/nonlin.dat`, and parameters from the command line are added. For the definition of a parameter via the command line we need for each parameter a pair of arguments: the key (without terminating ':' and the value.

```
nonlin "problem number" 1
```

will for instance overwrite the value of `problem number` in `nonlin.dat` with the value `1`.

Secondly, the mesh with the used DOFs and leaf data is initialized, problem dependent data, including the macro triangulation, are initialized by `init_problem(mesh)` (see Section 3.2.9), the structure for the adaptive method is filled and finally the adaptive method is started.

```
int main(int argc, char **argv)
{
  FUNCNAME("main");
  MESH       *mesh;
  ADAPT_STAT *adapt;
  int        k;

/*--------------------------------------------------------------------------*/
/*  first of all, init parameters from the init file and command line       */
/*--------------------------------------------------------------------------*/
  init_parameters(1, "INIT/nonlin.dat");
  for (k = 1; k+1 < argc; k += 2)
    ADD_PARAMETER(0, argv[k], argv[k+1]);

/*--------------------------------------------------------------------------*/
/*  get a mesh with DOFs and leaf data                                      */
/*--------------------------------------------------------------------------*/
  mesh = GET_MESH("Nonlinear problem mesh", init_dof_admin, init_leaf_data);

/*--------------------------------------------------------------------------*/
/*  init problem dependent data and read macro triangulation                */
/*--------------------------------------------------------------------------*/
  prob_data = init_problem(mesh);

/*--------------------------------------------------------------------------*/
/*  init adapt struture and start adaptive method                           */
/*--------------------------------------------------------------------------*/
  adapt = get_adapt_stat("nonlin", "adapt", 1, nil);
  adapt->estimate = estimate;
  adapt->get_el_est = get_el_est;
  adapt->build_after_coarsen = build;
  adapt->solve = solve;

  adapt_method_stat(mesh, adapt);
```

```
  WAIT_REALLY;
  return(0);
}
```

3.2.9 Initialization of problem dependent data

The file `nlprob.c` contains problem dependent data. On the first line `nonlin.h` is included and then two global variables used by all functions are declared:

```
#include "nonlin.h"
static REAL  k = 1.0, sigma = 1.0;
```

The following functions are used in the first example for testing the nonlinear solver (Problem 0):

```
static REAL u_0(const REAL_D x)
{
  REAL   x2 = SCP_DOW(x,x);
  return(exp(-10.0*x2));
}

static const REAL *grd_u_0(const REAL_D x)
{
  static REAL_D grd;
  REAL          r2 = SCP_DOW(x,x);
  REAL          ux = exp(-10.0*r2);

  grd[0] = -20.0*x[0]*ux;
  grd[1] = -20.0*x[1]*ux;
#if DIM_OF_WORLD == 3
  grd[2] = -20.0*x[2]*ux;
#endif

  return(grd);
}

static REAL f_0(const REAL_D x)
{
  REAL  r2 = SCP_DOW(x,x), ux  = exp(-10.0*r2), ux4 = ux*ux*ux*ux;
  return(sigma*ux4 - k*(400.0*r2 - 20.0*DIM)*ux);
}
```

For the computation of a stable and an unstable solution, depending on the initial choice of the discrete solution the following functions are used, which also use the global variable `U0` (Problem 1):

```
static REAL  U0 = 0.0;
static REAL g_1(const REAL_D x)
{
#if DIM_OF_WORLD == 2
  return(16*U0*x[0]*(1.0-x[0])*x[1]*(1.0-x[1]));
#else
  return(64*U0*x[0]*(1.0-x[0])*x[1]*(1.0-x[1])*x[2]*(1.0-x[2]));
#endif
}
```

```
static REAL f_1(const REAL_D x)
{
  return(1.0);
}
```

The last example needs functions for boundary data and right hand side and variables for the temperature at the edges, and $\sigma\, u_{ext}^4$ (Problem 2):

```
static REAL g2 = 300.0, sigma_uext4 = 0.0;
static REAL g_2(const REAL_D x)
{
  return(g2);
}

static REAL f_2(const REAL_D x)
{
  if (x[0] >= -0.25  &&  x[0] <= 0.25  &&  x[1] >= -0.25  &&  x[1] <= 0.25)
    return(150.0 + sigma_uext4);
  else
    return(sigma_uext4);
}
```

The function `init_problem()` initializes the entries of a `PROB_DATA` structure, adjusts the corresponding function pointers, reads the macro triangulation, performs some initial refinement and returns a pointer to the filled `PROB_DATA` structure.

```
const PROB_DATA *init_problem(MESH *mesh)
{
  FUNCNAME("init_problem");
  static PROB_DATA prob_data = {0};
  int              pn = 2, n_refine = 0;

  GET_PARAMETER(1, "problem number", "%d", &pn);
  switch (pn)
  {
  case 0:   /*---  problem with known true solution  -----------------------*/
    k = 1.0;
    sigma = 1.0;

    prob_data.g = u_0;
    prob_data.f = f_0;

    prob_data.u = u_0;
    prob_data.grd_u = grd_u_0;

    read_macro(mesh, "Macro/macro_big.stand", nil);
    break;
  case 1:   /*---  problem for computing a stable and an unstable sol.  ----*/
    k = 1.0;
    sigma = 1.0;

    prob_data.g = g_1;
    prob_data.f = f_1;
```

```
    prob_data.u0 = g_1;
    GET_PARAMETER(4, "U0", "%f", &U0);

    read_macro(mesh, "Macro/macro.stand", nil);
    break;
  case 2:   /*---  physical problem  --------------------------------------*/
    k = 2.0;
    sigma = 5.67e-8;
    sigma_uext4 = sigma*273*273*273*273;

    prob_data.g = g_2;
    prob_data.f = f_2;
    read_macro(mesh, "Macro/macro_big.stand", nil);
    break;
  default:
    ERROR_EXIT("no problem defined with problem no. %d\n", pn);
  }
  prob_data.k = k;
  prob_data.sigma = sigma;

  GET_PARAMETER(1, "global refinements", "%d", &n_refine);
  global_refine(mesh, n_refine*DIM);

  return(&prob_data);
}
```

3.2.10 Implementation of the nonlinear solver

We use the Newton solvers described in Section 2.15.6 for solving $F(u_h) = 0$ in $\mathring{X}_h$ for the nonlinear Poisson equation (3.1). We start with an initial guess $u_0 \in g_h + \mathring{X}_h$, where Dirichlet boundary values are set in the `build()` routine (compare Section 3.2.5).

For $v \in X_h$ the functional $F(v) \in \mathring{X}_h^*$ is defined by

$$\big\langle F(v),\, \varphi\big\rangle_{\mathring{X}^*\times\mathring{X}} = \int_\Omega k\nabla\varphi_j\nabla v + \sigma\,\varphi_j\,v^4\,dx - \int_\Omega (f + u_{ext}^4)\varphi_j\,dx \qquad \text{for all } \varphi_j \in \mathring{X}_h, \tag{3.2}$$

and the Frechet derivative $DF(v)$ of F is given by

$$\big\langle DF(v)\,\varphi_i,\, \varphi_j\big\rangle_{\mathring{X}^*\times\mathring{X}} = \int_\Omega k\nabla\varphi_j\nabla\varphi_i + 4\sigma\,v^3\,\varphi_j\,\varphi_i\,dx \qquad \text{for all } \varphi_i,\, \varphi_j \in \mathring{X}_h. \tag{3.3}$$

The Newton solvers need a function for assembling the right hand side vector of the discrete system (3.2), and the system matrix of the linearized equation (3.3) for some given v in X_h. The system matrix is always symmetric. It is positive definite, if $v \geq 0$, and is then solved by the conjugate gradient method. If $v \not\geq 0$ then BiCGStab is used. We choose the H^1 semi–norm for problem dependent norm $\|.\|$.

Problem dependent data structures for assembling and solving

Similar to the assemblage of the system matrix for the Poisson problem, we define a data structure `struct op_info` in order to pass information to the routines which describe the differential operator. In the assembling of the linearized system around a given finite element function v we additionally need the diffusion coefficient k and reaction coefficient σ. In general, v is not constant on the elements, thus we have to compute the zero order term by numerical quadrature on each element. For this we need access to the used quadrature for this term, and a vector storing the values of v for all quadrature nodes.

```
struct op_info
{
  REAL_D  Lambda[DIM+1];    /*  the gradient of the barycentric coordinates */
  REAL    det;              /*  |det D F_S|                                 */

  REAL    k, sigma;         /*  diffusion and reaction coefficient          */

  const QUAD_FAST *quad_fast;  /*  quad_fast for the zero order term        */
  const REAL      *v_qp;       /*  v at all quadrature nodes of quad_fast   */
};
```

The general Newton solvers pass data about the actual problem by `void` pointers to the problem dependent routines. Information that is used by these routines are collected in the data structure `NEWTON_DATA`

```
typedef struct newton_data NEWTON_DATA;
struct newton_data
{
  const FE_SPACE   *fe_space;     /* used finite element space              */

  REAL       k;                   /* diffusion coefficient                  */
  REAL       sigma;               /* reaction coefficient                   */
  REAL       (*f)(const REAL_D);  /* for evaluation f + sigma u_ext^4       */

  DOF_MATRIX *DF;                 /* pointer to system matrix               */

/*---  parameters for the linear solver  -----------------------------------*/
  OEM_SOLVER solver;              /* used solver: CG (v >= 0) else GMRes    */
  REAL       tolerance;
  int        max_iter;
  int        icon;
  int        restart;
  int        info;
};
```

All entries of this structure besides `solver` are initialized in the function `nlsolve()`. The entry `solver` is set every time the linearized matrix is assembled.

The assembling routine

Denote by $\{\varphi_0, \ldots, \varphi_{\mathring{N}}\}$ the basis of $\mathring{X}_h$, by $\{\varphi_0, \ldots, \varphi_N\}$ the basis of X_h. Let $\boldsymbol{A}$ be the stiffness matrix, i.e.

$$A_{ij} = \begin{cases} \int_\Omega k \nabla\varphi_j \nabla\varphi_i \, dx & i = 0, \ldots, N,\ j = 0, \ldots, \mathring{N} \\ \delta_{ij} & i = 0, \ldots, N,\ j = \mathring{N}+1, \ldots, N, \end{cases}$$

and $\boldsymbol{M} = \boldsymbol{M}(v)$ the mass matrix, i.e.

$$M_{ij} = \begin{cases} \int_\Omega \sigma\, v^3\, \varphi_j\, \varphi_i \, dx & i = 0, \ldots, N,\ j = 0, \ldots, \mathring{N} \\ 0 & i = 0, \ldots, N,\ j = \mathring{N}+1, \ldots, N. \end{cases}$$

The system matrix $\boldsymbol{L}$, representing $DF(v)$, of the linearized equation is then given as

$$\boldsymbol{L} = \boldsymbol{A} + 4\boldsymbol{M}.$$

The right hand side vector $\boldsymbol{F}$, representing $F(v)$ is for all non–Dirichlet DOFs j given by

$$\begin{aligned} F_j &= \int_\Omega k\nabla v\nabla\varphi_j + \sigma\, v^4\,\varphi_j\,dx - \int_\Omega (f + \sigma u_{ext}^4)\varphi_j\,dx \\ &= (\boldsymbol{A}\,\boldsymbol{v} + \boldsymbol{M}\,\boldsymbol{v})_j - \int_\Omega (f + \sigma u_{ext}^4)\varphi_j\,dx, \end{aligned} \tag{3.4}$$

where $\boldsymbol{v}$ denotes the coefficient vector of v. Thus, we want to use information assembled into $\boldsymbol{A}$ and $\boldsymbol{M}$ for both system matrix and right hand side vector.

Unfortunately, this can not be done *after* assembling $\boldsymbol{A}+4\,\boldsymbol{M}$ into the system matrix $\boldsymbol{L}$ due to the different scaling of $\boldsymbol{M}$ in the system matrix (factor 4) and right hand side (factor 1). Storing both matrices $\boldsymbol{A}$ *and* $\boldsymbol{M}$ is too costly, since matrices are the objects in finite element codes which need most memory.

The solution to this problem comes from the observation, that (3.4) holds also element-wise for the element contributions of the right hand side and element matrices $\boldsymbol{A}_S$ and $\boldsymbol{M}_S$ when replacing $\boldsymbol{v}$ by the local coefficient vector $\boldsymbol{v}_S$. Hence, on elements we compute the element contributions of $\boldsymbol{A}_S$ and $\boldsymbol{M}_S$, add them to the system matrix, and use them and the local coefficient vector $\boldsymbol{v}_S$ for adding the right hand side contribution to the load vector.

The resulting assembling routine is more complicated in comparison to the routine used for the linear Poisson problem. On the other hand, using ALBERT routines for the computation of element matrices, extracting local coefficient vectors, and boundary information the routine is still rather easy to implement. The implementation still does not depend on the actually used set of local basis functions.

The function `update()` which is now described in detail, can be seen as an example for the flexible implementation of rather complex nonlinear and time dependent problems which often show the same structure (compare the implementation of the assembling routine for the time dependent heat equation, Section 3.3.6). The assemblage is still quite easy, whereas the resulting code is quite efficient.

Similar to the linear Poisson solver, we provide a function `LALt()` for the second order term. Here we also need a function `c()` for the zero order term. This term is assembled using element wise quadrature and thus needs information about the function v used in the linearization at all quadrature nodes. Information for `LALt()` and `c()` is stored in the data structure `struct op_info`, see above. The members of this structure are initialized during mesh traversal in `update()`.

```
static const REAL (*LALt(const EL_INFO *el_info, const QUAD *quad,
                         int iq, void *ud))[DIM+1]
{
  struct op_info *info = ud;
  REAL           fac = info->k*info->det;
  int            i, j, k;
  static REAL    LALt[DIM+1][DIM+1];

  for (i = 0; i <= DIM; i++)
  {
    for (j = i; j <= DIM; j++)
    {
      for (LALt[i][j] = k = 0; k < DIM_OF_WORLD; k++)
```

```
        LALt[i][j] += info->Lambda[i][k]*info->Lambda[j][k];
      LALt[i][j] *= fac;
      LALt[j][i] = LALt[i][j];
    }
  }
  return((const REAL (*)[DIM+1]) LALt);
}

static REAL c(const EL_INFO *el_info, const QUAD *quad, int iq, void *ud)
{
  struct op_info *info = ud;
  REAL   v3;

  TEST_EXIT(info->quad_fast->quad == quad)("quads differ\n");
  v3 = info->v_qp[iq]*info->v_qp[iq]*info->v_qp[iq];
  return(info->sigma*info->det*v3);
}
```

As mentioned above, we use a general Newton solver and a pointer to the `update()` routine is adjusted in the function `nlsolve()` in the data structure for this solver. Such a solver does not have any information about the actual problem, nor information about the ALBERT data structures for storing DOF vectors and matrices. This is also reflected in the arguments of `update()`:

```
static void update(void *ud, int dim, const REAL *v, int up_DF, REAL *F);
```

Here, `dim` is the dimension of the discrete nonlinear problem, `v` is a *vector* storing the coefficients of the finite element function which is used for the linearization, `up_DF` is a flag indicating that $DF(v)$ should be assembled or not. If `F` is not `nil`, then $F(v)$ should be assembled and stored in the *vector* `F`. Information about the ALBERT finite element space, a pointer to a DOF matrix, etc. can be passed to `update()` by the `ud` pointer. The declaration

```
  NEWTON_DATA    *data = ud;
```

converts the `void *` pointer into a pointer `data` to a structure `NEWTON_DATA` which gives access to all information, used for the assembling (see above). The structure is initialized in `nlsolve()` before starting the Newton method.

The `update()` routine is split into three main parts: an initialization of the assembling functions (only done on the first call), a conversion of the vectors that are arguments to the routine into DOF vectors, and finally the assembling.

Initialization of the assembling functions The initialization of ALBERT functions for the assembling is similar to the initialization in the `build()` routine of the linear Poisson equation (compare Section 3.1.6). There are minor differences:

1. In addition to the assemblage of the 2nd order term (see the function `LALt()`), we now have to assemble the zero order term too (see the function `c()`). The integration of the zero order term has to be done by using an element wise quadrature which needs the values of v^3 at all quadrature nodes. The two element matrices are computed separately. This makes it possible to use them for the system matrix and right hand side.

2. In the solver for the Poisson problem, we have filled an `OPERATOR_INFO` structure with information about the differential operator. This structure is an argument to `fill_matrix_info()` which returns a pointer to a structure `EL_MATRIX_INFO`. This pointer is used for the complete

assemblage of the system matrix by some ALBERT routine. Here, we want to use only the function for computing the element matrices. Thus, we only need the entries `el_matrix_fct()` and `fill_info` of the `EL_MATRIX_INFO` structure, which are used to compute the element matrix (`fill_info` is the second argument to `el_matrix_fct()`). We initialize a function pointer `fill_a` with data pointer `a_info` for the computation of the element matrix $\boldsymbol{A}_S$ and a function pointer `fill_c` with data pointer `c_info` for the computation $\boldsymbol{M}_S$.

All other information inside the `EL_MATRIX_INFO` structure is used for the automatic assembling of element matrices into the system matrix by `update_matrix()`. Such information can be ignored, since this is now done in `update()`.

3. For the assembling of the element matrix into the system matrix and the element contribution of the right hand side into the load vector we need information about the number of local basis functions, `n_phi`, how to access global DOFs from the elements, `get_dof()`. This function uses the DOF administration `admin` of the finite element space. We also need information about the boundary type of the local basis functions, `get_bound()`, and for the computation of the values of v at quadrature nodes, we have to extract the local coefficient vector from the global one, `get_v_loc()`. These functions and the number of local basis functions can be accessed via the `bas_fcts` inside the `data->fe_space` structure. The used `admin` is the `admin` structure in `data->fe_space`.

Conversion of the vectors into DOF vectors The input vector `v` is a vector storing the coefficients of the function used for the linearization. It is not a DOF vector, but ALBERT routines for extraction a local coefficient vector need a DOF vector. Thus, we have to "convert" `v` into some DOF vector `dof_v`. This is done by setting the members `fe_space`, `size`, and `vec` in the structure `dof_v` to the used finite element space, `data->fe_space`, the size of the vector, `dim`, and the vector, `v`. In the assignment of the vector we have to use a cast to `(REAL *)` since `v` is a `const REAL *` whereas the member `dof_v.vec` is `REAL *` only. Nevertheless, values of `v` must not be changed.

After this initialization, all ALBERT tools working on DOF vectors can be used. But this vector is *not* linked to the list of DOF vectors of `fe_space->admin` and are *not* administrated by this `admin` (not enlarged during mesh modifications, e.g.)!

In the same way we have to convert `F` to a DOF vector `dof_F` if `F` is not `nil`.

The assemblage of the linearized system If the system matrix has to be assembled, then the DOF matrix `data->DF` is cleared and we check which solver can be used for solving the linearized equation.

If the right hand side has to be assembled, then this vector is initialized with values

$$-\int_\Omega (f + \sigma u_{ext}^4)\varphi_j \, dx.$$

For the assemblage of the element contributions we use the non–recursive mesh traversal routines. On each element we access the local coefficient vector `v_loc`, the global DOFs `dof` and boundary types `bound` of the local basis functions.

Next, we initialize the Jacobian of the barycentric coordinates and compute the values of v at the quadrature node by `uh_at_qp()` (then v^3 can easily be calculated in `c()`). Now, all members of `struct op_info` are initialized, and we compute the element matrices $\boldsymbol{A}_S$ by the function `fill_a()` and $\boldsymbol{M}_S$ by the function `fill_c()`.

These contributions are added to the system matrix if up_DF is not zero. Finally, the right hand side contribution for all non Dirichlet DOFs are computed, and zero Dirichlet boundary values are set for Dirichlet DOFs, if F is not nil.

```
static void update(void *ud, int dim, const REAL *v, int up_DF, REAL *F)
{
  FUNCNAME("update");
  static struct op_info *op_info = nil;
  static const REAL     **(*fill_a)(const EL_INFO *, void *) = nil;
  static void           *a_info = nil;
  static const REAL     **(*fill_c)(const EL_INFO *, void *) = nil;
  static void           *c_info = nil;

  static const DOF_ADMIN *admin = nil;
  static int              n_phi;
  static const REAL   *(*get_v_loc)(const EL *, const DOF_REAL_VEC *, REAL *);
  static const DOF    *(*get_dof)(const EL *, const DOF_ADMIN *, DOF *);
  static const S_CHAR *(*get_bound)(const EL_INFO *, S_CHAR *);

  NEWTON_DATA    *data = ud;
  TRAVERSE_STACK *stack = get_traverse_stack();
  EL_INFO        *el_info;
  FLAGS          fill_flag;
  DOF_REAL_VEC   dof_v = {nil, nil, "v"};
  DOF_REAL_VEC   dof_F = {nil, nil, "F(v)"};

/*--------------------------------------------------------------------------*/
/*  init functions for assembling DF(v) and F(v)                            */
/*--------------------------------------------------------------------------*/

  if (admin != data->fe_space->admin)
  {
    OPERATOR_INFO          o_info2 = {nil}, o_info0 = {nil};
    const EL_MATRIX_INFO   *matrix_info;
    const BAS_FCTS         *bas_fcts = data->fe_space->bas_fcts;
    const QUAD             *quad = get_quadrature(DIM, 2*bas_fcts->degree-2);

    admin = data->fe_space->admin;
    n_phi      = bas_fcts->n_bas_fcts;
    get_dof    = bas_fcts->get_dof_indices;
    get_bound  = bas_fcts->get_bound;
    get_v_loc = bas_fcts->get_real_vec;;

    if (!op_info) op_info = MEM_ALLOC(1, struct op_info);

    o_info2.row_fe_space = o_info2.col_fe_space = data->fe_space;

    o_info2.quad[2]        = quad;
    o_info2.LALt           = LALt;
    o_info2.LALt_pw_const  = true;
    o_info2.LALt_symmetric = true;
    o_info2.user_data      = op_info;
```

```
    matrix_info = fill_matrix_info(&o_info2, nil);
    fill_a = matrix_info->el_matrix_fct;
    a_info = matrix_info->fill_info;

    o_info0.row_fe_space = o_info0.col_fe_space = data->fe_space;

    o_info0.quad[0]        = quad;
    o_info0.c              = c;
    o_info0.c_pw_const     = false;
    o_info0.user_data      = op_info;

    op_info->quad_fast = get_quad_fast(bas_fcts, quad, INIT_PHI);

    matrix_info = fill_matrix_info(&o_info0, nil);
    fill_c = matrix_info->el_matrix_fct;
    c_info = matrix_info->fill_info;
  }

/*--------------------------------------------------------------------------*/
/*  make a DOF vector from input vector v_vec                               */
/*--------------------------------------------------------------------------*/
  dof_v.fe_space = data->fe_space;
  dof_v.size     = dim;
  dof_v.vec      = (REAL *) v; /* cast needed; dof_v.vec isn't const REAL * */
                               /* nevertheless, values are NOT changed      */

/*--------------------------------------------------------------------------*/
/*  make a DOF vector from F, if not nil                                    */
/*--------------------------------------------------------------------------*/
  if (F)
  {
    dof_F.fe_space = data->fe_space;
    dof_F.size     = dim;
    dof_F.vec      = F;
  }

/*--------------------------------------------------------------------------*/
/*  and now assemble DF(v) and/or F(v)                                      */
/*--------------------------------------------------------------------------*/
  op_info->k     = data->k;
  op_info->sigma = data->sigma;

  if (up_DF)
  {
/*--- if v_vec[i] >= 0 for all i => matrix is positive definite (p=1)  ----*/
    data->solver = dof_min(&dof_v) >= 0 ? CG : GMRes;
    clear_dof_matrix(data->DF);
  }

  if (F)
  {
    dof_set(0.0, &dof_F);
    L2scp_fct_bas(data->f, op_info->quad_fast->quad, &dof_F);
```

```
    dof_scal(-1.0, &dof_F);
  }

  fill_flag = CALL_LEAF_EL|FILL_COORDS|FILL_BOUND;
  el_info = traverse_first(stack, data->fe_space->mesh, -1, fill_flag);
  while (el_info)
  {
    const REAL   *v_loc = (*get_v_loc)(el_info->el, &dof_v, nil);
    const DOF    *dof   = (*get_dof)(el_info->el, admin, nil);
    const S_CHAR *bound = (*get_bound)(el_info, nil);
    const REAL   **a_mat, **c_mat;

/*--------------------------------------------------------------------------*/
/* initialization of values used by LALt and c                              */
/*--------------------------------------------------------------------------*/
    op_info->det   = el_grd_lambda(el_info, op_info->Lambda);
    op_info->v_qp = uh_at_qp(op_info->quad_fast, v_loc, nil);

    a_mat = fill_a(el_info, a_info);
    c_mat = fill_c(el_info, c_info);

    if (up_DF)  /*---  add element contribution to matrix DF(v)  ----------*/
    {
/*--------------------------------------------------------------------------*/
/*  add a(phi_i,phi_j) + 4*m(u^3*phi_i,phi_j) to matrix                     */
/*--------------------------------------------------------------------------*/
      add_element_matrix(data->DF, 1.0, n_phi, n_phi, dof, dof, a_mat, bound);
      add_element_matrix(data->DF, 4.0, n_phi, n_phi, dof, dof, c_mat, bound);
    }

    if (F)     /*---  add element contribution to F(v)  ---------------------*/
    {
      int   i, j;
/*--------------------------------------------------------------------------*/
/*  F(v) += a(v, phi_i) + m(v^4, phi_i)                                     */
/*--------------------------------------------------------------------------*/
      for (i = 0; i < n_phi; i++)
      {
        if (bound[i] < DIRICHLET)
        {
          REAL val = 0.0;
          for (j = 0; j < n_phi; j++)
            val += (a_mat[i][j] + c_mat[i][j])*v_loc[j];
          F[dof[i]] += val;
        }
        else
          F[dof[i]] = 0.0;  /*--- zero Dirichlet boundary conditions! ------*/
      }
    }

    el_info = traverse_next(stack, el_info);
  }
```

```
  free_traverse_stack(stack);

  return;
}
```

The linear sub–solver

For the solution of the linearized problem we use the `oem_solve_s()` function, which is also used in the solver for the linear Poisson equation (compare Section 3.1.7). Similar to the `update()` function, we have to convert the right hand side vector `F` and the solution vector `d` to DOF vectors. Information about the system matrix, parameters for the solver are passed by `ud`. The member `data->solver` is initialized in `update()`.

```
static int solve(void *ud, int dim, const REAL *F, REAL *d)
{
  NEWTON_DATA   *data = ud;
  int           iter;
  DOF_REAL_VEC  dof_F = {nil, nil, "F"};
  DOF_REAL_VEC  dof_d = {nil, nil, "d"};

/*--------------------------------------------------------------------------*/
/*  make DOF vectors from F and d                                           */
/*--------------------------------------------------------------------------*/
  dof_F.fe_space = dof_d.fe_space = data->fe_space;
  dof_F.size     = dof_d.size     = dim;
  dof_F.vec      = (REAL *) F; /* cast needed ...  */
  dof_d.vec      = d;

  iter = oem_solve_s(data->DF, &dof_F, nil, &dof_d, data->solver,
                     data->tolerance, data->icon, data->restart,
                     data->max_iter, data->info);

  return(iter);
}
```

The computation of the H^1 semi norm

The H^1 semi norm can easily be calculated by converting the input vector `v` into a DOF vector and then calling the ALBERT routine `H1_norm_uh()` (compare Section 2.10).

```
static REAL norm(void *ud, int dim, const REAL *v)
{
  NEWTON_DATA   *data = ud;
  DOF_REAL_VEC  dof_v = {nil, nil, "v"};

  dof_v.fe_space = data->fe_space;
  dof_v.size     = dim;
  dof_v.vec      = (REAL *) v;  /* cast needed ... */

  return(H1_norm_uh(nil, &dof_v));
}
```

The nonlinear solver

The function `nlsolve()` initializes the structure `NEWTON_DATA` with problem dependent information. Here, we have to allocate a DOF matrix for storing the system matrix (only on the first call), and initialize parameters for the linear sub–solver and problem dependent data (like heat conductivity k, etc.)

The structure `NLS_DATA` is filled with information for the general Newton solver (routines `update()`, `solve()`, and `norm()`). All problem dependent routines use the same structure `NEWTON_DATA` for problem dependent information. For getting access to the definition of `NLS_DATA` and prototypes for the Newton solvers, we have to include the `nls.h` header file.

The dimension of the discrete equation is

```
dim = u0->fe_space->admin->size_used;
```

where `u0` is a pointer to a DOF vector storing the initial guess. Note, that after the `dof_compress()` in the `build()` routine, `dim` holds the true dimension of the discrete equation. Without a `dof_compress()` there may be holes in the DOF vector, and `u0->fe_space->admin->size_used` bigger than the last *used* index, and again `dim` is the dimension of the discrete equation for the Newton solver. The ALBERT routines do not operate on unused indices, whereas the Newton solver do operate on unused indices too, because it does not know about used and unused indices. In this situation, all unused DOFs have to be cleared for the initial solution `u0` by

```
FOR_ALL_FREE_DOFS(u0->fe_space->admin, u0->vec[dof] = 0.0);
```

The same applies to the vector storing the right hand side in `update()`. The `dof_set()` function only initializes used indices.

Finally, we reallocate the workspace used by the Newton solvers (compare Section 2.15.6) and start the Newton method.

```
#include <nls.h>

int nlsolve(DOF_REAL_VEC *u0, REAL k, REAL sigma, REAL (*f)(const REAL_D))
{
  FUNCNAME("nlsolve");
  static NEWTON_DATA  data = {nil, 0, 0, nil, nil, CG, 0, 500, 2, 10, 1};
  static NLS_DATA     nls_data = {nil};
  int                 iter, dim = u0->fe_space->admin->size_used;

  if (!data.fe_space)
  {
/*--------------------------------------------------------------------------*/
/*--  init parameters for newton  -----------------------------------------*/
/*--------------------------------------------------------------------------*/
    nls_data.update      = update;
    nls_data.update_data = &data;
    nls_data.solve       = solve;
    nls_data.solve_data  = &data;
    nls_data.norm        = norm;
    nls_data.norm_data   = &data;

    nls_data.tolerance = 1.e-4;
    GET_PARAMETER(1, "newton tolerance", "%e", &nls_data.tolerance);
    nls_data.max_iter = 50;
    GET_PARAMETER(1, "newton max. iter", "%d", &nls_data.max_iter);
```

```
    nls_data.info = 8;
    GET_PARAMETER(1, "newton info", "%d", &nls_data.info);
    nls_data.restart = 0;
    GET_PARAMETER(1, "newton restart", "%d", &nls_data.restart);

/*--------------------------------------------------------------------------*/
/*--  init data for update and solve  -------------------------------------*/
/*--------------------------------------------------------------------------*/

    data.fe_space = u0->fe_space;
    data.DF       = get_dof_matrix("DF(v)", u0->fe_space);

    data.tolerance = 1.e-2*nls_data.tolerance;
    GET_PARAMETER(1, "linear solver tolerance", "%f", &data.tolerance);
    GET_PARAMETER(1, "linear solver max iteration", "%d", &data.max_iter);
    GET_PARAMETER(1, "linear solver info", "%d", &data.info);
    GET_PARAMETER(1, "linear solver precon", "%d", &data.icon);
    GET_PARAMETER(1, "linear solver restart", "%d", &data.restart);
  }
  TEST_EXIT(data.fe_space == u0->fe_space)("can’t change f.e. spaces\n");

/*--------------------------------------------------------------------------*/
/*--  init problem dependent parameters  -----------------------------------*/
/*--------------------------------------------------------------------------*/
  data.k     = k;
  data.sigma = sigma;
  data.f     = f;

/*--------------------------------------------------------------------------*/
/*--  enlarge workspace used by newton(_fs), and solve by Newton  ----------*/
/*--------------------------------------------------------------------------*/
  if (nls_data.restart)
  {
    nls_data.ws = REALLOC_WORKSPACE(nls_data.ws, 4*dim*sizeof(REAL));
    iter = nls_newton_fs(&nls_data, dim, u0->vec);
  }
  else
  {
    nls_data.ws = REALLOC_WORKSPACE(nls_data.ws, 2*dim*sizeof(REAL));
    iter = nls_newton(&nls_data, dim, u0->vec);
  }

  return(iter);
}
```

3.2.11 The parameter file for the nonlinear Poisson equation

The following parameter file `INIT/nonlin.dat` is read by `main()`.

```
problem number:       2
global refinements:   2
polynomial degree:    2
```

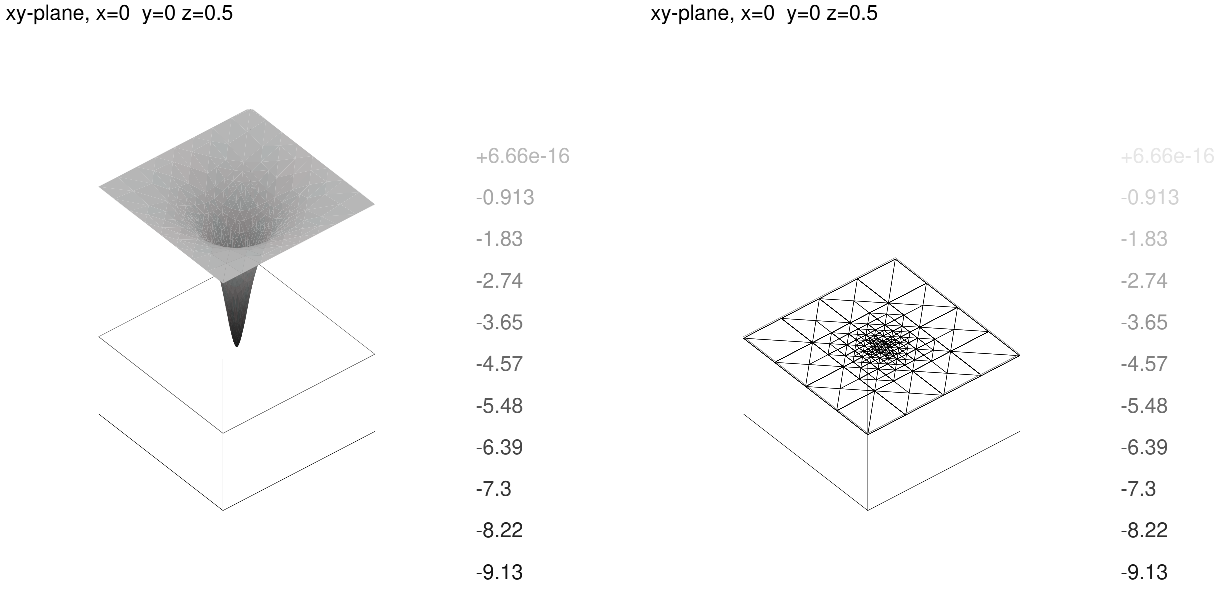


Figure 3.2: Graph of the unstable solution with corresponding mesh of the nonlinear Poisson problem in 3d on the clipping plane $z = 0.5$. The pictures were produced by the gltools.

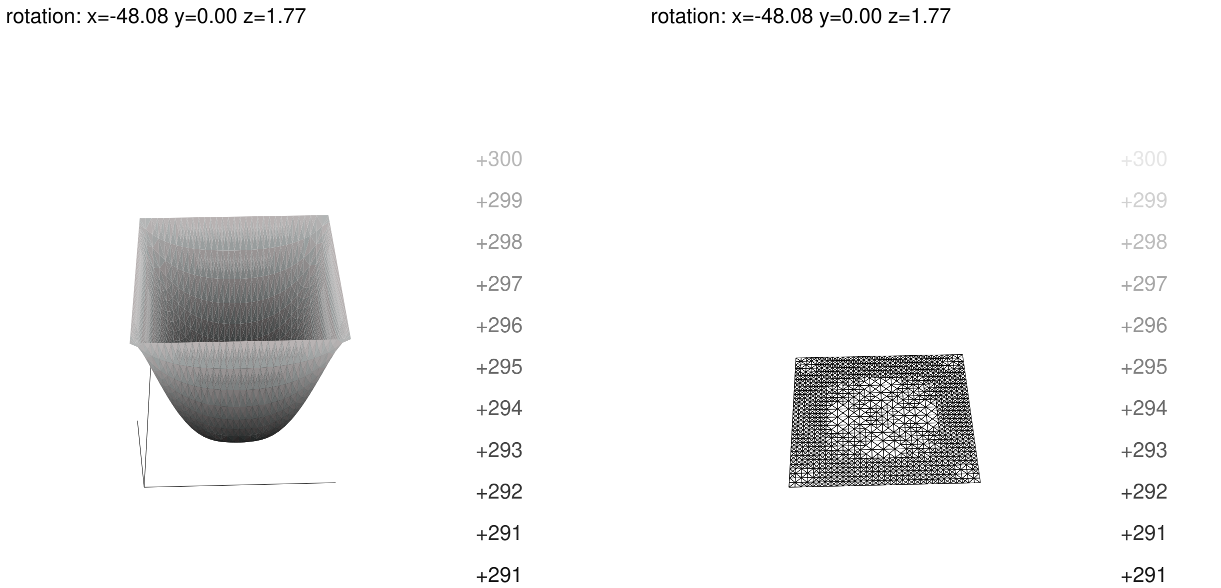


Figure 3.3: Graph of the solution to the physical problem with corresponding mesh of the nonlinear Poisson problem in 2d. The pictures were produced by the gltools.

```
U0:     -5.0          % height of initial guess for Problem 1

graphic windows:      300 300 300  % size of graphic windows

newton tolerance: 1.e-6     % tolerance for Newton
newton max. iter: 50        % maximal number of iterations of Newton
newton info:      6         % information level of Newton
newton restart:   10        % number of iterations for step size control

linear solver max iteration:  1000
linear solver restart:        10  %  only used for GMRES
linear solver tolerance:      1.e-8
linear solver info:           0
linear solver precon:         1   % 0: no precon 1: diag precon

error norm:           1   % 1: H1_NORM, 2: L2_NORM
estimator C0:         0.1 % constant of element residual
estimator C1:         0.1 % constant of jump residual
estimator C2:         0.0 % constant of coarsening estimate

adapt->strategy:        2   % 0: no adaption 1: GR 2: MS 3: ES 4:GERS
adapt->tolerance:       1.e-8
adapt->MS_gamma:        0.9
adapt->max_iteration:   8
adapt->info:            4

WAIT: 1
```

The unstable solution for Problem 1 in 3d is shown in Figure 3.2 and the solution to the physical problem is shown in Figure 3.3.

3.3 Heat equation

In this section we describe a model implementation for the (linear) heat equation

$$
\begin{aligned}
\partial_t u - \Delta u &= f && \text{in } \Omega \subset \mathbb{R}^d \times (0,T),\\
u &= g && \text{on } \partial\Omega \times (0,T),\\
u &= u_0 && \text{on } \Omega \times \{0\}.
\end{aligned}
$$

We describe here only differences to the implementation of the linear Poisson problem. For common (or similar) routines we refer to Section 3.1.

3.3.1 Global variables

Additionally to the finite element space `fe_space`, the matrix `matrix` and the vectors `u_h` and `f_h`, we need a vector for storage of the solution U_n from the last time step. This one is implemented as a global variable, too. All these global variables are initialized in `main()`.

```
static DOF_REAL_VEC   *u_old = nil;                  /* initialized by main() */
```

A global pointer to the `ADAPT_INSTAT` structure is used for access in the `build()` and `estimate()` routines, see below.

```
static ADAPT_INSTAT   *adapt_instat = nil;         /* initialized by main() */
```

3.3.2 Functions for leaf data

For time dependent problems, mesh adaption usually also includes coarsening of previously (for smaller t) refined parts of the mesh. For storage of local coarsening error estimates, the leaf data structure is enlarged by a second `REAL`. Functions `rw_el_estc()` and `get_el_estc()` are provided for access to that storage location.

```
struct heat_leaf_data
{
  REAL estimate;            /*  one real for the element indicator         */
  REAL est_c;               /*  one real for the coarsening indicator      */
};

static REAL *rw_el_estc(EL *el)
{
  if (IS_LEAF_EL(el))
    return(&((struct heat_leaf_data *)LEAF_DATA(el))->est_c);
  else
    return(nil);
}

static REAL get_el_estc(EL *el)
{
  if (IS_LEAF_EL(el))
    return(((struct heat_leaf_data *)LEAF_DATA(el))->est_c);
  else
    return(0.0);
}
```

3.3.3 Data of the differential equation

Data for the heat equation are the initial values u_0, right hand side f, and boundary values g. When the true solution u is known, it can be used for testing the error between discrete and exact solution.

The sample problem is defined such that the exact solution is

$$u(x,t) = \sin(\pi t)e^{-10|x|^2} \quad \text{on } (0,1)^d \times [0,1].$$

All library subroutines which evaluate a given data function (for integration, e.g.) are defined for space dependent functions only and do not know about a time variable. Thus, such a 'simple' space dependent function $f_{\text{space}}(x)$ has to be derived from a space–time dependent function $f(x,t)$. We do this by keeping the time in a global variable, and setting

$$f_{\text{space}}(x) := f(x,t).$$

```
static REAL eval_time_u = 0.0;
static REAL u(const REAL_D x)
{
```

```
  return(sin(M_PI*eval_time_u)*exp(-10.0*SCP_DOW(x,x)));
}

static REAL u0(const REAL_D x)
{
  eval_time_u = adapt_instat->start_time;
  return(u(x));
}

static REAL eval_time_g = 0.0;
static REAL g(const REAL_D x)              /* boundary values, not optional */
{
  eval_time_u = eval_time_g;
  return(u(x));
}

static REAL eval_time_f = 0.0;
static REAL f(const REAL_D x)              /* u_t - Delta u, not optional   */
{
  REAL  r2 = SCP_DOW(x,x), ux  = sin(M_PI*eval_time_f)*exp(-10.0*r2);
  REAL  ut = M_PI*cos(M_PI*eval_time_f)*exp(-10.0*r2);
  return(ut - (400.0*r2 - 20.0*DIM)*ux);
}
```

As indicated, the times for evaluation of boundary data and right hand side may be chosen independent of each other. This will be used in the time discretization scheme presented in the next section.

3.3.4 Time discretization

The model implementation uses a variable time discretization scheme. Initial data is interpolated on the initial mesh,

$$U_0 = I_0 u_0.$$

For $\theta \in [0,1]$, the solution $U_{n+1} \approx u(\cdot, t_{n+1})$ is given by $U_{n+1} \in I_{n+1}g(\cdot, t_{n+1}) + \mathring{X}_{n+1}$ such that

$$\begin{aligned} \frac{1}{\tau_{n+1}}(U_{n+1}, \Phi) + \theta(\nabla U_{n+1}, \nabla\Phi) = &\frac{1}{\tau_{n+1}}(I_{n+1}U_n, \Phi) + (1-\theta)(\nabla I_{n+1}U_n, \nabla\Phi) \\ &+ (f(\cdot, t_n + \theta\tau_{n+1}), \Phi) \qquad \text{for all } \Phi \in \mathring{X}_{n+1}. \end{aligned} \tag{3.5}$$

For $\theta = 0$, this is the forward (explicit) Euler scheme, for $\theta = 1$ the backward (implicit) Euler scheme. For $\theta = 0.5$, this is the Cranck–Nicholson scheme, which is of second order in time. For $\theta \in [0.5, 1.0]$, the scheme is unconditionally stable, while for $\theta < 0.5$ stability is only guaranteed if the time step size is small enough.

3.3.5 Initial data interpolation

Initial data u_0 is just interpolated on the initial mesh, thus the `solve()` entry in `adapt_instat->adapt_initial` will point to a routine `interpol_u0()` which implements this by the library interpolation routine. No `build()` routine is needed by the initial mesh adaption procedure.

```
static void interpol_u0(MESH *mesh)
{
  dof_compress(mesh);
  interpol(u0, u_h);

  return;
}
```

3.3.6 The assemblage of the discrete system

Using a matrix notation, the discrete problem (3.5) can be written as

$$\Big(\frac{1}{\tau_{n+1}}\boldsymbol{M} + \theta\boldsymbol{A}\Big)\boldsymbol{U}_{n+1} = \Big(\frac{1}{\tau_{n+1}}\boldsymbol{M} + (1-\theta)\boldsymbol{A}\Big)\boldsymbol{U}_n + \boldsymbol{F}_{n+1}.$$

Here, $\boldsymbol{M} = (\Phi_i, \Phi_j)$ (up to Dirichlet boundary nodes) denotes the mass matrix and $\boldsymbol{A} = (\nabla\Phi_i, \nabla\Phi_j)$ the stiffness matrix. The system matrix on the left hand side is (for $\theta \neq 0.5$) not the same as the one applied to the old solution on the right hand side, but both can be easily evaluated when both $\boldsymbol{M}$ and $\boldsymbol{A}$ are known. To calculate the product with U_n, it is sufficient to know (and sum) the local contributions from single elements.

A simple application of the matrix assemblage routines of Section 2.12.2 gives no access to the matrix during assemblage, thus an additional calculation would be needed for the right hand side. Thus, for efficiency reasons, we use the more general techniques of Section 2.12.1, which allow for a combined assemblage of the system matrix and the right hand side.

The subroutine `assemble()` generates both the system matrix and the right hand side. The mesh elements are visited via the non-recursive mesh traversal routines. On every leaf element, both the element mass matrix `c_mat` and the element stiffness matrix `a_mat` are calculated using the `el_matrix_fct()` provided by `fill_matrix_info()`. For this purpose, two different operators (the mass and stiffness operators) are defined and applied on each element. The stiffness operator uses the same `LALt()` function for the second order term as described in Section 3.1.6; the mass operator implements only the constant zero order coefficient $c = 1/\tau_{n+1}$, which is passed in `struct op_info` and evaluated in the function `c()`. The L^2 product with the right hand side f and the handling of Dirichlet boundary values are included in the single mesh traversal, too.

```
struct op_info
{
  REAL_D  Lambda[DIM+1];    /*  the gradient of the barycentric coordinates */
  REAL    det;              /*  |det D F_S|                                 */
  REAL    tau_1;            /*  1/timestepsize                              */
};

static REAL c(const EL_INFO *el_info, const QUAD *quad, int iq, void *ud)
{
  struct op_info *info = ud;

  return(info->tau_1*info->det);
}

static void assemble(DOF_REAL_VEC *u_old, DOF_MATRIX *matrix, DOF_REAL_VEC *fh,
                     DOF_REAL_VEC *u_h, REAL theta, REAL tau,
                     REAL (*f)(const REAL_D), REAL (*g)(const REAL_D))
```

```
{
  FUNCNAME("assemble");
  static struct op_info *op_info = nil;
  static const REAL     **(*fill_a)(const EL_INFO *, void *) = nil;
  static void           *a_info = nil;
  static const REAL     **(*fill_c)(const EL_INFO *, void *) = nil;
  static void           *c_info = nil;

  static const DOF_ADMIN *admin = nil;
  static int             n;
  static const REAL   *(*get_u_loc)(const EL *, const DOF_REAL_VEC *, REAL *);
  static const DOF    *(*get_dof)(const EL *, const DOF_ADMIN *, DOF *);
  static const S_CHAR *(*get_bound)(const EL_INFO *, S_CHAR *);

  TRAVERSE_STACK *stack = get_traverse_stack();
  EL_INFO        *el_info;
  FLAGS          fill_flag;
  const REAL     **a_mat, **c_mat;
  REAL           *f_vec;
  const QUAD     *quad;
  int            i, j;

  quad = get_quadrature(DIM, 2*u_h->fe_space->bas_fcts->degree);

/****************************************************************************/
/*  init functions for matrix assembling                                    */
/****************************************************************************/

  if (admin != u_h->fe_space->admin)
  {
    OPERATOR_INFO          o_info2 = {nil}, o_info0 = {nil};
    const EL_MATRIX_INFO   *matrix_info;
    const BAS_FCTS         *bas_fcts = u_h->fe_space->bas_fcts;

    admin      = u_h->fe_space->admin;

    n = bas_fcts->n_bas_fcts;
    get_dof    = bas_fcts->get_dof_indices;
    get_bound  = bas_fcts->get_bound;
    get_u_loc  = bas_fcts->get_real_vec;

    if (!op_info)
      op_info = MEM_ALLOC(1, struct op_info);

    o_info2.row_fe_space = o_info2.col_fe_space = u_h->fe_space;

    o_info2.quad[2]        = quad;
    o_info2.LALt           = LALt;
    o_info2.LALt_pw_const  = true;
    o_info2.LALt_symmetric = true;
    o_info2.user_data      = op_info;

    matrix_info = fill_matrix_info(&o_info2, nil);
```

```
    fill_a = matrix_info->el_matrix_fct;
    a_info = matrix_info->fill_info;

    o_info0.row_fe_space = o_info0.col_fe_space = u_h->fe_space;

    o_info0.quad[0]        = quad;
    o_info0.c              = c;
    o_info0.c_pw_const     = true;
    o_info0.user_data      = op_info;

    matrix_info = fill_matrix_info(&o_info0, nil);
    fill_c = matrix_info->el_matrix_fct;
    c_info = matrix_info->fill_info;
  }

  op_info->tau_1 = 1.0/tau;

/****************************************************************************/
/*  and now assemble the matrix and right hand side                         */
/****************************************************************************/

  clear_dof_matrix(matrix);
  dof_set(0.0, fh);
  f_vec = fh->vec;

  fill_flag = CALL_LEAF_EL|FILL_COORDS|FILL_BOUND;
  el_info = traverse_first(stack, u_h->fe_space->mesh, -1, fill_flag);
  while (el_info)
  {
    const REAL   *u_old_loc = (*get_u_loc)(el_info->el, u_old, nil);
    const DOF    *dof       = (*get_dof)(el_info->el, admin, nil);
    const S_CHAR *bound     = (*get_bound)(el_info, nil);

/****************************************************************************/
/* initialization of values used by LALt and c                              */
/****************************************************************************/
    op_info->det   = el_grd_lambda(el_info, op_info->Lambda);

    a_mat = fill_a(el_info, a_info);
    c_mat = fill_c(el_info, c_info);

/****************************************************************************/
/*  add theta*a(psi_i,psi_j) + 1/tau*m(4*u^3*psi_i,psi_j)                   */
/****************************************************************************/

    if (theta) add_element_matrix(matrix, theta, n, n, dof, dof, a_mat, bound);
    add_element_matrix(matrix, 1.0, n, n, dof, dof, c_mat, bound);

/****************************************************************************/
/*  f += -(1-theta)*a(u_old,psi_i) + 1/tau*m(u_old,psi_i)                   */
/****************************************************************************/

    if (1.0 - theta)
```

```
    {
      REAL theta1 = 1.0 - theta;
      for (i = 0; i < n; i++)
      {
        if (bound[i] < DIRICHLET)
        {
          REAL val = 0.0;
          for (j = 0; j < n; j++)
            val += (c_mat[i][j] - theta1*a_mat[i][j]) * u_old_loc[j];
          f_vec[dof[i]] += val;
        }
      }
    }
    else
    {
      for (i = 0; i < n; i++)
      {
        if (bound[i] < DIRICHLET)
        {
          REAL val = 0.0;
          for (j = 0; j < n; j++)
            val += c_mat[i][j]*u_old_loc[j];
          f_vec[dof[i]] += val;
        }
      }
    }
    el_info = traverse_next(stack, el_info);
  }

  free_traverse_stack(stack);

  L2scp_fct_bas(f, quad, fh);
  dirichlet_bound(g, fh, u_h, nil);

  return;
}
```

The `build()` routine for one time step of the heat equation just calls the `assemble()` routine described above, with evaluation times for right hand side and boundary values set according to the time discretization parameter `theta`.

```
static REAL     theta = 0.5;
static void build(MESH *mesh, U_CHAR flag)
{
  FUNCNAME("build");

  dof_compress(mesh);

  INFO(adapt_instat->adapt_space->info, 3)
    ("%d DOFs for %s\n", fe_space->admin->size_used, fe_space->name);

  eval_time_f = adapt_instat->time - (1-theta)*adapt_instat->timestep;
  eval_time_g = adapt_instat->time;
  assemble(u_old, matrix, f_h, u_h, theta, tau, f, g);
```

```
  return;
}
```

The resulting linear system is solved by calling the `oem_solve_s()` library routine. This is done via the `solve()` subroutine described in Section 3.1.7.

3.3.7 Error estimation

The initial error $\|U_0 - u_0\|_{L^2(\Omega)}$ is calculated exactly (up to quadrature error) by a call to `L2_err()`. Local error contributions are written via `rw_el_est()` to the `estimate` value in `struct heat_leaf_data`. The `err_max` and `err_sum` of the `ADAPT_STAT` structure (which will be `adapt_instat->adapt_initial`, see below) are set accordingly.

```
static REAL est_initial(MESH *mesh, ADAPT_STAT *adapt)
{
  adapt->err_sum = L2_err(u, u_h, nil, 0, rw_el_est, &adapt->err_max);
  return(adapt->err_sum);
}
```

In each time step, error estimation is done by the library routine `heat_est()` which generates both time and space discretization indicators, compare Section 3.3.7. Local element indicators are stored to the `estimate` or `est_c` entries of `struct heat_leaf_data` via `rw_el_est()` and `rw_el_estc()`. The `err_max` and `err_sum` entries of `adapt->adapt_space` are set accordingly. The temporal error indicator is the return value by `heat_est()` and is stored in a global variable for later access by `get_time_est()`. For known exact solution, the true error $\|u(\cdot, t_{n+1}) - U_{n+1}\|_{L^2(\Omega)}$ is calculated additionally. After each time step (in `close_timestep()`), maxima (over time steps) of estimate and error are calculated and compared.

```
static REAL time_est = 0.0;

static REAL get_time_est(MESH *mesh, ADAPT_INSTAT *adapt)
{
  return(time_est);
}

static REAL err_L2 = 0.0;

static FLAGS r_flag = 0;  /* = (INIT_UH | INIT_GRD_UH),  if needed by r()   */

static REAL r(const EL_INFO *el_info, const QUAD *quad, int iq, REAL t,
              REAL uh_iq, const REAL_D grd_uh_iq)
{
  REAL_D      x;
  coord_to_world(el_info, quad->lambda[iq], x);
  eval_time_f = t;
  return(-f(x));
}

static REAL estimate(MESH *mesh, ADAPT_STAT *adapt)
{
  FUNCNAME("estimate");
  static int     degree;
  static REAL    C[4] = {-1.0, 1.0, 1.0, 1.0};
```

```
  static const REAL_DD A = {{1.0,0.0}, {0.0,1.0}};

  eval_time_u = adapt_instat->time;

  if (C[0] < 0)
  {
    C[0] = 1.0;
    GET_PARAMETER(1, "estimator C0", "%f", &C[0]);
    GET_PARAMETER(1, "estimator C1", "%f", &C[1]);
    GET_PARAMETER(1, "estimator C2", "%f", &C[2]);
    GET_PARAMETER(1, "estimator C3", "%f", &C[3]);
  }

  degree = 2*u_h->fe_space->bas_fcts->degree;
  time_est = heat_est(u_h, adapt_instat, rw_el_est, rw_el_estc,
                      degree, C, u_old, A, r, r_flag);
  err_L2 = L2_err(u, u_h, nil, 0, nil, nil);

  INFO(adapt_instat->info, 2)
    ("time=%.4le  ||u-uh||L2 = %.8le\n", adapt_instat->time, err_L2);
  INFO(adapt_instat->info, 2)
    ("time=%.4le  estimate   = %.8le, max^2=%.3le\n", adapt_instat->time,
     adapt_instat->adapt_space->err_sum, adapt_instat->adapt_space->err_max);

  return(adapt_instat->adapt_space->err_sum);
}
```

3.3.8 Time steps

As described in Section 2.13.3, time dependent problems are calculated step by step in single time steps. Each time step is initialized by a call to `init_timestep()`, where the discrete solution `u_h` from the old time step (or from interpolation of initial data) is copied to `u_old`.

```
static void init_timestep(MESH *mesh, ADAPT_INSTAT *adapt)
{
  dof_copy(u_h, u_old);
  return;
}
```

One or several iterations with time step adjustment, solution of the discrete time step problem, and mesh adaption may follow. The problem dependent routines (`build()`, `solve()`, `estimate()`) were already described above. As time and time step size information are directly accessed through the global `adapt_instat` pointer, the `set_time()` routine which is usually thought to give such information, is not used here. When a time step is finished, the `close_timestep()` routine calculates the global (in time) error and error estimator by taking the maximum over the local errors (estimators). Current error maxima are printed if output is requested by `adapt_instat->info`. Finally, a graphic output of the solution and the mesh is generated via the `graphics()` routine described in Section 3.1.1.

```
static REAL err_Max = 0.0;                    /* max space-time error    */
static REAL est_Max = 0.0;                    /* max space-time estimate */

static void close_timestep(MESH *mesh, ADAPT_INSTAT *adapt)
```

```
{
  FUNCNAME("close_timestep");

  err_Max = MAX(err_Max, err_L2);

  est_Max = MAX(est_Max, adapt_instat->adapt_space->err_sum + time_est);

  INFO(adapt_instat->info, 3)
    ("time=%.4le: time_est = %.4le, timestep = %.4le\n",
     adapt_instat->time, time_est, adapt_instat->timestep);
  INFO(adapt_instat->info, 3)
    ("time=%.4le: Max error = %.8le\n", adapt_instat->time, err_Max);
  INFO(adapt_instat->info, 3)
    ("time=%.4le: Max est.  = %.8le, ratio=%.3le\n", adapt_instat->time,
     est_Max, err_Max/(est_Max+1.0E-20));

  graphics(mesh, u_h, get_el_est);
  return;
}
```

3.3.9 The main program for the heat equation

The main purposes of the main program are to read program parameters, generate a mesh and finite element space, generate the DOF matrix and vectors, and allocate and fill the parameter structure `ADAPT_INSTAT` for the adaptive method for time dependent problems. The (initial) time step size, read from parameter file, is reduced when an initial global mesh refinement is performed. This reduction is automatically adapted to the order of time discretization (2nd order when $\theta = 0.5$, 1st order otherwise). The complete numerical simulation is then done by a call to `adapt_method_instat()`.

```
/****************************************************************************/
/*  main program                                                            */
/****************************************************************************/

int main(int argc, char **argv)
{
  FUNCNAME("main");
  MESH   *mesh;
  int    n_refine = 0, k;
  char   filename[100];
  REAL   fac = 1.0;

/****************************************************************************/
/*  first of all, init parameters of the init file                          */
/****************************************************************************/

  init_parameters(0, "INIT/heat.dat");
  for (k = 1; k+1 < argc; k += 2)
    ADD_PARAMETER(4, argv[k], argv[k+1]);

/****************************************************************************/
/*  get a mesh, and read the macro triangulation from file                  */
/****************************************************************************/
```

```
  mesh = GET_MESH("ALBERT mesh", init_dof_admin, init_leaf_data);
  mesh->preserve_coarse_dofs = 0;
  GET_PARAMETER(1, "macro file name", "%s", filename);
  read_macro(mesh, filename, nil);
  GET_PARAMETER(1, "global refinements", "%d", &n_refine);
  global_refine(mesh, n_refine*DIM);
  graphics(mesh, nil, nil);
  GET_PARAMETER(4, "theta", "%e", &theta);
  if (theta < 0.5)
  {
    WARNING("You are using an explicit scheme\n");
    WARNING("Use a sufficiently small time step size!!!\n");
    fac = 1.0e-3;
  }
  MSG("theta = %lf\n", theta);

  matrix = get_dof_matrix("A", fe_space);
  f_h    = get_dof_real_vec("f_h", fe_space);
  u_h    = get_dof_real_vec("u_h", fe_space);
  u_h->refine_interpol = fe_space->bas_fcts->real_refine_inter;
  u_h->coarse_restrict = fe_space->bas_fcts->real_coarse_inter;
  u_old  = get_dof_real_vec("u_old", fe_space);
  u_old->refine_interpol = fe_space->bas_fcts->real_refine_inter;
  u_old->coarse_restrict = fe_space->bas_fcts->real_coarse_inter;
  dof_set(0.0, u_h);      /*  initialize u_h  !                         */

/****************************************************************************/
/*  init adapt structure and start adaptive method                          */
/****************************************************************************/

  adapt_instat = get_adapt_instat("heat", "adapt", 1, adapt_instat);
  if (theta == 0.5)
    adapt_instat->timestep *= fac*pow(2, -n_refine);
  else
    adapt_instat->timestep *= fac*pow(2, -n_refine*DIM);

  adapt_instat->start_time = 0.0;
  adapt_instat->end_time = 1.0;

  adapt_instat->adapt_initial->estimate = est_initial;
  adapt_instat->adapt_initial->solve = interpol_u0;
  adapt_instat->adapt_initial->get_el_est = get_el_est;

  adapt_instat->adapt_space->get_el_est   = get_el_est;
  adapt_instat->adapt_space->get_el_estc  = get_el_estc;
  adapt_instat->adapt_space->estimate = estimate;
  adapt_instat->adapt_space->build_after_coarsen = build;
  adapt_instat->adapt_space->solve = solve;

  adapt_instat->init_timestep = init_timestep;
  adapt_instat->get_time_est = get_time_est;
  adapt_instat->close_timestep = close_timestep;
```

```
  adapt_method_instat(mesh, adapt_instat);

  MSG("space-time Max error:  %.8le\n", err_Max);
  MSG("space-time estimate:   %.8le, ratio=%.3le\n",
      est_Max, err_Max/(est_Max+1.0E-20));

  WAIT_REALLY;
  return(0);
}
```

3.3.10 The parameter file for the heat equation

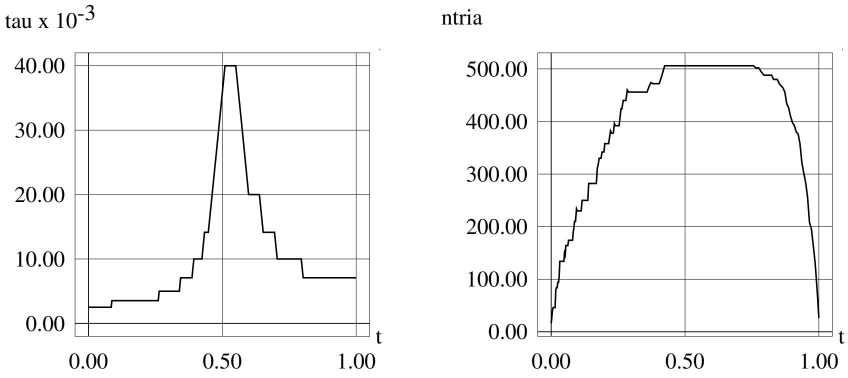


Figure 3.4: Time step size (left) and number of 2d mesh elements (right) over time.

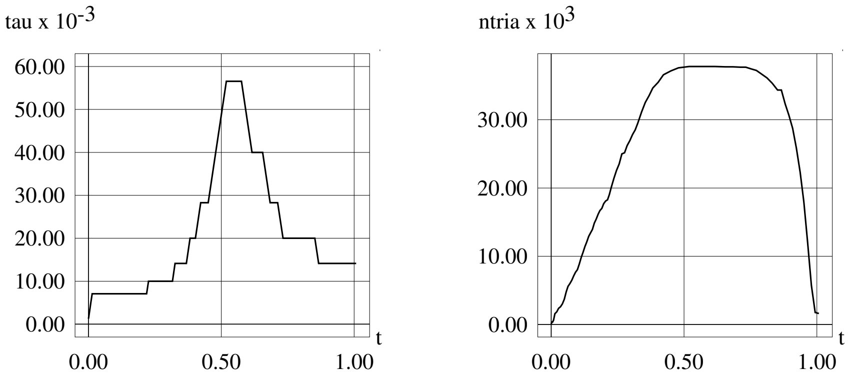


Figure 3.5: Time step size (left) and number of 3d mesh elements (right) over time.

Figures 3.4 and 3.5 show the variation of time step sizes and number of mesh elements over time, automatically generated by the adaptive method in two and three space dimensions, with prescribed error tolerance 0.01. Parameters for the program were given by the following data in `INIT/heat.dat`.

```
macro file name:       Macro/macro.stand
global refinements:    1
polynomial degree:     2
```

```
graphic windows:       300 300

solver:                2          % 1: BICGSTAB 2: CG 3: GMRES 4: ODIR 5: ORES
solver max iteration:  1000
solver restart:        10         %  only used for GMRES
solver tolerance:      1.e-8
solver info:           2
solver precon:         1          % 0: no precon 1: diag precon

adapt->tolerance:               0.01
adapt->timestep:                0.01
adapt->rel_initial_error:       0.5
adapt->rel_space_error:         0.5
adapt->rel_time_error:          0.5
adapt->strategy:                1   % 0=explicit, 1=implicit
adapt->max_iteration:           1
adapt->info:                    3

adapt->initial->strategy:       2    % 0=none, 1=GR, 2=MS, 3=ES, 4=GERS
adapt->initial->max_iteration: 10
adapt->initial->info:           2

adapt->space->strategy:         3    % 0=none, 1=GR, 2=MS, 3=ES, 4=GERS
adapt->space->ES_theta:         0.9
adapt->space->ES_theta_c:       0.05
adapt->space->max_iteration:    2
adapt->space->coarsen_allowed:  1   % 0|1
adapt->space->info:             2

estimator C0:                   1.0
estimator C1:                   1.0
estimator C2:                   1.0
estimator C3:                   1.0

theta:                          1.0
WAIT:                           0
```

3.4 File organization, ALBERT libraries, and Makefiles

3.4.1 File organization

Using the ALBERT library, the dimension enters in an application only by the parameters `DIM` and `DIM_OF_WORLD`. Thus, the code is usually the same for 2d and 3d simulations. Nevertheless, the object files do depend on the dimension, since `DIM` and `DIM_OF_WORLD` are symbolic constants (which define the length of vectors, e.g.). Hence, all object files have to be rebuilt, when changing the dimension. To make sure that this is done automatically we use the following file organization.

For an application we use three sub–directories

```
./2d        ./3d        ./Common
```

The directory `Common` contains all source files and header files that do not (or only slightly) depend on the dimension. All files from `Common` are the linked to the `2d` *and* `3d` directories (by `ln -s`

`../Common/* .`, executed in `./2d` and `./3d`). These directories may also hold some source files which strongly depend on the dimension (usually, there are none of these files!). In the 2[3]d directory the 2[3]d macro triangulations, init files for the 2[3]d simulations, the 2[3]d-Makefile, and the object files for 2[3]d are located. The source files are still the same for 2d and 3d (those located in Common) and changes in one directory will be detected by the Makefile in the other directory, too. On the other hand, the object files are different for 2d and 3d (they are not located in `Common`!).

3.4.2 ALBERT libraries

The ALBERT libraries not only depend on the dimension, but on the values of the symbolic constants `NEIGH_IN_EL` and `EL_INDEX` too. Additionaly, the object files in the library may depend on some machine defined optimization flag (like the `-mips2` or `-mips4` flag for SGI). For the organization of the ALBERT libraries we use the follwoing convention: the name of an ALBERT library is

```
libALBERT$(DIM)$(DIM_OF_WORLD)$(NEIGH_IN_EL)$(EL_INDEX)$(DEBUG)$(MOFLAG).a
```

where `DEBUG` is a variable which may be `0` or `1`, referring to a library with or without debugging information, and `MOFLAG` is a string storing the machine optimization flag or an empty string.

The matching ALBERT library is chosen automatically by the Makefile described below.

3.4.3 Makefiles

The Makefile used for the compilation of the above described problems in 2d is the following. For the three applications `ellipt`, `nonlin`, and `heat` the corresponding object files are defined in the last part of the file. A default target for the Makefile can be defined by the variable `DEFAULT`.

```
##########################################################################
#   Sample ALBERT Makefile for 2d                                        #
##########################################################################

DEFAULT = ellipt

# delete line, if ALBERT_LIB_PATH is an environment variable
ALBERT_LIB_PATH = /usr/people/kunibert/ALBERT-1.0/lib
include $(ALBERT_LIB_PATH)/Makefile

# uncomment line, for using GRAPE library
#GRAPE_LIB = /usr/people/kunibert/ALBERT-1.0/lib

# MOFLAG is machine dependent OFLAG -mips2-4 on SGIs, e.g.
#MOFLAG = -mips4
# default compile flags
#CFLAGS = -xansi
# default optimization flags
#OFLAG = -O3
# uncomment line for debugging
#OFLAG = -g
# uncomment line for using the debug library
#DEBUG = 1

##########################################################################
```

```
#   DFLAGS: DIM, DIM_OF_WORLD, NEIGH_IN_EL, EL_INDEX                   #
########################################################################
DIM = 2
DIM_OF_WORLD = 2
EL_INDEX = 1
NEIGH_IN_EL = 0

########################################################################
#   and now the user's files                                           #
########################################################################

ELLIPT_OFILES = ellipt.o graphics.o

ellipt: $(ELLIPT_OFILES)
	$(LD) $(MOFLAG) -o $@ $(ELLIPT_OFILES) $(LIBS)

NONLIN_OFILES = nonlin.o nlprob.o nlsolve.o graphics.o
$(NONLIN_OFILES): nonlin.h

nonlin: $(NONLIN_OFILES)
	$(LD) $(MOFLAG) -o $@ $(NONLIN_OFILES) $(LIBS)

HEAT_OFILES = heat.o graphics.o

heat: $(HEAT_OFILES)
	$(LD) $(MOFLAG) -o $@ $(HEAT_OFILES) $(LIBS)
```

The Makefile for 3d simulations only differs in the following lines

```
########################################################################
#   DFLAGS: DIM, DIM_OF_WORLD, NEIGH_IN_EL, EL_INDEX                   #
########################################################################
DIM = 3
DIM_OF_WORLD = 3
EL_INDEX = 1
NEIGH_IN_EL = 0
```

The most important variable inside the Makefile is `ALBERT_LIB_PATH` which contains the path, where the ALBERT libraries are installed. This variable may also be defined as an environment variable.

The variables `DIM`, `DIM_OF_WORLD`, `EL_INDEX`, and `NEIGH_IN_EL` are the symbolic constants defining the dimension of the triangulation, etc. They are not defined by a `#define` statement in `albert.h`, but defined via the C-preprocessor. Hence, a special rule for compiling ALBERT source is needed.

Such a rule and the matching library are defined in a master Makefile, located in the `ALBERT_LIB_PATH` and is included, after the definition of the library path. In that makefile, all libraries used are collected in the variable `LIBS` which is used on the link line for all targets. Default values are set for `MOFLAG`, `CFLAGS`, `OFLAG` (depending on the compilers), and `DEBUG` is set to `0`.

Finally, targets for removing object files (`cleano`), object files and backup files from emacs (`clean`), and a target for recompiling all source files for the default target (`new`) are defined.

For a Linux environment with GNU compilers and the BLAS library installed in `/usr/local/lib/libblas.a`, this master Makefile reads:

```
########################################################################
#   C and FORTRAN compilers and loader                                 #
########################################################################

CC = gcc
FC = g77
LD = gcc

########################################################################
#   default settings                                                   #
########################################################################

MOFLAG =
CFLAGS = -pedantic -Wall
OFLAG = -O3
DEBUG = 0

########################################################################
#   used libraries                                                     #
########################################################################

BLAS_LIB = -L/usr/local/lib -lblas
GLTOOLS_LIB = -lgltools$(MOFLAG)
PLOT_LIB = -lPLOTansi$(MOFLAG)
GRAPE_LIB =
GLX11_LIB = -lGL -L/usr/X11R6/lib -lX11 -lXext -rdynamic -ldl

########################################################################
#   define flags and compile commands                                  #
########################################################################

DFLAGS = -DDIM=$(DIM) -DDIM_OF_WORLD=$(DIM_OF_WORLD)\
         -DNEIGH_IN_EL=$(NEIGH_IN_EL) -DEL_INDEX=$(EL_INDEX)
.c.o:
	$(CC) -c $(MOFLAG) $(OFLAG) $(CFLAGS) $(DFLAGS) -I$(ALBERT_LIB_PATH) $<
.f.o:
	$(FC) -c $(MOFLAG) $(FFLAGS) $(OFLAG) $<

########################################################################
#   the ALBERT library and libraries used by ALBERT                    #
########################################################################

ALBERT_LIB =\
 -lALBERT$(DIM)$(DIM_OF_WORLD)$(NEIGH_IN_EL)$(EL_INDEX)$(DEBUG)$(MOFLAG)

LIBS = -L$(ALBERT_LIB_PATH) $(ALBERT_LIB) -lcsolver$(MOFLAG) $(BLAS_LIB)\
         $(GLTOOLS_LIB) $(PLOT_LIB) $(GRAPE_LIB) $(GLX11_LIB) -lm

########################################################################
#   some often used targets                                            #
########################################################################

default: $(DEFAULT)
```

```
.PHONY: clean cleano new
clean:
	-rm *.o *~

cleano:
	-rm *.o

new: cleano default
```

Bibliography

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

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

[3] ——, *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.

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

[5] J. Bey, *Tetrehedral grid refinement.* Report 18, SFB 382 Tübingen, 1995.

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

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

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

[9] P. G. Ciarlet, *The finite element method for elliptic problems*, North-Holland, 1987.

[10] R. Cools and P. Rabinowitz, *Monomial cubature rules since "'Stroud"': a compilation*, J. Comput. Appl. Math., 48 (1993), pp. 309–326.

[11] 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.

[12] J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson, *An extended set of Fortran Basic Linear Algebra Subprograms*, ACM Trans. Math. Softw., 14 (1988), pp. 1–32.

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

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

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

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

[17] D. Dunavant, *High degree efficient symmetrical Gaussian quadrature rules for the triangle*, Int. J. Numer. Methods Eng., 21 (1985), pp. 1129–114.

[18] G. Dziuk, *An algorithm for evolutionary surfaces*, Numer. Math., 58 (1991), pp. 603 – 611.

[19] 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.

[20] J. Fröhlich, J. Lang, and R. Roitzsch, *Selfadaptive finite element computations with smooth time controller and anisotropic refinement.* Preprint SC 96-16, ZIB Berlin, 1996.

[21] J. Fuhrmann and H. Langmach, *gltools: OpenGL based online visualization.* Software: `http://www.wias-berlin.de/\~gltools`.

[22] K. Gatermann, *The construction of symmetric cubature formulas for the square and the triangle*, Computing, 40 (1988), pp. 229–240.

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

[24] H. Kardestuncer, ed., *Finite Element Handbook*, McGraw-Hill, New York, 1987.

[25] 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.

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

[27] C. Lawson, R. Hanson, D. Kincaid, and F. Krough, *Basic Linear Algebra Subprograms for Fortran usage*, ACM Trans. Math. Softw., 5 (1979), pp. 308–325.

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

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

[30] P. Morin, R. H. Nochetto, and K. G. Siebert, *Data oscillation and convergence of adaptive FEM.* Preprint 17/99 Freiburg, 1999. To appear in SINUM.

[31] R. H. Nochetto, M. Paolini, and C. Verdi, *An adaptive finite element method for two-phase Stefan problems in two space dimensions. Part II: Implementation and numerical experiments*, SIAM J. Sci. Stat. Comput., 12 (1991), pp. 1207–1244.

[32] A. Schmidt and K. G. Siebert, *A posteriori estimators for the h–p version of the finite element method in 1d.* Preprint 35/97 Freiburg, 1997.

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

[34] SFB 256, *GRAPE – GRAphics Programming Environment Manual, Version 5.0*, Bonn, 1995.

[35] 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.

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

[37] A. H. Stroud, *Approximate calculation of multiple integrals*, Prentice-Hall, Englewood Cliffs, NJ, 1971.

[38] R. Verfürth, *A posteriori error estimates for nonlinear problems: Finite element discretization of elliptic equations*, Math. Comp., 62 (1994), pp. 445–475.

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

[40] 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.

Index

Data types, symbolic constants, functions, and macros

List of data types

List of symbolic constants

List of functions

List of macros