ALBERTA 3.0: Technical Manual

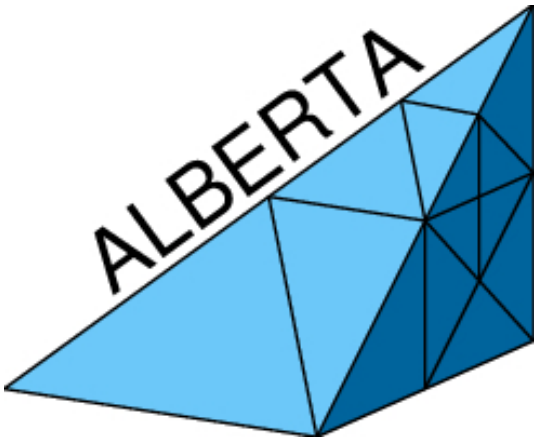
Alfred Schmidt
Zentrum fuer Technomathematik
Universität Bremen
Bibliothekstr. 2
D-28359 Bremen, Germany

Kunibert G. Siebert
Fachbereich Mathematik
Universität Stuttgart
Pfaffenwaldring 57
70569 Stuttgart, Germany

Daniel Köster
P+Z Engineering GmbH
(formerly Universität Augsburg)

Claus-Justus Heine
Fachbereich Mathematik
Universität Stuttgart
Pfaffenwaldring 57
70569 Stuttgart, Germany

http://www.alberta-fem.de



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

Version: ALBERTA-3.0, March 13, 2014

Preface to the Technical Manual for ALBERTA-3.0

This is the "Technical Manual" for the finite-element toolbox ALBERTA, version 3, a reference manual which intentionally lists all functions and data-structures exported to application programs. In other words: this manual contains the definition of what commonly is referred to as "API" – Application Program Interface. After the release of version 1.2 – which was accompanied by publishing the ALBERTA-book [24] through Springer (or vice-versa: the book was accompanied by the release of version 1.2) – there was already a successor labelled ALBERTA-2.0 (with slight bug-fixes in version 2.0.1), see www.alberta-fem.de. Version 2.0 was in its principal part the outcome of the labours of Daniel Köster.

Already at that time it was felt by the developers of ALBERTA that at least a reference manual – documenting the API – should be available as part of the source-code distribution of ALBERTA– or at least should be accessible through a less "fixed" medium than a book, prominently to make it easier to cover new developments and fix bogus documentation concerning API-functions, without having to republish the entire book. As a slightly strange side-effect, the reference manual starts with Chapter 2, which explains the example applications contained in the `alberta-3.0-demo` package. Occasionally, this manual contains back-references to "The Book", which is inconvenient, because that part is not yet publicly available. My apologies; the reader is referred to the ALBERTA-1.2 book [24]. The theoretical concepts explained there still hold.

On the other hand: providing "on-line" documentation does not grant the same merits as publishing a book. One way out of this dilemma was to separate a "book"-section from the API-description. The book intentionally describes the abstract concepts underlying the ALBERTA-toolbox, while the API-documention – this document – lists the available functions and data-structures in a (hopefully) application oriented manner, is available on-line, and thus can be maintained more easily.

ALBERTA-3.0 is primarily the product of extensions added by me to the toolbox during my time at the University of Freiburg. The principal differences to version 2.0 – according to my judgement – are

- Vector-valued basis functions.
- Direct sums of finite element spaces, which – together with the previous point – allow for the implementation of several of the known stable mixed discretizations for the Stokes-problem in a fairly convenient manner.
- An add-on package `libalbas` (distributed with the core-package) implementing some of the more fancy mixed Stokes-discretizations.
- Periodic boundary conditions, including, but *not* limited to mere translations, maintaining compatibility with the "sub-mesh" – or better: "trace-meshes" – introduced in version 2.0.
- (Iso-)parametric meshes of arbitrary degree (the support in ALBERTA-2.0 was limited to at most piece-wise quadratic parameterizations). To be honest: V3.0 does not support anything beyond degree 4, but simply because the underlying Lagrangian finite element spaces are only implemented up to degree 4.
- Space-meshes in arbitrary co-dimension, of course with support for higher order parameterizations.

- Iso-parametric higher-order boundary approximation, implementing the algorithms described in [15]. A fairly complicated issue.
- A cleaner separation of geometric data defined in the macro-triangulation and the interpretation of this data by the application. In particular, the implementation of boundary conditions has changed substantially, see Section 3.2.4.
- Limited support for Discontinuous Galerkin methods, implemented through a new structure describing kind of boundary operators.
- Likewise, differential operators optionally may be accompanied by contributions "living" on the boundary of the computational domain. This opens the possibility to, e.g., implement Robin boundary conditions without having to define a trace-mesh, simply because a boundary integral has to be computed to assemble the operator.
- Example-programs for most of the new features, distributed along with the suite of demo-programs (see Chapter 2).

There are special "Compatibility Notes" interspersed with the documentation for the individual structures and functions, concerning differences to the predecessor ALBERTA-2.0.

Most of the time my work on the toolbox through the recent years – after Daniel Köster stopped working on ALBERTA because of his occupation in his industrial employment – was a one-man show. However, lately there were noticable contributions by the following people: Rebecca Stotz (Paraview interface, static condensation for the "Mini"-element, synchronization between the reference manual and the source-code of the library, "`HOWTO-port-ellipt.txt`" document), Notger Noll (`C`-source-code for a block-matrix solver interface, work on the compatibility layer for `read_mesh()` function, inclusion of the `symmlq`-solver [20], synchronization between the reference manual and the source code), Christian Haarhaus (`read_mesh()` compatibility layer, thus enabling ALBERTA-3.0 to read version-1.2 data. Grid-generator interface from *FreeFem++* to ALBERTA), Björn Stinner (contributing code for mesh-smoothing through the computation of conformal mappings to the unit-sphere. "Ported" to recent versions of the toolbox by Rebecca Stotz)

Further, my thanks go to Thilo Moshagen for beta-testing and fruitful discussions, Thomas Bonesky for beta-testing, Robert Nürnberg and Ed Tucker for their bug-reports. In the likely case that somebody is missing in above lists: my apologies, if so, then he or she was left out unintentionally. ALBERTA-3.0 serves as a back-end for Thilo Moshagens *albertasystems* package (a `C++` toolbox for the discretization of systems of many scalar equations), my own *unfem++*-toolbox (a toolbox for unfitted finite elements); it is also supported by the recent development versions of *Dune* (which uses the implementation of the hierarchical mesh from ALBERTA, besides supporting "mesh-implementations" from a variety of other packages).

Of course, most prominently I'd like to thank the two principal authors of ALBERTA-1.2, Kunibert Siebert and Alfred Schmidt, and Daniel Köster.

Freiburg im Breisgau, March 13, 2014 Claus-Justus Heine

Contents

List of Figures

List of Tables

Chapter 2

Implementation of model problems

In this chapter we describe the implementation of two stationary model problems (the linear Poisson equation and a nonlinear reaction-diffusion equation) and of one time dependent model problem (the heat equation). Here we give an overview how to set up an ALBERTA program for various applications. We do not go into detail when refering to ALBERTA data structures and functions. A detailed description can be found in Chapter 3. We start with the easy and straight forward implementation of the Poisson problem to learn about the basics of ALBERTA. The examples with the implementation of the nonlinear reaction-diffusion problem and the time dependent heat equation are more involved and show the tools of ALBERTA for attacking more complex problems. Removing all LaTeX descriptions of functions and variables results in the source code for the adaptive solvers. During the installation of ALBERTA (described in Section 2.5) a tar-archive

```
PREFIX/share/alberta/alberta-VERSION-demo.tar.gz
```

is installed as well (`PREFIX` denoting the installation prefix, as specified by the `--prefix` parameter for the `configure` script). The tar-archive can be extracted at a location where the respective user has write permissions:

```
jane_john_doe@street ~ $ tar -xf PREFIX/share/alberta/alberta-VERSION-demo.tar.gz
jane_john_doe@street ~ $ cd alberta-VERSION-demo
jane_john_doe@street ~/alberta-VERSION-demo $ less README
jane_john_doe@street ~/alberta-VERSION-demo $ cd src/2d
jane_john_doe@street ~/alberta-VERSION-demo/src/2d $ make ellipt
jane_john_doe@street ~/alberta-VERSION-demo/src/2d $ ./ellipt
```

The archive extracts into a sub-directory having the same name as the base-name of the tar-archive. The corresponding ready-to-compile programs can be found in the files `ellipt.c`, `heat.c`, and `nonlin.c`, `nlprob.c`, `nlsolve.c` in the subdirectory `alberta2-demo/src/Common/`. Executable programs for different space dimensions can be generated in the subdirectories `alberta2-demo/src/1d/`, `alberta2-demo/src/2d/`, and `alberta2-demo/src/3d/` by calling `make ellipt`, `make nonlin`, and `make heat`. There are also a couple of other programs, please refer to the file `README` in the top-level directory of the demo-package. The idea was to generate one variant of the `ellipt.c` program for each new feature introduced for the current ALBERTA version (higher order parametric meshes, higher co-dimension parametric meshes, periodic meshes, vector-valued basis functions and direct sums of finite element spaces, limited support for DG-methods). Mostly, these programs have the name `ellipt-FEATURE.c`.

The make-files in the demo-package interprete a `DEBUG`-switch specified on the command-line. This can be useful when modifying the demo-programs to suite the user's own needs. The resulting programs will be compiled with debugging information, such that they can be run from within a source-level debugger. Mind the leading call to `make clean`, the `make`-program cannot know that it should remake the programs!

```
jane_john_doe@street ~/alberta-VERSION-demo/src/2d $ make DEBUG=1 clean ellipt
jane_john_doe@street ~/alberta-VERSION-demo/src/2d $ gdb ellipt
```

2.1 `libdemo.a`

The example programs share some common routines for processing command-line switches, parameter parsing and for some sort of online-graphics. These routines consequently have been put into a small library called `libdemo.a`. The proto-types for the support functions are provided through the file `alberta-demo.h`, its essential part looks like follows:

```
#include <limits.h>
#ifndef PATH_MAX
# define PATH_MAX 1024
#endif

#include <alberta.h>
#include "graphics.h"
#include "geomview-graphics.h"

extern void parse_parameters(int argc, char *argv[], const char *init_file);
```

2.1.1 Online-graphics

As can be seen in the source-code listing above, the definitions, for graphical are in turn included from `graphics.h` and `geomview-graphics.h`. The demo-programs described in this manual use only the definitions from `graphics.h`, resulting in either a home-brewed 2d graphics, or output through the `gltools` package, if that could be found during the configuration of the ALBERTA distribution.

```
void graphics(MESH *mesh, DOF_REAL_VEC *u_h, REAL (*get_est)(EL *el),
              REAL (*u)(const REAL_D x), REAL time);

void graphics_d(MESH *mesh, DOF_REAL_VEC_D *u_h, DOF_REAL_VEC *p_h,
                REAL (*get_est)(EL *el),
                const REAL *(*u)(const REAL_D val, REAL_D x), REAL time);
```

The proto-type for the `geomview`-interface looks like follows:

```
extern void togeomview(MESH *mesh,
                       const DOF_REAL_VEC *u_h,
                       REAL uh_min, REAL uh_max,
                       REAL (*get_est)(EL *el),
                       REAL est_min, REAL est_max,
                       REAL (*u_loc)(const EL_INFO *el_info,
                                     const REAL_B lambda,
                                     void *ud),
```

```
                    void *ud, FLAGS fill_flags,
                    REAL u_min, REAL u_max);
```

Geomview is used by the demonstration programs for parametric meshes in higher (co-)dimension. We refer the reader to the example programs for the calling conventions for the graphic-routines (although we know that these should be explained in some more detail). Specifically, when `gltools` is in use, then pressing the key "`h`" in one of the output-windows displays a very brief online help in the terminal the program is running in.

As ALBERTA was developed in an environment where mostly Unix-like operating systems were in use, the online-graphics uses the X window system (www.xorg.org), so redirection of graphical output to other other machines by means of the `DISPLAY` environment variable is possible.

2.1.2 parse_parameters()

We give a more detailed explanation for the following routine:

Prototype

```
void parse_parameters(int argc, char *argv[], const char *init_file);
```

Parameters

argc, argv The program's command-line parameters, as passed to the `main()` function. See any `C` programming manual.

init_file The name of the file containing the parameters, usually having the form `"INIT/<program>.dat"`, but the name is arbitrary and the choice is left to the application.

Description

The function `parse_parameters()` initializes the access to parameters defined in parameter files, commonly found in

```
alberta-VERSION-demo/src/2d/INIT/<program>.dat
```

and likewise for the other dimensions. The access to the parameters is explained in greater detail, especially in the section dealing with the demonstration for the Poisson-problem, see Section 2.2 below. The actual source-code for `parse_parameters()` is contained in `src/Common/cmdline.c` (the path being relative to the demo-package).

`parse_parameters()` implements some command-line switches, prominently the `-h` or `--help` switches:

```
jane_john_doe@street ~/alberta-VERSION-demo/src/2d $ ./ellipt --help
Usage: ./ellipt [-h] [-i INITFILE] [-p PARAMETERS]
[--help] [--init-file=INITFILE] [--parameters=PARAMETERS]
jane_john_doe@street ~/alberta-VERSION-demo/src/2d $ ./ellipt -i myparams
 --parameters="degree=3'do_graphics=0"
```

So `-i` or `--init-file` allows the user to override the name of the default parameter-file, and `-p` or `--parameters` allows the user to override specific parameters from the parameter-file, in the example above `jane_john_doe` request that the finite element simulation is to be run with Lagrange elements of degree 3 and that no graphical output should appear during the simulation. The general format of the argument to `--parameters` or `-p` is

```
KEY1=VALUE1'KEY2=VALUE2...
```

So "`=`" separates a given key from its value, and a single quote separates the key-value pairs. Note that it might be necessary to escape the single quote, or to enclose the entire argument by double quotes (as in the example given above).

2.2 Poisson equation

In this section we describe a model implementation for the Poisson equation

$$\begin{aligned} -\Delta u &= f && \text{in } \Omega \subset \mathbb{R}^d, \\ u &= g && \text{on } \Gamma_d, \\ \partial_\nu u + \alpha_r\, u &= g_n && \text{on } \Gamma_n, \text{ with } \partial\Omega = \Gamma_d \dot\cup \Gamma_n. \end{aligned}$$

Apart from the slightly complicated boundary conditions this is the most simple elliptic problem, but the program presents all major ingredients for general scalar stationary problems. Also, Poisson equations often occur as sub-problems in much more complicated settings. Modifications needed for a nonlinear problem are presented in Section 2.3.

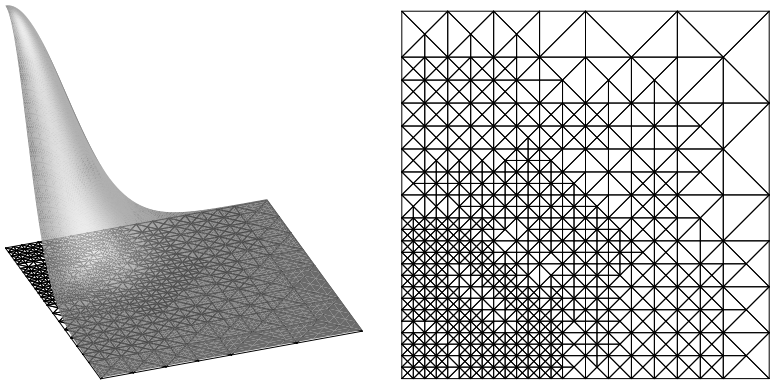


Figure 2.1: Solution of the linear Poisson problem and corresponding mesh. The pictures were produced by GRAPE.

Data and parameters described below lead in 2d to the solution and mesh shown in Figure 2.1. The implementation of the Poisson problem is split into several major steps which are now described in detail.

2.2.1 Include file and global variables

All ALBERTA source files must include the header file `alberta.h` with all ALBERTA type definitions, function prototypes and macro definitions:

```
#include <alberta.h>
```

This is realised by including the header file `alberta-demo.h` which additionally includes the header files `graphics.h` and `geomview-graphics.h` for graphical output:

```
#include "alberta-demo.h"
```

leads to:

```
#include <alberta.h>
#include "graphics.h"
#include "geomview-graphics.h"
```

For the linear scalar elliptic problem we use four global pointers to data structures holding the finite element space and components of the linear system of equations. These are used in different subroutines where such information cannot be passed via parameters.

```
static const FE_SPACE *fe_space;
static DOF_REAL_VEC   *u_h;
static DOF_REAL_VEC   *f_h;
static DOF_MATRIX     *matrix;
```

`fe_space` a pointer to the actually used finite element space; it is initialized by the function `main()`, see Section 2.2.4;

`u_h` a pointer to a DOF vector storing the coefficients of the discrete solution; it is initialized by the function `main()`

`f_h` a pointer to a DOF vector storing the load vector; it is initialized by the function `main()`

`matrix` a pointer to a DOF matrix storing the system matrix; it is initialized by the function `main()`

The data structure `FE_SPACE` is explained in Section 3.6.1, `DOF_REAL_VEC` in Section 3.3.2, and `DOF_MATRIX` in Section 3.3.4. Details about DOF administration `DOF_ADMIN` can be found in Section 3.3.1 and about the data structure `MESH` for a finite element mesh in Section 3.2.12.

We use another set of three global variable which store information about the boundary conditions in use:

```
static REAL robin_alpha = -1.0;
static bool pure_neumann = false;
static BNDRY_FLAGS dirichlet_mask; /* bit-mask of Dirichlet segments */
```

`robin_alpha` The zero-order factor if Robin-boundary conditions are prescribed, see Section 4.7.7.3.

`pure_neumann` When prescribing Neumann boundary conditions on all parts of the boundary, then the solution is only determined up to an additive constant. In this case it is necessary to perform a mean-value "correction" before, e.g., computing an error in the L^2-norm.

`dirichlet_mask` Initialized by a call to `GET_PARAMTER()` in the main-function. A bit-mask tagging boundary segments on which the discrete solution is subject to Dirichlet boundary conditions. See Section 3.2.4 and Section 4.7.7.1.

2.2.2 The main program for the Poisson equation

The main program is very simple, it just includes the main steps needed to implement any stationary problem. Special problem-dependent aspects are hidden in other subroutines described below.

We first read a parameter file (indicating which data, algorithms, and solvers should be used; the file is described below in Section 2.2.3). The call to `parse_parameters()` is further explained in the section Section 2.1.2 above. The parameters fetched from the parameter file at this point in the code are:

`dim` Dimension of the mesh.

`filename` The file-name for the macro-triangulation.

`degree` The desired polynomial degree for the finite element triangulation (should be between 1 and 4).

`n_refine` The number of global refinements of the mesh to be performed before starting the simulation.

`do_graphics` A boolean value for disabling all graphical output (individual windows can be disabled separately, see below in Section 2.2.3.

`dirichlet_bit` The number of the boundary segment where Dirichlet boundary conditions should be imposed. See also Section 3.2.4. Boundary segments having another number than `dirichlet_bit` are Neumann- or Robin-boundaries.

`robin_alpha` If positive, the zero-order parameter for a Robin-boundary condition. If negative and no boundary segment is a Dirichlet-boundary, then the discrete right-hand side will be forced to obey the mean-value zero compatibility condition. See Section 4.7.7.

Having fetched those basic parameters from the data file `INIT/elliptic.dat` we read the macro triangulation and initialize the mesh (the basic geometric data structure). The subdirectories `Macro/` in the `alberta-VERSION-demo/src/*d/` directories contain data for several sample macro triangulations. How to read and write macro triangulation files is explained in Section 3.2.15.

Now that the domain's geometry is defined, we allocate standard Lagrange basis functions and from them generate a finite element space through a call to `get_fe_space()`. The mesh is globally refined if necessary. A call to `graphics()` displays the initial mesh, unless the parameter `do_graphics` has been initialized to `false`, in which case no graphical output at all will appear.

Afterwards, the DOF vectors `u_h` and `f_h`, and the DOF matrix `matrix` are allocated. 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, there is no need for interpolation during mesh modifications or initialization. Additionally, we initialize the global variable `dirichlet_mask`, setting the bit `dirichlet_bit` to mark those parts of the boundary, which are subject to Dirichlet boundary conditions. The variable `dirichlet_mask` is later on used by several other routines: for matrix assembly, to install Dirichlet boundary conditions into the load vector, and during the computation of the error estimate.

The basic algorithmic data structure `ADAPT_STAT` introduced in Section 4.8.1 specifies the behaviour of the adaptive finite element method for stationary problems. A pre-initialized

data structure is accessed by the function `get_adapt_stat()`; the most important members (`adapt->tolerance`, `adapt->strategy`, etc.) are automatically initialized with values from the parameter file; other members can be also initialized by adding similar lines for these members to the parameter file (compare Section 4.8.4). Eventually, function pointers for the problem dependent routines have to be set (`estimate`, `get_el_est`, `build`, `solve`). Since the assemblage is done in one step after all mesh modifications, only `adapt->build_after_coarsen` is used, no assemblage is done before refinement or before coarsening. These additional assemblage steps are possible and may be needed in a more general application, for details see Section 4.8.1.

The adaptive procedure is started by a call of `adapt_method_stat()`. This automatically solves the discrete problem, computes the error estimate, and refines the mesh until the given tolerance is met, or the maximal number of iterations is reached, compare Section 4.8.1. Finally, `WAIT_REALLY` allows an inspection of the final solution by preventing a direct program exit with closure of the graphics windows. The `WAIT_REALLY`-blocker is not necessary when using the `gltools` package for the graphical output.

```
int main(int argc, char **argv)
{
  FUNCNAME("main");
  MACRO_DATA          *data;
  MESH                *mesh;
  int                  n_refine = 0, dim, degree = 1, dirichlet_bit = 1;
  const BAS_FCTS      *lagrange;
  static ADAPT_STAT   *adapt;
  char                 filename[PATH_MAX];

  /*****************************************************************************
   * first of all, initialize the access to parameters of the init file
   *****************************************************************************/

  parse_parameters(argc, argv, "INIT/ellipt.dat");

  GET_PARAMETER(1, "mesh dimension", "%d", &dim);
  GET_PARAMETER(1, "macro file name", "%s", filename);
  GET_PARAMETER(1, "polynomial degree", "%d", &degree);
  GET_PARAMETER(1, "global refinements", "%d", &n_refine);
  GET_PARAMETER(1, "online graphics", "%B", &do_graphics);
  GET_PARAMETER(1, "dirichlet boundary", "%d", &dirichlet_bit);
  GET_PARAMETER(1, "robin factor", "%f", &robin_alpha);

  /*****************************************************************************
   *  get a mesh, and read the macro triangulation from file
   *****************************************************************************/
  data = read_macro(filename);
  mesh = GET_MESH(dim, "ALBERTA mesh", data,
                  NULL /* init_node_projection() */,
                  NULL /* init_wall_trafos() */);
  free_macro_data(data);

  init_leaf_data(mesh, sizeof(struct ellipt_leaf_data),
                 NULL /* refine_leaf_data() */,
                 NULL /* coarsen_leaf_data() */);
```

```
  /*****************************************************************************
   *  initialize the global variables shared across build(), solve()
   *  and estimate().
   *****************************************************************************/
  lagrange = get_lagrange(mesh->dim, degree);
  TEST_EXIT(lagrange, "no lagrange BAS_FCTS\n");
  fe_space = get_fe_space(mesh, lagrange->name, lagrange, 1 /* rdim */,
                          ADM_FLAGS_DFLT);

  global_refine(mesh, n_refine * mesh->dim, FILL_NOTHING);

  if (do_graphics) {
    MSG("Displaying the mesh.\n");
    graphics(mesh, NULL /* u_h */, NULL /* get_est()*/ , NULL /* u_exact()
        */,
             HUGE_VAL /* time */);
  }

  matrix = get_dof_matrix("A", fe_space, NULL /* col_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 (dirichlet_bit > 0) {
    BNDRY_FLAGS_SET(dirichlet_mask, dirichlet_bit);
  }

  /*****************************************************************************
   *  init adapt structure and start adaptive method
   *****************************************************************************/
  adapt = get_adapt_stat(mesh->dim, "ellipt", "adapt", 2,
                         NULL /* ADAPT_STAT storage area, optional */);
  adapt->estimate = estimate;
  adapt->get_el_est = get_el_est;
  adapt->build_after_coarsen = build;
  adapt->solve = solve;

  adapt_method_stat(mesh, adapt);

  if (do_graphics) {
    MSG("Displaying u_h, u, (u_h-u) and the final estimate.\n");
    graphics(mesh, u_h, get_el_est, u, HUGE_VAL /* time */);
  }
  WAIT_REALLY;

  return 0;
}
```

2.2.3 The parameter file for the Poisson equation

The following parameter file `INIT/ellipt.dat` is used for the `ellipt.c` program in the 2d case:

```
mesh dimension:       2
macro file name:      Macro/macro.amc
global refinements:   1
polynomial degree:    3

dirichlet boundary:   1 % type of the Dirichlet boundary segment,
                        % must correspond to the boundary types used
                        % used in the macro triangulation. Use a value
                        % <= 0 to disable Dirichlet boundary
                        % conditions. Neumann boundary conditions will
                        % hold for all boundary segments with a type
                        % different from the value specified here.
robin factor:        -1 % > 0: Robin b.c.

online graphics:      true % global gfx kill-switch

% graphic windows: solution, estimate, mesh, and error if size > 0
graphic windows:      400 400 400 400
% for gltools graphics you can specify the range for the values of
% discrete solution for displaying:  min max
% automatical scaling by display routine if min >= max
gltools range:  0.0 -1.0

solver:                2 % 1: BICGSTAB 2: CG 3: GMRES 4: ODIR 5: ORES
solver max iteration:  10000
solver restart:        10  %  only used for GMRES
solver tolerance:      1.e-8
solver info:           2
solver precon:         2   % 0: no precon
                           % 1: diag precon
                           % 2: HB precon
                           % 3: BPX precon
                           % 4: SSOR, omega = 1.0, #iter = 3
                           % 5: SSOR, with control over omega and #iter
                           % 6: ILU(k)
precon ssor omega:    1.0  % for precon == 5
precon ssor iter:     1    % for precon == 5
precon ilu(k):        8    % for precon == 6

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:          1 % 0: no adaption / 1: GR / 2: MS / 3: ES / 4: GERS
adapt->tolerance:         1.e-4
adapt->MS_gamma:          0.5
adapt->MS_gamma_c:        0.1
adapt->ES_theta:          1.9
adapt->ES_theta_c:        0.2
adapt->GERS_theta_star:   0.6
adapt->GERS_nu:           0.1
adapt->GERS_theta_c:      0.1
```

```
adapt->coarsen_allowed:   1
adapt->max_iteration:     20
adapt->info:              8

WAIT: 1
```

The file `Macro/macro.amc` storing data about the macro triangulation for $\Omega = (0,1)^d$ can be found in Section 3.2.15 for 2d and 3d. The `polynomial degree` parameter selects the third order Lagrange elements. `dirichlet boundary` marks those parts of the boundary which are subject to Dirichlet boundary conditions, see also Section 3.2.4. The value of `dirichlet boundary` corresponds to the numbers assigned to boundary segments in the macro-triangulation.

By `graphic windows`, the number and sizes of graphics output windows are selected. This line is used by the `graphics()` routine. For gltools graphics, the range of function values might be specified (used for graph coloring and height). If no graphical output at all is desired, then `online graphics` can be set to `false`. Individual output windows can be disabled by setting their size to 0. The size is specified in units of screen pixels.

The solver for the linear system of equations is selected (here: the conjugate gradient solver), and corresponding parameters like preconditioner and tolerance. Some preconditioners need additional parameters, these are specified here as well.

Parameters for the error estimator include values of different constants and selection of the error norm to be estimated (H^1- or L^2-norm, selection leads to multiplication with different powers of the local mesh size in the error indicators), see Section 4.9.1.

An error tolerance and selection of a marking strategy with corresponding parameters are main data given to the adaptive method. For the meaning of the individual parameters the reader is referred to the conceptional Section 1.5.2 and Section 1.5.3 in the book-part of the manual, and to Section 1.5 which describes the implementation of adaptive methods in ALBERTA.

Finally, the `WAIT` parameter specifies whether the program should wait for user interaction at additional breakpoints, whenever a `WAIT` statement is executed as in the routine `graphics()`, for instance, in case the `gltools` package is not in use.

The solution and corresponding mesh in 2d for the above parameters are shown in Figure 2.1. As optimal parameter sets might differ for different space dimensions, separate parameter files exist in `1d/INIT/`, `2d/INIT/`, and `3d/INIT/`.

2.2.4 Initialization of the finite element space

In contrast to prior versions of ALBERTA, finite element spaces may be newly allocated at any time. Since this involves updating DOF information on all elements, however, it is advisable to allocate finite element spaces before refining a mesh, see also Sections 3.3.6 and 3.6.2.

For the scalar 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 initialized by `get_fe_space()` which automatically stores at the mesh information about the DOFs used by this finite element space.

It is possible to allocate several finite element spaces, for instance in a mixed finite element method, compare Section 3.6.2.

2.2.5 Functions for leaf data

As explained in Section 3.2.10, we can "hide" information which is only needed on a leaf element at the pointer to the second child. Such information, which we use here, is the local error indicator on an element. For this elliptic problem we need one `REAL` for storing this element indicator.

After 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. Here, leaf data is one structure `struct ellipt_leaf_data` and no transformation during mesh modifications is needed. The details of the `LEAF_DATA_INFO` data structure are stated in Section 3.2.10.

```
init_leaf_data(mesh, sizeof(struct ellipt_leaf_data),
               NULL /* refine_leaf_data() */,
               NULL /* coarsen_leaf_data() */);
```

The error estimation is done by the library function `ellipt_est()`, see Section 4.9.1. 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 4.8.1. 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 estimate */
};

static REAL *rw_el_est(EL *el)
{
  if (IS_LEAF_EL(el))
    return &((struct ellipt_leaf_data *)LEAF_DATA(el))->estimate;
  else
    return NULL;
}

static REAL get_el_est(EL *el)
{
  if (IS_LEAF_EL(el))
    return ((struct ellipt_leaf_data *)LEAF_DATA(el))->estimate;
  else
    return 0.0;
}
```

2.2.6 Data of the differential equation

Data for the Poisson problem 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

$$f(x) = -\Delta u(x) = -(400\,|x|^2 - 20\,d)\,\mathrm{e}^{-10\,|x|^2}.$$

Here, d denotes the space dimension, $\Omega \subset \mathbb{R}^d$. The functions `u()` and `grd_u()` are the implementation of u and ∇u and are optional (and usually not known for a general problem). The functions `g()`, `gn()` and `f()` are implementations of the boundary values and the right hand side and are not optional. Of course, `g()` needs only to be implemented when Dirichlet boundary conditions apply, likewise `gn()` only for inhomogeneous Robin or Neumann boundary conditions (see Section 4.7.7.3 and Section 4.7.7.2.

```
#define GAUSS_SCALE 10.0

static REAL u(const REAL_D x)
{
  return exp(-GAUSS_SCALE*SCP_DOW(x,x));
}

static const REAL *grd_u(const REAL_D x, REAL_D grd)
{
  static REAL_D buffer;
  REAL          ux = exp(-GAUSS_SCALE*SCP_DOW(x,x));
  int           n;

  if (!grd) {
    grd = buffer;
  }

  for (n = 0;  n < DIM_OF_WORLD; n++)
    grd[n] = -2.0*GAUSS_SCALE*x[n]*ux;

  return grd;
}

/*****************************************************************************
 * problem data: right hand side, boundary values
 *****************************************************************************/

static REAL g(const REAL_D x) /* boundary values, not optional */
{
  return u(x);
}

static REAL gn(const REAL_D x, const REAL_D normal) /* Neumann b.c. */
{
  return robin_alpha > 0.0
    ? SCP_DOW(grd_u(x, NULL), normal) + robin_alpha * u(x)
    : SCP_DOW(grd_u(x, NULL), normal);
}
```

```
static REAL f(const REAL_D x) /* -Delta u, not optional */
{
  REAL r2 = SCP_DOW(x,x), ux = exp(-GAUSS_SCALE*r2);
  return -(4.0*SQR(GAUSS_SCALE)*r2 - 2.0*GAUSS_SCALE*DIM_OF_WORLD)*ux;
}
```

A common principle in the implementation of functions of the type `grd_u` is that we store the result either at the caller-specified pointer `input`, if provided, or overwrite a local static `buffer` on each call.

2.2.7 The assemblage of the discrete system

For the assemblage of the discrete system we use the tools described in Sections 4.7.2, 4.7.6, and 4.7.7.1. For the matrix assemblage we have to provide an element-wise description of the differential operator. Following the description in Section 1.4.8 we provide the function `init_element()` for an initialization of the operator on an element and the function `LALt()` for the computation of $\det|DF_S|\Lambda A\Lambda^t$ on the actual element, where Λ is the Jacobian of the barycentric coordinates, DF_S the the Jacobian of the element parameterization, and A the matrix of the second order term. For $-\Delta$, we have $A = id$ and $\det|DF_S|\Lambda\Lambda^t$ is the description of the complete differential operator since no lower order terms are involved.

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_data` which stores the Jacobian and the determinant. The function `init_element()` calculates the Jacobian and the determinant by the library functions `el_grd_lambda_?d()` and the function `LALt()` uses these values in order to compute $\det|DF_S|\Lambda\Lambda^t$. The mesh dimension d given by `mesh->dim` is always less than or equal to the world dimension n given by the macro `DIM_OF_WORLD`, hence we comment out irrelevant parts of the code.

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 global system matrix (see also Example 4.7.3 in Section 4.7.2 for the access to a structure `matrix_info`). For more general equations with lower order terms, additional functions `Lb0`, `Lb1`, and/or `c` have to be defined at that point. This initialization is done on the first call of the function `build()` which is called by `adapt_method_stat()` during the adaptive cycle (compare Section 4.8.1).

By calling `dof_compress()`, unused DOF indices are removed such that the valid DOF indices are consecutive in their range. This guarantees optimal performance of the BLAS1 routines used in the iterative solvers and `admin->size_used` is the dimension of the current finite element space. This dimension is printed for information.

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, the boundary conditions are installed into the load-vector, and possibly also into the matrix in the case of Robin boundary conditions. Dirichlet boundary values are also interpolated into the vector `u_h` for the discrete solution. If only Dirichlet boundary conditions are desired, then the call to `boundary_conditions()` quoted below could be replaced by a less complicated call to `dirichlet_bound()`:

```
dirichlet_bound(f_h, u_h, NULL, dirichlet_mask, g);
```

Analogously, if only inhomogeneous Neumann boundary conditions should be implemented, then a call to `bndry_L2scp_fct_bas()` could replace the call to `boundary_conditions()`. Compare Sections 4.7.6, 4.7.7.1, 4.7.7.2, 4.7.7.3 and 4.7.7.

```
struct op_data
{
  REAL_BD  Lambda; /*  the gradient of the barycentric coordinates */
  REAL     det;    /*  |det D F_S|                                  */
};

static
bool init_element(const EL_INFO *el_info, const QUAD *quad[3], void *ud)
{
  struct op_data *info = (struct op_data *)ud;

  /* ..._0cd: co-dimension 0 version of el_grd_lambda(dim, ...) */
  info->det = el_grd_lambda_0cd(el_info, info->Lambda);

  return false; /* not parametric */
}

static
const REAL_B *LALt(const EL_INFO *el_info, const QUAD *quad,
                   int iq, void *ud)
{
  static REAL_BB LALt;
  struct op_data *info = (struct op_data *)ud;
  int            i, j, dim = el_info->mesh->dim;

  for (i = 0; i < N_VERTICES(dim); i++) {
    LALt[i][i] = info->det*SCP_DOW(info->Lambda[i], info->Lambda[i]);
    for (j = i+1; j < N_VERTICES(dim); j++) {
      LALt[i][j] = SCP_DOW(info->Lambda[i], info->Lambda[j]);
      LALt[i][j] *= info->det;
      LALt[j][i] = LALt[i][j];
    }
  }

  return (const REAL_B *)LALt;
}

static void build(MESH *mesh, U_CHAR flag)
{
  FUNCNAME("build");
  static const EL_MATRIX_INFO *matrix_info;

  dof_compress(mesh);
  MSG("%d_DOFs_for_%s\n", fe_space->admin->size_used, fe_space->name);

  if (!matrix_info) {
    /* information for matrix assembling (only once) */
    OPERATOR_INFO  o_info = { NULL, };
    static struct op_data user_data; /* storage for det and Lambda */

    o_info.row_fe_space    = o_info.col_fe_space = fe_space;
    o_info.init_element    = init_element;
```

```
    o_info.LALt.real        = LALt;
    o_info.LALt_pw_const    = true;         /* pw const. assemblage is faster
        */
    o_info.LALt_symmetric   = true;         /* symmetric assemblage is faster
        */
    BNDRY_FLAGS_CPY(o_info.dirichlet_bndry,
                    dirichlet_mask);          /* Dirichlet bndry conditions
                       */
    o_info.user_data = (void *)&user_data;    /* application data */
    o_info.fill_flag = CALL_LEAF_EL|FILL_COORDS; /* only FILL_BOUND is added
                                                  * automatically.
                                                  */
    matrix_info = fill_matrix_info(&o_info, NULL);
  }

  /* assembling of matrix */
  clear_dof_matrix(matrix);
  update_matrix(matrix, matrix_info, NoTranspose);

  /* assembling of load vector */
  dof_set(0.0, f_h);
  L2scp_fct_bas(f, NULL /* quadrature */, f_h);

  /* Boundary values, the combination alpha_r < 0.0 flags automatic
   * mean-value correction iff f_h has non-zero mean-value and no
   * non-Neumann boundary conditions were detected during mesh
   * traversal.
   */
  pure_neumann =
    !boundary_conditions(matrix, f_h, u_h, NULL /* bound */,
                         dirichlet_mask,
                         g, gn,
                         robin_alpha, /* < 0: mean-value correction */
                         NULL /* wall_quad, use default */);
}
```

2.2.8 The solution of the discrete system

The function `solve()` computes the solution of the resulting linear system. It is called by `adapt_method_stat()` (compare Section 4.8.1). The system matrix for the Poisson equation is positive definite and symmetric for non-Dirichlet DOFs. Thus, the solution of the resulting linear system is rather easy and we can use any preconditioned Krylov-space solver (`oem_solve_s()`), compare Section 4.10.2. 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 `max_iter`, the level of information printed by the linear solver, and the use of a preconditioner by the parameter `icon`, which may be `0` (no preconditioning), `1` (diagonal preconditioning), `2` (hierarchical basis preconditioning), `3` (BPX preconditioning), `4` (SSOR preconditioning, with given `omega` = 1.0, `#iter` = 3), `5` (SSOR preconditioning, with control over `omega` and `#iter`), or `6` (ILU(k) preconditioning). If GMRes is used, then the dimension of the Krylov-space for the minimizing procedure is needed, too. If ILU(k) is used, then the level k is needed, too (ILU(k) denotes the ILU-flavour described in [3]).

After solving the discrete system, the discrete solution (and mesh) is displayed by calling `graphics()`.

```
static void solve(MESH *mesh)
{
  FUNCNAME("solve");
  static REAL tol = 1.e-8, ssor_omega = 1.0;
  static int  max_iter = 1000, info = 2, restart = 0;
  static int  ssor_iter = 1, ilu_k = 8;
  static OEM_PRECON icon = DiagPrecon;
  static OEM_SOLVER solver = NoSolver;
  const PRECON *precon;

  if (solver == NoSolver) {
    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", &max_iter);
    GET_PARAMETER(1, "solver info", "%d", &info);
    if (icon == _SSORPrecon) {
        GET_PARAMETER(1, "precon ssor omega", "%f", &ssor_omega);
        GET_PARAMETER(1, "precon ssor iter", "%d", &ssor_iter);
    }
    if (icon == ILUkPrecon)
        GET_PARAMETER(1, "precon ilu(k)", "%d", &ilu_k);
    if (solver == GMRes) {
      GET_PARAMETER(1, "solver restart", "%d", &restart);
    }
  }

  if (icon == ILUkPrecon)
    precon = init_oem_precon(matrix, NULL, info, ILUkPrecon, ilu_k);
  else
    precon = init_oem_precon(matrix, NULL, info, icon, ssor_omega,
        ssor_iter);
  oem_solve_s(matrix, NULL, f_h, u_h,
              solver, tol, precon, restart, max_iter, info);

  if (do_graphics) {
    MSG("Displaying u_h, u and (u_h-u).\n");
    graphics(mesh, u_h, NULL /* get_el_est */, u, HUGE_VAL /* time */);
  }

  return;
}
```

2.2.9 Error estimation

The last ingredient missing for the adaptive procedure is a function for an estimation of the error. For an elliptic problem with constant coefficients in the second order term this can done by the library function `ellipt_est()` which implements the standard residual type error estimator and is described in Section 4.9.1. `ellipt_est()` needs a pointer to a function for writing the local error indicators (the function `rw_el_est()` described above in Section 2.2.5) and a function `r()` 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 involving the discrete solution or its derivative this flag has to be `INIT_UH` and/or `INIT_GRD_UH`, if needed by `r()`, compare Example 4.9.1. Finally, for inhomogeneous Neumann or Robin boundary conditions we must pass a pointer to yet another function `est_gn()` to `ellipt_est()` which describes the inhomogeneity. The information about which boundaries are subject to Dirichlet boundary conditions is provided through the bit-mask `dirichlet_mask`, which is passed to `ellipt_est()`, compare Section 3.2.4.

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()`. The degrees for quadrature formulas are chosen according to the degree of finite element basis functions. Additionally, as the exact solution for our test problem is known (defined by `u()` and `grd_u()`), the true error between discrete and exact solutions is calculated by the function `H1_err()` or `L2_err()`, and the ratio of the true and estimated errors is printed (which should be approximately constant). The experimental orders of convergence of the estimated and exact errors are calculated, which should both be, when using global refinement with d bisection refinements, `fe_space->bas_fcts->degree` for the H^1 norm and `fe_space->bas_fcts->degree+1` for the L^2 norm. Finally, the error indicators are displayed by calling `graphics()`.

```
static REAL r(const EL_INFO *el_info, const QUAD *quad, int iq,
              REAL uh_at_qp, const REAL_D grd_uh_at_qp)
{
  REAL_D x;

  coord_to_world(el_info, quad->lambda[iq], x);

  return -f(x);
}

static REAL est_gn(const EL_INFO *el_info,
                   const QUAD *quad,
                   int qp,
                   REAL uh_at_qp,
                   const REAL_D normal)
{
  /* we simply return gn(), exploiting the fact that the geometry cache
   * of the quadrature already contains the world-coordinates of the
   * quadrature points.
   */
  const QUAD_EL_CACHE *qelc =
    fill_quad_el_cache(el_info, quad, FILL_EL_QUAD_WORLD);

  if (robin_alpha > 0.0) {
    return gn(qelc->world[qp], normal) - robin_alpha * uh_at_qp;
  } else {
    return gn(qelc->world[qp], normal);
  }
}

#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    norm = -1;
  static REAL   C[3] = {1.0, 1.0, 0.0};
  static REAL   est_old = -1.0, err_old = -1.0;
  REAL          est, err;
  REAL_DD       A = {{0.0}};
  int           n;

  for (n = 0; n < DIM_OF_WORLD; n++) {
    A[n][n] = 1.0; /* set diagonal of A; all other elements are zero */
  }

  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]);
  }

  est = ellipt_est(u_h, adapt, rw_el_est, NULL /* rw_est_c() */,
                   -1 /* quad_degree */,
                   norm, C,
                   (const REAL_D *) A,
                   dirichlet_mask,
                   r, 0 /* (INIT_UH | INIT_GRD_UH), if needed by r() */,
                   est_gn, robin_alpha > 0.0 ? INIT_UH : 0);

  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, NULL /* quad */,
                 false /* relative error*/,
                 pure_neumann /* mean-value adjust */,
                 NULL /* rw_err_el()*/, NULL /* max_err_el2 */);
  else
    err = H1_err(grd_u, u_h, NULL /* quad */,
                 false /* relative error */,
                 NULL /* rw_err_el()*/, NULL /* max_err_el2 */);

  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));

  if (do_graphics) {
```

```
    MSG("Displaying the estimate.\n");
    graphics(mesh, NULL /* u_h */, get_el_est, NULL /* u_exact() */,
             HUGE_VAL /* time */);
  }

  return adapt->err_sum;
}
```

2.3 Nonlinear reaction–diffusion equation

In this section, we discuss the implementation of a stationary, nonlinear problem. Due to the nonlinearity, the computation of the discrete solution is more complex. The solver for the nonlinear reaction–diffusion equation and the solver for Poisson equation, described in Section 2.2, thus mainly differ in the routines `build()` and `solve()`.

Here we describe the solution by a Newton method, which involves the assemblage and solution of a linear system in each iteration. Hence, we do not split the assemble and solve routines in `build()` and `solve()` as in the solver for the Poisson equation (compare Sections 2.2.7 and 2.2.8), but only set Dirichlet boundary values for the initial guess in `build()` and solve the nonlinear equation (including the assemblage of linearized systems) in `solve()`. The actual solution process is implemented by several subroutines in the separate file `nlsolve.c`, see Sections 2.3.5 and 2.3.6.

Additionally we describe a simple way to handle different problem data easily, see Sections 2.3.1 and 2.3.8.

We consider the following nonlinear reaction–diffusion equation:

$$-k\Delta u + \sigma\, u^4 = f + \sigma\, u_{ext}^4 \qquad \text{in } \Omega \subset \mathbb{R}^d, \tag{2.1a}$$

$$u = g \qquad \text{on } \partial\Omega. \tag{2.1b}$$

For $\Omega \subset \mathbb{R}^2$, this equation models the heat transport in a thin plate Ω which radiates heat and is heated by an external heat source f. Here, k is the constant heat conductivity, σ the Stefan–Boltzmann 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 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:

$$\Omega = (-1,1)^2,\ k = 2,\ \sigma = 5.67\text{e-}8,\ g \equiv 300,\ u_{ext} = 273,\ f(x) = \begin{cases} 150, & \text{if } x \in (-\frac{1}{2},\frac{1}{2})^2, \\ 0, & \text{otherwise.} \end{cases}$$

2.3.1 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 `alberta.h` header file on the first line. This file is included by all three source files.

```
typedef struct prob_data PROB_DATA;
struct prob_data
{
  MACRO_DATA   *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, REAL_D input);
};

/*--- file nlprob.c ------------------------------------------------------*/
const PROB_DATA *init_problem(MESH *mesh);

/*--- file nlsolve.c -----------------------------------------------------*/
int nlsolve(DOF_REAL_VEC *, REAL, REAL, REAL (*)(const REAL_D));
```

The data structure `PROB_DATA` yields following information:

data pointer to a macro triangulation object;

k diffusion coefficient (constant heat conductivity);

sigma reaction coefficient (Stefan–Boltzmann 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 `NULL`;

u pointer to a function for evaluating the true solution, if not `NULL` (only for test purpose);

grd_u pointer to a function for evaluating the gradient of the true solution, if not `NULL` (only for test purpose).

The function `init_problem()` initializes problem data, like boundary values, right hand side, etc. which is stored in a `PROB_DATA` structure and reads data of the macro triangulation for the actual problem. The function `nlsolve()` implements the nonlinear solver by a Newton method including the assemblage and solution of the linearized sub-problems.

2.3.2 Global variables

In the main source file for the nonlinear solver `nonlin.c` we use the following global variables:

```
#include "nonlin.h"

#include "alberta-demo.h"          /* proto-types for support functions */

static bool do_graphics = true;    /* global graphics switch          */

static const FE_SPACE  *fe_space;  /* initialized by init_dof_admin() */
static DOF_REAL_VEC    *u_h;       /* initialized by build()          */
static const PROB_DATA *prob_data; /* initialized by main()           */
static BNDRY_FLAGS dirichlet_mask; /* bit-mask for Dirichlet segments */
```

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. The variable `dirichlet_mask` marks those segments on which Dirichlet boundary conditions are imposed, see Section 3.2.4. It is initialized by the `main()` function.

2.3.3 The main program for the nonlinear reaction–diffusion equation

The main program is very similar to the main program of the Poisson problem described in Section 2.2.2.

After initializing the access to the parameter file and processing command-line parameters (see Section 2.1.2), the mesh with the used leaf data is initialized, problem dependent data, including the macro triangulation, are initialized by `init_problem(mesh)` (see Section 2.3.8), a finite element space is allocated, 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; const BAS_FCTS *lagrange; ADAPT_STAT
*adapt; int dim, degree = 1, n_refine;

 /*****************************************************************************
  * first of all, initialize the access to parameters of the init file
  *****************************************************************************/
 parse_parameters(argc, argv, "INIT/nonlin.dat");

 GET_PARAMETER(1, "global refinements", "%d", &n_refine);
 GET_PARAMETER(1, "polynomial degree", "%d", &degree);
 GET_PARAMETER(1, "mesh dimension", "%d", &dim);
 GET_PARAMETER(1, "online graphics", "%d", &do_graphics);
```

```
  BNDRY_FLAGS_ALL(dirichlet_mask); /* Only Dirichlet b.c. supported here */
  /*****************************************************************************
   * init problem dependent data and read macro triangulation
   *****************************************************************************/

  prob_data = init_problem();

  /*****************************************************************************
   *  get a mesh with DOFs and leaf data
   *****************************************************************************/

  mesh = GET_MESH(dim,"Nonlinear problem mesh", prob_data->data, NULL, NULL);

  free_macro_data(prob_data->data);

  init_leaf_data(mesh, sizeof(LEAF_DAT),
 NULL /* refine_leaf_data() */,
 NULL /* coarsen_leaf_data() */);

  lagrange = get_lagrange(mesh->dim, degree);
  TEST_EXIT(lagrange, "no lagrange BAS_FCTS\n");

  fe_space = get_fe_space(mesh, lagrange->name, lagrange, 1, ADM_FLAGS_DFLT);

  global_refine(mesh, n_refine*mesh->dim, FILL_NOTHING);

  /*****************************************************************************
   *  init adapt structure and start adaptive method
   *****************************************************************************/
  adapt = get_adapt_stat(dim, "nonlin", "adapt", 1, NULL);
  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;
}
```

2.3.4 Initialization of leaf data

The functions for initializing 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 Section 2.2.5.

2.3.5 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);
  }

  /* set boundary values */
  dirichlet_bound(u_h, NULL, NULL, dirichlet_mask, prob_data->g);

  return;
}
```

2.3.6 The solve routine

The `solve()` routine solves the nonlinear equation by calling the function `nlsolve()` which is implemented in `nlsolve.c` and described below in Section 2.3.10. After solving the discrete problem, the new discrete solution and true error is displayed via the `graphics()` routine. The true error can be computed only for the first application, where the true solution is known (`prob_data->u()` and `prob_data->grd_u()` are not `NULL`).

```
static void solve(MESH *mesh)
{
  nlsolve(u_h, prob_data->k, prob_data->sigma, prob_data->f, dirichlet_mask);

  if (do_graphics) {
    graphics(mesh, u_h, NULL, prob_data->u, HUGE_VAL /* time */);
  }

  return;
}
```

2.3.7 The estimator for the nonlinear problem

In comparison to the Poisson program, the function `r()` which implements the lower order term in the element residual changes due to the term σu^4 in the differential operator, compare Section 4.9.1. The right hand side $f + \sigma u_{ext}^4$ is already 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 set. The initialization of parameters for the estimator is the same as in Section 2.2.9. 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    r_flag = INIT_UH;
  REAL_DD        A = {{0.0}};
  int            n;

  for (n = 0; n < DIM_OF_WORLD; n++)
    A[n][n] = prob_data->k;  /* set diagonal of A; other elements are zero  */

  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, NULL, degree, norm, C,
                   (const REAL_D *) 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, NULL, 0, NULL, NULL);
    else if (norm == H1_NORM  &&  prob_data->grd_u)
      err = H1_err(prob_data->grd_u, u_h, NULL, 0, NULL, NULL);

    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));
    }
    if (do_graphics) {
      graphics(mesh, NULL, get_el_est, NULL, HUGE_VAL /* time */);
    }

    return adapt->err_sum;
  }
```

2.3.8 Initialization of problem dependent data

The file `nlprob.c` contains all problem dependent data. On the first line, `nonlin.h` is included and then two variables for storing the values of the heat conductivity and the Stefan –Boltzmann constant are declared. These values are used by several functions:

```
#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 number: 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, REAL_D input)
{
  static REAL_D buffer = {};
  REAL         *grd = input ? input : buffer;
  REAL          ux = exp(-10.0*SCP_DOW(x,x));
  int           n;

  for (n = 0;  n < DIM_OF_WORLD; n++)
```

```
    grd[n] = -20.0*x[n]*ux;

  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_OF_WORLD)*ux);
}
```

For the computation of a stable and an unstable (but non-physical) solution, depending on the initial choice of the discrete solution, the following functions are used, which also use a global variable `U0`. Such an unstable solution in 3d is shown in Figure 2.2. Data is given as follows (`problem number: 1`):

```
static REAL U0 = 0.0;

static REAL g_1(const REAL_D x)
{
#if DIM_OF_WORLD == 1
  return(4.0*U0*x[0]*(1.0-x[0]));
#endif
#if DIM_OF_WORLD == 2
  return(16.0*U0*x[0]*(1.0-x[0])*x[1]*(1.0-x[1]));
#endif
#if DIM_OF_WORLD == 3
  return(64.0*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 $\sigma\, u_{ext}^4$. A solution to this problem is depicted in Figure 2.3 and problem data is (`problem number: 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);
}
```

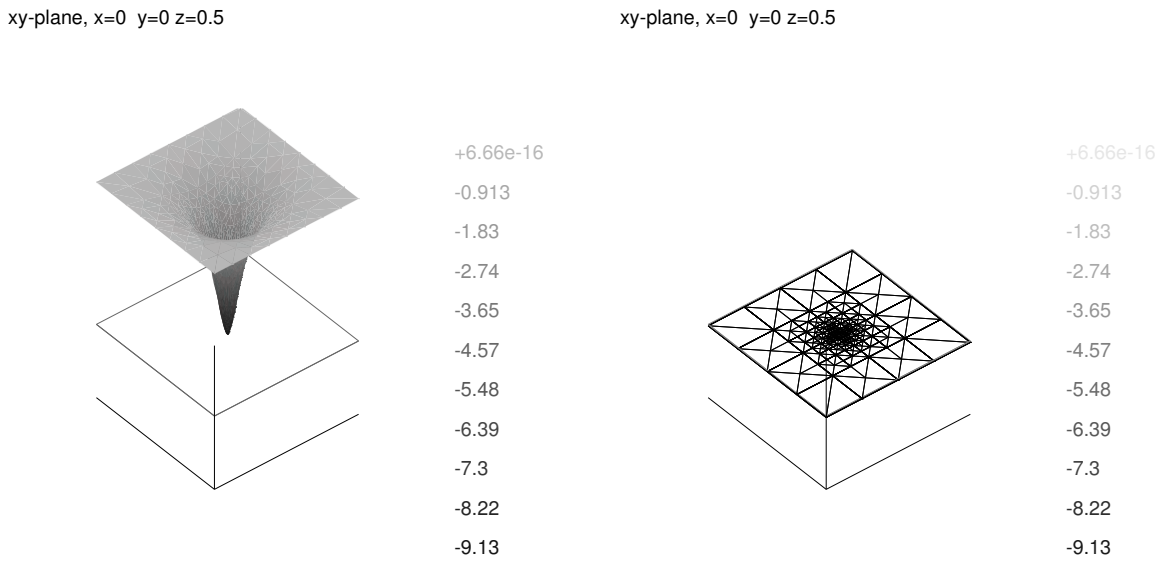


Figure 2.2: Graph of the unstable solution with corresponding mesh of the nonlinear reaction-diffusion problem in 3d on the clipping plane $z = 0.5$. The pictures were produced by the gltools.

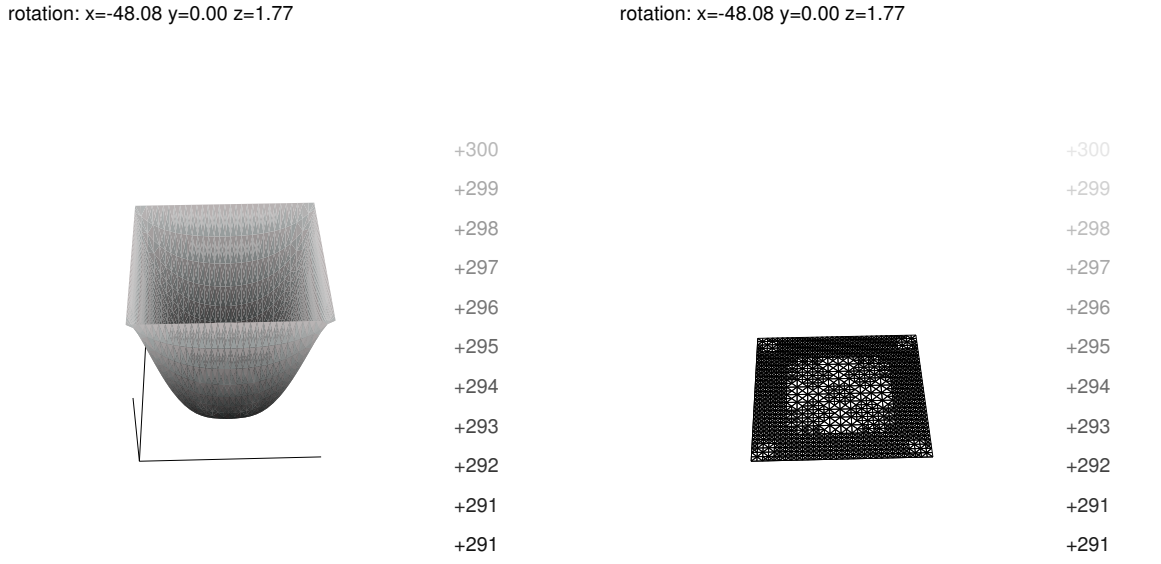


Figure 2.3: Graph of the solution to the physical problem with corresponding mesh of the nonlinear reaction-diffusion problem in 2d. The pictures were produced by the gltools.

Depending on the chosen problem via the parameter `problem number`, the function `init_problem()` initializes the entries of a `PROB_DATA` structure, adjusts the corresponding function pointers, reads the macro triangulation, and returns a pointer to the filled `PROB_DATA`

structure. Information stored in `PROB_DATA` is then used in the `build()` and the `nlsolve()` routines.

```
const PROB_DATA *init_problem(MESH *mesh)
{
  FUNCNAME("init_problem");
  static PROB_DATA prob_data;
  int              pn = 2;

  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;

    prob_data.data = read_macro("Macro/macro-big.amc");
    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(1, "U0", "%f", &U0);

    prob_data.data = read_macro("Macro/macro.amc");
    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;
    prob_data.data = read_macro("Macro/macro-big.amc");
    break;
  default:
    ERROR_EXIT("no problem defined with problem no. %d\n", pn);
  }
  prob_data.k = k;
  prob_data.sigma = sigma;

  return &prob_data;
}
```

2.3.9 The parameter file for the nonlinear reaction–diffusion equation

The following parameter file `INIT/nonlin.dat` is read by `main()` for 2d.

```
mesh dimension:       2
problem number:       2
global refinements:   1
polynomial degree:    2

online graphics:      false

U0:     -5.0          % height of initial guess for Problem 1

% graphic windows: solution, estimate, mesh, and error if size > 0
graphic windows:         500 500 0 0
% for gltools graphics you can specify the range for the values of
% discrete solution for displaying:  min max
% automatical scaling by display routine if min >= max
gltools range:  1.0 0.0

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:         2   % 0: no precon 1: diag precon
                                  % 2: HB precon 3: BPX 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-2
adapt->MS_gamma:        0.5
adapt->max_iteration:   15
adapt->info:            4

WAIT: 1
```

Besides the parameters for the Newton solver and the height of the initial guess `U0` in Problem 1, the file is very similar to the parameter file `ellipt.dat` for the Poisson problem, compare Section 2.2.3. As mentioned above, additional parameters may be defined or overwritten by command line arguments, see Section 2.3.3.

2.3.10 Implementation of the nonlinear solver

In this section, we now describe the solution of the nonlinear problem which differs most from the solver for the Poisson equation. It is the last module missing for the adaptive solver. We use the abstract Newton methods of Section 4.10.9 for solving

$$u_h \in g_h + \mathring{X}_h : \qquad F(u_h) = 0 \qquad \text{in } \mathring{X}_h^*,$$

where $g_h \in X_h$ is an approximation to boundary data g. Using the classical Newton method, 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 2.3.5). For $m \geq 0$ we 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 $u_m \in g_h + \mathring{X}_h$, $m \geq 0$. Newton methods with step size control solve similar defect equations and perform similar update steps, compare Section 4.10.9.

For $v \in g_h + \mathring{X}_h$ the functional $F(v) \in \mathring{X}_h^*$ of the nonlinear reaction–diffusion equation is defined by

$$\big\langle F(v),\, \varphi_j \big\rangle_{\mathring{X}_h^* \times \mathring{X}_h} = \int_\Omega k \nabla\varphi_j \nabla v + \sigma\, \varphi_j\, v^4\, dx - \int_\Omega (f + u_{ext}^4)\varphi_j\, dx \qquad \text{for all } \varphi_j \in \mathring{X}_h, \quad (2.2)$$

and the Frechet derivative $DF(v)$ of F is given by

$$\big\langle DF(v)\, \varphi_i,\, \varphi_j \big\rangle_{\mathring{X}_h^* \times \mathring{X}_h} = \int_\Omega k \nabla\varphi_j \nabla\varphi_i + 4\sigma\, v^3\, \varphi_j\, \varphi_i\, dx \qquad \text{for all } \varphi_i,\, \varphi_j \in \mathring{X}_h. \qquad (2.3)$$

The Newton solvers need a function for assembling the right hand side vector of the discrete system (2.2), and the system matrix of the linearized equation (2.3) 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. For $v \not\geq 0$ BiCGStab is used. We choose the H^1 semi–norm as problem dependent norm $\|.\|$.

2.3.10.1 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_BD  Lambda;          /*  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              */
  BNDRY_FLAGS      dirichlet_mask;

  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 BiCGStab */
  REAL       tolerance;
  REAL      ssor_omega;
  int        max_iter;
  int       ssor_iter;
  int        ilu_k;
  int        restart;
  int        info;
  OEM_PRECON icon;
  const PRECON *precon;
};
```

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.

2.3.10.2 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.

$$A_{ij} = \begin{cases} \int_\Omega k \nabla\varphi_j \nabla\varphi_i \, dx & i = 0, \ldots, \mathring{N},\ j = 0, \ldots, N, \\ \delta_{ij} & i = \mathring{N}+1, \ldots, N,\ j = 0, \ldots, N, \end{cases}$$

and $\boldsymbol{M} = \boldsymbol{M}(v)$ the mass matrix, i.e.

$$M_{ij} = \begin{cases} \int_\Omega \sigma\, v^3\, \varphi_j\, \varphi_i \, dx & i = 0, \ldots, \mathring{N},\ j = 0, \ldots, N, \\ 0 & i = \mathring{N}+1, \ldots, N,\ j = 0, \ldots, N. \end{cases}$$

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

$$\begin{aligned} F_j &= \int_\Omega k\nabla v\nabla\varphi_j + \sigma\, v^4\, \varphi_j\, dx - \int_\Omega (f + \sigma u_{ext}^4)\varphi_j\, dx \\ &= (\boldsymbol{A}\,\boldsymbol{v} + \boldsymbol{M}\,\boldsymbol{v})_j - \int_\Omega (f + \sigma u_{ext}^4)\varphi_j\, dx, \end{aligned} \tag{2.4}$$

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 (2.4) 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 S 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 very simple routine used for the linear Poisson problem. On the other hand, using ALBERTA 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 very 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 2.4.8). It demonstrates the functionality and flexibility of the ALBERTA tools: the assemblage of complex problems 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. Besides the additional scaling by the heat conductivity k, it is exactly the same as for the Poisson problem. For the nonlinear reaction–diffusion equation 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_B *LALt(const EL_INFO *el_info,
  const QUAD *quad,
  int iq, void *ud)
{
  struct op_data *info = (struct op_data *)ud;
  REAL           fac = info->k*info->det;
  int            i, j, k, dim = el_info->mesh->dim;
  static REAL_BB LALt;
```

```
    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_B *)LALt;
  }

  static REAL c(const EL_INFO *el_info, const QUAD *quad, int iq, void *ud)
  {
    struct op_data *info = (struct op_data *)ud;
    REAL  v3;

    DEBUG_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 inside 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 ALBERTA 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 whether $DF(v)$ should be assembled or not. If `F` is not `NULL`, then $F(v)$ should be assembled and stored in the *vector* `F`. Information about the ALBERTA finite element space, a pointer to a DOF matrix, etc. can be passed to `update()` by the `ud` pointer. The declaration

```
NEWTON_DATA    *data = (NEWTON_DATA *)ud;
```

converts the `void *` pointer `ud` into a pointer `data` to a structure `NEWTON_DATA` which gives access to all information, used for the assembling (see above). This structure is initialized in `nlsolve()` before starting the Newton method.

The `update()` routine contains three 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 ALBERTA functions for the assembling is similar to the initialization in the `build()` routine of the linear Poisson equation (compare Section 2.2.7). 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 ALBERTA routine. A detailed description of this structures and the general assemblage routines for matrices can be found in Section 4.7.2. 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 here, 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`, and 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`. For details about these functions we refer to Sections 3.5.1, 3.3.6, and 1.4.3.

Conversion of the vectors into DOF vectors. The input vector `v` of `update()` is a vector storing the coefficients of the function used for the linearization. It is not a DOF vector, but ALBERTA routines for extracting a local coefficient vector need a DOF vector. Thus, we have to "convert" `v` into some DOF vector `dof_v`. This is done by calls

```
init_dof_real_vec_skel(dof_v, "v", data->fe_space);
distribute_to_dof_real_vec_skel(dof_v, v);
```

We refer the reader to Section 3.7.3 for a more detailed discussion.

In the same way we have to convert `F` to a DOF vector `dof_F` if `F` is not `NULL`.

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()`. Hence v^3 can easily be calculated in `c()` at all quadrature nodes. Routines for evaluating finite element functions and their derivatives are described in detail in Section 4.3.

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 contributions for all non Dirichlet DOFs are computed, and zero Dirichlet boundary values are set for Dirichlet DOFs, if `F` is not `NULL`.

The following sources code listing quotes the entire `update()` sub-routine:

```
static void update(void *ud, int dim, const REAL *v, bool up_DF, REAL *F)
{
  /* Some quantities remembered across calls. Think of this routine
   * like being a "library function" ... The stuff is re-initialized
   * whenever the finite element space changes. We use fe_space->admin
   * to check for changes in the finite element space because
   * DOF_ADMIN's are persisitent within ALBERTA, while fe-space are
   * not.
   */
  static EL_MATRIX_INFO elmi2, elmi0;
  static const DOF_ADMIN *admin = NULL;
  static struct op_data op_data[1]; /* storage for det and Lambda */

  /* Remaining (non-static) variables. */
  const BAS_FCTS  *bas_fcts = NULL;
  int             n_phi;
  int             mesh_dim;
  NEWTON_DATA    *data = (NEWTON_DATA *)ud;
  FLAGS          fill_flag;
  DOF_REAL_VEC   dof_v[1];
  DOF_REAL_VEC   dof_F[1];

/*--------------------------------------------------------------------------*/
/*  init functions for assembling DF(v) and F(v)                            */
/*--------------------------------------------------------------------------*/

  bas_fcts = data->fe_space->bas_fcts;
  n_phi    = bas_fcts->n_bas_fcts;
  mesh_dim = bas_fcts->dim;

  if (admin != data->fe_space->admin) {
    OPERATOR_INFO o_info2 = { NULL, }, o_info0 = { NULL, };
    const QUAD    *quad;

    admin     = data->fe_space->admin;

    quad      = get_quadrature(mesh_dim, 2*bas_fcts->degree-2);
    o_info2.row_fe_space   = data->fe_space;
```

```
    o_info2.quad[2]        = quad;
    o_info2.LALt.real      = LALt;
    o_info2.LALt_pw_const  = true;
    o_info2.LALt_symmetric = true;
    o_info2.user_data      = op_data;

    fill_matrix_info(&o_info2, &elmi2);

    o_info0.row_fe_space   = data->fe_space;
    o_info0.quad[0]        = quad;
    o_info0.c.real         = c;
    o_info0.c_pw_const     = false;
    o_info0.user_data      = op_data;

    fill_matrix_info(&o_info0, &elmi0);

    op_data->quad_fast = get_quad_fast(bas_fcts, quad, INIT_PHI);
  }

/*--------------------------------------------------------------------------*/
/*  make a DOF vector from input vector v_vec                               */
/*--------------------------------------------------------------------------*/
  init_dof_real_vec_skel(dof_v, "v", data->fe_space);
  distribute_to_dof_real_vec_skel(dof_v, v);

/*--------------------------------------------------------------------------*/
/*  make a DOF vector from F, if not NULL                                   */
/*--------------------------------------------------------------------------*/
  if (F) {
    init_dof_real_vec_skel(dof_F, "F(v)", data->fe_space);
    distribute_to_dof_real_vec_skel(dof_F, F);
  }

/*--------------------------------------------------------------------------*/
/*  and now assemble DF(v) and/or F(v)                                      */
/*--------------------------------------------------------------------------*/
  op_data->k     = data->k;
  op_data->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 : BiCGStab;
    clear_dof_matrix(data->DF);
  }

  if (F)
  {
    dof_set(0.0, dof_F); //!! Seggi
    L2scp_fct_bas(data->f, op_data->quad_fast->quad, dof_F);
    dof_scal(-1.0, dof_F);
  }
```

```
  fill_flag = CALL_LEAF_EL|FILL_COORDS|FILL_BOUND;
  TRAVERSE_FIRST(data->fe_space->mesh, -1, fill_flag) {
    const EL_REAL_VEC  *v_loc;
    const EL_DOF_VEC   *dof;
    const EL_BNDRY_VEC *bndry_bits;
    EL_SCHAR_VEC bound[n_phi];
    const EL_MATRIX *elmat2, *elmat0;

    v_loc      = fill_el_real_vec(NULL, el_info->el, dof_v);
    dof        = get_dof_indices(NULL, data->fe_space, el_info->el);
    bndry_bits = get_bound(NULL, bas_fcts, el_info);

/*--------------------------------------------------------------------------*/
/* initialization of values used by LALt and c                              */
/*--------------------------------------------------------------------------*/

    op_data->det = el_grd_lambda_0cd(el_info, op_data->Lambda);

    op_data->v_qp = uh_at_qp(NULL, op_data->quad_fast, v_loc);

    elmat2 = elmi2.el_matrix_fct(el_info, elmi2.fill_info);

    elmat0 = elmi0.el_matrix_fct(el_info, elmi0.fill_info);

    /* Translate the geometric boundary classification into
     * Dirichlet/Neumann/Interior boundary condition
     * interpretation. Inside the loop over the mesh-elements we need
     * only to care about Dirichlet boundary conditions.
     */
    dirichlet_map(bound, bndry_bits, data->dirichlet_mask);

    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, elmat2, NoTranspose, dof, dof, bound);
      add_element_matrix(data->DF, 4.0, elmat0, NoTranspose, dof, dof, bound);
    }

    if (F)     /*---  add element contribution to F(v)  --------------------*/
    {
      int i;
/*--------------------------------------------------------------------------*/
/*  F(v) += a(v, phi_i) + m(v^4, phi_i)                                     */
/*--------------------------------------------------------------------------*/
      bi_mat_el_vec(1.0, elmat2, 1.0, elmat0, v_loc, 1.0, dof_F, dof, bound);
      for (i = 0; i < n_phi; i++) {
if (bound->vec[i] >= DIRICHLET) {
  F[dof->vec[i]] = 0.0;  /*--- zero Dirichlet boundary conditions! -*/
}
      }
    }
```

```
  } TRAVERSE_NEXT();

  /* Record that the boundary conditions are built into the matrix, needed
   * e.g. by the hierarchical preconditioners.
   */
  BNDRY_FLAGS_CPY(data->DF->dirichlet_bndry, data->dirichlet_mask);
```

2.3.10.3 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 2.2.8). 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 and 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 = (NEWTON_DATA *)ud;
  int           iter;
  DOF_REAL_VEC  dof_F[1];
  DOF_REAL_VEC  dof_d[1];

/*--------------------------------------------------------------------------*/
/*  make DOF vectors from F and d                                           */
/*--------------------------------------------------------------------------*/
  init_dof_real_vec_skel(dof_F, "F", data->fe_space);
  distribute_to_dof_real_vec_skel(dof_F, F);

  init_dof_real_vec_skel(dof_d, "d", data->fe_space);
  distribute_to_dof_real_vec_skel(dof_d, d);


  if (data->icon == ILUkPrecon)
  data->precon = init_oem_precon(data->DF, NULL, data->info, ILUkPrecon, data->ilu_k);
  else
  data->precon = init_oem_precon(data->DF, NULL, data->info, data->icon,
                                 data->ssor_omega, data->ssor_iter);
  iter = oem_solve_s(data->DF, NULL, dof_F, dof_d, data->solver,
     data->tolerance, data->precon, data->restart,
     data->max_iter, data->info);

  return iter;
}
```

2.3.10.4 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 ALBERTA routine `H1_norm_uh()` (compare Section 4.4).

```
static REAL norm(void *ud, int dim, const REAL *v)
```

```
{
  NEWTON_DATA   *data = (NEWTON_DATA *)ud;
  DOF_REAL_VEC  dof_v[1]; /* = {NULL, NULL, "v"};*/

  init_dof_real_vec_skel(dof_v, "v", data->fe_space);
  distribute_to_dof_real_vec_skel(dof_v, v);

  return H1_norm_uh(NULL, dof_v);
}
```

2.3.10.5 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 (the problem dependent routines `update()`, `solve()`, and `norm()` described above). All these routines use the same structure `NEWTON_DATA` for problem dependent information.

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 call to `dof_compress()` in the `build()` routine, `dim` holds the true dimension of the discrete equation. Without a `dof_compress()` there may be holes in DOF vectors, 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 ALBERTA routines do not operate on unused indices, whereas the Newton solvers do operate on unused indices too, because they do not know about used and unused indices. In this situation, all unused DOFs would 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 4.10.9) and start the Newton method.

```
int nlsolve(DOF_REAL_VEC *u0, REAL k, REAL sigma, REAL (*f)(const REAL_D),
    const BNDRY_FLAGS dirichlet_mask)
{
  FUNCNAME("nlsolve");
  static NEWTON_DATA  data =
    { NULL, { 0, }, 0, 0, NULL, NULL, CG, 1.e-8, 1.0, 1000, 1, 8, 0, 2, 0, NULL };
  static NLS_DATA     nls_data;
  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, NULL);

    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);
    if (data.icon == __SSORPrecon) {
     GET_PARAMETER(1, "linear precon ssor omega", "%f", &data.ssor_omega);
     GET_PARAMETER(1, "linear precon ssor iter", "%d", &data.ssor_iter);
    }
    if (data.icon == ILUkPrecon)
        GET_PARAMETER(1, "linear precon ilu(k)", "%d", &data.ilu_k);
    GET_PARAMETER(1, "linear solver restart", "%d", &data.restart);
  }
  TEST_EXIT(data.fe_space == u0->fe_space, "can't change f.e. spaces\n");
  BNDRY_FLAGS_CPY(data.dirichlet_mask, dirichlet_mask);

/*--------------------------------------------------------------------------*/
/*--  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;
}
```

2.4 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 2.2.

2.4.1 Global variables

Additionally to the finite element space `fe_space`, the matrix `matrix`, the vectors `u_h` and `f_h` and the bit-mask `dirichlet_mask` for marking Dirichlet boundary-segments, 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;
```

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;
```

Finally, a global variable `theta` is used for storing the parameter θ and `err_L2` for storing the actual L^2 error between true and discrete solution in the actual time step.

```
static REAL theta = 0.5;    /*---  parameter of the time discretization
  ---*/
static REAL err_L2  = 0.0; /*---  spatial error in a single time step
  ---*/
```

2.4.2 The main program for the heat equation

The main function initializes all program parameters from file and command line (compare Section 2.1.2), generates a mesh and a finite element space, the DOF matrix and vectors, and

allocates and fills the parameter structure `ADAPT_INSTAT` for the adaptive method for time dependent problems. This structure is accessed by `get_adapt_instat()` which already initializes besides the function pointers all members of this structure from the program parameters, compare Sections 4.8.4 and 2.2.2.

The (initial) time step size, read from the 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) and space discretization. For stability reasons, the time step size is scaled by a factor 10^{-3} if $\theta < 0.5$, see also Section 2.4.6.

Finally, the function pointers for the `adapt_instat()` structure are adjusted to the problem dependent routines for the heat equation and the complete numerical simulation is performed by a call to `adapt_method_instat()`.

The `heat.c` demo-program only implements Dirichlet boundary conditions by setting all bits of `dirichlet_mask` to 1. The implementation of more complicated boundary conditions is exemplified in the explanation for the `ellipt.c` program, see Section 2.2.

```
int main(int argc, char **argv)
{
  FUNCNAME("main");
  MACRO_DATA      *data;
  MESH            *mesh;
  const BAS_FCTS  *lagrange;
  int              n_refine = 0, p = 1, dim;
  char             filename[PATH_MAX];
  REAL             fac = 1.0;

  /*****************************************************************************
   * first of all, initialize the access to parameters of the init file
   *****************************************************************************/
  parse_parameters(argc, argv, "INIT/heat.dat");

  GET_PARAMETER(1, "mesh dimension", "%d", &dim);
  GET_PARAMETER(1, "macro file name", "%s", filename);
  GET_PARAMETER(1, "global refinements", "%d", &n_refine);
  GET_PARAMETER(1, "parameter theta", "%e", &theta);
  GET_PARAMETER(1, "polynomial degree", "%d", &p);
  GET_PARAMETER(1, "online graphics", "%d", &do_graphics);

  /*****************************************************************************
  *  get a mesh, and read the macro triangulation from file
  *****************************************************************************/
  data = read_macro(filename);
  mesh = GET_MESH(dim, "ALBERTA mesh", data,
                  NULL /* init_node_projection() */,
                  NULL /* init_wall_trafos() */);
  free_macro_data(data);

  init_leaf_data(mesh, sizeof(struct heat_leaf_data),
                 NULL /* refine_leaf_data() */,
                 NULL /* coarsen_leaf_data() */);

  /* Finite element spaces can be added at any time, but it is more
   * efficient to do so before refining the mesh a lot.
   */
  lagrange = get_lagrange(mesh->dim, p);
  TEST_EXIT(lagrange, "no lagrange BAS_FCTS\n");
```

```
fe_space = get_fe_space(mesh, lagrange->name, lagrange, 1, ADM_FLAGS_DFLT);

global_refine(mesh, n_refine * dim, FILL_NOTHING);

/*****************************************************************************
 *  initialize the global variables shared across build(), solve()
 *  and estimate().
 *****************************************************************************/
matrix = get_dof_matrix("A", fe_space, NULL);
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  !                              */

BNDRY_FLAGS_ALL(dirichlet_mask); /* Only Dirichlet b.c. supported here */

/*****************************************************************************
 *  init adapt_instat structure
 *****************************************************************************/
adapt_instat = get_adapt_instat(dim, "heat", "adapt", 2, adapt_instat);

/* Some animation in between ... */
if (do_graphics) {
  graphics(mesh, NULL, NULL, NULL, adapt_instat->start_time);
}

/*****************************************************************************
 *  adapt time step size to refinement level and polynomial degree,
 *  based on the known L2-error error estimates.
 *****************************************************************************/
if (theta < 0.5) {
  WARNING("You are using the explicit Euler scheme\n");
  WARNING("Use a sufficiently small time step size!!!\n");
  fac = 1.0e-3;
}

if (theta == 0.5) {
  adapt_instat->timestep *= fac*pow(2, -(REAL)(p*(n_refine))/2.0);
} else {
  adapt_instat->timestep *= fac*pow(2, -(REAL)(p*(n_refine)));
}
MSG("using initial timestep size = %.4le\n", adapt_instat->timestep);

eval_time_u0 = adapt_instat->start_time;

adapt_instat->adapt_initial->get_el_est = get_el_est;
adapt_instat->adapt_initial->estimate = est_initial;
adapt_instat->adapt_initial->solve = interpol_u0;

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->set_time       = set_time;
  adapt_instat->get_time_est   = get_time_est;
  adapt_instat->close_timestep = close_timestep;

  /*****************************************************************************
   * ... off we go ...
   *****************************************************************************/
  adapt_method_instat(mesh, adapt_instat);

  WAIT_REALLY;

  return 0;
}
```

2.4.3 The parameter file for the heat equation

The parameter file for the heat equation `INIT/heat.dat` (here for the 2d simulations) is similar to the parameter file for the Poisson problem. The main differences are additional parameters for the adaptive procedure, see Section 4.8.3. These additional parameters may also be optimized for 1d, 2d, and 3d.

Via the parameter `write finite element data` storage of meshes and finite element solution for post-processing purposes can be done. The parameter `write statistical data` selects storage of files containing number of DOFs, estimate, error, etc. versus time. Finally, `data path` can prescribe an existing path for storing such data.

```
mesh dimension:       2
macro file name:      Macro/macro.amc
global refinements:   4
polynomial degree:    1

online graphics:      1

% graphic windows: solution, estimate, mesh, and error if size > 0
graphic windows:       400 400 400 400
% for gltools graphics you can specify the range for the values of
% discrete solution for displaying:  min max
% automatical scaling by display routine if min >= max
gltools range:  -1.0 1.0

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-12
solver info:           2
solver precon:         1   % 0: no precon 1: diag precon
                           % 2: HB precon 3: BPX precon
                           % 4: SSOR, omega = 1.0, #iter = 3
```

```
                          % 5: SSOR, with control over omega and #iter
                          % 6: ILU(k)
precon ssor omega:    1.0 % for precon == 5
precon ssor iter:     1   % for precon == 5
precon ilu(k):        8   % for precon == 6

parameter theta:              1.0
adapt->start_time:            0.0
adapt->end_time:              2.0

adapt->tolerance:             1.0e-3
adapt->timestep:              1.0e-1
adapt->rel_initial_error:     0.5
adapt->rel_space_error:       0.5
adapt->rel_time_error:        0.5
adapt->strategy:              0   % 0=explicit, 1=implicit
adapt->max_iteration:         10
adapt->info:                  2

adapt->initial->strategy:     3    % 0=none, 1=GR, 2=MS, 3=ES, 4=GERS
adapt->initial->MS_gamma:     0.5
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:     0.2
adapt->space->max_iteration:  10
adapt->space->coarsen_allowed: 1   % 0|1
adapt->space->info:           2

estimator C0:                 0.1
estimator C1:                 0.1
estimator C2:                 0.1
estimator C3:                 0.1

write finite element data:    1  % write data for post-processing or not
write statistical data:       0  % write statistical data or not
data path:                    ./data  % path for data to be written

WAIT:                         0
```

Figures 2.4 and 2.5 show the variation of time step sizes and number of DOFs over time, automatically generated by the adaptive method in two and three space dimensions for a problem with time-periodic data. The number of DOFs is depicted for different spatial discretization order and shows the strong benefit from using a higher order method. The size of time steps was nearly the same for all spatial discretizations. Parameters for the adaptive procedure can be taken from the corresponding parameter files in 2d and 3d in the distribution.

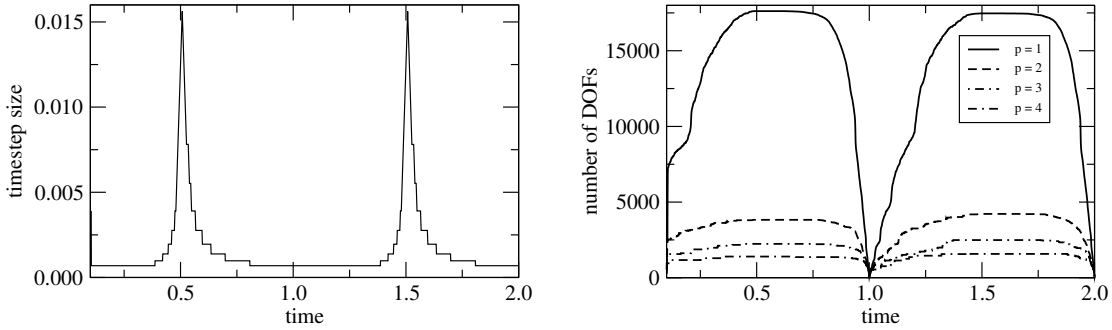


Figure 2.4: Time step size (left) and number of DOFs for different polynomial degree (right) over time in 2d.

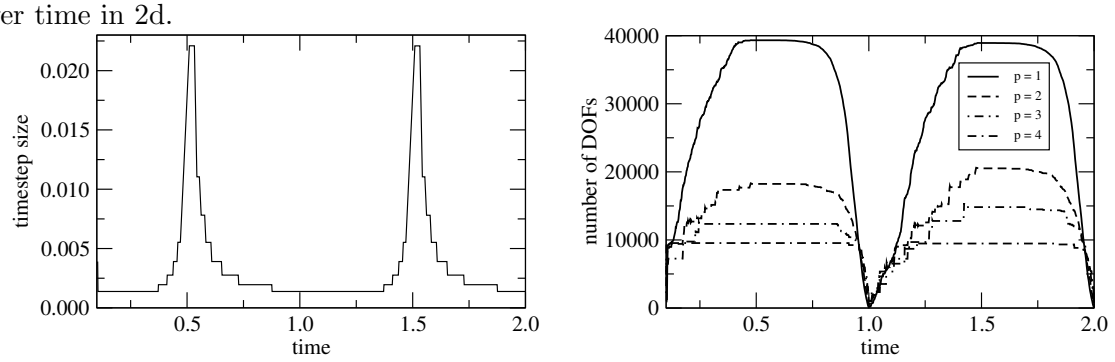


Figure 2.5: Time step size (left) and number of DOFs for different polynomial degree (right) over time in 3d.

2.4.4 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, in addition to the functions `rw_el_est()` and `get_el_est()` which were already defined in `ellipt.c`.

```
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_est(EL *el)
{
  if (IS_LEAF_EL(el))
    return &((struct heat_leaf_data *)LEAF_DATA(el))->estimate;
  else
    return NULL;
}

static REAL get_el_est(EL *el)
{
```

```
  if (IS_LEAF_EL(el))
    return ((struct heat_leaf_data *)LEAF_DATA(el))->estimate;
  else
    return 0.0;
}

static REAL *rw_el_estc(EL *el)
{
  if (IS_LEAF_EL(el))
    return &((struct heat_leaf_data *)LEAF_DATA(el))->est_c;
  else
    return NULL;
}

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;
}
```

2.4.5 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 computing the true 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 eval_time_u0 = 0.0;
static REAL u0(const REAL_D x)
{
  eval_time_u = eval_time_u0;
  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)                /* -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 depending on the kind of time discretization. The value of `eval_time_f` and `eval_time_g` are set by the function `set_time()`. Similarly, the evaluation time for the exact solution is set by `estimate()` where also the evaluation time of f is set for the evaluation of the element residual. In order to start the simulation not only at $t = 0$, we have introduced a variable `eval_time_u0`, which is set in `main()` at the beginning of the program to the value of `adapt_instat->start_time`.

2.4.6 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

$$\begin{aligned} \frac{1}{\tau_{n+1}}(U_{n+1}, \Phi) + \theta(\nabla U_{n+1}, \nabla\Phi) = &\frac{1}{\tau_{n+1}}(I_{n+1}U_n, \Phi) - (1-\theta)(\nabla I_{n+1}U_n, \nabla\Phi) \\ &+ (f(\cdot, t_n + \theta\tau_{n+1}), \Phi) \qquad \text{for all } \Phi \in \mathring{X}_{n+1}. \end{aligned} \tag{2.5}$$

For $\theta = 0$, this is the forward (explicit) Euler scheme, for $\theta = 1$ the backward (implicit) Euler scheme. For $\theta = 0.5$, we obtain 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. For that reason, the time step size is scaled by an additional factor of 10^{-3} in the main program if $\theta < 0.5$. But this might not be enough for guaranteeing stability of the scheme! We do recommend to use $\theta = 0.5, 1$ only.

2.4.7 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;
}
```

2.4.8 The assemblage of the discrete system

Using a matrix notation, the discrete problem (2.5) 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, $\boldsymbol{M} = (\Phi_i, \Phi_j)$ denotes the mass matrix and $\boldsymbol{A} = (\nabla\Phi_i, \nabla\Phi_j)$ the stiffness matrix (up to Dirichlet boundary DOFs). The system matrix on the left hand side is not the same as the one applied to the old solution on the right hand side. But we want to compute the contribution of the solution form the old time step U_n to the right hand side vector efficiently by a simple matrix–vector multiplication and thus avoiding additional element-wise integration. For doing this without storing both matrices $\boldsymbol{M}$ and $\boldsymbol{A}$ we are using the element-wise strategy explained and used in Section 2.3.6 when assembling the linearized equation in the Newton iteration for solving the nonlinear reaction–diffusion equation.

The subroutine `assemble()` generates both the system matrix and the right hand side at the same time. 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 2.2.7; the mass operator implements only the constant zero order coefficient $c = 1/\tau_{n+1}$, which is passed in `struct op_data` and evaluated in the function `c()`. The initialization and access to these operators is done in the same way as in Section 2.3.6 where this is described in detail. During the non-recursive mesh traversal, the element stiffness matrix and the mass matrix are computed and added to the global system matrix. Then, the contribution to the right hand side vector of the solution from the old time step is computed by a matrix–vector product of these element matrices with the local coefficient vector on the element of U_n and added to the global load vector (see Table 4.2 for `bi_mat_el_vec()`).

After this step, the the right hand side f and Dirichlet boundary values g are treated by the standard routines.

2.4.1 Compatibility Note. *In contrast to previous ALBERTA versions, the element-vectors and -matrices are no longer flat* `C`*-arrays, but "cooked" data-structures, with some support routines doing basis linear algebra. See Section 4.7.1.1.*

```
struct op_data          /* application data (resp. "user_data") */
{
  REAL_BD Lambda;       /*  the gradient of the barycentric coordinates */
  REAL    det;          /*  |det D F_S|                                  */
```

```
  REAL     tau_1;
};

static REAL c(const EL_INFO *el_info, const QUAD *quad, int iq, void *ud)
{
  struct op_data *info = (struct op_data *)ud;

  return info->tau_1*info->det;
}

static void assemble(DOF_MATRIX *matrix, DOF_REAL_VEC *fh, DOF_REAL_VEC *uh,
                     const DOF_REAL_VEC *u_old,
                     REAL theta, REAL tau,
                     REAL (*f)(const REAL_D), REAL (*g)(const REAL_D),
                     const BNDRY_FLAGS dirichlet_mask)
{
  /* Some quantities remembered across calls. Think of this routine
   * like being a "library function" ... The stuff is re-initialized
   * whenever the finite element space changes. We use fe_space->admin
   * to check for changes in the finite element space because
   * DOF_ADMIN's are persisitent within ALBERTA, while fe-space are
   * not.
   */
  static EL_MATRIX_INFO stiff_elmi, mass_elmi;
  static const DOF_ADMIN *admin = NULL;
  static const QUAD *quad = NULL;
  static struct op_data op_data[1]; /* storage for det and Lambda */

  /* Remaining (non-static) variables. */
  const BAS_FCTS *bas_fcts;
  FLAGS           fill_flag;
  REAL            *f_vec;
  int             nbf;
  EL_SCHAR_VEC    *bound;

  /* Initialize persistent variables. */
  if (admin != uh->fe_space->admin) {
    OPERATOR_INFO stiff_opi = { NULL, }, mass_opi = { NULL, };

    admin     = uh->fe_space->admin;

    stiff_opi.row_fe_space   = uh->fe_space;
    stiff_opi.quad[2]        = quad;
    stiff_opi.LALt.real      = LALt;
    stiff_opi.LALt_pw_const  = true;
    stiff_opi.LALt_symmetric = true;
    stiff_opi.user_data      = op_data;

    fill_matrix_info(&stiff_opi, &stiff_elmi);

    mass_opi.row_fe_space = uh->fe_space;
    mass_opi.quad[0]      = quad;
    mass_opi.c.real       = c;
    mass_opi.c_pw_const   = true;
    mass_opi.user_data    = op_data;
```

```
    fill_matrix_info(&mass_opi, &mass_elmi);

    quad = get_quadrature(uh->fe_space->bas_fcts->dim,
                          2*uh->fe_space->bas_fcts->degree);
  }

  op_data->tau_1 = 1.0/tau;

  /* Assemble the matrix and the right hand side. The idea is to
   * assemble the local mass and stiffness matrices only once, and to
   * use it to update both, the system matrix and the contribution of
   * the time discretisation to the RHS.
   */
  clear_dof_matrix(matrix);
  dof_set(0.0, fh);
  f_vec = fh->vec;

  bas_fcts = uh->fe_space->bas_fcts;
  nbf      = bas_fcts->n_bas_fcts;

  bound = get_el_schar_vec(bas_fcts);

  fill_flag = CALL_LEAF_EL|FILL_COORDS|FILL_BOUND;
  TRAVERSE_FIRST(uh->fe_space->mesh, -1, fill_flag) {
    const EL_REAL_VEC  *u_old_loc;
    const EL_DOF_VEC   *dof;
    const EL_BNDRY_VEC *bndry_bits;
    const EL_MATRIX *stiff_loc, *mass_loc;

    /* Get the local coefficients of u_old, boundary info, dof-mapping */
    u_old_loc  = fill_el_real_vec(NULL, el_info->el, u_old);
    dof        = get_dof_indices(NULL, uh->fe_space, el_info->el);
    bndry_bits = get_bound(NULL, bas_fcts, el_info);

    /* Initialization of values used by LALt and c. It is not
     * necessary to introduce an extra "init_element()" hook for our
     * OPERATOR_INFO structures; the line below is just what would be
     * contained in that function (compare with ellipt.c).
     *
     * Beware to replace the "..._0cd()" for co-dimension 0 by its
     * proper ..._dim() variant if ever "porting" this stuff to
     * parametric meshes on manifolds.
     */
    op_data->det = el_grd_lambda_0cd(el_info, op_data->Lambda);

    /* Obtain the local (i.e. per-element) matrices. */
    stiff_loc = stiff_elmi.el_matrix_fct(el_info, stiff_elmi.fill_info);
    mass_loc  = mass_elmi.el_matrix_fct(el_info, mass_elmi.fill_info);

    /* Translate the geometric boundary classification into
     * Dirichlet/Neumann/Interior boundary condition
     * interpretation. Inside the loop over the mesh-elements we need
     * only to care about Dirichlet boundary conditions.
     */
    dirichlet_map(bound, bndry_bits, dirichlet_mask);

    /* add theta*a(psi_i,psi_j) + 1/tau*m(4*u^3*psi_i,psi_j) */
```

```
    if (theta) {
      add_element_matrix(matrix,
                         theta, stiff_loc, NoTranspose, dof, dof, bound);
    }
    add_element_matrix(matrix, 1.0, mass_loc, NoTranspose, dof, dof, bound);

      /* compute the contributions from the old time-step:
       *
       * f += -(1-theta)*a(u_old,psi_i) + 1/tau*m(u_old,psi_i)
       */
    bi_mat_el_vec(-(1.0 - theta), stiff_loc,
                  1.0, mass_loc, u_old_loc,
                  1.0, fh, dof, bound);

  } TRAVERSE_NEXT();

  free_el_schar_vec(bound);

  /* Indicate that the boundary conditions are built into the matrix,
   * needed e.g. by the hierarchical preconditioners.
   */
  BNDRY_FLAGS_CPY(matrix->dirichlet_bndry, dirichlet_mask);

  /* Add the "force-term" to the right hand side (L2scp_...() is additive) */
  L2scp_fct_bas(f, quad, fh);

  /* Close the system by imposing suitable boundary conditions. Have a
   * look at ellipt.c for how to impose more complicated stuff; here
   * we only use Dirichlet b.c.
   */
  dirichlet_bound(fh, uh, NULL, dirichlet_mask, g);
}
```

The `build()` routine for one time step of the heat equation is nearly a dummy routine and just calls the `assemble()` routine described above. In order to avoid holes in vectors and matrices, as a first step, the mesh is compressed. This guarantees optimal performance of the BLAS1 routines used in the iterative solvers.

```
static void build(MESH *mesh, U_CHAR flag)
{
  FUNCNAME("build");

  dof_compress(mesh);

  INFO(adapt_instat->adapt_space->info, 2,
    "%d DOFs for %s\n", fe_space->admin->size_used, fe_space->name);

  assemble(matrix, f_h, u_h, u_old, theta, adapt_instat->timestep,
           f, g, dirichlet_mask);
}
```

The resulting linear system is solved by calling the `oem_solve_s()` library routine. This is done via the `solve()` subroutine described in Section 2.2.8.

2.4.9 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)
{
  err_L2 = adapt->err_sum =
    L2_err(u0, u_h, NULL, false, false, 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 2.4.9. Similar to the estimator for elliptic problems, a function `r()` is needed for computing contributions of lower order terms and the right hand side. The flag for passing information about the discrete solution U_{n+1} or its gradient to `r()` is set to zero in `estimate()` since no lower order term is involved.

Local element indicators are stored to the `estimate` or `est_c` entries inside the data structure `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()`. In this example, the true solution is known and thus the true error $\|u(\cdot, t_{n+1}) - U_{n+1}\|_{L^2(\Omega)}$ is calculated additionally for comparison.

```
static REAL r(const EL_INFO *el_info,
              const QUAD *quad, int iq,
              REAL uh_at_qp, const REAL_D grd_uh_at_qp,
              REAL t)
{
  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 REAL    C[4] = {-1.0, 1.0, 1.0, 1.0};
  static REAL_DD A = {{0.0}};
  FLAGS          r_flag = 0;  /* = (INIT_UH|INIT_GRD_UH), if needed by r()
      */
  int            n;
  REAL           space_est;

  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]);

    for (n = 0; n < DIM_OF_WORLD; n++) {
      A[n][n] = 1.0;    /* set diogonal of A; all other elements are zero */
    }
  }

  time_est = heat_est(u_h, u_old, adapt_instat, rw_el_est, rw_el_estc,
                      -1 /* quad_degree */,
                      C, (const REAL_D *)A, dirichlet_mask,
                      r, r_flag, NULL /* gn() */, 0 /* gn_flag */);

  space_est = adapt_instat->adapt_space->err_sum;
  err_L2 = L2_err(u, u_h, NULL, false, false, NULL, NULL);

  INFO(adapt_instat->info,2,
    "---8<----------------------------------------------------------\n");
  INFO(adapt_instat->info, 2,"time = %.4le with timestep = %.4le\n",
                             adapt_instat->time, adapt_instat->timestep);
  INFO(adapt_instat->info, 2,"estimate   = %.4le, max = %.4le\n", space_est,
                             sqrt(adapt_instat->adapt_space->err_max));
  INFO(adapt_instat->info, 2,"||u-uh||L2 = %.4le, ratio = %.2lf\n", err_L2,
                             err_L2/MAX(space_est,1.e-20));

  return adapt_instat->adapt_space->err_sum;
}
```

2.4.10 Time steps

Time dependent problems are calculated step by step in single time steps. In a fully implicit time-adaptive strategy, each time step includes an adaptation of the time step size as well as an adaptation of the corresponding spatial discretization. First, the time step size is adapted and then the mesh adaptation procedure is performed. This second part may again push the estimate for the time discretization error over the corresponding tolerance. In this case, the time step size is again reduced and the whole procedure is iterated until both, time and space discretization error estimates meet the prescribed tolerances (or until a maximal number of iterations is performed). For details and other time-adaptive strategies see Section 4.8.3.

Besides the `build()`, `solve()`, and `estimate()` routines for the adaptation of the initial grid and the grids in each time steps, additional routines for initializing time steps, setting time step size of the actual time step, and finalizing a time step are needed. For adaptive control of the time step size, the function `get_time_est()` gives information about the size of the time discretization error. The actual time discretization error is stored in the global variable `time_est` and its value is set in the function `estimate()`.

During the initialization of a new time step in `init_timestep()`, the discrete solution `u_h` from the old time step (or from interpolation of initial data) is copied to `u_old`. In the function `set_time()` evaluation times for the right hand side f and Dirichlet boundary data g are set accordingly to the chosen time discretization scheme. Since a time step can be rejected by the adaptive method by a too large time discretization error, this function can be called several

times during the computation of a single time step. On each call, information about the actual time and time step size is accessed via the entries `time` and `timestep` of the `adapt_instat` structure.

After accepting the current time step size and current grid by the adaptive method, the time step is completed by `close_time_step()`. The variables `space_est`, `time_est`, and `err_L2` now hold the final estimates resp. error, and `u_h` the accepted finite element solution for this time step. The final mesh and discrete solution can now be written to file for post-processing purposes, depending on the parameter value of `write finite element data`. The file name for the mesh/solution consists of the data path, the base name `mesh/u_h`, and the iteration counter of the actual time step. Such a composition can be easily constructed by the function `generate_filename()`, described in Section 3.1.6. Mesh and finite element solution are then exported to file by the `write_*_xdr()` routines in a portable binary format. Using this procedure, the sequence of discrete solutions can easily be visualized by the program `alberta_movi` which is an interface to GRAPE and comes with the distribution of ALBERTA, compare Section 4.11.3.

Depending on the parameter value of `write statistical data`, the evolution of estimates, error, number of DOFs, size of time step size, etc. are written to files by the function `write_statistical_data()`, which is included in `heat.c` but not described here. It produces for each quantity a two-column data file where the first column contains time and the second column estimate, error, etc. Such data can easily be evaluated by standard (graphic) tools.

Finally, a graphical output of the solution and the mesh is generated via the `graphics()` routine already used in the previous examples.

```
static REAL time_est = 0.0;

static REAL get_time_est(MESH *mesh, ADAPT_INSTAT *adapt)
{
  return(time_est);
}

static void init_timestep(MESH *mesh, ADAPT_INSTAT *adapt)
{
  FUNCNAME("init_timestep");

  INFO(adapt_instat->info,1,
    "---8<---------------------------------------------------------\n");
  INFO(adapt_instat->info, 1,"starting new timestep\n");

  dof_copy(u_h, u_old);
  return;
}

static void set_time(MESH *mesh, ADAPT_INSTAT *adapt)
{
  FUNCNAME("set_time");

  INFO(adapt->info,1,
    "---8<---------------------------------------------------------\n");
  if (adapt->time == adapt->start_time)
  {
    INFO(adapt->info, 1,"start time: %.4le\n", adapt->time);
```

```
  }
  else
  {
    INFO(adapt->info, 1,"timestep for (%.4le %.4le), tau = %.4le\n",
                        adapt->time-adapt->timestep, adapt->time,
                        adapt->timestep);
  }

  eval_time_f = adapt->time - (1 - theta)*adapt->timestep;
  eval_time_g = adapt->time;

  return;
}

static void close_timestep(MESH *mesh, ADAPT_INSTAT *adapt)
{
  FUNCNAME("close_timestep");
  static REAL err_max = 0.0;                    /* max space-time error
     */
  static REAL est_max = 0.0;                    /* max space-time estimate
     */
  static int  write_fe_data = 0, write_stat_data = 0;
  static int  step = 0;
  static char path[256] = "./";

  REAL        space_est = adapt->adapt_space->err_sum;
  REAL        tolerance = adapt->rel_time_error*adapt->tolerance;

  err_max = MAX(err_max, err_L2);
  est_max = MAX(est_max, space_est + time_est);

  INFO(adapt->info,1,
    "---8<---------------------------------------------------------\n");

  if (adapt->time == adapt->start_time)
  {
    tolerance = adapt->adapt_initial->tolerance;
    INFO(adapt->info,1,"start time: %.4le\n", adapt->time);
  }
  else
  {
    tolerance += adapt->adapt_space->tolerance;
    INFO(adapt->info,1,"timestep for (%.4le %.4le), tau = %.4le\n",
                       adapt->time-adapt->timestep, adapt->time,
                       adapt->timestep);
  }
  INFO(adapt->info,2,"max. est. = %.4le, tolerance = %.4le\n",
                     est_max, tolerance);
  INFO(adapt->info,2,"max. error = %.4le, ratio = %.2lf\n",
                     err_max, err_max/MAX(est_max,1.0e-20));

  if (!step) {
    GET_PARAMETER(1, "write finite element data", "%d", &write_fe_data);
    GET_PARAMETER(1, "write statistical data", "%d", &write_stat_data);
    GET_PARAMETER(1, "data path", "%s", path);
  }
```

```
  /*****************************************************************************
   * write mesh and discrete solution to file for post-processing
   *****************************************************************************/

  if (write_fe_data) {
    const char *fn;

    fn= generate_filename(path, "mesh", step);
    write_mesh_xdr(mesh, fn, adapt->time);
    fn= generate_filename(path, "u_h", step);
    write_dof_real_vec_xdr(u_h, fn);
  }

  step++;

  /*****************************************************************************
   * write data about estimate, error, time step size, etc.
   *****************************************************************************/

  if (write_stat_data) {
    int n_dof = fe_space->admin->size_used;
    write_statistics(path, adapt, n_dof, space_est, time_est, err_L2);
  }

  if (do_graphics) {
    graphics(mesh, u_h, get_el_est, u, adapt->time);
  }

  return;
}
```

2.5 Installation of ALBERTA and file organization

2.5.1 Installation

The ALBERTA-distribution comes in form a compressed tar archives

`alberta-VERSION.tar.bz2`

or

`alberta-VERSION.tar.gz,`

where `VERSION` has to be replaced by the respective version of the distribution, as you might have guessed. It includes all sources of ALBERTA, the model implementations of the examples described in Chapter 2, and tools for the installation. ALBERTA can only be installed on a Unix-like operating system.

The file `alberta-VERSION.tar.{gz|bz2}` has to be unpacked. This creates a sub-directory `alberta-VERSIONS/` in the current directory with all data of ALBERTA. Changing to this sub-directory, the installation procedure is fully explained in the `README`. For a platform independent installation the GNU configure tools are used, documented in the file `INSTALL`. Installation options for the `configure` script can be added on the command line and are described in the `README` file or printed with the command `configure --help`.

If ALBERTA should use one of the visualization packages gltools, GRAPE, OpenDX, or GMV, these have to be installed first, see the corresponding web sites

```
http://www.wias-berlin.de/software/gltools/
http://www.iam.uni-bonn.de/sfb256/grape/
http://www-xdiv.lanl.gov/XCM/gmv/GMVHome.html
http://www.opendx.org/
```

for obtaining the software. Paths to their installation directories must be passed as arguments to the `configure` script. As a hint: the simplest way is to install add-on packages following the layout advocated by the GNU tools, i.e. libraries go to `PREFIX/lib/`, header files to `PREFIX/include/` and so on, where `PREFIX` stands for a leading path-component common to all installation directories. There is also support for other visualization packages, but the packages mentioned above have to be installed prior to the ALBERTA package, because ALBERTA needs access to header-files and software libraries that come with those packages to use them.

2.5.2 File organization

Using the ALBERTA library, the global dimension n enters in an application only as a symbolic constant `DIM_OF_WORLD`. Thus, the code is usually the same, regardless of the dimension of the ambient space. Nevertheless, the object files do depend on the dimension, since `DIM_OF_WORLD` is preprocessor macro constant (defining 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, which is also reflected in the structure of `DEMO/src/` sub-directory with the implementation of the model problems. We use the sub-directories

```
./1d/ ./2d/ ./3d/ ./4d/ ./5d/ ./Common/
```

for organizing files. The directory `Common/` contains all source files and header files that do not (or only slightly) depend on the dimension. The directories `1d/`, `2d/`, `3d/`, `4d/` and `5d/` contain dimension-dependent data, like macro triangulations files and parameter files. Finally, a dimension-dependent `Makefile` is automatically created in `DEMO/src/*d` during installation of ALBERTA. These `Makefile`s contain all information about ALBERTA header files and all libraries used. In the 1d, 2d, or 3d sub-directories, the programs of the model problems for the corresponding space dimension are then generated by executing `make ellipt`, `make nonlin`, and `make heat`. There are additional demo programs, some of them are tied to a specific value of `DIM_OF_WORLD`. This is described in the file `README` in the top-level directory of the demo-package.

Chapter 3

Data structures and implementation

The ALBERTA toolbox provides two header files `alberta_util.h` and `alberta.h`, which contain the definitions of all data structures, macros, and subroutine prototypes. The file `alberta_util.h` is included in the header file `alberta.h`.

3.1 Basic types, utilities, and parameter handling

The file `alberta_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 `alberta_util.h`

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

3.1.1 Basic types

ALBERTA uses the following elementary symbolic constants and macro definitions:

```
#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 libraries for linear solver or graphics, e.g.).

3.1.2 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 where 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:

3.1.1 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 using message macros.

Usual output to `stdout` is done by the macro `MSG()` which has the same arguments as `printf()`:

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

3.1.2 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
```

A simpler way to print vectors of integer or real numbers is provided by the macros PRINT_INT_VEC and PRINT_REAL_VEC.

```
PRINT_INT_VEC(const char *s, const int *vec, int no);
PRINT_REAL_VEC(const char *s, const REAL *vec, int no);
```

Based on the MSG() and print_msg() mechanisms, a comma-separated list of the no vector elements is produced.

3.1.3 Example (PRINT_REAL_VEC()).

```
{
   FUNCNAME("test_routine");
   REAL_D point;
   ...

   PRINT_REAL_VEC("point coordinates", point, DIM_OF_WORLD);
}
```

generates for the second unit vector in 3D the output

```
test_routine:      point coordinates = (0.00000, 1.00000, 0.00000)
```

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, ...)                          \
  do {                                                  \
    if (msg_info&&(MIN(msg_info,(info))>=(noinfo))) {   \
      print_funcname(funcName); print_msg(__VA_ARGS__); \
    }                                                   \
  } while (0)

#define PRINT_INFO(info,noinfo, ...)                    \
```

```
  do {                                                    \
    if (msg_info&&(MIN(msg_info,(info))>=(noinfo))) {     \
      print_msg(__VA_ARGS__);                             \
    }                                                     \
  } while (0)
```

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 3.1.5.

3.1.4 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 with return value 1 after using the `ERROR`:

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

Furthermore, two macros for testing boolean values are available:

```
#define TEST(test, ...)                                       \
  do {                                                        \
    if (!(test)) {                                            \
      print_error_funcname(funcName, __FILE__, __LINE__);     \
      print_error_msg(__VA_ARGS__);                           \
    }                                                         \
  } while (0)
#define TEST_EXIT(test, ...)                                  \
  do {                                                        \
    if (!(test)) {                                            \
      print_error_funcname(funcName, __FILE__, __LINE__);     \
      print_error_msg_exit(__VA_ARGS__);                      \
    }                                                         \
  } while (0)
```

If `test` is not true both macros will print the specified 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.

3.1.5 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(const 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 ALBERTA 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 3.1.5).

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

If not disabled by the installer, ALBERTA libraries are also available in versions suited for debugging of code. In the debugging version the macro `ALBERTA_DEBUG` set to `1`. The functionality of some ALBERTA routines and macros is changed in the debugging versions.

Specifically, more safety tests are carried out that are normally unnecessary in optimized production versions of code. We provide the following additional message macros which are only active in the debug versions of ALBERTA:

```
DEBUG_TEST
DEBUG_TEST_EXIT
```

These macros have the same behaviour as the corresponding macros without the DEBUG-prefix if ALBERTA_DEBUG is set, and are ignored otherwise.

3.1.3 Memory allocation and deallocation

ALBERTA keeps track of the amount of memory which is allocated and de–allocated by the routines described below. Information about the currently used amount of allocated memory can be obtained by calling the function

```
void print_mem_use();
```

3.1.3.1 General Allocation

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

```
void *alberta_alloc(size_t, const char *, const char *,int);
void *alberta_realloc(void *, size_t, size_t, const char *, const char *, int);
void *alberta_calloc(size_t, size_t, const char *, const char *,int);
void alberta_free(void *, size_t);
```

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.

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

`alberta_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 `NULL` pointer, the `alberta_realloc()` function allocates a new block of the requested size.

`alberta_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.

`alberta_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 `alberta_alloc()`, `alberta_realloc()`, or `alberta_calloc()`.

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

```
TYPE* MEM_ALLOC(size_t, TYPE);
TYPE* MEM_REALLOC(TYPE *, size_t, size_t, TYPE);
TYPE* MEM_CALLOC(size_t, TYPE);
TYPE* MEM_FREE(TYPE *, size_t, 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().

3.1.6 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.

3.1.3.2 Allocation of matrices

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

```
void **alberta_matrix(size_t, size_t, size_t, const char *, const char *, int);
void free_alberta_matrix(void **, size_t, size_t, size_t);
```

alberta_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_alberta_matrix(ptr, nr, nc, el_size) frees the matrix pointed to by ptr, previously allocated by alberta_matrix().

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

```
TYPE** MAT_ALLOC(size_t, size_t, TYPE);
void   MAT_FREE(TYPE **, size_t, size_t, 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().

3.1.3.3 Allocation and management of workspace

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.

The following functions access and enlarge workspaces:

```
WORKSPACE *get_workspace(size_t, const char *, const char *, int);
WORKSPACE *realloc_workspace(WORKSPACE *,size_t,const char *,const char *,int);
```

Description:

get_workspace(size, name, file, line) return value is a pointer to a WORKSPACE structure holding a vector of length size bytes; 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.

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 NULL 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 macros

```
WORKSPACE* GET_WORKSPACE(size_t)
WORKSPACE* REALLOC_WORKSPACE(WORKSPACE*, size_t)
```

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

GET_WORKSPACE(ws, size) returns a pointer to WORKSPACE structure holding a vector of length size bytes.

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 NULL pointer, a new WORKSPACE structure is allocated.

The functions

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

are used for `WORKSPACE` deallocation. Description:

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

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

For convenience, the corresponding macros are defined as well.

```
void CLEAR_WORKSPACE(WORKSPACE *)
void FREE_WORKSPACE(WORKSPACE *)
```

3.1.3.4 Allocation of "scratch" memory with easy cleanup

Sometimes it is convenient to allocate a lot of objects dynamically; afterwards one always has the dilemma that one has to keep track of each object individually, in order to avoid memory leaks. The following support macros allow the allocation of many small objects of different size from a single pool, with the option to free up the memory for the entire pool at once. Individual object, however, may not be freed individually.

```
typedef struct obstack SCRATCH_MEM[1];
typedef struct obstack *SCRATCH_MEM_PTR; /* A reference to an existing pool */
```

As can be seen, currently these "scratch" memory regions are based on the GNU obstack framework, but an application should not rely on this fact.
Initialization of such a scratch memory area:

```
SCRATCHM_MEM handle;

SCRATCH_MEM_INIT(handle);
```

Allocation from a scratch-memory pool:

```
ptr = SCRATCH_MEM_ALLOC(handle, n_elem, type);
ptr = SCRATCH_MEM_CALLOC(handle, n_elem, type);
```

Cleaning up:

```
SCRATCH_MEM_ZAP(handle);
```

Afterwards, `handle` has to be reinitialized before it can be used again, calling `SCRATCH_MEM_INIT(handle)`.
Copying of scratch-memory handles:

```
SCRATCH_MEM to;
SCRATCH_MEM from;

SCRATCH_MEM_INIT(from);

SCRATCH_MEM_CPY(to, from);
```

Note that this is a shallow copy: only the administrative data structures are copied, not the underlying data. Calling `SCRATCH_MEM_ZAP()` with interchangeably either `to` or `from` as argument will destroy the underlying data.

3.1.4 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.

3.1.4.1 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 must 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 similar to an include statement in a C-program. Using the function `init_parameters()` (see below) for reading the parameter file, the named file is read by a recursive call of the function `init_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.

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.

3.1.4.2 Reading of parameter files

Initializing parameters from such files is done by

```
void init_parameters(int, const char *);
```

Description:

`init_parameters(info, filename)` initializes parameters from a file; `filename` is a string holding the name of the file and if values of the argument `info` and the global variable `msg_info` are not zero, a list of all defined parameters is printed to the message stream; if `init_parameters()` can not open the input file, or `filename` is a pointer to `NULL`, no parameters are 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.

3.1.4.3 Adding of parameters or changing of parameter values

Several calls of `init_parameters()` 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. Examples for giving parameter values from the command line and integrating them into the set of parameters are shown in Sections 2.3.3 and 2.4.2.

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

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

Description:

add_parameter(info, 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 `info` 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 3.1.5).

ADD_PARAMETER(info, key, value) acts like `add_parameter(info, 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.

3.1.4.4 Display and saving of parameters and parameter values

All a list of all parameters together with the actual parameter values can be printed to `stdout` using the function

```
void print_parameters(void);
```

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 *, int);
```

Description:

save_parameters(file, info) writes all successfully 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.

3.1.4.5 Getting parameter values

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

```
int get_parameter(int, const char *, const char *, ...);
int GET_PARAMETER(int, const char *, const char *, ...)
```

Description:

get_parameter(info, 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 `info` 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 a 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.

`%i` matches a decimal integer, whose format is the same as expected for the subject sequence of the `strtol(arg, NULL, 0)` 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.

`%B` matches a boolean value; the corresponding argument should be a pointer to a `bool` variable. The boolean value may be specified as any of the following strings: `true`, `t`, `yes`, `y`, `1`, `false`, `f`, `no`, `n`, `0`, `nil`, with the obvious meaning concerning the translation into the value for the `bool` data type of `C`.

`%*` 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 `info` 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()` is a parameter `parameter information` specified in a parameter file, see Section 3.1.5.

`GET_PARAMETER(info, key, format, ...)` is a macro and acts in the same way as the function `get_parameter(info, 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 beforehand! By the return value the number of converted arguments can be checked.

3.1.7 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`.

3.1.5 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 3.1.2).

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

- **`0`** no message at all is produced, although the value `info` may be non zero;
- **`1`** gives only messages if the value of `info` is non zero;
- **`2`** all error messages are printed, although the value of `info` may be zero;
- **`4`** all messages are printed, although the value of `info` 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 3.1.2).

3.1.6 Generating filenames for meshes and finite element data

During simulation of time-dependent problems one often wants to store meshes and finite element data for the sequence of time steps. A routine is provided to automatically generate file names composed from a given data path, a base-name for the file and a number which is iteration counter of the actual time step in time-dependent problems. Such a function simplifies the handling of a sequence of data for reading and writing. It also ensures that files are listed alphabetically in the given path (up to 1 million files with the same base-name).

```
const char *generate_filename(const char *, const char *, int);
```

Description:

generate_filename(path, file, index) composes a filename from the given `path`, the base-name `file` of the file and the (iteration counter) `index`. When no path is given, the current directory `"./"` is used, if the first character of `path` is `'~'`, `path` is assumed to be located in the home directory and the name of the path is expanded accordingly, using the environment variable `HOME`. A pointer to a string containing the full filename is the return value; this string is overwritten on the next call to `generate_filename()`.

`generate_filename("./output", "mesh",1)` returns `"./output/mesh000001"`, for instance. An example how to use `generate_filename()` in a time dependent problem is given in Section 2.4.10.

3.2 Data structures for the hierarchical mesh

3.2.1 Dimension of the mesh

The current version of ALBERTA supports meshes triangulated using d-dimensional simplices where $d \in \{1, 2, 3\}$. These are embedded in $\mathbb{R}^n$, with $n \geq d$. For most applications we have $d = n$. However, for finite element methods on curves ($d = 1$) or surfaces ($d = 2$) embedded in $\mathbb{R}^n$ (like mean curvature flow [9]), the vertex coordinates of the simplices have $n > d$ components. There are three principal constants which affect the storage layout of various data-types, from `alberta.h`:

```
/* DIM_OF_WORLD is a compile time constant, not defined in alberta.h */
#define DIM_LIMIT     3 /* limiting mesh-dimension */
#define DIM_MAX       MIN(DIM_OF_WORLD, DIM_LIMIT)
```

DIM_LIMIT Defined to the limit for the mesh-dimension. More than tetrahedral meshes are not supported, so this is defined to 3.

DIM_OF_WORLD Defined to the dimension of the ambient space, i.e. n in the notation used above.

DIM_MAX Defined to the maximum value of the mesh-dimension, given the current value of `DIM_OF_WORLD`.

Derived dimension dependent constants ALBERTA provides some expressions for the number of face-simplices of each possible co-dimension. In ALBERTA, the name "face" is reserved for the faces of tetrahedra; to denote the co-dimension 1 face-simplex for simplices of arbitrary dimensions the name "wall" is used. Besides that, there are expressions for the possible number of neighbours, the faculty of the mesh-dimension and the number of barycentric co-ordinates of given dimension. `alberta.h` defines the following generic macros:

```
#define N_VERTICES(DIM)  ((DIM)+1)
#define N_EDGES(DIM)     ((DIM)*((DIM)+1)/2)
#define N_WALLS(DIM)     ((DIM)+1)
#define N_FACES(DIM)     (((DIM) == 3) * N_WALLS(DIM))
#define N_NEIGH(DIM)     (((DIM) != 0) * N_WALLS(DIM))
```

```
#define N_LAMBDA(DIM)     N_VERTICES(DIM)
#define DIM_FAC(DIM)      ((DIM) < 2 ? 1 : (DIM) == 2 ? 2 : 6)
```

N_VERTICES() number of vertices of a simplex

N_EDGES() number of edges of a simplex

N_WALLS() number of co-dimension 1 face-simplices of a simplex

N_FACES() number of co-dimension 1 face-simplices of a simplex of dimension 3

N_NEIGH() possible number of neighbour elements across walls

N_LAMBDA() number barycentric co-ordinates

DIM_FAC() faculty of the mesh-dimension

From these generic macros `alberta.h` specializes variants with the suffixes:

_MAX maximum value given `DIM_OF_WORLD`

_LIMIT limiting value ever supported

_0D, _1D, _2D, _3D special value given the mesh-dimension

For example, the `N_VERTICES` macro exists with the following variants:

```
#define N_VERTICES_0D          1
#define N_VERTICES_1D          2
#define N_VERTICES_2D          3
#define N_VERTICES_3D          4
#define N_VERTICES_MAX         N_VERTICES(DIM_MAX)
#define N_VERTICES_LIMIT       N_VERTICES(DIM_LIMIT)
```

Finally we use the following definitions describing possible positions of degrees of freedom on an element:

```
typedef enum node_types {
  VERTEX = 0,
  CENTER,
  EDGE,
  FACE,
  N_NODE_TYPES
} NODE_TYPES;
```

The symbols refer to `DOFs` located at the face-simplices with the following meanings:

VERTEX The vertex of a simplex. In 1d the vertices are treated as the "walls" of an element.

CENTER The interior of an element. The `DOFs` of discontinuous basis-functions, e.g., are always treated as `CENTER-DOFs`.

EDGE The edges of an element. Note that 1d simplices do not have edges in ALBERTA as long as it concerns the location of `DOFs`. So 1d-meshes have only `VERTEX` and `CENTER` `DOFs`.

FACE The faces of an element. This is reserved for 3d only. Note that the co-dimension 1 face-simplex is denoted as "wall-simplex" within ALBERTA.

3.2.2 The local indexing on elements

For the handling of higher order discretizations where besides vertices DOFs can be located at edges (in 2d and 3d), faces (in 3d), or center, we also need a local numbering for edges, and faces. Finally, a local numbering of neighbours for handling neighbour information is needed, used for instance in the refinement algorithm itself and for error estimator calculation.

The `i`-th neighbour is always the element opposite the `i`-th vertex. The `i`-th edge/face is the edge/face opposite the `i`-th vertex in 2d respectively 3d; edges in 3d are numbered in the following way (compare Figure 3.1):

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

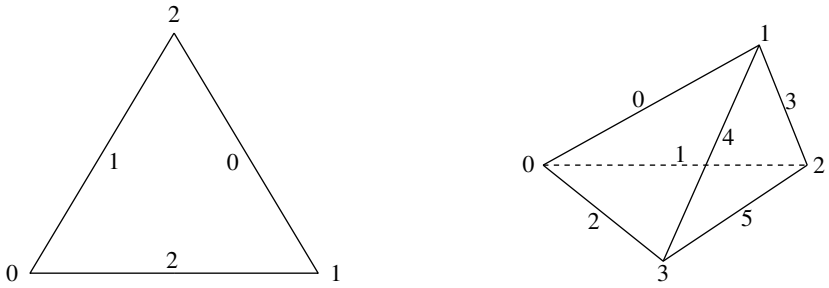


Figure 3.1: Local indices of edges/neighbours in 2d and local indices of edges in 3d.

The data structures described in the subsequent sections are based on this local numbering of vertices, edges, faces, and neighbours.

3.2.3 BLAS-like routines for `DIM_OF_WORLD`- and `N_LAMBDA_MAX`-arrays

The term "BLAS" stands for "Basic Linear Algebra Subroutines", see [14, 6]. There are several vector and array data-types associated with `DIM_OF_WORLD` and `N_LAMBDA_MAX`. The basic array types are

```
typedef REAL        REAL_D[DIM_OF_WORLD];
typedef REAL        REAL_B[N_LAMBDA_MAX];
```

`REAL_D` An array of the dimension of the ambient space.

`REAL_B` An array of the size of the maximum mesh-dimension at given `DIM_OF_WORLD`. Note that for a given mesh only the first `N_LAMBDA(mesh->dim)` components of a `REAL_B`-vector are actually used. Excess elements should be cleared to 0.

To support the static initialization of `REAL_B`-arrays there are macros `INIT_BARY_?D()`. The definitions of these macros depend on the values of `DIM_MAX`, we have the following defines in `alberta.h`:

```
#if DIM_MAX == 0
# define INIT_BARY_0D(a)          { 1.0 }
# define INIT_BARY_1D(a, b)       { 1.0 }
```

```
# define INIT_BARY_2D(a, b, c)      { 1.0 }
# define INIT_BARY_3D(a, b, c, d)   { 1.0 }
# define INIT_BARY_MAX(a, b, c, d) INIT_BARY_0D(a)
#elif DIM_MAX == 1
# define INIT_BARY_0D(a)            { (a), 0.0 }
# define INIT_BARY_1D(a, b)         { (a), (b) }
# define INIT_BARY_2D(a, b, c)      { (a), (b) }
# define INIT_BARY_3D(a, b, c, d)   { (a), (b) }
# define INIT_BARY_MAX(a, b, c, d) INIT_BARY_1D(a, b)
#elif DIM_MAX == 2
# define INIT_BARY_0D(a)            { (a), 0.0, 0.0 }
# define INIT_BARY_1D(a, b)         { (a), (b), 0.0 }
# define INIT_BARY_2D(a, b, c)      { (a), (b), (c) }
# define INIT_BARY_3D(a, b, c, d)   { (a), (b), (c) }
# define INIT_BARY_MAX(a, b, c, d) INIT_BARY_2D(a, b, c)
#elif DIM_MAX == 3
# define INIT_BARY_0D(a)            { (a), 0.0, 0.0, 0.0 }
# define INIT_BARY_1D(a, b)         { (a), (b), 0.0, 0.0 }
# define INIT_BARY_2D(a, b, c)      { (a), (b), (c), 0.0 }
# define INIT_BARY_3D(a, b, c, d)   { (a), (b), (c), (d) }
# define INIT_BARY_MAX(a, b, c, d) INIT_BARY_3D(a, b, c, d)
#else
# error Unsupported DIM_MAX
#endif
```

To have array-types for matrices like Jacobians and Hessians there is bunch of data-types in `alberta.h`. The suffixes which are composed from the two letters `D` and `B` code the ordering of the array dimensions, e.g. a `REAL_BD` is an array which's first index ranges from 0 to (`N_LAMBDA_MAX`-1) and which's second index ranges from 0 to (`DIM_OF_WORLD`-1). Currently, the following types are defined:

```
typedef REAL          REAL_B[N_LAMBDA_MAX];
typedef REAL_B        REAL_BB[N_LAMBDA_MAX];
typedef REAL          REAL_D[DIM_OF_WORLD];
typedef REAL_D        REAL_DD[DIM_OF_WORLD];
typedef REAL_D        REAL_BD[N_LAMBDA_MAX];
typedef REAL_BD       REAL_BBD[N_LAMBDA_MAX];
typedef REAL_DD       REAL_DDD[DIM_OF_WORLD];
typedef REAL_DD       REAL_BDD[N_LAMBDA_MAX];
typedef REAL_BDD      REAL_BBDD[N_LAMBDA_MAX];
typedef REAL_B        REAL_DB[DIM_OF_WORLD];
typedef REAL_BB       REAL_DBB[DIM_OF_WORLD];
typedef REAL_BB       REAL_BBB[N_LAMBDA_MAX];
typedef REAL_BBB      REAL_BBBB[N_LAMBDA_MAX];
typedef REAL_BBB      REAL_DBBB[DIM_OF_WORLD];
typedef REAL_BBBB     REAL_DBBBB[DIM_OF_WORLD];
typedef REAL_DB       REAL_BDB[N_LAMBDA_MAX];
typedef REAL_DBB      REAL_BDBB[N_LAMBDA_MAX];
```

To ease arithmetic with such vector- and matrix-types there is a variety of inline-functions defined in `alberta_inlines.h` (`alberta_inlines.h` is automatically included by `alberta.h`). We describe only a selection, for the full list we refer the reader to the header `alberta_inlines.h`. Some of the following functions are also available as matrix versions (e.g. `MAXEY_DOW(a,x,y)`, `MSCP_DOW(x,y)`, ...), but they aren't descripted seperately. The prefix `M` means that they expect `REAL_DD` matrices instead of `REAL_D` vectors. A tabular overview can

be found in Table 3.1 and Table 3.2.

Prototypes

```
REAL SCP_DOW(const REAL_D x, const REAL_D y)
REAL GRAMSCP_DOW(const REAL_DD A, const REAL_D x, const REAL_D y)
REAL NORM_DOW(const REAL_D x)
REAL NRM2_DOW(const REAL_D x)
REAL NORM1_DOW(const REAL_D x)
REAL NORM8_DOW(const REAL_D x)
REAL NRMP_DOW(const REAL_D x, REAL p)
REAL PNRMP_DOW(const REAL_D x, REAL p)
REAL DIST_DOW(const REAL_D x, const REAL_D y)
REAL DST2_DOW(const REAL_D x, const REAL_D y)
REAL DIST1_DOW(const REAL_D x, const REAL_D y)
REAL DIST8_DOW(const REAL_D x, const REAL_D y)
REAL SUM_DOW(const REAL_D x)
REAL POW_DOW(REAL a)

REAL *SET_DOW(REAL a, REAL_D x)
REAL *COPY_DOW(const REAL_D x, REAL_D y)
REAL *SCAL_DOW(REAL a, REAL_D x)
REAL *AX_DOW(REAL a, REAL_D x)
bool CMP_DOW(REAL val, const REAL_D a)

REAL *AXEY_DOW(REAL a, const REAL_D x, REAL_D y)
REAL *AXPY_DOW(REAL a, const REAL_D x, REAL_D y)
REAL *AXPBY_DOW(REAL a, const REAL_D x,
                REAL b, const REAL_D y, REAL_D z)
REAL *AXPBYP_DOW(REAL a, const REAL_D x,
                 REAL b, const REAL_D y, REAL_D z)
REAL *AXPBYPCZ_DOW(REAL a, const REAL_D x, REAL b, const REAL_D y,
                   REAL c, const REAL_D z, REAL_D w)
REAL *AXPBYPCZP_DOW(REAL a, const REAL_D x, REAL b, const REAL_D y,
                    REAL c, const REAL_D z, REAL_D w)
REAL WEDGE_DOW(const REAL_D x, const REAL_D y)
REAL *WEDGE_DOW(const REAL_D x, const REAL_D y, REAL_D z)

EXPAND_DOW(x)
FORMAT_DOW
SCAN_EXPAND_DOW(v)
SCAN_FORMAT_DOW

REAL *GRAD_DOW(int dim, const REAL_BD Lambda, const REAL_B b_grd, REAL_D
    x_grd)
REAL *GRAD_P_DOW(int dim, const REAL_BD Lambda,
                 const REAL_B b_grd, REAL_D x_grd)
REAL_D *D2_DOW(int dim, const REAL_BD Lambda,
               const REAL_BB b_hesse, REAL_DD x_hesse)
REAL_D *D2_P_DOW(int dim, const REAL_BD Lambda,
                 const REAL_BB b_hesse, REAL_DD x_hesse)

REAL *MV_DOW(const REAL_DD m, const REAL_D v, REAL_D b)
REAL *MTV_DOW(const REAL_DD m, const REAL_D v, REAL_D b)
REAL *GEMV_DOW(REAL a, const REAL_DD m, const REAL_D v, REAL beta, REAL_D b)
REAL *GEMTV_DOW(REAL a, const REAL_DD m, const REAL_D v, REAL beta, REAL_D b)
```

```
REAL *AFFINE_DOW(const AFF_TRAFO *trafo, const REAL_D x, REAL_D y)
REAL *AFFINV_DOW(const AFF_TRAFO *trafo, const REAL_D x, REAL_D y)
AFF_TRAFO *AFFAFF_DOW(const AFF_TRAFO *A, const AFF_TRAFO *B, AFF_TRAFO *C)
AFF_TRAFO *INVAFF_DOW(const AFF_TRAFO *A, AFF_TRAFO *B)

REAL MSCP_DOW(const REAL_DD x, const REAL_DD y)
REAL MNORM_DOW(const REAL_DD m)
REAL MNRM2_DOW(const REAL_DD m)
REAL MDIST_DOW(const REAL_DD a, const REA_DD b)
REAL MDST2_DOW(const REAL_DD a, const REAL_DD b)

REAL_D *MSET_DOW(REAL val, REAL_DD m)
REAL_D *MCOPY_DOW(const REAL_DD x, REAL_DD y)

REAL_D *MSCAL_DOW(REAL a, REAL_DD m)
REAL_D *MAX_DOW(REAL a, REAL_DD m)
bool MCMP_DOW(REAL val, const REAL_DD a)

REAL_D *MAXEY_DOW(REAL a, const REAL_DD x, REAL_DD y)
REAL_D *MAXPY_DOW(REAL a, const REAL_DD x, REAL_DD y)
REAL_D *MAXTPY_DOW(REAL a, const REAL_DD x, REAL_DD y)
REAL_D *MAXPBY_DOW(REAL a, const REAL_DD x,
                   REAL b, const REAL_DD y, REAL_DD z)
REAL_D *MAXPBYP_DOW(REAL a, const REAL_DD x,
                    REAL b, const REAL_DD y, REAL_DD z)
REAL_D *MAXPBYPCZ_DOW(REAL a, const REAL_DD x, REAL b, const REAL_DD y,
                      REAL c, const REAL_DD z, REAL_DD w)
REAL_D *MAXPBYPCZP_DOW(REAL a, const REAL_DD x, REAL b, const REAL_DD y,
                       REAL c, const REAL_DD z, REAL_DD w)

MEXPAND_DOW(m)
MFORMAT_DOW

REAL_D *MGRAD_DOW(int dim, const REAL_BD Lambda, const REAL_DB b_grd,
                  REAL_DD x_grd)
REAL_D *MGRAD_P_DOW(int dim, const REAL_BD Lambda, const REAL_DB b_grd,
                    REAL_DD x_grd)
REAL_DD *MD2_DOW(int dim, const REAL_BD Lambda, const REAL_BB *b_hesse,
                 REAL_DDD x_hesse)
REAL_DD *MD2_P_DOW(int dim, const REAL_BD Lambda, const REAL_BB *b_hesse,
                   REAL_DDD x_hesse)

REAL_D *MINVERT_DOW(const REAL_DD m, REAL_DD mi)
REAL_D *MM_DOW(const REAL_DD a, const REAL_DD b, REAL_DD c)
REAL_D *MTM_DOW(const REAL_DD a, const REAL_DD b, REAL_DD c)
REAL_D *MMT_DOW(const REAL_DD a, const REAL_DD b, REAL_DD c)
REAL MDET_DOW(const REAL_DD m)
```

Descriptions

For a more compact presentation, refer to Table 3.1 and 3.2.

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

`GRAMSCP_DOW(A, x, y)` in case `A` is a spd-matrix it returns the scalar product of the two vectors `x`, `y`, defined by `A`: $(x, y)_A := < x, Ay >$

`NORM_DOW(x)` returns the Euclidean norm of the vector `x`.

NRM2_DOW(x) returns the Euclidean scalar product of the vector x with itself. This means it returns the square of the Euclidean norm of the vector x.

NORM1_DOW(x) returns the 1-norm of the vector x. This means it returns the sum of the absolut values of the vector entries.

NORM8_DOW(x) returns the infinity norm or maximum norm of the vector x.

NRMP_DOW(x, p) returns the p-norm of the vector x.

PNRMP_DOW(x, p) returns the p-norm to the power of p of the vector x.

DIST_DOW(x, y) returns the Euclidean distance between the two vectors x, y.

DST2_DOW(x, y) returns the square of the Euclidean distance between two vectors x, y.

DIST1_DOW(x, y) returns the 1-norm of the vector (x-y).

DIST8_DOW(x, y) returns the infinity norm of the vector (x-y).

SUM_DOW(x) returns the sum of the vector entries of the vector x.

POW_DOW(a) returns a to the power of DIM_OF_WORLD.

SET_DOW(a, x) set all elements of vector x to a. Returns x

COPY_DOW(x, y) copies all elements of vector x to y. Returns y.

SCAL_DOW(a, x) scales all elements of the vector x with a. Returns x.

AX_DOW(a, x) scales all elements of vector x with a. Returns x.

CMP_DOW(val, a) returns true if all elements of the vector a have the same value val, and it returns false if there is any element of the vector a with value !=val.

AXEY_DOW(a, x, y) scales all elements of vector x with a and stores the resulting vector in y. Returns y.

AXPY_DOW(a, x, y) scales all elements of vector x with a and add it up to the vector y. Returns y.

AXPBY_DOW(a, x, b, y, z) scales all elements of vector x with a, scales all elements of the vector y with b and add these two vectors. The result is stored in the vector z. Returns z.

AXPBYP_DOW(a, x, b, y, z) scales all elements of vector x with a, scales all elements of vector y with b and add these two vectors up to the vector z. Returns z.

AXPBYPCZ_DOW(a, x, b, y, c, z, w) scales all elements of vector x with a, scales all elements of vector y with b, scales all elements of vector z with c and add these three vectors. The result is stored in the vector w. Returns w.

AXPBYPCZP_DOW(a, x, b, y, c, z, w) scales all elements of vector x with a, scales all elements of vector y with b, scales all elements of vector z with c and add these three vectors up to the vector w. Returns w.

WEDGE_DOW(a, b) for DIM_OF_WORLD==2 returns the product a[0]*b[1]-a[1]*b[0].

WEDGE_DOW(a, b, r) for DIM_OF_WORLD==3 fills r with the vector product $a \times b \in \mathbb{R}^3$. Returns r.

EXPAND_DOW(x) returns every entry of the vector x seperated with a comma. It is used for easier print-out of REAL_D. An example is stated below.

FORMAT_DOW Example for DIM_OF_WORLD == 2:

```
printf("text" FORMAT_DOW "more text\n", EXPAND_DOW(x));
```

equivalent to:

```
printf("text" "[%10.5le, %10.5le]" "more text\n", x[0], x[1]);
```

SCAN_EXPAND_DOW(v)

SCAN_FORMAT_DOW an example will explain both (DIM_OF_WORLD == 2):

```
printf{"text" SCAN_FORMAT_DOW "more text\n", SCAN_EXPAND_DOW(v));
```

equivalent to:

```
printf("text" "%f %f" "more text\n", &v[0], &v[1]);
```

GRAD_DOW(dim, Lambda, b_grd, x_grd) convert a barycentric gradient b_grd to a world gradient and stores the resulting vector in x_grd, given the gradient of the transformation to the reference element Lambda. Whereas dim is the dimension of the mesh. Returns x_grd.

GRAD_P_DOW(dim, Lambda, b_grd, x_grd) convert a barycentric gradient b_grd to a world gradient and add it up to the vector x_grd, given the gradient of the transformation to the reference element Lambda. Whereas dim is the dimension of the mesh. Returns x_grd.

D2_DOW(dim, Lambda, b_hesse, x_hesse) convert a barycentric Hesse matrix b_hesse to a world Hesse matrix and stores the resulting matrix in x_hesse, given the gradient of the transformation to the reference element Lambda. Whereas dim is the dimension of the mesh. Returns x_hesse.

D2_P_DOW(dim, Lambda, b_hesse, x_hesse) convert a barycentric Hesse matrix b_hesse to a world Hesse matrix and add it up to the matrix x_hesse, given the gradient of the transformation to the reference element Lambda. Whereas dim is the dimension of the mesh. Returns x_hesse.

MV_DOW(m, v, b) calculates the matrix-vector multiplication of the matrix m and the vector v: b += m*v. Returns b.

MTV_DOW(m, v, b) calculates the matrix-vector multiplication of the transpose of matrix m and the vector v: b += m^t*v. Returns b.

GEMV_DOW(a, m, v, beta, b) returns b = beta*b + a*(m*v). Where a and beta are scalar (type REAL), m a matrix (type REAL_DD) and v and b are vectors (type REAL_D).

GEMTV_DOW(a, m, v, beta, b) returns b = beta*b + a*(m^t*v). Where a and beta are scalar (type REAL), m a matrix (type REAL_DD) and v and b are vectors (type REAL_D).

AFFINE_DOW(trafo, x, y) calculates the affine transformation between the two vectors x and y and returns the vector y. It consists of a linear transformation (matrix-vector multiplication with the matrix trafo->M) followed by a translation (with the translation vector trafo->t). Adequate formular: y = trafo->M * x + trafo->t.

AFFINV_DOW(trafo, x, y) applies the inverse of the affine transformation between x and y. Returns y.

AFFAFF_DOW(A, B, C) returns ... A, B, C.

INVAFF_DOW(A, B) returns ... A, B.

MINVERT_DOW(m, mi) returns the inverted matrix mi of the matrix m.

MM_DOW(a, b, c) returns the matrix matrix multiplication of a and b and stores the resulting matrix in c. Returns c.

MTM_DOW(a, b, c) returns the matrix matrix multiplication of the transposed matrix of a and b and stores the resulting matrix in c. Returns c.

MMT_DOW(a, b, c) returns the matrix matrix multiplication of a and the transposed matrix of b and stores the resulting matrix in c. Returns c.

MDET_DOW(m) returns the determinant of matrix m.

3.2.4 Boundary types

3.2.1 Compatibility Note. *Previous versions of ALBERTA were fixing the boundary conditions – Dirichlet, Neumann, others – in the macro-data file and data-structures. This has changed: the new scheme is to assign only "street-numbers" to boundary segments in the macro-triangulation and leave the interpretation to the application program. This section describes some of the implications of this change. Compare also Compatibility Note 3.2.8 and Compatibility Note 3.2.2. The reader is also referred to the documentation for* `dirichlet_bound...()` *Section 4.7.7.1, especially to Example 4.7.6.*

In ALBERTA boundary conditions are first of all attached to boundary segments – and thus to the boundary walls of an element. Boundary segments carry "street-numbers" which are defined by the macro-triangulation. At the moment 255 different "boundary types" are possible, where type 0 is reserved for interior walls.

```
typedef U_CHAR BNDRY_TYPE;
```

Note that this is an *unsigned* value, and does not carry any information about the nature of a boundary condition (e.g. Dirichlet versus natural versus ...) imposed on a specific boundary segment to "close" a specific differential equation or system of equations.

Of course, for doing the linear algebra implied by the need to solve a discretized PDE it is often handy to assign boundary conditions to degrees of freedom (DOFs) of the finite element spaces. For doing so ALBERTA uses signed characters – S_CHAR – with the convention that positive numbers flag Dirichlet boundary conditions, negative numbers flag natural boundary conditions and 0 indicates an interior node. Specifically, there are three pre-defined constants

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

and some macro which may help the to make code more readable, namely

```
#define IS_NEUMANN(bound)   ((bound) <= NEUMANN)
#define IS_DIRICHLET(bound) ((bound) >= DIRICHLET)
#define IS_INTERIOR(bound)  ((bound) == 0)
```

There are some issues for assigning boundary conditions to DOFs. One point is that a DOF may belong to boundary segments with differing boundary classification, e.g. vertex DOFs in 2d and vertex and edge DOFs in 3d. To handle this point ALBERTA provides a boundary bit-mask data type for such DOFs, together with some support macros:

`REAL SCP_DOW(const REAL_D x, const REAL_D y)`	$(X,Y) = \sum_{i=0}^{d-1} X_i Y_i$
`REAL GRAMSCP_DOW(const REAL_DD A, const REAL_D x, const REAL_D y)`	$(X,Y)_A = \sum_{i,j=0}^{d-1} X_i A_{ij} Y_j$
`REAL NORM_DOW(const REAL_D x)`	$\|X\|_2 = \left(\sum_{i=0}^{d} \|X_i\|^2\right)^{\frac{1}{2}}$
`REAL NRM2_DOW(const REAL_D x)`	$\|X\|_2^2 = \sum_{i=0}^{d-1} \|X_i\|^2$
`REAL NORM1_DOW(const REAL_D x)`	$\|X\|_1 = \sum_{i=0}^{d-1} \|X_i\|$
`REAL NORM8_DOW(const REAL_D x)`	$\|X\|_\infty = \max_{i=0}^{d-1} \|X_i\|$
`REAL NRMP_DOW(const REAL_D x, REAL p)`	$\|X\|_p = \left(\sum_{i=0}^{d-1} \|X_i\|^p\right)^{\frac{1}{p}}$
`REAL PNRMP_DOW(const REAL_D x, REAL p)`	$\|X\|_p^p = \sum_{i=0}^{d-1} \|X_i\|^p$
`REAL DIST_DOW(const REAL_D x, const REAL_D y)`	$dist = \left(\sum_{i=0}^{d-1} \|X_i - Y_i\|^2\right)^{\frac{1}{2}}$
`REAL DST2_DOW(const REAL_D x, const REAL_D y)`	$dst2 = \sum_{i=0}^{d-1} \|X_i - Y_i\|^2$
`REAL DIST1_DOW(const REAL_D x, const REAL_D y)`	$dist1 = \sum_{i=0}^{d-1} \|X_i - Y_i\|$
`REAL DIST8_DOW(const REAL_D x, const REAL_D y)`	$dist8 = \max_{i=0}^{d-1} \|X_i - Y_i\|$
`REAL SUM_DOW(const REAL_D x)`	$sum = \sum_{i=0}^{d-1} X_i$
`REAL POW_DOW(REAL a)`	$pow = a^d$
`REAL *SCAL_DOW(REAL a, REAL_D x)` `REAL *AX_DOW(REAL a, REAL_D x)`	$X* = a$
`REAL *AXEY_DOW(REAL a, const REAL_D x, REAL_D y)`	$Y = aX$
`REAL *AXPY_DOW(REAL a, const REAL_D x, REAL_D y)`	$Y += aX$
`REAL *AXPBY_DOW(REAL a, const REAL_D x, REAL b, const REAL_D y, REAL_D z)`	$Z = aX + bY$
`REAL *AXPBYP_DOW(REAL a, const REAL_D x, REAL b, const REAL_D y, REAL_D z)`	$Z += aX + bY$
`REAL *AXPBYPCZ_DOW(REAL a, const REAL_D x, REAL b, const REAL_D y, REAL c, const REAL_D z, REAL_D w)`	$W = aX + bY + cZ$
`REAL *AXPBYPCZP_DOW(REAL a, const REAL_D x, REAL b, const REAL_D y, REAL c, const REAL_D z, REAL_D w)`	$W += aX + bY + cZ$
`REAL WEDGE_DOW(const REAL_D x, const REAL_D y)` `(for DIM_OF_WORLD == 2)`	$X[0] * Y[1] - X[1] * Y[0]$
`REAL *WEDGE_DOW(const REAL_D x, const REAL_D y, REAL_D z)` `(for DIM_OF_WORLD == 3)`	$Z = X \times Y$

Table 3.1: Implemented BLAS routines for `REAL_D` vectors (d = `DIM_OF_WORLD`, with the prefix `M` for `REAL_DD` matrices)

```
#define N_BNDRY_TYPES 256
typedef BITS_256              BNDRY_FLAGS;

/* Some "standard" bit-field operations, meant to hide the
 * N_BNDRY_TYPES argument.
 */
```

`REAL *MV_DOW(const REAL_DD m, const REAL_D v, REAL_D b)`	$b \mathrel{+}= M * v$
`REAL *MTV_DOW(const REAL_DD m, const REAL_D v, REAL_D b)`	$b \mathrel{+}= M^t * v$
`REAL *GEMV_DOW(REAL a, const REAL_DD m, const REAL_D v, REAL beta, REAL_D b)`	$b = beta * b + a * (M * v)$
`REAL *GEMTV_DOW(REAL a, const REAL_DD m, const REAL_D v, REAL beta, REAL_D b)`	$b = beta * b + a * (M^t * v)$

Table 3.2: Implemented BLAS routines for matrix-vectors multiplication.

```
#define BNDRY_FLAGS_INIT(flags)      bitfield_zap((flags), N_BNDRY_TYPES)
#define BNDRY_FLAGS_ALL(flags)       bitfield_fill((flags), N_BNDRY_TYPES)
#define BNDRY_FLAGS_CPY(to, from)    bitfield_cpy((to), (from), N_BNDRY_TYPES)
#define BNDRY_FLAGS_AND(to, from)    bitfield_and((to), (from), N_BNDRY_TYPES)
#define BNDRY_FLAGS_OR(to, from)     bitfield_or((to), (from), N_BNDRY_TYPES)
#define BNDRY_FLAGS_XOR(to, from)    bitfield_xor((to), (from), N_BNDRY_TYPES)
#define BNDRY_FLAGS_CMP(a, b)        bitfield_cmp((a), (b), N_BNDRY_TYPES)

/* bit 0 flags boundary segments, if not set we are in the interior */
#define BNDRY_FLAGS_IS_INTERIOR(mask) (!bitfield_tst((mask), 0))

/* Set bit 0 to mark this as a boundary bit-mask. */
#define BNDRY_FLAGS_MARK_BNDRY(flags) bitfield_set((flags), INTERIOR)

/* Return TRUE if SEGMENT has BIT set _and_ BIT != 0. */
#define BNDRY_FLAGS_IS_AT_BNDRY(segment, bit)   \
  ((bit) && bitfield_tst((segment), (bit)))

/* Set a bit in the boundary-type mask. The precise meaning of BIT:
 *
 * BIT == 0: clear the boundary mask (meaning: interior node)
 * BIT >  0: set bit BIT and also bit 0 (meaning: boundary node)
 */
#define BNDRY_FLAGS_SET(flags, bit)             \
  if ((bit) != INTERIOR) {                      \
    bitfield_set((flags), INTERIOR);            \
    bitfield_set((flags), (bit));               \
  } else {                                      \
    BNDRY_FLAGS_INIT(flags);                    \
  }

/* return TRUE if SEGMENT and MASK have non-zero overlap */
#define BNDRY_FLAGS_IS_PARTOF(segment, mask)    \
  bitfield_andp((segment), (mask), 1 /* offset */, N_BNDRY_TYPES)

/* FindFirstBoundaryBit, return INTERIOR for interior nodes, otherwise the
 * number of the first bit set in MASK.
 */
#define BNDRY_FLAGS_FFBB(mask) bitfield_ffs(mask, 1 /* offset */,
    N_BNDRY_TYPES)
```

There is also a support function which returns for a given finite element space on a given element the boundary classification of all local `DOF`s in terms of such bit-masks, namely `get_bound()`, see Section 4.7.1.3. To collect boundary information and interprete the informa-

tion returned by `get_bound()` the function `dirichlet_map()` can be used. Omitting details like the handling of direct sums of finite element spaces its implementation looks like follows. The effect is that `bound[loc_dof]` is set either to `DIRICHLET` or `INTERIOR`, depending on whether the input bit-mask `mask` and the boundary bit-masks of the local `DOF`s overlap.

```
void dirichlet_map(EL_SCHAR_VEC *bound,
                   const EL_BNDRY_VEC *bndry_bits,
                   const BNDRY_FLAGS mask)
{
  int loc_dof;

  for (loc_dof = 0; loc_dof < bound->n_components; loc_dof++) {
    if (BNDRY_FLAGS_IS_INTERIOR(bndry_bits->vec[loc_dof])) {
      bound->vec[loc_dof] = INTERIOR;
      continue;
    }
    if (BNDRY_FLAGS_IS_PARTOF(bndry_bits->vec[loc_dof], mask)) {
      bound->vec[loc_dof] = DIRICHLET;
    } else {
      bound->vec[loc_dof] = INTERIOR;
    }
  }
}
```

The use of the `dirichlet_map()` function is also demonstrated in the `assemble()`-function in the demo-program `heat.c`, see Section 2.4.8.

Besides the single-element mapper `dirichlet_map()` there is also support for filling an entire `DOF_SCHAR_VEC` at once with the boundary-type interpretation for a given finite element space. This task can be performed by the function `dirichlet_bound()` (and its variants), see Section 4.7.7.1.

Generally, many function and structures accepts an argument (or contain a component) of type `BNDRY_FLAGS` which determines on which part of the boundary they are acting. This concerns variants of `dirichlet_bound()` (Section 4.7.7.1), the variants of the support functions for Neumann or Robin boundary conditions (Section 4.7.7.2, Section 4.7.7.3), and the residual error-estimator support functions (Section 4.9). Data-structures affected are `DOF_MATRIX` (Section 3.3.4), `EL_MATRIX_INFO` (4.48), `EL_VEC_INFO` (4.70), `OPERATOR_INFO` (4.50), `BNDRY_OPERATOR_INFO` (4.51).

3.2.5 The `MACRO_EL` data structure

We now describe the macro triangulation and data type for an element of the macro triangulation. The macro triangulation is stored in an array of macro elements:

```
#define N_BNDRY_TYPES 256
typedef U_CHAR              BNDRY_TYPE;
typedef BITS_256            BNDRY_FLAGS;
typedef struct macro_el     MACRO_EL;

struct macro_el
{
  EL            *el;
  REAL_D        *coord[N_VERTICES_MAX];
```

```
  BNDRY_TYPE   wall_bound[N_WALLS_MAX];
#if DIM_MAX > 1
  BNDRY_FLAGS  vertex_bound[N_VERTICES_MAX];
#endif
#if DIM_MAX > 2
  BNDRY_FLAGS  edge_bound[N_EDGES_MAX];
#endif

  NODE_PROJ    *projection[N_NEIGH_MAX + 1];

  int          index;

  MACRO_EL     *neigh[N_NEIGH_MAX];
  S_CHAR       opp_vertex[N_NEIGH_MAX];
  S_CHAR       neigh_vertices[N_NEIGH_MAX][N_VERTICES(DIM_MAX-1)];
  AFF_TRAFO    *wall_trafo[N_NEIGH_MAX];
#if DIM_MAX > 1
  BNDRY_FLAGS  np_vertex_bound[N_VERTICES_MAX];
#endif
#if DIM_MAX > 2
  BNDRY_FLAGS  np_edge_bound[N_EDGES_MAX];
#endif

  S_CHAR       orientation;

  U_CHAR       el_type;

  struct {
    MACRO_EL     *macro_el;
    S_CHAR       opp_vertex;
  } master;
};
```

Description of the individual structure components:

el The root of the binary tree located at this macro element.

coord The pointer to the world coordinates of the element's vertices.

wall_bound The boundary classification of the respective wall. 0 means this is an interior wall, any other number between 1 and 255 is a "street number", the boundary classification as read from the macro triangulation. See also Compatibility Note 3.2.8. See also Section 3.2.4.

vertex_bound Only present for DIM_MAX > 1. The boundary classification of the given vertex.

3.2.2 Compatibility Note. *A vertex may belong to boundary segments with differing classification numbers ("street numbers"). To make this information accessible the* vertex_bound *component is now a bit-mask, see also Compatibility Note 3.2.8 and Section 3.2.4. The bit-mask has* 256 *slots. If bit i in* vertex_bound[v] *is set, then vertex number v is located on the boundary segment with classification number i. Bit* 0 *has a special meaning: if it s* not *set, then the vertex is an interior vertex, in order to allow for a fast check whether the vertex is a boundary vertex at all.*

Macros and inline functions which simplify the handling of the multi-bit bit-masks BNDRY_FLAGS *are described in Section 3.2.4.*

edge_bound Only present for DIM_MAX > 2. The boundary classification of a given edge. Compare the remarks in Compatibility Note 3.2.2 above.

projection pointers for possible projection of new nodes during refinement. projection[1], if set, applies to all new nodes. projection[1+nr] ($0 \leq nr \leq$ N_WALLS(dim)) applies to new nodes on specific walls and overrides projection[0]. For details see Section 3.2.14. NULL pointers signify no projection for the given case.

index The index of this macro element.

neigh neigh[i] pointer to the macro element opposite the i-th local vertex; it is a pointer to NULL if the vertex/edges/faces opposite the i-th local vertex belongs to the boundary.

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

neigh_vertices If this is a periodic mesh and wall number w in the macro-element is part of a periodic boundary, then neigh_vertices[w] is the tuple of local vertex numbers in the periodic neighbour the vertices of wall number w are mapped onto. This corresponds to the combinatoric face-transformations specified in the macro-triangulation file format (see 3.32) and the MACRO_DATA structure (see 3.41).

wall_trafo If non-NULL, then wall_trafo[w] is the geometrical face-transformation which maps the current mesh onto its periodic neighbour across the wall number w.

np_vertex_bound Non-periodic version of the component vertex_bound, see above. If the mesh carries a periodic structure, then it is nonetheless possible to use a non-periodic mesh-traversal and define non-periodic finite element spaces.

np_edge_bound Non-periodic version of the structure component edge_bound, see above. See also the remarks to np_vertex_bound above.

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.

orientation orientation of a tetrahedron — depending on the vertex numbering, this is +1 or -1 (only 3d).

master In the presence of trace-meshes (aka "sub-meshes") master gives the link to the macro-element of the ambient "master"-mesh containing the trace-mesh this MACRO_EL-structure belongs to. The current (trace)-element is the wall numbered master.opp_vertex in the ambient master.macro_el. See Section 3.9.

3.2.6 The EL data structure

The elements of the binary trees and information that should be present for tree elements are stored in the data structure:

```
typedef struct el      EL;

struct el
{
  EL          *child[2];
  DOF         **dof;
  S_CHAR      mark;
  REAL        *new_coord;

#if ALBERTA_DEBUG
  int         index;
```

```
#endif
};
```

The members yield following information:

child pointers to the two children for interior elements of the tree; `child[0]` is a pointer to `NULL` for leaf elements; `child[1]` is a pointer to user data on leaf elements if the user is storing data on leaf elements, otherwise `child[1]` is also a pointer to `NULL` for leaf elements (see Section 3.2.10).

dof vector of pointers to DOFs; these pointers may be available for the element vertices; for the edges (in 2d and 3d), for the faces (in 3d), and for the barycenter; 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, and finally those at the barycenter, if present; the offsets are defined in the `MESH` structure (see Sections 3.2.12, 3.4.1, 3.4.2).

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 3.4.1, 3.4.2).

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 `NULL` 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 `ALBERTA_DEBUG` is `true`.

3.2.7 The `EL_INFO` data structure

The `EL_INFO` data structure has entries for all information which is not stored on elements explicitely, but may be generated by the mesh traversal routines; most entries of the `EL_INFO` structure are only filled if requested (see Section 3.2.17).

```
typedef struct el_info    EL_INFO;

struct el_info
{
  MESH              *mesh;
  REAL_D            coord[N_VERTICES_MAX];
  const MACRO_EL    *macro_el;
  EL                *el;
  const EL_INFO     *parent;
  FLAGS             fill_flag;
  int               level;

  S_CHAR            macro_wall[N_WALLS_MAX];

  BNDRY_TYPE        wall_bound[N_WALLS_MAX];
  BNDRY_FLAGS       vertex_bound[N_VERTICES_MAX];
#if DIM_MAX > 2
  BNDRY_FLAGS       edge_bound[N_EDGES_MAX];
#endif
```

```
  const NODE_PROJ *active_projection;

  EL              *neigh[N_NEIGH_MAX];
  S_CHAR          opp_vertex[N_NEIGH_MAX];
  REAL_D          opp_coord[N_NEIGH_MAX];

  U_CHAR          el_type;
  S_CHAR          orientation;

  struct {
    EL              *el;
    int             opp_vertex;
    REAL_D          opp_coord;
    U_CHAR          el_type;
    S_CHAR          orientation;
  } master, mst_neigh;

  EL_GEOM_CACHE   el_geom_cache;
};
```

The members yield the following information:

mesh A pointer to the current mesh, this information is always present.

coord `coord[i]` is a `DIM_OF_WORLD` vector storing the Cartesian coordinates of the `i`-th vertex. This information is only present if the component `fill_flag` contains the flag `FILL_COORDS`.

macro_el The current element belongs to the binary tree located at the macro element `macro_el`. This information is always present.

el Pointer to the current element. This information is always present.

parent `el` is a child of element `parent`. This information is always present.

3.2.3 Compatibility Note. *In previous versions ALBERTA,* `parent` *was just a pointer of type* `EL *`*, now it is a pointer to the* `EL_INFO` *structure of the parent element.*

fill_flag Actually, the bit-wise "or" of multiple fill-flags, indicating which elements are called and which information should be present (see Section 3.2.17) in the `EL_INFO`-structure. *Note that components of the* `EL_INFO` *structure which are not flagged as valid by* `fill_flag` *need not be initialized and may contain random data.*

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

macro_wall `macro_wall[nr]` contains the number of the wall in the ambient macro-element the wall numbered `nr` of the current element is located at, or -1 if that wall is an interior wall (with respect to the ambient macro element). This piece of information is only present when `fill_flag` contains the flag `FILL_MACRO_WALLS`.

wall_bound The boundary classification of the walls of the current element. See also Compatibility Note 3.2.1. This piece of information is only valid if `fill_flag` contains the flag `FILL_BOUND`. *Note that is not necessary to request* `FILL_BOUND` *to access the boundary classification of the walls of the current element; this is done more efficiently by requesting* `FILL_MACRO_WALLS` *and then calling the function* `wall_bound(el_info, wall)`.

3.2.4 Compatibility Note. *In previous versions of ALBERTA the* `EL_INFO` *structure also optionally contained the boundary classification of "walls", but using the names* `vertex_bound` *for 1d-meshes,* `edge_bound` *for 2d meshes and* `face_bound` *for 3d meshes. As this was extremely unhandy a new name "wall" was introduced to refer to co-dimension* 1 *simplices (the name "face" was unluckily already occupied and "defined" to refer to faces of tetrahedra in 3d).*

`vertex_bound` Boundary classification of the vertices. This piece of information is only valid if `fill_flag` contains the flag `FILL_BOUND`.

3.2.5 Compatibility Note. *This is now a bit-field of type* `BNDRY_FLAGS`*. See also Compatibility Note 3.2.1.*

`edge_bound` Boundary classification of the edges ($d > 1$). This piece of information is only valid if `fill_flag` contains the flag `FILL_BOUND`.

3.2.6 Compatibility Note. *This is now a bit-field of type* `BNDRY_FLAGS`*. See also Compatibility Note 3.2.1.*

`active_projection` If not `NULL`, a pointer to the projection function which is used to project the newly created vertex during refinement.

`neigh` `neigh[i]` pointer to the element opposite the `i`-th local vertex; it is a pointer to `NULL` if the wall opposite the `i`-th local vertex belongs to the boundary. This piece of information is only present if `fill_flag` contains the flag `FILL_NEIGH`.

`opp_vertex` `opp_vertex[i]` is undefined if `neigh[i]` is a pointer to `NULL`; otherwise it is the local index of the neighbour's vertex opposite the common wall. This piece of information is only present if `fill_flag` contains the flag `FILL_NEIGH`.

`opp_coord` `opp_coord[i]` coordinates of the `i-th` neighbour's vertex opposite the common wall. This piece of information is only present if `fill_flag` contains the flag `FILL_OPP_COORDS`.

`el_type` The element's type (see Section 3.4.1); is filled automatically by the traversal routines (only 3d).

`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).

`master` If the current element belongs to a co-dimension 1 trace-mesh (aka "slave-mesh", "sub-mesh") then this data-structure contains information concerning the element of the master-mesh the current element belongs to. This piece of information is only valid if `fill_flag` contains the flag `FILL_MASTER_INFO`.

- **`el`** Always filled with `FILL_MASTER_INFO`.
- **`opp_vertex`** Always filled with `FILL_MASTER_INFO`.
- **`opp_coord`** Only filled if `FILL_COORD` is also set in `fill_flag`
- **`el_type`** Always filled with `FILL_MASTER_INFO`, if the master-mesh is 3d.
- **`orientation`** Always filled with `FILL_MASTER_INFO`, if the master-mesh is 3d.

`mst_neigh` Same information as `master`, but for neighbour across the slave element. Only filled if `fill_flag` contains `FILL_MASTER_NEIGH`.

el_geom_cache A storage area which is used to cache various geometric quantities of the current element, like the determinant of the transformation to the reference element, the normals of the walls of te current element. The data should only be accessed through the function `fill_el_geom_cache(el_info, fill_flags)`, see Section 3.2.8.

3.2.8 Caching of geometric element quantities

Often it would be useful to share data like the determinant of the transformation to the reference element or the derivative of that transformation between pieces of program-code which are separated by call-hierarchies, or maybe one simply does not want to blow-up the parameter lists of application provided function hooks. To this aim there exists a caching mechanism, called `EL_GEOM_CACHE`, which should only be accessed and is filled by calls to `fill_el_geom_cache()`. The reader is also referred to the documentation of `fill_quad_el_cache()`, Section 4.2.6, especially in the context of parametric meshes of higher polynomial order. Example 4.2.1 contains a simplistic example for both, `fill_el_geom_cache()` and `fill_quad_el_cache()`. The element-cache structure and the related definitions and proto-types are as follows:

```
typedef struct el_geom_cache EL_GEOM_CACHE;

struct el_geom_cache
{
  EL       *current_el;
  FLAGS    fill_flag;
  REAL     det;
  REAL_BD  Lambda;
  int      orientation[N_WALLS_MAX][2];
  int      rel_orientation[N_WALLS_MAX];
  REAL     wall_det[N_WALLS_MAX];
  REAL_D   wall_normal[N_WALLS_MAX];
};

#define FILL_EL_DET     (1 << 0)
#define FILL_EL_LAMBDA  (1 << 1)

#define FILL_EL_WALL_SHIFT(wall)        (2 + 4*(wall))
#define FILL_EL_WALL_MASK(wall)         (0x7 << FILL_EL_WALL_SHIFT(wall))

#define FILL_EL_WALL_DET(wall)              (1 <<
    (FILL_EL_WALL_SHIFT(wall)+0))
#define FILL_EL_WALL_NORMAL(wall)           (1 <<
    (FILL_EL_WALL_SHIFT(wall)+1))
#define FILL_EL_WALL_ORIENTATION(wall)      (1 <<
    (FILL_EL_WALL_SHIFT(wall)+2))
#define FILL_EL_WALL_REL_ORIENTATION(wall) (1 <<
    (FILL_EL_WALL_SHIFT(wall)+3))

#define FILL_EL_WALL_DETS                           \
  (FILL_EL_WALL_DET(0)|FILL_EL_WALL_DET(1)|         \
   FILL_EL_WALL_DET(2)|FILL_EL_WALL_DET(3))

#define FILL_EL_WALL_NORMALS                                \
  (FILL_EL_WALL_NORMAL(0)|FILL_EL_WALL_NORMAL(1)|           \
   FILL_EL_WALL_NORMAL(2)|FILL_EL_WALL_NORMAL(3))
```

```
#define FILL_EL_WALL_ORIENTATIONS                                    \
  (FILL_EL_WALL_ORIENTATION(0)|FILL_EL_WALL_ORIENTATION(1)|          \
   FILL_EL_WALL_ORIENTATION(2)|FILL_EL_WALL_ORIENTATION(3))

#define FILL_EL_WALL_REL_ORIENTATIONS                                \
  (FILL_EL_WALL_REL_ORIENTATION(0)|FILL_EL_WALL_REL_ORIENTATION(1)|  \
   FILL_EL_WALL_REL_ORIENTATION(2)|FILL_EL_WALL_REL_ORIENTATION(3))

static inline const EL_GEOM_CACHE *
fill_el_geom_cache(const EL_INFO *el_info, FLAGS fill_flag);
```

The members of `EL_GEOM_CACHE` have the following meaning:

`current_el` For internal use only.

`fill_flag` A bit-mask, bit-wise or of the fill flags listed above (3.21).

`det` The determinant of the transformation to the reference element. Filled by `fill_el_geom_cache(..., FILL_EL_DET)`. This is the cached value of the quantity computed by `el_det()`, see Section 4.1.

`Lambda` The derivative of the barycentric coordinates w.r.t. the Cartesian coordinates. Filled by `fill_el_geom_cache(..., FILL_EL_LAMBDA)`. This is the cached value of the quantity computed by `el_grd_lambda()`, see Section 4.1.

`orientation` An (absolute) orientation of the walls of the current element and its neighbour. `orientation[wall][0]` is the orientation of the wall of the current element, `orientation[wall][1]` is the orientation of the same wall, but relative to the neighbour. Filled by `fill_el_geom_cache(..., FILL_EL_WALL_ORIENTATION(wall))`. These are the cached values of two calls to `wall_orientation()`, see Section 4.1.

`rel_orientation` `rel_orientation[wall]` is the cached value of `wall_rel_orientation()`, see Section 4.1. Filled by `fill_el_geom_cache(..., FILL_EL_WALL_REL_ORIENTATION(wall))`.

`wall_det` The cached return value of `get_wall_normal)()`, see Section 4.1. Filled by `fill_el_geom_cache(..., FILL_EL_WALL_DET(wall))`.

`wall_det` The cached value of the quantity computed by `get_wall_normal)()`, see Section 4.1. Filled by `fill_el_geom_cache(..., FILL_EL_WALL_NORMAL(wall))`.

3.2.9 The `INDEX` macro

It is often very helpful — especially during program development — for every element to have 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.

In the debug versions of the ALBERTA libraries (`ALBERTA_DEBUG==1`) an element index is available. The macro

```
INDEX(el)
```

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

3.2.10 Application data on leaf elements

As mentioned in Section 1.2, it is often necessary to provide access to special user data which is needed only on leaf elements. Error indicators give examples for such data.

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. ALBERTA only needs the size of memory that is required to store leaf data. During refinement and coarsening ALBERTA automatically allocates and deallocates memory for user data on leaf elements. 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 `NULL`, but since we u:se a binary tree the pointer to the second child must be `NULL` if the pointer to the first child is a `NULL` 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 self-defined data type defined).

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 `NULL` pointer. Thus testing whether an element is a leaf element or not must only be done using the pointer to the first child. If no leaf data is stored on the mesh then the pointer to the second child is also a `NULL` 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.

Leaf data storage may be initialized only once for any given mesh. Please note that leaf data is not stored when exporting meshes to disk (see Section 3.3.8).

The following function initializes leaf data:

```
size_t init_leaf_data(MESH *mesh, size_t size,
                      void (*refine_leaf_data)(EL *parent, EL *child[2]),
                      void (*coarsen_leaf_data)(EL *parent, EL *child[2]));
```

`mesh` pointer to the mesh on which leaf data is to be stored

`size` size of memory area for storing leaf data; ALBERTA may increase the size of leaf data in order to guarantee an aligned memory access.

`refine_leaf_data` pointer to a function for transformation of leaf data during refinement; first, `refine_leaf_data(parent, child)` transforms leaf data from the parent to the two children if `refine_leaf_data` is not `NULL`; after that leaf data of the parent is destroyed.

`coarsen_leaf_data` pointer to a function for transformation of leaf data during coarsening; first, `coarsen_leaf_data(parent, child)` transforms leaf data from the two children to

the parent if `refine_leaf_data` is not `NULL`; after that leaf data the of the children is destroyed.

The 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])
```

The first macro `IS_LEAF_EL(el)` is true for leaf elements and false for elements inside the binary tree; for leaf elements, `LEAF_DATA(el)` returns a pointer to leaf data hidden at the pointer to the second child.

3.2.11 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 1d the patch is built from the current element only, 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 in 3d.

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 1d the vector has length `1`, in 2d 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_INFO       el_info;
  int           no;
  int           flag;
  RC_LIST_EL    *neigh[2];
  int           opp_vertex[2];
};
```

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

`el_info` information for element corresponding to this `RC_LIST_EL` structure. This is not a pointer since `EL_INFO` structures are often overwritten during mesh traversal.

`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 `NULL` 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).

This `RC_LIST_EL` vector is one argument to the interpolation and restriction routines for DOF vectors (see Section 3.3.3).

3.2.12 The MESH data structure

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

```
struct mesh
{
  const char    *name;

  int           dim;

  int           n_vertices;
  int           n_elements;
  int           n_hier_elements;

  int           n_edges;            /* Only used for dim > 1  */
  int           n_faces;            /* Only used for dim == 3 */
  int           max_edge_neigh;     /* Only used for dim == 3 */

  bool          is_periodic;   /* true if it is possible to define periodic*/
  int           per_n_vertices;/* DOF_ADMINS on this mesh. The per_n_...    */
  int           per_n_edges;   /* entries count the number of quantities on*/
  int           per_n_faces;   /* the periodic mesh (i.e. n_faces counts    */
                               /* periodic faces twice, n_per_faces not).   */
  AFF_TRAFO     *const*wall_trafos;
  int           n_wall_trafos;

  int           n_macro_el;
  MACRO_EL      *macro_els;

  REAL_D        bbox[2]; /* bounding box for the mesh */
  REAL_D        diam;    /* bbox[1] - bbox[0] */

  PARAMETRIC    *parametric;

  DOF_ADMIN     **dof_admin;
  int           n_dof_admin;

  int           n_dof_el;             /* sum of all dofs from all admins */
  int           n_dof[N_NODE_TYPES]; /* sum of vertex/edge/... dofs from
                                      * all admins
                                      */
  int           n_node_el;    /* number of used nodes on each element */
  int           node[N_NODE_TYPES]; /* index of first vertex/edge/... node*/

  unsigned int  cookie;     /* changed on each refine/coarsen. Use
                             * this to check consistency of meshes
                             * and DOF vectors when reading from
                             * files.
                             */

  void          *mem_info; /* pointer for administration; don't touch! */
};
```

The members yield following information:

name string with a textual description for the mesh, or NULL. Note that name will be duplicated by calling strdup(3) by the GET_MESH() call.

dim dimension d of the mesh. May be any number from 0 to DIM_OF_WORLD. Zero dimensional

meshes are simply isolated vertices lacking most of the features of 1d/2d/3d meshes. They were implemented for completeness.

n_vertices number of vertices of the mesh.

n_elements number of leaf elements of the mesh.

n_hier_elements number of all elements of the mesh.

n_edges number of edges of the mesh (2d and 3d).

n_faces number of faces of the mesh (3d).

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 (3d).

is_periodic a boolean value, set to `true` for periodic meshes, see Section 3.10.

per_n_vertices, per_n_edges, per_n_faces the respective quantities, but counted taking the periodic structure into account, `n_faces`, e.g., counts periodic faces twice, `per_n_faces` not.

wall_trafos, n_wall_trafos The geometric face transformation defining the periodic structure of the mesh, see Section 3.10.

n_macro_el number of macro elements.

macro_els pointer to the macro element array.

bbox the bounding box of the mesh.

diam diameter of the mesh in the `DIM_OF_WORLD` directions.

parametric is a pointer to `NULL` if the mesh contains no parametric elements; otherwise it is a pointer to a `PARAMETRIC` structure containing coefficients of the parameterization and related information; for more information see Section 3.8.

The last entries are used for the administration of DOFs and are explained in Section 3.3 in detail.

dof_admin vector of `dof_admin`s.

n_dof_admin number of `dof_admin`s.

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; if no DOFs are associated to the barycenter, then this value is 0.
- **n_dof[CENTER]** number of DOFs at the barycenter; if no DOFs are associated to the barycenter, then this value is 0.
- **n_dof[EDGE]** number of DOFs at an edge; if no DOFs are associated to edges, then this value is 0 (2d and 3d);
- **n_dof[FACE]** number of DOFs at a face; if no DOFs are associated to faces, then this value is 0 (3d);

node gives the index of the first node at vertex, edge (2d and 3d), face (3d), and barycenter:

- **node[VERTEX]** always has value 0; `dof[0],...,dof[N_VERTICES-1]` are always DOFs at the vertices, if DOFs are located at vertices.

node[CENTER] dof[node[CENTER]] are the DOFs at the barycenter, if DOFs are located at the barycenter.

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 (2d and 3d);

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 (3d);

The cookie value is automatically initialized with a random value if ALBERTA_DEBUG==0 and with a fixed number for ALBERTA_DEBUG==1. It is incremented on each mesh change (refinement or coarsening). On writing meshes or finite element coefficient vectors to disk the current cookie value is also stored. The purpose is to provide a safety check on reading meshes and vectors; if the cookies do not match, then ALBERTA issues a warning message since no guarantee can be given that coefficient vector and mesh will match.

Finally, the pointer mem_info is used for internal memory management and must not be changed.

3.2.13 Initialization of meshes

It is possible to handle any number of meshes of any dimension $d \leq n$ in a given simulation. A mesh must be allocated by the following function or macro

```
check_and_get_mesh(int dim, int dow, int neigh,
                   const char *version, const char *name,
                   const MACRO_DATA *macro_data,
                   NODE_PROJ *(*init_node_proj)(MESH *, MACRO_EL *, int),
                   AFF_TRAFO *(*init_wall_trafo)(MESH *, MACRO_EL *, int
                       wall));
#define GET_MESH(dim, name, macro_data, init_node_proj, init_wall_trafo) \
  check_and_get_mesh((dim), DIM_OF_WORLD, ALBERTA_DEBUG,                 \
                     ALBERTA_VERSION, (name), (macro_data),              \
                     (init_node_proj), (init_wall_trafos))
```

Descriptions

check_and_get_mesh(dim, dow, debug, version, name, macro_data, init_node_proj, init_wall_trafos)

Return a pointer to a filled mesh structure; several consistency checks are performed. The application should not change any entry in the returned structure. There is no other possibility to define new meshes inside ALBERTA. The arguments dow, debug and version are checked against the constants in the used library; if these values are identical, the mesh is allocated, otherwise an error message is produced and the program stops.

parameters

dim Desired dimension of the mesh ($1 \leq$ dim $\leq \min\{$DIM_OF_WOLRD, $3\}$).

dow Must be DIM_OF_WORLD.

debug Must be ALBERTA_DEBUG.

version Must be ALBERTA_VERSION.

name A string holding a textual description of mesh and is duplicated at the member name of the mesh.

macro_data A pointer to the desired macro triangulation, see Section 3.2.15 for details.

init_node_proj Optional, may be NULL. A pointer to a function that will perform the initialization of new vertex projections, see Section 3.2.14.

init_wall_trafos Optional, may be NULL. A pointer to a function which initializes face transformations in the context of periodic meshes.

GET_MESH(dim, name, macro_data, init_node_proj, init_wall_trafos)
Return a pointer to a filled mesh structure; this macro calls check_and_get_mesh() and automatically supplies this function with the three (missing) arguments; this macro should always be used for creation of meshes.

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

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

Description:

free_mesh(mesh) will de–allocate all memory used by mesh (elements, DOFs, etc.), and finally the data structure mesh too. Submeshes of this mesh are also freed, see also Section 3.9.

3.2.14 Projection of new nodes

During refinement of simplices ALBERTA usually places the new nodes at the midpoint of the refinement edge. Some applications require meshes having curved boundaries parametrized by a given continuous function. For these it is possible to automatically project new nodes on the boundary using this function. As the mesh is refined the curved interface is successively better approximated. Figure 3.2 illustrates 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.

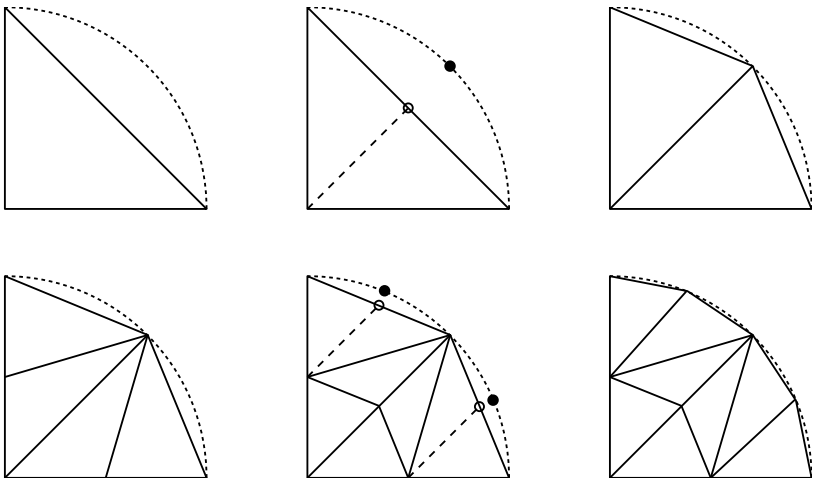


Figure 3.2: Refinement at curved boundary: refinement edge midpoints ∘ are projected to the curved boundary •

ALBERTA implements this in a very general way. It is possible to not only project nodes to boundaries, but also to arbitrary interfaces in the interior of the mesh. It is even possible

to project *all* new nodes of the mesh to a given surface, making it possible to triangulate parametrized embedded surfaces or curves.

The following type is used to describe node projection functions:

```
typedef struct node_projection  NODE_PROJECTION;
struct node_projection
{
  void (*func)(REAL_D old_coord, const EL_INFO *eli, const REAL_B lambda);
};
```

The component `func` must overwrite the given coordinate vector `old_coord` with the projected coordinates. As an alternative to world coordinates, the function may use the barycentric coordinates `lambda` describing a position on the element `eli`. The result must always be returned as world coordinates in the vector `old_coord`, however.

The idea is that the user provides a callback function `init_node_proj` during mesh initialization. This function must decide which vertices/edges/faces (for 1d/2d/3d) of which macro elements are to belong to the parametrized interface. All nodes belonging to the interface are automatically projected during refinement. ALBERTA calls `init_node_proj` several times for each macro element and thus builds the `projection` entries of the `MACRO_EL` structures, see Section 3.2.5.

During the allocation of a mesh with `check_and_get_mesh()`, see Section 3.2.13, the user may pass the function pointer `init_node_proj`. This function has the following form:

```
NODE_PROJECTION *init_node_proj(MESH *mesh, MACRO_EL *mel, int case);
```

Description:

`mesh` pointer to the mesh

`mel` pointer to the macro element

`case` ALBERTA calls `init_node_proj` once with `case==0` and additionally for `case==1` to `case==N_NEIGH(mesh->dim)+1` if `dim` $\in \{2, 3\}$.

If `init_node_proj` returns a `NODE_PROJECTION` for `case==0`, then all new nodes will be projected. If `init_node_proj` returns a `NODE_PROJECTION` for `case` $\in \{1, \ldots, $`N_NEIGH(dim)` $+ 1\}$, `dim` $\in \{2, 3\}$, then all new nodes on edge/face `case-1` will be projected. This overrides the `case==0` projection, if also set. A `NULL` value represents no projection.

3.2.7 Example (Triangulation of a unit ball)**.** The following code demonstrates the projection of boundary nodes to the unit sphere in any dimension.

```
static void ball_proj_func(REAL_D vertex, const EL_INFO *eli,
                           const REAL_B lambda)
{
  REAL norm = NORM_DOW(vertex);

  norm = 1.0 / MAX(1.0E-15, norm);
  SCAL_DOW(norm, vertex);

  return;
}

static NODE_PROJECTION *init_node_proj(MESH *mesh, MACRO_EL *mel, int c)
```

```
{
  static NODE_PROJECTION ball_proj = {ball_proj_func};

  if(c > 0 && !mel->neigh[c-1])
    return &ball_proj;
  else
    return nil;
}
```

3.2.15 Reading and writing macro triangulations

Data for macro triangulations can easily be stored in an `ASCII`-file (for binary macro files, see the end of this section). For the macro triangulation file we use a similar key-data format like for the parameter initialization (see Section 3.1.4.1). A line containing a `':'`-character defines a key. The key consists of all characters from the start of line up to the `':'`-char, including spaces. Everything after the colon potentially contains data, either on the same line or on the following lines. Data following a `'#'`-character is ignored, `'#'` is the comment-character. The following template lists all possible keys with a brief description of the data format. Luckily, an application does not need to specify all of the key-value pairs in all cases. A simple example is given further below, see Example 3.2.9, 3.2.10 and 3.2.11 below.

Macro-file template

```
# _This_ is a comment, introduced by a hash mark
DIM:            dim
DIM_OF_WORLD:   dow

number of vertices: nv
number of elements: ne

vertex coordinates:
# Comments may be mixed with data
# _This_ line and the line above are comments
<DIM_OF_WORLD coordinates of vertex[0]>
...
<DIM_OF_WORLD coordinates of vertex[nv-1]>

element vertices:
<N_VERTICES(dim) indices of vertices of simplex[0]>
...
<N_VERTICES(dim) indices of vertices of simplex[ne-1]>

element boundaries:
<N_NEIGH(dim) boundary descriptions of simplex[0]>
...
<N_NEIGH(dim) boundary descriptions of simplex[ne-1]>

element neighbours:
<N_NEIGH(dim) neighbour indices of simplex[0]>
...
<N_NEIGH(dim) neighbour indices of of simplex[ne-1]>

element type:
<element type of simplex[0]>
```

```
...
<element type of simplex[ne-1]>

number of wall transformations: <number of generators>

wall transformations:
<data for first group generator, an affine isometry in projective notation>
...
<data for last group generator, an affine isometry in projective notation>

element wall transformations:
<N_WALLS(dim) wall-transformations for simplex[0]>
...
<N_WALLS(dim) wall-transformations for simplex[ne-1]>

number of wall vertex transformations: <number of transformations>

wall vertex transformations:
<first mapping between periodic walls, identifying vertex indices>
...
<last mapping between periodic walls, identifying vertex indices>
```

Key-value descriptions 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. Generally, ordering of data is of little importance except that the `DIM` and `DIM_OF_WORLD` keys must come first, and that "natural" dependencies must be obeyed: the number of entities (vertices, elements, etc.) has to be specified before the data defining those entities, and data attached to entities must be defined after defining the entities it is attached to (e.g. neighbourhood relations have to be defined after defining the elements of the mesh).

`DIM` Mandatory. The mesh dimension.

`DIM_OF_WORLD` Mandatory. Dimension of the ambient space. The parameter `DIM_OF_WORLD` must match the libary value of `DIM_OF_WORLD`. By these values it is checked whether the provided data matches the versions of the ALBERTA-libraries in use. ALBERTA supports `DIM_OF_WORLD` > 3 (but only meshes of dimension up to 3). ALBERTA-libraries with higher co-dimension can be selected through switches for the `configure`-script prior to compiling the ALBERTA-package.

`number of vertices` Mandatory. Number of vertex coordinates following the `vertex coordinates` keyword. The number of vertices must be specified prior to defining the coordinates themselves.

`number of elements` Mandatory. The number of elements of the macro triangulation. This must be specified before defining any other data attached to elements, like the mesh connectivity or the neighbourhood relations.

`vertex coordinates` Mandatory. The coordinates, specified by tuples of floating point values of dimension `DIM_OF_WORLD`.

element vertices Mandatory. The mesh connectivity. The simplices are defined by their vertices, specified as offsets into the coordinate data defined in the `vertex coordinates` section. Counting starts at 0, so the first vertex has the number 0. The data-line `0 3 4`, e.g., would define a triangle defined by the vertices 0, 3 and 4. Note that the ordering of vertices defines the refinement edge, which is always located between the vertices with the local number 0 and 1. This ordering of vertices (and the element type for 3d) determines the distribution of the refinement edges for the children.

element boundaries Optional. For each element one line, which assigns a number between 0 and 255 (respectively −128 and +127) to each co-dimension 1 sub-simplex of each element. The `element boundaries`-key may be omitted. If this is the case, each boundary segment is assigned a number of 1. Note that interior walls have to be assigned a value of 0.

In the context of periodic meshes, periodic boundaries can still carry a classification number. This number is accessible in the `MACRO_EL`-structure (see 3.18) and during non-periodic mesh-traversal.

3.2.8 Compatibility Note. *If the macro file contains boundary "types", then those are treated as mere "street numbers" by the current ALBERTA version. Previous versions used positive numbers to indicate that a given boundary segment was subject to Dirichlet boundary condition and negative numbers were used to indicate that the respective segment carried natural boundary conditions.*

This was dropped because

1. *the macro-triangulation should carry geometric information only*
2. *it imposed too many restrictions, especially for the case were different components of systems of differential equations may be subject to different kind of boundary conditions on the same boundary segment*

Therefore the new scheme is now to only provide a classification of boundary segments by the macro triangulation. The interpretation of this classification is then left to the application program.

Vertices (2d) and edges (3d) may in fact belong to boundary segments with different "street numbers". This information is for example accessible through the function `get_bound()`*, see Section 4.7.1.3. See also Section 3.2.4.*

The current ALBERTA versions prefer positive boundary-types, the `BNDRY_TYPE` *data type is in fact an* `unsigned char` *at the moment. Negative boundary type from "old" macro-data files are interpreted as positive numbers by the usual 2-complement arithmetic.*

element neighbours Optional. Neighbourhood relationships. This information may be omitted from the macro-triangulation in which case it is computed by ALBERTA. This computation is costly for large triangulations, so if neighbourhood information is available, it is advisable to include it in the macro-triangulation if the macro triangulation is a mesh with many simplices. If given then for each wall of each element the number of the neighbouring element has to be specified, or −1 if there is no such neighbour.

`element type` Optional. This key is relevant only for 3d. In 3d, each element carries a "type" between 0 an 2 (inclusive). This type influences the mesh refinement algorithm, see Section 1.1.1. If the `element type` key is omitted, then ALBERTA assigns each macro-element a type of 0.

`number of wall transformations` Optional. The number of face transformations which define a periodic structure on the mesh. See below under `wall transformations`.

`wall transformations` Optional. For ALBERTA, a periodic mesh is (part of) the fundamental domain of a crystallographic group. A fundamental domain of such group comes with a dedicated set of generators of the crystallographic group: the face-transformations which map the current fundamental domain to its neighbour across a given face (the notion "face" is already occupied within ALBERTA, denoting co-dimension 1 face-simplices in 3d, so "wall" denotes what "face" should have been used for: a co-dimension 1 face-simplex, separating a simplex from its direct neighbour).

The group-generators have to be specified in projective notation, acting on column vectors. For example, a simple translation by an amount of 2 in x_2-direction in 3d would be specified as

```
1 0 0 0
0 1 0 2
0 0 1 0
0 0 0 1
```

ALBERTA assumes that the group generators are affine isometries, consequently, the inverses of the generators need not be specified. It is not necessary to format the matrices as shown above, ALBERTA reads as many white-space separated numbers as it needs. See also Section 3.10.

`element wall transformations` Optional. The corresponding information is computed by ALBERTA when reading the macro-file if it is omitted. If specified, the data defines for each wall of each element the index into the array of wall-transformations which maps the current mesh to its periodic neighbour across the given wall. Per convention counting starts at 1, where negative numbers denote the inverse of the given wall-transformation. A number of 0 indicates that the specific wall does not carry a face transformation (this applies to all interior walls as well as non-periodic parts of the boundary). See also Section 3.10.

`number of wall vertex transformations` Optional. Number of combinatoric face transformations. See below `wall vertex transformations`.

`wall vertex transformations` Optional, computed on the fly if omitted. If specified the data following this key defines combinatoric face transformation by mapping boundary faces – given by the global number of their vertices – onto other boundary faces. For instance, to map a 2d boundary face – an edge – connecting vertex 0 and 1 onto the boundary edge between the vertices numbered 6 and 7 the following data would have to be specified:

```
0 6
1 7
```

The ordering is important. Above lines implies that vertex 0 is identified with vertex number 6 and vertex number 1 is identified with vertex 7 – or that the corresponding edges are identified with the orientation implied by the given ordering of the vertices. See also Section 3.10.

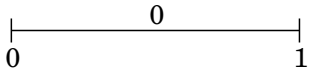
3.2.9 Example (The standard triangulation of the unit interval in $\mathbb{R}^1$)**.** The easiest example is the macro triangulation for the interval $(0, 1)$ in 1d. We just have one element and two vertices.

```
DIM: 1
DIM_OF_WORLD: 1

number of elements: 1
number of vertices: 2

element vertices:
0 1

vertex coordinates:
 0.0 0.0
 1.0 0.0
```



Macro triangulation of the unit interval.

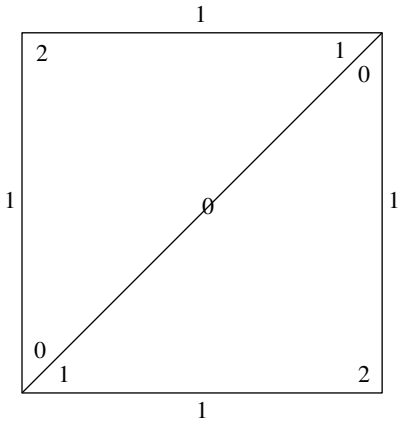
3.2.10 Example (The standard triangulation of the unit square in $\mathbb{R}^2$)**.** Still rather simple is the macro triangulation for the unit square $(0, 1) \times (0, 1)$ in 2d. Here, we have two elements and four vertices. The refinement edge is the diagonal for both elements.

```
DIM: 2
DIM_OF_WORLD: 2

number of elements: 2
number of vertices: 4

element vertices:
2 0 1
0 2 3

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



Macro triangulation of the unit square.

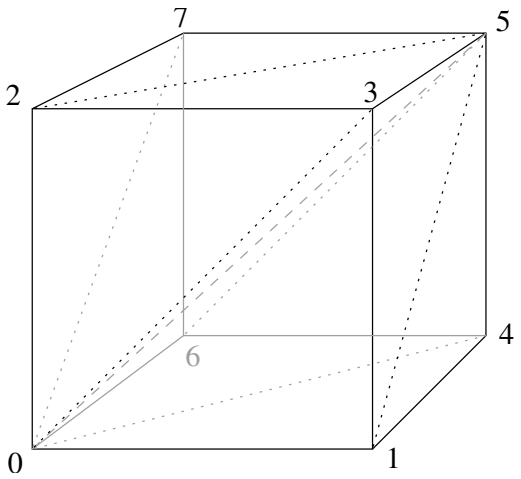
3.2.11 Example (The standard triangulation of the unit cube in $\mathbb{R}^3$)**.** More involved is already the macro triangulation for the unit cube $(0, 1)^3$ in 3d. Here, we have eight vertices and six elements, all meeting at one diagonal; the shown specification of `element vertices` prescribes this diagonal as the refinement edge for all elements.

```
DIM:            3
DIM_OF_WORLD:   3

number of vertices: 8
number of elements: 6

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

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



Macro triangulation of the unit cube.

3.2.12 Example (A triangulation of three quarters of the unit disc)**.** Here, we describe a more complex example where we are dealing with a curved boundary and mixed type boundary condition. Due to the curved boundary, we have to initialize the projection mechanism when allocating a mesh as described in Section 3.2.14. The actual projection is easy to implement, since we only have normalize the coordinates for nodes belonging to the curved boundary. We assume that the two straight edges belong to the Neumann boundary, and the curved boundary is the Dirichlet boundary. For handling mixed boundary types we have to specify `element boundaries` in the macro triangulation file. Information about `element boundaries` is also used inside the function `init_node_proj`.

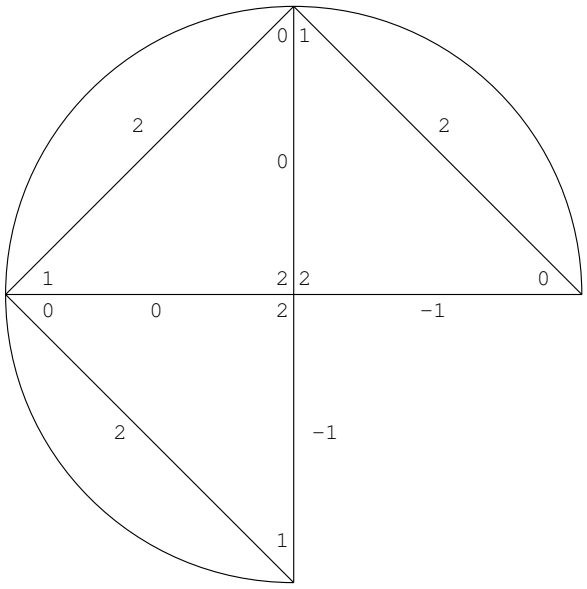
```
DIM: 2
DIM_OF_WORLD: 2

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.

The function `init_node_proj()` to initialize projection of nodes can be implemented similarly to Example 3.2.7. The projection routine `ball_proj_func` remains the same.

```
static NODE_PROJECTION *init_node_proj(MESH *mesh, MACRO_EL *mel, int c)
{
  static NODE_PROJECTION ball_proj = {ball_proj_func};

  if(c > 0 && mel->edge_bound[c-1] == 2)
    return &ball_proj;
  else
    return nil;
}
```

3.2.15.1 Reading macro triangulations from a file

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

```
MACRO_DATA *read_macro(const char *filename);
```

Description:

`read_macro(filename)` reads data of the macro triangulation from the `ASCII`-file `filename` and returns a pointer to a filled `MACRO_DATA` structure (see Section 3.2.16). Using index information from the file, all information concerning element vertices, neighbour relations can be calculated directly.

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. Please note that projection of nodes as described in Section 3.2.14 is only possible for new nodes arising during refinement.

A binary data format allows faster import of a macro triangulation, especially when the macro triangulation already consists of many elements. Macro data written previously by binary `write_macro` routines (see below) can be read in native or machine independent binary format by the two routines

```
MACRO_DATA *read_macro_bin(const char *filename);
MACRO_DATA *read_macro_xdr(const char *filename);
```

Description:

`read_macro_bin(filename)` reads data of the macro triangulation from the native binary file `filename`; the file `filename` was previously generated by the function `write_macro_bin()`, see below.

`read_macro_xdr(filename)` reads data of the macro triangulation from the machine independent binary file `filename`, the file `filename` was previously generated by the function `write_macro_xdr()`, see below.

3.2.15.2 Dumping macro triangulations to a file

The counterpart of functions for reading macro triangulations are functions for writing macro triangulations to file. To be more general, it is possible to create a macro triangulation from the triangulation given by the leaf elements of a mesh. As mentioned above, it can be faster to use a binary format than the textual formal for writing and reading macro triangulations with many elements.

```
int write_macro(MESH *, const char *);
int write_macro_bin(MESH *, const char *);
int write_macro_xdr(MESH *, const char *);
```

Description:

`write_macro(mesh, name)` writes the triangulation given by 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`.

`write_macro_bin(mesh, name)` writes the triangulation given by the leaf elements of `mesh` as a macro triangulation to the file specified by `name` in native binary format.

`write_macro_xdr(mesh, name)` writes the triangulation given by the leaf elements of `mesh` as a macro triangulation to the file specified by `name` in machine independent binary format.

For exporting meshes including the whole hierarchy, see Section 3.3.8

3.2.16 Import and export of macro triangulations from/to other formats

When meshes are created using a simplicial grid generation tool, then data will usually not be in the ALBERTA macro triangulation format described above in Section 3.2.15. In order to simplify the import of such meshes, an array-based data structure `MACRO_DATA` is provided, using flat `C`-arrays for storing the data, and indirect index-arrays to bind the data to elements and define the mesh connectivity. Such a data structure can easily be filled by an import routine; the filled data structure can then converted into an ALBERTA mesh. Of course, another possibility is to convert the data to ALBERTA's textual macro-file format as described in Section 3.2.15 above. The `MACRO_DATA` structure is defined as

```
typedef struct macro_data MACRO_DATA;

struct macro_data
{
  int dim;                /* dimension of the elements */

  int n_total_vertices;
  int n_macro_elements;

  REAL_D *coords;         /* Length will be n_total_vertices */

  int *mel_vertices;      /* mel_vertices[i*N_VERTICES(dim)+j]:
                           * global index of jth vertex of element i
                           */

  int *neigh;             /* neigh[i*N_NEIGH(dim)+j]:
                           * neighbour j of element i or -1 at boundaries
                           */
  int *opp_vertex;        /* opp_vertex[i*N_NEIGH(dim)+j]: if set (need not
                           * be) the local vertex number w.r.t. the neighbour
                           * of the vertex opposit the separating wall.
                           */
  BNDRY_TYPE *boundary;   /* boundary[i*N_NEIGH(dim)+j]:
                           * boundary type of jth co-dim 1 facet of element i
                           *
                           * WARNING: In 1D the local index corresponds
                           * to vertex 1 & vice versa! (Consistent with
                           * macro_data.neigh)
                           */
  U_CHAR *el_type;        /* el_type[i]: type of element i only used in 3d! */

  /********* the remainder is only needed for periodic meshes **********/

  int (*wall_vtx_trafos)[N_VERTICES(DIM_MAX-1)][2]; /* the wall trafos */
  /* Wall transformations are in terms of mappings between
   * vertices. i-th wall trafo: global vertex number
   * wall_vtx_trafos[i][v][0] maps to wall_vtx_trafos[i][v][1],
   * v loops through the local vertex number of the respective wall.
   */
  int n_wall_vtx_trafos;/* for periodic meshes: number of
                         * combinatorical wall trafos.
                         */
  int *el_wall_vtx_trafos;
  /* el_wall_vtx_trafos[i*N_WALLS(dim)+j] number of the wall
   * transformation of the j-th wall for the i-th element. > 0:
```

```
     * #wall_trafo+1. < 0: inverse of -(#wall_trafo+1)
     */
    AFF_TRAFO *wall_trafos; /* The group generators of the space group
                             * defining the periodic structure of the
                             * mesh.
                             */
    int n_wall_trafos;
    int *el_wall_trafos; /* N = el_wall_trafos[i*N_NEIGH(dim)+j]:
                          *
                          * number of the wall transformation mapping to
                          * the neighbouring fundamental domain across
                          * the given wall.
                          *
                          * If negative: inverse of generator -N-1
                          * If positive:           generator +N-1
                          */
  };
```

The members yield following information:

dim dimension of the triangulation.

n_total_vertices number of vertices.

n_macro_elements number of mesh elements.

coords REAL_D array of size n_total_vertices holding the point coordinates of all vertices.

mel_vertices integer array of size n_macro_elements * N_VERTICES(dim) storing element index information; mel_vertices[i*N_VERTICES[dim]+j] is the index of the jth vertex of element i.

neigh integer array of size n_macro_elements*N_NEIGH(dim), where neigh[i*N_NEIGH(dim)+j] is the index of the jth neighbour element of element i, or -1 in case of a boundary.

boundary S_CHAR array of size n_macro_elements*N_NEIGH(dim), where boundary[i*N_NEIGH(dim)+j] is the boundary type of the jth vertex/edge/face of element i (in 1d/2d/3d). Please note that the index 0 corresponds to vertex 1 and vice versa in 1d, consistent with the numbering used for neigh.

el_type a U_CHAR vector of size n_macro_elements holding the element type of each mesh element (only 3d).

wall_vtx_trafos, n_wall_vtx_trafos correspond to the data specified with the key wall vertex transformations, see Section 3.2.15. This field stores face-transformations in terms of mappings between vertices. For the i-the face transformation the global vertex number wall_vtx_trafos[i][v][0] maps to the global vertex number wall_vtx_trafos[i][v][1], v loops through the *local* vertex number of the respective wall.

el_wall_vtx_trafos If el_wall_vtx_trafos[i*N_WALLS(dim)+j]! = 0 then it is the number of the face-transformation the j-th wall on the for the i-th element is subject to. Negative number indicate that the inverse of the respective face-transformation is attached to that wall. Note that one has to subtract 1 from this value before using it as index into wall_vtx_trafos, because arrays in C are indexed starting with 0.

wall_trafos, n_wall_trafos The group generators and their number of the space group defining the periodic structure of the mesh. See Section 3.10.

el_wall_trafos If `N = el_wall_trafos[i*N_NEIGH(dim)+j]` $! = 0$ then N is the number of the face-transformation mapping the mesh to the neighboring fundamental domain across the given wall. If N is negative, then the actual face-transformation is the inverse of the N-th transformation. Note that one has to subtract 1 from this value before using it as index into `wall_trafos`, because arrays in `C` are indexed starting with 0.

A `MACRO_DATA` structure can be allocated and freed by

```
MACRO_DATA *alloc_macro_data(int dim, int nv, int ne, FLAGS);
void free_macro_data(MACRO_DATA *);
```

Description:

alloc_macro_data(dim, n_vertices, n_elements, flags) allocates a `dim`-dimensional `MACRO_DATA` structure together with all arrays needed to hold `n_vertices` vertices and `n_elements` mesh elements. The `coords` and `mel_vertices` arrays are allocated in any case, while `neigh`, `boundary` and `el_type` arrays are allocated only when requested as indicated by the corresponding flags `FILL_NEIGH`, `FILL_BOUNDARY`, and `FILL_EL_TYPE` set by a bitwise OR in `flags`.

free_macro_data(data) frees all previously allocated storage for `MACRO_DATA data` and all the arrays in it.

Once `MACRO_DATA` structure is filled, it can be saved to file in the ALBERTA macro triangulation format, or it can be directly be converted into a `MESH`.

```
void macro_data2mesh(MESH *mesh, const MACRO_DATA *data,
                     NODE_PROJECTION *(*n_proj)(MESH *,MACRO_EL *,int));
int write_macro_data(MACRO_DATA *, const char *);
int write_macro_data_bin(MACRO_DATA *, const char *);
int write_macro_data_xdr(MACRO_DATA *, const char *);
```

Description:

macro_data2mesh(mesh, macro_data, n_proj) converts the triangulation with data given in `macro_data` into a `MESH` structure. It sets most entries in `mesh`, allocates macro elements needed, assigns DOFs according to `mesh->n_dof`, and calculates `mesh->diam`. The coordinates in `macro_data->coords` are copied to a newly allocated array, thus the entire `MACRO_DATA` structure can be freed after calling this routine. When not `nil`, the `n_proj` function is used to initialize projection of new nodes.

write_macro_data(macro_data, name) writes the macro triangulation with data stored in `macro_data` in the ALBERTA format described in Section 3.2.15 to file `name`. The return value is `0` when an error occured and `1` in case the file was written successfully.

write_macro_data_bin(macro_data, name) writes data of the macro triangulation stored in `macro_data` in native binary format to file `name`; the return value is `0` when an error occured and `1` in case the file was written successfully.

write_macro_data_xdr(macro_data, name) writes data of the macro triangulation stored in `macro_data` in machine independent binary format to file `name`; the return value is `0` when an error occured and `1` in case the file was written successfully.

It is appropriate to check whether a macro triangulation given in a `MACRO_DATA` structure allows for recursive refinement, by testing for possible recursion cycles. An automatic correction by choosing other refinement edges may be done, currently implemented only in 2d.

```
void macro_test(MACRO_DATA *, const char *);
```

Description:

macro_test(macro_data, name) checks the triangulation given in `macro_data` for potential cycles during recursive refinement. In the case that such a cycle is detected, the routine tries to correct this by renumbering element vertices (which is currently implemented only in 2d) and then writes the new, changed triangulation using `write_macro_data()` to a file `name`, when the second parameter is not `nil`.

3.2.17 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. The 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: `CALL_EVERY_EL_PREORDER` operates first on a parent element before operating on both children, `CALL_EVERY_EL_POSTORDER` operates first on both children before operating on their parent, and `CALL_EVERY_EL_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 (see also Section 3.2.7):

FILL_NOTHING no information needed at all.

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

FILL_BOUND the boundary classification EL_INFO.wall_bound, EL_INFO.vertex_bound and EL_INFO.edge_bound (in 2d and 3d) are filled. If an application only needs the boundary classification of the walls of an element, then it is probably more efficient to request the FILL_MACRO_WALLS fill-flag and call bndry_type = wall_bound(el_info, wall) to obtain this information.

FILL_NEIGH neighbour element information EL_INFO.neigh and EL_INFO.opp_vertex is generated.

FILL_OPP_COORDS information about opposite vertex coordinates EL_INFO.opp_coords of neighbours is filled; the flag FILL_OPP_COORDS can only be selected in combination with FILL_COORDS|FILL_NEIGH.

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

FILL_PROJECTION information about projection routines for new vertices is generated using this flag. The entries EL_INFO.active_projection are set.

FILL_MACRO_WALLS the mapping of the local wall-numbers (i.e. faces in 3d, edges in 2d, points in 1d) to the numbering of the walls on the ambient macro-element is maintained during mesh-traversal. The entries EL_INFO.macro_wall are set.

FILL_NON_PERIODIC for periodic meshes, ignore the periodic structure when computing the neighborhood relations and the boundary classification.

FILL_MASTER_INFO for trace-meshes (AKA "slave-meshes"). During mesh-traversal on the trace-mesh generate certain information about the ambient "master"-element. Certain fields of EL_INFO.master are valid, depending on which other traversal flags are set.

FILL_MASTER_NEIGH for trace-meshes, implies FILL_MASTER_INFO explained above. For trace-meshes sliding through an ambient bulk-mesh additionally compute information about the neighbour of the ambient master-element across the wall forming the element on the trace-mesh. Certain fields of EL_INFO.master are valid, depending on which other traversal flags are set.

FILL_ANY macro definition for a bitwise OR of any possible fill flags, used for separating the fill flags from the CALL_... flags.

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

```
void fill_macro_info(MESH *, const MACRO_EL *, EL_INFO *);
void fill_elinfo(int, const EL_INFO *, EL_INFO *);
```

Description:

fill_macro_info(mesh, mel, el_info) fills el_info with macro element information of mel required by el_info->flag and sets el_info->mesh to mesh;

fill_elinfo(ichild, parent_info, el_info) fills el_info for the child ichild using hierarchy information and parent data parent_info depending on parent_info->flag.

3.2.17.1 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 the next macro element in the array `mesh->macro_els`.
- For every `EL el`, all elements in the subtree `el->child[0]` are visited before any element in the subtree `el->child[1]`.
- The traversal order of an element and its two child trees is determined by the flags `CALL_EVERY_EL_PREORDER`, `CALL_EVERY_EL_INORDER`, and `CALL_EVERY_EL_POSTORDER`, as defined above in Section 3.2.17.

This order can only be changed by explicitly calling the `traverse_neighbour()` routine during non-recursive traversal, see below.

3.2.17.2 Recursive mesh traversal routines

Recursive traversal of mesh elements is done by the routine

```
void mesh_traverse(MESH *,int,FLAGS,void (*)(const EL_INFO *,void *),void *);
```

Description:

`mesh_traverse(mesh, level, fill_flag, el_fct, data)` 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 in the `fill_flag`; 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`. The pointer `data` is used for opaque user data that should be made available to the `el_fct` routine.
It is possible to use the recursive mesh traversal recursively, by calling `mesh_traverse()` from `el_fct`.

3.2.13 Example. An example of a mesh traversal is the computation of the measure of the computational domain. On each leaf element, the volume of the element is computed by the library function `el_volume()` and added to a global variable `measure_omega`, which finally holds the measure of the domain after the mesh traversal.

```
static void measure_el(const EL_INFO *el_info, void *measure_omega)
{
  *((int *)measure_omega) += el_volume(el_info);
  return;
}

...

measure_omega = 0.0;
mesh_traverse(mesh, -1, CALL_LEAF_EL|FILL_COORDS, measure_el,
    &measure_omega);
MSG("|Omega| = %e\n", measure_omega);
```

`el_volume()` computes the element volume and thus needs information about the elements vertex coordinates.

3.2.14 Example. We give an implementation of the `CALL_EVERY_EL_...` routines to show the simple structure of all recursive traversal routines. A data structure `TRAVERSE_INFO`, only used by the traversal routines, holds the traversal flag and a pointer to the element function `el_fct()`:

```
static void recursive_traverse(EL_INFO *el_info, TRAVERSE_INFO *trinfo)
{
  EL       *el = el_info->el;
  EL_INFO el_info_new;

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

    fill_elinfo(0, el_info, &el_info_new);
    recursive_traverse(&el_info_new, trinfo);

    if (trinfo->flag & CALL_EVERY_EL_INORDER)
      trinfo->el_fct(el_info, trinfo->data);

    fill_elinfo(1, el_info, &el_info_new);
    recursive_traverse(&el_info_new, trinfo);

    if (trinfo->flag & CALL_EVERY_EL_POSTORDER)
      trinfo->el_fct(el_info, trinfo->data);
  }
  else
  {
    trinfo->el_fct(el_info, trinfo->data);
  }
  return;
}

static void mesh_traverse_every_el(MESH *mesh, FLAGS fill_flag,
                                   void (*el_fct)(const EL_INFO *, void *),
                                   void *data);
{
  EL_INFO        el_info;
  TRAVERSE_INFO traverse_info;
  int            n;

  el_info.fill_flag = (flag & FILL_ANY);
  el_info.mesh = mesh;

  traverse_info.mesh    = mesh;
  traverse_info.el_fct = el_fct;
  traverse_info.flag    = flag;
  traverse_info.data    = data;

  for(n = 0; n < mesh->n_macro_el; n++) {
    fill_macro_info(mesh->macro_els + n, &el_info);
    recursive_traverse(&el_info, &traverse_info);
  }
```

```
  return;
}
```

3.2.17.3 Non–recursive mesh traversal routines

Some applications may profit from or actually require a non–recursive form of mesh traversal, where the element routine gets pointers to visited elements, one after another. For example, mesh refinement and coarsening routines (see Sections 3.4.1 and 3.4.2), the gltools and GRAPE graphic interface (see Sections 4.11.2 and 4.11.3) are functions which use a non–recursive access to the mesh elements.

Note that currently non-recursive level-based traversal indicated by the traversal flags `CALL_EL_LEVEL`, `CALL_LEAF_EL_LEVEL` or `CALL_MG_LEVEL` is not implemented.

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).

```
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. For the non–recursive mesh traversal no pointer to an element function `el_fct()` has to be provided, because all operations are done by the routines which call the traversal functions. A mesh traversal is launched by each call to `traverse_first()` which also initializes the traverse stack. Advancing to the next element is done by the function `traverse_next()`. The following non–recursive routines are provided:

```
TRAVERSE_STACK *get_traverse_stack(void);
void free_traverse_stack(TRAVERSE_STACK *staci);
const EL_INFO *traverse_first(TRAVERSE_STACK *stack, MESH *, int level,
    FLAGS fill_flags);
const EL_INFO *traverse_next(TRAVERSE_STACK *stack, const EL_INFO *el_info);
TRAVERSE_FIRST(MESH *mesh, int level, FLAGS fill_flags);
TRAVERSE_NEXT();
const EL_INFO *subtree_traverse_first(TRAVERSE_STACK *stack,
                                      const EL_INFO *local_root,
                                      int level, FLAGS flags);
```

Descriptions:

`get_traverse_stack()` returns a pointer to a data structure `TRAVERSE_STACK`.

`free_traverse_stack(stack)` frees the traverse stack `stack` previously accessed by `get_traverse_stack()`.

`traverse_first(stack, mesh, level, fill_flag)` launches the non–recursive mesh traversal; the return value is a pointer to an `el_info` structure of the first element to be visited;

`stack` is a traverse stack previously accessed by `get_traverse_stack()`;

`mesh` is a pointer to a mesh to be traversed, `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;

`fill_flag` specifies 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;

`traverse_next(stack, el_info)` returns an `EL_INFO` structure with data about the next element of the mesh traversal or a pointer to `NULL`, if `el_info->el` is the last element to be visited;
information which elements are visited and which data has to be filled is accessible via the traverse stack `stack`, initialized by `traverse_first()`. After calling `traverse_next()`, all `EL_INFO` information about previous elements is invalid, the structure may be overwritten with new data.

`TRAVERSE_FIRST(mesh, level, fill_flags), TRAVERSE_NEXT()` are convenience macros which internally call the functions `get_traverse_stack()`, `traverse_first()`, `traverse_next()` and `free_traverse_stack()`. `TRAVERSE_FIRST()` defines a local variable with name `el_info` which holds the information about the current element.

`subtree_traverse_first(stack, local_root, level, flags)` Like `traverse_first()`, but restricts the traversal to the sub-tree starting at `local_root`. Note that `local_root` is saved on the traverse-`stack`, so it is possible to initiate a sub-tree traversal from within the recursive `mesh_traverse()` routines with this function, or from within another non-recursive traversal loop, if that uses another `TRAVERSE_STACK`.

Usually, the interface to a graphical environment uses the non–recursive mesh traversal, compare the gltools (Section 4.11.2) and GRAPE interfaces (Section 4.11.3).

3.2.15 Example. The computation of the measure of the computational domain with the non–recursive mesh traversal routines is shown in the following code segment.

```
REAL measure_omega(MESH *mesh)
{
  TRAVERSE_STACK *stack = get_traverse_stack();
  const EL_INFO  *el_info;
  FLAGS          fill_flag;
  REAL           measure_omega = 0.0;

  el_info = traverse_first(stack, mesh, -1, CALL_LEAF_EL|FILL_COORDS);
  while (el_info)
  {
    measure_omega += el_volume(el_info);
    el_info = traverse_next(stack, el_info);
  }
  free_traverse_stack(stack);

  return(measure_omega);
}
```

3.2.16 Example. The same example as above, but implemented with the convenience macros `TRAVERSE_FIRST()`, `TRAVERSE_NEXT()`:

```
REAL measure_omega(MESH *mesh)
{
  REAL measure_omega = 0.0;

  TRAVERSE_FIRST(mesh, -1, CALL_LEAF_EL|FILL_COORDS) {
    measure_omega += el_volume(el_info);
```

```
  } TRAVERSE_NEXT();
  return measure_omega;
}
```

3.2.17.4 Neighbour traversal

Some applications, like the recursive refinement algorithm, need the possibility to jump from one element to another 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 one element to a neighbour is

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

Description:

`traverse_neighbour(stack, el_info, i)` returns a pointer to an `EL_INFO` structure with information about the `i`-th neighbour opposite the `i`-th vertex of `el_info->el`;

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

Calling `traverse_neighbour()`, all `EL_INFO` information about a previous element is invalid, and can only be regenerated 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 `NULL`; nevertheless, information from the old `EL_INFO` may be overwritten and lost. To avoid such behavior, one should check for boundary vertices/edges/faces (1d/2d/3d) before calling `traverse_neighbour()`.

3.2.17.5 Access to an element at world coordinates x

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. In a characteristic method, the point x is usually given by $x = x_0 - \boldsymbol{V}\tau$, where x_0 is the starting point, $\boldsymbol{V}$ the advection and τ the time step size. For points x_0 close to the boundary it may happen that x does not belong to the computational domain. In this situation it is convenient to know the point on the domain's boundary which lies on the line segment between the old point x_0 and the new point x. This point is uniquely determined by the scalar value s such that $x_0 + s\,(x - x_0) \in \partial\text{Domain}$.

The following function accesses an element at world coordinates x:

```
int find_el_at_pt(MESH *, const REAL_D, EL_INFO **, FLAGS, REAL [N_LAMBDA],
                  const MACRO_EL *, const REAL_D, REAL *);
```

Description:

`find_el_at_pt(mesh, x, el_info_p, fill_flag, bary, start_mel, x0, sp)` fills element information in an `EL_INFO` structure and corresponding barycentric coordinates of the element where the point `x` is located; the return value is `true` if `x` is inside the domain, or `false` otherwise. Arguments of the function are:

`mesh` is the mesh to be traversed;

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

`el_info_p` is the return address for a pointer to the `EL_INFO` for the element at `x` (or when `x` is outside the domain but `x0` was given, of the element containing the point $x_0 + s\,(x - x_0) \in \partial\text{Domain}$);
`fill_flag` are the flags which specify which information should be filled in the `EL_INFO` structure, coordinates are included in any case as they are needed by the routine itself;
`bary` pointer where to return the barycentric coordinates of `x` on `*el_info_p->el` (or, when `x` is outside the domain but `x0` was given, of the point $x_0 + s\,(x - x_0) \in \partial\text{Domain}$);
`start_mel` an initial guess for the macro element containing `x`, or `NULL`;
`x0` starting point of a characteristic method, see above, or `NULL`;
`sp` return address for the relative distance to domain boundary in a characteristic method if `x0 != nil`, see above, or `NULL`.

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 4.1. 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 `x` and then finds its path through the corresponding element tree based on the macro barycentric coordinates. For domains with curved boundary, it is possible that in some cases a point inside the domain is considered as external.

3.3 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. 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, 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. One of the main aspects of the DOF administration is to keep track of all used and unused indices. One possibility to remove holes from 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 3.3.2.

In ALBERTA, 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 3.3.1) 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 3.2.12. The FE_SPACE structure describing a finite element space references the corresponding set of DOFs, compare Sections 1.4.2, 3.5.1. Several FE_SPACEs 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 3.6.2. For special applications, additional DOF sets, that are not connected to any finite element space may also be defined (compare Section 3.6.2).

In Sections 3.3.5 and 3.3.6, 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 3.3.2 and 3.3.4. Several BLAS routines are available for such data, see Section 3.3.7.

3.3.1 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 long     DOF_FREE_UNIT;

/* Possible values for DOF_ADMIN->flags */
# define ADM_FLAGS_DFLT             0         /* nothing special */
# define ADM_PRESERVE_COARSE_DOFS   (1 << 0)  /* preserve non-leaf DOFs */
# define ADM_PERIODIC               (1 << 1)  /* periodic ADMIN on a
                                               * periodic mesh
                                               */
#define ADM_FLAGS_MASK (ADM_PRESERVE_COARSE_DOFS | ADM_PERIODIC)

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 */

  FLAGS         flags;
```

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

  int   n_dof[N_NODE_TYPES];   /* dofs from THIS dof_admin */
  int   n0_dof[N_NODE_TYPES]; /* start of THIS admin's DOFs in the mesh. */
  /****************************************************************************/
  DOF_INT_VEC      *dof_int_vec;          /* linked list of int vectors */
  DOF_DOF_VEC      *dof_dof_vec;          /* linked list of dof vectors */
  DOF_DOF_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_PTR_VEC      *dof_ptr_vec;          /* linked list of void * vectors */
  DOF_MATRIX       *dof_matrix;           /* linked list of matrices */

  DBL_LIST_NODE    compress_hooks;        /* linked list of custom compress
                                           * handlers.
                                           */

 /****************************************************************************
  * pointer for administration; don't touch!
  ****************************************************************************/

  void             *mem_info;
};
```

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;

flags The bit-wise or of flags controlling the behavior of the DOF-administrator:

ADM_PERIODIC The DOF-administrator identifies DOFs across periodic boundaries, compare Section 3.10.

ADM_PRESERVE_COARSE_DOFS Do not delete DOFs on the coarse-levels of the mesh during mesh-refinement. This must be set to implement, e.g., multi-grid methods for higher order elements. See also Section 3.4.1.1 andSection 3.4.1.

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 3.3.6. These values are usually set by `get_fe_space()` as a copy from `bas_fcts->n_dof` (compare Section 3.5.1).

n0_dof start indices n0_dof[VERTEX/CENTER/EDGE/FACE] of the first dofs defined by this dof_admin in the element's dof[VERTEX/CENTER/EDGE/FACE] 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/CENTER/EDGE/FACE], are all set automatically by get_fe_space(). See Section 3.3.6 for details and usage;

dof_*_vec, dof_matrix pointers to linked lists of DOF_*_VEC, 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 3.3.2;

compress_hooks Root to a doubly linked list of custom handlers executed when dof_compress() is called. An application may install arbitrarily many custom compress-handlers via add_dof_compress_hook(), and delete them via del_dof_compress_hook(). See further below.

mem_info used internally for memory management.

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 3.3.5 checks this case. Hole removal is done for all DOF_ADMINs of a mesh by the function

```
void dof_compress(MESH *);

typedef struct dof_comp_hook DOF_COMP_HOOK;
struct dof_comp_hook
{
  DBL_LIST_NODE node; /* our link to the compress_hooks list */
  void (*handler)(DOF first, DOF last, const DOF *new_dof, void *app_data);
  void *application_data;
};

void add_dof_compress_hook(const DOF_ADMIN *admin, DOF_COMP_HOOK *hook);
void del_dof_compress_hook(DOF_COMP_HOOK *hook);
```

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.

add_dof_compress_hook(admin, hook), del_dof_compress_hook(hook) Add to or delete from the list of application defined DOF-compress functions. dof_compress() will

call all installed handlers in turn. The calling convention for the `handler(first, last, new_dof[], app_data)` component in the `DOF_COMP_HOOK`-structure are:

`first, last` Bounds for the index range where something has changed. The application may assume that all DOFs outside the index range `first,...,last` have not been renumbered.

`new_dof[]` The index permutation, `new_dof[old_dof]` is the new index assigned to the DOF with the old number `old_dof`.

`app_data` This is the structure-component `application_data` stored in the `DOF_COMP_HOOK`-structure.

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 *, int);
```

Description:

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

3.3.2 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 3.3.3.

ALBERTA includes data types for vectors of type `REAL`, `REAL_D`, `S_CHAR`, `U_CHAR`, `int`, and `void *`. Below, the `DOF_REAL_VEC`, structure is described in detail. Structures `DOF_REAL_D_VEC`, `DOF_SCHAR_VEC`, `DOF_UCHAR_VEC`, `DOF_PTR_VEC`, and `DOF_INT_VEC` are declared similarly, the only difference between them is the type of the structure entry `vec`. The exception is the `DOF_REAL_VEC_D` type, which is used to model vector-valued finite element functions where the underlying basis functions are either vector- or scalar-valued.

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;

  DOF             size;
  int             reserved; /* stride for DOF_REAL_VEC_D */

  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);

  DBL_LIST_NODE     chain;        /* chain link for direct sum of fe-spaces */
  const DOF_REAL_VEC *unchained;

  EL_REAL_VEC_D     *vec_loc;

  void              *mem_info;
};
```

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 the vector, or `NULL`;

size current size of `vec`;

stride the stride of `vec`. In the context of `DIM_OF_WORLD`-valued problems the underlying basis function may or may not be vector-valued for themselves. If they are scalar, `stride` is set to `DIM_OF_WORLD`, if the basis functions are vector-valued then `stride` is set to 1.

reserved A place holder in all `DOF_XXX_VEC`-structures to make sure that it is possible to cast a `DOF_REAL_VEC_D` to a `DOF_REAL_VEC` or a `DOF_REAL_D_VEC`. Note that for `DOF_REAL_VEC` structures `reserved` will internally be tied to 1, while it is tied to `DIM_OF_WORLD` for `DOF_REAL_D_VEC` structures.

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

refine_interpol, coarse_restrict interpolation and restriction routines, see Section 3.3.3. For `REAL` and `REAL_D` vectors, these usually point to the corresponding routines from `fe_space->bas_fcts`, compare Section 3.5.1. 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.

chain If the underlying finite element space has the structure of a direct sum, then this list-node component is the link to the individual components of that direct sum. See Section 3.7.

unchained The name is misleading, as explained in Section 3.7.5, the reader should, however, have a look at Section 3.7 first.

vec_loc an element-vector. This element vector is used if the corresponding `BAS_FCTS.get_real_vec()` hook is called with `result == NULL`.

mem_info private pointer for administration, must not be changed.

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 `NULL`, compare Section 3.3.3.

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 `NULL`, 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 *fesp);
DOF_REAL_D_VEC  *get_dof_real_d_vec(const char *name, const FE_SPACE *fesp);
DOF_REAL_VEC_D  *get_dof_real_vec_d(const char *name, const FE_SPACE *fesp);
DOF_INT_VEC     *get_dof_int_vec(const char *name, const FE_SPACE *fesp);
DOF_INT_VEC     *get_dof_dof_vec(const char *name, const FE_SPACE *fesp);
DOF_INT_VEC     *get_int_dof_vec(const char *name, const FE_SPACE *fesp);
DOF_SCHAR_VEC   *get_dof_schar_vec(const char *name, const FE_SPACE *fesp);
DOF_UCHAR_VEC   *get_dof_uchar_vec(const char *name, const FE_SPACE *fesp);
DOF_PTR_VEC     *get_dof_ptr_vec(const char *name, const FE_SPACE *fesp);
void            free_dof_real_vec(DOF_REAL_VEC *vec);
void            free_dof_real_d_vec(DOF_REAL_D_VEC *vec);
void            free_dof_real_vec_d(DOF_REAL_VEC_D *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);
void            free_dof_ptr_vec(DOF_PTR_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`. The `name` argument is duplicated calling `strdup(3)`.

If no finite element space is specified, then the vector will not be controlled by any `DOF_ADMIN`. In this case the user is responsible for setting `size` and allocating memory for

vec. Given these entries ALBERTA will free such vectors correctly using free_dof_*_vec. *Allocating a* DOF_REAL_VEC_D *without an underlying finite element space is not supported.*

3.3.1 Compatibility Note. *In contrast to previous ALBERTA versions the* get_dof_..._vec() *routines now make a copy of the finite element space, which is in turn deallocated upon the call to* free_dof_..._vec().

There is a special list for each type of DOF vectors in the DOF_ADMIN structure. All DOF_REAL_VECs, DOF_REAL_D_VECs, DOF_UCHAR_VECs, DOF_SCHAR_VECs, and DOF_PTR_VECs 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, ..., 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 NULL. 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 DOF vector is produced by the routines:

```
void print_dof_int_vec(const DOF_INT_VEC *vec);
void print_dof_real_vec(const DOF_REAL_VEC *vec);
void print_dof_real_d_vec(const DOF_REAL_D_VEC *vec);
void print_dof_real_vec_dow(const DOF_REAL_VEC_D *vec);
void print_dof_schar_vec(const DOF_SCHAR_VEC *vec);
void print_dof_uchar_vec(const DOF_UCHAR_VEC *vec);
void print_dof_ptr_vec(const DOF_PTR_VEC *vec);
```

Description:

print_dof_*_vec(dof_vec) prints the elements of the DOF vector dof_vec together with its name to the message stream.

3.3.3 Interpolation and restriction of DOF vectors during mesh adaptation

During mesh refinement and coarsening, new DOFs are produced, or old ones are deleted. In many cases, information stored in DOF_*_VECs 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_*_VECs with non-NULL pointers in all DOF_ADMINs 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 3.4.1 and 3.4.2, 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_ADMINs 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 of mesh elements 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 Sections 3.5.4.1-3.5.4.4.

3.3.4 The DOF_MATRIX data structure

3.3.2 Compatibility Note. *Previous versions of ALBERTA defined extra-types for vector-valued problems, like* DOF_DOWB_MATRIX, DOWB_OPERATOR_INFO *etc. The "*DOWB*" ("DimOfWorldBlocks") variants, however, already incorporated all the functionality of the ordinary scalar-only versions. Therefore the scalar-ony versions of most data-structures have been abandoned and were replaced by the "*DOWB*" variants, which in turn were renamed to use the scalar-only names. For example, in the current implementation a* DOF_MATRIX *is in fact what older versions called a* DOF_DOWB_MATRIX*; and implements the scalar-only case as well as the block-matrix case.*

Not only vectors indexed by DOFs are available in ALBERTA, but also matrices which operate on these DOF_*_VECs. 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 [18]. This is further explained below on page 127. A DOF_MATRIX structure is usually filled by local operations on single elements, using the update_matrix() routine, compare Section 4.7.1, which automatically generates space for new matrix entries by adding new MATRIX_ROWs, if needed. In view of problems which involve vector-fields of size DIM_OF_WORLD ALBERTA supports block-matrices with entries of type REAL_D and REAL_DD (as well, of course, as matrices with scalar entries).

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.

Type definitions

```
typedef struct dof_matrix DOF_MATRIX
struct dof_matrix
{
  DOF_MATRIX      *next;
  const FE_SPACE  *row_fe_space;
  const FE_SPACE  *col_fe_space;
  const char      *name;

  MATRIX_ROW      **matrix_row;  /* lists of matrix entries   */
  DOF             size;          /* size of vector matrix_row */
  MATENT_TYPE     type;          /* type of matrix entries.   */
  size_t          n_entries;     /* total number of entries in the matrix */

  BNDRY_FLAGS     dirichlet_bndry; /* bit-mask for Dirichlet b.c. */

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

  DBL_LIST_NODE   row_chain;
  DBL_LIST_NODE   col_chain;
  const DOF_MATRIX *unchained;

  void            *mem_info;
};
```

Description of the individual structure components:

next linked list of `DOF_MATRIX` structures in `row_fe_space->admin`;

row_fe_space `FE_SPACE` structure with information about corresponding row DOFs and basis functions;

col_fe_space `FE_SPACE` structure with information about corresponding column DOFs and basis functions;

name a textual description for the matrix, or `NULL`;

matrix_row vector of pointers to `MATRIX_ROW`s, one for each row, see below;

size current size of the `matrix_row` vector.

type the type of the element entries, one of `MATENT_REAL`, `MATENT_REAL_D` or `MATENT_REAL_DD`;

dirichlet_bndry a bit-mask describing which parts of the boundary should be treated as Dirichlet-boundary by `update_matrix()`;

n_entries the total number of entries currently stored in the matrix, updated by `add_element_matrix()` and reset to 0 by `clear_dof_matrix()`;

refine_interpol, coarse_restrict interpolation and restriction routines as for `DOF_*_VEC`s. When implementing interpolation or restriction routines for matrices it is up to the user to remove matrix entries corresponding to obsolete DOFs. This functionality is not implemented in ALBERTA at the moment since it would involve an expensive search over all matrix entries after mesh changes.

row_chain, col_chain List pointers, in the context of direct sums of finite element spaces a DOF_MATRIX is actually a block-matrix, where the individual blocks are again DOF_MATRIXes, each acting on a summand of the direct sum of finite element spaces. See section 3.7.

unchained Normally only a pointer to NULL, dof_matrix_sub_chain() uses this pointer to form a matrix which acts only on a part of a direct sum of finite element spaces. See Section 3.7.

mem_info private pointer for administration, should not be changed.

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 alberta.h. For $d = 2$ 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 hopefully should not produce too much overhead.

```
typedef struct matrix_row           MATRIX_ROW;
typedef struct matrix_row_real      MATRIX_ROW_REAL;
typedef struct matrix_row_real_d    MATRIX_ROW_REAL_D;
typedef struct matrix_row_real_dd   MATRIX_ROW_REAL_DD;

# define ROW_LENGTH 9

/* The actual size of this structure is determined by the type of the
 * matrix entries. The correct length is allocated in
 * get_matrix_row().
 */
# define SIZEOF_MATRIX_ROW(type)                                        \
   (sizeof(MATRIX_ROW) - sizeof(REAL_DD) + ROW_LENGTH*sizeof(type))

struct matrix_row
{
  MATRIX_ROW  *next;
  MATENT_TYPE type;
  DOF         col[ROW_LENGTH];      /* column indices */
  union {
    REAL     real[1];
    REAL_D   real_d[1];
    REAL_DD  real_dd[1];
  } entry;
};

struct matrix_row_real
{
  MATRIX_ROW_REAL *next;
  MATENT_TYPE      type;
  DOF              col[ROW_LENGTH];     /* column indices */
  REAL             entry[ROW_LENGTH];   /* matrix entries */
};
```

```
struct matrix_row_real_d
{
  MATRIX_ROW_REAL_D *next;
  MATENT_TYPE        type;
  DOF                col[ROW_LENGTH];     /* column indices */
  REAL_D             entry[ROW_LENGTH];   /* matrix entries */
};

struct matrix_row_real_dd
{
  MATRIX_ROW_REAL_DD *next;
  MATENT_TYPE         type;
  DOF                 col[ROW_LENGTH];     /* column indices */
  REAL_DD             entry[ROW_LENGTH];   /* matrix entries */
};

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

Descriptions for the individual macros and structure components:

ROW_LENGTH the maximum number of data-times in one list element,

MATRIX_ROW a super-type which combines all MATRIX_ROW data-types for all block-types. Components and their meaning:

next list node pointer

type one of MATENT_{REAL, REAL_D, REAL_DD}, specifying the block-type of the data entries

col the global DOF-indices for the entries stored in this row-component. Entries can be flagged as unused by setting MATRIX_ROW.col[idx] to UNUSED_ENTRY, if MATRIX_ROW.col[idx] == NO_MORE_ENTRIES, then this signals that the remainder of this row component does not contain any more data. The other values following such an entry are undefined.

entry A union for the actual data entries, with sub-components for each block-type. Note that get_matrix_row() actually allocates enough space to hold ROW_LENGTH many entries in each row. The actual size of the allocated matrix-row structure can be determined by calling the macro SIZEOF_MATIX_ROW(type) with type REAL, REAL_D or REAL_DD.

SIZEOF_MATRIX_ROW(type) See above.

MATRIX_ROW_{REAL, REAL_D, REAL_DD} Data-types for each individual block-type. They differ from the super-type MATRIX_ROW only in replacing the entry union by a simple field which holds ROW_LENGTH many entries of the given matrix-entry type.

Support routines The following routines are available for DOF-matrices:

```
DOF_MATRIX *get_dof_matrix(const char *name,
                           const FE_SPACE *row_fe_space,
                           const FE_SPACE *col_fe_space);
void        free_dof_matrix(DOF_MATRIX *matrix);
void        clear_dof_matrix(DOF_MATRIX *matrix);
void        print_dof_matrix(const DOF_MATRIX *matrix);
MATRIX_ROW *get_matrix_row(const FE_SPACE *fe_space, MATENT_TYPE type);
FOR_ALL_MAT_COLS(type, matrow, what)
```

Description:

`get_dof_matrix(name, row_fe_space, col_fe_space)` allocates a new `DOF_MATRIX` structure operating between the finite element spaces `col_fe_space` and `row_fe_space`. If no `col_fe_space` is given, then `col_fe_space` is set to `row_fe_space`. `name` is a textual description for the name of the new matrix, it is duplicated using `strdup(3)`. The new matrix is automatically linked into the `row_fe_space->admin->dof_matrix` list. A `matrix_row` vector of length `row_fe_space->admin->size` is allocated and all entries are set to `NULL`.

`free_dof_matrix(matrix)` frees the DOF matrix `matrix` previously accessed by the function `get_dof_matrix()`. First, all `MATRIX_ROW`s in `matrix->matrix_row` are freed, then `matrix->matrix_row`, and finally the structure `*matrix`.

`clear_dof_matrix(matrix)` clears all entries of the DOF matrix `matrix`. This is done by *removing* all entries from the DOF matrix, i. e. all `MATRIX_ROW`s in `matrix->matrix_row` are freed and all entries in `matrix->matrix_row` are set to `NULL`.

`print_dof_matrix(matrix)` prints the elements of the DOF matrix `matrix` together with its name to the message stream.

`get_matrix_row(fe_space, type)` Allocate a new `MATRIX_ROW` for a `DOF_MATRIX` with `row_fe_space == fe_space`. The second argument specifies the type of the entries and is one of `REAL`, `REAL_D` or `REAL_DD`.

`FOR_ALL_MAT_COLS(type, matrow, what)` Because the `MATRIX_ROW`-structure is somewhat complicated this defines an "iterator" over all entries of a matrix-row and hides the dirty details from the application program. The meaning of the arguments is like follows:

`type` One of `REAL`, `REAL_D` or `REAL_DD`. `type` must be the same as `DOF_MATRIX.type` respectively `MATRIX_ROW.type`, or the results will be unpredictable.

`matrow` A pointer to the matrix-row to iterator over. `row` may be `NULL`.

`what` A block of statements to execute for each entry of the matrix-row. When `what` is executed the following variables are pre-defined to give access to the data of the current entry:

`col_idx` The index into `MATRIX_ROW.data`.

`col_dof` The global `DOF`-index of the current entry.

Example 3.3.4 contains two examples for the iteration over the columns of `MATRIX_ROW`s, one using the `FOR_ALL_MAT_COLS` macro, and another one with yields the same results without the use of this macro.

3.3.5 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(const DOF_ADMIN *, todo);
FOR_ALL_FREE_DOFS(const DOF_ADMIN *, todo);
```

Description:

`FOR_ALL_DOFS(admin, todo)` loops over all used DOFs of `admin`; `todo` is a list of C-statements which are to be executed for every used DOF index. During `todo`, the local variable `int dof` holds the current index of the used entry; it must not be altered by `todo`;

`FOR_ALL_FREE_DOFS(admin, todo)` loops over all unused DOFs of `admin`; `todo` is a list of C-statements which are to be executed for every unused DOF index. During `todo`, the local variable `int dof` holds the current index of the unused entry; it must not be altered by `todo`.

In the context of direct sums of finite element spaces, there are other macros called `FOREACH_DOF(fe_space,...)` which wrap `FOR_ALL[_FREE]_DOFS()` into an outer loop over the components of the direct sum, see Section 3.7.2. Two examples illustrate the usage of the `FOR_ALL[_FREE]_DOFS()`:

3.3.3 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 `alpha` by:

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

The BLAS-1 routine `dof_set()` is written this way, compare Section 3.3.7.

3.3.4 Example (Matrix-vector multiplication)**.** As a more complex example we give the main loop from an implementation of the matrix-vector product in `dof_mv()`, compare Sections 3.3.4 and 3.3.7, specifically the explanations for `FOR_ALL_MAT_COLS()` on page 129:

```
FOR_ALL_DOFS(admin, {
    REAL sum = 0.0;
    FOR_ALL_MAT_COLS(REAL, a->matrix_row[dof], {
        sum += row->entry[col_idx] * xvec[col_dof];
      });
    yvec[dof] = sum;
  });
```

Without the use of the `FOR_ALL_MAT_COLS()`-macro, the same example looks like follows:

```
FOR_ALL_DOFS(admin, {
    REAL sum = 0.0;
    MATRIX_ROW_REAL *row;
    for (row = (MATRIX_ROW_REAL *)a->matrix_row[dof];
         row != NULL;
         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;
    });
```

3.3.6 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 3.2.6, the `EL` data structure holds a vector of pointers to DOF vectors, that contain data for *all* DOFs on the element from all `DOF_ADMIN`s:

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

The lengths of these vectors are 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 3.2.12):

```
struct mesh
{
  ...
  DOF_ADMIN **dof_admin;
  int       n_dof_admin;

  int             n_dof_el;
  int             n_dof[N_NODE_TYPES];
  int             n_node_el;
  int             node[N_NODE_TYPES];
  ...
};
```

The meaning of these entries 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`, `CENTER`, `EDGE`, and `FACE` DOFs from all `DOF_ADMIN` structures;

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

node The entry `node[i]`, `i` $\in$ {`VERTEX`, `CENTER`, `EDGE`, `FACE`} 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 calls of the subroutine `get_fe_space()` (compare Section 3.6.2).

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*, *center*, *edge*, and *face* nodes. Using the macros from Section 3.2.1, there may be `N_VERTICES(dim)` vertex nodes, `N_EDGES(dim)` edge nodes (2d or 3d), `N_FACES(dim)` 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. Each allocation of a finite element space forces ALBERTA to adjust this information. For this reason it is advisable to allocate finite element spaces before refining a mesh. 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`, `CENTER`, `EDGE`, or `FACE` (compare Figure 3.3). 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.

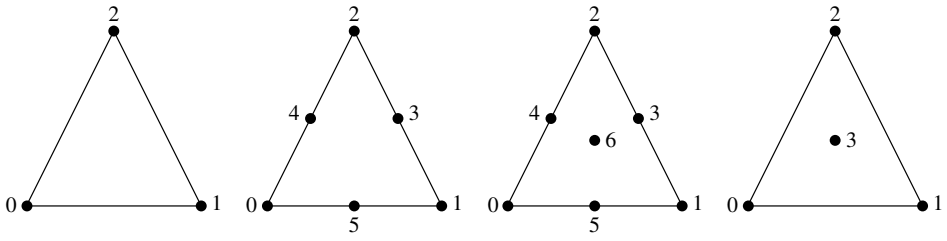


Figure 3.3: DOF vector indices in `el->dof` for DOFs at vertices, vertices and edges, vertices, edges and center, and vertices and center (in 2d). Corresponding `mesh->node` values are {0,0,0,0}, {0,0,3,0}, {0,6,3,0}, and {0,3,0,0}. Note that the indices increase according to the sequence `VERTEX`/`EDGE`/`FACE`/`CENTER` for historical reasons.

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 to DOFs for a finite element space on an element, the BAS_FCTS structure provides a routine

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

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 3.5.1).

3.3.7 BLAS routines for DOF vectors and matrices

Several basic linear algebra subroutines (BLAS [14, 6]) are implemented for DOF vectors and DOF matrices, see Table 3.3. Note that the table only lists the functions for DOF-vectors and matrices storing scalar values.

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() accept a transpose argument: transpose = NoTranspose = 0 indicates the use of the original matrix, while transpose = Transpose = 1 indicates that the transposed matrix should be used. We use the C-BLAS definition,

```
typedef enum { NoTranspose, Transpose, ConjugateTranspose } MatrixTranspose;
```

The mask argument accepted by the matrix-vector routines is a flag-vector: if specified, then the matrix operates only on those DOFs i with mask[i] != DIRICHLET, clearing all those DOFs in the result to 0, compare Section 4.7.7.1.

Analogue routines exist for DOF_REAL_D_VEC and DOF_REAL_VEC_D objects, with the convention to attach a _d- respectively a _dow-suffix for the variants dealing with DIM_OF_WORLD-valued finite element functions, e.g. for the nrm2()-function there exist the calls:

```
REAL dof_nrm2(const DOF_REAL_VEC *arg);
REAL dof_nrm2_d(const DOF_REAL_D_VEC *arg);
REAL dof_nrm2_dow(const DOF_REAL_VEC_D *arg);
```

Additionally, the matrix-vector routines are available as versions pairing vector- and scalar-valued DOF-vectors. Of course, the inner block-type to the DOF_MATRIX must match the requirements of the arguments in this case. So to multiply a scalar finite element function with a matrix, resulting in a vector-valued finite element function there exist the variants

```
void dof_mv_rdr(MatrixTranspose transpose,
                const DOF_MATRIX *a, const DOF_SCHAR_VEC *mask,
                const DOF_REAL_VEC *x, DOF_REAL_D_VEC *y);
void dof_mv_dow_scl(MatrixTranspose transpose,
                    const DOF_MATRIX *A, const DOF_SCHAR_VEC *mask,
                    const DOF_REAL_VEC *x, DOF_REAL_VEC_D *y);
```

3.3.8 Reading and writing of meshes and vectors

Section 3.2.15 described the input and output of ASCII files for macro triangulations. Locally refined triangulations including the mesh hierarchy and corresponding DOFs are saved in

`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_SCHAR_VEC *mask,`
`  const DOF_REAL_VEC *x,`
`  REAL beta, DOF_REAL_VEC *y)`
	$Y_i = (\alpha * A * X + \beta * Y)_i$
	or
	$Y_i = (\alpha * A^t * X + \beta * Y)_i$
	if `mask != NULL` and `mask[i]!=DIRICHLET`

`void dof_mv(MatrixTranspose transpose,`
`  const DOF_MATRIX *a,`
`  const DOF_SCHAR_VEC *mask`
`  const DOF_REAL_VEC *x, DOF_REAL_VEC *y)`
	$Y_i = (A * X)_i$
	or
	$Y_i = (A^t * X)_i$
	if `mask != NULL` and `mask[i]!=DIRICHLET`

Table 3.3: Implemented BLAS routines for DOF vectors and matrices. Corresponding routines for `DIM_OF_WORLD`-valued coefficient vectors are, of course, also available, see Section 3.3.7

binary 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 usually depends 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 *name, REAL time);
MESH *read_mesh(const char *name, REAL *timeptr,
                NODE_PROJECTION *(*init_node_proj)(MESH *, MACRO_EL *, int),
                MESH *master);
```

The routine `write_mesh` stores information about the mesh in a file named `name`. 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. User defined leaf data (see Section 3.2.10) is not written. If the mesh carries a parametric structure defined by `use_lagrange_parametric()` (see Section 3.8.1), then this structure will be dumped to

disk, read back by the corresponding `read_mesh()` routines. Geometric face-transformations attached to periodic meshes (compare Section 3.10) will also be dumped to disk and restored by `read_mesh()`. If `mesh` is a co-dimension 1 trace-mesh of another master-mesh (see Section 3.9), then this binding will also be dumped to disk, and can optionally be restored by a call to `read_mesh()`.

Routine `read_mesh` reads a complete mesh from file `name`, which was created by `write_mesh`. The corresponding time, if any, is stored at `timeptr`. The argument `init_node_proj` is used in the same way as in `GET_MESH()`, compare Sections 3.2.13 and 3.2.14. If the argument `master` is non-`NULL`, then ALBERTA attempts to bind the read-back mesh as a co-dimension 1 trace-mesh to this "master"-mesh, compare Section 3.9. This will only work if `master` is just in the state it had when the trace-mesh was dumped to disk, otherwise the behaviour is undefined and the application supposedly will crash very quickly; a good example which works is when both, master- and trace-mesh, are dumped to disk and restored sequentially:

```
exterrn MESH *master, *trace;

write_mesh(master, "master.mesh", HUGE_VAL);
write_mesh(trace, "trace.mesh", HUGE_VAL);

... /* other stuff, 1.000.000 lines of code later: */

master = read_mesh("master.mesh", NULL, NULL, NULL);
trace  = read_mesh("trace.mesh", NULL, NULL, master);
```

3.3.5 Compatibility Note. `read_mesh()` *is supposed to be able to read data generated by previous versions of* ALBERTA.

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

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

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

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

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

```
MESH *read_mesh_xdr(const char *name, REAL *timeptr,
                    NODE_PROJECTION *(*init_node_proj)(MESH *,MACRO_EL
                        *,int));



int write_dof_int_vec_xdr(const DOF_INT_VEC *div, const char *name);
int write_dof_real_vec_xdr(const DOF_REAL_VEC *drv, const char *name);i
int write_dof_real_d_vec_xdr(const DOF_REAL_D_VEC *drdv, const char *name);
int write_dof_real_vec_d_xdr(const DOF_REAL_VEC_D *drvd, const char *name);
int write_dof_schar_vec_xdr(const DOF_SCHAR_VEC *dsv, const char *name);
int write_dof_uchar_vec_xdr(const DOF_UCHAR_VEC *duv, const char *name);

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

All flavours of the IO-routines come also with a version which accepts a `stdio-FILE` pointer as argument. These routines are pre-fixed by the letter "`f`", in the spirit of `fprintf(3)`. For example, the corresponding proto-type for the `write_mesh_xdr()` function is

```
bool fwrite_mesh_xdr(MESH *mesh, FILE *fp, REAL time);
```

Likewise for all other functions: just replace the file-name argument by the file-pointer argument. Intentionally, this feature has been introduced to let the IO-routines act on streaming data. Note that the compatibility mode of `read_mesh()` with respect to meshes generated by ALBERTA-1.2 will not work if the actual file underlying the file-pointer does not allow for random access (e.g. if it is a pipe or socket).

3.4 The refinement and coarsening implementation

3.4.1 The refinement routines

For the refinement of a mesh the following symbolic constant is defined and the refinement is done by the functions

```
#define MESH_REFINED   1

U_CHAR refine(MESH *mesh, FLAGS fill_flags);
U_CHAR global_refine(MESH *mesh, int n_bisections, FLAGS fill_flags);
```

3.4.1 Compatibility Note. *In previous versions, the last parameter* `fill_flags` *was missing. To obtain the old behaviour,* `FILL_NOTHING` *should be passed for the parameter* `fill_flags`*.*

Description:

refine(mesh, fill_flags) refines all leaf elements 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 at least one element was refined, and 0 otherwise. Every refinement has to be done via this routine. The basic steps of this routine are described below.

Parameters

mesh The mesh which will be refined, possibly.

FLAGS fill_flags Request additional data filled in during the mesh-traversal, useful for custom mesh-adaptation call-back in DOF-vectors and -matrices. Additionally – if any mesh-adaptation call-backs have been registered – then the set of fill-flags will be augmented by the requirements of the related basis-function sets.

Return Value Either MESH_REFINED, if at least one element has been sub-divided, or 0 otherwise.

---

global_refine(mesh, num_bisections, fill_flags) 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.

Parameters

MESH *mesh The mesh which will be refined, possibly.

num_bisections The number of bisections to perform on each element. This is an upper limit: the resulting mesh will have no "hanging-nodes", this conformal closure may require more refinement steps than requested by num_bisections.

FLAGS *fill_flags Request additional data filled in during the mesh-traversal, useful for custom mesh-adaptation call-back in DOF-vectors and -matrices.

Return Value Either MESH_REFINED, if at least one element has been sub-divided, or 0 otherwise.

3.4.1.1 Basic steps of the refinement algorithm

The refinement of a mesh is principally done in two steps — each step corresponding to one mesh traversal. In the first step no coordinate information is necessary, only a topological refinement is performed. If new nodes are created that belong to a a then these can be projected in the second step where coordinate information is calculated.

Again using the notion of "refinement edge" for the element itself in 1d, the algorithm performs the following steps:

1. The whole mesh is refined only topologically. This part consists 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 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 3.3.3); these routines must not use coordinate information;
- a deallocation of DOFs on the parent when `preserve_coarse_dofs == 0`, see Section 3.6.2.

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 `active_projection()` function in the `EL_INFO` structure. This entry is passed down from the corresponding macro element during the mesh traversal of this step. 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.

The topological refinement is done by the recursive refinement Algorithm 1.1.5. In 1d, no recursion is needed. In 2d and 3d, all elements at the refinement edge of a 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 refinement patch. The elements of this patch are stored in a vector `ref_list` with elements of type `RC_LIST_EL`, compare Section 3.2.11. This vector is an argument for the functions for interpolation of DOF vectors during refinement, compare Section 3.3.3.

In 1d the vector has length `1`. In 2d the length is `2` if the refinement edge is an interior edge; for a boundary edge the length is `1` since only the element itself has to be refined. For 1d and 2d, only the `el` entry of the components is set and used.

In 3d this vector is allocated with length `mesh->max_edge_neigh`. As mentioned in Section 3.2.11 we can define an orientation of the edge and by this orientation we can define the right and left neighbors (inside the patch) of an element at this edge.

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 initialized by the user) 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 `init_leaf_data()` call). Then memory for parents' leaf data is freed and information stored there 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 3.3.3):

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 corresponding `DOF_ADMIN` 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 1d we have

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

in 2d

```
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];
```

In 3d for `child[1]` this passing of DOFs additionally depends on the element type `el_type` of the parent. 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]` and 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];
```

In 1d

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

and in 3d and 3d

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

is the newly allocated DOF at the midpoint of the refinement edge (compare Figure 1.4 on page 4 for the 1d and 2d situation and Figure 1.5 on page 5 for the 3d situation).

In the case that we have DOFs at the midpoint of edges (only 2d and 3d) 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];
```

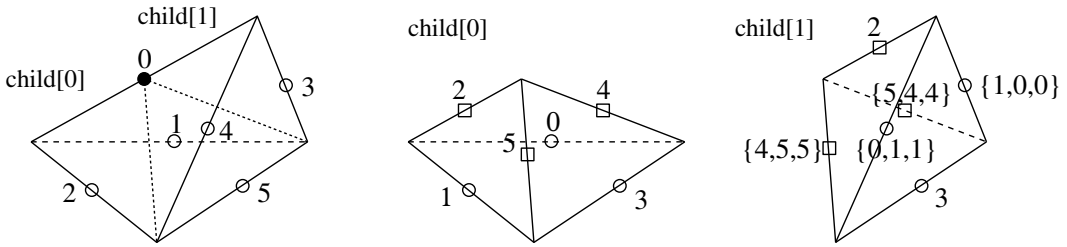


Figure 3.4: Edge DOFs that are freed •, passed on ○, and newly alloc ated □

We also have to create new DOFs (compare Figure 3.4). 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 a DOF should only be calculated once.

All DOF vectors and matrices having a pointer to a function `refine_interpol()` in the corresponding data structure are interpolated to the fine grid. Such a function essentially depends 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 3.5.4.

After such an interpolation, DOFs of higher degree on parent elements may no longer be of interest (when not using a higher order multigrid method).

In such a case `DOF_ADMIN.flags & ADM_PRESERVE_COARSE_DOFS` should evaluate to 0 and in this case all DOFs on the parent that are not handed over to the children will be removed. The following DOFs are removed on the parent for all patch elements (some DOFs are shared by several elements): The DOFs at the center

```
el->dof[mesh->node[CENTER]]
```

are removed in all dimensions. In 2d, additionally DOFs in the refinement edge

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

are removed and in 3d the DOFs in the refinement edge and the DOFs in the two faces adjacent to the refinement edge

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

are deleted on the parent. Note that only the information about DOF indices is deleted, the pointers `el->dof[n]`, `n`$\in \{0, \ldots,$ `mesh->n_node_el-1`$\}$, themselves remain available after refinement. This setting of DOF pointers and pointers to children is the main part of the refinement module.

3.4.2 The coarsening routines

For the coarsening of a mesh the following symbolic constant is defined and the coarsening is done by the functions

```
#define MESH_COARSENED 2

U_CHAR coarsen(MESH *mesh, FILL_FLAGS fill_flags);
U_CHAR global_coarsen(MESH *mesh, int num_bisections, FILL_FLAGS flags_flags);
```

Description:

`coarsen(mesh, fill_flags)` 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.

`global_coarsen(mesh, n_bisections, fill_flags)` sets all element markers for leaf elements of `mesh` to `n_bisections`; the mesh is then coarsened by `coarsen()`; depending on the actual distribution of coarsening edges on the mesh, this may not result in a `|n_bisections|` global coarsening; the return value is `coarsen(mesh)` if `mark` is negative, 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 reset. 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 DOF indices of the parents were deleted during refinement, then new indices are now allocated. DOF pointers on the parents (`parent->dof[n]`) do not need to be touched, as they remain valid after refinement, see Section 3.4.1.

If leaf data is stored at the pointer of `child[1]`, then memory for the parent's leaf data is allocated. If a function `coarsen_leaf_data` was provided during the call of `init_leaf_data()` then leaf data is transformed from children to parent. Finally, leaf data on both children is freed.

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 and matrices where such a function is available in the corresponding 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 3.5.4.

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 `NULL` and the pointer to the second child is adjusted to the `leaf_data` of the parent, or also set to `NULL`. Thus, all fine grid information is lost at that moment, which makes clear that a restriction of data has to be done in advance.

3.5 Implementation of basis functions

In order to construct a finite element space, we have to specify a set of local basis functions. 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. Together with this 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.

At the moment the following finite elements are supported by ALBERTA:

- standard Lagrange finite elements of order n, $n \in \{1, 2, 3, 4\}$;
- discontinuous polynomial elements of order n, $n \in \{0, 1, 2\}$; these are defined as arbitrary polynomials of maximal degree n on each element with no continuity restriction;
- orthonormal discontinuous polynomial elements of order 1 and 2; these functions are normalized and orthogonal w.r.t. the L^2-scalar product on the reference element.

We present these elements in the subsequent sections. A tselection of more complicated basis functions is implemented in an add-on library called `libalbas`, with the focus on stable discretizations for the Stokes-problem. This is not discussed here.

3.5.1 Data structures for basis functions

For the handling of local basis functions, i.e. a basis of the function space $\bar{\mathbb{P}}$ on the reference element (compare Section 1.4.2) we use functions of the following type. The structure describing the set of local basis functions (`BAS_FCTS`, see page 145) contains arrays of such function-pointers:

```
typedef REAL
  (*BAS_FCT)(const REAL_B lambda, const BAS_FCTS *thisptr);
typedef const REAL *
  (*GRD_BAS_FCT)(const REAL_B lambda, const BAS_FCTS *thisptr);
typedef const REAL_B *
  (*D2_BAS_FCT)(const REAL_B lambda, const BAS_FCTS *thisptr);
typedef const REAL_BB *
  (*D3_BAS_FCT)(const REAL_B lambda, const BAS_FCTS *thisptr);
typedef const REAL_BBB *
  (*D4_BAS_FCT)(const REAL_B lambda, const BAS_FCTS *thisptr);

typedef const REAL *
(*BAS_FCT_D)(const REAL_B lambda, const BAS_FCTS *thisptr);
typedef const REAL_B *
(*GRD_BAS_FCT_D)(const REAL_B lambda, const BAS_FCTS *thisptr);
typedef const REAL_BB *
(*D2_BAS_FCT_D)(const REAL_B lambda, const BAS_FCTS *thisptr);
```

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 $\lambda \in \mathbb{R}^{d+1}$ 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 $\bar{\varphi} \in \bar{\mathbb{P}}$:

$$\nabla_\lambda \bar{\varphi}(\lambda) = \left(\frac{\partial \bar{\varphi}(\lambda)}{\partial \lambda_0}, \ldots, \frac{\partial \bar{\varphi}(\lambda)}{\partial \lambda_d} \right);$$

the arguments of such a function are barycentric coordinates and the return value is a pointer to a `const REAL` vector of length `N_LAMBDA` storing $\nabla_\lambda \bar{\varphi}(\lambda)$; this vector will be overwritten during 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 $\bar{\varphi} \in \bar{\mathbb{P}}$:

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

the arguments of such a function are barycentric coordinates and the return value is a pointer to a `N_LAMBDA` $\times$ `N_LAMBDA` matrix storing $D^2_\lambda \bar{\varphi}$; this matrix will be overwritten during the next call of the function.

D3_BAS_FCT, D4_BAS_FCT serve primarily for debugging purposes and need not be present. The format is the similar to the second barycentric derivatives, the third derivatives are a tensor of rank 3, and the fourth derivatives are a tensor of rank 4.

BAS_FCT_D, GRD_BAS_FCT_D, D2_BAS_FCT_D Basis functions may optionally be `REAL_D`-valued. In this case we factor the basis functions into a scalar part which is multiplied by a direction, with the obvious implications for the derivatives of such basis functions. This is further explained below where the corresponding components of the `BAS_FCTS` structure are discussed.

```
#define PHI(bfcts, i, lambda)      (bfcts)->phi[i](lambda, bfcts)
#define GRD_PHI(bfcts, i, lambda)  (bfcts)->grd_phi[i](lambda, bfcts)
#define D2_PHI(bfcts, i, lambda)   (bfcts)->D2_phi[i](lambda, bfcts)
#define D3_PHI(bfcts, i, lambda)   (bfcts)->D3_phi[i](lambda, bfcts)
#define D4_PHI(bfcts, i, lambda)   (bfcts)->D4_phi[i](lambda, bfcts)

#define PHI_D(bfcts, i, lambda)      (bfcts)->phi_d[i](lambda, bfcts)
#define GRD_PHI_D(bfcts, i, lambda)  (bfcts)->grd_phi_d[i](lambda, bfcts)
#define D2_PHI_D(bfcts, i, lambda)   (bfcts)->D2_phi_d[i](lambda, bfcts)
```

Description:

PHI(bfcts, i, lambda) The individual basis function pointers expect that a pointer to the `BAS_FCTS`-structure they belong to is passed as last argument. To decrease the potential for coding errors we advocate to use

```
value = PHI(bfcts, nr, lambda, nr);
```

instead of using the equivalent construct

```
value = bfcts->phi[nr](lambda, bfcts);
```

The other macro work analogously.

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). Further information includes the traces of the local finite element space on the walls of reference element, which are needed to define trace-meshes (AKA sub-meshes, see Section 3.9). Also, finite element spaces may form a direct sum, e.g. to implement stable discretisations of the Stokes-problem (see Section 3.7); if this is the case then the local basis-functions also reflect this fact. Finally, basis-functions may need a per-element initializer, for example if they depend on the geometry of the mesh-simplex. Such information is stored in the `BAS_FCTS` data structure:

```
typedef struct bas_fcts BAS_FCTS;

struct bas_fcts
{
  const char    *name;          /*  textual description */
  int           dim;            /*  dimension of the corresponding mesh. */
  int           rdim;           /*  dimension of the range, 1 or DIM_OF_WORLD */
  int           n_bas_fcts;     /*  nu_mber of basisfunctions on one el */
  int           n_bas_fcts_max;/*  max. number in presence of init_element()*/
  int           degree;         /*  maximal degree of the basis functions,
                                 *  may vary on a per-element basis if
                                 *  init_element() is != NULL.
                                 */
  int           n_dof[N_NODE_TYPES];    /* dofs from these bas_fcts */
  int           trace_admin; /* If >= 0, then the basis function set
                                * needs a DOF_ADMIN living on a trace
                                * mesh with id TRACE_ADMIN.
                                */

  /************** link to next set of bfcts in a direct sum **************/
  DBL_LIST_NODE   chain;

  /* A pointer to the unchained version. It simply points back to the
   * same structure if this is an unchained basis-function
   * structure.
   */
  const BAS_FCTS *unchained;

  /*************** per-element initializer (maybe NULL) ******************/

  INIT_ELEMENT_DECL;

  /*************** the basis functions themselves        *****************/
  const BAS_FCT     *phi;
  const GRD_BAS_FCT *grd_phi;
```

```
const D2_BAS_FCT  *D2_phi;
const D3_BAS_FCT  *D3_phi; /* Optional, implemented for Lagrange bfcts. */
const D4_BAS_FCT  *D4_phi; /* Optional, implemented for Lagrange bfcts. */

/* Vector valued basis functions are always factored as phi[i]() *
 * phi_d[i](). If phi_d[i]() is piece-wise constant, then
 * dir_pw_const should be true. The directions are never cached in
 * QUAD_FAST, only the scalar factor.
 */
const BAS_FCT_D     *phi_d;
const GRD_BAS_FCT_D *grd_phi_d;
const D2_BAS_FCT_D  *D2_phi_d;

bool dir_pw_const; /*Direction is p.w. constant on the reference element.*/

/************* the trace space on the wall ****************************/
const BAS_FCTS *trace_bas_fcts; /* The trace space */

/* The local DOF mapping for the trace spaces,
 * < 3d:
 * [0][0][wall][slave local dof] == master local dof,
 * 3d:
 * [type > 0][orient < 0][wall][slave local dof] == master local dof.
 */
const int       *trace_dof_map[2][2][N_WALLS_MAX];

/* This obscure component can vary from wall to wall in the presence
 * of an INIT_ELEMENT() method. It is _always_ equal to
 * trace_bas_fcts->n_bas_fcts ... BUT ONLY after the respective
 * element initializer has been called for trace_bas_fcts on the
 * trace mesh. If an INIT_ELEMENT() method is present then it _MUST_
 * initialize trace_dof_map _AND_ n_trace_bas_fcts. Of course, in 3D
 * only the components corresponding to type and orientation of the
 * current EL_INFO object have to be taken care of by the
 * INIT_ELEMENT() method.
 */
int             n_trace_bas_fcts[N_WALLS_MAX];

/*************** interconnection to DOF_ADMIN and mesh   ***************/
const EL_DOF_VEC *(*get_dof_indices)(DOF *result,
                                     const EL *, const DOF_ADMIN *,
                                     const BAS_FCTS *thisptr);
const EL_BNDRY_VEC *(*get_bound)(BNDRY_FLAGS *bndry_bits,
                                 const EL_INFO *eli,
                                 const BAS_FCTS *thisptr);

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

void (*interpol)(EL_REAL_VEC *coeff,
                 const EL_INFO *el_info, int wall,
                 int n, const int *indices,
                 LOC_FCT_AT_QP f, void *ud,
                 const BAS_FCTS *thisptr);
void (*interpol_d)(EL_REAL_D_VEC *coeff,
                   const EL_INFO *el_info, int wall,
                   int n, const int *indices,
                   LOC_FCT_D_AT_QP f, void *ud,
```

```
                        const BAS_FCTS *thisptr);
  void (*interpol_dow)(EL_REAL_VEC_D *coeff,
                       const EL_INFO *el_info, int wall,
                       int n, const int *indices,
                       LOC_FCT_D_AT_QP f, void *ud,
                       const BAS_FCTS *thisptr);

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

  const EL_INT_VEC     *(*get_int_vec)(int result[],
                                       const EL *, const DOF_INT_VEC *);
  const EL_REAL_VEC    *(*get_real_vec)(REAL result[],
                                        const EL *, const DOF_REAL_VEC *);
  const EL_REAL_D_VEC  *(*get_real_d_vec)(REAL_D result[],
                                          const EL *, const DOF_REAL_D_VEC *);
  const EL_REAL_VEC_D  *(*get_real_vec_d)(REAL result[],
                                          const EL *, const DOF_REAL_VEC_D *);
  const EL_UCHAR_VEC   *(*get_uchar_vec)(U_CHAR result[],
                                         const EL *, const DOF_UCHAR_VEC *);
  const EL_SCHAR_VEC   *(*get_schar_vec)(S_CHAR result[],
                                         const EL *, const DOF_SCHAR_VEC *);
  const EL_PTR_VEC     *(*get_ptr_vec)(void *result[],
                                       const EL *, const DOF_PTR_VEC *);

  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);

  void   (*real_refine_inter_d)(DOF_REAL_VEC_D *, RC_LIST_EL *, int);
  void   (*real_coarse_inter_d)(DOF_REAL_VEC_D *, RC_LIST_EL *, int);
  void   (*real_coarse_restr_d)(DOF_REAL_VEC_D *, RC_LIST_EL *, int);

  void   *ext_data; /* Implementation dependent extra data */
};

/* Barycentric coordinates of Lagrange nodes. */
#define LAGRANGE_NODES(bfcts)                                   \
  ((const REAL_B *)(*(void **)(bfcts)->ext_data))
```

The entries yield following information:

name string containing a textual description or `NULL`.

dim dimension d of the mesh triangulation.

rdim dimension of the range space. This is either 1 or `DIM_OF_WORLD` for vector valued basis functions like edge and face bubbles.

n_bas_fcts number of local basis functions.

n_bas_fcts_max maximum number of local basis functions. The number of basis functions on a given element may vary if the `init_element`-hook is non-`NULL`. In this case `n_bas_fcts_max` gives the upper limit and can be used to lay-out array dimensions, e.g. In the standard case this component does not differ from `n_bas_fcts`. See Section 3.7.

degree maximal polynomial degree of the basis functions; this entry is 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 4.2.1); additionally, degree is used by some graphics routines (see Section 4.11.1.1).

n_dof vector with the count of DOFs for this set of basis functions; n_dof[VERTEX,CENTER,EDGE,FACE] is the number of DOFs tied to the vertices, center, edges (only 2d and 3d), faces (only 3d), of an element; the corresponding DOF administration of the finite element space uses such information.

trace_admin is used for the purpose of defining basis functions with DOFs attached to a trace-mesh, e.g. to define face-bubbles attached to part of the boundary of the mesh. In this case trace_admin is the unique ID of a trace-mesh which carries the DOF_ADMIN for this basis-function set. In this situation the basis functions "live" on the bulk mesh (i.e. extend in to to bulk-phase of the element containing the face belonging to the trace-mesh), but the degrees of freedom are maintained on the trace mesh.

chain contains the link to the other parts if this instance forms part of a chain of basis functions. This is implemented as doubly linked list. In the standard case the list-node just points back to itself. Compare Section 3.5.3 and Section 3.7.

unchained points to a copy of the basis function structure which is unaware of being part of a direct sum. In the standard case unchained just points back to the same basis function structure it is part of.

INIT_ELEMENT_DECL is used for the initialization of element dependent local finite element spaces; this is needed, e.g. to define basis functions which depend on the element geometry like discretizations of the $H(\mathrm{div})$, or for some more complicated discretizations for the Stokes-problem which, e.g., make use of edge- and face-bubbles. See Section 3.11.

phi vector of function pointers for the evaluation of local basis functions in barycentric coordinates;

```
(*bfcts->phi[i])(lambda, bfcts)
```

returns the value $\bar{\varphi}^{\mathtt{i}}(\lambda)$ of the i-th basis function at lambda for $0 \le \mathtt{i} < \mathtt{n\_bas\_fcts}$. We advocate the use of the PHI()-macro instead:

```
PHI(bfcts, i, lambda)
```

grd_phi vector of function pointers for the evaluation of gradients of the basis functions in barycentric coordinates;

```
(*bfcts->grd_phi[i])(lambda)
```

returns a pointer to a vector of length N_LAMBDA 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 $0 \le \mathtt{k} \le d$, $0 \le \mathtt{i} < \mathtt{n\_bas\_fcts}$; this vector is overwritten on the next call of (*grd_phi[i])(). We advocate the use of the GRD_PHI()-macro instead:

```
GRD_PHI(bfcts, i, lambda)
```

D2_phi vector of function pointers for the evaluation of second derivatives of the basis functions in barycentric coordinates;

```
(*bfcts->D2_phi[i])(lambda, bfcts)
```

returns a pointer to a N_LAMBDA × N_LAMBDA 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]= $\bar{\varphi}^{\mathtt{i}}_{,\lambda_{\mathtt{k}}\lambda_{\mathtt{l}}}(\lambda)$ $0 \le \mathtt{k}, \mathtt{l} \le d$, $0 \le \mathtt{i} <$ n_bas_fcts; this matrix is overwritten on the next call of (*D2_phi[i])(). We advocate the use of the D2_PHI()-macro instead:

```
D2_PHI(bfcts, i, lambda)
```

D3_PHI(), D4_PHI() These do similar things as the other hooks, however, they need not be present in a specific BAS_FCTS implementation.

PHI_D() the directional part of the basis functions if rdim == DIM_OF_WORLD. ALBERTA always factors vector-valued basis functions into a scalar factor times a directional part, so the actual value of the i-th basis functions has to be calculated as

```
REAL_D value;
AXEY_DOW(PHI(bfcts, i, lambda), PHI_D(bfcts, i, lambda), value);
```

Expanding all the macros and inline functions, the above is equivalent to

```
REAL_D value;
const REAL *vector = bfcts->phi_d[i](lambda, bfcts);
REAL scalar = bfctgs->phi[i](lambda, bfcts);
int k;

for (k = 0; k < DIM_OF_WORLD; k++) {
  value][k] = scalar * vector[k];
}
```

Note that ALBERTA never caches the directional part of vector-valued basis functions in its QUAD_FAST or other quadrature caches; it is assumed that it changes on an element-to-element basis. Vector-valued basis functions and the associated support functions are discussed in further detail below in Section 3.5.2.

GRD_PHI_D() The gradient of the directional part of a vector-valued basis-function instance. Note that GRD_PHI_D may be empty if dir_pw_const == true.

D2_PHI_D() The second derivative of the directional part of a vector-valued basis function instance. Note that D2_PHI_D may be empty if dir_pw_const == true.

---

dir_pw_const if this is set to true then the directional part of the vector valued BAS_FCTS-instance (i.e. rdim == DIM_OF_WORLD) is constant on each mesh element (e.g. the normal to a face on affine linear elements). If this is the case then the computation of the derivatives of the basis functions is drastically simplified. See below in Section 3.5.2.

trace_bas_fcts A pointer to a basis function structure describing the trace of the current basis-function set on the boundaries of the reference element. In the case of Lagrange elements, this is again a Lagrange space of the same degree, but one dimension lower. The trace-spaces form a chain, which finally is terminated by a dimensions-0 "dummy" basis function set.

The trace space may, of course, be of other nature than the bulk basis function set. An element bubble, for instance, already has zero trace on the boundary. The trace-space of face-bubbles will be a DIM_OF_WORLD-valued element bubble, pointing in direction of the normal on the lower-dimensional element and so on.

The trace spaces play a role when boundary integrals involving basis functions are computed (see, for instance, BNDRY_OPERATOR_INFO structure – Section 4.7.3 – or bndry_L2_scp_fct_bas()). They are also used to define global traces of finite element function in the context of trace meshes, see Section 3.9 and Section 1.6 on page 40. Example 3.5.9 shows the use of the trace-space in the context of an interpolation routine for linear basis functions.

trace_dof_map[][][][] The mapping of the local degrees of freedom from the trace-set to the bulk-set of local basis functions, for each co-dimension 1 face-simplex ("wall"). To be more concrete:

- dim < 3:

```
trace_dof_map[0][0][wall][trace_dof] == bulk_dof
```

- dim $= 3$:

```
trace_dof_map[type > 0][orient < 0][wall][trace_dof] == bulk_dof
```

In this context, type and orient denote the respective components of the EL_INFO structure, compare also the conceptual discussion of trace-meshes in Section 1.6 on page 40.

n_trace_bas_fcts[] The dimension of the local trace-space, for each co-dimension 1 face-simplex. In the standard case, the trace-space has the same dimension on each wall, but in the context of per-element initializers (see Section 3.11) the dimension may vary from wall to wall.

get_dof_indices(result, el, admin, self) 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);

Parameters

DOF *result Storage for the result; result must be the base-address of an array to DOFs with at least n_bas_fcts elements, or NULL, in which case the result is returned in a statically allocated storage area which is overwritten on the next call to get_dof_indices(). On return result[i] stores the global DOF associated to the i–th basis function.

const EL *el the current mesh-element;

const DOF_ADMIN *admin the DOF-admin for the corresponding finite element space;

const BAS_FCTS *self a pointer to the current basis function instance; might be used if the set of local basis functions depends on the mesh element.

Return Value A pointer to a const EL_DOF_VEC element vector (see page 253) if (result == NULL) or NULL if (result != NULL). The vec component of the EL_DOF_VEC contains the data which otherwise would have been stored in result. The contents of the return value is overwritten on the next call to get_dof_indices(). If the BAS_FCTGS-instance forms part of a chain of basis functions (see Section 3.5.3), then only the DOFs associated to this instance are computed. To get the DOFs for all parts of the direct sum the global function get_dof_indices() has to be called, see Sections 4.7.1.3 and 3.7.

To reduce the potential of coding errors we advocate the use of the `GET_DOF_INDICES()` macro:

```
GET_DOF_INDICES(bfcts, result, el, admin)
```

instead of calling

```
bfcts->get_dof_indices(result, el, admin, bfcts)
```

directly.

`get_bound(bndry_bits, el_info, self)` pointer to a function which fills a vector with the boundary types of the basis functions.

3.5.1 Compatibility Note. *In contrast to all previous versions of ALBERTA the boundary-type of a given basis function is a bit-mask, and not a mere number. Each bit corresponds to the number that has been assigned to a given boundary segment in the macro-triangulation. Basis-functions tied to vertex-*`DOF`*s, e.g., may belong to different boundary segments in which case the bit-mask may contain more than one bit set. This is further discussed in Section 3.2.4.*

Otherwise the calling conventions are similar to the conventions for `get_dof_indices()` (see above), there also exists a macro `GET_BOUND(self, bndry_bits, el_info)`. This function needs boundary information; thus, all routines using this function on the elements need the `FILL_BOUND` flag during mesh traversal.

Parameters

`BNDRY_BITS *bndry_bits` storage for the reuslt;

`const EL_INFO *el_info` the current elements's `EL_INFO` descriptor;

`const BAS_FCTS *self` a pointer to the current basis function instance.

Return Value a pointer to a statically allocated storage area of type `const EL_BNDRY_VEC`, see page 253. If the `BAS_FCTS`-instance forms part of a direct sum, then only the boundary bit-masks associated with this instance are computed. To get the information for all parts of the direct sum the global function `get_bound()` has to be called, see Sections 4.7.1.3 and 3.7.

`interpol[_d|_dow](coeff, el_info, wall, n, indices, f, ud, thisptr)`
When using ALBERTA 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 4.7.7.1):

The `interpol`-hooks are function-pointers to functionws which performs the local interpolation of a `REAL[_D]` valued function on an element. If this instance of a local basis function set forms part of a chain of basis functions (see Section 3.5.3, then it is possible to call the functions `el_interpol()` – respectively their `..._d` and `..._dow` variants – to interpolate `f` onto the local functions space defined by the entire chain, see Section 4.7.1.3. The `BAS_FCTS.interpol`-hook will only perform the interpolation for a single member of such a chain.

Parameters

EL_REAL_VEC *coeff Mandatory, storage for the result. The vector valued version need an EL_REAL_D_VEC respetively an EL_REAL_VEC_D.

const EL_INFO *el_info Element descriptor.

wall If the interpolation is to be performed over the boundary of the mesh, then this is the number of the wall in EL_INFO to integrate over, in this case only the coefficients for the basis functions with non-zero trace on the respective wall will be computed.

n number of items in indices. If the interpolation is to be performed for all local DOFs, then -1 should be passed for n.

indices for selective interpolation of only some of the local DOFs, indices may contain as many entries as indicated by n. The components of indices are then the local DOF number for which the coefficients should be computed. If indices == NULL, then the coefficients for all local basis functions will be computed.

REAL (*f)(const EL_INFO *el_info, const QUAD *quad, int iq, void *ud) The application provided function to interpolate onto the local function space defined by this local basis function set. For simple Lagrange elements QUAD will just be a "lumping" quadrature rule, with "quadrature" nodes on the Lagrange nodes of the basis function set. But interpolation may in fact require larger efforts, in which case QUAD may be a "real" quadrature rule. If the interpolation is to be taken over a boundary segment, then QUAD will be a co-dimension 1 quadrature rule, see also Section 4.2.1.

For interpolation of DIM_OF_WORLD-valued basis functions f must be a function pointer of the format

```
const REAL *(*f)(REAL_D result, const EL_INFO *el_info,
    const QUAD *quad, int iq, void *ud);
```

ud Application data-pointer, forwarded to f as last argument.

thisptr A pointer to the local basis function set. To reduce the potential of coding errors we advocate the use of the INTERPOL() macro:

```
INTERPOL(bfcts, coeff, el_info, wall, n, indices, f, ud)
```

instead of calling

```
bfcts->interpol(coeff, el_info, wall, n, indices, f, ud, bfcts)
```

directly. Likewise for INTERPOL_D() and INTERPOL_DOW.

Return Value void.

```
const EL_INT_VEC *
(*get_int_vec)(int res[], const EL *el, const DOF_INT_VEC *dv)
const EL_REAL_VEC *
(*get_real_vec)(REAL res[], const EL *el, const DOF_REAL_VEC *dv)
const EL_REAL_D_VEC *
(*get_real_d_vec)(REAL_D res[], const EL *el, const DOF_REAL_D_VEC *dv)
const EL_REAL_VEC_D *
```

(*get_real_vec_d)(REAL res[], const EL *el, const DOF_REAL_VEC_D *dv)
const EL_UCHAR_VEC *
(*get_uchar_vec)(U_CHAR res[], const EL *el, const DOF_UCHAR_VEC *dv)
const EL_SCHAR_VEC *
(*get_schar_vec)(S_CHAR res[], const EL *el, const DOF_SCHAR_VEC *dv)
const EL_PTR_VEC *
(*get_ptr_vec)(void *res[], const EL *el, const DOF_PTR_VEC *dv) These are pointers to functions which fills a local per-element coefficient vector with values of a `DOF_*_VEC` at the DOFs of the basis functions. The calling convention is much the same as for the `get_dof_indices()`- and `get_bound()`-hooks. Also, in the context of chains of basis functions (see Section 3.5.3 below) it should be noted that these function hooks only work on a single component of that chain. However, each of the hooks has global function as counter-part which does the job for the entire chain, see Section 4.7.1.3.

3.5.2 Compatibility Note. *The calling convention has changed with respect to previous versions of ALBERTA. In particular, there are now dedicated structures for storing local per-element coefficient vectors.*

Note that the `get_real_vec_d`-hook accepts a *scalar* `res`-argument of type `REAL` and returns a `EL_REAL_VEC_D` because in the context of vector-valued basis functions the coefficient vector are scalars, see bewlow Section 3.5.2.
A detailed description of the parameters is only given for the `get_real_vec()`-hook. The others work similar.

Parameters

res An optional argument to store the coefficients in. `res` maybe `NULL`, in which case the return value if a pointer to `dv->vec_loc`. If `res` is non-`NULL`, then the return value of this function is `NULL`.

3.5.3 Compatibility Note. *This implies that it is safe to call* `get_real_vec(NULL, ...)` *repeatedly with* different `DOF_REAL_VEC` *instances, since the storage area for the return value is now tied to the argument* `dv`*, reducing the potential for coding errors.*

On the other hand, previous versions were returning a pointer to the argument `res`*, if that was non-NULL. Of course, that can no longer work, because the return value is now a fully-fledged element vector, while the* `res`*-argument is just a flat* `C`*-array.*

el The element to compute the local coefficients for.

dv The global coefficient vector to fetch the data-values from.

Return Value `NULL` if (`res != NULL`), and `dv->vec_loc` otherwise.

---

void (*real_refine_inter)(DOF_REAL_VEC *, RC_LIST_EL *rcl, int n)
void (*real_coarse_inter)(DOF_REAL_VEC *, RC_LIST_EL *rcl, int n)
void (*real_coarse_restr)(DOF_REAL_VEC *, RC_LIST_EL *rcl, int n)
void (*real_d_refine_inter)(DOF_REAL_D_VEC *, RC_LIST_EL *rcl, int n)
void (*real_d_coarse_inter)(DOF_REAL_D_VEC *, RC_LIST_EL *rcl, int n)

```
void (*real_d_coarse_restr)(DOF_REAL_D_VEC *, RC_LIST_EL *rcl, int n)
void (*real_refine_inter_d)(DOF_REAL_VEC_D *, RC_LIST_EL *rcl, int n)
void (*real_coarse_inter_d)(DOF_REAL_VEC_D *, RC_LIST_EL *rcl, int n)
void (*real_coarse_restr_d)(DOF_REAL_VEC_D *, RC_LIST_EL *rcl, int n)
```

Since the interpolation of finite element functions during refinement and coarsening, as well as 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 above function pointers. Not all basis-function implementations may come with a full set of interpolation respectively restriction routines.

Note also that these function-pointers are not used automatically; it is the responsibility of the application program to hook them into the global `DOF_REAL_[_D]_VEC[_D]` *coefficient vectors, only then ALBERTA will make use of these function during mesh adaptation.*

To give some aid in performing this job in the context of the more-complicated direct-sums of finite element spaces, there are global functions `set_refine_inter[_d|_dow]()`, `set_coarse_inter[_d|_dow]()`, `set_coarse_resrt[_d|_dow]()` which do this job for an entire chain of coefficient vectors (i.e. hook the corresponding function from the relevant `BAS_FCTS`-component into the hook in the hook of the `DOF`-vector instance). See also Section 3.7.

In Section 3.5.4.1 and 3.5.4.2 examples for the implementation of those functions are given.

Functionally, the three different flavours have the following meaning:

`real[_d]_refine_inter[_d]` pointer to a function for interpolating a `REAL[_D]` valued function during refinement; i.e. for interpolating the `DOF_REAL[_D]_VEC[_D]` vector `vec` on the refinement patch `rcl` onto the finer grid; information about all parents of the refinement patch is accessible in the vector `rcl` of length `n`.

`real[_d]_coarse_inter[_d]` pointer to a function for interpolating a `REAL[_D]` valued function during coarsening; i.e. for interpolating the `DOF_REAL[_D]_VEC` vector `vec` on the coarsening patch `rcl` onto the coarser grid; information about all parents of the refinement patch is accessible in the vector `rcl` of length `n`.

`real[_d]_coarse_restr[_d]` pointer to a function for restriction of `REAL[_D]` valued linear functionals during coarsening; i.e. for restricting the `DOF_REAL[_D]_VEC` vector `vec` on the coarsening patch `rcl` onto the coarser grid; information about all parents of the refinement patch is accessible in the vector `rcl` of length `n`.

`ext_data` A `void`-pointer to implementation dependent data tied to a specific basis-functions implementation. For standard Lagrange basis-functions, this pointer gives access to the local Lagrange nodes ("local" meaning their barycentric coordinates) via the macro

```
const REAL_B *nodes = LAGRANGE_NODES(bfcts);
```

Although this is defined as a macro in `alberta.h`, an application program must not assume that this macro will not change in future versions of the tool-box. The ordering of data behind the `ext_data` pointer is opaque, nothing should be assumed about it.

3.5.4 Remark. The access to 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 `NULL`, ALBERTA 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:

```
const EL_INT_VEC *
default_get_int_vec(int *vec, const EL *el, const DOF_INT_VEC *dof_vec)
{
  FUNCNAME("get_int_vec");
  int *rvec = vec == NULL ? dof_vec->vec_loc->vec : vec;
  const BAS_FCTS *bas_fcts = dof_vec->fe_space->bas_fcts;
  int n_bas_fcts = bas_fcts->n_bas_fcts;
  DOF index[n_bas_fcts];
  int  i;

  GET_DOF_INDICES(dof_vec->fe_space->bas_fcts,
                  el, dof_vec->fe_space->admin, index);

  for (i = 0; i < n_bas_fcts; i++) {
    rvec[i] = dof_vec->vec[index[i]];
  }

  return vec ? NULL : dof_vec->vec_loc;
}
```

A specialized implementation for linear finite elements e.g. is more efficient:

```
const EL_INT_VEC *
get_int_vec(int *ivec, const EL *el, const DOF_INT_VEC *vec)
{
  FUNCNAME("get_int_vec");
  int             i, n0;
  int             *v = vec->vec;
  int             *rvec = ivec ? ivec : vec->vec_loc->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 vec ? rvec : vec->vec_loc;
}
```

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. All Lagrange finite elements described below are already registered, with names `"lagrange1_1d"` to `"lagrange4_3d"`. The discontinuous polynomial finite elements are registered with `"disc_lagrange0_1d"` to `"disc_lagrange2_3d"`. 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 the 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 `NULL`.

Lagrange elements can be accessed by a call of `get_lagrange()`, see Section 3.5.4.5, discontinuous polynomial elements by `get_discontinuous_lagrange()`, see Section 3.5.5.

3.5.2 Vector-valued basis functions

As a new feature, the current version of this finite element toolbox contains support for vector-valued basis functions like edge- or face-bubbles, or Raviart-Thomas elements. The actual implementation of those basis functions has been moved to an add-on module `libalbas`, see Section 3.5.7 below. An example for the implementation of face-bubbles can be found in

```
albertadist/add_ons/lib_albas/src/wall_bubbles.c
```

The current implementation assumes that it is efficient to factor vector-valued basis functions into a scalar part which does *not* depend on the element geometry and a vector-valued part – actually: `DIM_OF_WORLD`-valued – which depends on the element geometry. This is reflected by the `BAS_FTCS` data-structure: the `BAS_FCTS.phi` jump-tables correspond to the scalar factor, while the vector-valued part is stored in the jump-table `BAS_FCTS.phi_d`, and analogously for the jump-tables for the derivatives.

Often vector-valued basis functions will carry a per-element initializer, a function pointer `BAS_FCTS.init_element(el_info, self)`, which is invoked with the current `EL_INFO`-descriptor and the basis-function instance itself. This function-hook can be used to update geometry information like coordinates or wall-normals. In this context, the component `BAS_FCTS.fill_flags` is also of particular importance, it contains the collection of mesh-traversal flags (see Section 3.2.17) which are needed in order for the `init_element()`-hook to do its job properly. See also Section 3.11.

If the evaluation of basis functions is computationally costly, then it is of special importance to cache values of basis functions (and their derivatives) at quadrature points (see Section 4.2.2). For vector-valued basis-functions, these caches are only maintained for the scalar factor, as the vector-valued factor is assumed to vary from element to element anyway. In order to simplify the "recombination" of the scalar- and the vector-factor, the library provides three functions to perform this task (arguably this part of the documentation would belong to Section 4.2.2):

```
const REAL_D *const*get_quad_fast_phi_dow(const QUAD_FAST *cache);
const REAL_DB *const*get_quad_fast_grd_phi_dow(const QUAD_FAST *cache);
const REAL_DBB *const*get_quad_fast_D2_phi_dow(const QUAD_FAST *cache);
```

These three function take a pointer to a properly initialized `QUAD_FAST`-structure as returned by `get_quad_fast()` and return arrays containing the values of the products of the scalar- and vector-part of the basis-functions. This way an application can call those functions, and then use the returned arrays in the same way it used the components of the `QUAD_FAST`-structure. The ordering of indices is also the same, e.g.

3.5.5 Example.

```
const REAL_D *const*phi_d = get_quad_fast_phi_dow(qfast);

for (iq = 0; iq < qfast->n_points; iq++) {
  for (b = 0; b < qfast->n_bas_fcts; b++) {
    do_something_fct(phi_d[iq][b]);
  }
}
```

It is also worth noting that the coefficient-vectors for finite-element functions based on vector-valued basis-functions contain scalars (the vector-nature of the finite element space is induced by the vector-nature of the values of the basis functions, thus the coefficients for the basis functions are scalars). ALBERTA's assemble infra-structure (see Section 4.7) has full support for assembling linear systems based on vector-valued basis functions, and to freely pair scalar- and `DIM_OF_WORLD`-valued finite element spaces. Actually, the support-functions for the assembling of the discrete systems use up most of the compilation time during the installation of the ALBERTA-package.

3.5.3 Chains of basis function sets

The current version of this finite element toolbox has support for direct sums of finite-element spaces. Each component of such a sum is defined by a set of local basis functions which is part of a chain of basis functions. The chain-connectivity is implemented as a doubly linked cyclic list, the corresponding list-link can be found in the `BAS_FCTS.chain` component. In order to chain single basis function implementations together the support function `chain_bas_fcts()` has to be called:

```
BAS_FCTS *chain_bas_fcts(const BAS_FCTS *head, BAS_FCTS *tail);
```

`chain_bas_fcts(head, tail)` Clone the set of basis-functions specified as `head`, if `tail != NULL`, then add the copy of `head` as head to the list specified by `tail`. The original instance of `head` is hooked into the copy using the `BAS_FCTS.unchained` pointer; it remains a single, un-chained `BAS_FCTS`-instance. The chain-connectivity is implemented using the doubly-linked list-node `BAS_FCTS.chain`.

The chained basis functions will be given the name

`"HEAD_NAME#TAIL_NAME"`

During the construction of the name any `"_Xd"`-suffixes are discarded in order not to make the name too complicated. At the end of the name-chain an appropriate `_Xd`-suffix is added. The `_Xd`-suffixes are used only for debugging, they are ignored everywhere else.

A basis-function chain is cyclic. The trace-spaces of the given sets of basis functions are chained together accordingly. If any part of the chain needs a per-element initialization (see Section 3.11), then `chain_bas_fcts()` assigns the resulting chain a special `init_element()` hook which follows the convention described in Section 3.11 and which calls the per-element initializers of each member of the chain in turn.

Note: this function does *not* call `new_bas_fcts()`; the caller has to do so after constructing the desired chain.

More about direct sums of finite-element spaces should be found in Section 3.7. It is generally a bad idea to chain sets of basis functions together which are not linearly independent from each other ....

3.5.6 Example. Forming the velocity-part of the "Mini"-element, and looping over the generated two-element chain, printing the name of the unchained instances. Note that the `CHAIN_DO()`-macro rolls over the entire list, which is cyclic. So inside the loop `bas_fcts` points to different components of the chain, after the loop `bas_fcts` again points to the first instance of the chain.

This is a certain potential for bugs when jumping out of the loop, using, e.g., a `break` *statement. This should be avoided.* `continue` *statements should also be avoided because the trailing* `CHAIN_WHILE()`*-part increments the list-pointer.*

```
lagrange = get_lagrange(dim, degree);
bubble   = get_bas_fcts(dim, "Bubble");
bas_fcts = chain_bas_fcts(lagrange, chain_bas_fcts(bubble, NULL));
old_fcts = new_bas_fcts(bas_fcts);

if (old_fcts != NULL) {
  MSG("Overriding old definition for \"%s\"\n", old_fcts->name);
}

MSG("New name: \"%s\"\n", bas_fcts->name);
CHAIN_DO(bas_fcts, BAS_FCTS) {
  MSG("Rolling component name: \"%s\"\n", bas_fcts->name);
  MSG("Component name: \"%s\"\n", bas_fcts->unchained->name);
} CHAIN_WHILE(bas_fcts, BAS_FCTS);
MSG("Again, the name of the entire chain: \"%s\"\n", bas_fcts->name);
```

3.5.4 Lagrange finite elements

ALBERTA provides 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

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

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, center, edges, or faces of an element. The corresponding DOF is a vertex, center, edge, or face 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.

3.5.4.1 Piecewise linear finite elements

Piecewise linear, continuous finite elements are uniquely defined by their values at the vertices of the triangulation. On each element we have `N_VERTICES(dim)` basis functions which are the barycentric coordinates of the element. Thus, in 1d we have two, 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 Tables 3.4, 3.5 and 3.6. The calculation of derivatives is straight forward. The global DOF index of the `i`–th

function	position	Lagrange node
$\bar{\varphi}^0(\lambda) = \lambda_0$	vertex 0	$\lambda^0 = (1,0)$
$\bar{\varphi}^1(\lambda) = \lambda_1$	vertex 1	$\lambda^1 = (0,1)$

Table 3.4: Local basis functions for linear finite elements in 1d.

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 3.5: Local basis functions for linear finite elements in 2d.

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 3.6: Local basis functions for linear finite elements in 3d.

basis functions on element `el` is stored for linear finite elements at

```
el->dof[i][admin->n0_dof[VERTEX]]
```

Setting `nv = admin->n0_dof[VERTEX]` the associated DOFs are shown in Figure 3.5.

For linear finite elements we want to give examples for the implementation of some routines in the corresponding `BAS_FCTS` structure.

3.5.7 Example (Accessing DOFs for piecewise linear finite elements)**.** The implementation of `get_dof_indices()` can be done in as in the following 2d example code, compare Figure 3.5 and Remark 3.5.4 with the implementation of the function `get_int_vec()` for accessing a local element vector from a global `DOF_INT_VEC` for piecewise linear finite elements.

```
static const EL_DOFVEC *
get_dof_indices1_2d(DOF *vec, const EL *el, const DOF_ADMIN *admin,
                    const BAS_FCTS *thisptr)
{
  static DEF_EL_VEC_CONST(DOF, rvec_space, N_BAS_LAG_1_2d, N_BAS_LAG_1_2d);
  DOF *rvec = vec ? vec : rvec_space->vec;
  int n0, /* node, */ ibas;
  DOF **dofptr = el->dof, dof;

  /* node = admin->mesh->node[VERTEX]; */
  n0   = admin->n0_dof[VERTEX];
  for (ibas = 0; ibas < N_BAS_LAG_1_2D; ibas++) {
    dof = dofptr[/* node+ */ibas][n0];
    body;
  }

  return vec ? NULL : rvec_space;
}
```

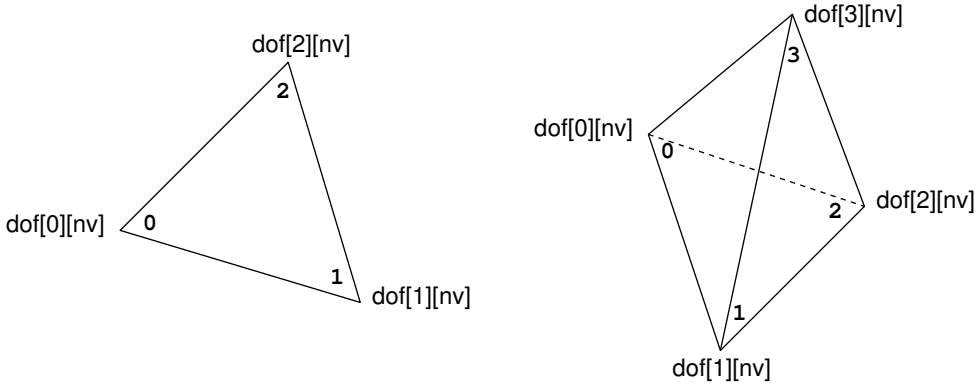


Figure 3.5: DOFs and local numbering of the basis functions for linear elements in 2d and 3d.

3.5.8 Example (Accessing the boundary types of DOFs for piecewise linear finite elements in 2d)**.** The `get_bound()` function fills the `bound` vector with the boundary type of the vertices, shown here for 2d:

```
static const EL_BNDRY_VEC *
get_bound1_2d(BNDRY_FLAGS *vec, const EL_INFO *el_info, const BAS_FCTS
    *thisptr)
{
  FUNCNAME("get_bound1_2d");
```

```
  static DEF_EL_VEC_CONST(BNDRY, rvec_space, N_BAS_LAG_1_2d,
      N_BAS_LAG_1_2d););
  BNDRY_FLAGS *rvec = vec ? vec : rvec_space->vec;
  int          i;

  DEBUG_TEST_FLAG(FILL_BOUND, el_info);

  for (i = 0; i < N_VERTICES_2D; i++) {
    BNDRY_FLAGS_CPY(rvec[i], el_info->vertex_bound[i]);
  }

  return vec ? NULL : rvec_space;
}
```

3.5.9 Example (Interpolation for piecewise linear finite elements in 2d)**.** For the interpolation `interpol()` routine we have to evaluate the given function at the vertices. Assuming a 2d mesh, the interpolation can be implemented as follows. Note the use of a "lumping" quadrature rule; the application supplied function `f()` has the task to return its value at the quadrature point `iq`.

```
static void interpol1_2d(EL_REAL_VEC *el_vec,
                         const EL_INFO *el_info, int wall,
                         int no, const int *b_no,
                         REAL (*f)(const EL_INFO *el_info,
                                   const QUAD *quad, int iq,
                                   void *ud),
                         void *f_data,
                         const BAS_FCTS *thisptr)
{
  FUNCNAME("interpol1_2d");
  LAGRANGE_DATA *ld = &lag_1_2d_data;
  REAL *rvec = el_vec->vec;
  const QUAD *lq;
  const int *trace_map;
  int i;

  DEBUG_TEST_EXIT(ld->lumping_quad != NULL,
                  "called for uninitialized Lagrange basis functions\n");

  if (wall < 0) {
    lq = ld->lumping_quad;
    trace_map = NULL;
  } else {
    int type = el_info->el_type > 0;
    int orient = el_info->orientation < 0;
    lq = &ld->trace_lumping_quad[type][orient][wall];
    trace_map = thisptr->trace_dof_map[type][orient][wall];
  }

  DEBUG_TEST_EXIT(!b_no || (no >= 0 && no <= lq->n_points),
                  "not for %d points\n", no);

  el_vec->n_components = thisptr->n_bas_fcts;

  if (b_no) {
    for (i = 0; i < no; i++) {
```

```
      int ib = wall < 0 ? b_no[i] : trace_map[b_no[i]];
      rvec[ib] = f(el_info, lq, b_no[i], f_data);
    }
  } else {
    for (i = 0; i < lq->n_points; i++) {
      int ib = wall < 0 ? i : trace_map[i];
      rvec[ib] = f(el_info, lq, i, f_data);
    }
  }
}
```

3.5.10 Example (Interpolation and restriction routines for piecewise linear finite elements in 2d)**.** 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.

```
static void real_refine_inter1_2d(DOF_REAL_VEC *drv, RC_LIST_EL *list, int n)
{
  FUNCNAME("real_refine_inter1_2d");
  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_info.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[2][n0];  /*     newest vertex is dim==2 */
  vec[dof_new] = 0.5*(vec[dof0] + vec[dof1]);

  return;
}



static void real_coarse_restr1_2d(DOF_REAL_VEC *drv, RC_LIST_EL *list, int n)
{
  FUNCNAME("real_coarse_restr1_2d");
  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_info.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[2][n0];      /*   newest vertex is dim==2 */
    vec[dof0] += 0.5*vec[dof_new];
    vec[dof1] += 0.5*vec[dof_new];

    return;
}
```

3.5.4.2 Piecewise quadratic finite elements

Piecewise quadratic, continuous finite elements are uniquely defined by their values at the vertices and the edges' midpoints (center in 1d) of the triangulation. In 1d we have three, 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 Tables 3.7, 3.8, and 3.9.

function	position	Lagrange node
$\bar{\varphi}^0(\lambda) = \lambda_0(2\lambda_0 - 1)$	vertex 0	$\lambda^0 = (1,0)$
$\bar{\varphi}^1(\lambda) = \lambda_1(2\lambda_1 - 1)$	vertex 1	$\lambda^1 = (0,1)$
$\bar{\varphi}^2(\lambda) = 4\lambda_0\,\lambda_1$	center	$\lambda^2 = (\frac{1}{2},\frac{1}{2})$

Table 3.7: Local basis functions for quadratic finite elements in 1d.

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 3.8: 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 3.9: Local basis functions for quadratic finite elements in 3d.

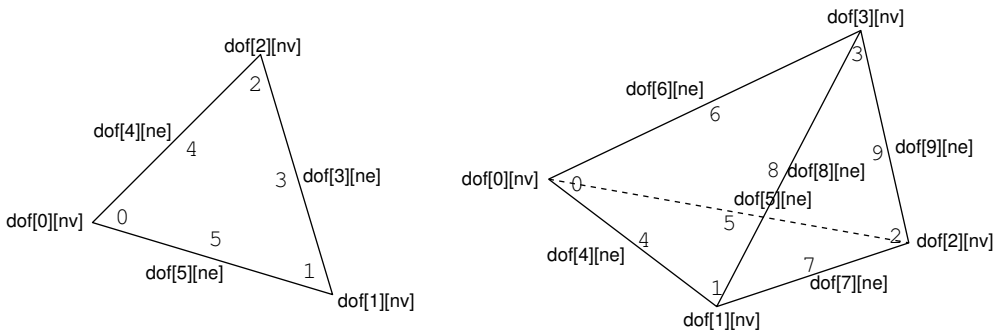


Figure 3.6: DOFs and local numbering of the basis functions for quadratic elements in 2d and 3d.

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 3.6.

3.5.11 Example (Accessing DOFs for piecewise quadratic finite elements)**.** The function `get_dof_indices()` for quadratic finite elements can be implemented in 2d by (compare Figure 3.6):

```
static const EL_DOF_VEC *
get_dof_indices2_2d(DOF *vec, const EL *el, const DOF_ADMIN *admin,
                    const BAS_FCTS *thisptr)
{
  static DEF_EL_VEC_CONST(DOF, rvec_space, N_BAS_LAG_2_2D, N_BAS_LAG_2_2D);
  DOF_VEC_DOF *rvec = vec ? vec : rvec_space->vec;
  int n0, ibas, inode;
  DOF **dofptr = el->dof, dof;

  n0 = (admin)->n0_dof[VERTEX];
  for (ibas = inode = 0; ibas < N_VERTICES_2D; inode++, ibas++) {
    dof = dofptr[inode][n0];
    body;
  }
  n0 = (admin)->n0_dof[EDGE];
  for (inode = 0; inode < N_EDGES_2D; inode++, ibas++) {
    dof = dofptr[N_VERTICES_2D+inode][n0];
```

```
    body;
  }

  return vec ? NULL : rvec_space;
}
```

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.

3.5.12 Example (Interpolation during refinement for piecewise quadratic finite elements 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:

```
static void real_refine_inter2_3d(DOF_REAL_VEC *drv, RC_LIST_EL *list, int n)
{
  FUNCNAME("real_refine_inter2_3d");
  DOF pdof[N_BAS_LAG_2_3D];
  DOF cdof[N_BAS_LAG_2_3D];
  EL         *el;
  REAL       *v = NULL;
  DOF        cdofi;
  int        i, lr_set;
  int        node0, n0;
  const DOF_ADMIN *admin;
  const BAS_FCTS *bas_fcts;

  if (n < 1) return;
  el = list->el_info.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_STRUCT(admin, drv->fe_space);
  GET_STRUCT(bas_fcts, drv->fe_space);

  get_dof_indices2_3d(pdof, el, admin, bas_fcts);
```

```
  node0 = drv->fe_space->mesh->node[EDGE];
  n0 = admin->n0_dof[EDGE];

/*--------------------------------------------------------------------------*/
/*  values on child[0]                                                      */
/*--------------------------------------------------------------------------*/

  get_dof_indices2_3d(cdof, el->child[0], admin, bas_fcts);

  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_info.el;
    get_dof_indices2_3d(pdof, el, admin, bas_fcts);

    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;

    DEBUG_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;
}
```

3.5.13 Example (Interpolation during refinement for piecewise quadratic finite elements in 3d)**.** Here, 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.

```
static void real_refine_inter2_3d(DOF_REAL_VEC *drv, RC_LIST_EL *list, int n)
{
  FUNCNAME("real_refine_inter2_3d");
  EL          *el;
  REAL        *v = nil;
  const DOF *cdof;
  DOF         pdof[N_BAS2_3D], 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_info.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;
  GET_STRUCT(admin, drv->fe_space);

  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_info.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;

    DEBUG_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;
}
```

3.5.4.3 Piecewise cubic finite elements

For Lagrange elements of third order we have four basis functions in 1d, ten basis functions in 2d, and 20 in 3d; the basis functions and the corresponding Lagrange nodes in barycentric coordinates are shown in Tables 3.10, 3.11, and 3.12.

function	position	Lagrange node
$\bar{\varphi}^0(\lambda) = \frac{1}{2}(3\lambda_0 - 1)(3\lambda_0 - 2)\lambda_0$	vertex 0	$\lambda^0 = (1, 0)$
$\bar{\varphi}^1(\lambda) = \frac{1}{2}(3\lambda_1 - 1)(3\lambda_1 - 2)\lambda_1$	vertex 1	$\lambda^1 = (0, 1)$
$\bar{\varphi}^2(\lambda) = \frac{9}{2}(3\lambda_0 - 1)\lambda_0\lambda_1$	center	$\lambda^2 = (\frac{2}{3}, \frac{1}{3})$
$\bar{\varphi}^3(\lambda) = \frac{9}{2}(3\lambda_1 - 1)\lambda_1\lambda_0$	center	$\lambda^3 = (\frac{1}{3}, \frac{2}{3})$

Table 3.10: Local basis functions for cubic finite elements in 1d.

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 3.11: Local basis functions for cubic finite elements in 2d.

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 numbering of vertices on the two triangles. In Figure 3.7 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 3.7 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]]
```

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 3.12: Local basis functions for cubic finite elements in 3d.

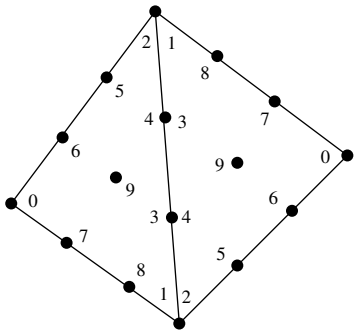


Figure 3.7: Cubic DOFs on a patch of two triangles.

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 to local DOFs from global ones.

3.5.14 Example (Accessing DOFs for piecewise cubic finite elements)**.** For the implementation, 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 one degree of freedom at each

face and none at the center. The DOFs at an edge are accessed according to the orientation of the edge. We present the implementation in 2d:

```
#define N_BAS_LAG_3_2D (N_VERTICES_2D+2*N_EDGES_2D+1)

static const EL_DOF_VEC *
get_dof_indices3_2d(DOF *vec, const EL *el, const DOF_ADMIN *admin,
                    const BAS_FCTS *thisptr)
{
  static DEF_EL_VEC_CONST(type, rvec_space, N_BAS_LAG_3_2D, N_BAS_LAG_3_2D);
  DOF *rvec = vec ? vec : rvec_space->vec;
  int n0, ibas, inode;
  DOF **dofptr = el->dof, dof;

  n0 = admin->n0_dof[VERTEX];
  for (ibas = 0; ibas < N_VERTICES_2D; ibas++) {
    dof = dofptr[ibas][n0];
    body;
  }
  n0 = admin->n0_dof[EDGE];
  for (inode = 0, ibas = N_VERTICES_2D; inode < N_EDGES_2D; inode++) {
    if (dofptr[vertex_of_edge_2d[inode][0]][0]
        < dofptr[vertex_of_edge_2d[inode][1]][0]) {
      dof = dofptr[N_VERTICES_2D+inode][n0];
      body;
      ibas++;
      dof = dofptr[N_VERTICES_2D+inode][n0+1];
      body;
      ibas++;
    } else {
      dof = dofptr[N_VERTICES_2D+inode][n0+1];
      body;
      ibas++;
      dof  = dofptr[N_VERTICES_2D+inode][n0];
      body;
      ibas++;
    }
  }
  n0 = admin->n0_dof[CENTER];
  dof = dofptr[6][n0];
  body;

  return vec ? NULL : rvec_space;
}
```

3.5.4.4 Piecewise quartic finite elements

For Lagrange elements of fourth order we have 5 basis functions in 1d, 15 in 2d, and 35 in 3d; the basis functions and the corresponding Lagrange nodes in barycentric coordinates are shown in Tables 3.13, 3.14, and 3.15.

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 can then 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

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	$\lambda^0 = (1,0)$
$\bar\varphi^1(\lambda) = \frac{1}{3}(4\lambda_1-1)(2\lambda_1-1)(4\lambda_1-3)\lambda_1$	vertex 1	$\lambda^1 = (0,1)$
$\bar\varphi^2(\lambda) = \frac{16}{3}(4\lambda_0-1)(2\lambda_0-1)\lambda_0\lambda_1$	center	$\lambda^2 = (\frac{3}{4},\frac{1}{4})$
$\bar\varphi^3(\lambda) = 4(4\lambda_0-1)(4\lambda_1-1)\lambda_0\lambda_1$	center	$\lambda^3 = (\frac{1}{2},\frac{1}{2})$
$\bar\varphi^4(\lambda) = \frac{16}{3}(4\lambda_1-1)(2\lambda_1-1)\lambda_0\lambda_1$	center	$\lambda^4 = (\frac{1}{4},\frac{3}{4})$

Table 3.13: Local basis functions for quartic finite elements in 1d.

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 3.14: Local basis functions for quartic finite elements in 2d.

DOFs on an element at that face. Such an orientation can again be defined by DOF indices at the face's vertices.

3.5.4.5 Access to Lagrange elements

The Lagrange elements described above are already implemented in ALBERTA; access to Lagrange elements is given by the function

```
const BAS_FCTS *get_lagrange(int, int);
```

Description:

`get_lagrange(dim, degree)` returns a pointer to a filled `BAS_FCTS` structure for Lagrange elements of order `degree`, where $1 \le$ `degree` ≤ 4, for dimension `dim`; no additional call of `new_bas_fcts()` is needed.

3.5.5 Discontinuous Lagrange finite elements

Similar to the standard Lagrange elements described above, discontinuous polynomial finite elements are piecewise polynomial functions. However, the functions are not globally continu-

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 3.15: Local basis functions for quartic finite elements in 3d.

ous. This is implemented in ALBERTA by assigning all DOFs to be of type `CENTER`, implying that they will not be shared among elements. At the moment these elements are available for polynomial degree 0 (piecewise constants), degree 1 (piecewise linears), and degree 2 (piecewise quadratics). Much of the implementation is similiar to the case of standard Lagrange

elements, hence we will not provide a detailed description.

Access to discontinuous elements is provided by the function

```
const BAS_FCTS  *get_discontinuous_lagrange(int dim, int degree);
```

Description:

get_discontinuous_lagrange(dim, degree) returns a pointer to a filled BAS_FCTS structure for discontinuous polynomial elements of order degree, where $0 \leq$ degree ≤ 2 for dimension dim; no additional call of new_bas_fcts() is needed.

3.5.6 Discontinuous orthogonal finite elements

In the context of discontinuous Galerkin methods it is often easier to use basis-functions which are orthogonal w.r.t. the L^2 scalar product, especially in the context of explicit time discretization schemes where the use of orthogonal basis functions eliminates the need for the inversion of the mass-matrix. ALBERTA implements discontinuous L^2-orthogonal basis functions of degree 1 and 2 in all supported mesh-dimensions.

Access to discontinuous elements is provided by the function

```
const BAS_FCTS *get_disc_ortho_poly(int dim, int degree);
```

Description:

get_disc_ortho_poly(dim, degree) returns a pointer to a filled BAS_FCTS structure for discontinuous polynomial elements of order degree, where $1 \leq$ degree ≤ 2 for dimension dim; no additional call of new_bas_fcts() is needed.

3.5.7 Basis-function plug-in module

ALBERTA also supports a rudimentary plug-in scheme, if get_bas_fcts(dim, name) cannot find an instance of basis functions requested by the parameter name, then it looks for the environment variable ALBERTA_BAS_FCTS_LIB_XD respectively ALBERTA_BAS_FCTS_LIB_XD_DEBUG, where the X has to be replaced by the value of DIM_OF_WORLD. The environment variable is supposed to contain the full path to a shared library containing additional basis-function implementations. The module must define (and export) a function in C-calling convention named const BAS_FCTS *bas_fcts_init(int dim, dim dow, const char *name), which is called to resolve the request of the application program. By pointing to a basis-function plug-in module via that environment variable it is, e.g., possible to post-process finite element data using "fancy" basis function implementations with the GRAPE or Paraview interface tools (see Section 4.11), without having to recompile and relink the converter-tools (like alberta2paraview_Xd).

Additionally, the function

```
typedef const BAS_FCTS *
(*BAS_FCTS_INIT_FCT)(int dim, int dow, const char *name);

void add_bas_fcts_plugin(BAS_FCTS_INIT_FCT init_fct);
```

can be used to supply additional plug-in functions defining even more basis-functions. During the installation of the ALBERTA package the basis-function add-on module in `add_ons/libalbas/` is compiled and installed under the name

```
PREFIX/lib/libalbas_Xd[_debug].DYNEXT,
```

where the `X` again has to be replaced by the value of `DIM_OF_WORLD` and `DYNEXT` stands for the architecture dependent extension attached to dynamic libraries.

`libalbas` currently implements wall-bubbles (i.e. face-bubbles and edge-bubbles), element bubbles and the corresponding trace-spaces. There is also a very rudimentary and untested version of the lowest order Raviart-Thomas element. Additionally, the function `stokes_pair()` implements some of the known stable mixed discretizations for the Stokes-problem.

```
typedef struct stokes_pair STOKES_PAIR;
struct stokes_pair
{
  const BAS_FCTS *velocity;
  const BAS_FCTS *pressure;
  /* const BAS_FCTS *slip_stress; */
};

STOKES_PAIR stokes_pair(const char *name, unsigned dim, unsigned degree);
```

The application must be linked against `libalbas` and include the header file `albas.h` to use it. `stokes_pair()` can be invoked with the symbolic names `"Mini"`, `"BernardiRaugel"`, `"CrouzeixRaviar"` and `"TaylorHood"` and returns the requested Stokes-discretization, which in all cases except for Taylor-Hood consists of a chain of local basis functions where the first part of the chain contains the Lagrange-component and the other parts the "bubbly" add-ons used to stabilize the resulting Stokes-pair. See also Section 3.5.3 and Section 3.7.

3.6 Implementation of finite element spaces

3.6.1 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;
  int               rdim;
  DBL_LIST_NODE     chain;
  const FE_SPACE   *unchained;
};
```

Description:

name holds a textual description of the finite element space. Note that **name** is duplicated by calling `strdup(3)`

admin pointer to the DOF administration for the DOFs of this finite element space, see Section 3.3.1.

bas_fcts pointer to the local basis functions, see Section 3.5.1.

mesh pointer to the underlying mesh, see Section 3.2.12.

rdim The dimension of the range of the elements of this finite element space, as ALBERTA nowadays supports vector-valued basis functions it becomes now important whether a given finite element space is actually meant for scalar functions or for vector fields. See also Section 3.5.2.

chain List pointer to a chain of finite element spaces which form a direct sum, see Section 3.7. Such a direct sum is based on a chain of local basis functions as described by Section 3.5.3.

unchained If the finite element space is part of a direct sum of finite element spaces (and thus **chain** is the link to the other elements of this direct sum) then **unchained** is a copy of the FE_SPACE which is unaware of this fact, i.e. `FE_SPACE.unchained.chain` points back to itself. If the finite element space does not form part of a direct sum, then **unchained** simply points back to the same FE_SPACE. See also Section 3.7 and Section 3.5.

Some remarks:

- 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.
- Using direct sums of finite element spaces which are chained together using the `FE_SPACE.chain`-component has the effect that all derived structures are also chains of objects, coefficient vectors become chains of coefficient vectors, matrices become block matrices, where the blocks are chained together using chains for rows and columns. The same holds for the per-element vectors and matrices.
- ALBERTA provides full support for these chains in its infra-structure for the assembling of the discrete systems, as well as in the solver infra-structure and in the support functions for the computation of errors and error estimates.

3.6.2 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 BAS_FCTS *bas_fcts, int rdim,
                              FLAGS adm_flags);
const FE_SPACE  *get_dof_space(MESH *mesh, const char *name,
                               const int ndof[N_NODE_TYPES],
                               FLAGS adm_flags);
```

Descriptions

get_fe_space(mesh, name, bas_fcts, rdim, adm_flags) defines a new finite element space on mesh; it looks for an existing dof_admin defined on mesh, which manages DOFs uniquely defined by bas_fcts->dof_admin->n_dof (compare Section 3.3.1); if such a dof_admin is not found, a new dof_admin is created.

Parameters

mesh A pointer to the underlying triangulation.

name A fancy name, used for pretty-printing and debugging.

bas_fcts The underlying basis functions. If bas_fcts is a disjoint union, or "chain", of local basis functions sets, then the resulting finite element space will be the direct sum of the finite element spaces defined by the respective components of the disjoint union of local basis function sets, see Section 3.7 and Section 3.5. The component unchained of the FE_SPACE structure will – for each component of the direct sum – point to an FE_SPACE instance which is ignorant of the fact that it forms part of a direct sum of finite element spaces.

rdim The dimension of the range of the elements of the finite element space. Now that ALBERTA also support DIM_OF_WORLD-valued basis functions, this parameter plays an important role; without attaching a "range-dimension" to a finite element space it would be hardly possible to assemble discrete systems in a consistent manner. Of course, if the underlying basis functions are vector-valued for themselves, then rdim has to equal DIM_OF_WORLD as well. If the underlying local basis functions are scalar-valued, then rdim may be either 1 to generate a finite element space consisting of scalar functions, or DIM_OF_WORLD to generate a finite element space consisting of DIM_OF_WORLD-valued functions. Other values for rdim are not supported.

adm_flags Currently adm_flags is the bit-wise or of ADM_PRESERVE_COARSE_DOFS and/or ADM_PERIODIC. If the flag ADM_PRESERVE_COARSE_DOFS is set, then it requests that DOFs normally be deleted during refinement should be preserved instead. This is necessary for higher order multi-grid implementations as well as for the internal maintenance of submeshes, see Section 3.9. For a detailed description of which DOFs would normally be deleted, see Section 3.3.

ADM_PERIODIC requests a periodic finite element space where DOFs across periodic walls are identified. See Section 3.10.

return value The return value is a newly created FE_SPACE structure, where name is duplicated, and the members mesh, bas_fcts and admin are adjusted correctly.

get_dof_space(mesh, name, n_dof, adm_flags) performs a similar task as get_fe_space(), however, the resulting "space" is not bound to a BAS_FCTS instance. Instead, the argument n_dof determines the distribution of the DOFs across the sub-simplices. The elements of n_dof determine how many degrees of freedom are tied to each sub-simplex on each element, the mapping is defined by the NODE_TYPE enumeration type, see the source code listing on page 74, i.e. VERTEX == 0, CENTER == 1, EDGE == 2, FACE == 3. The following code-snippet defines an "FE_SPACE" with 42 DOFs on each face:

```
int n_dof[N_NODE_TYPES] = {0, 0, 0, 42};
const FE_SPACE *face_dof_space;

face_dof_space =
  get_dof_space(mesh, "face_dofs", n_dof, ADM_FLAGS_DFLT);
```

The selection of finite element spaces defines 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. Each call of `get_fe_space()` requires the internal adjustment of the `el->dof` pointer arrays which is potentially expensive, since information about which elements share a vertex/edge/face must be calculated for the current triangulation. It is therefore advisable (but not necessary) to allocate all finite element spaces before refining the mesh.

Since a mesh only gives 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.

3.6.1 Example (Initializing DOFs for Stokes and Navier–Stokes)**.** Now, as an example we look at a possible `main` function. In the example we want to define two finite element spaces on the mesh, for a mixed finite element formulation of the Stokes or 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, dim;

int main()
{
  const MESH       *mesh;
  const BAS_FCTS   *lagrange;
  MACRO_DATA       *data;

  TEST_EXIT(degree > 1)("degree must be greater than 1\n");

  ...

  data = read_macro(filename);
  mesh = GET_MESH(dim, "ALBERTA mesh", data, NULL, NULL);
  free_macro_data(data);

  lagrange = get_lagrange(mesh->dim, degree);
  u_fe = get_fe_space(
    mesh, "Velocity space", lagrange, DIM_OF_WORLD, ADM_FLAGS_DFLT);

  lagrange = get_lagrange(mesh->dim, degree-1);
  p_fe = get_fe_space(mesh, "Pressure space", lagrange, 1, ADM_FLAGS_DFLT);

  ...

  return;
}
```

This will provide all DOFs for the two finite element spaces on the elements and the corresponding DOF_ADMINs will have information about the access to 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() before allocating finite element spaces.

3.7 Direct sums of finite element spaces

Sometimes it is necessary to use finite element spaces which are direct sums of a standard space plus a more or less bizarre add-on. The velocity space for several stable mixed discretizations of the Stokes problem, for instance, has this structure: it consists of piece-wise linear elements plus an element bubble for the so-called "Mini"-element, piece-wise linear elements plus face-bubbles for the "Bernardi-Raugel"-element, for the "Crouzeix-Raviart" element it consists of piece-wise quadratic elements plus an element-bubble in 2d, and forms a direct sum with three components in 3d, where face-bubble have to be added in addition to the element-bubble, which was already present in 2d.

3.7.1 Data structures for disjoint unions and direct sums

ALBERTA support such direct sums of finite element spaces. The fundaments for such direct sums are formed by "chains" of BAS_FCTS-structures, modeling the disjoint union of local basis-function sets, see Section 3.5.3. A disjoint union of basis functions sets is implemented using a cyclic, doubly linked list. This affects all structures which are functionally based on the structure of the local set of basis functions: the FE_SPACE-structure, the DOF_XXX_VEC coefficient vectors, and their local counter parts name EL_XXX_VEC (XXX being used as a place holder for the type, e.g. XXX ≡ REAL), the matrix structure DOF_MATRIX (and its local count-part), the frame-work used for assembling the discrete systems and – of course – the quadrature caches defined by the QUAD_FAST structure. Basically, all structures which are directly of indirectly derived from such a disjoint union of local basis function sets inherit this "disjoint union" layout and come with a list-node component which implements this connectivity within ALBERTA. The list node itself is a simple doubly-linked list node, namely

```
typedef struct dbl_list_node DBL_LIST_NODE;

struct dbl_list_node
{
  struct dbl_list_node *next;
  struct dbl_list_node *prev;
};
```

In all the structures needing such a list-node, there are components named ...chain, compare for instance the source code listing for the BAS_FCTS structure on page 145:

```
struct bas_fcts
{
  ... /* other stuff */
  DBL_LIST_NODE chain;
  ... /* more stuff */
};
```

This becomes even more complicated in the context of matrix-structures, the `EL_MATRIX` structure (compare the source-code listing on page 252), for instance, needs two list-node components, namely

```
struct el_matrix
{
  ... /* other stuff */
  DBL_LIST_NODE row_chain;
  DBL_LIST_NODE col_chain;
  ... /* more stuff */
};
```

because the local row-space as well as the local column-space may be direct sums of local finite element spaces. So matrices carry a block-matrix structure if the underlying spaces are direct sums, and the `col_chain` and `row_chain` give the link between the different blocks, each block being a single `EL_MATRIX` structure (or whatever other matrix-structure).

Conceptionally, all these lists are *cyclic*, and there is no dedicated list-head. This may bear the risk for certain kinds of programming errors, but is, on the other hand, quite nice for the implementation, because in this setting an ordinary `BAS_FCTS` structure which is not a disjoint union of several basis function sets is at the same time a disjoint union with one component, so the code does not need to differentiate between direct sums and single-component objects, thus eliminating the need to introduce new data-types to model direct sums of function spaces.

3.7.2 List-management and looping constructs

This section describes some basic support macro and functions for list-management like adding to direct sum or deleting from them, as well as some loop-constructs. Generally, all the macros come in three flavours: with a `CHAIN_...-`, `ROW_CHAIN_...` and a `COL_CHAIN_...` prefix, acting on the `chain`, `row_chain` and `col_chain` list-nodes in the respective data-structures. This is the only difference between the three flavours of macros, so we describe only the variant with the `CHAIN`-prefix.

`CHAIN_INIT(elem)` Initialize `elem->chain`; that is make `elem->chain.next` and `elem->chain.prev` to `&elem->chain`. This defines the empty, respectively one-component list.

`CHAIN_INITIALIZER(name)` Perform the same task as `CHAIN_INIT(elem)`, but in the context of a static initialization, e.g.

```
static BAS\_FCTS bfcts = {
  ... /* other stuf */,
  CHAIN_INITIALIZER(bfcts),
  ... /* more stuff */
};
```

`CHAIN_LENGTH(head)` Compute the number of list elements in the cyclic list `head->chain`.

`CHAIN_SINGLE(var)` Evaluate to `true` if `var->chain` is the one-element list.

`CHAIN_NEXT(var, type)`

`CHAIN_PREV(var, type)` Return a pointer to the element following, respectively preceding `var`. The argument `type` must denote the data-type of `var`, e.g.

```
const BAS_FCTS *next_bfcts = CHAIN_NEXT(bfcts, const BAS_FCTS);
const BAS_FCTS *prev_bfcts = CHAIN_PREV(bfcts, const BAS_FCTS);
```

CHAIN_ADD_HEAD(head, elem)

CHAIN_ADD_TAIL(head, elem) Add `elem` to the head, respectively to the tail of `head->chain`. Adding to the head means that `elem` will become the element *following* `head`, and adding to the tail means that `elem` will become the list element preceding `head`. In particular, adding to either the end or tail of an one-element list will produce the same results.

CHAIN_DEL(elem) Delete `elem->chain` from any list it may belong to, and call `CHAIN_INIT(elem)` afterwards. The result will be that `elem` becomes a one-element list.

CHAIN_FOREACH(ptr, head, type) Loop over all element of `head->chain` which follow `head->chain`, excluding the element pointed to by `head` itself. Something similar to `CHAIN_LENGTH(head)` mentioned above could for instance be implemented as

```
int bfcts_chain_length(const BAS_FCTS *head)
{
  const BAS_FCTS *pos;

  int len = 1;
  CHAIN_FOREACH(pos, head, const BAS_FCTS) {
    ++len;
  }
  return len;
}
```

CHAIN_FOREACH_SAVE(ptr, next, head, type) Similar to `CHAIN_FOREACH()`, but allow for deletion of list-elements during the loop. For this to work an additional pointer has to be provided which is points to the element following the current element. This way, the current element – `pos` – maybe safely removed from the list and deleted during the loop:

```
typedef struct my_chained_object
{
  ... /* stuff */
  DBL_LIST_NODE chain;
  ... /* other suff */
};

int delete_my_chained_object(MY_CHAINED_OBJECT *list)
{
  MY_CHAINED_OBJECT *pos, *next;

  CHAIN_FOREACH_SAFE(pos, next, list, MY_CHAINED_OBJECT) {
    CHAIN_DEL(pos);
    MEM_FREE(pos, 1, MY_CHAINED_OBJECT);
  }
  MEM_FREE(head, 1, MY_CHAINED_OBJECT);
}
```

CHAIN_FOREACH_REV(ptr, head, type)

CHAIN_FOREACH_REV_SAVE(ptr, next, head, type) Same as the non-REV-counterparts explained above, but the loop is perform in the reverse direction, following `head->chain.prev` instead of `head->chain.next`.

CHAIN_DO(list, type)

CHAIN_WHILE(list, type) Perform a loop over the list, include the first element (in contrast to `CHAIN_FOREACH()` which always skips the first element:

```
int bfcts_chain_length(const BAS_FCTS *pos)
{
  int len = 0;
  CHAIN_DO(pos, const BAS_FCTS) {
    ++len;
  } CHAIN_WHILE(pos, const_BAS_FCTS);
  return len;
}
```

CHAIN_DO_REV(list, type)

CHAIN_WHILE_REV(list, type) Same as the `CHAIN_DO()-CHAIN_WHILE()` pair, but loop in reverse direction, following `list->head.prev` instead of `list->head.next`.

FOREACH_DOF(fe_space, todo, next) A replacement for `FOR_ALL_DOFS()`, which implements an outer loop over the components of the chain, calling `FOR_ALL_DOFS()` for each component in turn. In this setting `todo` is a code-block which is executed for each `DOF` and `next` is a code block which is executed at the end of the inner `FOR_ALL_DOFS()` call and should be used to roll data to the next chain-component. The first argument is moved on to Compare also Section 3.3.5. Example:

```
void print_all_values(const DOF_REAL_VEC *dof_vec)
{
  FOREACH_DOF(dof_vec->fe_space,
              /* todo-block*/
              MSG("value: %e\n", dof_vec->vec[dof]),
              /* next-block */
              dof_vec = CHAIN_NEXT(dof_vec, const DOF_REAL_VEC));
}
```

FOREACH_DOF_DOW(fe_space, todo, todo_cart, next) A special version of `FOREACH_DOF()` for chains mixing vector-valued finite element functions based on either scalar- or `DIM_OF_WORLD`-valued basis functions: in this context the coefficient vectors for scalar basis functions consist of vector valued coefficients, while the coefficient vectors for scalar basis-functions consist of scalars, e.g.

```
void print_all_values_dow(const DOF_REAL_VEC_D *dof_vec)
{
  FOREACH_DOF_DOW(dof_vec->fe_space,
              /* todo-block*/
              MSG("value: %e\n", dof_vec->vec[dof]),
              /* todo_cart-block */
              MSG("value: "FORMAT_DOW"\n",
                  EXPAND_DOW(((const DOF_REAL_D_VEC *)dof_vec)->vec[dof])),
              /* next-block */
              dof_vec = CHAIN_NEXT(dof_vec, const DOF_REAL_VEC_D));
}
```

Note the difference between a DOF_REAL_VEC_D coding a vector valued finite-element function, and a DOF_REAL_D_VEC, coding for a vector storing REAL_D-valued coefficients. The name todo_cart stems from the fact that the parts of the direct sum belonging to scalar-valued basis functions is in fact a Cartesian product space of scalar finite element spaces.

FOREACH_FREE_DOF(fe_space, todo, next)

FOREACH_FREE_DOF_DOW(fe_space, todo, todo_cart, next) Similar to the other two loop-macros, but in the inner loop the FOR_ALL_FREE_DOFS)-macro is called, see Section 3.3.5.

3.7.3 Managing temporary coefficient vectors

Sometimes it is useful to hook a contiguous, flat array of values into a "dummy" DOF_XXX_VEC structure. Most iterative solver available from third party sources, for instance, as well as the "OEM"-library functions (Orthogonal Error Methods, see Section 4.10) expect matrix-vector routines which accept pointers to such arrays, but the matrix-vector routines implementing the operation of DOF_MATRIXes on finite element coefficient vectors only accept arguments of type DOF_REAL[_D]_VEC[_D]-type (see Section 3.3.7).

3.7.1 Compatibility Note. *Prior to the introduction of the support for direct sums of finite element spaces, this task was quite easy, have a look at the following code-excerpt, implementing a matrix-vector routine for an* older *version of ALBERTA:*

```
void mat_vec_s(void *ud, int dim, const REAL *x, REAL *y)
{
  DOF_REAL_VEC  dof_x = {nil, nil, "x", 0, nil, nil, nil};
  DOF_REAL_VEC  dof_y = {nil, nil, "y", 0, nil, nil, nil};
  struct mv_data  *data      = (struct mv_data *)ud;
  const DOF_ADMIN *admin     = data->matrix->row_fe_space->admin;

  dof_x.fe_space = data->matrix->col_fe_space;
  dof_y.fe_space = data->matrix->row_fe_space;
  dof_x.size = dof_y.size = dim;
  dof_x.vec = (REAL *) x;
  dof_y.vec = y;

  dof_mv(data->transpose, data->matrix, &dof_x, &dof_y);
}
```

However, this will no longer work, because the dof_mv() *routine expects its argument to model direct sums of finite element spaces, and even for the standard case it expects the* dof_x.chain *and* dof_y.chain *list-nodes to be initialized properly, defining "direct sums" consisting of a single summand.*

To aid the task of defining such "dummy"-vectors, there are some support functions which take care of transferring the direct-sum-structure of the finite element space in question to the temporaries which are needed to interface, e.g., to the matrix-vector routines pairing DOF_MATRIXes with DOF-vectors. To improve the readability of the code, it is maybe advisable to use the new routines anyway. The example given above in Compatibility Note 3.7.1 collapses to the following, using the routines explained further below:

```
void mat_vec_s(void *ud, int dim, const REAL *x, REAL *y)
  struct mv_data *data = (struct mv_data *)ud;
  DOF_REAL_VEC *dof_x = data->x_skel;
  DOF_REAL_VEC *dof_y = data->y_skel;

  distribute_to_dof_real_vec_skel(data->x_skel, x);
  distribute_to_dof_real_vec_skel(data->y_skel, y);

  dof_mv(data->transpose, data->matrix, data->mask, dof_x, dof_y);
}
```

Well, it spares only a few lines. But on the other hand, prescribing an API for tasks like this increases portability between different versions of ALBERTA, because only with such an API it is possible to hide the more "dirty" details, or future extensions, from application programs. We continue with the description of the available functions. The example program for the non-linear reaction diffusion program contained in the demo-package (and described in Section 2.3 also makes use of these support functions.

The available functions are as follows:

```
size_t dof_real_vec_d_length(const FE_SPACE *fe_space);
size_t dof_real_d_vec_length(const FE_SPACE *fe_space);
size_t dof_real_vec_length(const FE_SPACE *fe_space);

DOF_REAL_VEC *init_dof_real_vec_skel(DOF_REAL_VEC vecs[],
                                     const char *name,
                                     const FE_SPACE *fe_space);
DOF_REAL_D_VEC *init_dof_real_d_vec_skel(DOF_REAL_D_VEC vecs[],
                                         const char *name,
                                         const FE_SPACE *fe_space);
DOF_REAL_VEC_D *init_dof_real_vec_d_skel(DOF_REAL_VEC_D vecs[],
                                         const char *name,
                                         const FE_SPACE *fe_space);
DOF_SCHAR_VEC *init_dof_schar_vec_skel(DOF_SCHAR_VEC vecs[],
                                       const char *name,
                                       const FE_SPACE *fe_space);

DOF_REAL_VEC *get_dof_real_vec_skel(const char *name,
                                    const FE_SPACE *fe_space,
                                    SCRATCH_MEM scr);
DOF_REAL_D_VEC *get_dof_real_d_vec_skel(const char *name,
                                        const FE_SPACE *fe_space,
                                        SCRATCH_MEM scr);
DOF_REAL_VEC_D *get_dof_real_vec_d_skel(const char *name,
                                        const FE_SPACE *fe_space,
                                        SCRATCH_MEM scr);
DOF_SCHAR_VEC *get_dof_schar_vec_skel(const char *name,
                                      const FE_SPACE *fe_space,
                                      SCRATCH_MEM scr);

void distribute_to_dof_real_vec_skel(DOF_REAL_VEC *skel, const REAL *data);
void distribute_to_dof_real_d_vec_skel(DOF_REAL_D_VEC *skel, const REAL
   *_data);
void distribute_to_dof_real_vec_d_skel(DOF_REAL_VEC_D *skel, const REAL *data);
void distribute_to_dof_schar_vec_skel(DOF_SCHAR_VEC *skel, const S_CHAR *data);

void copy_to_dof_real_vec(DOF_REAL_VEC *vecs, const REAL *data);
```

```
void copy_to_dof_real_d_vec(DOF_REAL_D_VEC *vecs, const REAL *_data);
void copy_to_dof_real_vec_d(DOF_REAL_VEC_D *vecs, const REAL *data);
void copy_to_dof_schar_vec(DOF_SCHAR_VEC *vecs, const S_CHAR *data);

void copy_from_dof_real_vec(REAL *data, const DOF_REAL_VEC *vecs);
void copy_from_dof_real_d_vec(REAL_D *data, const DOF_REAL_D_VEC *vecs);
void copy_from_dof_real_vec_d(REAL *data, const DOF_REAL_VEC_D *vecs);
void copy_from_dof_schar_vec(S_CHAR *data, const DOF_SCHAR_VEC *vecs);
```

Descriptions for each of the functions listed above:

Synopsis

```
length = dof_real_vec_d_length(fe_space);
length = dof_real_d_vec_length(fe_space);
length dof_real_vec_length(fe_space);
```

Description

Compute the total dimension of `fe_space`.

Parameters

`fe_space` The finite element space to compute the dimension of.

Return Value

The total dimension of the direct sum of finite element spaces. Note that vector-valued coefficients are counted with their `DIM_OF_WORLD`-multiplicity. The return value is of type `size_t`.

Synopsis

```
head_vec = init_dof_real_vec_skel(&dof_vec_storage[0], name, fe_space);
head_vec = init_dof_real_d_vec_skel(&dof_vec_storage[0], name, fe_space);
head_vec = init_dof_real_vec_d_skel(&dof_vec_storage[0], name, fe_space);
head_vec = init_dof_schar_vec_skel(&dof_vec_storage[0], name, fe_space);
```

Description

Turn an uninitialized storage area consisting of sufficiently many `DOF_REAL[_D]_VEC[_D]` or `DOF_SCHAR_VEC` objects and turn it into a concatenated list, describing a coefficient vector for the finite element space specified by `fe_space`. The resulting dof-vectors are, of course, not hooked into the lists of `fe_space->admin`, and are not subject to automatic resizing during mesh adaptation. Further, they do not carry storage for data, i.e. their `vec` component does not point to a valid storage area (but see `distribute_to_dof_XXX_vec_skel()` below). Therefore we call the resulting object a "skeleton", which also explains the name of this function.

Arguments

`dof_vec_storage` Pointer to a storage area, pointing to sufficiently many DOF-vectors, stored consecutively in memory (i.e. `dof_vec_storage` is a flat array of sufficient size). The number of the objects needed can be determined by calling `CHAIN_LENGTH(fe_space)`.

name A descriptive name for the skeleton. It is hooked into the `name` component of each of the individual DOF-vectors.

fe_space The underlying finite element space. `fe_space` determines the layout of the resulting chained coefficient vector.

Return Value

The first component of the multi-component coefficient vector.

Synopsis

```
head_vec = get_dof_real_vec_skel(name, fe_space, scr);
head_vec = get_dof_real_d_vec_skel(name, fe_space, scr);
head_vec = get_dof_real_vec_d_skel(name, fe_space, scr);
head_vec = get_dof_schar_vec_skel(name, fe_space, scr);
```

Description

Allocate and initialize a temporary DOF-vector from a scratch-memory pool, see Section 3.1.3.4. This functionally equivalent to

```
DOF_REAL_VEC *get_dof_real_vec_skel(const char *name,
                                    const FE_SPACE *fe_space,
                                    SCRATCH_MEM scr)
{
  DOF_REAL_VEC *vecs;

  vecs = SCRATCH_MEM_ALLOC(scr, CHAIN_LENGTH(fe_space), DOF_REAL_VEC);

  return init_dof_real_vec_skel(vecs, name, fe_space);
}
```

Likewise for the other types of DOF-vectors.

Arguments

name Symbolic name.

fe_space The underlying finite element space.

scr Pointer to a scratch-memory pool, see Section 3.1.3.4. Consequently, the objects generated here can and will be destroyed when the scratch-memory pool is deleted by calling `SCRATCH_MEM_ZAP(scr)`.

Return Value

A pointer to the head of the chain.

Synopsis

```
distribute_to_dof_real_vec_skel(dof_vec_skel, contiguous_data);
distribute_to_dof_real_d_vec_skel(dof_vec_skel, contiguous_data);
distribute_to_dof_real_vec_d_skel(dof_vec_skel, contiguous_data);
distribute_to_dof_schar_vec_skel(dof_vec_skel, contiguous_data);
```

Description

Distribute a contiguous piece of data specified by `contiguous_data` to a DOF-vector skeleton as generated by a call to `get_dof_XXX_vec_skel()` or `init_dof_XXX_vec_skel()` described above. "Distribute" in this context means to initialize the `vec` component of each part of the DOF-vector chain with the proper location into `contiguous_data`. The data will be distributed to the individual components according to the dimension of the component of the finite element space they belong to.

This function must be called prior to passing a DOF-vector skeleton to any function expecting a "real" DOF-vector.

To only copy data between contiguous arrays and DOF-vectors, see `copy_to|from_dof_XXX_vec()` below.

Arguments

`dof_vec_skel` The DOF-vector skeleton.

`contiguous_data` A piece of contiguous data with `dof_XXX_vec_length(fe_space)` many items.

Synopsis

```
copy_to_dof_real_vec(dof_vec, contiguous_data);
copy_to_dof_real_d_vec(dof_vec, contiguous_data);
copy_to_dof_real_vec_d(dof_vec, contiguous_data);
copy_to_dof_schar_vec(dof_vec, contiguous_data);
```

Description

Copy data from a flat array containing at least `dof_XXX_vec_length()` many items to a DOF-vector object, taking care of the chained structure of coefficient vectors belonging to direct sums of finite element spaces.

This function will overwrite all the data stored in `dof_vec`.

Arguments

`dof_vec` The destination of the copy operation.

`contiguous_data` The source of the copy operation.

Return Value

Synopsis

```
copy_from_dof_real_vec(contiguous_data, dof_vec);
copy_from_dof_real_d_vec(contiguous_data, dof_vec);
copy_from_dof_real_vec_d(contiguous_data, dof_vec);
copy_from_dof_schar_vec(contiguous_data, dof_vec);
```

Description

Copy data from a DOF-vector to a flat array containing at least `dof_XXX_vec_length()` many items, taking care of the chained structure of coefficient vectors belonging to direct sums of finite element spaces.

This function will overwrite all the data stored in `contiguous_data`.

Arguments

`contiguous_data` Destination of the copy operation.

`dof_vec` Source of the copy operation.

Return Value

3.7.4 Data transfer during mesh adaptation

If the underlying finite element space is indeed a direct sum, then it is an inconvenient task to install the default refinement and coarsening functions into each component of the chain. For a single-component sum, the following suffices:

```
extern DOF_REAL_VEC_D *vector;

vector->refine_inter = vector->fe_space->bas_fcts->real_refine_inter_d;
```

However, if `vector` is only the first part of a chain, then the following elements of the chain are not touched by this operation, one would have to do something similar to the following:

```
extern DOF_REAL_VEC_D *vector;

CHAIN_DO(uh,DOF_REAL_VEC_D) {
  uh->refine_interpol = uh->fe_space->bas_fcts->real_refine_inter_d;
} CHAIN_WHILE(uh, DOF_REAL_VEC_D);
```

There are small inline functions defined through the inclusion which perform just this, above code, e.g., is wrapped into the following function:

```
static inline void set_refine_inter_dow(DOF_REAL_VEC_D *uh)
{
  CHAIN_DO(uh,DOF_REAL_VEC_D) {
    uh->refine_interpol = uh->fe_space->bas_fcts->real_refine_inter_d;
  } CHAIN_WHILE(uh, DOF_REAL_VEC_D);
}
```

As the code is self-explaining (at least after reading Section 3.7.2 and Section 3.3.3), we only list the proto-types here:

```
static inline void set_refine_inter(DOF_REAL_VEC *uh);
static inline void set_refine_inter_d(DOF_REAL_D_VEC *uh);
static inline void set_refine_inter_dow(DOF_REAL_VEC_D *uh);

static inline void set_coarse_inter(DOF_REAL_VEC *uh);
static inline void set_coarse_inter_d(DOF_REAL_D_VEC *uh);
static inline void set_coarse_inter_dow(DOF_REAL_VEC_D *uh);

static inline void set_coarse_restrict(DOF_REAL_VEC *uh);
static inline void set_coarse_restrict_d(DOF_REAL_D_VEC *uh);
static inline void set_coarse_restrict_dow(DOF_REAL_VEC_D *uh);
```

3.7.5 Forming direct sub-sums

Sometimes it is handy to refer only to selected components of a chain of objects. The following routines perform this task by forming sub-chains of objects, which then belong to a direct sub-sum, so to say:

```
BAS_FCTS *bas_fcts_sub_chain(SCRATCH_MEM scr, const BAS_FCTS *bas_fcts,
                             FLAGS which);
void update_bas_fcts_sub_chain(BAS_FCTS *bas_fcts);
FE_SPACE *fe_space_sub_chain(SCRATCH_MEM scr, const FE_SPACE *fe_space,
                             FLAGS which);
void update_fe_space_sub_chain(FE_SPACE *fe_space);

DOF_REAL_VEC *dof_real_vec_sub_chain(SCRATCH_MEM scr,
                                     const DOF_REAL_VEC *vec,
                                     FLAGS which);
DOF_REALD_VEC *dof_real_d_vec_sub_chain(SCRATCH_MEM scr,
                                        const DOF_REAL_D_VEC *vec,
                                        FLAGS which);
DOF_REAL_VEC_D *dof_real_vec_d_sub_chain(SCRATCH_MEM scr,
                                         const DOF_REAL_VEC_D *vec,
                                         FLAGS which);
DOF_DOF_VEC *dof_dof_vec_sub_chain(SCRATCH_MEM scr,
                                   const DOF_DOF_VEC *vec,
                                   FLAGS which);
DOF_INT_VEC *dof_int_vec_sub_chain(SCRATCH_MEM scr,
                                   const DOF_INT_VEC *vec,
                                   FLAGS which);
DOF_UCHAR_VEC *dof_uchar_vec_sub_chain(SCRATCH_MEM scr,
                                       const DOF_UCHAR_VEC *vec,
                                       FLAGS which);
DOF_SCHAR_VEC *dof_schar_vec_sub_chain(SCRATCH_MEM scr,
                                       const DOF_SCHAR_VEC *vec,
                                       FLAGS which);
DOF_PTR_VEC *dof_ptr_vec_sub_chain(SCRATCH_MEM scr,
                                   const DOF_PTR_VEC *vec,
                                   FLAGS which);

void update_dof_real_vec_sub_chain(const DOF_REAL_VEC *sub_vec);
void update_dof_real_d_vec_sub_chain(const DOF_REAL_D_VEC *sub_vec);
void update_dof_real_vec_d_sub_chain(const DOF_REAL_VEC_D *sub_vec);
void update_dof_dof_vec_sub_chain(const DOF_DOF_VEC *sub_vec);
void update_dof_int_vec_sub_chain(const DOF_INT_VEC *sub_vec);
void update_dof_uchar_vec_sub_chain(const DOF_UCHAR_VEC *sub_vec);
void update_dof_schar_vec_sub_chain(const DOF_SCHAR_VEC *sub_vec);
void update_dof_ptr_vec_sub_chain(const DOF_PTR_VEC *sub_vec);


DOF_MATRIX *dof_matrix_sub_chain(SCRATCH_MEM scr, const DOF_MATRIX *A,
                                 FLAGS row_which, FLAGS col_which);
void update_dof_matrix_sub_chain(DOF_MATRIX *sub_M);
```

The general idea is to make shallow copies of selected components of the original chain, shallow in the sense that the copies share the underlying data (e.g. such a shallow copy of a `DOF_REAL_VEC` would share the `vec` component with the original instance). Those copies are then chained-together, forming sub-chains. The selection of the components is performed by means of a bit-mask, called `which` in the proto-types listed above. If bit n in the `which`-mask

is set, then the component number n takes part in forming the sub-chain. Analogously for matrices where we need a two masks, one for the rows, and another one for the columns of the block-matrix.

Descriptions for the individual groups of functions:

Synopsis

```
sub_chain = bas_fcts_sub_chain(scratch_mem, master_chain, which);
sub_chain = fe_space_sub_chain(scratch_mem, master_chain, which);
sub_chain = dof_real_vec_sub_chain(scratch_mem, master_chain, which);
sub_chain = dof_real_vec_d_sub_chain(scratch_mem, master_chain, which);
sub_chain = dof_real_d_vec_sub_chain(scratch_mem, master_chain, which);
sub_chain = dof_dof_vec_sub_chain(scratch_mem, master_chain, which);
sub_chain = dof_int_vec_sub_chain(scratch_mem, master_chain, which);
sub_chain = dof_uchar_vec_sub_chain(scratch_mem, master_chain, which);
sub_chain = dof_schar_vec_sub_chain(scratch_mem, master_chain, which);
sub_chain = dof_ptr_vec_sub_chain(scratch_mem, master_chain, which);
sub_matrix =
  dof_matrix_sub_chain(scratch_mem, matrix, row_which, col_which)
```

Description

Form a sub-chain of the specified "master"-chain, using the number of bits set in `which` to select the components to copy. Sub-chains are chains consisting of shallow copies of the members of the master-chain, which share the underlying coefficient data which the members of the master chain. A sub-chain is up-to-date after generating it, however, if the size of the master objects changed, prominently because of mesh adaptation, the corresponding update routine has to be called to update the sub-chain accordingly, see below. Note that for DOF-vectors and -matrices the structure-component `unchained` of the sub-chain objects will point to the original objects. Note also that this does *not* hold for sub-chains of `BAS_FCTS` and `FE_SPACE` objects: here the component `unchained` will always point to an instance of those objects which is not concatenated which any other object, i.e. is indeed an unchained copy.

Note that the sub-chain will be destroyed when the scratch-memory handle `scratch_mem` is deleted by calling `SCRATCH_MEM_ZAP(scratch_mem)`.

Arguments

`scratch_mem` A pointer to a scratch-memory area, see Section 3.1.3.4.

`master_chain` The master-chain.

`which` A bit mask which determines which parts of `master_chain` take part in forming the sub-chain: if bit n is set in the `which`-mask, then component number n of the master-chain will make its way into the sub-chain.

Return Value

A pointer to the first element of the sub-chain.

Synopsis

```
update_dof_real_vec_sub_chain(sub_chain);
update_dof_real_d_vec_sub_chain(sub_chain);
update_dof_real_vec_d_sub_chain(sub_chain);
update_dof_dof_vec_sub_chain(sub_chain);
update_dof_int_vec_sub_chain(sub_chain);
update_dof_uchar_vec_sub_chain(sub_chain);
update_dof_schar_vec_sub_chain(sub_chain);
update_dof_ptr_vec_sub_chain(sub_chain);
update_dof_matrix_sub_chain(sub_chain);
```

Description

Update a sub-chain after mesh-adaptation. Note that there are no "updaters" for sub-chains of `BAS_FCTS` and `FE_SPACE` objects, simply because the sub-chains need not be updated in this case.

Otherwise, the application must call `update_XXX_sub_chain()` after adapting the mesh. Otherwise the meta-data stored in the elements forming the sub-chain will be inconsistent with the state of the mesh.

Arguments

`sub_chain` The head of the sub-chain. The master chain is not needed, because it can be accessed via `sub_chain->unchained`.

3.8 Data structures for parametric meshes

The current version of ALBERTA offers support for so-called *parametric meshes* which are triangulations where some or all of the simplices are non-linear images of the reference element. Typically, the transformation from the reference element $\hat{S}$ to the curved simplex S is a polynomial, but in principle this need not be the case. ALBERTA has predefined polynomial parameterisations up to polynomial degree 4: $S = F_S(\hat{S})$, $F_S \in \mathbb{P}_k(\hat{S})$ for $k = 1, 2, 3, 4$. The limitation $k \leq 4$ just means that piecewise polynomial parameterisations up to the maximal degree for the Lagrange basis functions within ALBERTA are supported (Section 3.5).

The standard case for applications is the iso-parametric approximation of curved boundaries; care has to be taken when the polynomial degree of the parameterisation is so high that some of the Lagrange-nodes fall into the interior of the simplex. ALBERTA implements the algorithm developed in [15]. The suite of demo-programs shipped with the ALBERTA-package contains a program called `ellipt-isoparam`, which implements the discretization of Poisson's equation on an iso-parametric triangulation of a unit-disc.

Many other applications besides isoparametric boundary approximation are conceivable, for example in moving finite elements, where the positions of nodes may change with time and need to be described by a time dependent parameterization. Stationary example programs for 1, 2 and 3 dimensional parametric meshes can again be found in the demo-suite:

`src/Common/ellipt-sphere.c` Poisson's equation on the 1-, 2- and 3-dimensional unit-sphere, i.e. $S^k \subset \mathbb{R}^{k+1}$ $(1 \leq k \leq 3)$.

`src/Common/ellipt-torus.c` Poisson's equation on the 1- , 2- and 3-torus, i.e. $T^k \subset \mathbb{R}^{k+1}$ $(1 \leq k \leq 3)$.

src/3d/ellipt-moebius.c Poisson's equation on an embedded Moebius-strip (yes, ALBERTA can handle unorientable meshes).

src/4d/ellipt-klein-bottle.c Embedded Klein's bottle.

src/5d/ellipt-klein-3-bottle.c Embedded non-orientable 3-manifold in $\mathbb{R}^5$, similar to a Klein's bottle, but one dimension higher.

Using parametric elements does not imply a fundamental change of data structures within ALBERTA. The mesh still consists of a hierarchical collection of `EL` structures, however these only represent the topological structure of the mesh. The coordinate and shape information of all elements, standard or parametric, is stored using an internal `DOF_REAL_D_VEC coords` representing the global parametrization encoded in F_S for all S. The finite element space containing `coords` is a standard Lagrange space of order 1, 2, 3 or 4.

A mesh may be turned into a parametric mesh with piece-wise polynomial parameterization by calling the function `use_lagrange_parametric()` described below. This allocates `coords` and turns some or all mesh elements into parametric simplices, depending on the options determined by the user. The shape of the parametric simplices is furthermore uniquely determined by the value of `coords` at the Lagrange nodes. There are interface routines `get_lagrange_coords()`, `copy_lagrange_coords()` and `get_lagrange_touched_edges()` to give an application access to the coordinate data, see below in Sections 3.8.1-3.8.4.

Note that on curved elements the ordinary routines to convert between barycentric coordinates and Cartesian coordinates, or to compute their derivatives (see Section 4.1), may no longer be used. Instead, the corresponding hooks in the `PARAMETRIC`-structure described below have to be called. It may be convenient in this case to use calls to the per-element quadrature caches (see Section 4.2.6). An exception is the case of affine-linear "parametric" meshes, or the case of affine-linear mesh elements of only partially parametric meshes: there the standard routines described in Section 4.1 may still be used.

We start with a more detailed description of how to use "standard" piece-wise polynomial parameterizations and continue with the description of the general interface in Section Section 3.8.2 further below.

3.8.1 Piece-wise polynomial parametric meshes

The following functions are available to access and manipulate meshes with "standard" piece-wise polynomial parameterizations:

```
typedef enum param_strategy {
  PARAM_ALL = 0,
  PARAM_CURVED_CHILDS = 1,
  PARAM_STRAIGHT_CHILDS = 2
} PARAM_STRATEGY;
#define PARAM_PERIODIC_COORDS 0x04

void use_lagrange_parametric(MESH *mesh, int degree,
                             NODE_PROJECTION *n_proj, FLAGS flags);
DOF_REAL_D_VEC *get_lagrange_coords(MESH *mesh);
DOF_UCHAR_VEC *get_lagrange_touched_edges(MESH *mesh);
void copy_lagrange_coords(MESH *mesh, DOF_REAL_D_VEC *coords, bool to_mesh);
```

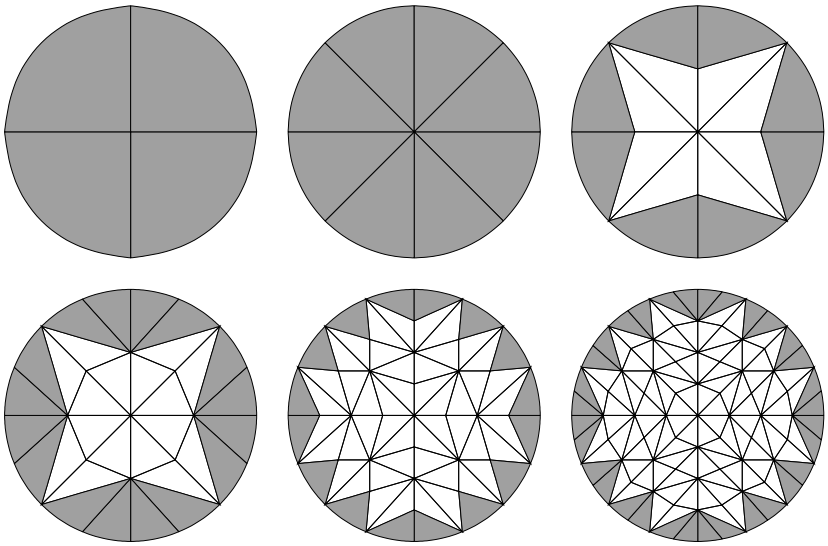


Figure 3.8: Successive refinements of the triangulation of a disc with `strategy == PARAM_STRAIGHT_CHILDS`. Parametric simplices are shaded in gray.

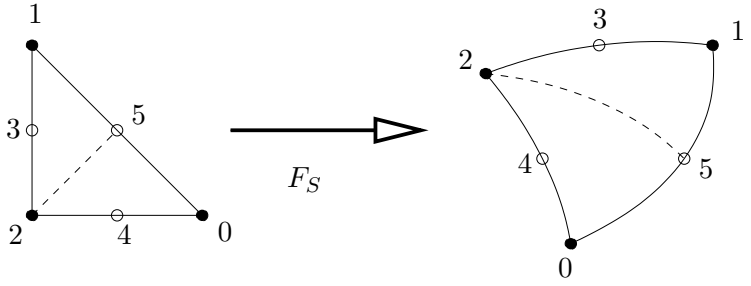


Figure 3.9: Mapping of the standard simplex under a quadratic transformation F_S with standard numbering of the local Lagrange nodes. The curve $\lambda_0 = \lambda_1$ is shown dashed.

3.8.1 Function (`use_lagrange_parametric()`).

Prototype

```
void use_lagrange_parametric(MESH *mesh, int degree,
                             NODE_PROJECTION *selective, FLAGS
                                 strategy);

typedef enum param_strategy {
  PARAM_ALL = 0, PARAM_CURVED_CHILDS = 1, PARAM_STRAIGHT_CHILDS = 2
} PARAM_STRATEGY;

#define PARAM_PERIODIC_COORDS 0x04
```

Synopsis

```
use_lagrange_parametric(mesh, degree, selective, strategy);
```

Description

Convert the given `mesh` into a parametric mesh. The mesh may already be refined. Parametric simplices will be the image of the reference simplex under a polynomial transformation of specified `degree`. The maximal value of `degree` is limited by the maximal degree of the Lagrange basis functions implemented in ALBERTA (currently 4). Internally a coordinate vector `coords` is allocated within the standard Lagrange finite element space of order `degree` Specifying `1` means that simplices will still be the images of an affine transformation, which is useful for special applications.

The `coords` vector employs special `refine_interpol` and `coarse_restrict` entries to enable the described refinement of curved simplices. Concerning the coarsening of the mesh, all parents of parametric elements are automatically parametric elements themselves. The information describing the shape of children is passed back up to parents in a straight forward fashion.

The function generates a filled `PARAMETRIC` structure and sets the entry `mesh->parametric` to point at it. Only one call of the function is possible per mesh. If the mesh belongs to a submesh-hierarchy, then `use_lagrange_parametric()` must be called on the top-level master mesh. The sub-meshes will then inherit the parametric structure from the top-level master mesh. Sub-meshes are discussed in Section 3.9.

When `use_lagrange_parametric()` is invoked, then this will initiate a mesh-traversal to initialize the coordinate vector `coords` mentioned above. On all curved elements – see the parameter `selective` – the corresponding projection routine will be invoked to project the affine (non-curved) coordinates of the Lagrange nodes to whatever manifold is defined by the projection function. As described in Section 3.2.14 ALBERTA allows for a default projection for the entire element, or for distinct projections attached to the "walls" of the elements.

Parameters

`mesh` The mesh to be equipped with a parametric structure.

`degree` The degree of the parameterization. Currently, the maximum degree is 4, limited only by the maximum degree of the Lagrange basis functions implemented in ALBERTA. ALBERTA takes special care – implementing the algorithm explained in [15] – that higher degree iso-parametric boundary approximation will yield optimal convergence rates.

`selective` Optional, maybe `NULL`. If non-`NULL`, then ALBERTA only treats those elements as curved ones which carry exactly this `NODE_PROJECTION` structure. If `selective == NULL`, then all elements carrying a projection routine (see Section 3.2.14) will be treated as curved elements.

`strategy` The parameter `strategy` splits in two parts: `(strategy & PARAM_STRATEGY_MASK)` determines which newly created simplices are treated as parametric simplices during refinement of the mesh. The remaining flag is `PARAM_PERIODIC_COORDS`. It determines whether the finite element function which holds the coordinate information of the parametric mesh is itself a periodic function. The demo-program `demo/src/4d/ellipt-klein-bottle.c` contains an example application.

The following values are defined for `(strategy & PARAM_STRATEGY_MASK)`.

PARAM_ALL All elements of the mesh will be treated as parametric elements, implying that determinants and Jacobeans will be calculated at all quadrature points during assembly. This is useful e.g. for triangulations of embedded curved manifolds. Please note that during refinement a parent element will be split along the surface defined by the equation $lambda_0 = lambda_1$.

PARAM_CURVED_CHILDS Only those elements of the mesh affected by `n_proj` will be treated as parametric elements. Simplices are split along the surface $lambda_0 = lambda_1$ during mesh refinement. Using `PARAM_CURVED_CHILDS` should be avoided for parameterisations of degree > 2, maybe it should not be used at all.

PARAM_STRAIGHT_CHILDS Only those elements of the mesh affected by `n_proj` will be treated as parametric elements. `PARAM_STRAIGHT_CHILDS` should be used for the approximation of curved boundaries. This keeps the number of curved simplices as small as possible and ALBERTA takes care to position the Lagrange nodes of the parametric elements such that optimal approximation order can be achieved; this is not trivial, see [15].

Examples

See below Example 3.8.5.

3.8.2 Function (`get_lagrange_coords()`).

Prototype

```
DOF_REAL_D_VEC *get_lagrange_coords(MESH *mesh);
```

Synopsis

```
coord_dof_vec = get_lagrange_coords(mesh);
```

Description

Returns the internal `DOF_REAL_D_VEC coords` used to store the coordinates of parametric elements. The user may change entries of this vector by hand, if some care is used if the parametric mesh was initialized with `strategy != PARAM_ALL`. See below the description for `get_lagrange_touched_edges()`.

See also `copy_lagrange_coords()` for a more secure interface to the coordinate information.

Parameters

mesh A `mesh`-structure carrying a parametric structure previously initialized by a call to `use_lagrange_parametric()`, see Section 3.8.1 above.

Return Value

A pointer to the underlying coordinate function, a `DOF_REAL_D_VEC` belonging to a finite element space of the piece-wise polynomial degree as specified by the `degree` parameter passed to `use_lagrange_parametric()`.

3.8.3 Function (`copy_lagrange_coords()`).

Prototype

```
typedef enum param_copy_direction {
  COPY_FROM_MESH = false,
  COPY_TO_MESH   = true
} PARAM_COPY_DIRECTION;

void copy_lagrange_coords(MESH *mesh, DOF_REAL_D_VEC *coords, bool
    to_mesh);
```

Synopsis

```
copy_lagrange_coords(mesh, coord_copy, to_mesh);
```

Description

This is the recommended interface to the coordinate information for (Lagrange-) parametric meshes. Only the coordinate *values* are copied; the function also makes sure that affine elements remain affine by using linear interpolation between the vertices of a simplex if that simplex has no curved edge. The state of the edges is determined by the `touched_edges` vector returned by `get_lagrange_touched_edges()`, see below. `copy_lagrange_coords()` handles also a case when a mesh has no parametric structure, but uses `EL->new_coord` to store coordinate information for the vertices, see Section 3.2.14. See also `get_lagrange_coords()`.

Parameters

`mesh` A `mesh`-structure carrying a parametric structure previously initialized by a call to `use_lagrange_parametric()`, see Section 3.8.1 above.

`coord_copy` A `DOF_REAL_D_VEC`, storage for the coordinate information. Note that `coord_copy` is not itself installed as coordinate vector in the mesh, just the coordinate data is copied to and from `coord_copy`, where the direction of the copy-operation is specified by the parametric `to_mesh`, see below.

`to_mesh` If `true`, then the coordinate data is copied from `coord_copy` to the mesh, otherwise the coordinate function of the mesh is copied to `coord_copy`.

3.8.4 Function (`get_lagrange_touched_edges()`).

Prototype

```
DOF_UCHAR_VEC *get_lagrange_touched_edges(MESH *mesh);
```

Synopsis

```
touched_edges_vec = get_lagrange_touched_edges(mesh);
```

Description

Returns the internally used `DOF_UCHAR_VEC touched_edges`. Internally ALBERTA maintains the "projection-state" of all edges. `1` means that the corresponding edge has suffered a projection, `0` means that it is still in the affine linear state. A simplex is treated as parametric simplex if and only if any of its edges has been projected. Otherwise a simplex is not curved. The flags vector is only used if `strategy != PARAM_ALL`, this function will produce a warning and return `NULL` if `strategy == PARAM_ALL`.

When changing the coordinate vector returned by `get_lagrange_coords()` it falls into the responsibility of the application to also change the projection status of the edges.

Parameters

`mesh` A `mesh`-structure carrying a parametric structure previously initialized by a call to `use_lagrange_parametric()`, see Section 3.8.1 above.

Return Value

A pointer to a `DOF_SCHAR_VEC`, with one DOF per edge, indicating whether the respective edge is curved or not, with `touched_edges->vec[dof] == true` meaning the edge is curved and `touched_edges->vec[dof] == false` meaning the edge is *not* curved.

3.8.5 Example (Isoparametric elements for the unit ball)**.** We turn again to the triangulation of the unit ball treated in Example 3.2.7.

```
static void ball_proj_func(REAL_D x,
                           const EL_INFO *el_info, const REAL_B lambda)
{
  SCAL_DOW(1.0/NORM_DOW(x), x);
}

static NODE_PROJECTION ball_proj = {ball_proj_func};

static NODE_PROJECTION *init_node_proj(MESH *mesh, MACRO_EL *mel, int c)
{
  if(c > 0 && !mel->neigh[c-1])
    return &ball_proj;
  else
    return NULL;
}

int main()
{
```

```
  MESH           *mesh;
  const BAS_FCTS *bas_fcts;
  const FE_SPACE *fe_space;
  MACRO_DATA     *data;

  ...

  data = read_macro("ball.amc");
  mesh = GET_MESH(MESH_DIM, "ALBERTA mesh", data,
  init_node_proj, NULL /* init_wall_trafos */);
  free_macro_data(data);

  bas_fcts = get_lagrange(mesh->dim, /* degree == */ 3);
  use_lagrange_parametric(mesh, 3, NULL, PARAM_STRAIGHT_CHILDS);

  ...
}
```

ALBERTA compares the node-projections of all elements with the value of `&ball_proj`, in our example only the boundary faces will have produce a match. Since `strategy == PARAM_STRAIGHT_CHILDS`, ALBERTA will only use parametric elements in a narrow boundary layer, see Figure 3.8.

3.8.2 The `PARAMETRIC` structure

A parametric mesh is described by the structure `PARAMETRIC`. The structure is a collection of function pointers – "methods" – which define the parameterisation. The piecewise polynomial parameterisations predefined in ALBERTA work in arbitrary co-dimension, see Section 3.8.1 above.

```
typedef struct parametric       PARAMETRIC;

struct parametric
{
  char *name;
  bool not_all;
  bool use_reference_mesh;
  bool (*init_element)(const EL_INFO *el_info, const PARAMETRIC *parametric);

  void (*coord_to_world)(const EL_INFO *info, const QUAD *quad,
                         int n, const REAL_B lambda[], REAL_D *world);
  void (*world_to_coord)(const EL_INFO *info, int n,
 const REAL_D world[],
 REAL_B lambda[], int *k);
  void (*det)(const EL_INFO *info, const QUAD *quad,
              int n, const REAL_B lambda[], REAL dets[]);
  void (*grd_lambda)(const EL_INFO *info,  const QUAD *quad,
                     int n, const REAL_B lambda[],
                     REAL_BD Lambda[], REAL_BDD DLambda[], REAL dets[]);
  void (*grd_world)(const EL_INFO *info,  const QUAD *quad,
    int n, const REAL_B lambda[],
    REAL_BD grd_Xtr[], REAL_BDB D2_Xtr[], REAL_BDBB D3_Xtr[]);
```

```
    void (*wall_normal)(const EL_INFO *el_info, int wall,
        const QUAD *wall_quad,
        int n, const REAL_B lambda[],
        REAL_D nu[], REAL_DB grd_nu[], REAL_DBB D2_nu[],
        REAL dets[]);
    void (*inherit_parametric)(MESH *slave);
    void (*unchain_parametric)(MESH *slave);
    void *data;
};
```

Description:

name a textual description of the parametric structure, intended as debugging aid.

not_all , if nonzero, signifies that not all of the mesh elements are to be parametric (curved) simplices. This entry must not be changed by the application program.

use_reference_mesh , if set, means that certain routines should use the reference triangulation consisting of standard simplices instead of the parametric mesh, see the description further below. Is set to `false` by default.

init_element(el_info, parametric) This is a per-element initialiser which must be called for each `el_info` during a mesh traversal before calling any other function hook of the `PARAMETRIC` structure. The argument `parametric` must point to the `PARAMETRIC` structure itself.

A specific implementation of a parametric mesh should use the `init_element()`-hook to perform all necessary initialisations needed to define the transformation from the reference element to the given mesh element. The return value should be `true` if the given element indeed is curved, and `false` if it is just an affine image of the reference element. In the latter case `init_element(el_info, ...)` is supposed to fill `el_info->coord` with the current element's coordinate information – despite the fact that the `el_info` argument carries the `const` attribute. This way the normal per-element functions can be used (e.g. `el_det()`, `el_grd_lambda()` etc.) instead of the parametric replacements defined in the `PARAMETRIC` structure. This simplifies the program flow (and source code) for applications using only partially parametric meshes a lot.

coord_to_world(el_info, quad, n, lambda, world) Implements the function F_S itself. Given an element `el_info`, a vector of barycentric coordinates `lambda` of length `n`, this function writes the corresponding vector of length `n` of world coordinates into `world`. Using this function on multiple sets of coordinates at once may be more efficient than repeatedly calling this function. If the `quad` attribute is not `NULL`, then `quad->n_points` and `quad->lambda` instead of `n` and `lambda`. Additionally, a specific parametric implementation may handle the case `quad != NULL` more efficiently by using caching `QUAD_FAST` quadratures and the like.

world_to_coord() This entry replaces the standard `world_to_coord()` function available for standard simplices. It represents the inverse F_S^{-1}. Currently, there is only a partial implementation available, which may or may not work in the context of iso-parametric boundary approximation.

det(el_info, quad, n, lambda, dets) This function computes $|\det DF_S(\hat{x}(\lambda))|$ which is required for numerical integration, see Remark 1.4.3. The barycentric coordinates are again passed as an array `lambda` of length `n`. The absolute value of the determinant at each λ is written into the array `dets`. Since this routine is mostly used for numerical

integration the user may pass a pointer quad to a quadrature structure instead of lambda. The function will then calculate the determinants at all quadrature nodes of the given numerical quadrature. Additionally, a specific parametric implementation may handle the case quad != NULL more efficiently by using caching QUAD_FAST quadratures and the like. See Section 4.2 for details on using numerical quadrature routines and structures.

grd_lambda(el_info, quad, n, lambda, Lambda, DLambda, dets) This routine is similar to the entry dets above. It additionally fills the array Lambda with the values of the derivative Λ_S of the barycentric coordinates defined in Section 1.4.3.1. Optionally, grd_lambda() also computes the second derivatives of the barycentric coordinates. The second derivatives of the barycentric coordinates are necessary to compute the second derivatives of finite element functions on curved simplices, e.g. for the implementation of residual error estimators. The arguments DLambda and dets may be NULL.

grd_world(el_info, quad, n, lambda, grd_Xtr, D2_Xtr, D3_Xtr) Compute the derivatives of the Cartesian coordinates with respect to the barycentric coordinates. The arguments D2_Xtr and D3_Xtr may be NULL, in which case the quantities are simply not computed. The tr-suffix stands for "transposed", meaning that actually the transposed of the Jacobians is computed. This way, in the affine linear case grd_Xtr is just the matrix formed by the vertex coordinates as rows.

wall_normal(el_info, wall, quad, n, lambda, nu, grd_nu, D2_nu, dets) This function hook is the parametric replacement for library function get_wall_normal(). Again, quad->lambda and quad->n_points is used instead of lambda and n if quad != NULL. quad must be a co-dimension 1 quadrature as returned by get_wall_quad() or get_bndry_quad(). Either of the arguments nu, grd_nu, D2_nu or dets may be NULL; otherwise normals stores the outer unit normal field of the face opposite of the vertex with local number wall and dets stores the values of the surface element. The derivatives of the normal-field are, for instance, needed for vector-valued basis functions like face- or edge-bubbles ("wall-bubbles"). To this aim the outer normal field is extended into the interior of an element by setting it constant on the coordinate lines defined by the barycentric coordinates on the reference element.

inherit_parametric(slave), unchain_parametric(slave)
inherit_parametric() is used by get_submesh(), unchain_parametric() is used by unchain_submesh(). An application which defines its own PARAMETRIC structure can set both pointers to NULL if the sub-mesh feature is not needed.

data This void * pointer is intended for the purpose of chaining implementation specific information to the PARAMETRIC structure. In a C++ context the function hooks defined in the PARAMETRIC structure could be virtual methods, and implementations would just inherit the PARAMETRIC base-class.

Using the flag FILL_COORDS on a mesh traversal (see Section 3.2.17) would fill the EL_INFO structures with coordinate information of the so-called *reference mesh* based on the original macro triangulation. The reference mesh is what is would be used without a call to use_lagrange_parametric(). This reference mesh is normally hidden from the application unless specifically requested by setting the entry PARAMETRIC->use_reference_mesh to true. Furthermore, the mesh traversal routines ignore the FILL_COORDS flag unless use_reference_mesh is true. However, special applications may profit from accessing the reference mesh. On the other hand, most ALBERTA routines, e. g. routines to evaluate

derivatives of basis functions, will automatically use the parametric mesh structure when present.

The function pointers `PARAMETRIC->coord_to_world`, `PARAMETRIC->world_to_coord`, `PARAMETRIC->det`, `PARAMETRIC->grd_lambda`, `PARAMETRIC->wall_normal` should be used instead of the standard routines for standard simplicial triangulations

- `world_to_coord()`
- `coord_to_world()`
- `el_det()`
- `el_volume()`
- `el_grd_lambda()`
- `get_wall_normal()`

described in detail in Section 4.1. The exception are affine elements on only partially parametric meshes: if `PARAMETRIC->init_element()` returns `false` then the standard routines may be used instead of the function hooks of the `PARAMETRIC` structure. The same holds when using a "parametric" mesh of piece-wise polynomial degree through the `use_lagrange_parametric()` call, simply because this implementation "fakes" a partially parametric mesh which is non-curved on all elements. The use of the standard routines in the affine-linear context can simplify application programs quite a bit.

If `ALBERTA_DEBUG==1` and `use_reference_mesh == false` then using the standard library routines on parametric simplices will exit with an error message. This is a safety measure to prevent accidental misuse.

3.8.6 Example (Use of a parametric mesh)**.** This example shows how to write a routine which performs a global interpolation of a given function onto a finite element space. This is a much simplified version of the `interpol()`-implementation which can be found in `alberta/src/Common/eval.c` (path relative to the top-level directory of the source distribution of ALBERTA). Compare also with Section 4.7.8. The simplifications mostly concern the missing support for direct sums of finite element spaces, but as this is a scalar-only example, the restriction does not seem to be too severe.

The function `interpol_simple()` defined here takes a pointer to an application defined function `REAL (*f)(const REAL_D arg)`, and a `DOF_REAL_VEC` and loops over all mesh-elements, calling the local interpolation routines in turn on all elements. We assume here that the evaluation of `f()` is extremely costly, so we are careful not to evaluate `f()` too often. There are two helper-function, `inter_fct_loc()` and `inter_fct_loc_param()`, which are used as arguments to the actual call to the `bfcts->interpol()` hook. Note that the code uses the non-parametric version if either `mesh->parametric` is `NULL`, or if `mesh->parametric->init_element()` returns `false`.

The example also shows the use of another type of per-element initializers: basis functions may also carry such a function-hook, refer to Section 3.11 for a detailed description.

```
static
REAL inter_fct_loc(const EL_INFO *el_info, const QUAD *quad, int iq,
                   void *ud)
{
```

```
  FCT_AT_X fct = *(FCT_AT_X *)ud;
  REAL_D world;

  coord_to_world(el_info, quad->lambda[iq], world);

  return fct(world);
}

static
REAL inter_fct_loc_param(const EL_INFO *el_info, const QUAD *quad, int iq,
                         void *ud)
{
  const PARAMETRIC *parametric = el_info->mesh->parametric;
  FCT_AT_X fct = *(FCT_AT_X *)ud;
  REAL_D world;

  parametric->coord_to_world(el_info, NULL, 1, quad->lambda + iq, &world);

  return fct(world);
}

void interpol_simple(DOF_REAL_VEC *dv, FCT_AT_X f)
{
  /* Some abbreviations ... */
  const FE_SPACE    *fe_space = dv->fe_space;
  const BAS_FCTS    *bfcts    = fe_space->bas_fcts;
  const DOF_ADMIN   *admin    = fe_space->admin;
  MESH              *mesh     = fe_space->mesh;
  const PARAMETRIC  *param    = mesh->parametric;
  EL_REAL_VEC *vec_loc;
  bool         is_param;
  FLAGS        fill_flags;
  int          indices[bfcts->n_bas_fcts_max];
  DOF          dofs[bfcts->n_bas_fcts_max];

  /* Initialize each component of vec to HUGE_VAL, misusing it as
   * flag-argument
   */
  FOR_ALL_DOFS(admin, dv->vec[dof] = HUGE_VAL);

  /* Get an element vector to store the result of the interpolation in */
  vec_loc = get_el_real_vec(bfcts);

  /* Basis functions may need special fill-flags */
  fill_flags = FILL_COORDS|bfcts->fill_flags;
  TRAVERSE_FIRST(mesh, -1, CALL_LEAF_EL|fill_flags) {
    int i, n_indices;
    REAL val;

    /* Basis-functions may need a per-element initialization */
    if (INIT_ELEMENT(el_info, bfcts) == INIT_EL_TAG_NULL) {
      continue;
    }

    /* Call the per-element initializer of mesh->parametric(), if needed */
    is_param = param != NULL && param->init_element(el_info, param);
```

```
    /* Determine which of the local coefficients need to be computed */
    GET_DOF_INDICES(bfcts, el_info->el, admin, dofs);
    for (i = 0, n_indices = 0; i < bfcts->n_bas_fcts; i++) {
      if ((val = dv->vec[dofs[i]]) == HUGE_VAL) {
        indices[n_indices++] = i;
      } else {
        /* "partial" interpolation may need information about the
         * omitted DOFs nevertheless.
         */
        vec_loc->vec[i] = val;
      }
    }

    /* Do the actual interpolation. The parametric version could be
     * handled more efficiently if n_indices == n_bas_fcts; in this
     * case we would only need a single call to
     * param->coord_to_world(). Implementing such (and other
     * optimizations) is left to the reader as an exercise).
     */
    if (n_indices == bfcts->n_bas_fcts) {
      /* Interpolation for all DOFs. The parametric version could be
       * handled more efficiently in this case: we would only need a
       * single call to param->coord_to_world(). Implementing such
       * (and other * optimizations) is left to the reader as an
       * exercise).
       */
      INTERPOL(bfcts, vec_loc, el_info, -1, -1, NULL,
               is_param ? inter_fct_loc_param : inter_fct_loc, &f);
      /* Store the computed values in the global DOF-vector, no need
       * for the indices indirection
       */
      for (i = 0; i < bfcts->n_bas_fcts; i++) {
        dv->vec[dofs[i]] = vec_loc->vec[i];
      }
    } else {
      /* partial interpolation */

      INTERPOL(bfcts, vec_loc, el_info, -1, n_indices, indices,
               is_param ? inter_fct_loc_param : inter_fct_loc, &f);
      /* Store the computed values in the global DOF-vector. Note that
       * BOTH, the global and the local coefficient vector, are
       * accessed indirectly over the indices array.
       */
      for (i = 0; i < n_indices; i++) {
        dv->vec[dofs[indices[i]]] = vec_loc->vec[indices[i]];
      }
    }
  } TRAVERSE_NEXT();

  free_el_real_vec(vec_loc); /* Cleanup after ourselves */

  if (INIT_ELEMENT_NEEDED(bfcts)) {
    /* We possibly did not ran over all elementse, initialize any
     * left-over DOFs to 0.0.
     */
    FOR_ALL_DOFS(admin,
                 if (dv->vec[dof] == HUGE_VAL) {
```

```
      dv->vec[dof] = 0.0;
    });
  }
}
```

3.9 Implementation of submeshes

The concepts and motivations behind submeshes in ALBERTA were already introduced in Section 1.6. Shortly, a submesh or slave mesh (maybe better "trace-mesh") of a given d-dimensional master mesh is a collection of certain $d-1$-dimensional subsimplices that should be refined and coarsened in a conforming way to the master mesh. For parametric master meshes, all submeshes should also be parametric.

The philosophy of submeshes is that their use should not involve major changes of data structures nor excessive overhead in memory or CPU time as a price for their features. We first describe how to allocate and use submeshes. The ideas of how refining and coarsening work for submeshes is described later.

3.9.1 Allocating submeshes

Submeshes are ALBERTA`MESH` objects with some special properties. The user defines a submesh by selecting certain subsimplices of macro elements. The mechanism uses a callback method similar to the case of node projections, refer Section 3.2.14.

```
MESH *get_submesh(MESH *, const char *,
                  int (*)(MESH *, MACRO_EL *, int, void *), void *);
```

`get_submesh(master, name, binding_method, data)` allocates a submesh with the identifier `name` of the given `master`. The binding method is a callback function which is called by ALBERTA for each macro element and each vertex/edge/face in 1d/2d/3d.

Given the `master` mesh, a macro element `mel`, a vertex/edge/face `face`, and arbitrary user `data`, this function should return `true` if the subface is to be part of the submesh and `false` otherwise. An example is shown below. The argument `data` is passed to the callback and may contain arbitrary user data.

Calling this function will return the complete submesh. More than one submesh may be defined. If the master mesh is already refined, then the submesh will automatically be refined to maintain the conformity property (1.27) on page 41. If the master mesh used projection of nodes, then the node projection is inherited by the slave mesh and automatically initialized in such a way that all submesh vertices undergo the same projection as the master vertices.

If the master mesh is a parametric mesh (or is later defined to be one), then the parametric structure is inherited in a straight forward manner to the submesh, a mechanism which is only implemented for `use_lagrange_parametric`. This implies that the submesh elements will also be described by element transformations of the same polynomial degree as for the master mesh, and that the shape of submesh elements matches the shape of curved master mash subsimplices.

The numbering of vertices on the macro triangulation of the submesh is done in such a way as to always guarantee matching refinement edges of submesh and master mesh. Furthermore, the orientation of the submesh elements for 2d submeshes follows a right hand rule for the outward pointing unit normal of the master macro element, see Figure Figure 3.10.

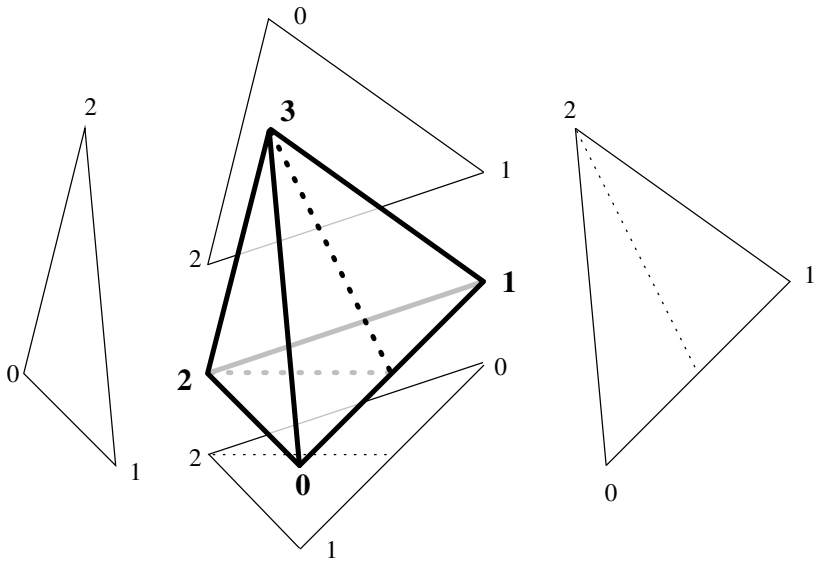


Figure 3.10: Local vertex numbering of the 2d subsimplices of a 3d element of type 0 and positive orientation. The 2d submesh numbering depends on type and orientation of the master elements.

The connection of submesh and master mesh is described internally using two special `DOF_PTR_VEC`s. One of these, called `master_binding`, is based on the submesh and contains pointers from slave elements into master elements. To be precise, each `CENTER` DOF of this vector is a pointer to the master `EL` structure describing the element along which the submesh element lies.

The second vector, called `slave_binding`, is based on the master mesh and points in the opposite direction. It maps `VERTEX/EDGE/FACE` DOFs of the master element to the `EL` structures of the slaves. If no slave element lies along the `VERTEX/EDGE/FACE` then the pointer is `NULL`. Figure Figure 3.11 illustrates this.

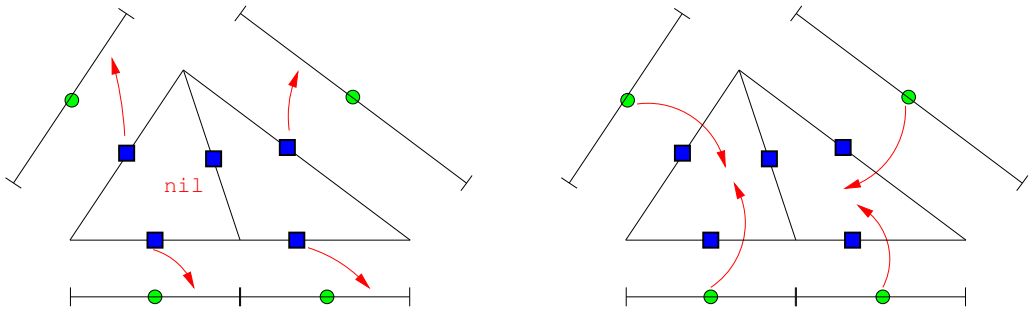


Figure 3.11: 2d master triangle with submesh intervals on all sides. Left: vector `slave_binding` connecting `EDGE` DOFs to slave elements. Right: vector `master_binding` connecting `CENTER` DOFs to master elements.

Both vectors have their `refine_interpol` and `coarse_restrict` entries set to special internal routines. These routines automatically take care of updating the submesh during refinement and coarsening, which is useful since mesh changes are most easily done simulta-

neously on refinement or coarsening patches. This is described in more detail below.

Submeshes can be disconnected from the master mesh using the function

```
void unchain_submesh(MESH *slave);
```

This function deletes the connection between master mesh and submesh. It does *not* delete the submesh. After doing this, the submesh and master mesh are entirely independent and separate MESH objects for ALBERTA and may be refined and coarsened independently.

3.9.2 Routines for submeshes

The following tools are available for submeshes:

```
MESH *read_submesh(MESH *master,
                   const char *slave_filename,
                   int (*binding_method)(MESH *master, MACRO_EL *el,
                                         int face, void *data),
                   NODE_PROJECTION *(*)(MESH *, MACRO_EL *, int),
                   void *data);
MESH *read_submesh_xdr(MESH *master,
                       const char *slave_filename,
                       int (*binding_method)(MESH *master, MACRO_EL *el,
                                             int face, void *data),
                       NODE_PROJECTION *(*)(MESH *, MACRO_EL *, int),
                       void *data);
MESH *get_bndry_submesh(MESH *master, const char *name);
MESH *get_bndry_submesh_by_type(MESH *master, const char *name,
                                BNDRY_TYPE type);
MESH *get_bndry_submesh_by_segment(MESH *master, const char *name,
                                   BNDRY_FLAGS segment);
MESH *read_bndry_submesh(MESH *master, const char *slave_filename);
MESH *read_bndry_submesh_xdr(MESH *master, const char *slave_filename);
MESH *read_bndry_submesh_by_type(MESH *master,
 const char *slave_filename, BNDRY_TYPE type);
MESH *read_bndry_submesh_by_type_xdr(MESH *master,
     const char *slave_filename,
                                     BNDRY_TYPE type);
MESH *read_bndry_submesh_by_segment(MESH *master,
    const char *slave_filename,
                                    BNDRY_FLAGS segment);
MESH *read_bndry_submesh_by_segment_xdr(MESH *master,
       const char *slave_filename,
                                       BNDRY_FLAGS segment);

void get_slave_dof_mapping(const FE_SPACE *m_fe_space, DOF_INT_VEC *s_map);
MESH *get_master(MESH *slave);
const DOF *get_master_dof_indices(const EL_INFO *s_el_info,
  const FE_SPACE *m_fe_space,
  DOF *result);
void trace_dof_real_vec(DOF_REAL_VEC *svec, const DOF_REAL_VEC *mvec);
void trace_dof_real_d_vec(DOF_REAL_D_VEC *svec, const DOF_REAL_D_VEC *mvec);
void trace_dof_int_vec(DOF_INT_VEC *svec, const DOF_INT_VEC *mvec);
void trace_dof_dof_vec(DOF_DOF_VEC *svec, const DOF_DOF_VEC *mvec);
void trace_int_dof_vec(DOF_DOF_VEC *svec, const DOF_DOF_VEC *mvec);
```

```
void trace_dof_uchar_vec(DOF_UCHAR_VEC *svec, const DOF_UCHAR_VEC *mvec);
void trace_dof_schar_vec(DOF_SCHAR_VEC *svec, const DOF_UCHAR_VEC *mvec);
void trace_dof_ptr_vec(DOF_PTR_VEC *svec, const DOF_PTR_VEC *mvec);
void update_master_matrix(DOF_MATRIX *m_dof_matrix,
  const EL_MATRIX_INFO *s_minfo);
void update_master_real_vec(DOF_REAL_VEC *m_drv,
                            const EL_VEC_INFO *s_vec_info);
void update_master_real_d_vec(DOF_REAL_D_VEC *m_drdv,
      const EL_VEC_D_INFO *s_vec_info);
```

3.9.1 Compatibility Note. *The functionality of the function* `get_master_el()` *has been shifted to the* `EL_INFO` *structure; information about the "master"-element is computed during mesh-traversal if requested by the* `FILL_MASTER_INFO` *and* `FILL_MASTER_NEIGH` *fill-flags (see also Section 3.2.17).*

Description of the individual functions:

`read_submesh(master,filename,binding_method,init_node_proj,data)` This function must be used to read a submesh from disk which was previously saved by `write_mesh()`, see Section 3.3.8. Note that a `write_mesh()` call using the master mesh does *not* store submeshes as well. After this call the submesh is again connected with the master mesh. The `init_node_proj` must be the same function as originally passed to the master mesh. The `binding_method` must also be the same function as used to define the submesh originally. The reason for passing these pointers again is that there is no way to store the C code describing these functions in a file.

`read_submesh_xdr()` Analogous function for submeshes stored by `write_mesh_xdr()`.

`get_bndry_submesh(master, name)` A convenience function, internally `get_submesh()` is called with an appropriate `binding_method` which turns all boundary simplices into a sub-mesh.

`get_bndry_submesh_by_type(master, name, type)` Like the function above, but allows for the specification of a boundary type. See Section 3.2.4.

`get_bndry_submesh_by_segment(master, name, segment)` Like the function above, but allows for the specification of a boundary type bit-mask. See Section 3.2.4.

`read_bndry_submesh(master, filename)`

`read_bndry_submesh_xdr(master, filename)`

`read_bndry_submesh_by_type(master, filename, type)`

`read_bndry_submesh_by_type_xdr(master, filename, type)`

`read_bndry_submesh_by_segment(master, filename, segment)`

`read_bndry_submesh_by_segment_xdr(master, filename, segment)`
Counterparts to `read_submesh()` and `read_submesh_xdr()` to read back sub-meshes generated by `get_bndry_submesh()` and `get_bndry_submesh_by_type()`, respectively.

`get_slave_dof_mapping(m_fe_space, s_map)` Fills the vector `s_map` on the submesh with the corresponding DOF indices of the finite element space `m_fe_space` on the master mesh. This only works if `m_fe_space` and `s_map->fe_space` are Lagrange type spaces of equal degree. The master DOF indices are not updated during mesh changes, hence the use of a `DOF_INT_VEC`, see Section 3.3.2.

`get_master(slave)` returns the master mesh of `slave`.

fill_slave_el_info(slv_el_info, el_info, face, slave_mesh) Fills a EL_INFO element descriptor refering to the slave mesh.

fill_master_el_info(mst_el_info, el_info, face, fill_flags) Fills a EL_INFO element descriptor refering to the slave mesh. fill_flags determines what kind of information is provided.

trace_to_bulk_coords(result, lambda, el_info)

bulk_to_trace_coords(result, lambda, el_info) Given local coordinates on either the master or the trace mesh construct the matching local coordinates for the peer-element. el_info always refers to the lower-dimensional slave-mesh.

get_master_dof_indices(result, s_el_info, m_fe_space) Find the DOFs of m_fe_space – a finite element space belonging to a master mesh – belonging to the s_el_info – an element descriptor for an element of the slave mesh. If result is not NULL, then it is used as storage for the DOF-indices and its address is returned. Otherwise the address of a static storage area is returned which holds the results until it is overwritten on the call to get_master_dof_indices().

get_master_bound(result, s_el_info, m_fe_space) Same for the boundary classification.

trace_<TYPE>_vec(slave_vec, master_vec) Implement discrete trace operators. The vector slave_vec must be based on a submesh of the mesh defining master_vec. The entries of slave_vec are overwritten with values of master_vec along the interface. The finite element spaces of slace_vec and master_vec must be compatible, i.e. slave_vec->fe_space->bas_fcts must be the trace space master_vec->fe_space->bas_fcts->trace_bas_fcts. <TYPE> is one of {dof_real, dof_real_d, dof_int, dof_dof, int_dof, dof_schar, dof_uchar, dof_ptr}, i.e. there is a trace operation for all DOF-vector types.

update_master_matrix(m_dof_matrix, s_minfo)

update_master_real_vec(m_drv, s_vec_info)

update_master_real_d_vec(m_drdv, s_vec_info)
These functions take element-matrix descriptors s_minfo designed for the slave-mesh and update a matrix for the master-mesh. This can, e.g., be used to assemble Robin boundary conditions and the like.

3.9.3 Refinement and coarsening of submeshes

As explained above, submeshes and master meshes are automatically refined and coarsened simultaneously to maintain matching nodes and edges. To guarantee this property we need to be careful in choosing the enumeration of vertices of the submesh. In the most complicated case of a 2d submesh of a 3d master mesh, the numbering of a slave element tied to a given face of a master tetrahedron depends on the master tetrahedron's orientation, type, and the face index. Figure Figure 3.10 demonstrates one given case. The 2d submesh triangles possess the property that they are oriented in a right hand rule with the thumb pointing away from the master element.

Any mesh, whether master mesh or submesh may be refined or coarsened by a call to the routines refine() or coarsen, see Sections 3.4.1 and 3.4.2 respectively. As mentioned before, an entire hierarchy of submeshes from 3d down to 1d is possible.

If a top-level master mesh is to be refined, then the refinement algorithm is carried out as usual. The vector `slave_binding` based on the top-level master mesh has an entry `refine_interpol` set to a special internal routine. This routine is called for each master refinement patch. It creates a corresponding submesh refinement patch for the submesh elements adjoining the master patch. The submesh patch is then refined (in the process calling in turn any `refine_interpol`s of the submesh).

If a submesh is to be refined, ALBERTA first transfers the refinement markers of the submesh elements to the corresponding master elements using `master_binding`. Then `refine()` is called recursively for the master mesh. Once we reach the top-level master mesh we proceed as in the prior paragraph. The submesh refinement markers are reset during the refinement of the master meshes. The diagram of Figure 3.12 describes this process.

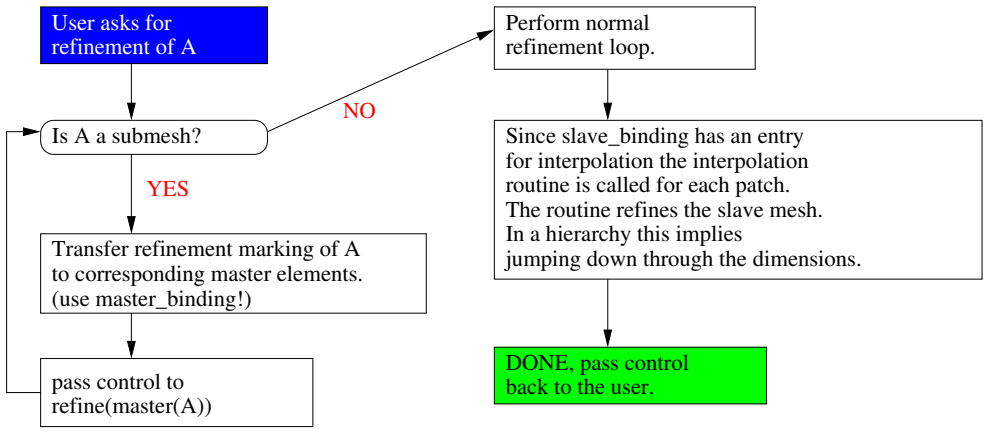


Figure 3.12: Modified refinement algorithm.

Coarsening of master and submeshes works in much the same way. Note that coarsening marks must be transferred to the master mesh to enable any change — this will overwrite refinement marks on the master mesh elements along the interface! Furthermore, the master mesh receives the same value of the coarsening mark as the slave mesh, which may not be enough to guarantee the coarsening if the current refinement edge of a master element does not lie along the interface with the submesh.

Another caveat is that the user should be careful when using other refinement interpolation or coarsening interpolation/restriction routines on the master mesh that perform certain operations using submeshes. The state of the submesh is undefined at the time of calling these routines.

To conclude, submeshes offer advantages in many calculations where information based on surfaces or interfaces is necessary. The price of using a submesh is the additional overhead of one `MESH` structure plus the memory needed to store two `DOF_PTR_VEC`s (and their respective DOF administration). The vector `slave_binding` is based on the master mesh, while the vector `master_binding` is based on the submesh. Both vector require the feature `preserve_coarse_dofs`, see Section 3.4.1.1 for details. Allocating submeshes once the master mesh is strongly refined is to be avoided, since a DOF administration for new element nodes may have to be set up on the fly. This can be expensive in terms of CPU time. The user should code the macro triangulations in a way that the interface defining the submesh is easily accessible.

3.10 Periodic finite element spaces

In ALBERTA, a periodic mesh is thought of (part of) the fundamental domain of crystallographic group. The periodic structure is induced by a set of face-transformation. Given a fundamental domain of a crystallographic group, the face-transformations are the special set of generators of that group which map the given fundamental domain to its neighbour across on of the walls separating it from its neighbour.

3.10.1 Definition of periodic meshes

The most convenient way to define a periodic structure for a mesh is to specify geometric face-transformations through the file which defines the macro-triangulation; this has already been explained in Section 3.2.15. An 2d-example, defining a periodic torus, would look like follows. A corresponding example for 3d can be found in the suite of demo-programs which is shipped with the ALBERTA-package. One thing which might be striking in Example 3.10.1 on the right is that the triangulation seems to be unnecessarily complicated: it is possible to triangulate a square with just two triangles, instead of the 8 elements which are used in this example. The reason is the following: ALBERTA uses the global numbering of the vertex-nodes to compute the relative orientation of neighboring elements. This is needed, e.g., during mesh refinement and coarsening, or when computing integrals over the walls of the elements, assembling jump terms. Now, if the mesh is periodic then the vertex nodes used to orient neighboring elements are actually identified. Therefore, the simplest macro triangulation with only two triangles would have just one vertex-DOF, all vertices would have been identified, making it impossible to orient neighboring elements.

> *Therefore ALBERTA imposes the restriction a face-transformation must not map any vertex to a vertex on the same element.*

3.10.1 Example. A macro triangulation for a topological torus.

```
DIM: 2
DIM_OF_WORLD: 2

number of elements: 8
number of vertices: 9

element vertices:
 4 0 1
 2 4 1
 4 2 5
 8 4 5
 4 8 7
 6 4 7
 4 6 3
 0 4 3

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

number of wall transformations: 2

wall transformations:
# generator #1
 1 0 2
 0 1 0
 0 0 1
# generator #2
 1 0 0
 0 1 2
 0 0 1
```

There is, however, some limited support to cope with coarse macro-triangulations: if it encounters a periodic macro-triangulations which violates this restriction then it tries to resolve the issue by running some steps of global refinement over the mesh in the hope

that the refined meshes fulfill the restriction. It then converts the refined meshes into a macro triangulation and starts over with the refine macro-mesh. Thus, the following macro-triangulation could be used by an application:

3.10.2 Example. Coarse periodic macro triangulation.

```
DIM: 2
DIM_OF_WORLD: 2

number of elements: 2
number of vertices: 4

element vertices:
 2 0 1    0 2 3

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

number of wall transformations: 2

wall transformations:
# generator #1
 1 0 2
 0 1 0
 0 0 1
# generator #2
 1 0 0
 0 1 2
 0 0 1
```

However, the application program will end up with a mesh which is based on a refined mesh which probably looks very similar to the triangulation defined by Example 3.10.1.

There are two other methods to define a periodic structure on a mesh: by specifying combinatoric face-transformations in the macro triangulation, this is explained in Section 3.2.15, or by passing geometric face-transformation to the `GET_MESH()` call, see Section 3.2.13. Similar to the mechanism of initializing node-projections (see Example 3.2.7) it is possible to pass a second routine to the `GET_MESH()` call to initialize face-transformations:

3.10.3 Example. 2d. Initialization of face-transformations through an `init_wall_trafos()`-hook passed to `GET_MESH()`. For this to work the macro-data file has assigned different boundary "street-numbers" to the different periodic boundary segments: type 1 corresponds to a translation in x_1-direction and type 2 to a translation in x_2-direction. The `init_wall_trafos()` function is called with fully-features macro-elements (except for the missing periodic structure, of course). The convention is to return `NULL` if no face-transformation applies, and a pointer to the face-transformation if the boundary-wall with local number `wall` in `mel` belongs to a periodic boundary segment.

```
static AFF_TRAFO *init_wall_trafos(MESH *mesh, MACRO_EL *mel, int wall)
{
  static AFF_TRAFO wall_trafos[DIM_OF_WORLD] = {
    { {{1.0, 0.0},
       {0.0, 1.0}}, {2.0, 0.0} },
    { {{1.0, 0.0},
```

```
        {0.0, 1.0}}, {0.0, 2.0} }
  };
  static AFF_TRAFO inverse_wall_trafos[DIM_OF_WORLD] = {
    { {{1.0, 0.0},
       {0.0, 1.0}}, {-2.0, 0.0} },
    { {{1.0, 0.0},
       {0.0, 1.0}}, {0.0, -2.0} }
  };

  switch (mel->wall_bound[wall]) {
  case 1: /* translation in x[0] direction */
    if (mel->coord[(wall+1) % N_VERTICES(mesh->dim)][0] > 0.0) {
      return &wall_trafos[0];
    } else {
      return &inverse_wall_trafos[0];
    }
  case 2: /* translation in x[1] direction */
    if (mel->coord[(wall+1) % N_VERTICES(mesh->dim)][1] > 0.0) {
      return &wall_trafos[1];
    } else {
      return &inverse_wall_trafos[1];
    }
  }
  return NULL;
}
```

3.10.2 Periodic meshes and finite element spaces

Defining a periodic structure on a mesh only generates a mesh *could* carry periodic finite element spaces. `GET_MESH()` indicates this by setting `MESH.is_periodic` to `true`. Additionally, the following components of the `MESH`-structure are maintained by ALBERTA and are automatically updated during mesh adaptation.

`is_periodic` Set to `true` by `GET_MESH()` if the mesh admits periodic finite element spaces.

`per_n_vertices, per_n_edges, per_n_faces` Number of vertices, edges and faces taking the identification of those sub-simplices on periodic boundary segments into account, i.e. `MESH.n_faces` counts periodic faces twice, `MESH.per_n_faces` counts them only once.

`wall_trafos` If specified by the application this list contains the geometric face-transformations and their inverses. This can be helpful, sometimes an application may have the need to compute the orbit of geometric objects under the action of the underlying crystallographic group. Internally, ALBERTA has the need to compute orbits of vertices and edges when adding new periodic finite element spaces to the mesh.

`n_wall_trafos` Self-explanatory.

Having defined a periodic structure on a mesh, an application must do more to actually define periodic function spaces: it must pass the flag `ADM_PERIODIC` to `get_fe_space()` respectively `get_dof_space()`. Otherwise the returned space will be *non*-periodic. Periodic finite element spaces are implemented by actually identifying degrees of freedom, so – at least for scalar problems or in the context of mere translations – nothing more has to be done to implement periodic boundary conditions. This is exercised by the example `src/Common/ellipt-periodic.c` which can be found in the demo-package.

3.10.4 Example. Allocating periodic and non-periodic finite element space on the same mesh.

```
const BAS_FCTS *bfcts = get_lagrange(dim, degree)
const FE_SPACE *per_fe_space =
  get_fe_space(mesh, "periodic space", bfcts, 1 /* range dimension */,
               ADM_PERIODIC);
const FE_SPACE *std_fe_space =
  get_fe_space(mesh, "periodic space", bfcts, 1 /* range dimension */,
               ADM_FLAGS_DFLT);
```

There are many good reasons to allow non-periodic finite element spaces on a periodic-admissible mesh, some of them are:

- Parts of a specific problem may require periodic boundary conditions, others not.
- In the context of parametric meshes the coordinate functions defining the geometry of the mesh are – of course – non-periodic.
- Vector-valued problems: e.g. for the simulation of fluids the velocity field can in general not be chosen as a vector field consisting of component-wise periodic functions. This is actually only possible in the most simple case were the face-transformations are mere translations. Otherwise the identification of the velocity field across a periodic boundary segment requires first the transformation of the components of the vector field by the face-transformation.

  This implies that in this context the linear systems have to be actively modified, a mere identification of `DOF`s does not suffice.

Above reasoning implies that it is desirable to be able to loop over the mesh ignoring its periodic structure altogether. This can be achieved like demonstrated below:

3.10.5 Example. Non-periodic mesh-traversal on a periodic mesh. The resulting `EL_INFO`-structures are completely unaware of the periodic structure, in particular the periodic neighbors are *not* filled in. This is easily achieved by setting the `FILL_NON_PERIODIC` fill-flag.

```
TRAVERSE_FIRST(mesh, -1,
               CALL_LEAF_EL|FILL_NON_PERIODIC|
               FILL_NEIGH|FILL_MACRO_WALLS) {
  int w;

  for (w = 0; w < N_WALLS(mesh->dim); w++) {
    int mwall = el_info->macro_wall[w];
    if (el_info->neigh[w] == NULL &&
        el_info->macro_el->neigh_vertices[mwall][0] >= 0) {
      MSG("I'm a non-periodic element,
                but my macro-element has periodic boundaries!\n");
    }
  }

} TRAVERSE_NEXT();
```

3.10.3 Element-wise access to periodic data

Data like the face-transformations is stored only on the macro-element level, not in the `EL_INFO`-structure. However, requesting the `FILL_MACRO_WALLS` fill-flag gives an application the link between the wall numbering of the current element and the numbering of walls of the macro-element it is contained in, see also Section 3.2.7 and Example 3.10.6. The following additional information is available through the `MACRO_EL`-structure when the mesh carries a periodic structure:

`np_vertex_bound` The *non*-periodic boundary classification of the vertices, i.e. in ignorance of the periodicity of the mesh.

`np_edge_bound` Same, but for edges.

`neigh_vertices` As explained in Section 3.2.5

`wall_trafo` The geometric face-transformations for each wall. `wall_trafo[wall]` is `NULL` if the corresponding boundary segment is non-periodic, or is an interior wall.

3.10.6 Example. A demonstration of how to access information about periodic boundary conditions during mesh-traversal. Long version:

```
TRAVERSE_FIRST(mesh, -1, CALL_LEAF_EL|FILL_MACRO_WALLS) {
  int w, mwall;

  for (w = 0; w < N_WALLS(mesh->dim); w++) {
    if ((mwall = el_info->macro_wall[w]) < 0) {
      continue; /* interior wall */
    }
    if (el_info->macro_el->wall_trafo[mwall] != NULL) {
      MSG("Hurray, a face transformation!\n");
    }
  }
} TRAVERSE_NEXT();
```

Slightly shorter, using the `wall_trafo()` call:

```
TRAVERSE_FIRST(mesh, -1, CALL_LEAF_EL|FILL_MACRO_WALLS) {
  int w;
  const AFF_TRAFO *face_trafo;

  for (w = 0; w < N_WALLS(mesh->dim); w++) {
    if ((face_trafo = wall_trafo(el_info, w)) != NULL) {
      MSG("Hurray, a face transformation @%p!\n", face_trafo);
    }
  }
} TRAVERSE_NEXT();
```

3.10.4 Periodicity and trace-meshes

In principle, the periodic structure of a mesh is inherited by its trace-meshes. However, this may not make sense in all cases. For instance, if the master-mesh is a topological torus then the attempt to define a trace mesh consisting of all periodic boundaries will fail – or at least that trace-mesh will have no consistent periodic structure. The reason is simple: periodicity is induced by mapping walls to walls with face-transformations. In general this implies that

the orbits of co-dimension 2 and co-dimension 3 face-simplices under the group spanned by the face-transformations contain more than two elements. So if the intersection of the trace-mesh with those orbits also contains more than 2 elements, then the periodic structure on the trace-mesh cannot be well-defined.

So in general a trace-mesh of a periodic master mesh must be perpendicular to the periodic boundary segments of the ambient master-mesh.

3.11 Per-element initializers for quadrature rules and basis function sets

This section is *not* concerned with the per-element initializers contained in the `OPERATOR_INFO`, `BNDRY_OPERATOR_INFO` and `PARAMETRIC` data-structures, they obey other rules and are explained in the respective sections, see Section 4.7.3 and Section 3.8.

3.11.1 Basics

Several data-structure allow for a function hook which is used to perform per-element initialization. This is useful in, e.g., for vector-valued basis functions which depend on the element geometry like face-bubbles, or Raviart-Thomas elements, and in other contexts. Other prominent examples can be quadrature rules in the context of unfitted finite element methods, or for the integration of discontinues functions, where the discontinuity is co-dimension 1 sub-manifold intersecting the mesh, cutting wildly through the element, e.g. for interface problems.

The basic data structures allowing for such initializers are

`BAS_FCTS` (Section 3.5)

`QUAD` (Section 4.2.1)

`WALL_QUAD` (Section 4.2.4)

Naturally, the quadrature caches should derive from any per-element initializers present in the underlying quadrature and basis-function data-structures, so the following data-structures may have per-element initializers as well:

`QUAD_FAST` (Section 4.2.2)

`WALL_QUAD_FAST` (Section 4.2.5)

`Q11_PSI_PHI` (Section 4.7.5)

`Q01_PSI_PHI` (Section 4.7.5)

`Q10_PSI_PHI` (Section 4.7.5)

`Q00_PSI_PHI` (Section 4.7.5)

The derived initializers are assigned during the construction of the quadrature caches, examining the underlying `BAS_FCTS` and `QUAD` structures. The `get_quad_fast()` and `get_wall_quad_fast()` routines, as well as the constructors for the `..._PSI_PHI`-caches take care of this automatically.

The initialization-subroutine is hooked as a function-pointer into the data-structure. The basic definitions are

```
typedef unsigned int INIT_EL_TAG;
typedef INIT_EL_TAG (*INIT_ELEMENT_FCT)(const EL_INFO *el_info, void
    *thisptr);

enum {
  INIT_EL_TAG_NONE = 0, /* invalid tag */
  INIT_EL_TAG_DFLT = 1, /* default case */
  INIT_EL_TAG_NULL = 2  /* something is 0, e.g. no quad-points, basis
                         * functions are identically zero and so on.
                         */
};

#define INIT_ELEMENT(el_info, object)                                    \
  (if (object->init_element != NULL)                                     \
   ? object->init_element(el_info, (void *)object) : INIT_EL_TAG_DFLT)

#define INIT_OBJECT(object) (void)INIT_ELEMENT(NULL, object)
```

The idea behind the "tag" definitions is that an object possibly may have a default-state on the majority of mesh-elements, possibly may evaluate to an empty object on many elements (e.g. the number of basis functions is zero, or the quadrature rule has no points), and has a special state on some of the elements. This is, for example, the case when defining a quadrature rule in the context of unfitted finite elements, where most mesh-elements belong to the interior of the domain, may are located outside the domain of computation, and some are actually intersected by the boundary. To handle such cases efficiently, `INIT_ELEMENT(el_info, object)` must follow these conventions:

- INIT_ELEMENT(el_info, object) evaluates to `INIT_EL_TAG_DFLT` when no per-element initializer is present.

- An `init_element()` method *must* allow a `NULL` pointer for the `el_info` argument. If called with `el_info == NULL` the `init_element()` method must restore its default state. The "default case" is what the implementation defines as default; for performance reasons the default case should be the one which applies to the majority of mesh elements. The convenience-macro `INIT_OBJECT(object)` just forwards to `INIT_ELEMENT(NULL, object)`.

- The return value of the `init_element()` method must be `INIT_EL_TAG_DFLT` for the default case.

- The return value of the `init_element()` method must be `INIT_EL_TAG_NULL` for the `NULL` case, meaning, e.g., the number of basis functions is zero, or the number of quadrature points is zero. The application can assume that in the `NULL` case the structure does not contain any real data.

- In all other cases the return value is a tag which is used to efficiently cache values of intermediate computations, e.g. the values of basis functions at quadrature points. This tag should be locally unique, meaning that consecutive invocations of `init_element()` should return different tags for different simplexes. This can be used for optimizations: if the tag returned by an `init_element()` routine does not change, then the calling function may assume that the underlying object has not changed.

The meaning of the reserved tag-names is

INIT_EL_TAG_NONE An invalid tag that does not correspond to any state of the underlying object. If an object is in this state, then the data contained in the object is undefined and its per-element initializer must be called before accessing any components of the object (with the exception of the `init_element()`-hook itself, of course.

INIT_EL_TAG_NULL The object is in the `NULL`-state. An application can assume that in this state a quadrature rule contains no points or that the local basis function set is empty.

INIT_EL_TAG_DFLT The object is in a default state. What that means is implementation dependent. A sensible implementation should choose as default state the state it attains on the majority of mesh elements. It can be that an objects does not have any sensible default state, for example if it is a local basis functions set depending on the element geometry.

...any other number defines a unique state. The implementation underlying a quadrature rule or a local basis function set should make sure that repeated calls to the `init_element()`-hook return the same tag. This can then be used by applications to cache derived data across sub-routine calls, using the tag-value to invalidate the caches.

3.11.2 Per-element initializers and vector-valued basis functions

There is one thing special for vector-valued geometry-dependent basis function sets. Following the rules developed above, the `init_element()`-hook for such functions would have to return a unique tag on each element, thus invalidating the quadrature caches on each new element. Because this is inefficient, ALBERTA's implementation factors vector-valued basis-functions into a geometry dependent vector-valued part and a geometry-independent scalar part. Vector-valued `BAS_FCTS` instances carrying a `init_element()` method should therefore return `INIT_EL_TAG_DFLT` if just the vector-valued factor has changed, but the scalar factor is not affected by the `init_element()`-method. This way the quadratures caches defined by `QUAD_FAST` and the `..._PSI_PHI`-caches are not invalidated, which helps to keep the assembling of linear systems efficient. Compare also the remarks in Section 3.5.2, dealing with vector-valued basis functions in general.

3.11.3 Tag management

To aid the implementation of the scheme described above there are some support structures and macros concerning the management of the tags returned by the `init_element()` hooks:

```
/* Tag context. */
typedef struct init_el_tag_ctx {
  INIT_EL_TAG  tag;
  unsigned int cnt;
} INIT_EL_TAG_CTX;

#define INIT_EL_TAG_CTX_INIT(ctx)                \
  {                                              \
    (ctx)->tag = INIT_EL_TAG_DFLT;               \
    (ctx)->cnt = 0;                              \
  }

/* Generate a new unique tag != NULL & DFLT */
```

```
#define INIT_EL_TAG_CTX_UNIQ(ctx)                                    \
  {                                                                  \
    (ctx)->tag = INIT_EL_TAG_NULL + (++((ctx)->cnt));                \
    if ((ctx)->tag == INIT_EL_TAG_NONE) {                            \
      (ctx)->cnt = 1;                                                \
      (ctx)->tag = INIT_EL_TAG_NULL + 1;                             \
    }                                                                \
  }
#define INIT_EL_TAG_CTX_NULL(ctx) (ctx)->tag = INIT_EL_TAG_NULL
#define INIT_EL_TAG_CTX_DFLT(ctx) (ctx)->tag = INIT_EL_TAG_DFLT
#define INIT_EL_TAG_CTX_TAG(ctx)  (ctx)->tag

#define INIT_ELEMENT_DECL                                            \
  INIT_ELEMENT_FCT init_element;                                     \
  FLAGS fill_flags;                                                  \
  INIT_EL_TAG_CTX tag_ctx

#define INIT_ELEMENT_INITIALIZER(init_el, flags)                     \
  (init_el), (flags), { INIT_EL_TAG_DFLT, 0 }
```

The meaning of the components of `INIT_EL_TAG_CTX`:

`tag` The current tag value.

`cnt` A counter, used to generate locally unique per-element tags.

An application *must not* access the two components of the tag-context directly, but has direct the access to the tag and the counter through the access macros defined above. Some for the implementation an object carrying such a per-element initializers. Obeying this rule ensures compatibility with future version of ALBERTA, hopefully. The meaning of the tag-management-macros is as follows:

`INIT_EL_TAG_CTX_INIT(ctx)` Initialize the points `ctx`, pointing to an existing tag-context.

`INIT_EL_TAG_CTX_UNIQ(ctx)` Generate a new uniquq tag by incrementing `ctx->cnt`. The macro takes care of jumping over the reserved tags `INIT_EL_TAG_NONE`, `INIT_EL_TAG_CTX_NULL` and `INIT_EL_TAG_DFLT`, thus protecting the generated tags against wrap-around.

`INIT_EL_TAG_CTX_NULL(ctx)` Set `ctx->tag` to `INIT_EL_TAG_NULL`.

`INIT_EL_TAG_CTX_DFLT(ctx)` Set `ctx->tag` to `INIT_EL_TAG_DFLT`.

`INIT_EL_TAG_CTX_TAG(ctx)` Return the current tag.

3.11.4 Mesh-traversal and per-element initializers

Objects which depend on the mesh-element they are living on often may require special information, for instance about the geometry of the mesh-element. Because this information is only selectively available during a mesh-traversal – ALBERTA fills most information in a root-to-leaf manner, compare Section 3.2.17 – there is danger that the current `EL_INFO` structure does not carry enough information in order for the `init_element()`-method to do its work. To cope with this problem an object with such an initializer should also record its needs concerning the available information during mesh-traversal in an additional component `FLAGS fill_flag`. It is advisable that implementations for element-dependent objects make use of the following definitions from `alberta.h`:

```
#define INIT_ELEMENT_DECL                                \
  INIT_ELEMENT_FCT init_element;                         \
  FLAGS fill_flags;                                      \
  INIT_EL_TAG_CTX tag_ctx

#define INIT_ELEMENT_INITIALIZER(init_el, flags)         \
  (init_el), (flags), { INIT_EL_TAG_DFLT, 0 }
```

The `INIT_ELEMENT_DECL` macro should be inserted in the definition of each structure carrying such an initializer, e.g.

```
struct foobar
{
  ... /* other stuff */
  INIT_ELEMENT_DECL;
  ... /* other stuff */
};
```

The macro `INIT_ELEMENT_INITIALIZER()` can be used during the (static) initialization of such data-structures, e.g.

```
static struct foobar = {
  ... /* other stuff */,
  INIT_ELEMENT_INITIALIZER(FILL_NEIGH|FILL_COORDS, foobar_init),
  ... /* other stuff */
};
```

Compare also the definitions for the data-structures in the source-listings on the pages 223, 145, 232, 227, 233, 280, 284, 282, 286. Example 3.8.6 demonstrates in a half-real world setting how to take care of such fill-flags and per-element initializers of `BAS_FCTS` structures, see there.

Chapter 4

Tools for finite element calculations

4.1 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 4.7, e.g.). Some other applications will need the calculation of barycentric coordinates for given world coordinates (see Section 3.2.17, e.g.). Finally, derivatives of finite element functions on elements involve the Jacobian of the barycentric coordinates (see Section 4.3, e.g.).

In case of a grid with parametric elements, these operations strongly depend on the element parameterization and no general routines can be supplied. For non-parametric simplices, ALBERTA supplies functions to perform these basic tasks:

```
const REAL *coord_to_world(const EL_INFO *, const REAL *, REAL_D);
int world_to_coord(const EL_INFO *, const REAL *, REAL_B);
REAL el_grd_lambda(const EL_INFO *, REAL [N_LAMBDA][DIM_OF_WORLD]);
REAL el_det(const EL_INFO *);
REAL el_volume(const EL_INFO *);
REAL get_wall_normal(const EL_INFO *el_info, int i0, REAL *normal);
```

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 `NULL` 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()`;

`coord_to_world()` needs vertex coordinates information; the flag `FILL_COORDS` has to be set during mesh traversal when calling this routine on elements.

`world_to_coord(el_info, world, lambda)` calculates the barycentric coordinates with respect to the element `el_info->el` of a point with world coordinates `world` and stores them in the vector given by `lambda`. The return value is `-1` when the point is inside the

simplex (or on its boundary), otherwise the index of the barycentric coordinate with largest negative value (between 0 and d);

`world_to_coord()` needs vertex coordinates information; the flag `FILL_COORDS` has to be set during mesh traversal when calling this routine on elements.

Note that – with the exception of the 1d code – this function is only implemented for the co-dimension 0 case, i.e. `mesh->dim` and `DIM_OF_WORLD` have to be equal, otherwise a call to this function will terminate the application with a corresponding error-message.

`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. For $d < n$ the tangential gradient and the value of Gram's determinant are calculated.

`el_grd_lambda()` needs vertex coordinates information; the flag `FILL_COORDS` has to be set during mesh traversal when calling this routine on elements.

`el_det(el_info)` returns the the absolute value of the determinant of the affine linear parameterization's Jacobian, or Gram's determinant for $d < n$.

`el_det()` needs vertex coordinates information; the flag `FILL_COORDS` has to be set during mesh traversal when calling this routine on elements.

`el_volume(el_info)` returns the the volume of the simplex; `el_volume()` needs vertex coordinates information; the flag `FILL_COORDS` has to be set during mesh traversal when calling this routine on elements.

`get_wall_normal(el_info, wall, normal)` compute the outer unit normal of the face opposite to vertex `wall`. The result is stored in `normal`. The return value is the "surface element" of the given face, i.e. Gram's determinant of the transformation to the respective face of the reference element. `normal` may be `NULL`. In the case of non-zero co-dimension `normal` is contained in the sub-space spanned by the edges of the given simplex.

All functions described above also come with a `..._Xd` variant, e.g. `coord_to_world_2d()`. For the case of 0 co-dimension there are also wrapper functions `..._0cd` which call the appropriate `..._Xd` variant with `X == DIM_OF_WORLD`. The `..._Xd` and `..._0cd` variants are very slightly faster because otherwise the dimension of the underlying mesh has to be read out of the mesh structure – e.g. via `el_info->mesh_dim` – and only then the functions branch to the appropriate `..._Xd` variant.

4.2 Data structures for numerical quadrature

For the numerical calculation of general integrals

$$\int_S f(x)\,dx$$

we use quadrature formulas described in 1.4.7. ALBERTA supports numerical integration in zero, one, two, and three dimensions on the standard simplex $\hat{S}$ in barycentric coordinates.

4.2.1 The QUAD data structure

A quadrature formula is described by the following structure, which is defined both as type QUAD and QUADRATURE:

```
extern n_quad_points_max[DIM_MAX+1];
typedef struct quadrature   QUAD;
typedef struct quadrature   QUADRATURE;

struct quadrature
{
  char          *name;
  int           degree;

  int           dim;
  int           codim;
  int           subsplx;

  int           n_points;
  int           n_points_max;
  const REAL_B  *lambda;
  const REAL    *w;

  void          *metadata;

  INIT_ELEMENT_DECL;
};
```

Description:

name Textual description of the quadrature.

degree Quadrature is exact of degree degree.

dim Quadrature for dimension dim. The barycentric co-cordinates of the quadrature points always have dim+1 valid components.

codim Co-dimension; codim is always 0 for quadratures returned by get_quadrature(), and 1 for quadratures returned by get_wall_quad() and get_bndry_quad().

subsplx For codim == 1 the number of the wall-simplex this quadrature can be used for; this implies that lambda[iq][subsplx] zero.

n_points The number of quadrature points.

n_points_max The maximal number of quadrature points. The number of quadrature points can vary from simplex to simplex if INIT_ELEMENT_METHOD(quad) is not NULL.

lambda Vector lambda[0],...,lambda[n_points-1] of quadrature points given in barycentric coordinates (thus having N_LAMBDA_MAX components).

w vector w[0],...,w[n_points-1] of quadrature weights.

metadata Pointer to an internal data structure for per-element quadrature caches and the like, see e.g. Section 4.2.6

INIT_ELEMENT_DECL Function pointer to a per-element initializer. This pointer is always NULL for quadratures returned by get_quadrature(), get_wall_quad() and get_bndry_quad(). External extension modules make use of it. See Section 3.11.

Currently, numerical quadrature formulas exact up to degree 19 in one (Gauss formulas), up to degree 17 in two, up to degree 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 [5]. The implemented quadrature formulas are taken from [8, 11, 13, 26].

Using a conical product rule it is possible to construct new (non-symmetric) quadrature formulas from the existing ones, if that is really needed.

Functions for numerical quadrature are

```
const QUAD *get_quadrature(int dim, int degree);
REAL integrate_std_simp(const QUAD *quad, REAL (*f)(const REAL *));
const QUAD *get_product_quad(const QUAD *oq);
const QUAD *get_lumping_quadrature(int dim);
void register_quadrature(QUAD *quad);
bool new_quadrature(const QUAD *quad);
```

Description:

`get_quadrature(dim, degree)` returns a pointer to a `QUAD` structure for numerical integration in `dim` dimensions which is exact of degree $\min(19, \texttt{degree})$ for `dim==1`, $\min(17, \texttt{degree})$ for `dim==2`, and $\min(7, \texttt{degree})$ for `dim==3`.

It is possible to extend the maximal degrees by installing an application-defined quadrature rule via `new_quadrature()`.

`register_quadrature(quad)` Equip an application-defined quadrature with internal used meta-data for quadrature caches; this function also updates `n_quad_points_max[quad->dim]`. To install `quad` as a default quadrature which will returned on request by `get_quadrature()` the functions `new_quadrature()` has to be called additionally.

`new_quadrature(quad)` Install the given quadrature as new default quadrature for its dimension and polynomial degree; this means that `get_quadrature()` will return a pointer to `quad` when called with `quad->dim` and `quad->degree`.

`get_product_quad(quad)` Return a conical product quadrature rule. The returned quadrature formula is non-symmetric and works for one dimension higher than `quad` and is exact of the same degree as `quad`. The 3D formula for degree 7 found in [26] is of this type, for example.

Note that `get_product_quad()` installs the new formula calling `new_quadrature()`, so the formula will be available through `get_quadrature()`.

`get_lumping_quadrature(dim)` Returns a lumping quadrature with quadrature nodes at the vertices of the reference simplex.

`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

$$\sum_{k=0}^{\texttt{quad->n\_points-1}} \texttt{quad->w[k] * (*f)(quad->lambda[k]);}$$

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 initialize values and gradients of functions at the quadrature nodes:

```
REAL *f_at_qp(REAL vec[], const QUAD *quad, REAL (*f)(const REAL_B lambda));
REAL_D *grd_f_at_qp(REAL_D vec[], const QUAD *quad,
                    const REAL *(*f)(const REAL_B));
REAL_D *f_d_at_qp(REAL_D vec[], const QUAD *quad,
                  const REAL *(*f)(const REAL_B lambda));
REAL_DD *grd_f_d_at_qp(REAL_DD vec[], const QUAD *quad,
                       const REAL_D *(*f)(const REAL_B lambda));

REAL *f_loc_at_qp(REAL vec[], const EL_INFO *el_info, const QUAD *quad,
                  REAL (*f)(const EL_INFO *el_info,
                            const QUAD *quad, int iq, void *ud),
                  void *ud);
REAL_D *grd_f_loc_at_qp(REAL_D vec[], const EL_INFO *el_info,
                        const QUAD *quad, const REAL_BD Lambda,
                        GRD_LOC_FCT_AT_QP grd_f, void *ud);
REAL_D *param_grd_f_loc_at_qp(REAL_D vec[], const EL_INFO *el_info,
                              const QUAD *quad, const REAL_BD Lambda[],
                              GRD_LOC_FCT_AT_QP grd_f, void *ud);
REAL_D *f_loc_d_at_qp(REAL_D vec[], const EL_INFO *el_info, const QUAD *quad,
                      const REAL *(*f)(REAL_D result, const EL_INFO *el_info,
                                       const QUAD *quad, int iq, void *ud),
                      void *ud);
REAL_DD *grd_f_loc_d_at_qp(REAL_DD vec[], const EL_INFO *el_info,
                           const QUAD *quad, const REAL_BD Lambda,
                           GRD_LOC_FCT_D_AT_QP grd_f, void *ud);
REAL_DD *param_grd_f_loc_d_at_qp(REAL_DD vec[], const EL_INFO *el_info,
                                 const QUAD *quad, const REAL_BD Lambda[],
                                 GRD_LOC_FCT_D_AT_QP grd_f, void *ud);

REAL *fx_at_qp(REAL vec[], const EL_INFO *el_info, const QUAD *quad,
               FCT_AT_X f);
REAL_D *grd_fx_at_qp(REAL_D vec[], const EL_INFO *el_info, const QUAD *quad,
                     GRD_FCT_AT_X grd_f);
REAL_D *fx_d_at_qp(REAL_D vec[], const EL_INFO *el_info, const QUAD *quad,
                   FCT_D_AT_X f);
REAL_DD *grd_fx_d_at_qp(REAL_DD vec[], const EL_INFO *el_info,
                        const QUAD *quad, GRD_FCT_D_AT_X grd_f);
```

Description:

`f_at_qp(vec, quad, f)` 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 `NULL`, 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(vec, quad, grd_f)` 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 NULL, 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 $0 \leq$ j $<$ DIM_OF_WORLD and $0 \leq$ i $<$ quad->n_points,

f_d_at_qp(vec, quad, fd) 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 NULL, 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 NULL, 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 $0 \leq$ j $<$ DIM_OF_WORLD and $0 \leq$ i $<$ quad->n_points.

grd_f_d_at_qp(vec, quad, grd_fd) 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 NULL, 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 NULL, 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 $0 \leq$ j,k $<$ DIM_OF_WORLD and $0 \leq$ i $<$ quad->n_points,

f_loc_at_qp(vec, el_info, quad, f, ud)

grd_f_loc_at_qp(vec, el_info, quad, Lambda, grd_f, ud)

param_grd_f_loc_at_qp(vec, el_info, quad, Lambda, grd_f, ud)

f_loc_d_at_qp(vec, el_info, quad, fd, ud)

grd_f_loc_d_at_qp(vec, el_info, quad, Lambda, grd_fd, ud)

param_grd_f_loc_d_at_qp(vec, el_info, quad, Lambda, grd_fd, ud)

fx_at_qp(vec, el_info, quad, f)

grd_fx_at_qp(vec, el_info, quad, grd_f)

fx_d_at_qp(vec, el_info, quad, fd)

grd_fx_d_at_qp(vec, el_info, quad, grd_fd)

4.2.2 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 (with respect to barycentric coordinates) 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 newly on every element.

Information that should be pre–computed can be specified by the following symbolic constants:

```
#define INIT_PHI          0x01
#define INIT_GRD_PHI      0x02
#define INIT_D2_PHI       0x04
#define INIT_D3_PHI       0x08
#define INIT_D4_PHI       0x10
#define INIT_TANGENTIAL   0x80
```

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;

  FLAGS            init_flag;

  int              dim;
  int              n_points;
  int              n_bas_fcts;
  int              n_points_max;
  int              n_bas_fcts_max;
  const REAL       *w;              /* shallow copy of quad->w */
  const REAL       (*const*phi);    /* [qp][bf] */
  const REAL_B     (*const*grd_phi);
  const REAL_BB    (*const*D2_phi);
  const REAL_BBB   (*const*D3_phi);
  const REAL_BBBB  (*const*D4_phi);

  /* For vector valued basis functions with a p.w. constant
   * directional derivative we cache that direction and make it
   * available for applications. The component is initialized by the
   * INIT_ELEMENT() method.
   *
```

```
     * So: phi_d[i] gives the value of the directional factor for the
     * i-th basis function. If (!bas_fcts->dir_pw_const), then phi_d is
     * NULL.
     */
    const REAL_D *phi_d;

    /* chain to next structure, if bas_fcts->chain is non-empty */
    DBL_LIST_NODE  chain;

    /* a clone of this structure, but as single item. */
    const QUAD_FAST *unchained;

    INIT_ELEMENT_DECL;

    void *internal;
  };
```

The entries yield following information:

quad Values stored for numerical quadrature `quad`.

bas_fcts Values stored for basis functions `bas_fcts`.

dim Clone of `quad->dim`.

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`, `INIT_D3_PHI` or `INIT_D4_PHI`. Not all basis functions have support for higher derivatives. There is one additional fill-flag, `INIT_TANGENTIAL` with the meaning that only the tangential derivatives of the basis functions will be computed if `quad` is a co-dimension 1 quadrature rule.

n_points The number of quadrature points; equals `quad->n_points`.

n_bas_fcts number of basis functions; equals `bas_fcts->n_bas_fcts`.

n_points_max The maximum number of quadrature points. If `quad->init_element()` is non-`NULL`, then the number of basis functions can vary on a per-element basis.

n_bas_fcts_max The maximum number of basis functions. If `bas_fcts->init_element` is non-`NULL`, then the number of basis functions can vary on a per-element basis.

w Vector of quadrature weights; `w = quad->w`.

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` $<$ `n_points`, and $0 \leq$ `k` $\leq d$;

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` $\leq d$.

D3_phi Matrix storing all third derivatives (with respect to the barycentric coordinates) if the flag INIT_D3_PHI is set;

D3_phi[i][j][k][l] Stores the value bas_fcts->D3_phi[j](quad->lambda[i])[k][l][m] for $0 \leq$ j $<$ n_bas_fcts, $0 \leq$ i $<$ n_points, and $0 \leq$ k,l,m $\leq d$.

D4_phi Matrix storing all fourth derivatives (with respect to the barycentric coordinates) if the flag INIT_D4_PHI is set;

D4_phi[i][j][k][l] Stores the value bas_fcts->D4_phi[j](quad->lambda[i])[k][l][m][n] for $0 \leq$ j $<$ n_bas_fcts, $0 \leq$ i $<$ n_points, and $0 \leq$ k,l,m,n $\leq d$.

phi_d The directional part of vector-valued basis functions, if that is constant on each element. This means, if bas_fcts->rdim == DIM_OF_WORLD and bas_fcts->dir_pw_const, then phi_d contains valid data, probably after calling QUAD_FAST.init_element() with the current element and the instance of the QUAD_FAST structure in question. See also Section 3.5.2.

chain If bas_fcts forms part of a chain of basis functions because the corresponding finite element space is a direct sum, then this codeget_quad_fast() will also generate a chain of QUAD_FAST-structures, one for each component. The chain forms a doubly linked list, and the chain-component is the list node. See also Section 3.5.3 and Section 3.7.

unchained A clone of the current structure, but as single element. Points back to the structure itself if the underlying basis functions do not form part of chain of basis function sets. See Section 3.5.3 and Section 3.7.

INIT_ELEMENT_DECL Per element initializer, see Section 3.11.

internal Pointer to internal meta-data stuff.

A filled structure can be accessed by a call of

```
const QUAD_FAST *get_quad_fast(const BAS_FCTS *, const QUAD *, U_CHAR);
```

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 NULL pointer, if INIT_PHI flag is set and bas_fcts->phi is NULL, INIT_GRD_PHI flag is set and bas_fcts->grd_phi is NULL, and INIT_D2_PHI flag is set and bas_fcts->D2_phi is NULL.

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 on 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 passed to the element routine.

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.

4.2.3 Integration over subsimplices (walls)

The weak formulation of non-homogeneous Neumann or Robin boundary values needs integration over $d-1$ dimensional boundary simplices of d dimensional mesh elements (compare Section 1.6), and the evaluation of jump residuals for error estimators (compare Sections 1.5, 4.9) needs integration over all interior $d-1$ dimensional sub–simplices. The quadrature formulas and data structures described above are available for any d dimensional simplex, $d = 0, 1, 2, 3$. The above task can therefore be accomplished by using a $d-1$ dimensional quadrature formula and augmenting the corresponding d dimensional barycentric coordinates of quadrature points on edges/faces to $d+1$ dimensional coordinates on adjacent mesh elements.

When an integral over an edge/face involves values from both adjacent elements (in the computation of jump residuals e. g.) it is necessary to have a common orientation of the edge/face from both elements. Only a common orientation of the edges/faces ensures that augmenting d dimensional barycentric coordinates of quadrature points on the edge/face to $d+1$ dimensional barycentric coordinates on the adjacent mesh elements results in the same points from both sides.

This augmentation process, taking the relative orientation of neighboring simplices into account, is taken care of by dedicated co-dimension 1 quadrature rules, see Section 4.2.4 and 4.2.5. Additionally, the calculation of Gram's determinant for the $d-1$ dimensional transformation as well as vertex/edge/face normals is needed. See Section 4.1 above.

Low-level access to the relative orientation of neighboring simplices is provided through the routines and look-up tables

```
int wall_orientation(int dim, const EL *el, int wall, int **vec);
int wall_rel_orientation(
  int dim, const EL *el, const EL *neigh, int wall, int oppv);
```

```
const int sorted_wall_vertices_1d[N_WALLS_1D][DIM_FAC_1D][2*N_VERTICES_0D-1];
const int sorted_wall_vertices_2d[N_WALLS_2D][DIM_FAC_2D][2*N_VERTICES_1D-1];
const int sorted_wall_vertices_3d[N_WALLS_3D][DIM_FAC_3D][2*N_VERTICES_2D-1];
```

Description:

wall_orientation(dim, el, wall, vec) can be used to match the local enumeration of the vertices of faces separating neighbouring simplexes. The return value is a unique number between 1 and dim!. On return vec – if non-NULL– contains a permutation of the local numbering of the vertices of face number wall on el. If neigh_vec is the corresponding permutation for the neighbour element, then (*vec)[i] and (*neigh_vec)[i] refer to the same vertex, e.g. el_info->coord[(*vec)[i]] is the same as neigh_info->coord[(*neigh_vec)[i]].

Actually, the return value of wall_orientation() is just the index into the look-up tables sorted_wall_vertices_Xd[][][], such that vec, if non-NULL point upon return to sorted_wall_vertices_Xd[wall][retval].

The principal purpose of this function is to match quadrature points during the numerical integration of jumps of derivatives of finite element function across the faces of the triangulation, see Section 4.2.3.

wall_rel_orientation(dim, el, neigh, wall, oppv) can be used to compute a relative orientation of a given wall separating two elements with respect to both elements. The return value

```
perm = wall_rel_orientation(dim, el, neigh, wall, oppv);
```

can be used as an offset into sorted_wall_vertices_Xd in the sense that

```
nv = sorted_wall_vertices_Xd[oppv][perm][i];
```

matches

```
v = vertex_of_wall_Xd[wall][i];
```

So it holds el->dof[v][0] == neigh->dof[nv][0].

4.2.4 The WALL_QUAD data structure

A collection of quadrature rules for the integration over walls (3d: faces, 2d: edges) of a simplex. The quadrature points of these rules are given in barycentric coordinates with dim+1 valid components; the component corresponding to the respective wall will be set to zero.

Each of the quadrature rules WALL_QUAD quad[wall] may have its own INIT_ELEMENT method. INIT_ELEMENT(el_info, WALL_QUAD) may or may not be called: it is legal to only call INIT_ELEMENT(el_info, WALL_QUAD quad[wall]) individually. If INIT_ELEMENT(el_info, WALL_QUAD) is called, then it has to initialize all quadrature rules for all walls, so the subordinate initializers need not be called in this case.

```
typedef struct wall_quadrature   WALL_QUAD;

struct wall_quadrature
{
  const char *name;
  int   degree;
  int   dim;
  int   n_points_max;
  QUAD quad[N_WALLS_MAX];

  INIT_ELEMENT_DECL;

  void *metadata;
};
```

Description:

name Textual description of the quadrature.

degree Quadrature is exact of degree `degree`.

dim Quadrature for dimension `dim`; the barycentric coordinates of the quadrature points have `dim+1` valid components.

n_points_max The maximal number of quadrature points.

quad Quadrature rules for each wall. These are co-dimension 1 rules.

INIT_ELEMENT_DECL Function pointer to a per-element initializer. This pointer is always `NULL` for quadratures returned by `get_wall_quad()`. External extension modules may make use of it. See Section 3.11.

metadata Pointer to an internal data structure for per-element quadrature caches and the like.

Functions for numerical quadrature are:

```
const WALL_QUAD *get_wall_quad(int dim, int degree);
void register_wall_quadrature(WALL_QUAD *wall_quad);
const QUAD *get_neigh_quad(const EL_INFO *el_info, const WALL_QUAD
    *wall_quad, int neigh);
```

Description:

get_wall_quad(dim, degree) returns a pointer to a `WALL_QUAD` structure for numerical integration in `dim` dimensions.

register_wall_quadrature(wall_quad) initializes the meta-data for the given `WALL_QUAD`, no need to call this if the `WALL_QUAD` has been acquired by `get_wall_quad()`, only needed for externally defined extension quadrature rules.

4.2.5 The `WALL_QUAD_FAST` data structure

Convenience structure for `WALL_QUAD`: its is legal to call `get_quad_fast(bas_fcts, WALL_QUAD::quad[wall], ...)` (see Section 4.2.2 "The `QUAD_FAST` data structure") individually, however `get_wall_quad_fast()` does this in a single run. If `INIT_ELEMENT(el_info, WALL_QUAD_FAST)` is called, then the sub-ordinate initializers `INIT_ELEMENT(el_info,WALL_QUAD_FAST::quad_fast[wall])` need not be called.

```
typedef struct wall_quad_fast  WALL_QUAD_FAST;

struct wall_quad_fast
{
  const WALL_QUAD *wall_quad;
  const BAS_FCTS  *bas_fcts;

  FLAGS           init_flag;
  const QUAD_FAST *quad_fast[N_WALLS_MAX];

  INIT_ELEMENT_DECL;
};
```

The entries yield following information:

wall_quad values stored for numerical quadrature quad;

bas_fcts values stored for basis functions bas_fcts;

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;

quad_fast[N_WALLS_MAX] Pointer to N_WALLS_MAX quad_fast structures.

INIT_ELEMENT_DECL Function pointer to for a per-element initialiser. This pointer is always NULL for quadratures returned by get_quadrature(), get_wall_quad() and get_bndry_quad(). External extension modules make use of it. See Section 3.11.

```
const WALL_QUAD *get_wall_quad_fast(const BAS_FCTS *, const WALL_QUAD *,
    FLAGS init_flag);
QUAD_FAST *get_neigh_quad_fast(const EL_INFO *el_info,
                               const WALL_QUAD_FAST *wqfast,
                               int neigh);
```

Description:

get_wall_quad_fast(bas_fcts, wall_quad, init_flag) bas_fcts is a pointer to a filled BAS_FCTS structure, wall_quad a pointer to some quadrature (accessed by get_wall_quad(), e.g.) and init_flag indicates which information should be filled into the QUAD_FAST structure. The function returns a pointer to a filled QUAD_FAST structure where all demanded information is computed and stored.

get_neigh_quad(el_info, wall_quad, neigh) returns a suitable quadrature for integrating over the given wall (neigh number), but the barycentric co-ordinates of QUAD->lambda are relative to the neighbour element.

get_neigh_quad_fast(el_info, wall_quad, neigh) returns a suitable QUAD_FAST structure for integrating over the given wall, but relative to the neighbour element. If the returned QUAD_FAST object has a per-element initializer, then it must be called with an EL_INFO structure for the neighbour element. It is also legal to just call get_quad_fast(bas_fcts, get_neigh_quad(el_info, wall_quad, neigh), ...) but get_neigh_quad_fast() is slightly more efficient.

4.2.6 Caching of geometric quantities on quadrature nodes

Like for geometric quantities which are constant on a given mesh-element it is useful to share geometric data attached to quadrature nodes between different places of program code, see also Section 3.2.8. For this purpose there is a per-quadrature-per-element cache, called `QUAD_EL_CACHE`, which can be filled and accessed through calls to the function `fill_quad_el_cache()`. This is in particular useful for higher-order parametric meshes, where for example the transformation to the reference element is no longer piece-wise constant on each element. Internally, the `QUAD_EL_CACHE` is maintained as part of the "metadata" attached to each quadrature rule, see 4.2. The per-quadrature node cache and the related definitions and proto-types are as follows:

```
typedef struct quad_el_cache QUAD_EL_CACHE;

struct quad_el_cache
{
  EL       *current_el;
  FLAGS    fill_flag;
  REAL_D   *world;
  struct {
    REAL       *det;
    REAL_BD    *Lambda;
    REAL_BDD   *DLambda;
    REAL_BD    *grd_world;
    REAL_BDB   *D2_world;
    REAL_BDBB  *D3_world;
    REAL       *wall_det;      /* for co-dim 1 */
    REAL_D     *wall_normal;   /* for co-dim 1 */
    REAL_DB    *grd_normal;    /* for co-dim 1 */
    REAL_DBB   *D2_normal;     /* for co-dim 1 */
  } param;
};

#define FILL_EL_QUAD_WORLD          0x0001
#define FILL_EL_QUAD_DET            0x0002
#define FILL_EL_QUAD_LAMBDA         0x0004
#define FILL_EL_QUAD_DLAMBDA        0x0008
#define FILL_EL_QUAD_GRD_WORLD      0x0010
#define FILL_EL_QUAD_D2_WORLD       0x0020
#define FILL_EL_QUAD_D3_WORLD       0x0040
#define FILL_EL_QUAD_WALL_DET       0x0100
#define FILL_EL_QUAD_WALL_NORMAL    0x0200
#define FILL_EL_QUAD_GRD_NORMAL     0x0400
#define FILL_EL_QUAD_D2_NORMAL      0x0800

static inline const QUAD_EL_CACHE *fill_quad_el_cache(const EL_INFO *el_info,
                                                      const QUAD *quad,
                                                      FLAGS fill);
```

The quadrature cache can be obtained and filled by calls to `fill_quad_el_cache()`, see also below Example 4.2.1. The members of `QUAD_EL_CACHE` have the following meaning:

`current_el` For internal use only.

`fill_flag` A bit-mask, bit-wise or of the fill flags listed above (4.17).

world The world co-ordinates of the quadrature points, filled by `fill_quad_el_cache(..., FILL_EL_QUAD_WORLD)`.

param A cache for geometric quantities which are constant on each element for affine-linear meshes, but vary between quadrature points for higher-order parametric meshes.

- **det** The determinant of the transformation to the reference element, filled by filled by `fill_quad_el_cache(..., FILL_EL_QUAD_DET)`.
- **Lambda** The derivative of the barycentric coordinates w.r.t. the Cartesian coordinates, filled by `fill_quad_el_cache(..., FILL_EL_QUAD_LAMBDA)`.
- **DLambda** The second derivatives of the barycentric coordinates w.r.t. the Cartesian coordinates, filled by `fill_quad_el_cache(..., FILL_EL_QUAD_DLAMBDA)`.
- **grd_world** The first derivatives of the Cartesian coordinates w.r.t. the barycentric coordinates, filled by `fill_quad_el_cache(..., FILL_EL_QUAD_GRD_WORLD)`.
- **D2_world** The second derivatives of the Cartesian coordinates w.r.t. the barycentric coordinates, filled by `fill_quad_el_cache(..., FILL_EL_QUAD_D2_WORLD)`.
- **D3_world** The third derivatives of the Cartesian coordinates w.r.t. the barycentric coordinates, filled by `fill_quad_el_cache(..., FILL_EL_QUAD_D3_WORLD)`.
- **wall_det** The determinant of the transformation of the walls to the reference element's walls. This can be filled only for co-dimension 1 quadratures by `fill_quad_el_cache(..., FILL_EL_QUAD_WALL_DET)`.
- **wall_normal** The outer wall-normal. This can be filled only for co-dimension 1 quadratures by `fill_quad_el_cache(..., FILL_EL_QUAD_WALL_NORMAL)`.
- **grd_normal** The first derivative of the outer normal-field with respect to the barycentric coordinates. This can be filled only for co-dimension 1 quadratures by `fill_quad_el_cache(..., FILL_EL_QUAD_GRD_NORMAL)`.
- **D2_normal** The second derivative of the outer normal-field with respect to the barycentric coordinates. This can be filled only for co-dimension 1 quadratures by `fill_quad_el_cache(..., FILL_EL_QUAD_D2_NORMAL)`.

4.2.1 Example. A simple example which computes the measure of the region occupied by the mesh (of course, this can be achieved more efficiently by computing a boundary integral ...). This example is, of course, quite artificial – and in this context it would be more efficient *not* to read through the per-element caches.

```
const PARAMERIC *param = mesh->parametric;
const QUAD *quad = get_quadrature(mesh->dim, 3 /* degree */);
REAL meas = 0.0;
TRAVERSE_FIRST(mesh, -1, CALL_LEAF_EL|FILL_COORDS) {
  int iq;
  if (param->init_element(el_info, param)) {
    const QUAD_EL_CACHE *qelc = fill_quad_el_cache(el_info, quad,
        FILL_EL_QUAD_DET);
    for (iq = 0; iq < quad->n_points; iq++) {
      meas += quad->w[iq] * qelc->param.det[iq];
    }
  } else {
    const EL_GEOM_CACHE *elgc = fill_el_geom_cache(el_info, FILL_EL_DET);
    meas += elgc->det / (REAL)DIM_FAC(mesh->dim);
  }
} TRAVERSE_NEXT()
```

4.3 Functions for the evaluation of finite elements

Finite element functions are evaluated locally on single elements using barycentric coordinates (compare Section 1.4.3). ALBERTA 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 currently only implemented for (non–parametric) simplices.

Recalling (1.4.3) on page 19 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 $\left(\bar{\varphi}^1, \ldots, \bar{\varphi}^m\right)$ is a basis of $\bar{\mathbb{P}}$ and $\left(u_S^1, \ldots, u_S^m\right)$ the local coefficient vector of u_h on S. Derivatives are evaluated on S 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 Section 1.4.3.1.

These formulas are used for all evaluation routines. Information about values of basis functions and their 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).

The following routines calculate values of a finite element function at a single point, given in barycentric coordinates:

```
REAL eval_uh(const REAL_B lambda, const EL_REAL_VEC *uh_loc,
             const BAS_FCTS *bfcts);
REAL *eval_grd_uh(REAL_D result, const REAL_B lambda, const REAL_BD Lambda,
                  const EL_REAL_VEC *uh_loc, const BAS_FCTS *bfcts);
REAL_D *eval_D2_uh(REAL_DD result, const REAL_B lambda, const REAL_BD Lambda,
                   const EL_REAL_VEC *uh_loc, const BAS_FCTS *bfcts);

REAL *eval_uh_d(REAL_D result, const REAL_B lambda,
                const EL_REAL_D_VEC *uh_loc, const BAS_FCTS *bfcts);
REAL_D *eval_grd_uh_d(REAL_DD result, const REAL_B lambda,
                      const REAL_BD Lambda, const EL_REAL_D_VEC *uh_loc,
                      const BAS_FCTS *bfcts);
REAL eval_div_uh_d(const REAL_B lambda, const REAL_BD Lambda,
                   const EL_REAL_D_VEC *uh_loc, const BAS_FCTS *bfcts);
REAL_DD *eval_D2_uh_d(REAL_DDD result, const REAL_B lambda,
                      const REAL_BD Lambda, const EL_REAL_D_VEC *uh_loc,
                      const BAS_FCTS *bfcts);

REAL *eval_uh_dow(REAL_D result, const REAL_B lambda,
                  const EL_REAL_VEC_D *uh_loc, const BAS_FCTS *bfcts);
REAL_D *eval_grd_uh_dow(REAL_DD result, const REAL_B lambda,
                        const REAL_BD Lambda, const EL_REAL_VEC_D *uh_loc,
                        const BAS_FCTS *bfcts);
REAL eval_div_uh_dow(const REAL_B lambda, const REAL_BD Lambda,
```

```
                        const EL_REAL_VEC_D *uh_loc, const BAS_FCTS *bfcts);
REAL_DD *eval_D2_uh_dow(REAL_DDD result, const REAL_B lambda,
                        const REAL_BD Lambda, const EL_REAL_VEC_D *uh_loc,
                        const BAS_FCTS *bfcts);
```

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 statically allocated memory area. This area is overwritten during the next call. If the first argument of such a function is not `NULL`, then it is a pointer to a storage area where the results are stored. This memory area must be of correct size, no check is performed.

4.3.1 Compatibility Note. *Former versions of ALBERTA expected the argument providing optional storage for the result at the last place in the parameter list. In the current version of the library, storage for the result is still optional, but generally passed as first argument to the respective function.*

The functions for `DIM_OF_WORLD`-valued discrete functions come in two variants, one for discrete functions based on scalar-valued local basis function sets, where the coefficients are `DIM_OF_WORLD`-valued, and one for discrete functions which may be based on either scalar-valued or `DIM_OF_WORLD`-valued local basis functions, modeled by `DOF_REAL_VEC_D` – and locally by `EL_REAL_VEC_D` – objects. The names for the latter functions have a `..._dow` suffix, the others a `..._d` suffix. Besides the slightly differing argument types the calling conventions for both variants are the same, so they are documented together in the descriptions following below.

`eval_uh(lambda, uh_loc, bas_fcts)` the function returns $u_h(\lambda)$.

`eval_grd_uh(result, lambda, Lambda, uh_loc, bas_fcts)` the function returns a pointer `ptr` to a vector of length `DIM_OF_WORLD` storing $\nabla u_h(\lambda)$, i.e.

$$\mathtt{ptr[i]} = u_{h,x_{\mathtt{i}}}(\lambda), \qquad \mathtt{i} = 0, \ldots, \mathtt{DIM\_OF\_WORLD} - 1;$$

`result` is optional and provides storage for the result if non-`NULL`. See Compatibility Note 4.3.1.

`eval_D2_uh(result, lambda, Lambda, uh_loc, bas_fcts)` 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.

$$\mathtt{ptr[i][j]} = u_{h,x_{\mathtt{i}}x_{\mathtt{j}}}(\lambda), \qquad \mathtt{i,j} = 0, \ldots, \mathtt{DIM\_OF\_WORLD} - 1;$$

`result` is optional and provides storage for the result if non-`NULL`. See Compatibility Note 4.3.1.

`eval_uh_[d|dow](result, lambda, uh_loc, bas_fcts)` the function returns a pointer `ptr` to a vector of length `DIM_OF_WORLD` storing $u_h(\lambda)$, i.e.

$$\mathtt{ptr[k]} = u_{h\mathtt{k}}(\lambda), \qquad \mathtt{k} = 0, \ldots, \mathtt{DIM\_OF\_WORLD} - 1;$$

`result` is optional and provides storage for the result if non-`NULL`. See Compatibility Note 4.3.1.

eval_grd_uh_[d|dow](result, lambda, Lambda, uh_loc, bas_fcts) 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.

$$\texttt{ptr[k][i]} = u_{h\,\texttt{k},x_\texttt{i}}(\lambda), \qquad \texttt{k},\texttt{i} = 0,\ldots,\texttt{DIM\_OF\_WORLD} - 1;$$

result is optional and provides storage for the result if non-NULL. See Compatibility Note 4.3.1.

eval_div_uh_[d|dow](lambda, Lambda, uh_loc, bas_fcts) the function returns $\operatorname{div} u_h(\lambda)$.

eval_D2_uh_[d|dow](result, lambda, Lambda, uh_loc, bas_fcts) the function returns a pointer ptr to a vector of (DIM_OF_WORLD × DIM_OF_WORLD) matrices of length DIM_OF_WORLD storing $D^2 u_h(\lambda)$, i.e.

$$\texttt{ptr[k][i][j]} = u_{h\,\texttt{k},x_\texttt{i}x_\texttt{j}}(\lambda), \qquad \texttt{k},\texttt{i},\texttt{j} = 0,\ldots,\texttt{DIM\_OF\_WORLD} - 1;$$

result is optional and provides storage for the result if non-NULL. See Compatibility Note 4.3.1.

Using pre–computed values of basis functions at the evaluation point, these routines can be implemented more efficiently.

```
REAL eval_uh_fast(const EL_REAL_VEC *uh_loc, const QUAD_FAST *qfast, int iq);
const REAL *eval_grd_uh_fast(REAL_D grd_uh, const REAL_BD Lambda,
                             const EL_REAL_VEC *uh_loc,
                             const QUAD_FAST *qfast, int iq);
const REAL_D *eval_D2_uh_fast(REAL_DD result, const REAL_BD Lambda,
                              const EL_REAL_VEC *uh_loc,
                              const QUAD_FAST *qfast, int iq);

const REAL *eval_uh_d_fast(REAL_D result, const EL_REAL_D_VEC *uh_loc,
                           const QUAD_FAST *qfast, int iq);
const REAL_D *eval_grd_uh_d_fast(REAL_DD result, const REAL_BD Lambda,
                                 const EL_REAL_D_VEC *uh_loc,
                                 const QUAD_FAST *qfast, int iq);
REAL eval_div_uh_d_fast(const REAL_BD Lambda, const EL_REAL_D_VEC *uh_loc,
                        const QUAD_FAST *qfast, int iq);
const REAL_DD *eval_D2_uh_d_fast(REAL_DDD result, const REAL_BD Lambda,
                                 const EL_REAL_D_VEC *uh_loc,
                                 const QUAD_FAST *qfast, int iq);

const REAL *eval_uh_dow_fast(REAL_D result, const EL_REAL_VEC_D *uh_loc,
                             const QUAD_FAST *qfast, int iq);
const REAL_D *eval_grd_uh_dow_fast(REAL_DD result, const REAL_BD Lambda,
                                   const EL_REAL_VEC_D *uh_loc,
                                   const QUAD_FAST *qfast, int iq);
REAL eval_div_uh_dow_fast(const REAL_BD Lambda, const EL_REAL_VEC_D *uh_loc,
                          const QUAD_FAST *qfast, int iq);
const REAL_DD *eval_D2_uh_dow_fast(REAL_DDD result, const REAL_BD Lambda,
                                   const EL_REAL_VEC_D *uh_loc,
                                   const QUAD_FAST *qfast, int iq);
```

4.3.2 Compatibility Note. *Former versions of ALBERTA didn't expect the arguments*

*..., **const** QUAD_FAST *qfast, **int** iq, ...*

– meaning the quadrature cache and the index of the quadrature point – but instead expected the actual cached-values to be passed, i.e. for the computation of the gradient

..., qfast->grd_phi[iq], qfast->n_bas_fcts, ...

*There is some potential for confusion, in particular because the proto-types listed in the old documentation often omit the parameter name and only give the parameter type. In the new version, .., **int** iq, ... denotes the index of the quadrature point. The number of basis functions on the reference element is not needed, because the evaluation functions fetch this quantity themselves from the QUAD_FAST data structure.*

Description: In the following `Lambda` $= \Lambda$ denotes the Jacobian of the barycentric coordinates, `uh_loc` the local coefficient vector (of type EL_REAL_VEC, EL_REAL_D_VEC etc.) on an element.

`eval_uh_fast(uh_loc, qfast, iq)` the function returns $u_h(\lambda)$;
`qfast` is a quadrature cache storing the values $\bar\varphi^0(\lambda), \dots, \bar\varphi^{\mathtt{m}-1}(\lambda)$.

`eval_grd_uh_fast(grd, Lambda, uh_loc, qfast, iq)` the function returns a pointer `ptr` to a vector of length `DIM_OF_WORLD` storing $\nabla u_h(\lambda)$, i.e.

$$\mathtt{ptr[i]} = u_{h,x_\mathtt{i}}(\lambda), \qquad \mathtt{i} = 0, \dots, \mathtt{DIM\_OF\_WORLD} - 1;$$

`grd` is optional and provides storage for the result if non-`NULL`. See Compatibility Note 4.3.1.
`qfast` is a quadrature cache storing $\nabla_\lambda \bar\varphi^0(\lambda), \dots, \nabla_\lambda \bar\varphi^{\mathtt{m}-1}(\lambda)$;

`eval_D2_uh_fast(D2, Lambda, uh_loc, qfast, iq)` the function returns a pointer `ptr` to a matrix of size (`DIM_OF_WORLD` × `DIM_OF_WORLD`) storing $D^2 u_h(\lambda)$, i.e.

$$\mathtt{ptr[i][j]} = u_{h,x_\mathtt{i}x_\mathtt{j}}(\lambda), \qquad \mathtt{i,j} = 0, \dots, \mathtt{DIM\_OF\_WORLD} - 1;$$

`D2` is optional and provides storage for the result if non-`NULL`. See Compatibility Note 4.3.1.
`qfast` is a quadrature cache storing $D^2_\lambda \bar\varphi^0(\lambda), \dots, D2_\lambda \bar\varphi^{\mathtt{m}-1}(\lambda)$.

`eval_uh_[d|dow]_fast(result, uh_loc, qfast, iq)` 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.

$$\mathtt{ptr[k][i]} = u_{h\mathtt{k},x_\mathtt{i}}(\lambda), \qquad \mathtt{k,i} = 0, \dots, \mathtt{DIM\_OF\_WORLD} - 1;$$

`qfast` is a quadrature cache storing the values $\bar\varphi^0(\lambda), \dots, \bar\varphi^{\mathtt{m}-1}(\lambda)$;
`result` is optional and provides storage for the resulty if non-`NULL`. See Compatibility Note 4.3.1.

`eval_grd_uh_[d|dow]_fast(grd, Lambda, uh_loc, qfast, iq)` 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.

$$\mathtt{ptr[k][i]} = u_{h\mathtt{k},x_\mathtt{i}}(\lambda), \qquad \mathtt{k,i} = 0, \dots, \mathtt{DIM\_OF\_WORLD} - 1;$$

`qfast` is a quadrature cache storing $\nabla_\lambda \bar\varphi^0(\lambda), \dots, \nabla_\lambda \bar\varphi^{\mathtt{m}-1}(\lambda)$; `grd` is optional storage for the result if non-`NULL`. See Compatibility Note 4.3.1.

`eval_div_uh_[d|dow]_fast(Lambda, uh_loc, qfast, iq)` the function returns $\operatorname{div} u_h(\lambda)$;
`qfast` is a quadrature cache storing $\nabla_\lambda \bar\varphi^0(\lambda), \dots, \nabla_\lambda \bar\varphi^{\mathtt{m}-1}(\lambda)$. Unused entries must be set to `0.0`.

`eval_D2_uh_[d|dow]_fast(D2, Lambda, uh_loc, qfast, iq)` the function returns a pointer `ptr` to a vector of (`DIM_OF_WORLD` × `DIM_OF_WORLD`) matrices of length `DIM_OF_WORLD` storing $D^2 u_h(\lambda)$, i.e.

$$\texttt{ptr[k][i][j]} = u_{h\texttt{k},x_\texttt{i}x_\texttt{j}}(\lambda), \qquad \texttt{k,i,j} = 0,\ldots,\texttt{DIM\_OF\_WORLD} - 1;$$

`qfast` is a quadrature cache storing $D^2_\lambda \bar{\varphi}^0(\lambda),\ldots,D2_\lambda \bar{\varphi}^{\texttt{m}-1}(\lambda)$;
`D2` is optional storage for the result if non-`NULL`. See Compatibility Note 4.3.1.

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 these quadrature points.

```
REAL *uh_at_qp(REAL *result, const QUAD_FAST *qfast,
               const EL_REAL_VEC *uh_loc);
REAL_D *grd_uh_at_qp(REAL_D *result, const QUAD_FAST *qfast,
                     const REAL_BD Lambda, const EL_REAL_VEC *uh_loc);
REAL_DD *D2_uh_at_qp(REAL_DD *result, const QUAD_FAST *qfast,
                     const REAL_BD Lambda, const EL_REAL_VEC *uh_loc);

REAL_D *param_grd_uh_at_qp(REAL_D vec[], const QUAD_FAST *qfast,
                           const REAL_BD Lambda[], const EL_REAL_VEC
                              *uh_loc);
REAL_DD *param_D2_uh_at_qp(REAL_DD *result, const QUAD_FAST *qfast,
                           const REAL_BD Lambda[], const REAL_BDD DLambda[],
                           const EL_REAL_VEC *uh_loc);

REAL_D *uh_d_at_qp(REAL_D *result, const QUAD_FAST *qfast,
                   const EL_REAL_D_VEC *uh_loc);
REAL_DD *grd_uh_d_at_qp(REAL_DD *result, const QUAD_FAST *qfast,
                        const REAL_BD Lambda, const EL_REAL_D_VEC *uh_loc);
REAL *div_uh_d_at_qp(REAL *result, const QUAD_FAST *qfast,
                     const REAL_BD Lambda, const EL_REAL_D_VEC *uh_loc);
REAL_DDD *D2_uh_d_at_qp(REAL_DDD vec[], const QUAD_FAST *qfast,
                        const REAL_BD Lambda, const EL_REAL_D_VEC *uh_loc);

REAL_DD *param_grd_uh_d_at_qp(REAL_DD vec[], const QUAD_FAST *qfast,
                              const REAL_BD Lambda[],
                              const EL_REAL_D_VEC *uh_loc);
REAL *param_div_uh_d_at_qp(REAL vec[], const QUAD_FAST *qfast,
                           const REAL_BD Lambda[],
                           const EL_REAL_D_VEC *uh_loc);
REAL_DDD *param_D2_uh_d_at_qp(REAL_DDD vec[], const QUAD_FAST *qfast,
                              const REAL_BD grd_lam[],
                              const REAL_BDD DLambda[],
                              const EL_REAL_D_VEC *uh_loc);

REAL_D *uh_dow_at_qp(REAL_D *result, const QUAD_FAST *qfast,
                     const EL_REAL_VEC_D *uh_loc);
REAL_DD *grd_uh_dow_at_qp(REAL_DD *result, const QUAD_FAST *qfast,
                          const REAL_BD Lambda, const EL_REAL_VEC_D *uh_loc);
REAL *div_uh_dow_at_qp(REAL *result, const QUAD_FAST *qfast,
                       const REAL_BD Lambda, const EL_REAL_VEC_D *uh_loc);
REAL_DDD *D2_uh_dow_at_qp(REAL_DDD vec[], const QUAD_FAST *qfast,
                          const REAL_BD Lambda, const EL_REAL_VEC_D *uh_loc);
```

```
REAL_DD *param_grd_uh_dow_at_qp(REAL_DD vec[], const QUAD_FAST *qfast,
                                const REAL_BD *Lambda,
                                const EL_REAL_VEC_D *uh_loc);
REAL *param_div_uh_dow_at_qp(REAL vec[], const QUAD_FAST *qfast,
                             const REAL_BD *Lambda,
                             const EL_REAL_VEC_D *uh_loc);
REAL_DDD *param_D2_uh_dow_at_qp(REAL_DDD *result, const QUAD_FAST *qfast,
                                const REAL_BD *Lambda, const REAL_BDD
                                    *DLambda,
                                const EL_REAL_VEC_D *uh_loc);
```

Description: In the following `uh_loc` denotes the local coefficient vector (of type `EL_REAL_VEC`, `EL_REAL_D_VEC` etc.) on an element.

`uh_at_qp(result, qfast, uh_loc)` 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`;

`result` is optional and provides storage for the result if non-`NULL`. See Compatibility Note 4.3.1.

`grd_uh_at_qp(result, qfast, Lambda, uh_loc)` 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`;

`result` is optional and provides storage for the result if non-`NULL`. See Compatibility Note 4.3.1.

`D2_uh_at_qp(result, qfast, Lambda, uh_loc)`

`param_grd_uh_at_qp(result, qfast, Lambdas, uh_loc)` version for parametric meshes; must be passed a vector storing the gradients of the barycentric coordinates at each quadrature point. The same holds for the other `param_`-prefixed routines.

`[param_]D2_uh_at_qp(result, qfast, Lambda[s], uh_loc, D2)` The function returns a pointer `ptr` to a vector of length `qfast->n_points` of (`DIM_OF_WORLD` × `DIM_OF_WORLD`) matrices storing $D^2 u_h$ at all quadrature points of `qfast->quad`, i.e.

$$\texttt{ptr[l][i][j]} = u_{h,x_\texttt{i}x_\texttt{j}}(\texttt{qfast->quad->lambda[l]})$$

where $\texttt{l} = 0, \ldots, \texttt{qfast->quad->n\_points} - 1$, and $\texttt{i}, \texttt{j} = 0, \ldots, \texttt{DIM\_OF\_WORLD} - 1$;

the `INIT_D2_PHI` flag must be set in `qfast->init_flag`;

`result` is optional and provides storage for the result if non-`NULL`. See Compatibility Note 4.3.1.

uh_[d|dow]_at_qp(result, qfast, uh_loc) 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.

$$\texttt{ptr[l][k]} = u_{h\texttt{k}}(\texttt{qfast->quad->lambda[l]})$$

where $\texttt{l} = 0, \ldots, \texttt{qfast->quad->n\_points} - 1$, and $\texttt{k} = 0, \ldots, \texttt{DIM\_OF\_WORLD} - 1$;
the INIT_PHI flag must be set in qfast->init_flag;
result is optional and provides storage for the result if non-NULL. See Compatibility Note 4.3.1.

grd_uh_[d|dow]_at_qp(result, qfast, Lambda, uh_loc)

div_uh_[d|dow]_at_qp(result, qfast, Lambda, uh_loc)

D2_uh_[d|dow]_at_qp(result, qfast, Lambda[], uh_loc) The function returns a pointer ptr to a vector of length qfast->n_points of (DIM_OF_WORLD × DIM_OF_WORLD × DIM_OF_WORLD) tensors storing $D^2 u_h$ at all quadrature points qfast->quad, i.e.

$$\texttt{ptr[l][k][i][j]} = u_{h\texttt{k},x_\texttt{i}x_\texttt{j}}(\texttt{qfast->quad->lambda[l]})$$

where $\texttt{l} = 0, \ldots, \texttt{qfast->quad->n\_points} - 1$, and $\texttt{k,i,j} = 0, \ldots, \texttt{DIM\_OF\_WORLD} - 1$;
the INIT_D2_PHI flag must be set in qfast->init_flag;
result is optional and provides storage for the result if non-NULL. See Compatibility Note 4.3.1.

param_grd_uh_[d|dow]_at_qp(vec[], qfast, Lambda[], uh_loc) The function returns a pointer ptr to a vector of length qfast->n_points of (DIM_OF_WORLD × DIM_OF_WORLD) matrices storing ∇u_h at all quadrature points of qfast->quad, i.e.

$$\texttt{ptr[l][k][i]} = u_{h\texttt{k},x_\texttt{i}}(\texttt{qfast->quad->lambda[l]})$$

where $\texttt{l} = 0, \ldots, \texttt{qfast->quad->n\_points} - 1$, and $\texttt{k,i} = 0, \ldots, \texttt{DIM\_OF\_WORLD} - 1$;
the INIT_GRD_PHI flag must be set in qfast->init_flag;
vec is optional and provides storage for the result if non-NULL. See Compatibility Note 4.3.1.

param_div_uh_[d|dow]_at_qp(result[], qfast, Lambda[], uh_loc)

param_D2_uh_[d|dow]_at_qp(result, qfast, Lambda, DLambda, uh_loc) Second derivatives for parametric meshes. Note that one needs the second derivatives DLambda of the barycentric co-ordinates with respect to the cartesian co-ordiantes for this function. Also note that – in the case of non-zero co-dimension – the matrix $(\underline{\nabla}(\underline{\nabla} u)_i)_j$ built from the components of the second tangential derivatives is *not* symmetric in general.
vec is optional and provides storage for the result if non-NULL. See Compatibility Note 4.3.1.

```
REAL *eval_bar_grd_uh(REAL_B result, const REAL_B lambda,
                      const EL_REAL_VEC *uh_loc, const BAS_FCTS *bfcts);
REAL_B *eval_bar_grd_uh_d(REAL_DB result, const REAL_B lambda,
                          const EL_REAL_D_VEC *uh_loc, const BAS_FCTS
                            *bfcts);
REAL_B *eval_bar_grd_uh_dow(REAL_DB result, const REAL_B lambda,
                            const EL_REAL_VEC_D *uh_loc,
                            const BAS_FCTS *bfcts);
```

```
REAL *eval_bar_grd_uh_fast(REAL_B result, const EL_REAL_VEC *uh_loc,
                           const QUAD_FAST *qfast, int iq);
REAL_B *eval_bar_grd_uh_d_fast(REAL_DB result, const EL_REAL_D_VEC *uh_loc,
                               const QUAD_FAST *qfast, int iq);
REAL_B *eval_bar_grd_uh_dow_fast(REAL_DB result, const EL_REAL_VEC_D *uh_loc,
                                 const QUAD_FAST *qfast, int iq);

REAL_B *bar_grd_uh_at_qp(REAL_B *result, const QUAD_FAST *qfast,
                         const EL_REAL_VEC *uh_loc);
REAL_DB *bar_grd_uh_d_at_qp(REAL_DB *result, const QUAD_FAST *qfast,
                            const EL_REAL_D_VEC *uh_loc);
REAL_DB *bar_grd_uh_dow_at_qp(REAL_DB *result, const QUAD_FAST *fast,
                              const EL_REAL_VEC_D *uh_loc);

REAL_B *eval_bar_D2_uh(REAL_BB result, const REAL_B lambda,
                       const EL_REAL_VEC *uh_loc, const BAS_FCTS *bfcts);
REAL_BB *eval_bar_D2_uh_d(REAL_DBB result, const REAL_B lambda,
                          const EL_REAL_D_VEC *uh_loc, const BAS_FCTS
                            *bfcts);
REAL_BB *eval_bar_D2_uh_dow(REAL_DBB result, const REAL_B lambda,
                            const EL_REAL_VEC_D *uh_loc,
                            const BAS_FCTS *bfcts);

REAL_B *eval_bar_D2_uh_fast(REAL_BB result, const EL_REAL_VEC *uh_loc,
                            const QUAD_FAST *qfast, int iq);
REAL_BB *eval_bar_D2_uh_d_fast(REAL_DBB result, const EL_REAL_D_VEC *uh_loc,
                               const QUAD_FAST *qfast, int iq, bool update)
REAL_BB *eval_bar_D2_uh_dow_fast(REAL_DBB result, const EL_REAL_VEC_D
    *uh_loc,
                                 const QUAD_FAST *qfast, int iq);

REAL_BB *bar_D2_uh_at_qp(REAL_BB *result, const QUAD_FAST *qfast,
                         const EL_REAL_VEC *uh_loc);
REAL_DBB *bar_D2_uh_d_at_qp(REAL_DBB vec[], const QUAD_FAST *qfast,
                            const EL_REAL_D_VEC *uh_loc);
REAL_DBB *bar_D2_uh_dow_at_qp(REAL_DBB vec[], const QUAD_FAST *qfast,
                              const EL_REAL_VEC_D *uh_loc);
```

Description: These functions compute the respective derivatives with respect to barycentric co-ordinates. Otherwise they are functionally equivalent to the functions without the `bar_`-prefix.

4.4 Calculation of norms for finite element functions

ALBERTA 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.

```
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 *);
REAL H1_norm_uh_dow(const QUAD *, const DOF_REAL_VEC_D *);
REAL L2_norm_uh_dow(const QUAD *, const DOF_REAL_VEC_D *);
```

Descriptions

H1_norm_uh(quad, uh) returns an approximation to the H^1 semi norm $(\int_\Omega |\nabla u_h|^2)^{1/2}$ 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 `NULL`; 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 $(\int_\Omega |u_h|^2)^{1/2}$ 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 `NULL`; otherwise a quadrature which is exact of degree `2*uh->fe_space->bas_fcts->degree` is used.

H1_norm_uh_[d|dow](quad, uh_d) returns an approximation to the H^1 semi norm of a vector valued 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 `NULL`; otherwise a quadrature which is exact of degree `2*uh_d->fe_space->bas_fcts->degree-2` is used.

L2_norm_uh_[d|dow](quad, uh_d) returns an approximation to the L^2 norm of a vector valued 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 `NULL`; otherwise a quadrature which is exact of degree `2*uh_d->fe_space->bas_fcts->degree` is used.

4.5 Interface for application provided functions

Often the library function in the ALBERTA package require certain application provided functions, e.g. for assembling the "right hand side", for computations of the "true" error, for inhomogeneous boundary conditions or for interpolation of (non-discrete) functions onto finite element spaces. This section defines some basis calling conventions concerning these application provided functions.

Most of these function must conform to one of the following proto-types:

```
typedef REAL (*FCT_AT_X)(const REAL_D x);
typedef const REAL *(*GRD_FCT_AT_X)(const REAL_D x, REAL_D result);
typedef const REAL_D *(*D2_FCT_AT_X)(const REAL_D x, REAL_DD result);

typedef const REAL    *(*FCT_D_AT_X)(const REAL_D x, REAL_D result);
typedef const REAL_D  *(*GRD_FCT_D_AT_X)(const REAL_D x, REAL_DD result);
typedef const REAL_DD *(*D2_FCT_D_AT_X)(const REAL_D x, REAL_DDD result);

typedef REAL (*LOC_FCT_AT_QP)(const EL_INFO *el_info,
                              const QUAD *quad, int iq,
                              void *ud);
typedef const REAL *(*LOC_FCT_D_AT_QP)(REAL_D result,
                                       const EL_INFO *el_info,
                                       const QUAD *quad, int iq,
                                       void *ud);
typedef const REAL *(*GRD_LOC_FCT_AT_QP)(REAL_D res,
                                         const EL_INFO *el_info,
                                         const REAL_BD Lambda,
```

```
                                                const QUAD *quad, int iq,
                                                void *ud);
typedef const REAL_D *(*GRD_LOC_FCT_D_AT_QP)(REAL_DD res,
                                                const EL_INFO *el_info,
                                                const REAL_BD Lambda,
                                                const QUAD *quad, int iq,
                                                void *ud);
```

4.5.1 Datatype (FCT_AT_X).

Prototype

```
typedef REAL (*FCT_AT_X)(const REAL_D x);
```

Synopsis

```
FCT_AT_X fptr;

result = fptr(x);
```

Description

Evaluate at the point `x` and return a scalar value. This is the simplest function-type.

Parameters

`x` The point of evaluation.

Return Value

The function value.

4.5.2 Datatype (GRD_FCT_AT_X).

Prototype

```
const REAL *GRD_FCT_AT_X(const REAL_D x, REAL\_D result);
```

Synopsis

```
GRD_FCT_AT_X fptr;

result = fptr(x, result);
result = fptr(x, NULL);
```

Description

Evaluate the first derivative at the point `x`.

Parameters

`x` The point of evaluation, in Cartesian coordinates.

`result` Storage for the result, or `NULL`.

Return Value The address of `result`, if `result != NULL`, otherwise a pointer to a statically allocated storage area, see Example 4.5.3 below.

4.5.3 Example.

```
const REAL *grd_g_implementation(const REAL_D x, REAL_D result) {
  static REAL_D storage; /* mind the "static" key-word!!! */

  if (result == NULL) {
    result = storage;
  }

  ... /* mighty complicated computations for "result" */

  return result;
}
```

`const REAL_D *D2_FCT_AT_X(const REAL_D x, REAL_DD result)` Evaluate the second derivative at the point `x`.

Parameters

`x` The point of evaluation, in Cartesian coordinates.

`result` Storage for the result, or `NULL`.

Return Value The address of `result`, if `result != NULL`, otherwise a pointer to a statically allocated storage area, see Example 4.5.4 below.

4.5.4 Example.

```
const REAL_D *D2_g_implementation(const REAL_D x, REAL_DD result) {
  static REAL_DD storage; /* mind the "static" key-word!!! */

  if (result == NULL) {
    result = storage;
  }

  ... /* mighty complicated computations for "result" */

  return (const REAL_D *)result;
}
```

`const REAL *FCT_D_AT_X(const REAL_D x, REAL_D result)`

`const REAL_D *GRD_FCT_D_AT_X(const REAL_D x, REAL_DD result)`

`const REAL_DD *D2_FCT_D_AT_X(const REAL_D x, REAL_DDD result)` Evaluate a vector valued function at the point `x`. There is, of course, no difference between the `GRD_FCT_AT_X` and the `FCT_D_AT_X` function pointers.

Parameters

`x` The point of evaluation, in Cartesian coordinates.

`result` Storage for the result, or `NULL`.

Return Value The address of `result`, if `result != NULL`, otherwise a pointer to a statically allocated storage area, see Example 4.5.5 below.

4.5.5 Example.

```
const REAL *g_implementation(const REAL_D x, REAL_D result) {
  static REAL_D storage; /* mind the "static" key-word!!! */

  if (result == NULL) {
    result = storage;
  }

  ... /* mighty complicated computations for "result" */

  return result;
}
```

REAL LOC_FCT_AT_QP(

const EL_INFO *el_info, const QUAD *quad, int iq, void *ud)

Evaluate the function at `quad->lambda[iq]`. This looks slightly more complicated than the simple `FCT_AT_X` types, but passing the `EL_INFO` descriptor along with quadrature rule opens the door to implement even complicated functions in an efficient and simpler way than is possible with the simple `FCT_AT_X` types. See also Example 4.7.5.

Parameters

el_info The current `EL_INFO` descriptor.

quad The quadrature rule storing the evaluation points.

iq The number of the evaluation point.

ud Application data pointer.

Return Value The function value.

---

const REAL *GRD_LOC_FCT_AT_QP(

REAL_D res, const EL_INFO *el_info, const QUAD *quad, int iq, void *ud)

const REAL *LOC_FCT_D_AT_QP(

REAL_D res, const EL_INFO *el_info, const QUAD *quad, int iq, void *ud)

const REAL_D *GRD_LOC_FCT_D_AT_QP(

REAL_DD res, const EL_INFO *el_info, const QUAD *quad, int iq, void *ud)

More or less self-explanatory, the convention for the **res** argument are the same as for the `FCT_AT_X` types: a `NULL` pointer must be accepted, and then a pointer to a statically allocated storage area has to be returned, otherwise the result has to be stored in **res**, and the return value must be **res**, too.

4.6 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. ALBERTA supplies functions to calculate the error in several norms. For test purposes, the integral error routines may be used as "error estimators" in an adaptive method. The local element error $\int_S |\nabla(u-u_h)|^2$ or $\int_S |u-u_h|^2$ can be used as an error indicator and can be stored on the element leaf data, for example. ALBERTA provides also functions for the computation of the mean value of a given function, respectively the mean value difference of a given non-discrete function and a discrete function.

Not all variants of the functions listed below will be explained in detail further below. For the calling conventions for the application supplied function pointer we refer the reader to Section 4.5 on page 244.

```
REAL max_err_at_qp(FCT_AT_X u, const DOF_REAL_VEC *uh, const QUAD *quad);
REAL max_err_at_qp_loc(LOC_FCT_AT_QP u_loc, void *ud, FLAGS fill_flag,
                       const DOF_REAL_VEC *uh,
                       const QUAD *quad);
REAL max_err_dow_at_qp(FCT_D_AT_X u,
                       const DOF_REAL_VEC_D *uh,
                       const QUAD *quad);
REAL max_err_dow_at_qp_loc(LOC_FCT_D_AT_QP u_loc,
                           void *ud, FLAGS fill_flag,
                           const DOF_REAL_VEC_D *uh,
                           const QUAD *quad);

REAL max_err_at_vert(FCT_AT_X u, const DOF_REAL_VEC *uh);
REAL max_err_at_vert_loc(LOC_FCT_AT_QP u_at_qp,
                         void *ud, FLAGS fill_flag,
                         const DOF_REAL_VEC *uh);
REAL max_err_dow_at_vert(FCT_D_AT_X u, const DOF_REAL_VEC_D *uh);
REAL max_err_dow_at_vert_loc(LOC_FCT_D_AT_QP u_at_qp,
                             void *ud, FLAGS fill_flag,
                             const DOF_REAL_VEC_D *uh);

REAL L2_err(FCT_AT_X u, const DOF_REAL_VEC *uh,
            const QUAD *quad,
            bool rel_err, bool mean_value_adjust,
            REAL *(*rw_err_el)(EL *el), REAL *max_l2_err2);
REAL L2_err_loc(LOC_FCT_AT_QP u_loc, void *ud, FLAGS fill_flag,
                const DOF_REAL_VEC *uh,
                const QUAD *quad,
                bool rel_err, bool mean_value_adjust,
                REAL *(*rw_err_el)(EL *el), REAL *max_l2_err2);
REAL L2_err_weighted(FCT_AT_X weight, FCT_AT_X u, const DOF_REAL_VEC *uh,
                     const QUAD *quad,
                     bool rel_err, bool mean_value_adjust,
                     REAL *(*rw_err_el)(EL *el), REAL *max_l2_err2);
REAL L2_err_dow(FCT_D_AT_X u,
                const DOF_REAL_VEC_D *uh,
                const QUAD *quad,
                bool rel_err, bool mean_value_adjust,
                REAL *(*rw_err_el)(EL *el), REAL *max_l2_err2);
REAL L2_err_loc_dow(LOC_FCT_D_AT_QP u_loc,
                    void *ud, FLAGS fill_flag,
                    const DOF_REAL_VEC_D *uh,
```

```
                        const QUAD *quad,
                        bool rel_err, bool mean_value_adjust,
                        REAL *(*rw_err_el)(EL *el), REAL *max_l2_err2);
REAL L2_err_dow_weighted(FCT_AT_X weight, FCT_D_AT_X u,
                         const DOF_REAL_VEC_D *uh,
                         const QUAD *quad,
                         bool rel_err, bool mean_value_adjust,
                         REAL *(*rw_err_el)(EL *el), REAL *max_l2_err2);

REAL H1_err(GRD_FCT_AT_X grd_u, const DOF_REAL_VEC *uh,
            const QUAD *quad, bool rel_err, REAL *(*rw_err_el)(EL *),
            REAL *max_el_err2);
REAL H1_err_loc(GRD_LOC_FCT_AT_QP grd_u_loc,
                void *ud, FLAGS fill_flag,
                const DOF_REAL_VEC *uh,
                const QUAD *quad, bool rel_err, REAL *(*rw_err_el)(EL *),
                REAL *max_el_err2);
REAL H1_err_weighted(FCT_AT_X weight, GRD_FCT_AT_X grd_u,
                     const DOF_REAL_VEC *uh, const QUAD *quad,
                     bool rel_err, REAL *(*rw_err_el)(EL *),
                     REAL *max_el_err2);
REAL H1_err_dow(GRD_FCT_D_AT_X grd_u, const DOF_REAL_VEC_D *uh,
                const QUAD *quad,
                bool rel_err, REAL *(*rw_err_el)(EL *), REAL *max_el_err2);
REAL H1_err_loc_dow(GRD_LOC_FCT_D_AT_QP grd_u_loc, void *ud, FLAGS fill_flag,
                    const DOF_REAL_VEC_D *uh, const QUAD *quad,
                    bool rel_err,
                    REAL *(*rw_err_el)(EL *), REAL *max_el_err2);
REAL H1_err_dow_weighted(FCT_AT_X weight, GRD_FCT_D_AT_X grd_u,
                         const DOF_REAL_VEC_D *uh,
                         const QUAD *quad,
                         bool rel_err, REAL *(*rw_err_el)(EL *),
                         REAL *max_el_err2);

REAL mean_value(MESH *mesh, REAL (*f)(const REAL_D), const DOF_REAL_VEC *fh,
                const QUAD *quad);
REAL mean_value_loc(MESH *mesh, LOC_FCT_AT_QP f_at_qp,
                    void *ud, FLAGS fill_flags,
                    const DOF_REAL_VEC *fh, const QUAD *quad);
const REAL *mean_value_dow(MESH *mesh, FCT_D_AT_X f, const DOF_REAL_VEC_D
    *fh,
                           const QUAD *quad, REAL_D mean);
const REAL *mean_value_loc_dow(REAL_D mean, MESH *mesh,
                               LOC_FCT_D_AT_QP f_at_qp,
                               void *ud, FLAGS fill_flag,
                               const DOF_REAL_VEC_D *fh,
                               const QUAD *quad);
```

Descriptions

`max_err_at_qp(u, uh, quad)` the function returns the maximal error, $\max|u - u_h|$, 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 `NULL`, 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 absolute error $(\int_\Omega |\nabla(u-u_h)|^2)^{1/2}$ (`rel_err == 0`) or relative error $(\int_\Omega |\nabla(u-u_h)|^2 / \int_\Omega |\nabla u|^2)^{1/2}$ (`rel_err == 1`) 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 `NULL`, a quadrature which is exact of degree `2*uh->fe_space->bas_fcts->degree-2` is used;

if `rw_el_err` is not `NULL`, the return value of `(*rw_el_err)(el)` provides for each mesh element `el` an address where the local error is stored; if `max` is not `NULL`, `*max` is the maximal local error on an element on output.

`L2_err(u, uh, quad, rel_err, rw_el_err, max)` the function returns an approximation to the absolute error $(\int_\Omega |u-u_h|^2)^{1/2}$ (`rel_err == 0`) or the relative error $(\int_\Omega |u-u_h|^2 / \int_\Omega |u|^2)^{1/2}$ (`rel_err == 1`) 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 `NULL`, a quadrature which is exact of degree `2*uh->fe_space->bas_fcts->degree-2` is used;

if `rw_el_err` is not `NULL`, the return value of `(*rw_el_err)(el)` provides for each mesh element `el` an address where the local error is stored; if `max` is not `NULL`, `*max` is the maximal local error on an element on output.

`max_err_at_qp_[d|dow](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 `NULL`, a quadrature which is exact of degree `2*uh_d->fe_space->bas_fcts->degree-2` is used.

`H1_err2_[d|dow](grd_u_d, uh_d, quad, rel_err, rw_el_err, max)` the function returns an approximation to the absolute error (`rel_err == 0`) or relative error (`rel_err == 1`) 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 `NULL`, a quadrature which is exact of degree `2*uh_d->fe_space->bas_fcts->degree-2` is used;

if `rw_el_err` is not `NULL`, the return value of `(*rw_el_err)(el)` provides for each mesh element `el` an address where the local error is stored; if `max` is not `NULL`, `*max` is the maximal local error on an element on output.

`L2_err2_[d|dow](u_d, uh_d, quad, rel_err, rw_el_err, max)` the function returns an approximation to the absolute error (`rel_err == 0`) or relative error (`rel_err == 1`) 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_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 `NULL`, a quadrature which is exact of degree `2*uh_d->fe_space->bas_fcts->degree-2` is used;
if `rw_el_err` is not `NULL`, the return value of `(*rw_el_err)(el)` provides for each mesh element `el` an address where the local error is stored; if `max` is not `NULL`, `*max` is the maximal local error on an element on output.

`mean_value(mesh, f, fh, quad)`

`mean_value_[d|dow](mesh, f, fh, quad, mean)`

`mean_value_loc(mesh, f_at_qp, ud, fill_flags, fh, quad)`

`mean_value_loc_[d|dow](mean, mesh, f_at_qp, ud, fill_flags, fh, quad)`
compute the mean value of either a finite element function or a non-discrete function. If both are given return the difference of their mean values `(f-fh)`.

4.7 Tools for the assemblage of linear systems

This section describes data structures and subroutines for matrix and vector assembly. Section 4.7.1 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 4.7.2. This includes library routines for the efficient implementation of a general second order linear elliptic operator. Section 4.7.5 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 4.7.6. The incorporation of Dirichlet boundary values into the right hand side is presented in Section 4.7.7.1. Finally, routines for generation of interpolation coefficients are described in Section 4.7.8.

4.7.1 Element matrices and vectors

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 on each element S in (1.4) on page 18. It 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 3.5.1.

The updating of the system matrix is not that easy. As mentioned in Section 1.4.8, the system matrix is usually sparse and we use special data structures for storing these matrices, compare Section 3.3.4. 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, the exact 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_dof_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 129. During the updating of a matrix for the value corresponding to 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 functions 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.

4.7.1 Compatibility Note. *Previous versions of ALBERTA defined extra-types for vector-valued problems, like* `DOF_DOWB_MATRIX`*,* `DOWB_OPERATOR_INFO` *etc. The "*`DOWB`*" ("DimOfWorldBlocks") variants, however, already incorporated all the functionality of the ordinary scalar-only versions. Therefore the scalar-ony versions of most data-structures have been abandoned and were replaced by the "*`DOWB`*" variants, which in turn were renamed to use the scalar-only names. For example, in the current implementation a* `DOF_MATRIX` *is in fact what older versions called a* `DOF_DOWB_MATRIX`*; and implements the scalar-only case as well as the block-matrix case.*

4.7.1.1 Element matrix and vector structures

Block-matrix types

```
typedef enum matent_type {
  MATENT_NONE    = -1,
  MATENT_REAL    =  0,
  MATENT_REAL_D  =  1,
  MATENT_REAL_DD =  2
} MATENT_TYPE;
```

Description: This enumeration type defines symbolic types for block-matrix entries. `MATENT_REAL` means scalar blocks, `MATENT_REAL_D` stands for diagonal blocks, and `MATENT_REAL_DD` is a code for full matrix blocks. In general, data-structures make use of these types to store the matrix blocks in an efficient way.

Structure for element matrices

```
typedef struct el_matrix EL_MATRIX;

struct el_matrix
{
  MATENT_TYPE type;
  int n_row, n_col;
  int n_row_max, n_col_max;
  union {
    REAL    *const*real;
    REAL_D  *const*real_d;
    REAL_DD *const*real_dd;
  } data;
  DBL_LIST_NODE row_chain;
  DBL_LIST_NODE col_chain;
};
```

Description: A data structure to store per-element contributions during the assembling of discrete systems. There is some limited support for the operation of element-matrices on element-vectors and global DOF-vectors, see Section 4.7.1.4.

type One out of MATENT_REAL, MATENT_REAL_D or MATENT_REAL_DD. The entries stored in EL_MATRIX->data have to be interpreted accordingly. See MATENT_TYPE on page 252.

n_row is the number of rows of the element matrix

n_col is the number of columns of the element matrix

n_row_max is the maximal number of rows. The number of rows can vary from element to element if the underlying basis functions have a per-element initializer.

n_col_max is the maximal number of columns.

data, data.real, data.real_d, data.real_dd EL_MATRIX->data is a union, its components should be accessed according to the symmetry type indicated by EL_MATRIX->type.

row_chain, col_chain If the underlying finite element spaces are a direct sum of function spaces, then the resulting element matrices have a block-layout. The link to the other parts of the resulting block-matrix is implemented using cyclic doubly linked lists, row_chain and col_chain are the corresponding list-nodes. There is a separate section explaining how to handle such chains of objects, see Section 3.7.

Structures for element vectors

```
typedef struct el_int_vec        EL_INT_VEC;
typedef struct el_dof_vec        EL_DOF_VEC;
typedef struct el_uchar_vec      EL_UCHAR_VEC;
typedef struct el_schar_vec      EL_SCHAR_VEC;
typedef struct el_bndry_vec      EL_BNDRY_VEC;
typedef struct el_ptr_vec        EL_PTR_VEC;
typedef struct el_real_vec       EL_REAL_VEC;
typedef struct el_real_vec_d     EL_REAL_VEC_D;
typedef struct el_real_d_vec     EL_REAL_D_VEC;
```

The el_*_vec structures are declared similary, the only difference between them is the type of the structure entry vec. Below, the EL_REAL_VEC structure is given:

Source Code Listing 4.38: data-type: EL_REAL_VEC

```
struct el_real_vec
{
  int             n_components;
  int             n_components_max;
  DBL_LIST_NODE   chain;
  int             reserved;
  REAL            vec[1]; /* different type in EL_INT_VEC, ... */
};
```

and the EL_REAL_VEC_D structure is described in detail:

```
struct el_real_vec_d
{
  int             n_components;
  int             n_components_max;
  DBL_LIST_NODE   chain;
```

```
  int             stride; /* either 1 or DIM_OF_WORLD */
  REAL            vec[1];
};
```

Description:

n_components The actual number of components available in and following EL_XXX_VEC->vec. Note that the actual number of components is – of course – larger than 1 in general get_el_XXX_vec(bas_fcts) takes care of allocating enough space.

n_components_max Behing EL_XXX_VEC->vec[0] may actually be more space available than the number of currently valid entries as indicated by EL_XXX_VEC->n_components; this is the maximum size to access without crossing the bounds of the data segment allocated for this element vector.

chain If the underlying basis-function implementation is part of a chain of sets of basis functions, then this structure is inherited also by the element vectors: they are chained using a doubly linked list, chain is the corresponding list-node. There is a separate section about such chained objects, see Section 3.7.

stride, reserved For element vectors other than an EL_REAL_VEC_D this is a reserved value and actually tied to the constant value 1 with the exception of a EL_REAL_D_VEC were reserved is fixed at DIM_OF_WORLD. For EL_REAL_VEC_D this varies, based on the dimension of the range of the underlying basis function implementation. For vector-valued basis functions EL_REAL_VEC_D->stride is again tied to 1, for scalar-valued basis functions EL_REAL_VEC_D->stride is fixed at DIM_OF_WORLD, in both cases it gives the number of REAL's belonging to a single DOF. See also DOF_REAL_VEC_D on page 122.

vec[1] Start of the data-segment, EL_XXX_VEC->n_components items contain valid data, EL_XXX_VEC->n_components_max items are allocated. Note that for a EL_REAL_VEC_D vector the numbers have to be multiplied by EL_REAL_VEC_D->stride to get the actual number of REAL's allocated.

4.7.1.2 Accumulating per-element contributions

The following functions can be used on elements for updating matrices and vectors.

```
void add_element_matrix(DOF_MATRIX *matrix, REAL factor,
                        const EL_MATRIX *el_matrix, MatrixTranspose
                            transpose,
                        const EL_DOF_VEC *row_dof, const EL_DOF_VEC *col_dof,
                        const EL_SCHAR_VEC *bound);
void add_element_vec(DOF_REAL_VEC *drv, REAL factor, const EL_REAL_VEC
    *el_vec,
                     const EL_DOF_VEC *dof, const EL_SCHAR_VEC *bound);
void add_element_d_vec(DOF_REAL_D_VEC *drdv, REAL factor,
                       const EL_REAL_D_VEC *el_vec, const EL_DOF_VEC *dof,
                       const EL_SCHAR_VEC *bound);
void add_element_vec_dow(DOF_REAL_VEC_D *drdv, REAL factor,
                         const EL_REAL_VEC_D *el_vec, const EL_DOF_VEC *dof,
                         const EL_SCHAR_VEC *bound);
```

Descriptions

add_element_matrix(mat, factor, el_mat, transpose, row_dof, col_dof, bound)

Updates the global DOF_MATRIX mat by adding element contributions. If row_dof equals col_dof, 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, for example. In general, add_element_matrix() does the following: for all i the values fac*el_mat->data.{REAL,REAL_D,REAL_DD}[i][j] are added to the entries at the position (row_dof->vec[i],col_dof->vec[j]) in the global matrix mat ($0 \leq$ i $<$ el_mat->n_row, $0 \leq$ j $<$ el_mat->n_col). If such an entry exists in the row number row_dof->vec[i] the global matrix mat the value is simply 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 imply an enlargement of the row by adding a new MATRIX_ROW structure to the list of matrix rows.

Note that the first element matrix added to mat after calling clear_dof_matrix() determines the block-type of the global matrix mat. It is possible to add element-matrices with higher block-symmetry to global DOF_MATRIXes with lower block-symmetry, for example it is allowed to add el_mat to mat if el_mat->type == MATENT_REAL and mat->type == MATENT_REAL_DD.

Parameters

mat the global DOF_MATRIX.

factor is a multiplier for the element contributions; usually factor is 1 or -1;

el_mat is a matrix of size n_row × n_col storing the element contributions;

transpose the original matrix is used if transpose == NoTranspose (= 0) and the transposed matrix if transpose == Transpose (= 1);

row_dof is a vector of length row_dof->n_components storing the global row indices;

col_dof is a vector of length col_dof->n_components storing the global column indices, col_dof may be a NULL 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;

bound is either NULL or an EL_SCHAR_Vec stucture storing a vector of length bound->n_components. In this case bound->n_components must match either row_dof->n_components or col_dof->n_components, depending on the value of transpose.

If bound->vec[i] >= DIRICHLET, then the following happens:

row_dof == col_dof In the global matrix the row row_dof->vec[i] is cleared to zero, with the exception of the diagonal entry, which is set to 1.0.

row_dof != col_dof In the global matrix the row row_dof->vec[i] is cleared to zero.

All other contributions of el_mat are added to matrix as usual. This allows for a convenient way to implement inhomogeneous Dirichlet boundary conditions, without having to modify the right-hand-side of the discrete systems explicitly.

---

add_element_vec(drv, factor, el_vec, dof, bound)

add_element_d_vec(drv, factor, el_vec, dof, bound)

add_element_vec_d(drv, factor, el_vec, dof, bound)

These do similar things as `add_element_matrix()`, but with element vectors. Section 4.7.1.4 also lists other routines which might be helpful in this context.

4.7.1.3 Allocation and filling of element vectors

Prototypes

```
EL_DOF_VEC *get_dof_indices(EL_DOF_VEC *dofs, const FE_SPACE *fe_space,
                            const EL *el);
EL_BNDRY_VEC *get_bound(EL_BNDRY_VEC *bndry, const BAS_FCTS *bas_fcts,
                        const EL_INFO *el_info);
void el_interpol(EL_REAL_VEC *coeff, const EL_INFO *el_info, int wall,
                 const EL_INT_VEC *indices, LOC_FCT_AT_QP f, void *ud,
                 const BAS_FCTS *bas_fcts);
void el_interpol_dow(EL_REAL_VEC_D *coeff, const EL_INFO *el_info, int wall,
                     const EL_INT_VEC *indices, LOC_FCT_D_AT_QP f,
                     void *f_data, const BAS_FCTS *bas_fcts);
void dirichlet_map(EL_SCHAR_VEC *bound, const EL_BNDRY_VEC *bndry_bits,
                   const BNDRY_FLAGS mask);

const EL_INT_VEC *
fill_el_int_vec(EL_INT_VEC *el_vec, EL *el, const DOF_INT_VEC *dof_vec);
const EL_REAL_VEC *
fill_el_real_vec(EL_REAL_VEC *el_vec, EL *el, const DOF_REAL_VEC *dof_vec);
const EL_REAL_D_VEC *
fill_el_real_d_vec(EL_REAL_D_VEC *el_vec, EL *el, const DOF_REAL_D_VEC
    *dof_vec);
const EL_REAL_VEC_D *
fill_el_real_vec_d(EL_REAL_VEC_D *el_vec, EL *el, const DOF_REAL_VEC_D
    *dof_vec);
const EL_UCHAR_VEC *
fill_el_uchar_vec(EL_UCHAR_VEC *el_vec, EL *el, const DOF_UCHAR_VEC
    *dof_vec);
const EL_SCHAR_VEC *
fill_el_schar_vec(EL_SCHAR_VEC *el_vec, EL *el, const DOF_SCHAR_VEC
    *dof_vec);

EL_INT_VEC *get_el_int_vec(const BAS_FCTS *bas_fcts);
EL_DOF_VEC *get_el_dof_vec(const BAS_FCTS *bas_fcts);
EL_UCHAR_VEC *get_el_uchar_vec(const BAS_FCTS *bas_fcts);
EL_SCHAR_VEC *get_el_schar_vec(const BAS_FCTS *bas_fcts);
EL_BNDRY_VEC *get_el_bndry_vec(const BAS_FCTS *bas_fcts);
EL_PTR_VEC *get_el_ptr_vec(const BAS_FCTS *bas_fcts);
EL_REAL_VEC *get_el_real_vec(const BAS_FCTS *bas_fcts);
EL_REAL_D_VEC *get_el_real_d_vec(const BAS_FCTS *bas_fcts);
EL_REAL_VEC_D *get_el_real_vec_d(const BAS_FCTS *bas_fcts);

void free_el_int_vec(EL_INT_VEC *el_vec);
void free_el_dof_vec(EL_DOF_VEC *el_vec);
void free_el_uchar_vec(EL_UCHAR_VEC *el_vec);
void free_el_schar_vec(EL_SCHAR_VEC *el_vec);
void free_el_bndry_vec(EL_BNDRY_VEC *el_vec);
void free_el_ptr_vec(EL_PTR_VEC *el_vec);
void free_el_real_vec(EL_REAL_VEC *el_vec);
```

```
void free_el_real_d_vec(EL_REAL_D_VEC *el_vec);
void free_el_real_vec_d(EL_REAL_VEC_D *el_vec);

DEF_EL_VEC_VAR(VECNAME, name, _size, _size_max, _init);
DEF_EL_VEC_CONST(VECNAME, name, _size, _size_max);
ALLOC_EL_VEC(VECNAME, _size, _size_max);
```

Descriptions

`get_dof_indices(dofs, fe_space, el)` Compute the mapping between the local `DOF`-indices on `el` and the global `DOF`-indices according to `fe_space->admin`.

Parameters

`dofs` Storage for the result or `NULL`. In the latter case the mapping is returned in a statically allocated `EL_DOF_VEC`. *Note: this storage area will be overwritten on the next call to this function, even if the* `fe_space` *argument differs.*

`fe_space` The finite element space to compute the mapping for.

`el` The current mesh element (*not* the current `EL_INFO` pointer, use `EL_INFO->el`).

return Either again the argument `dofs` or – if `dofs == NULL` – a pointer to a statically allocated `EL_DOF_VEC`.

examples With pre-allocated `EL_DOF_VEC`:

```
EL_DOF_VEC *dofs = get_el_dof_vec(fe_space->bas_fcts);
TRAVERSE_FIRST(mesh, -1, CALL_LEAF_EL) {
  int i;

  get_dof_indices(dofs, fe_space, el_info->el);
  for (i = 0; i < bas_fcts->n_bas_fcts; i++) {
    MSG("dofs[%d] = %d\n", dofs->vec[i]);
  }
} TRAVERSE_NEXT();
free_el_dof_vec(dofs);
```

Without pre-allocated `EL_DOF_VEC`:

```
TRAVERSE_FIRST(mesh, -1, CALL_LEAF_EL) {
  int i;
  EL_DOF_VEC *dofs = get_dof_indices(NULL, fe_space, el_info->el);

  for (i = 0; i < bas_fcts->n_bas_fcts; i++) {
    MSG("dofs[%d] = %d\n", dofs->vec[i]);
  }
} TRAVERSE_NEXT();
```

`get_bound(bndry, bas_fcts, el_info)` Extract the boundary types of the local DOFs of `bas_fcts`. The boundary types are returned in form of a bit-mask. If bit `j` in the bit-mask `bndry[i]` is set, then the local DOF number `i` belongs to the boundary segment which has been assigned the number `j` in the macro-triangulation. Boundary types range from 1 to 255.

Parameters

EL_BNDRY_VEC *bndry Storage for the result or NULL. In the latter case the data is returned in a statically allocated EL_BNDRY_VEC.

BAS_FCTS *bas_fcts The local basis functions.

const EL_INFO *el_info The current mesh element info structure. (*not* the current EL_INFO pointer.

return Either again the argument bndry or – if bndry == NULL – a pointer to a statically allocated EL_BNDRY_VEC.

examples With pre-allocated EL_BNDRY_VEC:

```
EL_BNDRY_VEC *bndry = get_el_bndry_vec(bas_fcts);
TRAVERSE_FIRST(mesh, -1, CALL_LEAF_EL|FILL_BOUND) {
  int i, j;
  get_bound(bndry, bas_fcts, el_info);
  for (i = 0; i < bas_fcts->n_bas_fcts; i++) {
    for (j = 1; j < N_BNDRY_TYPES; j++) {
      if (BNNDRY_FLAGS_IS_INTERIOR(bndry->vec[i])) {
        MSG("Local dof %d is an interior DOF\n");
      } else if (BNDRY_FLAGS_IS_AT_BNDRY(bndry->vec[i], j)) {
        MSG("Local dof %d belongs to boundary segment %d\n", i, j);
      }
    }
  }
} TRAVERSE_NEXT();
free_el_bndry_vec(bndry);
```

Without pre-allocated EL_BNDRY_VEC:

```
TRAVERSE_FIRST(mesh, -1, CALL_LEAF_EL) {
  int i,j;
  EL_BNDRY_VEC *bndry = get_bound(NULL, bas_fcts, el_info);

  for (i = 0; i < bas_fcts->n_bas_fcts; i++) {
    for (j = 1; j < N_BNDRY_TYPES; j++) {
      if (BNNDRY_FLAGS_IS_INTERIOR(bndry->vec[i])) {
        MSG("Local dof %d is an interior DOF\n");
      } else if (BNDRY_FLAGS_IS_AT_BNDRY(bndry->vec[i], j)) {
        MSG("Local dof %d belongs to boundary segment %d\n", i, j);
      }
    }
  }
} TRAVERSE_NEXT();
```

---

fill_el_int_vec(el_vec, el, dof_vec)

fill_el_real_vec(el_vec, el, dof_vec)

fill_el_real_d_vec(el_vec, el, dof_vec)

fill_el_real_vec_d(el_vec, el, dof_vec)

fill_el_uchar_vec(el_vec, el, dof_vec)

fill_el_schar_vec(el_vec, el, dof_vec)

Fill the respective element vector with data. The description below is for `fill_el_real_vec()`, the other versions work similar.

Parameters

EL_REAL_VEC *el_vec Storage for the result or `NULL`. In the latter case the return value is `DOF_REAL_VEC->vec_loc`; the data will be overwritten on the next call to `fill_el_real_vec()` with the same `dof_vec` argument. Calling `fill_el_real_vec()` with *other* DOF-vectors will *not* invalidate the data.

EL *el The current mesh element (*not* the current `EL_INFO` pointer, use `EL_INFO->el`).

DOF_REAL_VEC *dof_vec The global DOF-vector to extract the data from.

return Either again a the pointer `el_vec` or – if `el_vec == NULL` a pointer to a statically allocated result space which will be overwritten on the next call to `fill_el_real_vec()`. *Warning:* see "bugs" below.

---

get_el_int_vec(bas_fcts)

get_el_dof_vec(bas_fcts)

get_el_uchar_vec(bas_fcts)

get_el_schar_vec(bas_fcts)

get_el_bndry_vec(bas_fcts)

get_el_ptr_vec(bas_fcts)

get_el_real_vec(bas_fcts)

get_el_real_d_vec(bas_fcts)

get_el_real_vec_d(bas_fcts)

The `get_el_*_vec()` routines automatically allocates enough memory for the element data vector `vec` as indicated by `bas_fcts->n_bas_fcts`.

Parameters **const BAS_FCTS *bas_fcts**

return A pointer to a dynamically allocated element vector of the respective type.

examples See the first example for the `fill_el_real_vec()` function.

---

free_el_int_vec(el_vec)

free_el_dof_vec(el_vec)

free_el_uchar_vec(el_vec)

free_el_schar_vec(el_vec)

free_el_bndry_vec(el_vec)

free_el_ptr_vec(el_fcts)

free_el_real_vec(bas_fcts)

free_el_real_d_vec(bas_fcts)

free_el_real_vec_d(bas_fcts)

The `free_el_XXX_vec()` routines free all previously allocated storage for `el_XXX_vec` data.

Parameters `const BAS_FCTS *bas_fcts`

return `void`

examples See the first example for the `fill_el_real_vec()` function.

`DEF_EL_VEC_VAR(VECNAME, name, size, size_max, init)`
This is a macro which defines a (local) variable with id `name`, pointing to an `EL_VECNAME_VEC` of size `size`, holding a maximal number of elements `max_size`, which is initialised if `init` is `true`. `size` and `size_max` may be variables.

`DEF_EL_VEC_CONST(VECNAME, name, size, size_max)`
This is a macro which defines a (local) variable with id `name`, pointing to an `EL_VECNAME_VEC` of size `size`, holding a maximal number of elements `max_size`. `size` and `size_max` must be constant values.

`ALLOC_EL_VEC(VECNAME, size, size_max)` This macro allocates a `EL_VECNAME_VEC` with enough storage to hold `size_max` elements; the `n_components` component of the element vector structure is set to `size`.

`el_interpol(coeff, el_info, wall, indices, f, ud, bas_fcts)`
`el_interpol_dow(coeff, el_info, wall, indices, f, f_data, ud, bas_fcts)`
`dirichlet_map(bound, bndry_bits, mask)`

4.7.1.4 BLAS-like Element-matrix and -vector operations

The source code listing below lists the proto-types, refer to Table 4.1 and Table 4.2 for a description of the respective operations. The routines in Table 4.2 take an argument

```
const EL_SCHAR_VEC *bound.
```

In this case the operations will act only on the rows r which are not masked-out by `bound->vec[r] >= DIRICHLET`. The `bound` argument maybe `NULL`.

```
EL_REAL_VEC *el_bi_mat_vec(REAL a, const EL_MATRIX *A,
                           REAL b, const EL_MATRIX *B,
                           const EL_REAL_VEC *u_h,
                           REAL c, EL_REAL_VEC *f_h);
EL_REAL_D_VEC *el_bi_mat_vec_d(REAL a, const EL_MATRIX *A,
                               REAL b, const EL_MATRIX *B,
                               const EL_REAL_D_VEC *u_h,
                               REAL c, EL_REAL_D_VEC *f_h);
EL_REAL_VEC_D *el_bi_mat_vec_dow(REAL a, const EL_MATRIX *A,
                                 REAL b, const EL_MATRIX *B,
                                 const EL_REAL_VEC_D *u_h,
                                 REAL c, EL_REAL_VEC_D *f_h);
EL_REAL_VEC *el_bi_mat_vec_rrd(REAL a, const EL_MATRIX *A,
                               REAL b, const EL_MATRIX *B,
                               const EL_REAL_D_VEC *u_h,
                               REAL c, EL_REAL_VEC *f_h);
```

```
EL_REAL_VEC *el_bi_mat_vec_scl_dow(REAL a, const EL_MATRIX *A,
                                   REAL b, const EL_MATRIX *B,
                                   const EL_REAL_VEC_D *u_h,
                                   REAL c, EL_REAL_VEC *f_h);
EL_REAL_D_VEC *el_bi_mat_vec_rdr(REAL a, const EL_MATRIX *A,
                                 REAL b, const EL_MATRIX *B,
                                 const EL_REAL_VEC *u_h,
                                 REAL c, EL_REAL_D_VEC *f_h);
EL_REAL_VEC_D *el_bi_mat_vec_dow_scl(REAL a, const EL_MATRIX *A,
                                     REAL b, const EL_MATRIX *B,
                                     const EL_REAL_VEC *u_h,
                                     REAL c, EL_REAL_VEC_D *f_h);

EL_REAL_VEC *el_gen_mat_vec(REAL a, const EL_MATRIX *A,
                            const EL_REAL_VEC *u_h,
                            REAL b, EL_REAL_VEC *f_h);
EL_REAL_D_VEC *el_gen_mat_vec_d(REAL a, const EL_MATRIX *A,
                                const EL_REAL_D_VEC *u_h,
                                REAL b, EL_REAL_D_VEC *f_h);
EL_REAL_VEC_D *el_gen_mat_vec_dow(REAL a, const EL_MATRIX *A,
                                  const EL_REAL_VEC_D *u_h,
                                  REAL b, EL_REAL_VEC_D *f_h);
EL_REAL_VEC *el_gen_mat_vec_rrd(REAL a, const EL_MATRIX *A,
                                const EL_REAL_D_VEC *u_h,
                                REAL b, EL_REAL_VEC *f_h);
EL_REAL_VEC *el_gen_mat_vec_scl_dow(REAL a, const EL_MATRIX *A,
                                    const EL_REAL_VEC_D *u_h,
                                    REAL b, EL_REAL_VEC *f_h);
EL_REAL_D_VEC *el_gen_mat_vec_rdr(REAL a, const EL_MATRIX *A,
                                  const EL_REAL_VEC *u_h,
                                  REAL b, EL_REAL_D_VEC *f_h);
EL_REAL_VEC_D *el_gen_mat_vec_dow_scl(REAL a, const EL_MATRIX *A,
                                      const EL_REAL_VEC *u_h,
                                      REAL b, EL_REAL_VEC_D *f_h);

EL_REAL_VEC *el_mat_vec(REAL a, const EL_MATRIX *A,
                        const EL_REAL_VEC *u_h, EL_REAL_VEC *f_h);
EL_REAL_D_VEC *el_mat_vec_d(REAL a, const EL_MATRIX *A,
                            const EL_REAL_D_VEC *u_h, EL_REAL_D_VEC *f_h);
EL_REAL_VEC_D *el_mat_vec_dow(REAL a, const EL_MATRIX *A,
                              const EL_REAL_VEC_D *u_h, EL_REAL_VEC_D *f_h);
EL_REAL_VEC *el_mat_vec_rrd(REAL a, const EL_MATRIX *A,
                            const EL_REAL_D_VEC *u_h, EL_REAL_VEC *f_h);
EL_REAL_VEC *el_mat_vec_scl_dow(REAL a, const EL_MATRIX *A,
                                const EL_REAL_VEC_D *u_h, EL_REAL_VEC *f_h);
EL_REAL_D_VEC *el_mat_vec_rdr(REAL a, const EL_MATRIX *A,
                              const EL_REAL_VEC *u_h, EL_REAL_D_VEC *f_h);
EL_REAL_VEC_D *el_mat_vec_dow_scl(REAL a, const EL_MATRIX *A,
                                  const EL_REAL_VEC *u_h, EL_REAL_VEC_D
                                      *f_h);

void bi_mat_el_vec(REAL a, const EL_MATRIX *A,
                   REAL b, const EL_MATRIX *B,
                   const EL_REAL_VEC *u_h, REAL c, DOF_REAL_VEC *f_h,
                   const EL_DOF_VEC *dof, const EL_SCHAR_VEC *bound);
void bi_mat_el_vec_d(REAL a, const EL_MATRIX *A,
                     REAL b, const EL_MATRIX *B,
```

```
                        const EL_REAL_D_VEC *u_h, REAL c, DOF_REAL_D_VEC *f_h,
                        const EL_DOF_VEC *dof, const EL_SCHAR_VEC *bound);
  void bi_mat_el_vec_dow(REAL a, const EL_MATRIX *A,
                         REAL b, const EL_MATRIX *B,
                         const EL_REAL_VEC_D *u_h, REAL c, DOF_REAL_VEC_D *f_h,
                         const EL_DOF_VEC *dof, const EL_SCHAR_VEC *bound);
  void bi_mat_el_vec_rrd(REAL a, const EL_MATRIX *A,
                         REAL b, const EL_MATRIX *B,
                         const EL_REAL_D_VEC *u_h, REAL c, DOF_REAL_VEC *f_h,
                         const EL_DOF_VEC *dof, const EL_SCHAR_VEC *bound);
  void bi_mat_el_vec_scl_dow(REAL a, const EL_MATRIX *A,
                             REAL b, const EL_MATRIX *B,
                             const EL_REAL_VEC_D *u_h, REAL c, DOF_REAL_VEC
                                 *f_h,
                             const EL_DOF_VEC *dof, const EL_SCHAR_VEC *bound);
  void bi_mat_el_vec_rdr(REAL a, const EL_MATRIX *A,
                         REAL b, const EL_MATRIX *B,
                         const EL_REAL_VEC *u_h, REAL c, DOF_REAL_D_VEC *f_h,
                         const EL_DOF_VEC *dof, const EL_SCHAR_VEC *bound);
  void bi_mat_el_vec_dow_scl(REAL a, const EL_MATRIX *A,
                             REAL b, const EL_MATRIX *B,
                             const EL_REAL_VEC *u_h, REAL c, DOF_REAL_VEC_D
                                 *f_h,
                             const EL_DOF_VEC *dof, const EL_SCHAR_VEC *bound);

  void gen_mat_el_vec(REAL a, const EL_MATRIX *A,
                      const EL_REAL_VEC *u_h, REAL b, DOF_REAL_VEC *f_h,
                      const EL_DOF_VEC *dof, const EL_SCHAR_VEC *bound);
  void gen_mat_el_vec_d(REAL a, const EL_MATRIX *A,
                        const EL_REAL_D_VEC *u_h, REAL b, DOF_REAL_D_VEC *f_h,
                        const EL_DOF_VEC *dof, const EL_SCHAR_VEC *bound);
  void gen_mat_el_vec_dow(REAL a, const EL_MATRIX *A,
                          const EL_REAL_VEC_D *u_h, REAL b, DOF_REAL_VEC_D
                              *f_h,
                          const EL_DOF_VEC *dof, const EL_SCHAR_VEC *bound);
  void gen_mat_el_vec_rrd(REAL a, const EL_MATRIX *A,
                          const EL_REAL_D_VEC *u_h, REAL b, DOF_REAL_VEC *f_h,
                          const EL_DOF_VEC *dof, const EL_SCHAR_VEC *bound);
  void gen_mat_el_vec_scl_dow(REAL a, const EL_MATRIX *A,
                              const EL_REAL_VEC_D *u_h, REAL b, DOF_REAL_VEC
                                  *f_h,
                              const EL_DOF_VEC *dof, const EL_SCHAR_VEC
                                  *bound);
  void gen_mat_el_vec_rdr(REAL a, const EL_MATRIX *A,
                          const EL_REAL_VEC *u_h, REAL b, DOF_REAL_D_VEC *f_h,
                          const EL_DOF_VEC *dof, const EL_SCHAR_VEC *bound);
  void gen_mat_el_vec_dow_scl(REAL a, const EL_MATRIX *A,
                              const EL_REAL_VEC *u_h, REAL b, DOF_REAL_VEC_D
                                  *f_h,
                              const EL_DOF_VEC *dof, const EL_SCHAR_VEC
                                  *bound);

  void mat_el_vec(REAL a, const EL_MATRIX *A,
                  const EL_REAL_VEC *u_h, DOF_REAL_VEC *f_h,
                  const EL_DOF_VEC *dof, const EL_SCHAR_VEC *bound);
  void mat_el_vec_d(REAL a, const EL_MATRIX *A,
                    const EL_REAL_D_VEC *u_h, DOF_REAL_D_VEC *f_h,
```

```
                    const EL_DOF_VEC *dof, const EL_SCHAR_VEC *bound);
void mat_el_vec_dow(REAL a, const EL_MATRIX *A,
                    const EL_REAL_VEC_D *u_h, DOF_REAL_VEC_D *f_h,
                    const EL_DOF_VEC *dof, const EL_SCHAR_VEC *bound);
void mat_el_vec_rrd(REAL a, const EL_MATRIX *A,
                    const EL_REAL_D_VEC *u_h, DOF_REAL_VEC *f_h,
                    const EL_DOF_VEC *dof, const EL_SCHAR_VEC *bound);
void mat_el_vec_scl_dow(REAL a, const EL_MATRIX *A,
                        const EL_REAL_VEC_D *u_h, DOF_REAL_VEC *f_h,
                        const EL_DOF_VEC *dof, const EL_SCHAR_VEC *bound);
void mat_el_vec_rdr(REAL a, const EL_MATRIX *A,
                    const EL_REAL_VEC *u_h, DOF_REAL_D_VEC *f_h,
                    const EL_DOF_VEC *dof, const EL_SCHAR_VEC *bound);
void mat_el_vec_dow_scl(REAL a, const EL_MATRIX *A,
                        const EL_REAL_VEC *u_h, DOF_REAL_VEC_D *f_h,
                        const EL_DOF_VEC *dof, const EL_SCHAR_VEC *bound);

EL_MATRIX *el_mat_set(REAL a, EL_MATRIX *result);
EL_MATRIX *el_mat_axey(REAL a, const EL_MATRIX *A, EL_MATRIX *result);
EL_MATRIX *el_mat_axpy(REAL a, const EL_MATRIX *A, EL_MATRIX *result);
EL_MATRIX *el_mat_axpby(REAL a, const EL_MATRIX *A,
                        REAL b, const EL_MATRIX *B, EL_MATRIX *result);
```

4.7.2 Data structures and functions for matrix assemblage

The following structure holds full information for the assembling of scalar 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
{
  const FE_SPACE *row_fe_space;
  const FE_SPACE *col_fe_space;

  MATENT_TYPE     krn_blk_type;

  BNDRY_FLAGS     dirichlet_bndry;
  REAL            factor;

  EL_MATRIX_FCT   el_matrix_fct;
  void            *fill_info;

  const EL_MATRIX_FCT *neigh_el_mat_fcts;
  void            *neigh_fill_info;

  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 columns DOFs of the resulting matrix.

`f = el_mat_vec(a, A, u, f)`
`f = el_mat_vec_d(a, A, u, f)`
`f = el_mat_vec_dow(a, A, u, f)`
`f = el_mat_vec_rrd(a, A, u, f)`
`f = el_mat_vec_scl_dow(a, A, u, f)`
`f = el_mat_vec_rdr(a, A, u, f)`
`f = el_mat_vec_dow_scl(a, A, u, f)`
	$f_i \leftarrow (a\,A\,u)_i$

`f = el_gen_mat_vec(a, A, u, b, f)`
`f = el_gen_mat_vec_d(a, A, u, b, f)`
`f = el_gen_mat_vec_dow(a, A, u, b, f)`
`f = el_gen_mat_vec_rrd(a, A, u, b, f)`
`f = el_gen_mat_vec_scl_dow(a, A, u, b, f)`
`f = el_gen_mat_vec_rdr(a, A, u, b, f)`
`f = el_gen_mat_vec_dow_scl(a, A, u, b, f)`
	$f_i \leftarrow (a\,A\,u + b\,f)_i$

`f = el_bi_mat_vec(a, A, b, B, u, c, f)`
`f = el_bi_mat_vec_d(a, A, b, B, u, c, f)`
`f = el_bi_mat_vec_dow(a, A, b, B, u, c, f)`
`f = el_bi_mat_vec_rrd(a, A, b, B, u, c, f)`
`f = el_bi_mat_vec_scl_dow(a, A, b, B, u, c, f)`
`f = el_bi_mat_vec_rdr(a, A, b, B, u, c, f)`
`f = el_bi_mat_vec_dow_scl(a, A, b, B, u, c, f)`
	$f_i \leftarrow ((a\,A + b\,B)\,u + c\,f)_i$

`A = el_mat_set(a, A)`
`B = el_mat_axey(a, A, B)`
`B = el_mat_axpy(a, A, B)`
`C = el_mat_axpby(a, A, b, B, C)`
	$A_{ij} \leftarrow a$
	$B_{ij} \leftarrow a\,A_{ij}$
	$B_{ij} \leftarrow a\,A_{ij} + B_{ij}$
	$C_{ij} \leftarrow a\,A_{ij} + b\,B_{ij}$

Table 4.1: BLAS-operations for element-vectors and -matrices. A and B denote element matrices, u and f element vectors, a, b, c are numbers.

`krn_blk_type` defines the block-matrix type of matrix entries

`dirichlet_boundary` bndry-type bit-mask for Dirichlet-boundary conditions built into the matrix

`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` × `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.

`neigh_el_mat_fcts` If the `BNDRY_OPERATOR_INFO` (code-listing 4.51) structure passed to `fill_matrix_info()` was flagged as discontinuous, then this is the base-address of an array storing `N_NEIGH(mesh->dim)` many element-matrix functions which pair the

`f = mat_el_vec(a, A, u, f, dof, mask)`
`mat_el_vec_d(a, A, u, f, dof, mask)`
`mat_el_vec_dow(a, A, u, f, dof, mask)`
`mat_el_vec_rrd(a, A, u, f, dof, mask)`
`mat_el_vec_scl_dow(a, A, u, f, dof, mask)`
`mat_el_vec_rdr(a, A, u, f, dof, mask)`
`mat_el_vec_dow_scl(a, A, u, f, dof, mask)`
	$f[dof[i]] \leftarrow (a\,A\,u)_i$
	if `mask[i] != DIRICHLET`
	or `mask == NULL`

`gen_mat_el_vec(a, A, u, b, f, dof, mask)`
`gen_mat_el_vec_d(a, A, u, b, f, dof, mask)`
`gen_mat_el_vec_dow(a, A, u, b, f, dof, mask)`
`gen_mat_el_vec_rrd(a, A, u, b, f, dof, mask)`
`gen_mat_el_vec_scl_dow(a, A, u, b, f, dof, mask)`
`gen_mat_el_vec_rdr(a, A, u, b, f, dof, mask)`
`gen_mat_el_vec_dow_scl(a, A, u, b, f, dof, mask)`
	$f[dof[i]] \leftarrow (a\,A\,u)_i + b\,f[dof[i]]$
	if `mask[i] != DIRICHLET`
	or `mask == NULL`

`bi_mat_el_vec(a, A, b, B, u, c, f, dof, mask)`
`bi_mat_el_vec_d(a, A, b, B, u, c, f, dof, mask)`
`bi_mat_el_vec_dow(a, A, b, B, u, c, f, dof, mask)`
`bi_mat_el_vec_rrd(a, A, b, B, u, c, f, dof, mask)`
`bi_mat_el_vec_scl_dow(a, A, b, B, u, c, f, dof, mask)`
`bi_mat_el_vec_rdr(a, A, b, B, u, c, f, dof, mask)`
`bi_mat_el_vec_dow_scl(a, A, b, B, u, c, f, dof, mask)`
	$f[dof[i]]$
	$\leftarrow ((a\,A + b\,B)\,u)_i + c\,f[dof[i]]$
	if `mask[i] != DIRICHLET`
	or `mask == NULL`

Table 4.2: BLAS-operations for element-vectors and -matrices. A and B denote element matrices, u an element vector. f is a global `DOF`-vector. `mask` is an `EL_SCHAR_VEC` masking out certain *local* `DOF`s. `mask` may be `NULL`. `dof` is `EL_DOF_VEC` mapping local to global `DOF`s. a, b, c are numbers.

local basis-function set with the local basis function set on the neighbor. Intentionally, this is meant to support assembling linear systems in the context of DG-methods. The idea is that `EL_MATRIX_INFO.neigh_el_mat_fcts[neigh_nr](el_info, EL_MATRIX_INFO.neigh_fill_info)` assembles a jump-term where the local basis functions on the element described by `el_info` are used as test-functions (corresponding to the rows of the element matrix) and the local basis function set on the neighbour element defines the local space of ansatz-functions (column-space).

`neigh_fill_info` Data pointer passed to the element-matrix functions stored in `neigh_el_mat_fcts`.

`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 *dof_matrix, const EL_MATRIX_INFO *minfo,
                   MatrixTranspose transpose);;
```

Description:

update_matrix(matrix, info, transpose) 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_flag` 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 `NULL`).

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 to zero, or `info->get_col_admin` or `info->get_col_dof` is a `NULL` pointer; when 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 `NULL` 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()` acts *additive*, the element-contributions are added to the data already present in `dof_matrix`. This makes several calls for the assemblage of one matrix possible. `clear_dof_matrix()` can be used to erase the contents of `dof_matrix` prior to calling `update_matrix()`.

Parameters

dof_matrix The global `DOF_MATRIX` to add data to.

minfo The element-matrix handle, as returned by `fill_matrix_info()` or

transpose

4.7.3 Matrix assemblage for second order problems

Now we want to describe some tools which enable an easy assemblage of the system matrix in the case of scalar elliptic problems. For this we have to provide a function for the calculation of the element matrix. For a general scalar problem the element matrix $\boldsymbol{L}_S = (L_S^{ij})_{i,j=1,\dots,m}$ is given by (recall (1.21) on page 29)

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

where $\bar{A}$, $\bar{b}$, and $\bar{c}$ are functions depending on given data and on the actual element, namely

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

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 (1.22) on page 30) for this term. Here we can use pre–computed integrals of the basis functions on the standard element and transform th-em 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 4.7.5.

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 instance 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 instance $\bar b^0 = \bar b$ 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; /* range fe-space */
  const FE_SPACE *col_fe_space; /* domain fe-space */

  const QUAD     *quad[3];

  bool           (*init_element)(const EL_INFO *el_info,
                                 const QUAD *quad[3], void *apd);

  union {
```

```
    const REAL_B *(*real)(const EL_INFO *el_info,
                          const QUAD *quad, int iq, void *apd);
    const REAL_BD *(*real_d)(const EL_INFO *el_info,
                             const QUAD *quad, int iq, void *apd);
    const REAL_BDD *(*real_dd)(const EL_INFO *el_info,
                               const QUAD *quad, int iq, void *apd);
  } LALt;
  MATENT_TYPE     LALt_type; /* MATENT_REAL, _REAL_D or _REAL_DD */
  bool            LALt_pw_const;
  bool            LALt_symmetric;
  int             LALt_degree;

  union {
    const REAL *(*real)(const EL_INFO *el_info,
                        const QUAD *quad, int iq, void *apd);
    const REAL_D *(*real_d)(const EL_INFO *el_info,
                            const QUAD *quad, int iq, void *apd);
    const REAL_DD *(*real_dd)(const EL_INFO *el_info,
                              const QUAD *quad, int iq, void *apd);
  } Lb0;
  bool            Lb0_pw_const;
  union {
    const REAL *(*real)(const EL_INFO *el_info,
                        const QUAD *quad, int iq, void *apd);
    const REAL_D *(*real_d)(const EL_INFO *el_info,
                            const QUAD *quad, int iq, void *apd);
    const REAL_DD *(*real_dd)(const EL_INFO *el_info,
                              const QUAD *quad, int iq, void *apd);
  } Lb1;
  bool            Lb1_pw_const;
  MATENT_TYPE     Lb_type; /* MATENT_REAL, _REAL_D or _REAL_DD */
  bool            Lb0_Lb1_anti_symmetric;
  int             Lb_degree;

  union {
    REAL (*real)(const EL_INFO *el_info,
                 const QUAD *quad, int iq, void *apd);
    const REAL *(*real_d)(const EL_INFO *el_info,
                          const QUAD *quad, int iq, void *apd);
    const REAL_D *(*real_dd)(const EL_INFO *el_info,
                             const QUAD *quad, int iq, void *apd);
  } c;
  bool            c_pw_const;
  MATENT_TYPE     c_type; /* MATENT_REAL, _REAL_D or _REAL_DD */
  int             c_degree;

  BNDRY_FLAGS     dirichlet_bndry; /* bndry-type bit-mask for
                                    * Dirichlet-boundary conditions
                                    * built into the matrix
                                    */
  FLAGS           fill_flag;
  void            *user_data; /* application data, passed to init_element */
};
```

4.7.2 Compatibility Note. *Former versions of the* ALBERTA *toolkit had special "*`DOWB_OPERATOR_INFO`*" and* `DOF_DOWB_MATRIX`*" definitions to model block-matrix structures*

with DIM_OF_WORLD × DIM_OF_WORLD *blocks,* 1 × DIM_OF_WORLD *and* DIM_OF_WORLD × 1 *blocks and* 1 × 1 *blocks (i.e. not-blocked). Because those structures included the scalar case as well, the ordinary scalar-only* OPERATOR_INFO *and* DOF_MATRIX *structures have been abandoned altogether, and the* ..._DOWB_... *versions were renamed, dropping the bizarre* DOWB *component of their names.*

Description of the OPERATOR_INFO structure:

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 NULL pointer;

if init_element is not NULL, 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() (calculate 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.; the return value is of interest in the case of parametric meshes and should be true if the element is a curved element and false otherwise.

LALt, LALt.real, LALt.real_d, LALt.real_dd is a pointer to a function for the evaluation of $\bar{A}$ at quadrature nodes on the element; LALt may be a NULL pointer, if no second order term has to be integrated.

if LALt is not NULL, LALt(el_info, quad, iq, user_data) returns a pointer to a matrix of size N_LAMBDA × N_LAMBDA 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 and EL_INFO the current element descriptor.

The element-type of the returned matrix is determined by LALt_type, i.e. the actual return type is either REAL_BB for MATENT_REAL, REAL_BBD for MATENT_REAL_D or REAL_BBDD for MATENT_REAL_DD. Note that one of the B's is missing in the structure definition above because the LALt is supposed to return the address of the first element of the matrix.

LALt_type codes the block-matrix type, see MATENT_TYPE on page 252.

LALt_pw_const should be true 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; this entry also influences the assembly on parametric meshes with strategy>0, see Section 3.8.1: ALBERTA will assume a constant value of $\bar{A}$ for non–curved elements on a parametric mesh and optimize by only calling LALt once with iq==0;

LALt_symmetric should be true 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;

LALt_degree If LALt_pw_const == false, the LALt_degree gives a hint about which quadrature rule should be used to integrate the second order term. This has only an effect if quad[2] == NULL. In that case, ALBERTA takes LALt_degree and the row- and column finite element spaces into account to select a suitable quadrature formula.

Lb0, Lb0.real, Lb0.real_d, Lb0.real_dd is a pointer to a function for the evaluation of $\bar{b}^0$, at quadrature nodes on the element; Lb0 may be a NULL pointer, if this first order term has not to be integrated;

if Lb0 is not NULL, Lb0(el_info, quad, iq, user_data) returns a pointer to a vector of length N_LAMBDA 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 should be true 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; for parametric meshes the same remarks as for LALt_symmetric above hold;

Lb1, Lb1.real, Lb1.real_d, Lb1.real_dd is a pointer to a function for the evaluation of $\bar{b}^1$, at quadrature nodes on the element; Lb1 may be a NULL pointer, if this first order term has not to be integrated;

if Lb1 is not NULL, Lb1(el_info, quad, iq, user_data) returns a pointer to a vector of length N_LAMBDA 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 should be true 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;

Lb_type see LALt_type.

Lb0_Lb1_anti_symmetric should be true 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 NULL, $\bar{b}^0 = -\bar{b}^1$, 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;

S ee the explanations for LALt_degree above.

c, c.real, c.real_d, c.real_dd is a pointer to a function for the evaluation of $\bar{c}$ at quadrature nodes on the element; c may be a NULL pointer, if no zero order term has to be integrated;

if c is not NULL, 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_type see LALt_type.

c_pw_const should be true 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; the same remarks about parametric meshes as for the other *_pw_const entries hold;

S ee the explanations for LALt_degree above.

dirichlet_bndry A bit mask flagging those components of the boundary of the triangulation which are subject to Dirichlet boundary conditions. See Section 3.2.4.

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. A better name would maybe "application data", at any rate this is the channel were an application program can communicate data – like the determinant of the transformation to the reference element – from `init_element()` to the operator kernels `LALt` & friends, without using global variables. *The data behind this pointer must be persistent for the entire life time of the application program. Especially,* `user_data` *must not point to the stack area of some sub-routine call.*

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.

Sometimes it is necessary to add contributions of boundary integrals to the system matrix. One example are "Robin" boundary conditions (see Section 4.7.7.3), other important examples include capillary boundary conditions in the context of free boundary problems, or penalty terms to penalize tangential stresses. Another context which requires integration over the boundaries of all mesh elements is the implementation of discontinuous Galerkin (DG) methods. To aid these tasks there is a `BNDRY_OPERATOR_INFO` structure, which resembles in its layout the (bulk-) `OPERATOR_INFO` structure; it is defined as follows:

```
typedef struct bndry_operator_info  BNDRY_OPERATOR_INFO;

struct bndry_operator_info
{
  const FE_SPACE *row_fe_space;
  const FE_SPACE *col_fe_space;

  const WALL_QUAD *quad[3];

  bool          (*init_element)(const EL_INFO *el_info, int wall,
                                const WALL_QUAD *quad[3], void *ud);

  union {
    const REAL_B *(*real)(const EL_INFO *el_info,
                          const QUAD *quad, int iq, void *apd);
    const REAL_BD *(*real_d)(const EL_INFO *el_info,
                             const QUAD *quad, int iq, void *apd);
    const REAL_BDD *(*real_dd)(const EL_INFO *el_info,
                               const QUAD *quad, int iq, void *apd);
  } LALt;
  MATENT_TYPE     LALt_type; /* MATENT_REAL, _REAL_D or _REAL_DD */
  bool            LALt_pw_const;
  bool            LALt_symmetric;
  int             LALt_degree;

  union {
    const REAL *(*real)(const EL_INFO *el_info,
```

```
                        const QUAD *quad, int iq, void *apd);
    const REAL_D *(*real_d)(const EL_INFO *el_info,
                            const QUAD *quad, int iq, void *apd);
    const REAL_DD *(*real_dd)(const EL_INFO *el_info,
                              const QUAD *quad, int iq, void *apd);
  } Lb0;
  bool            Lb0_pw_const;
  union {
    const REAL *(*real)(const EL_INFO *el_info,
                        const QUAD *quad, int iq, void *apd);
    const REAL_D *(*real_d)(const EL_INFO *el_info,
                            const QUAD *quad, int iq, void *apd);
    const REAL_DD *(*real_dd)(const EL_INFO *el_info,
                              const QUAD *quad, int iq, void *apd);
  } Lb1;
  bool            Lb1_pw_const;
  MATENT_TYPE     Lb_type; /* MATENT_REAL, _REAL_D or _REAL_DD */
  bool            Lb0_Lb1_anti_symmetric;
  int             Lb_degree;

  union {
    REAL (*real)(const EL_INFO *el_info,
                 const QUAD *quad, int iq, void *apd);
    const REAL *(*real_d)(const EL_INFO *el_info,
                          const QUAD *quad, int iq, void *apd);
    const REAL_D *(*real_dd)(const EL_INFO *el_info,
                             const QUAD *quad, int iq, void *apd);
  } c;
  bool            c_pw_const;
  MATENT_TYPE     c_type; /* MATENT_REAL, _REAL_D or _REAL_DD */
  int             c_degree;

  /* boundary segment(s) we belong to; if
   * BNDRY_FLAGS_IS_INTERIOR(bndry_type), then the operator is invoked
   * on all interior faces, e.g. to implement a DG-method.
   */
  BNDRY_FLAGS   bndry_type;

  bool          discontinuous; /* assemble jumps w.r.t. the neighbour */
  bool          tangential;    /* use tangential gradients */

  FLAGS         fill_flag;
  void          *user_data;
};
```

Description: Because the general layout is the same as for the bulk-`OPERATOR_INFO` structure explained above we document only the differences here. There are three additional components in the structure:

`bndry_type` This is bit-mask and determines on which part of the boundary the operator should be invoked. See also Section 3.2.4. If `BNDRY_FLAGS_IS_INTERIOR(bndry_type)` evaluates to `true` (i.e. if bit 0 is set in `bndry_type`, then the operator is invoked on all walls of the triangulation, for instance to implement a DG-method.

`discontinuous` This is a boolean flag. If set to `true`, then the operator is treated as a DG-operator. This means, that it is invoked once for each wall of each element with the

set of local basis functions on the neighbor element being used to define the column space (i.e. as ansatz-functions) and the set of local basis function on the current element defining the row-space (i.e. the test-functions).

One instance of `BNDRY_OPERATOR_INFO` can only be used to either implement a jump term or a term living on a single element. To have both, two instances have to be defined. To this aim `fill_matrix_info_ext()` accepts multiple `BNDRY_OPERATOR_INFO` structures. The program-code `src/Common/ellipt-dg.c` in the `alberta-demo`-package implements a very simplistic DG-method as example: jumps over element boundaries are penalized by zero-order term.

`tangential` This is a boolean flag. If set to `true`, then only the tangential component of the gradients of the basis functions is used.

Information stored in `OPERATOR_INFO` and `BNDRY_OPERATOR_INFO` structures is used by the following functions which return 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 operators defined by the `OPERATOR_INFO` and `BNDRY_OPERATOR_INFO` structures:

```
const EL_MATRIX_INFO *
fill_matrix_info(const OPERATOR_INFO *operator_info,
                 EL_MATRIX_INFO *matrix_info);
const EL_MATRIX_INFO *
fill_matrix_info_ext(EL_MATRIX_INFO *matrix_info,
                     const OPERATOR_INFO *operator_info,
                     const BNDRY_OPERATOR_INFO *bop_info,
                     ... /* more bndry-ops */);
```

Description:

`fill_matrix_info(op_info, mat_info)`

`fill_matrix_info_ext(op_info, mat_info, bop_info, ...)`

Return a pointer to a filled `EL_MATRIX_INFO` structure for the assemblage of the system matrix for the operator defined in `op_info` and `bop_info`. The difference between the two functions is that the `..._ext()`-variant ("extended") allows for additional arguments describing components of the differential operator which have to be assembled by boundary integrals. Multiple such boundary-operators can be passed to `fill_matrix_info_ext()`, the final boundary operator must be followed by a `NULL` pointer. So

```
fill_matrix_info_ext(mat_info, operator_info, NULL);
```

is equivalent to

```
fill_matrix_info(operator_info, mat_info);
```

There is the artificial restriction that at most 255 different `BNDRY_OPERATOR_INFO` structures may be passed.

If the argument `mat_info` is a `NULL` pointer, a new structure `mat_info` is allocated and filled; otherwise the structure `mat_info` is filled; all members are newly assigned.

If the underlying finite element spaces form a direct sum, then this is taken care of automatically, and the return EL_MATRIX_INFO structure will assemble block-matrices, where each block corresponds to the pairing of one component of the direct sum forming the ansatz-space and one component of the direct sum forming the space of test functions. See also Section 3.7 and Section 3.5.3

The remaining part of this section is rather a description what happens "back-stage", when calling the `fill_matrix_info[_ext]()` functions. The components of EL_MATRIX_INFO are initialized like follows:

`row_fe_space, col_fe_space` `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 `NULL` pointers, an error message is given, and the program stops; if one of these pointers is `NULL`, 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).

`krn_blk_type` Based on the matrix block-type of the zero, first and second order kernels `oinfo->c_type`, `oinfo->Lb_type` and `oinfo->LALt_type` and on the dimension of the range of the row- and column finite element spaces `krn_blk_type` is set to either `MATENT_REAL`, `MATENT_REAL_D` or `MATENT_REAL_DD` to reflect the block-matrix structure of the element matrix.

`dirichlet_bndry` is just a copy of `oinfo->dirichlet_bndry`, see also section `S:boundary`.

`factor` is initialized to `1.0`. Note that the structure returned carries the `const` qualifier; the clean way to obtain EL_MATRIX_INFO structures with a modifiable `factor` component is to pass storage to `fill_matrix_info[_ext]()` via the `matrix_info` parameter.

`el_matrix_fct` 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 available function for the element integration for the operator defined in `op_info`, depending on `op_info->LALt_pw_const` and similar hints;

Denote by `row_degree` and `col_degree` the degree of the basis functions connected to the rows and columns. Internally, a three-element vector "`quad`" of quadratures rules is used for the element integration, if not specified by `op_info->quad`. The quadratures are chosen according to the following rules: 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 `NULL`, `quad[2] = op_info->quad[2]` is used, otherwise `quad[2]` is a quadrature which is exact of degree `row_degree+col_degree-2+oinfo->LALt_degree`. If the 2nd order term is not integrated then `quad[2]` is set to `NULL`.
- If the 1st order term has to be integrated and `op_info->quad[1]` is not `NULL`, `quad[1] = op_info->quad[1]` is used; otherwise: if `op_info->Lb_pw_const` is zero and `quad[2]` is not `NULL`, `quad[1] = quad[2]` is used, otherwise `quad[1]` is a quadrature which is exact of degree

row_degree+col_degree-1+oinfo->Lb_degree. If the 1st order term is not integrated then quad[1] is set to NULL.

- If the zero order term has to be integrated and op_info->quad[0] is not NULL, quad[0] = op_info->quad[0] is used; otherwise: if op_info->c_pw_const is zero and quad[2] is not NULL, quad[0] = quad[2] is used, if quad[2] is NULL and quad[1] is not NULL, quad[0] = quad[1] is used, or if both quadratures are NULL, quad[0] is a quadrature which is exact of degree row_degree+col_degree+oinfo->c_degree. If the zero order term is not integrated then quad[0] is set to NULL.

el_matrix_fct() roughly works as follows:

- If op_info->init_element is not NULL 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 the last argument is a pointer to the application-data pointer oinfo->user_data.
- If op_info->LALt is not NULL, 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 NULL, 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 NULL, 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 `NULL`, 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 `NULL`, 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 an arbitrary order on the elements, if not `NULL` (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 commonly used data for these functions is always initialized first by `op_info->init_element()`, if this function pointer is not `NULL`.
- 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()`).

`neigh_el_mat_fcts[]` See the documentation of the discontinuous component of the `BNDRY_OPERATOR_INFO` structure.

`fill_flag` 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 boundary bit-mask `op_info->dirichlet_bndry` has bits set (see also Section 3.2.4), then the `FILL_BOUND` flag is added to `mat_info->fill_flag`.

4.7.3 Example (Implementation of the differential operator $-\Delta$). The following source fragment gives an example of the implementation for the operator $-\Delta$ and the access to a `MATRIX_INFO` structure for the automatic assemblage of the system matrix for this problem for any set of used basis functions.

The source fragment shown here is part of the implementation for a Poisson equation, which is the first model problem described in detail in Section 2.2. However, we will generalize the code given in Section 2.2 to include the case of parametric meshes. The assemblage of the discrete system including the load vector and Dirichlet boundary values is spelled out in Section 2.2.7.

For the Poisson equation we only have to implement a constant second order term. For passing information about the gradient of the barycentric coordinates (at all quadrature points) from `init_element()` to the function `LALt` we define the following structure

```
struct app_data
{
  REAL_BD *Lambda;
  REAL    *det;
};
```

The function `init_element()` calculates the Jacobians Λ and determinants `det` of the barycentric coordinates and stores these in the above defined structure. In the case of a parametric mesh we fill `Lambda` with the Jacobians and `det` with the determinants at all quadrature points of `quad[2]`. For a non–parametric mesh we only fill the zeroth entry of `Lambda` and `det`. If `init_element()` returns `false`, then `LALt()` is only called once for the current simplex with `iq==0`, otherwise it is called for each quadrature point in `quad[2]`. Note that we need a higher order quadrature than usual to calculate the integral exactly for a curved parametric element.

```
static bool init_element(const EL_INFO *el_info, const QUAD *quad[3], void
    *ud)
{
  struct app_data *data = (struct app_data *)ud;
  PARAMETRIC      *parametric = el_info->mesh->parametric;

  if (parametric && parametric->init_element(el_info, parametric)) {
    parametric->grd_lambda(el_info, quad[2], 0, NULL,
                           data->Lambda, NULL, data->det);
    return true;
  } else {
    data->det[0] = el_grd_lambda(el_info, data->Lambda[0]);
    return false;
  }
}
```

The function `LALt` now has to calculate the scaled matrix product $|\det DF_S|\Lambda\Lambda^t$. Note that `LALt()` is invoked only for the first quadrature point (`iq == 0`), if the `OPERATOR_INFO`-structure claims that the second-order kernel is piece-wise constant and `parametric->init_element()` returns `false`, so using the index `iq` into the fields `det` and `Lambda` does not access invalid data, and the assembling linear systems remains relatively efficient, even in the context of iso-parametric boundary approximation.

```
const REAL_B *LALt(const EL_INFO *el_info, const QUAD *quad,
                   int iq, void *ud)
{
  struct app_data *data = (struct app_data *)ud;
  int             i, j;
  static REAL_BB  LALt; /* mind the "static" keyword! */

  for (i = 0; i < N_VERTICES(MESH_DIM); i++) {
    LALt[i][i]  = SCP_DOW(data->Lambda[iq][i], data->Lambda[iq][i]);
    LALt[i][i] *= data->det[iq];
    for (j = i+1; j <  N_VERTICES(MESH_DIM); j++) {
      LALt[i][j]  = SCP_DOW(data->Lambda[iq][i], data->Lambda[iq][j]);
      LALt[i][j] *= data->det[iq];
      LALt[j][i]  = LALt[i][j];
    }
  }

  return (const REAL_B *)LALt;
}
```

Before assembling the system matrix for the operator $-\Delta$, we first have to initialize an `EL_MATRIX_INFO` structure. A pointer to this `EL_MATRIX_INFO` structure is the second argument

to the function `update_matrix()`, which finally assembles the system matrix (compare Section 4.7.2).

For the initialization we have to fill an `OPERATOR_INFO` structure collecting all information about the differential operator. For $-\Delta$ we only need pointers to the functions `init_element()` and `LALt()` described above. The differential operator is constant and symmetric, and information about vertex coordinates is needed for computing the Jacobian of the barycentric coordinates. Information about Dirichlet boundary values should be assembled into the system matrix, hence the entry `operator_info->use_get_bound` is set `true`.

The functions `init_element()` and `LALt()` do not depend on the finite element space which is used. This functions can be used for any conforming finite element discretization for the Poisson equation; all information needed about the actually used finite elements is a pointer to this finite element space; we assume that this pointer is stored in the variable `fe_space`.

```
const EL_MATRIX_INFO *matrix_info = NULL;
static struct app_data app_data; /* Must be static! */
OPERATOR_INFO  o_info = { NULL, };

if(mesh->parametric)
  quad = get_quadrature(2, 2*fe_space->bas_fcts->degree + 2);
else
  quad = get_quadrature(2, 2*fe_space->bas_fcts->degree-2);

app_data.Lambda = MEM_ALLOC(quad->n_points, REAL_BD);
app_data.det    = MEM_ALLOC(quad->n_points, REAL);

o_info.quad[2]         = quad;
o_info.row_fe_space    = o_info.col_fe_space = fe_space;
o_info.init_element    = init_element;
o_info.LALt.real       = LALt;
o_info.LALt_pw_const   = true;       /* pw const. assemblage is faster */
o_info.LALt_symmetric  = true;       /* symmetric assemblage is faster */
o_info.user_data       = &app_data; /* application data */

/* Use, e.g., Dirichlet boundary conditions. */
BNDRY_FLAGS_CPY(o_info.dirichlet_bndry, dirichlet_mask);

o_info.fill_flag = CALL_LEAF_EL|FILL_COORDS;

matrix_info = fill_matrix_info(&o_info, NULL);
```

Full information about the operator is now available via the `matrix_info` structure and the system matrix `matrix` can then easily be assembled for the selected finite element space by

```
clear_dof_matrix(matrix);
update_matrix(matrix, matrix_info, NoTranspose);
```

4.7.4 Matrix assemblage for coupled second order problems

The corresponding mechanism for coupled vector valued problems is very similar, except for the additional indices necessary to describe coupling. We start by stating the form of the element matrix with the generalized first order term and different finite element spaces (see also 1.4.5):

$$
\begin{aligned}
L^{ij}_{S,\mu\nu} := &\int_{\hat S} \nabla_\lambda \bar\psi^i(\lambda(\hat x)) \cdot \bar A^{\mu\nu}(\lambda(\hat x))\, \nabla_\lambda \bar\varphi^j(\lambda(\hat x))\, d\hat x \\
&+ \int_{\hat S} \bar\psi^i(\lambda(\hat x))\, \bar b^{0,\mu\nu}(\lambda(\hat x)) \cdot \nabla_\lambda \bar\varphi^j(\lambda(\hat x))\, d\hat x \\
&+ \int_{\hat S} \nabla_\lambda \bar\psi^i(\lambda(\hat x)) \cdot\ \bar b^{1,\mu\nu}(\lambda(\hat x))\, \bar\varphi^j(\lambda(\hat x))\, d\hat x \\
&+ \int_{\hat S} \bar c^{\mu\nu}(\lambda(\hat x))\, \bar\psi^i(\lambda(\hat x))\, \bar\varphi^j(\lambda(\hat x))\, d\hat x,
\end{aligned}
$$

with

$$
\begin{aligned}
\bar A^{\mu\nu}(\lambda) &:= \big(\bar a^{\mu\nu}_{kl}(\lambda)\big)_{k,l=0,\dots,d} := |\det DF_S(\hat x(\lambda))|\, \Lambda(x(\lambda))\, A^{\mu\nu}(x(\lambda))\, \Lambda^t(x(\lambda)), \\
\bar b^{0,\mu\nu}(\lambda) &:= \Big(\bar b^{0,\mu\nu}_l(\lambda)\Big)_{l=0,\dots,d} := |\det DF_S(\hat x(\lambda))|\, \Lambda(x(\lambda))\, b^{0,\mu\nu}(x(\lambda)), \\
\bar b^{1,\mu\nu}(\lambda) &:= \Big(\bar b^{1,\mu\nu}_l(\lambda)\Big)_{l=0,\dots,d} := |\det DF_S(\hat x(\lambda))|\, \Lambda(x(\lambda))\, b^{1,\mu\nu}(x(\lambda)), \quad \text{and} \\
\bar c^{\mu\nu}(\lambda) &:= |\det DF_S(\hat x(\lambda))|\, c^{\mu\nu}(x(\lambda))
\end{aligned}
$$

for $\mu, \nu = 1, \dots, n$, $n =$ `DIM_OF_WORLD`.

To store information about the coupled operator and finite element spaces, we use the same `OPERATOR_INFO` (see page 267) structure as for the scalar problems, we only have to adjust the respective `MATENT_TYPE` structure components to the correct block-matrix type. Also, the same `fill_matrix_info()` and `add_element_matrix()` routines are used for scalar and vector valued problems.

4.7.5 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

$$
\begin{aligned}
\hat Q\Big(\sum_{k,l=0}^{d} (\bar a_{S,kl} \bar\psi^i_{,\lambda_k}\, \bar\varphi^j_{,\lambda_l})\Big) &= \sum_{k,l=0}^{d} \bar a_{S,kl} \hat Q\Big(\bar\psi^i_{,\lambda_k}\, \bar\varphi^j_{,\lambda_l}\Big), \\
\hat Q\Big(\sum_{l=0}^{d} (\bar b^0_{S,l}\, \bar\psi^i\, \bar\varphi^j_{,\lambda_l})\Big) &= \sum_{l=0}^{d} \bar b^0_{S,l}\, \hat Q\Big(\bar\psi^i\, \bar\varphi^j_{,\lambda_l}\Big), \\
\hat Q\Big(\sum_{k=0}^{d} (\bar b^1_{S,k}\, \bar\psi^i_{,\lambda_k}\, \bar\varphi^j)\Big) &= \sum_{k=0}^{d} \bar b^1_{S,k}\, \hat Q\Big(\bar\psi^i_{,\lambda_k}\, \bar\varphi^j)\Big), \qquad \text{and} \\
\hat Q\Big((\bar c_S\, \bar\psi^i\, \bar\varphi^j)\Big) &= \bar c_S\, \hat Q\Big(\bar\psi^i\, \bar\varphi^j\Big).
\end{aligned}
\tag{4.1}
$$

The values of the quadrature applied to the basis functions 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 $\{\bar\psi_i\}_{i=1,\dots,n}$ and $\{\bar\varphi_j\}_{j=1,\dots,m}$ the values

$$\hat Q^{11}_{ij,kl} := \hat Q\Big(\bar\psi^i_{,\lambda_k}\,\bar\varphi^j_{,\lambda_l}\Big) \qquad \text{for } 1\le i\le n,\ 1\le j\le m,\ 0\le k,l\le d,$$

if $A\neq 0$,

$$\hat Q^{01}_{ij,l} := \hat Q\Big(\bar\psi^i\,\bar\varphi^j_{,\lambda_l}\Big) \qquad \text{for } 1\le i\le n,\ 1\le j\le m,\ 0\le l\le d,$$

if $b^0\neq 0$,

$$\hat Q^{10}_{ij,k} := \hat Q\Big(\bar\psi^i_{,\lambda_k}\,\bar\varphi^j\Big) \qquad \text{for } 1\le i\le n,\ 1\le j\le m,\ 0\le k\le d$$

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 = 10.0 * REAL_EPSILON;
```

and all computed values `val` with $|$`val`$|\le$ `TOO_SMALL` are treated as zeros. As we are considering here integrals over the standard simplex, non-zero integrals are usually of order one, such that the above constant is of the order of roundoff errors for double precision.

The following data structure is used for storing values $\hat Q^{11}$ for two sets of basis functions integrated with a given quadrature. Note that in the context of "chained" basis-functions (see Section 3.5.3 the cache-structure nevertheless hold data for only a single component of such a multi-component chain.

```
typedef struct q11_psi_phi    Q11_PSI_PHI;

struct q11_psi_phi
{
  const BAS_FCTS     *psi;
  const BAS_FCTS     *phi;
  const QUAD         *quad;

  const Q11_PSI_PHI_CACHE *cache;

  INIT_ELEMENT_DECL;
};

typedef struct q11_psi_phi_cache
{
  int   n_psi;
  int   n_phi;
```

```
  const int   *const*n_entries;
  const REAL *const*const*values;
  const int   *const*const*k;
  const int   *const*const*l;
} Q11_PSI_PHI_CACHE;
```

Description:

struct q11_psi_phi :

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.

cache Pointer to the actual data in the cache.

INIT_ELEMENT_DECL Optional per-element initializer. This entry is initialized when calling `get_q11_psi_phi()` if either the underlying basis functions or the underlying quadrature rule has per-element initializers. See Section 3.11.

struct q11_psi_phi_cache :

n_psi Dimension of the local space of test-functions (row space), equals `Q11_PSI_PHI.psi->n_bas_fcts`.

n_phi Dimension of the local space of ansatz-functions (column space), equals `Q11_PSI_PHI.phi->n_bas_fcts`.

n_entries matrix of size `n_psi` × `n_phi` storing the count of non zero integrals;
`n_entries[i][j]` is the count of non zero values of $\hat{Q}^{11}_{\mathtt{ij},\mathtt{kl}}$ ($0 \le \mathtt{k,l} \le d$) for the pair (`psi[i]`, `phi[j]`), $0 \le \mathtt{i} <$ `n_psi`, $0 \le \mathtt{j} <$ `n_phi`.

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.

The following formulas summarize the relationship between the cache data-structure and the formulas (4.1) at the beginning of this section:

$$\mathtt{values[i][j][r]} = \hat{Q}^{11}_{\mathtt{ij},\mathtt{k[i][j][r]},\mathtt{l[i][j][r]}} = \hat{Q}\Big(\bar{\psi}^{\mathtt{i}}_{,\lambda_{\mathtt{k[i][j][r]}}}\, \bar{\varphi}^{\mathtt{j}}_{,\lambda_{\mathtt{l[i][j][r]}}}\Big),$$

for $0 \le \mathtt{r} <$ `n_entries[i][j]`. Using these pre–computed values we have for all elements S

$$\sum_{k,l=0}^{d} \bar{a}_{S,kl} \hat{Q}\Big(\bar{\psi}^{i}_{,\lambda_k}\, \bar{\varphi}^{j}_{,\lambda_l}\Big) = \sum_{\mathtt{r}=0}^{\mathtt{n\_entries[i][j]-1}} \bar{a}_{S,\mathtt{k[i][j][r]},\mathtt{l[i][j][r]}} \,\mathtt{*values[i][j][r]}.$$

The following function initializes a `Q11_PSI_PHI` structure:

```
const Q11_PSI_PHI *get_q11_psi_phi(const BAS_FCTS *psi, const BAS_FCTS *phi,
                                   const QUAD *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 `NULL` pointers, nothing is done and the return value is `NULL`; if one of the pointers is a `NULL` 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 `NULL`, 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 $|\texttt{val}| \leq$ `TOO_SMALL` are treated as zeros.

4.7.4 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):

```
static Q11_PSI_PHI_CACHE *q11_psi_phi;

if (!q11_psi_phi) {
  q11_psi_phi = get_q11_psi_phi(psi, phi, quad[2])->cache;
}

LALt = LALt_fct(el_info, quad, 0, user_data);
n_entries = q11_psi_phi->n_entries;

for (i = 0; i < q11_psi_phi->n_psi; i++)
{
  for (j = 0; j < q11_psi_phi->n_phi; 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;

typedef struct q01_psi_phi_cache
```

```
{
  int   n_psi;
  int   n_phi;

  const int  *const*n_entries;
  const REAL *const*const*values;
  const int  *const*const*l;
} Q01_PSI_PHI_CACHE;

struct q01_psi_phi
{
  const BAS_FCTS     *psi;
  const BAS_FCTS     *phi;
  const QUAD         *quad;

  const Q01_PSI_PHI_CACHE *cache;

  INIT_ELEMENT_DECL;
};
```

Description:

struct q01_psi_phi :

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.

cache Pointer to the actual data in the cache.

INIT_ELEMENT_DECL Optional per-element initializer. This entry is initialized when calling `get_q11_psi_phi()` if either the underlying basis functions or the underlying quadrature rule has per-element initializers. See Section 3.11.

struct q01_psi_phi_cache :

n_psi Dimension of the local space of test-functions (row space), equals `Q11_PSI_PHI.psi->n_bas_fcts`.

n_phi Dimension of the local space of ansatz-functions (column space), equals `Q11_PSI_PHI.phi->n_bas_fcts`.

n_entries matrix of size `psi->n_bas_fcts` × `phi->n_bas_fcts` storing the count of non zero integrals;

`n_entries[i][j]` is the count of non zero values of $\hat{Q}^{01}_{\mathtt{ij},\mathtt{l}}$ ($0 \le \mathtt{l} \le d$) for the pair (`psi[i]`, `phi[j]`), $0 \le \mathtt{i} <$ `n_psi`, $0 \le \mathtt{j} <$ `n_phi`.

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.

The following formulas summarize the relationship between the cache data-structure and the formulas (4.1) at the beginning of this section:

$$\texttt{values[i][j][r]} = \hat{Q}^{01}_{\texttt{ij,l[i][j][r]}} = \hat{Q}\Big(\bar{\psi}^{\texttt{i}}\,\bar{\varphi}^{\texttt{j}}_{,\lambda_{\texttt{l[i][j][r]}}}\Big),$$

for $0 \le \texttt{r} < \texttt{n\_entries[i][j]}$. Using these pre–computed values we have for all elements S

$$\sum_{l=0}^{d} \bar{b}^0_{S,l}\hat{Q}\Big(\bar{\psi}^{i}\,\bar{\varphi}^{j}_{,\lambda_l}\Big) = \sum_{\texttt{r=0}}^{\texttt{n\_entries[i][j]-1}} \bar{b}^0_{S,\texttt{l[i][j][r]}}\,\texttt{*values[i][j][r]}.$$

The following function initializes a `Q01_PSI_PHI` structure:

```
const Q01_PSI_PHI *get_q01_psi_phi(const BAS_FCTS *psi, const BAS_FCTS *phi,
                                   const QUAD *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 `NULL` pointers, nothing is done and the return value is `NULL`; if one of the pointers is a `NULL` 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 `NULL`, 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 $|\texttt{val}| \le \texttt{TOO\_SMALL}$ 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;

typedef struct q10_psi_phi_cache
{
  int         n_psi;
  int         n_phi;

  const int  *const*n_entries;
  const REAL *const*const*values;
  const int  *const*const*k;
} Q10_PSI_PHI_CACHE;

struct q10_psi_phi
{
  const BAS_FCTS     *psi;
  const BAS_FCTS     *phi;
  const QUAD         *quad;
```

```
  const Q10_PSI_PHI_CACHE *cache;

  INIT_ELEMENT_DECL;
};
```

Description:

struct q10_psi_phi :

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.

cache Pointer to the actual data in the cache.

INIT_ELEMENT_DECL Optional per-element initializer. This entry is initialized when calling `get_q11_psi_phi()` if either the underlying basis functions or the underlying quadrature rule has per-element initializers. See Section 3.11.

struct q10_psi_phi_cache :

n_psi Dimension of the local space of test-functions (row space), equals `Q11_PSI_PHI.psi->n_bas_fcts`.

n_phi Dimension of the local space of ansatz-functions (column space), equals `Q11_PSI_PHI.phi->n_bas_fcts`.

n_entries matrix of size `psi->n_bas_fcts` × `phi->n_bas_fcts` storing the count of non zero integrals;

`n_entries[i][j]` is the count of non zero values of $\hat{Q}^{10}_{\mathtt{ij},\mathtt{k}}$ ($0 \le \mathtt{k} \le d$) for the pair (`psi[i]`, `phi[j]`), $0 \le \mathtt{i} <$ `n_psi`, $0 \le \mathtt{j} <$ `n_phi`.

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.

The following formulas summarize the relationship between the cache data-structure and the formulas (4.1) at the beginning of this section:

$$\mathtt{values[i][j][r]} = \hat{Q}^{10}_{\mathtt{ij,k[i][j][r]}} = \hat{Q}\Big(\bar{\psi}^{\mathtt{i}}_{,\bar{\lambda}_{\mathtt{k[i][j][r]}}}\,\bar{\varphi}^{\mathtt{j}}\Big),$$

for $0 \le \mathtt{r} <$ `n_entries[i][j]`. Using these pre–computed values we have for all elements S

$$\sum_{k=0}^{d} \bar{b}^1_{S,k}\hat{Q}\Big(\bar{\psi}^{i}_{,\lambda_k}\,\bar{\varphi}^{j}\Big) = \sum_{\mathtt{r}=0}^{\mathtt{n\_entries[i][j]-1}} \bar{b}^1_{S,\mathtt{k[i][j][r]}} \,\mathtt{*values[i][j][r]}.$$

The following function initializes a `Q10_PSI_PHI` structure:

```
const Q10_PSI_PHI *get_q10_psi_phi(const BAS_FCTS *psi, const BAS_FCTS *phi,
                                   const QUAD *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 NULL pointers, nothing is done and the return value is NULL; if one of the pointers is a NULL 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 NULL, 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 |val| $\leq$ 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;

typedef struct q00_psi_phi_cache
{
  int         n_psi;
  int         n_phi;

  const REAL *const*values;
} Q00_PSI_PHI_CACHE;

struct q00_psi_phi
{
  const BAS_FCTS     *psi;
  const BAS_FCTS     *phi;
  const QUAD         *quad;

  const Q00_PSI_PHI_CACHE *cache;

  INIT_ELEMENT_DECL;
};
```

Description:

struct q00_psi_phi :

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.

cache Pointer to the actual data in the cache.

INIT_ELEMENT_DECL Optional per-element initializer. This entry is initialized when calling get_q11_psi_phi() if either the underlying basis functions or the underlying quadrature rule has per-element initializers. See Section 3.11.

struct q00_psi_phi_cache :

n_psi Dimension of the local space of test-functions (row space), equals `Q11_PSI_PHI.psi->n_bas_fcts`.

n_phi Dimension of the local space of ansatz-functions (column space), equals `Q11_PSI_PHI.phi->n_bas_fcts`.

values matrix storing the integrals.

The following formulas summarize the relationship between the cache data-structure and the formulas (4.1) at the beginning of this section:

$$\texttt{values[i][j]} = \hat{Q}^{00}_{\texttt{ij}} = \hat{Q}\Big(\bar{\psi}^{\texttt{i}}\,\bar{\varphi}^{\texttt{j}}\Big),$$

for the pair (`psi[i]`, `phi[j]`), $0 \le$ `i` $<$ `psi->n_bas_fcts`, $0 \le$ `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 *psi, const BAS_FCTS *phi,
                                   const QUAD *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 `NULL` pointers, nothing is done and the return value is `NULL`; if one of the pointers is a `NULL` 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 `NULL`, 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`.

4.7.6 Data structures and functions for updating coefficient vectors

Besides the general routines `update_real_vec()`, `update_real_d_vec()` and `update_real_vec_dow()`, 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, taken either over the interior of the mesh or over boundary segments.

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;
typedef struct el_vec_info_d   EL_VEC_INFO_D;

typedef const EL_REAL_VEC *
(*EL_VEC_FCT)(const EL_INFO *el_info, void *fill_info);
```

```
typedef struct el_vec_info  EL_VEC_INFO;
struct el_vec_info
{
  const FE_SPACE *fe_space;

  BNDRY_FLAGS     dirichlet_bndry;
  REAL            factor;

  EL_VEC_FCT      el_vec_fct;
  void            *fill_info;

  FLAGS           fill_flag;
};

typedef const EL_REAL_D_VEC *
(*EL_VEC_D_FCT)(const EL_INFO *el_info, void *fill_info);

typedef struct el_vec_d_info  EL_VEC_D_INFO;
struct el_vec_d_info
{
  const FE_SPACE *fe_space;

  BNDRY_FLAGS     dirichlet_bndry;
  REAL            factor;

  EL_VEC_D_FCT    el_vec_fct;
  void            *fill_info;

  FLAGS           fill_flag;
};

typedef const EL_REAL_VEC_D *
(*EL_VEC_FCT_D)(const EL_INFO *el_info, void *fill_info);

typedef struct el_vec_info_d  EL_VEC_INFO_D;
struct el_vec_info_d
{
  const FE_SPACE *fe_space;

  BNDRY_FLAGS     dirichlet_bndry;
  REAL            factor;

  EL_VEC_FCT_D    el_vec_fct;
  void            *fill_info;

  FLAGS           fill_flag;
};
```

Description:

`fe_space` the underlying finite element space

`dirichlet_bndry` a bit mask marking the boundary segments which are subject to Dirichlet boundary conditions, see also Section 3.2.4.

`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 an element vector of the respective

data type, see e.g. `EL_REAL_VEC` on page 253. This vector stores the computed values for the element described by `el_info`. `fill_info` is a pointer to data needed by `el_vec_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 held in a `EL_VEC[_D]_INFO` structure:

```
void update_real_vec(DOF_REAL_VEC *dv, const EL_VEC_INFO *vec_info);
void update_real_d_vec(DOF_REAL_D_VEC *dv, const EL_VEC_D_INFO *vec_info);
void update_real_vec_dow(DOF_REAL_VEC_D *dv, const EL_VEC_INFO_D *vec_info)
```

`update_real[_d]_vec[_dow](dv, info)` updates the vector `dr` 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 `dv->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 `NULL`).

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 `NULL` pointer; then the element vector is computed by `info->el_vec_fct(el_info, info->fill_info)`; these contributions are finally added to `dv` multiplied by `info->factor` by a call of `add_element[_d]_vec[_dow]()` with all information about global DOFs, the element vector, and boundary types, if available;

`update_real[_d]_vec[_dow]()` 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|_dow](0.0, dv)`.

L^2- and H^1-scalar- products over the bulk phase In many applications, the load vector is just the L^2- or H^1-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 following functions.

Prototypes

```
void L2scp_fct_bas(FCT_AT_X f, const QUAD *quad, DOF_REAL_VEC *fh);
void L2scp_fct_bas_d(FCT_D_AT_X f, const QUAD *, DOF_REAL_D_VEC *fh);
void L2scp_fct_bas_dow(FCT_D_AT_X f, const QUAD *quad, DOF_REAL_VEC_D *fh);

void L2scp_fct_bas_loc(DOF_REAL_VEC *fh,
                       LOC_FCT_AT_QP f, void *f_data, FLAGS fill_flag,
                       const QUAD *quad);
void L2scp_fct_bas_loc_d(DOF_REAL_D_VEC *fh,
```

```
                              LOC_FCT_D_AT_QP f, void *fd, FLAGS fill_flag,
                              const QUAD *quad);
void L2scp_fct_bas_loc_dow(DOF_REAL_VEC_D *fh,
                             LOC_FCT_D_AT_QP f, void *fd, FLAGS fill_flag,
                             const QUAD *quad);

void H1scp_fct_bas(GRD_FCT_AT_X grd_f,
                   const QUAD *quad, DOF_REAL_VEC *fh);
void H1scp_fct_bas_d(GRD_FCT_D_AT_X grd_f,
                     const QUAD *quad, DOF_REAL_D_VEC *fh);
void H1scp_fct_bas_dow(GRD_FCT_D_AT_X grd_f,
                       const QUAD *quad, DOF_REAL_VEC_D *fh);

void H1scp_fct_bas_loc(DOF_REAL_VEC *fh,
                       GRD_LOC_FCT_AT_QP grd_f, void *fd,
                       FLAGS fill_flag, const QUAD *quad);
void H1scp_fct_bas_loc_d(DOF_REAL_VEC_D *fh,
                         GRD_LOC_FCT_D_AT_QP grd_f, void *fd,
                         FLAGS fill_flag, const QUAD *quad);
void H1scp_fct_bas_loc_dow(DOF_REAL_VEC_D *fh,
                           GRD_LOC_FCT_D_AT_QP grd_f, void *fd,
                           FLAGS fill_flag, const QUAD *quad);
```

Descriptions

`L2scp_fct_bas(f, quad, fh)`

`L2scp_fct_bas_d(f, quad, fh)`

`L2scp_fct_bas_dow(f, quad, fh)` Approximate the L^2 scalar products of a given function with all basis functions by numerical quadrature and add 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 `NULL` 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 added;

`quad` is the quadrature for the approximation of the integral on each leaf element of `fh->fe_space->mesh`; if `quad` is a `NULL` 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 `NULL`, 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 `NULL` 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 added;
`quad` is the quadrature for the approximation of the integral on each leaf element of `fhd->fe_space->mesh`; if `quad` is a `NULL` 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.

`L2scp_fct_bas_dow(fd, quad, fhd)`

`L2scp_fct_bas_loc(fh, f_at_qp, fct_data, fill_flag, quad)`

`L2scp_fct_bas_loc_dow(fh, f_at_qp, ud, fill_flag, quad)`

`H1scp_fct_bas(grd_f, quad, fh)`

`H1scp_fct_bas_dow(grd_fd, quad, fhd)`

4.7.7 Boundary conditions

The following six functions act as a front-end to the functions explained further below, therefore we refer the reader to Section 4.7.7.1, 4.7.7.2 and 4.7.7.3 for a deeper discussion of the implementation of Dirichlet, Neumann and Robin boundary conditions within ALBERTA.

```
bool boundary_conditions(DOF_MATRIX *matrix, DOF_REAL_VEC *fh,
                         DOF_REAL_VEC *uh, DOF_SCHAR_VEC *bound,
                         const BNDRY_FLAGS dirichlet_segment,
                         REAL (*g)(const REAL_D x),
                         REAL (*gn)(const REAL_D x, const REAL_D normal),
                         REAL alpha_r, const WALL_QUAD *wall_quad);
bool boundary_conditions_loc(DOF_MATRIX *matrix, DOF_REAL_VEC *fh,
                             DOF_REAL_VEC *uh, DOF_SCHAR_VEC *bound,
                             const BNDRY_FLAGS dirichlet_segment,
                             LOC_FCT_AT_QP g_at_qp, LOC_FCT_AT_QP gn_at_qp,
                             void *app_data, FLAGS fill_flags,
                             REAL alpha_r, const WALL_QUAD *wall_quad);
bool boundary_conditions_d(DOF_MATRIX *matrix, DOF_REAL_D_VEC *fh,
                           DOF_REAL_D_VEC *uh, DOF_SCHAR_VEC *bound,
                           const BNDRY_FLAGS dirichlet_segment,
                           const REAL *(*g)(const REAL_D x, REAL_D res),
                           const REAL *(*gn)(const REAL_D x,
                                             const REAL_D normal,
                                             REAL_D res),
                           REAL alpha_r, const WALL_QUAD *wall_quad);
bool boundary_conditions_loc_d(DOF_MATRIX *matrix, DOF_REAL_D_VEC *fh,
                               DOF_REAL_D_VEC *uh, DOF_SCHAR_VEC *bound,
                               const BNDRY_FLAGS dirichlet_segment,
                               LOC_FCT_D_AT_QP g_at_qp,
                               LOC_FCT_D_AT_QP gn_at_qp,
                               void *app_data, FLAGS fill_flags,
                               REAL alpha_r, const WALL_QUAD *wall_quad);
bool boundary_conditions_dow(DOF_MATRIX *matrix, DOF_REAL_VEC_D *fh,
                             DOF_REAL_VEC_D *uh, DOF_SCHAR_VEC *bound,
```

```
                                    const BNDRY_FLAGS dirichlet_segment,
                                    const REAL *(*g)(const REAL_D x, REAL_D res),
                                    const REAL *(*gn)(const REAL_D x,
                                                      const REAL_D normal,
                                                      REAL_D res),
                                    REAL alpha_r, const WALL_QUAD *wall_quad);
bool boundary_conditions_loc_dow(DOF_MATRIX *matrix, DOF_REAL_VEC_D *fh,
                                 DOF_REAL_VEC_D *uh, DOF_SCHAR_VEC *bound,
                                 const BNDRY_FLAGS dirichlet_segment,
                                 LOC_FCT_D_AT_QP g_at_qp,
                                 LOC_FCT_D_AT_QP gn_at_qp,
                                 void *app_data, FLAGS fill_flags,
                                 REAL alpha_r, const WALL_QUAD *wall_quad);
```

Description: These "compound" functions implement Dirichlet, Neumann or Robin boundary conditions, and optionally perform a mean-value correction of the "right hand side" in the context of pure Neumann boundary conditions if `alpha_r` < 0 (in order to satisfy the conditions for the "right hand side" which may be violated in the discrete context because of quadrature errors).

For the differences between the code..._loc() and non-`..._loc()` versions the reader is referred to the section dealing with `dirichlet_bound_loc()` (see Section 4.7.7.1). A brief discussion of the calling convention for the various functions pointers passed to the library functions can also be found in Section 4.5.

Parameters

`matrix` As explained in Section 4.7.7.3, passed on to `robin_bound()`.

`fh` As explained in Section 4.7.7.1, Section 4.7.7.2 and Section 4.7.7.3. Passed on to `dirichlet_bound()` and `bndry_L2scp_fct_bas()`.

`uh` As explained in Section 4.7.7.1. Passed on to `dirichlet_bound()`.

`bound` As explained in Section 4.7.7.1. Passed on to `dirichlet_bound()`.

`dirichlet_segment` As explained in Section 4.7.7.1. Passed on to `dirichlet_bound()`. The respective bit-masks passed to `bndry_L2scp_fct_bas()` and `robin_bound()` are computed as bit-wise complement of `dirichlet_segment`. See also Section 3.2.4.

`g` As explained in Section 4.7.7.1. Passed on to `dirichlet_bound()`.

`gn` As explained in Section 4.7.7.2, Section 4.7.7.3. Passed on to `bndry_L2scp_fct_bas()`.

`app_data` `..._loc()`-variants only. As explained in Section 4.7.7.1, Section 4.5. Passed on as application-data pointer to the application provided function hooks.

`fill_flags` `..._loc()`-variants only. Additional fill-flags needed by `g()` or `gn`.

`alpha_r` As explained in Section 4.7.7.3. Passed on to `robin_bound()`. `alpha_r` is also abused to request an automatic mean-value correction of the load-vector: if `alpha_r` is negative, and Neumann boundary conditions were imposed on all boundary segments, then `boundary_conditions()` will attempt such a mean-value correction in order to keep fulfill the compatibility condition for the load-vector in the discrete setting. Of course, if the differential operator has lower order parts, then the load-vector need not have mean-value 0.

Robin boundary conditions will only be assembled if `alpha_r` is strictly larger than 0.

`wall_quad` As explained in Section 4.7.7.3 and Section 4.7.7.2. Passed on to `robin_bound()` and `bndry_L2scp_fct_bas()`.

Return Value

`true` if any part of the boundary was subject to Dirichlet boundary conditions.

4.7.7.1 Dirichlet boundary conditions

For the solution of the discrete system (1.13) on page 25 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 start value for an iterative solver. This way the first matrix-vector operation performed by an iterative solver will have the effect to transform an inhomogeneous Dirichlet boundary problem to a homogeneous one by applying the differential operator to the boundary values and subtracting the result from the "right hand side". Whether or not it is also necessary to copy these coefficients to the load vector depends on how the matrices act on the coefficients:

- If the matrix-rows corresponding to Dirichlet-nodes $k_1, \ldots, k_M$ have been replaced by unit-vectors e_{k_l} $(1 \le l \le M)$, then it is also necessary to copy the Dirichlet nodes to the load vector (compare (1.12) on page 25). 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 zero at all Dirichlet `DOF`s.

  This is the case when Dirichlet bit-masks have been copied to `OPERATOR_INFO.dirichlet_bndry` (compare Section 3.2.4 and 4.50); the resulting `DOF_MATRIX` will then be assembled (Section 4.7.2) in this way, replacing any row corresponding to a Dirichlet-node by the corresponding unit-vector.

- If the matrix-rows corresponding to Dirichlet-nodes have not been replaced by unit-vectors, then it is still possible to solve a Dirichlet-problem by passing a `DOF_SCHAR_VEC` to the matrix-vector routines (compare Section 4.10, describing the linear solvers available in ALBERTA). However, in this case the matrix-vector subroutines will clear all Dirichlet-nodes to zero, see Section 3.3.7. Therefore, in this case it is necessary to clear the coefficients of the "right hand side" which correspond to Dirichlet-nodes. See the Example 4.7.6 for simple examples how to perform this task.

Note that the matrices generated this way – either by clearing Dirichlet-rows or by masking out Dirichlet rows – are not symmetric (compare also (1.11) on page 24) even if the underlying differential operator is symmetric. However, the restriction of the matrix to the space spanned by the non-Dirichlet `DOFs` *is* symmetric, so any iterative solver for symmetric matrices will work, provided one either sets the Dirichlet-values also in the load-vector (if the matrix acts as identity on the Dirichlet `DOF`s) or clears the Dirichlet-nodes in the load-vector (if the matrix acts as zero-operator on the Dirichlet `DOF`s).

The following functions will set Dirichlet boundary values for all DOFs on the Dirichlet boundary, using an interpolation of the boundary values g:

```
bool dirichlet_bound(DOF_REAL_VEC *fh, DOF_REAL_VEC *uh,
                     DOF_SCHAR_VEC *bound,
                     const BNDRY_FLAGS dirichlet_segments,
                     REAL (*g)(const REAL_D));
bool dirichlet_bound_d(DOF_REAL_VEC_D *fh, DOF_REAL_VEC_D *uh,
                       DOF_SCHAR_VEC *bound,
                       const BNDRY_FLAGS dirichlet_segments,
                       const REAL *(*g)(const REAL_D, REAL_D));
bool dirichlet_bound_dow(DOF_REAL_VEC_D *fh, DOF_REAL_VEC_D *uh,
                         DOF_SCHAR_VEC *bound,
                         const BNDRY_FLAGS dirichlet_segments,
                         const REAL *(*g)(const REAL_D, REAL_D));
bool dirichlet_bound_loc(DOF_REAL_VEC *fh, DOF_REAL_VEC *uh,
                         DOF_SCHAR_VEC *bound,
                         const BNDRY_FLAGS dirichlet_segments,
                         LOC_FCT_AT_QP g, void *ud, FLAGS fill_flags);
bool dirichlet_bound_loc_d(DOF_REAL_VEC_D *fh, DOF_REAL_VEC_D *uh,
                           DOF_SCHAR_VEC *bound,
                           const BNDRY_FLAGS dirichlet_segments,
                           LOC_FCT_D_AT_QP g, void *ud,
                           FLAGS fill_flags);
bool dirichlet_bound_loc_dow(DOF_REAL_VEC_D *fh, DOF_REAL_VEC_D *uh,
                             DOF_SCHAR_VEC *bound,
                             const BNDRY_FLAGS dirichlet_segments,
                             LOC_FCT_D_AT_QP g, void *ud,
                             FLAGS fill_flags);
```

Descriptions

`dirichlet_bound(fh, uh, bound, dirichlet_segments, g)` 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.

Boundary values are set element-wise on the leaf elements. The boundary type of the walls of an element is accessed through the function `wall_bound(el_info, wall)`. If the corresponding bit is set in `dirichlet_segments`, then the local interpolation operator of the basis functions underlying `fh/uh->fe_space` is invoked to compute the coefficients of the `DOF`s located on that wall.

This variant of the `dirichlet_bound...()` is rather simplistic; the `dirichlet_bound_loc..()` pass more information to the function implementing the boundary values and also allow for manipulating the amount of information available while looping over the mesh.

Parameters

`fh, 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 `NULL` pointer; if both arguments are `NULL` pointers, nothing is done, except of filling the `bound` argument, it that is non `NULL`; if both arguments are not `NULL`, `fh->fe_space` must equal `uh->fe_space`.

`bound` is a vector for storing the boundary type for each used DOF; `bound` may be `NULL`; if it is not `NULL`, 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_segments Bit-mask marking those parts of the boundary which are subject to Dirichlet boundary conditions. If bit number $k > 0$ is set in dirichlet_segments then the part of the boundary with boundary classification k is marked as Dirichlet boundary. Compare Section 3.2.4.

REAL (*g)(const REAL_D arg) is a pointer to a function for the evaluation of the boundary data; if g is a NULL pointer, all coefficients at Dirichlet DOFs are set to 0.0. arg are the Cartesian co-ordinates where the value of g should be computed.

Return Value true if any part of the boundary of the mesh is subject to Dirichlet boundary conditions, as indicated by the argument dirichlet_segments, false otherwise.

---

dirichlet_bound_d(fh, uh, bound, dirichlet_segments, g) does the same as dirichlet_bound(), but fh and uh are DOF_REAL_D_VEC vectors.

The calling convention for const REAL (*g)(const REAL_D arg, REAL_D result) is such that g must allow for result being a NULL-pointer. If so, a pointer to a statically allocated storage area must be returned, otherwise result must be filled with the value of g at the evaluation point arg, see Example 4.5.5 in Section 4.5. Otherwise everything works as for dirichlet_bound(), see above for the documentation.

---

dirichlet_bound_dow(fh, uh, bound, dirichlet_segments, g) does the same as dirichlet_bound_d(), but fh and uh are DOF_REAL_VEC_D vectors, that is, uh and fh may belong to a finite element space which is a direct sum, composed of several finite element spaces (note the location of the _D suffix in the data-type names DOF_REAL_VEC_D and DOF_REAL_D_VEC!). The calling convention for const REAL (*g)(const REAL_D arg, REAL_D result) is the same as explained above for dirichlet_bound_d().

---

dirichlet_bound_loc(fh, uh, bound, dirichlet_segments, g, ud, fill_flags) This function differs from its counterpart without the _loc-suffix only in the calling convention for the function implementing the Dirichlet boundary conditions. We document only the differing or additional arguments here and refer the reader to the documentation of dirichlet_bound() above:

REAL (*g)(const EL_INFO *el_info, const QUAD *quad, int iq, void *ud) The function pointer to the function implementing the Dirichlet boundary values. In contrast to the corresponding function-pointer passed to dirichlet_bound() this function is invoked with a co-dimension 1 quadrature rule (compare the interpol-hooks in the BAS_FCTS structure, 3.89, and the definition of the QUAD-structure, 4.2) and a quadrature point, and first and not least with a filled EL_INFO-structure.

This means that g has full-access to all the information available through the EL_INFO element descriptor. The amount of data filled-in during mesh-traversal can additionally be controlled by setting specific fill-flags through the argument fill_flags, which is passed as last argument to dirichlet_bound_loc(). The last argument ud to g is the same as the pointer ud passed as pre-last argument to dirichlet_bound_loc() and may be used by an application to reduce the amount of global variables and thus the potential of bugs implied by the use of global variables. The following simple example shows how to get hold of the Cartesian co-ordinates of the quadrature

point, and how to use, e.g., the boundary classification available through the EL_INFO structure:

4.7.5 Example.

```
struct g_data
{
  BNDRY_TYPE special_wall_type; /* other stuff */
};

REAL g_impl(const EL_INFO *el_info, const QUAD *quad, int iq,
    void *ud)
{
  struct g_data *data = (struct g_data *)ud;
  BNDRY_TYPE btype = wall_bound(el_info, quad->sub_splx);
  REAL result;
  const QUAD_EL_CACHE *qelc =
    fill_quad_el_cache(el_info, quad, FILL_EL_QUAD_WORLD);

  if (btype == data->special_wall_type) {
    ... /* do special things */
    return sin(qelc->world[iq][0];
  } else {
    ... /* do "normal" things */
    return sin(qelc->world[iq][1];
  }
}

... /* 1.000.000 lines of code later ... */
struct g_data g_data_instance = { 42 };
dirichlet_bound_loc(fh, uh, bound, dirichlet_bits, g_impl,
   &g_data_instance, FILL_COORDS|FILL_MACRO_WALLS);
```

ud Application-data-pointer, forwarded as ud argument to the application supplied g function-pointer.

fill_flags Additional fill-flags for the loop over the mesh. The application must make sure that fill_flags contains all flags corresponding to information needed by the function g().

---

dirichlet_bound_loc_dow(
fh, uh, bound, dirichlet_segments, g, ud, fill_flags)

dirichlet_bound_loc(fh, uh, bound, dirichlet_segments, g, ud, fill_flags)

These two function differ from dirichlet_bound_loc() only in the calling convention for

```
const REAL *(*g)(REAL_D result, const EL_INFO *el_info, const QUAD
    *quad, int iq, void *ud).
```

As in the example 4.33 the implementation of g() must allow for result being NULL, returning a pointer to a static storage area in this case.

4.7.6 Example. This example demonstrates how to clear the Dirichlet-nodes in the load-vector if Dirichlet boundary conditions are implemented using a DOF_SCHAR_VEC to mask-out

Dirichlet nodes. Note that this example applies *only* if the DOF_SCHAR_VEC is also passed to the linear solvers. Otherwise Dirichlet boundary conditions have to be incorporated into the matrix

scalar problem

```
extern REAL g(const REAL_D x); /* defined somewhere else, e.g. */
extern DOF_REAL_VEC *uh, *fh;  /* defined somewhere else, e.g. */
DOF_SCHAR_VEC *bound =
  get_dof_schar_vec("dirichlet_mask_vector", fh->fe_space);
BNDRY_FLAGS dirichlet_bits;
BNDRY_FLAGS_INIT(dirichlet_bits);
BNDRY_FLAGS_SET(dirichlet_bits, 3); /* e.g. */

... /* other stuff */

dirichlet_bound(NULL, uh, bound, dirichlet_bits, g);
FOR_ALL_DOFS(fh->fe_space->admin,
             if (bound->vec[dof] >= DIRICHLET) {
               fh->vec[dof] = 0.0;
             });

... /* other stuff */

oem_solve_s(matrix, bound, fh, uh, ... /* other parameters */);
```

simple vector valued problem

```
extern const REAL *g(const REAL_D x, REAL_D result); /* defined
    somewhere else, e.g. */
extern DOF_REAL_D_VEC *uh, *fh; /* defined somewhere else, e.g. */
extern DOF_SCHAR_VEC *bound;
extern BNDRY_FLAGS dirichlet_bits;

... /* other stuff */

dirichlet_bound_d(NULL, uh, bound, dirichlet_bits, g);
FOR_ALL_DOFS(fh->fe_space->admin,
             if (bound->vec[dof] >= DIRICHLET) {
               SET_DOW(0.0, fh->vec[dof]);
             });

... /* other stuff */

oem_solve_d(matrix, bound, fh, uh, ... /* other parameters */);
```

vector valued problem, using an FE-space which is a direct sum

Note the difference between a DOF_REAL_D_VEC which contains DIM_OF_WORLD-sized REAL_D vectors and a DOF_REAL_VEC_D which contains scalars of type REAL if the underlying basis function are themselves vector-valued, or REAL_D-vectors if the underlying basis functions are scalar-valued. The first code-block of the FOREACH_DOF_DOW-macro is for the case where the basis functions are vector-valued (and hence the coefficients are scalars) and the second code-block is for the case where the basis functions are

scalar-valued (and hence the coefficients are vectors). The `FOREACH_DOF_DOW()` macro is further explained in Section 3.7.2.

```
extern const REAL *g(const REAL_D x, REAL_D result); /* defined
    somewhere else, e.g. */
extern DOF_REAL_VEC_D *uh, *fh; /* defined somewhere else, e.g. */
extern DOF_SCHAR_VEC *bound;
extern BNDRY_FLAGS dirichlet_bits;

... /* other stuff */

dirichlet_bound_dow(NULL, uh, bound, dirichlet_bits, g);
FOREACH_DOF_DOW(fh->fe_space,
                if (bound->vec[dof] >= DIRICHLET) {
                  fh->vec[dof] = 0.0;
                },
                if (bound->vec[dof] >= DIRICHLET) {
                  SET_DOW(0.0, ((DOF_REAL_D_VEC *)fh)->vec[dof]);
                },
                fh = CHAIN_NEXT(fh, DOF_REAL_VEC_D);
                bound = CHAIN_NEXT(bound, DOF_SCHAR_VEC));

... /* other stuff */

oem_solve_dow(matrix, bound, fh, uh, ... /* other parameters */);
```

4.7.7.2 Neumann boundary conditions

For the implementation of inhomogeneous Neumann boundary conditions it is necessary to compute L^2 scalar products between the inhomogeneity and the basis functions on the Neumann boundary segments. The following functions compute the L^2 scalar product over the boundary of the domain. They return `true` if not all boundary segments of the mesh belong to the segment specified by `bndry_seg`. If `bndry_seg == NULL` then the scalar product is computed over the entire boundary (i.e. over all walls without neighbour). Besides computing the L^2-scalar product over boundary segments there are also functions to compute the L^2-scalar-product over trace meshes (or "sub-meshes", see Section 3.9). For the calling conventions for the application provided function pointers the reader is referred to Section 4.5, and the relevant part of the discussion of `dirichlet_bound_loc()` in Section 4.7.7.1.

All function work additive, the contributions of the per-element integrals are added to any data already present in `fh`.

Prototypes

```
bool bndry_L2scp_fct_bas_loc(DOF_REAL_VEC *fh,
                             LOC_FCT_AT_QP f_at_qp, void *ud, FLAGS
                                 fill_flag,
                             const BNDRY_FLAGS bndry_seg,
                             const WALL_QUAD *quad);
bool bndry_L2scp_fct_bas_loc_dow(DOF_REAL_VEC_D *fh, LOC_FCT_D_AT_QP f_at_qp,
                                 void *ud, FLAGS fill_flag,
                                 const BNDRY_FLAGS bndry_seg,
                                 const WALL_QUAD *quad);
```

```
bool bndry_L2scp_fct_bas_dow(DOF_REAL_VEC_D *fh,
                             const REAL *(*f)(const REAL_D x,
                                              const REAL_D normal,
                                              REAL_D result),
                             const BNDRY_FLAGS bndry_seg,
                             const WALL_QUAD *quad);
bool bndry_L2scp_fct_bas(DOF_REAL_VEC *fh,
                         REAL (*f)(const REAL_D x, const REAL_D normal),
                         const BNDRY_FLAGS bndry_seg, const WALL_QUAD *quad);

void trace_L2scp_fct_bas(DOF_REAL_VEC *fh, FCT_AT_X f,
                         MESH *trace_mesh, const QUAD *quad);
void trace_L2scp_fct_bas_loc(DOF_REAL_VEC *fh,
                             LOC_FCT_AT_QP f, void *fd, FLAGS fill_flag,
                             MESH *trace_mesh,
                             const QUAD *quad);
void trace_L2scp_fct_bas_dow(DOF_REAL_VEC_D *fh, FCT_D_AT_X f,
                             MESH *trace_mesh,
                             const QUAD *quad);
void trace_L2scp_fct_bas_loc_dow(DOF_REAL_VEC_D *fh,
                                 LOC_FCT_D_AT_QP f, void *fd, FLAGS
                                     fill_flag,
                                 MESH *trace_mesh,
                                 const QUAD *quad);
```

Descriptions

`bndry_L2scp_fct_bas()`

Parameters

`fh` The load-vector to add the boundary integrals to.

`f` Application supplied "right hand side".

`ud` Data pointer for `f` for the `..._loc()`-variants.

`fill_flags` Additional fill-flags needed by `f` for the `..._loc()`-variants.

`bndry_seg` A bit-mask specifying the part of the boundary which is the domain of integration. See Section 3.2.4.

`quad` Optional application supplied quadrature rule. Maybe `NULL`, in which case a default quadrature rule is used. See Section 4.2.4 for how to obtain such a structure, function `get_wall_quad()`.

Return Value

`true` if part of the boundary did *not* belong to the domain of integration.

`trace_L2scp_fct_bas()`

Parameters

`fh` The load-vector.

`f` The user-supplied inhomogeneity.

`fd` The application data pointer passed on to `f` for the `..._loc()`-variants.

`fill_flags` Additional fill-flags for the `..._loc()`-variants.

trace_mesh The domain of integration.

quad A user supplied quadrature rule. May be NULL in which case a default quadrature rule will be used. The quadrature rule must have the dimension of trace_mesh, naturally.

Return Value

4.7.7.3 Robin boundary conditions

```
void robin_bound(DOF_MATRIX *matrix, const BNDRY_FLAGS robin_seg,
                 REAL alpha_r, const WALL_QUAD *wall_quad,
                 REAL exponent);
```

Description: Incorporate so-called "Robin boundary" conditions into the matrix, i.e. a boundary condition of the form

$$\frac{\partial u}{\partial \nu}(x) + \alpha(x)\, u(x) = g(x) \quad \text{on } \partial\Omega.$$

In the context of weak formulations for elliptic second-order PDEs, this results into two boundary integrals, one has to be added to the linear form on the "right hand side", and the other one is a contribution to bilinear-form on the "left hand side", namely

$$\int_{\partial\Omega} \alpha\, u\, \phi\, do \quad \text{and} \quad \int_{\partial\Omega} g\, \phi\, do$$

The contribution to the right hand side can be assembled using one of the bndry_L2scp_fct_bas()-variants, the contribution the left hand side should be added to the system matrix. robin_bound() implements this for the case where α is actually just a constant coefficient.

robin_bound(matrix, robin_seg, alpha_r, wall_quad, exponent)

Parameters

matrix The system matrix, the contributions from the Robin boundary condition are added to matrix.

robin_segment A boundary bit-mask, marking all boundary segments which are subject to the Robin boundary condition. The position of the bits set in robin_segment correspond to the boundary numbers assigned to the mesh boundary in the macro triangulation, compare Section 3.2.15 and Section 3.2.4.

alpha_r The constant coefficient from the Robin boundary condition.

wall_quad Optional. A collection of co-dimension 1 quadrature formulas for doing the integration. If wall_quad == NULL, then robin_bound() chooses a default quadrature formula, based on the polynomial degree of the underlying basis-functions.

exponent If exponent > 0.0, then the boundary integral will be weighted by the factor $h(T)^{-\text{exponent}}$, where $h(T)$ denotes the local mesh-width.

4.7.8 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(FCT_AT_X f, DOF_REAL_VEC *fh);
void interpol_d(const REAL *(*f)(const REAL_D, REAL_D), DOF_REAL_D_VEC *fh);
void interpol_dow(FCT_D_AT_X f, DOF_REAL_VEC_D *fh);
void interpol_loc(DOF_REAL_VEC *fh,
                  LOC_FCT_AT_QP f, void *f_data, FLAGS fill_flags);
void interpol_loc_d(DOF_REAL_D_VEC *fh,
                    LOC_FCT_D_AT_QP f, void *f_data, FLAGS fill_flags);
void interpol_loc_dow(DOF_REAL_VEC_D *fh,
                      LOC_FCT_D_AT_QP f, void *f_data, FLAGS fill_flags);
```

Description:

interpol(f, fh) computes the coefficients of the interpolant of a function and stores these in a DOF vector;

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 3.5.1).

Parameters

f is a pointer to a function for the evaluation of the function to be interpolated; if `f` is a `NULL` pointer, all coefficients are set to `0.0`.

fh is a DOF vector for storing the coefficients; if `fh` is a `NULL` pointer, nothing is done.

interpol_d(fd, fhd) computes the coefficients of the interpolant of a vector valued function and stores these in a DOF vector. This version is for the case where the underlying basis-functions are themselves scalars, consequently the coefficient vector `fh` has the type `DOF_REAL_D_VEC`. Otherwise this function differs from the scalar counter-part only in the calling convention for the application supplied function `f`, which is the same as for `dirichlet_bound_d()`, see also Example 4.5.5

interpol_dow(fct, uh) same as `interpol_d()`, but for the case where the underlying basis function are either scalar- or `DIM_OF_WORLD`-valued and the finite-element space may optionally be a direct sum of finite element spaces.

interpol_loc(vec, fct_at_qp, app_data, fill_flags)

interpol_loc_d(vec, fct_at_qp, app_data, fill_flags)

interpol_loc_dow(vec, fct_at_qp, app_data, fill_flags) The `..._loc...`-variants differ from the other `interpol()`-flavours only in the calling convention for the application supplied function and the additional `fill_flags` argument. This has already be explained above, see also Example 4.7.5.

4.8 Data structures and procedures for adaptive methods

4.8.1 ALBERTA adaptive method for stationary problems

The basic data structure ADAPT_STAT for stationary adaptive methods contains pointers to problem dependent routines to build the linear or nonlinear system(s) of equations on an adapted mesh, and to a routine which solves the discrete problem and computes the new discrete solution(s). 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.

ADAPT_STAT also holds parameters used for the adaptive procedure. Some of the parameters are optional or used only when a special marking strategy is selected.

```
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:

name textual description of the adaptive method, or NULL.

tolerance given tolerance for the (absolute or relative) error.

p power p used in estimate (1.23), $1 \leq p < \infty$.

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 problem dependent function for computing the global error estimate and the local error indicators; must not be `NULL`;

`estimate(mesh, adapt)` computes the error estimate and fills the entries `adapt->err_sum` and `adapt->err_max` with

$$\texttt{adapt->err\_sum} = \Big(\sum_{S \in \mathcal{S}_h} \eta_S(u_h)^p \Big)^{1/p}, \quad \texttt{adapt->err\_max} = \max_{S \in \mathcal{S}_h} \eta_S(u_h)^p.$$

The return value is the total error estimate `adapt->err_sum`. User data, like additional parameters for `estimate()`, can be passed via the `est_info` entry of the `ADAPT_STAT` structure to a (problem dependent) parameter structure. Usually, `estimate()` stores the local error indicator(s) $\eta_S(u_h)^p$ (and coarsening error indicator(s) $\eta_{c,S}(u_h)^p$) in `LEAF_DATA(el)`. For sample implementations of error estimators for quasi-linear elliptic and parabolic problems, see Section 4.9.

get_el_est pointer to a problem dependent subroutine returning the value of the local error indicator; must not be `NULL` if via the entry `strategy` adaptive refinement is selected and the specialized marking routine `marking` is `NULL`;

`get_el_est(el)` returns the value $\eta_S(u_h)^p$, of the local error indicator on leaf element `el`; usually, local error indicators are computed by `estimate()` and stored in `LEAF_DATA(el)`, which is problem dependent and thus not directly accessible by general–purpose routines. `get_el_est()` is needed by the ALBERTA marking strategies.

get_el_estc pointer to a function which returns the local coarsening error indicator;

`get_el_estc(el)` returns the value $\eta_{c,S}(u_h)^p$ of the local coarsening error indicator on leaf element `el`, usually computed by `estimate()` and stored in `LEAF_DATA(el)`; if not `NULL`, `get_el_estc()` is called by the ALBERTA marking routines; this pointer may be `NULL`, which means $\eta_{c,S}(u_h) = 0$.

marking specialized marking strategy; if `NULL`, a standard ALBERTA marking routine is selected via the entry `strategy`;

`marking(mesh, adapt)` selects and marks elements for refinement or coarsening; the return value is

0 no element is marked;

MESH_REFINED elements are marked but only for refinement;

MESH_COARSENED elements are marked but only for coarsening;

MESH_REFINED|MESH_COARSENED elements are marked for refinement and coarsening.

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 `NULL`.

err_sum variable to hold the sum of local error indicators $(\sum_{S \in \mathcal{S}} \eta_S(u_h)^p)^{1/p}$; the value for this entry must be set by the 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 must be set by the function `estimate()`.

build_before_refine pointer to a subroutine that builds parts of the (non-)linear system(s) before any mesh adaptation; if it is NULL, this assemblage stage omitted;

build_before_refine(mesh, flag) launches the assembling of the assembling of the discrete system on mesh; flag gives information which part of the system has to be built; the mesh will be refined if the MESH_REFINED bit is set in flag and it will be coarsened if the bit MESH_COARSENED is set in flag.

build_before_coarsen pointer to a subroutine that builds parts of the (non-)linear system(s) between the refinement and coarsening; if it is NULL, this assemblage stage omitted;

build_before_coarsen(mesh, flag) performs an intermediate assembling step on mesh (compare Section 1.4.4 for an example when such a step is needed); flag gives information which part of the system has to be built; the mesh was refined if the MESH_REFINED bit is set in flag and it will be coarsened if the bit MESH_COARSENED is set in flag.

build_after_coarsen pointer to a subroutine that builds parts of the (non-)linear system(s) after all mesh adaptation; if it is NULL, this assemblage stage omitted;

build_before_coarsen(mesh, flag) performs the final assembling step on mesh; flag gives information which part of the system has to be built; the mesh was refined if the MESH_REFINED bit is set in flag and it was coarsened if the bit MESH_COARSENED is set in flag.

solve pointer to a subroutine for solving the discrete (non-)linear system(s); if it is NULL, the solution step is omitted;

solve(mesh) computes the new discrete solution(s) on mesh.

refine_bisections number of bisection steps for the refinement of an element marked for refinement; used by the ALBERTA marking strategies; default value is d.

coarsen_allowed flag used by the ALBERTA marking strategies to allow (true) or forbid (false) mesh coarsening;

coarse_bisections number of bisection steps for the coarsening of an element marked for coarsening; used by the ALBERTA marking strategies; default value is d.

strategy parameter to select an ALBERTA marking routine; 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 4.8.2.

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.

The routine adapt_method_stat() implements the whole adaptive procedure for a stationary problem, using the parameters given in ADAPT_STAT:

```
void adapt_method_stat(MESH *, ADAPT_STAT *);
```

Description:

adapt_method_stat(mesh, adapt_stat) solves adaptively a stationary problem on `mesh` by the adaptive procedure described in Section 1.5.1; `adapt_stat` is a pointer to a filled `ADAPT_STAT` data structure, holding all information about the problem to be solved and parameters for the adaptive method.

The main loop of the adaptive method is given in the following source fragment:

```
void adapt_method_stat(MESH *mesh, ADAPT_STAT *adapt)
{
  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 subroutine `adapt_mesh()`, which combines marking, refinement, coarsening and the linear system building routines:

```
static U_CHAR adapt_mesh(MESH *mesh, ADAPT_STAT *adapt)
{
  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);
  }
```

4.8.1 Remark. As the same procedure is used for time dependent problems in single time steps, different pointers to routines for building parts of the (non-)linear systems make it possible, for example, to assemble 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 losing any information, compare Section 1.4.4.

4.8.2 Remark. For time dependent problems, the system matrices usually 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.

4.8.2 Standard ALBERTA marking routine

When the `marking` procedure pointer in the `ADAPT_STAT` structure is `NULL`, then the standard ALBERTA marking routine is called. The `strategy` entry, allows the selection of one of five different marking strategies (compare Sections 1.5.2 and 1.5.3). Elements are only marked for coarsening and coarsening parameters are only used if the entry `coarsen_allowed` is `true`. The number of bisection steps for refinement and coarsening is selected by the entries `refine_bisections` and `coarse_bisections`.

`strategy=0`: no refinement or coarsening is performed;

`strategy=1`: Global Refinement (GR):
the mesh is refined globally, no coarsening is performed;

`strategy=2`: Maximum Strategy (MS):
the entries `MS_gamma`, `MS_gamma_c` are used as refinement and coarsening parameters;

`strategy=3`: Equidistribution strategy (ES):
the entries `ES_theta`, `ES_theta_c` are used as refinement and coarsening parameters;

`strategy=4`: Guaranteed error reduction strategy (GERS):
the entries `GERS_theta_star`, `GERS_nu`, and `GERS_theta_c` are used as refinement and coarsening parameters.

4.8.3 Remark. 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 for the equidistribution strategy described in Section 1.5.3. The implemented equidistribution strategy uses the inequality

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

instead of

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

4.8.3 ALBERTA adaptive method for time dependent problems

Similar to the data structure `ADAPT_STAT` for collecting information about the adaptive solution for a stationary problem, the data structure `ADAPT_INSTAT` is used for gather all information needed for the time and space adaptive solution of instationary problems. Using a time stepping scheme, in each time step a stationary problem is solved; the adaptive method for this is based on the `adapt_method_stat()` routine described in Section 4.8.1, the `ADAPT_INSTAT` structure includes two `ADAPT_STAT` parameter structures. Additional entries give information about the time adaptive procedure.

```
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 *, ADAPT_INSTAT *);
  void   (*set_time)(MESH *, ADAPT_INSTAT *);
  void   (*one_timestep)(MESH *, ADAPT_INSTAT *);
  REAL   (*get_time_est)(MESH *, ADAPT_INSTAT *);
  void   (*close_timestep)(MESH *, ADAPT_INSTAT *);

  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:

name textual description of the adaptive method, or NULL.

adapt_initial mesh adaption parameters for the initial mesh, compare Section 4.8.1.

adapt_space mesh adaption parameters during time steps, compare Section 4.8.1.

time actual time, end of time interval for current time step.

start_time initial time for the adaptive simulation.

end_time final time for the adaptive simulation.

timestep current time step size, will be changed by the time adaptive method.

init_timestep pointer to a routine called at beginning of each time step; if NULL, initialization of a new time step is omitted;
`init_timestep(mesh, adapt)` initializes a new time step;

set_time pointer to a routine called after changes of the time step size if not NULL;
`set_time(mesh, adapt)` is called by the adaptive method each time when the actual time `adapt->time` has changed, i. e. at a new time step and after a change of the time step size `adapt->timestep`; information about actual time and time step size is available via `adapt`.

one_timestep pointer to a routine which implements one (adaptive) time step, if NULL, a default routine is called;
`one_timestep(mesh, adapt)` implements the (adaptive) solution of the problem in one single time step; information about the stationary problem of the time step is available in the `adapt->adapt_space` data structure.

get_time_est pointer to a routine returning an estimate for the time error; if NULL, no time step adaptation is done;
`get_time_est(mesh, adapt)` returns an estimate η_τ for the current time error at time `adapt->time` on `mesh`.

close_timestep pointer to a routine called after finishing a time step, may be NULL.
`close_timestep(mesh, adapt)` is called after accepting the solution(s) of the discrete problem on `mesh` at time `adapt->time` by the time–space adaptive method; can be used for visualization and export to file for post–processing of the `mesh` and discrete solution(s).

strategy parameter for the default ALBERTA`one_timestep` routine; possible values are:

- **0** explicit strategy,
- **1** implicit strategy.

max_iteration parameter for the default `one_timestep` routine; maximal number of time step size adaptation steps, only used by the implicit strategy.

tolerance given total error tolerance tol.

rel_initial_error portion Γ_0 of tolerance allowed for initial error, compare Section 1.5.4;

rel_space_error portion Γ_h of tolerance allowed for error from spatial discretization in each time step, compare Section 1.5.4.

rel_time_error portion Γ_τ of tolerance allowed for error from time discretization in each time step, compare Section 1.5.4.

time_theta_1 safety parameter θ_1 for the time adaptive method in the default ALBERTA`one_timestep()` routine; the tolerance for the time estimate η_τ is $\theta_1 \, \Gamma_\tau \, tol$, compare Algorithm 1.5.8.

time_theta_2 safety parameter θ_2 for the time adaptive method in the default ALBERTAone_timestep() routine; enlargement of the time step size is only allowed for $\eta_\tau \leq \theta_2 \, \Gamma_\tau \, tol$, compare Algorithm 1.5.8.

time_delta_1 factor δ_1 used for the reduction of the time step size in the default ALBERTAone_timestep() routine, compare Algorithm 1.5.8.

time_delta_2 factor δ_2 used for the enlargement of the time step size in the default ALBERTAone_timestep() routine, compare Algorithm 1.5.8.

info level of information produced by the time–space adaptive procedure.

Using information given in the ADAPT_INSTAT data structure, the space and time adaptive procedure is performed by:

```
void adapt_method_instat(MESH *, ADAPT_INSTAT *);
```

Description:

adapt_method_instat(mesh, adapt_instat) solves an instationary problem on mesh by the space–time adaptive procedure described in Section 1.5.4; adapt_instat is a pointer to a filled ADAPT_INSTAT data structure, holding all information about the problem to be solved and parameters for the adaptive method.

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)
{
/*--------------------------------------------------------------------------*/
/*  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->adapt_space->tolerance
     = adapt->tolerance * adapt->rel_space_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);
  }
}
```

4.8.3.1 The default ALBERTA`one_timestep()` routine

The default `one_timestep()` routine provided by ALBERTA implements both the explicit strategy 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.

4.8.4 Initialization of data structures for adaptive methods

ALBERTA provides functions for the initialization of the data structures `ADAPT_STAT` and `ADAPT_INSTAT`. Both functions do *not* fill any function pointer entry in the structures! These function pointers have to be adjusted in the application.

```
ADAPT_STAT *get_adapt_stat(const int, const char *, const char *,
                           int, ADAPT_STAT *);

ADAPT_INSTAT *get_adapt_instat(const int, const char *, const char *,
                               int, ADAPT_INSTAT *);
```

Description:

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	d	prefix->refine_bisections
coarsen_allowed	0	prefix->coarsen_allowed
coarse_bisections	d	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 4.3: Initialized members of an ADAPT_STAT structure, the default values and the key for the initialization by GET_PARAMETER().

get_adapt_stat(dim, name, prefix, info, adapt) returns a pointer to a partly initialized ADAPT_STAT structure; if the argument adapt is NULL, a new structure is created, the name name is duplicated at the name entry of the structure, if name is not NULL; if name is NULL, and prefix is not NULL, this string is duplicated at the name entry; dim is the mesh dimension d; for a newly created structure, all function pointers of the structure are initialized with NULL; all other members are initialized with some default value; if the argument adapt is not NULL, this initialization part is skipped, the name and function pointers are not changed;

if prefix is not NULL, 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 using the function 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 == 4 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 4.3 shows the initialized members, the default values and the key used for the initialization by GET_PARAMETER();

get_adapt_instat(dim, name, prefix, info, adapt) returns a pointer to a partly initialized ADAPT_INSTAT structure; if the argument adapt is NULL, a new structure is created, the name name is duplicated at the name entry of the structure, if name is not

member	default	parameter key
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 4.4: Initialization of the main parameters in an ADAPT_INSTAT structure for the time-adaptive strategy; initialized members, the default values and keys used for the initialization by GET_PARAMETER().

NULL; if name is NULL, and prefix is not NULL, this string is duplicated at the name entry; dim is the mesh dimension d; for a newly created structure, all function pointers of the structure are initialized with NULL; all other members are initialized with some default value; if the argument adapt is not NULL, this default initialization part is skipped;

if prefix is not NULL, 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;

Tables 4.4–4.6 shows the initialized members, the default values and the key used for the initialization by GET_PARAMETER(). The tolerances in the sub-structures adapt_initial and adapt_space are set to the values adapt->tolerance*adapt->rel_initial_error and adapt->tolerance*adapt->rel_space_error, respectively. A special initialization is done for the info parameters: when adapt_initial->info or adapt_space->info are negative, then they are set to adapt->info-2.

4.9 Implementation of error estimators

4.9.1 Error estimator for elliptic problems

ALBERTA provides a residual type error estimator for non–linear elliptic problems of the type

$$\begin{aligned} -\nabla \cdot A\nabla u(x) + f\Big(x, u(x), \nabla u(x)\Big) &= 0 && x \in \Omega, \\ u(x) &= g_d && x \in \Gamma_D, \\ \nu \cdot A\nabla u(x) &= g_n && x \in \Gamma_N, \end{aligned}$$

member	default	parameter key
adapt_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	d	prefix->initial->refine_bisections
adapt_initial->coarsen_allowed	0	prefix->initial->coarsen_allowed
adapt_initial->coarse_bisections	d	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

Table 4.5: Initialization of the adapt_initial sub-structure of an ADAPT_INSTAT structure for the adaptation of the initial grid; initialized members, the default values and keys used for the initialization by GET_PARAMETER().

member	default	parameter key
adapt_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	d	prefix->space->refine_bisections
adapt_space->coarsen_allowed	1	prefix->space->coarsen_allowed
adapt_space->coarse_bisections	d	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

Table 4.6: Initialization of the adapt_space sub-structure of an ADAPT_INSTAT structure for the adaptation of the grids during time-stepping; initialized members, the default values and keys used for the initialization by GET_PARAMETER().

where $A \in \mathbb{R}^{n\times n}$ is a positive definite matrix and $\partial\Omega = \Gamma_D \cup \Gamma_N$. ALBERTA implements for this kind of equations the L^2 and H^1 per-element estimators $\eta_{S,0}$ and $\eta_{S,1}$ ($S \in \mathcal{S}$)

$$\begin{aligned}\eta_{S,0}^2 &:= C_0^2\, h_S^4 \,\| -\nabla\cdot A\nabla u_h + f(.,u_h,\nabla u_h)\|_{L^2(S)}^2 \\ &\quad + C_1^2 \sum_{\Gamma\subset\partial S\cap\Omega} h_S^3 \,\|\, [\![A\nabla u_h]\!]\,\|_{L^2(\Gamma)}^2 + C_1^2 \sum_{\Gamma\subset\partial S\cap\Gamma_N} h_S^3 \,\|\nu\cdot A\nabla u_h - g_n\|_{L^2(\Gamma)}^2, \\ \eta_{S,1}^2 &:= C_0^2\, h_S^2 \,\| -\nabla\cdot A\nabla u_h + f(.,u_h,\nabla u_h)\|_{L^2(S)}^2 \\ &\quad + C_1^2 \sum_{\Gamma\subset\partial S\cap\Omega} h_S \,\|\, [\![A\nabla u_h]\!]\,\|_{L^2(\Gamma)}^2 + C_1^2 \sum_{\Gamma\subset\partial S\cap\Gamma_N} h_S \,\|\nu\cdot A\nabla u_h - g_n\|_{L^2(\Gamma)}^2,\end{aligned}$$

where $[\![.]\!]$ denotes the jump of the normal component across an interior co-dimension 1 sub-simplex (vertex/edge/face) $\Gamma \subset \partial S$.

Verfürth proved for $g_d \equiv 0$ and $g_n \equiv 0$ in [27] – under suitable assumptions on f, u and u_h in the non-linear case – the estimate

$$\|u - u_h\|^2_{H^1(\Omega)} \leq \sum_{S \in \mathcal{S}} \eta^2_{S,1},$$

and Bänsch and Siebert [2] proved a similar the L^2-estimate for the semi–linear case $f = f(x,u)$ and $g_d \equiv 0$ and $\Gamma_N = \emptyset$:

$$\|u - u_h\|^2_{L^2(\Omega)} \leq \sum_{S \in \mathcal{S}} \eta^2_{S,0}.$$

The following functions implement above estimators for scalar and vector-valued functions; the implementation works also for meshes with non-zero co-dimension as well as for periodic meshes.

```
REAL ellipt_est(const DOF_REAL_VEC *uh, ADAPT_STAT *adapt,
                REAL *(*rw_est)(EL *), REAL *(*rw_estc)(EL *),
                int quad_deg,
                NORM norm, REAL C[3], const REAL_DD A,
                const BNDRY_FLAGS dirichlet_bndry,
                REAL (*f)(const EL_INFO *el_info,
                          const QUAD *quad, int qp,
                          REAL uh_qp, const REAL_D grd_uh_gp),
                FLAGS f_flags,
                REAL (*gn)(const EL_INFO *el_info,
                           const QUAD *quad, int qp,
                           REAL uh_qp, const REAL_D normal),
                FLAGS gn_flags);

REAL ellipt_est_dow(const DOF_REAL_VEC_D *uh, ADAPT_STAT *adapt,
                    REAL *(*rw_est)(EL *), REAL *(*rw_estc)(EL *),
                    int quad_deg,
                    NORM norm, REAL C[3],
                    const void *A, MATENT_TYPE A_type, MATENT_TYPE A_blocktype,
                    bool sym_grad,
                    const BNDRY_FLAGS dirichlet_bndry,
                    const REAL *(*f)(REAL_D result,
                                     const EL_INFO *el_info,
                                     const QUAD *quad, int qp,
                                     const REAL_D uh_qp,
                                     const REAL_DD grd_uh_gp),
                    FLAGS f_flags,
                    const REAL *(*gn)(REAL_D result,
                                      const EL_INFO *el_info,
                                      const QUAD *quad, int qp,
                                      const REAL_D uh_qp,
                                      const REAL_D normal),
                    FLAGS gn_flags);

REAL ellipt_est_d(const DOF_REAL_D_VEC *uh, ADAPT_STAT *adapt,
```

```
REAL *(*rw_est)(EL *), REAL *(*rw_estc)(EL *),
int quad_deg,
NORM norm, REAL C[3],
const void *A, MATENT_TYPE A_type, MATENT_TYPE A_blocktype,
bool sym_grad,
const BNDRY_FLAGS dirichlet_bndry,
const REAL *(*f)(REAL_D result,
                 const EL_INFO *el_info,
                 const QUAD *quad, int qp,
                 const REAL_D uh_qp,
                 const REAL_DD grd_uh_gp),
FLAGS f_flags,
const REAL *(*gn)(REAL_D result,
                  const EL_INFO *el_info,
                  const QUAD *quad, int qp,
                  const REAL_D uh_qp,
                  const REAL_D normal),
FLAGS gn_flags);
```

Description:

`ellipt_est(uh, adapt, rw_est, rw_estc, quad_deg, norm, C, A, dirichlet_bndry, f, f_flags, gn, gn_flags)`

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 `NULL` pointer, nothing is done, the return value is `.0`.

`adapt` is a pointer to an `ADAPT_STAT` structure; if not `NULL`, the entries `adapt->p=2`, `err_sum`, and `err_max` of `adapt` are set by `ellipt_est()` (compare Section 4.8.1).

`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 3.2.10); if this function is `NULL`, 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 4.8.1).

`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 3.2.10); if this function is `NULL`, 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.

`quad_deg` 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 `alberta.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, wall residual, and coarsening term respectively. If `C` is `NULL`, then all constants are set to 1.0.

`A` is the constant matrix of the second order term.

dirichlet_bndry A bit-mask marking those parts of the boundary which are subject to Dirichlet boundary conditions, see Section 3.2.4.

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 `NULL` pointer, $f \equiv 0$ is assumed;

`f(el_info, quad, qp, uh_qp, grd_uh_qp)` returns the value of the lower oder terms on element `el_info->el` at the quadrature node `quad->lambda[qp]`, where `uh_qp` is the value and `grd_uh_qp` the gradient (with respect to the Cartesian coordinates) of the discrete solution at that quadrature point. See also `f_flag` below:

f_flag specifies whether the function `f()` actually needs values of `uh_qp` or `grd_uh_qp`, `f_flag` may be 0 or `INIT_UH` or `INIT_GRD_UH` or their bitwise composition (`|`). The arguments `uh_qp` and `grd_uh_qp` of `f()` only hold valid information if the flags `INIT_UH` respectively `INIT_GRD_UH` are set.

gn(el_info, quad, qp, uh_qp, normal) is a pointer to a function for the evaluation of non-homogeneous Neumann boundary data. `gn` may be `NULL`, in which case zero Neumann boundary conditions are assumed. The argument `normal` always contains the normal of the Neumann boundary facet. In the case of non-vanishing co-dimension `normal` lies in the lower-dimensional space which is spanned by the mesh simplex defined by `el_info`. `gn()` is evaluated on those parts of the boundary which are *not* flagged as Dirichlet-boundaries by the argument `dirichlet_bndry`.

gn_flag controls whether the argument `uh_qp` of the function `gn()` actually contains the value of `uh` at the quadrature point `qp`. Note that the argument `normal` always contains valid data.

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 `NULL`).

```
ellipt_est_d(uh, adapt, rw_est, rw_estc, quad_deg, norm, C,
             A, A_type, A_blocktype, sym_grad,
             dirichlet_bndry, f, f_flags, gn, gn_flags)
ellipt_est_dow(uh, adapt, rw_est, rw_estc, quad_deg, norm, C,
               A, A_type, A_blocktype, sym_grad,
               dirichlet_bndry, f, f_flags, gn, gn_flags)
```

Similar function for a (coupled) vector valued elliptic problem. We document only the arguments which are different from the arguments of `ellipt_est()`:

A now represents a tensor ($A_{ij}^{\mu\nu} \in \mathbb{R}^{n\times n, n\times n}$, $i, j, \mu, \nu = 0, \ldots, n-1$. The indexing is

$$\texttt{A[i][j][mu][nu]} = A_{\texttt{ij}}^{\texttt{mu,nu}},$$

with `i,j,mu,nu==0,...,DIM_OF_WORLD-1`, see Section 1.4.6. `A` describes the coefficients of the principal part of a coupled system of elliptical equations:

$$-\sum_{\nu,i,j=0}^{n-1} \partial_i A_{ij}^{\mu\nu} \partial_j u^\nu + \text{lower order terms} = f^\mu \quad (\mu = 0, \ldots, n-1).$$

The `quasi-stokes.c` demo-program contains an example.

A_blocktype must be one of MATENT_REAL, MATENT_REAL_D or MATENT_REAL_DD. It specifies the symmetry type for coupling of the PDE system. Note that the storage layout of A is determined by the argument A_blocktype:

```
MATENT_REAL:     REAL A[DIM_OF_WORLD][DIM_OF_WORLD];
MATENT_REAL_D:   REAL_D A[DIM_OF_WORLD][DIM_OF_WORLD];
MATENT_REAL_DD:  REAL_DD A[DIM_OF_WORLD][DIM_OF_WORLD];
```

A_blocktype == MATENT_REAL or A_blocktype == MATENT_REAL_D means that the system is actually decoupled.

A_type must be one of MATENT_REAL, MATENT_REAL_D or MATENT_REAL_DD. It specifies the symmetry type of A with respect to the first two indices. For a Laplacian, for example, one would use DOWBM_SCAL. Note that the value of A_type does *not* change the storage layout of the array A.

sym_grad If set to true then it is assumed that the symmetric gradient has to be used for the computation of the jump- and Neumann-residuals. The demo-program quasi-stokes.c uses this feature to implement an error estimator for the Stokes equation with stress boundary conditions.

f If the first argument of the function pointer f(result,...) is not NULL then the result *must* be stored in the argument result and f() must return the base address of the array result. If result is NULL, then f() must store the result in a non-volatile storage area and return the address of that area.

dirichlet_bndry A bit-mask marking those parts of the boundary which are subject to Dirichlet boundary conditions, see Section 3.2.4.

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 NULL pointer, $f \equiv 0$ is assumed;

f(el_info, quad, qp, uh_qp, grd_uh_qp) returns the value of the lower oder terms on element el_info->el at the quadrature node quad->lambda[qp], where uh_qp is the value and grd_uh_qp the gradient (with respect to the Cartesian coordinates) of the discrete solution at that quadrature point. See also f_flag below:

f_flag specifies whether the function f() actually needs values of uh_qp or grd_uh_qp, f_flag may be 0 or INIT_UH or INIT_GRD_UH or their bitwise composition (|). The arguments uh_qp and grd_uh_qp of f() only hold valid information if the flags INIT_UH respectively INIT_GRD_UH are set.

gn(el_info, quad, qp, uh_qp, normal) is a pointer to a function for the evaluation of non-homogeneous Neumann boundary data. gn may be NULL, in which case zero Neumann boundary conditions are assumed. The argument normal always contains the normal of the Neumann boundary facet. In the case of non-vanishing co-dimension normal lies in the lower-dimensional space which is spanned by the mesh simplex defined by el_info. gn() is evaluated on those parts of the boundary which are *not* flagged as Dirichlet-boundaries by the argument dirichlet_bndry.

gn_flag controls whether the argument uh_qp of the function gn() actually contains the value of uh at the quadrature point qp. Note that the argument normal always contains valid data.

4.9.1 Example (Linear problem)**.** Consider the scalar linear model problem (1.7) with constant coefficients A, b, and c:

$$\begin{aligned} -\nabla \cdot A\nabla u + b \cdot \nabla u + c\,u &= r \qquad \text{in } \Omega, \\ u &= 0 \qquad \text{on } \partial\Omega. \end{aligned}$$

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 (1.7) 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`.

4.9.2 Error estimator for parabolic problems

Similar to the stationary case, the ALBERTA library provides an error estimator for the non–linear parabolic problem

$$\begin{aligned} \partial_t u - \nabla \cdot A\nabla u(x) + f\Big(x, t, u(x), \nabla u(x)\Big) &= 0 && x \in \Omega, t > 0, \\ u(x,t) &= g_d && x \in \Gamma_D, t > 0, \\ \nu \cdot A\nabla u(x,t) &= g_n && x \in \Gamma_N, t > 0, \\ u(x,0) &= u_0 && x \in \Omega, \end{aligned}$$

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 4.6).

For the estimation of the spatial discretization error, the coarsening error, and the time

discretization error, the ALBERTA estimator implements the following (local) indicators

$$\eta_S^2 = C_0^2\, h_S^4 \left\| \frac{U_{n+1} - I_{n+1}U_n}{\tau_{n+1}} - \nabla \cdot A\nabla U_{n+1} + f(.,t_{n+1},U_{n+1},\nabla U_{n+1}) \right\|^2_{L^2(S)}$$
$$+ C_1^2 \sum_{\Gamma\subset\partial S\cap\Omega} h_S^3 \,\|\, [\![A\nabla U_{n+1}]\!] \,\|^2_{L^2(\Gamma)} + C_1^2 \sum_{\Gamma\subset\partial S\cap\Gamma_N} h_S^3 \,\|\nu\cdot A\nabla U_{n+1} - g_n\|^2_{L^2(\Gamma)},$$

$$\eta_{S,c}^2 = C_2^2\, h_S^3 \,\|\, [\![\nabla U_n]\!] \,\|^2_{L^2(\Gamma_c)}$$

$$\eta_\tau = C_3 \|U_{n+1} - I_{n+1}U_n\|_{L^2(\Omega)}.$$

The coarsening indicator is motivated by the fact that for piecewise linear Lagrange finite element functions it holds $\|U_n - I_{n+1}U_n\|^2_{L^2(S)} = \eta^2_{S,c}$ with $C_2 = C_2(d)$ and Γ_c the face that would be removed during a coarsening operation. The implementation is done by the functions

```
REAL heat_est(const DOF_REAL_VEC *uh, ADAPT_INSTAT *adapt,
              REAL *(*rw_est)(EL *), REAL *(*rw_estc)(EL *),
              int quad_degree, REAL C[4], const DOF_REAL_VEC *uh_old,
              const REAL_DD A, const BNDRY_FLAGS dirichlet_bndry,
              REAL (*f)(const EL_INFO *el_info, const QUAD *quad, int qp,
                        REAL uh_qp, const REAL_D grd_uh_gp, REAL time),
              FLAGS f_flags,
              REAL (*gn)(const EL_INFO *el_info, const QUAD *quad, int qp,
                         REAL uh_qp, const REAL_D normal, REAL time),
              FLAGS gn_flags);
REAL heat_est_dow(const DOF_REAL_D_VEC *uh, ADAPT_INSTAT *adapt,
                  REAL *(*rw_est)(EL *), REAL *(*rw_estc)(EL *),
                  int quad_degree, REAL C[4], const DOF_REAL_D_VEC *uh_old,
                  const void *A, MATENT_TYPE A_type, MATENT_TYPE A_blocktype,
                  bool sym_grad,
                  BNDRY_FLAGS dirichlet_bndry,
                  const REAL *(*f)(REAL_D result,
                                   const EL_INFO *el_info,
                                   const QUAD *quad, int qp,
                                   const REAL_D uh_qp,
                                   const REAL_DD grd_uh_gp,
                                   REAL time),
                  FLAGS f_flags,
                  const REAL *(*gn)(REAL_D result,
                                    const EL_INFO *el_info,
                                    const QUAD *quad, int qp,
                                    const REAL_D uh_qp,
                                    const REAL_D normal,
                                    REAL time),
                  FLAGS gn_flags);
REAL heat_est_d(const DOF_REAL_D_VEC *uh,
const DOF_REAL_D_VEC *uh_old,
ADAPT_INSTAT *adapt,
REAL *(*rw_est)(EL *),
REAL *(*rw_estc)(EL *),
int quad_degree,
```

```
  REAL C[4],
  const void *A,
  MATENT_TYPE A_type,
  MATENT_TYPE A_blocktype,
  bool sym_grad,
  const BNDRY_FLAGS dirichlet_bndry,
  const REAL *(*f)(REAL_D result,
   const EL_INFO *el_info,
   const QUAD *quad,
   int qp,
   const REAL_D uh_qp,
   const REAL_DD grd_uh_gp,
   REAL time),
  FLAGS f_flags,
  const REAL *(*gn)(REAL_D result,
    const EL_INFO *el_info,
    const QUAD *quad,
    int qp,
    const REAL_D uh_qp,
    const REAL_D normal,
    REAL time),
  FLAGS gn_flags);
```

Description:

heat_est(uh, adapt, rw_el_est, rw_el_estc, degree, C, uh_old, A, dirichlet_bndry, f, f_flag, gn, gn_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 `NULL` pointer, nothing is done, the return value is `0.0`.

adapt is a pointer to an `ADAPT_INSTAT` structure; if it is not `NULL`, then the entries `adapt_space->p=2`, `adapt_space->err_sum` and `adapt_space->err_max` of `adapt` are set by `heat_est()` (compare Section 4.8.1).

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 3.2.10); if this function is `NULL`, 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 4.8.1).

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 3.2.10); if this function is `NULL`, 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. The coarsening indicator is not used at the moment.

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, wall residual, coarsening term, and time residual, respectively. If C is NULL, 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 NULL pointer, nothing is done, the return value is 0.0.

A is the constant matrix of the second order term.

dirichlet_bndry A bit mask marking those parts of the boundary which are subject to Dirichlet boundary conditions. See Section 3.2.4.

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 NULL 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.

gn(el_info, quad, qp, uh_qp, normal) is a pointer to a function for the evaluation of non-homogeneous Neumann boundary data. gn may be NULL, in which case zero Neumann boundary conditions are assumed. The argument normal always contains the normal of the Neumann boundary facet. In the case of non-vanishing co-dimension normal lies in the lower-dimensional space which is spanned by the mesh simplex defined by el_info.

gn_flag controls whether the argument uh_qp of the function gn() actually contains the value of uh at the quadrature point qp. Note that the argument normal always contains valid data.

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 NULL).

```
heat_est_d(uh, adapt, rw, rwc, deg, C, uh_old,
           A, A_type, A_blocktype, sym_grad,
           dirichlet_bndry, f, f_flag)
heat_est_dow(uh, adapt, rw, rwc, deg, C, uh_old,
             A, A_type, A_blocktype, sym_grad,
             dirichlet_bndry, f, f_flag)
```

Coupled vector valued version. See ellipt_est_dow() above.

There are also some less high-level support functions which allow for custom contributions to the per-element error estimates. We will not document this in detail, but rather refer the reader to the stokes.c and quasi-stokes.c demo-programs.

```
const void *ellipt_est_init(const DOF_REAL_VEC *uh,
                            ADAPT_STAT *adapt,
                            REAL *(*rw_est)(EL *),
                            REAL *(*rw_estc)(EL *),
                            const QUAD *quad,
                            const WALL_QUAD *wall_quad,
                            NORM norm,
                            REAL C[3],
                            const REAL_DD A,
                            const BNDRY_FLAGS dirichlet_bndry,
                            REAL (*f)(const EL_INFO *el_info,
                                      const QUAD *quad,
                                      int qp,
                                      REAL uh_qp,
                                      const REAL_D grd_uh_gp),
                            FLAGS f_flags,
                            REAL (*gn)(const EL_INFO *el_info,
                                       const QUAD *quad,
                                       int qp,
                                       REAL uh_qp,
                                       const REAL_D normal),
                            FLAGS gn_flags);
const void *heat_est_init(const DOF_REAL_VEC *uh,
                          const DOF_REAL_VEC *uh_old,
                          ADAPT_INSTAT *adapt,
                          REAL *(*rw_est)(EL *),
                          REAL *(*rw_estc)(EL *),
                          const QUAD *quad,
                          const WALL_QUAD *wall_quad,
                          REAL C[4],
                          const REAL_DD A,
                          const BNDRY_FLAGS dirichlet_bndry,
                          REAL (*f)(const EL_INFO *el_info,
                                    const QUAD *quad,
                                    int qp,
                                    REAL uh_qp,
                                    const REAL_D grd_uh_gp,
                                    REAL time),
                          FLAGS f_flags,
                          REAL (*gn)(const EL_INFO *el_info,
                                     const QUAD *quad,
                                     int qp,
                                     REAL uh_qp,
                                     const REAL_D normal,
                                     REAL time),
                          FLAGS gn_flags);

REAL element_est(const EL_INFO *el_info, const void *est_handle);
void element_est_finish(const EL_INFO *el_info,
                        REAL est_el, const void *est_handle);
const REAL *element_est_uh(const void *est_handle);
const REAL_D *element_est_grd_uh(const void *est_handle);
REAL ellipt_est_finish(ADAPT_STAT *adapt, const void *est_handle);
REAL heat_est_finish(ADAPT_INSTAT *adapt, const void *est_handle);
```

There are similar proto-types for the vector-valued case. Now, what are these functions good for? The `stokes.c` program makes use of this framework to add a contribution concern-

ing the divergence constraint. Of course, this is an ad-hoc error indicator, and only meant to demonstrate the programming frame-work. The functions `element_est_uh[_dow]()` and `element_est_grd_uh[_dow]()` give the application access to the values of the discrete solution at the quadrature points (respectively to its Jaocbians). Otherwise, the general layout is like follows:

```
void *est_handle = ellipt_est_init(...);
TRAVERSE_FIRST(mesh, -1, <suitable fill-flags>) {
  REAL est_el = element_est(el_info, est_handle);

  ... /* add whatever you like to est_el */

  element_est_finish(el_info, est_el, est_handle);
} TRAVERSE_NEXT();
REAL est = ellipt_est_finish(adapt, est_handle);
```

The relevant excerpt from `stokes.c` reads as follows:

```
  est_handle = ellipt_est_dow_init(u_h, adapt, rw_el_est, NULL /* rw_estc */,
                                   quad, NULL /* wall_quad */,
                                   H1_NORM, C,
                                   A, MATENT_REAL, MATENT_REAL,
                                   false /* !sym_grad */,
                                   dirichlet_mask,
                                   r, INIT_GRD_UH,
                                   NULL /* inhomog. Neumann res. */, 0);

  fill_flags =
     FILL_NEIGH|FILL_COORDS|FILL_OPP_COORDS|FILL_BOUND|CALL_LEAF_EL;
  fill_flags |= u_fe_space->bas_fcts->fill_flags;
  fill_flags |= p_fe_space->bas_fcts->fill_flags;
  TRAVERSE_FIRST(mesh, -1, fill_flags) {
    const EL_GEOM_CACHE *elgc;
    const QUAD_EL_CACHE *qelc;
    REAL est_el;

    est_el = element_est_dow(el_info, est_handle);

    if (C[3]) {
      REAL div_uh_el, div_uh_qp;
      const REAL_DD *grd_uh_qp;
      int qp, i;

      grd_uh_qp = element_est_grd_uh_d(est_handle);
      div_uh_el = 0.0;
      if (!(el_info->fill_flag & FILL_COORDS)) {
        qelc = fill_quad_el_cache(el_info, quad, FILL_EL_QUAD_DET);

        for (qp = 0; qp < quad->n_points; qp++) {
          div_uh_qp = 0;
          for (i = 0; i < DIM_OF_WORLD; i++) {
            div_uh_qp += grd_uh_qp[qp][i][i];
          }
          div_uh_el += qelc->param.det[qp]*quad->w[qp]*SQR(div_uh_qp);
        }
      } else {
        elgc = fill_el_geom_cache(el_info, FILL_EL_DET);
```

```
      for (qp = 0; qp < quad->n_points; qp++) {
        div_uh_qp = 0;
        for (i = 0; i < DIM_OF_WORLD; i++) {
          div_uh_qp += grd_uh_qp[qp][i][i];
        }
        div_uh_el += quad->w[qp]*SQR(div_uh_qp);
      }
      div_uh_el *= elgc->det;
    }

    est_el += C[3] * div_uh_el;
  }

  element_est_dow_finish(el_info, est_el, est_handle);
} TRAVERSE_NEXT();
est = ellipt_est_dow_finish(adapt, est_handle);
```

4.10 Solver for linear and nonlinear systems

ALBERTA provides own solvers for general linear and nonlinear systems. The solvers use dense `REAL`-vectors for storing coefficients. They are aware of ALBERTA's DOF-vector and -matrix data structures and work with an application provided subroutine for the matrix-vector multiplication, and in case 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 (Orthogonal Error Methods) module, a built-in ALBERTA interface for solving systems involving a `DOF_MATRIX` and `DOF_REAL[_D]_VEC[_D]` objects, and the access to 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. Most of the implemented methods (and more) are described for example in [17, 23].

4.10.1 Krylov-space solvers for general linear systems

Very efficient solvers for linear systems are Krylov-space solvers (or Orthogonal Error Methods). 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 library solvers work on dense flat vectors and do not need to know the storage of the system matrix, or the matrix used for preconditioning. Matrix-vector multiplication and preconditioning is done by application provided routines.

Most of the implemented `OEM` solvers are a `C`-translation from the solvers of the FORTRAN OFM library (Orthogonale Fehler Methoden), by Dörfler [7]. `SymmLQ` is the algorithm described in [20], and `TfQMR` is described in TO BE DETERMINED. All solvers allow for *left* preconditioning and some also for *right* preconditioning.

The data structure (defined in `alberta_util.h`) for passing information about matrix-vector multiplication, preconditioning and tolerances, etc. to the solvers is

```
typedef int (*OEM_MV_FCT)(void *data, int dim, const REAL *rhs, REAL *u);

typedef struct oem_data OEM_DATA;
struct oem_data
{
  OEM_MV_FCT mat_vec;
  void       *mat_vec_data;
  OEM_MV_FCT mat_vec_T;
  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;

  REAL       (*scp)(void *, int, const REAL *, const REAL *);
  void       *scp_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 the system matrix 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 application.

mat_vec_data pointer to application 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 the transposed system matrix 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 application data.

mat_vec_T_data pointer to application data for the matrix-vector multiplication with the transposed system matrix, first argument to `mat_vec_T()`.

left_precon pointer to function for left preconditioning; it may be a `NULL` 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 application data.

left_precon_data pointer to application data for the left preconditioning, first argument to `left_precon()`.

Method	Matrix	Operations	Storage
BiCGstab	symmetric	2 MV + 12 V	$5N$
CG	symmetric positive definite	1 MV + 5 V	$3N$
GMRES	regular	k MV + ...	$(k+2)N + k(k+4)$
ODir	symmetric positive	1 MV + 11 V	$5N$
ORes	symmetric	1 MV + 12 V	$7N$
SymmLQ	symmetric		$6N$
TfQMR	regular		$11N$

Table 4.7: OEM methods with applicable matrix types, numbers of operations per iteration (MV matrix-vector products, V vector operations), and storage requirements (N number of unknowns, k GMRES subspace dimension)

`right_precon` pointer to function for right preconditioning; it may be a `NULL` 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 application data.

`right_precon_data` pointer to application data for the right preconditioning, first argument to `right_precon()`.

`scp` pointer to a function for computing a problem dependent scalar product; it may be a `NULL` pointer; in this case the Euclidian scalar product is used;

`scp(scp_data, dim, x, y)` computes the problem dependent scalar product of the two vectors `x` and `y` of length `dim`; `scp_data` is a pointer to application data.

`scp_data` pointer to application data for computing the scalar product, first argument to `scp()`.

`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 `NULL`, then the used solver allocates memory, which is freed before exit.

`tolerance` tolerance for the residual; if the norm of the residual is less than or equal to `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 final residual on exit.

The following linear solvers are currently implemented. Table 4.7 gives an overview over the implemented solvers, the matrix types they apply to, and the cost of one iteration.

```
int oem_bicgstab(OEM_DATA *oem_data, int dim, const REAL *rhs, REAL *u0);
int oem_cg(OEM_DATA *oem_data, int dim, const REAL *rhs, REAL *u0);
int oem_gmres(OEM_DATA *oem_data, int dim, const REAL *rhs, REAL *u0);
int oem_gmres_k(OEM_DATA *oem_data, int dim, const REAL *rhs, REAL *u0);
int oem_odir(OEM_DATA *oem_data, int dim, const REAL *rhs, REAL *u0);
int oem_ores(OEM_DATA *oem_data, int dim, const REAL *rhs, REAL *u0);
int oem_tfqmr(OEM_DATA *oem_data, int dim, const REAL *rhs, REAL *u0);
int oem_symmlq(OEM_DATA *oem_data, int dim, const REAL *rhs, REAL *u0);
```

Description:

`oem_bicgstab(oem_data, dim, rhs, u0)` 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, `rhs` the right hand side vector, and `u0` 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, rhs, u0)` 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, `rhs` the right hand side vector, and `u0` 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, rhs, u0)` 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, `rhs` the right hand side vector, and `u0` 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`, otherwise `restart=10` will be used; `oem_gmres()` needs a workspace for storing `(oem_data->restart+2)*dim + oem_data->restart*(oem_data->restart+4)` additional `REAL`s.

`oem_gmres_k(oem_data, dim, rhs, u0)` performs just one restart step (minimization on a k-dimensional Krylov subspace) of the GMRES method. This routine can be used as subroutine in other solvers. For example, `oem_gmres()` just iterates this until the tolerance is met. Other applications are nonlinear GMRES solvers, where a new linearization is done after each linear GMRES restart step.

`oem_odir(oem_data, dim, rhs, u0)` 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, `rhs` the right hand side vector, and `u0` 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, rhs, u0)` 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, `rhs` the right hand side vector, and `u0` 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.

`oem_symmlq(oem_data, dim, rhs, u0)` solves a symmetric linear system. `oem_data` stores information about matrix vector multiplication, preconditioning, tolerances, etc. `dim` is the dimension of the linear system, `rhs` the right hand side vector, and `u0` the initial guess on input and the solution on output; `oem_symmlq()` needs a workspace for storing `6*dim` additional `REAL`s; the return value is the number of iterations. `oem_symmlq()` supports uses left preconditioning.

`oem_tfqmr(oem_data, dim, rhs, u0)` solves a linear system using a transpose-free QMR method 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, `rhs` the right hand side vector, and `u0` the initial guess on input and the solution on output; `oem_tfqmr()` needs a workspace for storing `11*dim` additional `REAL`s; the return value is the number of iterations.

4.10.2 Krylov-space solvers for DOF matrices and vectors

4.10.1 Compatibility Note. *The support for additional preconditioners, as well as the block-matrix structure induced by the support for direct sums of finite element spaces (see Section 3.7) made it necessary to provide a more flexible and extendible interface to the implemented preconditioners. Additionally, some of the preconditioners need further parameters.*

Therefore, the selection of a particular preconditioner has been moved to separate functions `init_oem_precon()`, `vinit_oem_precon()` *and* `init_precon_from_type()`, *the latter requiring a special support structure* `PRECON_TYPE` *to pass parameters on to the preconditioners.*

Solver-functions, which previously accepted a mere integer to select a particular preconditioner, now need a pointer to a `PRECON`*-structure, see below Section 4.10.7.*

We describe here the interface between ALBERTA's DOF-vectors and -matrices and the available general OEM-solvers described in the previous Section 4.10.1. At the highest level, there are three function, namely `oem_solve_s()`, `oem_solve_d()` and `oem_solve_dow()`. The calling conventions for the three functions are functionally identical, except for the data-type of the DOF-vector arguments. 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 $x, b \in \mathbb{R}^N$. Vector valued problems need a closer examination, there are two cases:

1. `DIM_OF_WORLD`-valued finite element spaces based on scalar basis functions:

   `oem_solve_d()` and `oem_solve_dow()` can both either be used for decoupled or coupled `DIM_OF_WORLD`-valued problems. Decoupled problems are of the form

$$\begin{bmatrix} A & 0 & \dots & 0 \\ 0 & A & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & A \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix}$$

with $A \in \mathbb{R}^{N\times N}$ and $u_i, f_i \in \mathbb{R}^N$, $i = 1,\dots,n$, where $n =$ DIM_OF_WORLD. The vectors $(u_1,\dots,u_n)$ and $(f_1,\dots,f_n)$ are stored in DOF_REAL_D_VECs, whereas the matrix is stored as a single scalar DOF_MATRIX.

Coupled DIM_OF_WORLD-valued problems lead in this context to matrices of the form

$$\begin{bmatrix} A^{00} & \dots & A^{0n} \\ \vdots & \ddots & \vdots \\ A^{n0} & \dots & A^{nn} \end{bmatrix} \begin{bmatrix} u_1 \\ \vdots \\ u_n \end{bmatrix} = \begin{bmatrix} f_1 \\ \vdots \\ f_n \end{bmatrix}$$

with $A^{\mu\nu} \in \mathbb{R}^{N\times N}$ and $u_\nu, f_\mu \in \mathbb{R}^N$, $\mu,\nu = 1,\dots,n$, where $n =$ DIM_OF_WORLD. The vectors $(u_1,\dots,u_n)$ and $(f_1,\dots,f_n)$ are again stored in DOF_REAL_D_VECs. One prominent example is the discretisation of a Stokes-problem with prescribed stresses on the boundary: in this case the weak formulation has to be based on the deformation tensor, which leads to matrix of above type. The matrix is still stored as a DOF_MATRIX structure, but its entries are DIM_OF_WORLD × DIM_OF_WORLD blocks: the data is stored as an $N \times N$ matrix of small $d \times d$ blocks in analogy to DOF_REAL_D_VECs. See also 1.4.6. Compare also Compatibility Note 4.7.1.

2. Finite element spaces based on DIM_OF_WORLD-valued basis functions:

   In this case the DOF-vectors are scalar-valued, and the resulting DOF-matrix is just a scalar matrix, compare also Section 1.4.6.

Note that the interface routines to the OEM-solvers are aware of direct sums of finite element spaces, as described in Section 3.7, the resulting block-matrices generated by the assemble-framework will be handled correctly, including the cases where a standard Lagrangian finite element space is augmented by vector-valued basis functions like face-bubbles.

An application selects a particular solver by passing one of the following enumeration values to oem_solve_[s|d|dow]():

```
typedef enum {
  NoSolver, BiCGStab, CG, GMRes, ODir, ORes, TfQMR, GMRes_k, SymmLQ
} OEM_SOLVER;
```

New identifiers may be added to this enumeration when new solvers are added to ALBERTA. In more detail, the three high-level interface function are described below:

4.10.2 Function (oem_solve_[s|d|dow]()).

Prototypes

```
int oem_solve_s(const DOF_MATRIX *A, const DOF_SCHAR_VEC *bound,
                const DOF_REAL_VEC *f, DOF_REAL_VEC *u,
                OEM_SOLVER solver,
                REAL tol, const PRECON *precon,
                int restart, int max_iter, int info);
int oem_solve_d(const DOF_MATRIX *A, const DOF_SCHAR_VEC *bound,
                const DOF_REAL_D_VEC *f, DOF_REAL_D_VEC *u,
                OEM_SOLVER solver,
                REAL tol, const PRECON *precon,
                int restart, int max_iter, int info);
```

```
int oem_solve_dow(const DOF_MATRIX *A, const DOF_SCHAR_VEC *bound,
                  const DOF_REAL_VEC_D *f, DOF_REAL_VEC_D *u,
                  OEM_SOLVER solver,
                  REAL tol, const PRECON *precon,
                  int restart, int max_iter, int info);
```

Synopsis

```
iterations = oem_solve_[s|d|dow](A, mask, f, u,
                                 solver, tol, precon,
                                 restart, max_iter, info);
```

Description

Attempt to solve the linear system defined by the matrix `A`, an optional restriction to a sub-space by masking out DOFs via `mask`, a load-vector `f` and an initial guess and storage `u` for the approximative solution.

Parameters

`A` Pointer to a `DOF_MATRIX` storing the system matrix.

`mask` Pointer to a `DOF_SCHAR_VEC` masking out parts of the finite element space: if `mask->vec[d] >= DIRICHLET`, then `A` will act as if the d-th row would be zero. Compare also the discussion in the section about Dirichlet boundary condition, see Section 4.7.7.1

`f` A pointer to a `DOF_REAL[_D]_VEC[_D]` storing the load-vector of the linear system.

`u` A pointer to a `DOF_REAL[_D]_VEC[_D]` storing the initial guess on input and the approximative solution on output. In the context of interpolated Dirichlet boundary conditions special provisions have to be taken for the "Dirichlet-nodes". Compare the discussion in Section 4.7.7.1.

`solver` Use the respective OEM-solver; see above for the available keywords.

`tol` Tolerance for the residual; if the norm of the residual is less or equal `tol`, `oem_solve_[s|d|dow]()` returns the actual iterate as the approximative solution of the system.

`precon` A pointer to a structure describing the preconditioner to use, see further below in Section 4.10.7.

4.10.3 Compatibility Note. *Previous versions used a simple number here, but as the preconditioner frame-work has become much more complicated, because of the support for direct sums of finite element spaces, the code for the selection of the preconditioner has been separated from the entry-point to the solvers.*

`restart` Only used by `gmres`: the maximum dimension of the Krylov-space.

`max_iter` Maximal number of iterations to be performed by the linear solver. This can be compared with the return value – which gives the number of iterations actually performed – to determine whether the solver has achieved its goal.

info This 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.

Return Value

The number of iterations the solver needed until the norm of the residual was below tol, or max_iter if the solver was not able to reach its goal before the prescribed maximum iteration count was exhausted.

There is also an interface to the OEM-solvers which splits the call to the OEM-methods into an initialization part, an execution part and a cleanup part. This is useful when the same solver applies the same matrix to varying load-vectors. One example is the implementation of a CG-method for Schur's complement operator of a saddle-point problem (see Section 4.10.4 below). The following functions implement this interface:

```
typedef int (*OEM_MV_FCT)(void *data, int dim, const REAL *rhs, REAL *u);

OEM_MV_FCT get_oem_solver(OEM_SOLVER);
OEM_DATA *init_oem_solve(const DOF_MATRIX *A,
                         const DOF_SCHAR_VEC *mask,
                         REAL tol, const PRECON *precon,
                         int restart, int max_iter, int info);
void release_oem_solve(const OEM_DATA *oem);
int call_oem_solve_s(const OEM_DATA *oem, OEM_SOLVER solver,
                     const DOF_REAL_VEC *f, DOF_REAL_VEC *u);
int call_oem_solve_dow(const OEM_DATA *oem, OEM_SOLVER solver,
                       const DOF_REAL_VEC_D *f, DOF_REAL_VEC_D *u);
int call_oem_solve_d(const OEM_DATA *oem, OEM_SOLVER solver,
                     const DOF_REAL_D_VEC *f, DOF_REAL_D_VEC *u);
```

See Example 4.10.9-4.10.11 for short code skeletons explaining the use of these functions. The descriptions for the individual functions are as follows:

4.10.4 Function (get_oem_solver()).

Synopsis

```
solver_fct = get_oem_solver(solver_num);
```

Description

Return a function pointer for the solver indicated by solver_num which shuld be one of the symbols BiCGStab, CG GMRes, ODir, ORes, TfQMR, GMRes_k, SymmLQ.

Parameters

solver_num As explained above.

Return Value

A function pointer conforming to the type

```
int (*OEM_MV_FCT)(void *data, int dim, const REAL *rhs, REAL *u);
```

4.10.5 Function (`init_oem_solve()`).

Synopsis

```
oem_data_handle =
  init_oem_solve(A, mask, tol, precon, restart, max_iter,
      info);
```

Description

Initialize a `OEM_DATA`-handle which can be passed to the function pointers returned by `get_oem_solver()` (see above). The specific solver to use, as well as the storage for the solution and the load-vector, is left unspecified here; these data is given as parameters to `.call_oem_solve_[s—d—dow]()`, see below. The data handle returned by this functions eventually should be deleted by a call to `realeas_oem_solve()`, which is also described below.

Parameters

The parameters have the same meaning as the respective parameters to `oem_solve_[s|d|dow()`; the explanations are just repeated here:

`A` Pointer to a `DOF_MATRIX` storing the system matrix.

`mask` Pointer to a `DOF_SCHAR_VEC` masking out parts of the finite element space: if `mask->vec[d] >= DIRICHLET`, then `A` will act as if the d-th row would be zero. Compare also the discussion in the section about Dirichlet boundary condition, see Section 4.7.7.1

`tol` Tolerance for the residual; if the norm of the residual is less or equal `tol`, `oem_solve_[s|d|dow]()` returns the actual iterate as the approximative solution of the system.

`precon` A pointer to a structure describing the preconditioner to use, see further below in Section 4.10.7.

4.10.6 Compatibility Note. *Previous versions used a simple number here, but as the preconditioner frame-work has become much more complicated, because of the support for direct sums of finite element spaces, the code for the selection of the preconditioner has been separated from the entry-point to the solvers.*

`restart` Only used by `gmres`: the maximum dimension of the Krylov-space.

`max_iter` Maximal number of iterations to be performed by the linear solver. This can be compared with the return value – which gives the number of iterations actually performed – to determine whether the solver has achieved its goal.

`info` This 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.

Return Value

A pointer to an initialized `OEM_DATA`-structure, see the source-code listing on page 325.

4.10.7 Function (`release_oem_solve()`).

Synopsis

```
release_oem_solve(oem_data_handle);
```

Description

Release an `OEM_DATA`-handle previously acquired by a call to `init_oem_solve_[s|d|dow]()` as explained above.

Parameters

`oem_data_handle` The `OEM_DATA`-pointer to destroy.

4.10.8 Function (`call_oem_solve_[s|d|dow]()`).

Synopsis

```
iterations = call_oem_solve_[s|d|dow](oem_data_handle,
    solver, f, u);
```

Description

Call an iterative solver, as indicated by `solver`, trying to solve the linear system described by `oem_data_handle` for the unknown `u`, given the load-vector `f`. `u` is at the same time the storage for the solution and the initial guess for the iterative solver.

Parameters

With the exception of `oem_data_handle` the parameters have the same meaning as the respective parameters to `oem_solve_[s|d|dow]()`; the explanations are just repeated here:

`oem_data_handle` A `OEM_DATA`-structure, as returned by a previous call to `init_oem_solve()` (or filled in "by hand").

`f` A pointer to a `DOF_REAL[_D]_VEC[_D]` storing the load-vector of the linear system.

`u` A pointer to a `DOF_REAL[_D]_VEC[_D]` storing the initial guess on input and the approximative solution on output. In the context of interpolated Dirichlet boundary conditions special provisions have to be taken for the "Dirichlet-nodes". Compare the discussion in Section 4.7.7.1.

`solver` Use the respective OEM-solver; see above for the available keywords.

Return Value

The number of iterations the solver needed until the norm of the residual was below `tol`, or `max_iter` if the solver was not able to reach its goal before the prescribed maximum iteration count was exhausted.

4.10.9 Example. The high-level function

```
iterations = oem_solve_[s|d|dow](A, mask, f, u,
                                 solver, tol, precon,
                                 restart, max_iter, info);
```

is implemented as follows:

```
int oem_solve_s(const DOF_MATRIX *A, const DOF_SCHAR_VEC *mask,
                const DOF_REAL_VEC *f,  DOF_REAL_VEC *u,
                OEM_SOLVER solver, REAL tol, const PRECON *precon,
                int restart, int max_iter, int info)
{
  const OEM_DATA *oem;
  int iter;

  oem = init_oem_solve(A, mask, tol, precon, restart, max_iter, info);
  iter = call_oem_solve_s(oem, solver, f, u);
  release_oem_solve(oem);

  return iter;
}
```

4.10.10 Example. If it is clear which solver to use, then the call through `call_oem_solve_[s|d|dow]()` in Example 4.10.9 can be replaced by a direct call to the solver-routine like follows. Note, however, that this is a simplified example which does not take into account that `u->fe_space` could be a direct sum of finite element spaces, as explained in Section 3.7. Of course, it is just fine for application to ignore the "direct sum" feature if it is clear that it is not needed. See Example 4.10.11 for an example of how to deal with direct sums. The reader should also remember that – for simple applications – it is sufficient to use the high-level routines `oem_solve_[s|d|dow]()`, see also Example 4.10.9 for the connection between the example given here and the high-level routines.

```
const OEM_DATA *oem;
int iter;
OEM_MV_FCT solver_fct;
int dim;

oem        = init_oem_solve(A, mask, tol, precon, restart, max_iter, info);
solver_fct = get_oem_solver(CG); /* e.g. */
dim        = dof_real_vec_length(u->fe_space);
FOR_ALL_FREE_DOFS(u->fe_space->admin,
  if (dof < dim) u->vec[dof] = f->vec[dof] = 0.0);
...
solver_fct(oem, dim, f->vec, u->vec); /* maybe do this multiple times ... */
...
```

```
FOR_ALL_FREE_DOFS(u->fe_space->admin,
  if (dof < dim) f_other->vec[dof] = 0.0);
solver_fct(oem, dim, f_other->vec, u->vec); /* ... with other load-vectors */
...
release_oem_solver();
```

4.10.11 Example. A similar code-skeleton, taking direct sums of finite element spaces into account (see Section 3.7.3) would look like as quoted below. The interested reader maybe also wants to have a look at the source code `alberta-VERSION/alberta/src/Common/oem_solve.c` in the ALBERTA distribution. See Example 4.10.10 for a simpler example, ignoring that "direct sum" feature. The reader should also remember that – for simple applications – it is sufficient to use the high-level routines `oem_solve_[s|d|dow]()`, see also Example 4.10.9 for the connection between the example given here and the high-level routines.

```
const OEM_DATA *oem;
int iter;
OEM_MV_FCT solver_fct;
int dim;
REAL *uvec, *fvec;

oem        = init_oem_solve(A, mask, tol, precon, restart, max_iter, info);
solver_fct = get_oem_solver(CG); /* e.g. */
dim        = dof_real_vec_length(u->fe_space);
if (!CHAIN_SINGLE(u)) {
  uvec = MEM_ALLOC(dim, REAL);
  fvec = MEM_ALLOC(dim, REAL);
  copy_from_dof_real_vec(uvec, u);
  copy_from_dof_real_vec(fvec, f);
} else {
  FOR_ALL_FREE_DOFS(u->fe_space->admin,
      if (dof < dim) u->vec[dof] = f->vec[dof] = 0.0);
  fvec = f->vec;
  uvec = u->vec;
}
...
solver_fct(oem, dim, fvec, uvec);
...
release_oem_solver();
if (!CHAIN_SINGLE(u)) {
  copy_to_dof_real_vec(u, uvec);
  MEM_FREE(uvec, dim, REAL);
  MEM_FREE(fvec, dim, REAL);
}
```

4.10.3 SOR solvers for DOF-matrices and -vectors

The SOR and SSOR methods are implemented directly for linear systems defined by `DOF_MATRIX` and `DOF_REAL_[D_]VEC[_D]`s.

4.10.12 Remark. In contrast to the other solvers for linear systems, the SOR- and SSOR-methods described in this section do *not* support direct sums of finite element spaces (see Section 3.7).

```
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 the linear system for a scalar or decoupled vector valued problem in ALBERTA by the [Symmetric] Successive Over Relaxation 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 then `omega=1.0` is used;

`tol`: tolerance for the maximum norm of the correction; if this norm is less than or equal to `tol`, then `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.

4.10.4 Saddle-point problems, CG solver for Schur's complement

On the linear-algebra level, a linaer saddle-point problem is of the form

$$\begin{bmatrix} A & B \\ B^* & 0 \end{bmatrix} \begin{bmatrix} v \\ p \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix}, \quad f, v \in X,\ g, p \in Y, \tag{4.2}$$

with matrices A and B, unknown vectors v and p and a load vector consisting of the vector f and g. Usually, A has its origin in the discretization of an unconstraint minimization problem, B^* plays the role of a linear constraint, and p is the corresponding Lagrangian multiplier. Y is the finite element space for the Lagrangian multiplier, and X a possibly different finite element space for the principal unknown v:

If A is invertible, then it is possible to transform (4.2) into an equation for p only:

$$T\,p = B^*\,A^{-1}\,f - g, \quad T := B^*\,A^{-1}\,B, \tag{4.3}$$

where v can be reconstructed from p by $v = A^{-1}(f - B\,p)$. If A is symmetric positive definite, then so is T, and thus it is possible to solve (4.3) by means of a CG-method in this case which, interestingly, even computes v as a by-product.

In the same spirit as for the iterative solvers for "ordinary" problems, this *SPCG*-method is implemented in a fairly abstract manner, using a special data-structure to describe the saddle-point problem. The actual CG-iteration is executed by a call to the function `oem_spcg(oem_sp_data,...)`, described below in Section 4.10.14. It is the task of the application to fill that `OEM_SP_DATA`-structure (see Section 4.10.13 below). However, there are interface functions to aid the implementation of such a saddle-point solver with ALBERTA's DOF-matrices and -vectors, see Section 4.10.5 below.

```
typedef struct oem_sp_data OEM_SP_DATA;
int oem_spcg(OEM_SP_DATA *data, int dimX, const REAL *f, REAL *u, int dimY,
             const REAL *g, REAL *p);
```

4.10.13 Datatype (`OEM_SP_DATA`).

Definition

```
typedef int (*OEM_MV_FCT)(void *data, int dim, const REAL *rhs, REAL
    *u);
typedef void (*OEM_GEMV_FCT)(void *data,
                             REAL factor,
                             int dimX, const REAL *x, int dimY, REAL
                                 *y);

typedef struct oem_sp_data OEM_SP_DATA;
struct oem_sp_data
{
  OEM_MV_FCT    solve_Auf;
  void          *solve_Auf_data;
  OEM_GEMV_FCT  B;
  void          *B_data;
  OEM_GEMV_FCT  Bt;
  void          *Bt_data;
  OEM_MV_FCT    project;
  void          *project_data;
  int           (*precon)(void *ud,
                          int dimY, const REAL *g_Btu, const REAL *r,
                              REAL *Cr);
  void          *precon_data;

  WORKSPACE     *ws;

  REAL          tolerance;
  int           restart;
  int           max_iter;
  int           info;

  REAL          initial_residual;
  REAL          residual;
};
```

Components

solve_Auf() An application provided function for solving $A\,x = b$, for given initial guess and solution x and load-vector b. This can, e.g. be one of the solver-functions for ordinary problems, see Section 4.10.1.

solve_Auf_data Application data passed to `solve_Auf()` as first argument. If `solve_Auf()` is one of the solver-functions described in Section 4.10.1 or a function pointer returned by `get_oem_solver()`, then this should be a pointer to a `OEM_DATA` structure, as returned for instance by `init_oem_solver()`, see above in Section 4.10.2.

B() A pointer to an application provided function with the calling convention

```
B(B_data, factor, dimY, p, dimX, v);
```

This function must implement the operation `v` = `v` + `factor` B, p In the abstract setting the range of the operator underlying `B()` is the same as the range of the unconstrained operator `A()` such that it makes sense to apply the inverse of `A()` to the result of `B()`.

B_data Application data passed to `B()` as first argument.

Bt() The pendent to `B()`: A pointer to an application provided function with the calling convention

```
Bt(Bt_data, factor, dimX, v, dimY, p);
```

This function must implement the operation `p` = `p`+`factor` B^* `v` For practical reasons – e.g. in the context of a Stokes problem – the range of the discrete operator `Bt()` need not necessarily be the finite element space for the Lagrangian multiplier (see `precon()` and `project()` below), but often is rather the dual of that space.

Bt_data Application data passed to `Bt()` as first argument.

project() A function pointer pointing to an application provided function which has the task to project the result from `Bt()` to the finite element space for the Lagrangian multiplier. `project()` maybe `NULL` is such a projection is not needed. Arguably, this could already have been incorporated into `Bt()`, however, it is sometimes more efficient to let the discrete operator `Bt()` map to the dual of the space for the Lagrangian multiplier. See also `precon()` below. Often `project()` will just be an L^2-projection involving the inversion of a mass-matrix, which can for instance be done by a CG-method, or maybe even more efficiently with mass-lumping.

project_data Application data pointer passed as first argument to `project()`.

precon() A function pointer pointing to an application provided function which should implement a preconditioner "$C()$" for the CG-method for Schur's complement operator. `precon()` may be `NULL`. The calling convention is

```
iterations = precon(precon_data, dim, g_Btu, r, Cr);
```

where the non-self-explanatory arguments have the following meaning:

g_Btu The current value of $g - B^*u$, where u is the current iterate for the principal unknown v in the CG-method. This is the result of a call to `Bt()`, and most likely lives in the dual of the space for the Lagrangian multiplier.

r This is `project(g_Btu)`, this lives in space for the Lagrangian multiplier.

The result value of `precon()` must be stored in `Cr`. `Cr` must belong to the space for the Lagrangian multiplier. As an example, it is known that for a Quasi-Stokes problem

$$\mu\, u - \nu\, \Delta u + \nabla p = f, \quad \nabla \cdot u = 0,$$

a good choice for a preconditioner for Schur's complement CG-method is

$$C(\mathtt{r}) = \nu\, \mathtt{r} + \mu\, q, \text{ with } -\Delta q = \mathtt{g\_Btu}.$$

Note that we have omitted the boundary conditions, which, of course, have to be applied to close the differential equations mentioned above. The reader is referred to standard text-books dealing with the discretizations of saddle-point problems.

`precon_data` Data pointer passed as first argument to `precon()`.

`ws` A pointer to a work-space area. May be `NULL`. If supplied, it must point to an initialized work-space of size

$$2 * \mathtt{dimY} + \mathtt{dimX} + \max(\mathtt{dimX}, \mathtt{dimY})$$

if `precon() == NULL` and

$$3 * \mathtt{dimY} + \mathtt{dimX} + \max(\mathtt{dimX}, \mathtt{dimY}).$$

If `ws == NULL`, then `oem_spcg()` will allocate a work-space area by itself.

`tolerance` `oem_spcg()` will terminate if the norm of the CG-residual for the Lagrangian multiplier falls below `tolerance`.

`restart` Not used by `oem_spcg()`. Could be used when implementing similar iterative methods for non-symmetric saddle-point problems, e.g. by means of applying GMRES.

`max_iter` `ome_spcg()` will terminate after this many iterations of the main-loop.

`info` An integer controlling the amount of information printed to the terminal the application program runs in.

`initial_residual` Output. Upon return from `oem_spcg()` this component stores the initial residual.

`residual` Output. Upon return from `oem_spcg()` this component stores the final residual. This could be used for error recovery, e.g. if the iteration terminates because the maximum number of iterations (as specified by `max_iter`) was exhausted.

4.10.14 Function (`oem_spcg()`).

Synopsis

```
iterations = oem_spcg(sp_data, dimX, f, u, dimY, g, p);
```

Description

This function implement a CG-method for the inversion of Schur's complement operator for a linear symmetric saddle-point problem.

Parameters

sp_data A pointer to a correctly filled OEM_SP_DATA structure, as explained above. Upon return from oem_spcg(), the fields initial_residual and residual will contain the initial and the final residual of the CG-iterations.

dimX The dimension of the space for the principal unknown u.

f Load-vector for the principal equation.

u Storage for the principal unknown, and start-value for the principal unknown for the CG-method.

dimY Dimension of the space for the Lagrangian multiplier.

g Load-vector for the constraint equation.

p Storage for the Lagrangian multiplier and start-value for the CG-method.

Return Value

The number of times the main-loop of the CG-iteration was executed. If this is equal to sp_data->max_iter, then the application should also inspect sp_data->residual to determine whether the approximative solution is still acceptable.

4.10.5 Saddle-pointer solvers for DOF-matrices and -vectors

Similar to the functions explained in Section 4.10.2 there are also interface functions to mediate between the more low-level oem_spcg() function described in the previous Section 4.10.4 and the DOF-vectors and -matrices generated by ALBERTA's assemble frame-work, as described in Section 4.7. The functions below have the slight disadvantage that they take too many arguments. The interface functions support direct sums of finite element spaces (see Section 3.7) which is of some importance in the context of mixed discretizations for the Stokes-problem.

There are two interfaces available: one for a saddle-point problem with a single linear constraint, and one for a saddle-point problem with multiple linear constraints, with the restriction that the constraints are decoupled. We start with the single-constraint version oem_sp_solve() in Section 4.10.15 and continue with the multiple-constraint functions init_sp_constraint(), release_sp_constraint() and oem_sp_schur_solve() in the Sections 4.10.17-4.10.19. There is one additional support function sp_dirichlet_bound() which deals with compatibility conditions in the context of a divergence constraint and Dirichlet boundary conditions, see Section 4.10.21.

The suite of demo-programs contains example programs for the discretization of Stokes and Quasi-Stokes problems, the interested reader is referred to the programs

```
alberta-VERSION-demo/src/Common/stokes.c
```

and

```
alberta-VERSION-demo/src/Common/quasi-stokes.c.
```

The prototypes for the available functions read as follows:

```
int oem_sp_solve_dow_scl(OEM_SOLVER sp_solver,
                         REAL sp_tol, REAL tol_incr,
                         int sp_max_iter, int sp_info,
                         const DOF_MATRIX *A, const DOF_SCHAR_VEC *bound,
                         OEM_SOLVER A_solver,
                         int A_max_iter, const PRECON *A_precon,
                         DOF_MATRIX *B,
                         DOF_MATRIX *Bt,
                         DOF_MATRIX *Yproj,
                         OEM_SOLVER Yproj_solver,
                         int Yproj_max_iter, const PRECON *Yproj_precon,
                         DOF_MATRIX *Yprec,
                         OEM_SOLVER Yprec_solver,
                         int Yprec_max_iter, const PRECON *Yprec_precon,
                         REAL Yproj_frac, REAL Ypre_frac,
                         const DOF_REAL_VEC_D *f,
                         const DOF_REAL_VEC *g,
                         DOF_REAL_VEC_D *x,
                         DOF_REAL_VEC *y);
int oem_sp_solve_ds(OEM_SOLVER sp_solver,
                    REAL sp_tol, REAL tol_incr,
                    int sp_max_iter, int sp_info,
                    const DOF_MATRIX *A, const DOF_SCHAR_VEC *bound,
                    OEM_SOLVER A_solver,
                    int A_max_iter, const PRECON *A_precon,
                    DOF_MATRIX *B,
                    DOF_MATRIX *Bt,
                    DOF_MATRIX *Yproj,
                    OEM_SOLVER Yproj_solver,
                    int Yproj_max_iter, const PRECON *Yproj_precon,
                    DOF_MATRIX *Yprec,
                    OEM_SOLVER Yprec_solver,
                    int Yprec_max_iter, const PRECON *Yprec_precon,
                    REAL Yproj_frac, REAL Ypre_frac,
                    const DOF_REAL_D_VEC *f,
                    const DOF_REAL_VEC *g,
                    DOF_REAL_D_VEC *x,
                    DOF_REAL_VEC *y);

REAL sp_dirichlet_bound_dow_scl(MatrixTranspose transpose,
                                const DOF_MATRIX *Bt,
                                const DOF_SCHAR_VEC *bound,
                                const DOF_REAL_VEC_D *u_h,
                                DOF_REAL_VEC *g_h);
REAL sp_dirichlet_bound_ds(MatrixTranspose transpose,
                           const DOF_MATRIX *Bt,
                           const DOF_SCHAR_VEC *bound,
                           const DOF_REAL_D_VEC *u_h,
                           DOF_REAL_VEC *g_h);

typedef struct sp_constraint
{
  const DOF_MATRIX     *B, *Bt;
  const DOF_SCHAR_VEC *bound;
  OEM_MV_FCT           project;
  OEM_DATA             *project_data;
  OEM_MV_FCT           precon;
```

```
  OEM_DATA              *precon_data;
  REAL                  proj_factor, prec_factor;
} SP_CONSTRAINT;

SP_CONSTRAINT *init_sp_constraint(const DOF_MATRIX *B,
                                  const DOF_MATRIX *Bt,
                                  const DOF_SCHAR_VEC *bound,
                                  REAL tol, int info,
                                  const DOF_MATRIX *Yproj,
                                  OEM_SOLVER Yproj_solver,
                                  int Yproj_max_iter,
                                  const PRECON *Yproj_prec,
                                  const DOF_MATRIX *Yprec,
                                  OEM_SOLVER Yprec_solver,
                                  int Yprec_max_iter,
                                  const PRECON *Yprec_prec,
                                  void (*Yprec_bndry)(void *data,
                                                      const DOF_REAL_VEC *r,
                                                      DOF_REAL_VEC *mod_r,
                                                      DOF_REAL_VEC *Cr),
                                  void *Yprec_bndry_data,
                                  REAL Yproj_frac, REAL Yprec_frac);
void release_sp_constraint(SP_CONSTRAINT *constraint_data);
int oem_sp_schur_solve(OEM_SOLVER sp_solver,
                       REAL sp_tol, int sp_max_iter, int sp_info,
                       OEM_MV_FCT principal_inverse,
                       OEM_DATA *principal_data,
                       const DOF_REAL_VEC_D *f,
                       DOF_REAL_VEC_D *u,
                       SP_CONSTRAINT *constraint,
                       const DOF_REAL_VEC *g,
                       DOF_REAL_VEC *p,
                       ...);
```

4.10.15 Function (`oem_sp_solve_[dow_scl|ds]()`).

Synopsis

```
iterations = oem_sp_solve_[dow_scl|ds](
  sp_solver,
  sp_tol, tol_incr, sp_max_iter, sp_info,
  A, mask, A_solver, A_max_iter, A_precon,
  B, Bt,
  Yproj, Yproj_solver, Yproj_max_iter, Yproj_precon,
  Yprec, Yprec_solver, Yprec_max_iter, Yprec_precon,
  Yproj_frac, Yprec_frac,
  f, g, x, y);
```

Description

This function implements an interface between the DOF-vector and -matrix level and the low-level functions described in Section 4.10.4 above. Internally, `oem_sp_solve()` emits

calls to init_oem_solve() and initializes the support data-structure OEM_SP_DATA. Then finally the function oem_spcg() is called, see also Section 4.10.4.

oem_sp_solve() implements a preconditioner C of the form

$$C(r) = \texttt{Yproj\_frac} * \texttt{Yproj}(r) + \texttt{Yprec\_frac} * \texttt{Yprec}^{-1}(r), \tag{4.4}$$

which has the form of the usual preconditioner for a Quasi-Stokes problem, which was already mentioned in the explanation for the parameter precon() for the function oem_spcg(), see Section 4.10.14.

Parameters

sp_solver The solver used for the *outer* iteration. Currently, only a CG-method for a symmetric and positive (semi-) definite Schur's complement operator is implemented, so sp_solver must equal the symbol CG.

sp_tol The tolerance for the *outer* CG-loop.

tol_incr A decrease in tolerance for the iterative solvers for the sub-problems, like inverting the principal part A of the operator. The tolerances for the solvers for the sub-problems will be sp_tol / tol_incr.

sp_max_iter The maximum number of iterations for the outer CG-loop.

sp_info The verbosity level. The solvers for the sub-problems will inherit a decreased verbosity level of max(0, spinfo − 3).

A The matrix for the principal part of the saddle-point problem.

bound A DOF_SCHAR_VEC used to exclude DOFs from the operation of the matrix-vector routines. See semantics are as explained in the explanations for the argument mask to the function init_oem_solve(), see Section 4.10.5.

A_solver The solver to use to invert A, compare with the explanations for get_oem_solver() in Section 4.10.4 and the parameter solver to init_oem_solver().

A_max_iter The maximum number of iterations for the linear solver used to invert A.

A_precon A pointer to the descriptor for the preconditioner to use for the inversion of A, see Section 4.10.7 below.

B A pointer to the matrix implementing B, see (4.2).

Bt A pointer to the matrix implementing B^*, see (4.2). Bt may be NULL, in which case the matrix B is used, passing the Transpose flag to the matrix-vector routines, see Section 3.3.7. An application calling oem_sp_solve() with Bt == NULL most likely will want to make use of the optional parameter mask above in order to implement Dirichlet boundary conditions.

Yproj The matrix for the back-projection of the result from applying Bt to the finite element space for the constraint. Compare the remarks in the explanation of the component project() of the OEM_SP_DATA structure.

Yproj_solver The solver to use for inverting `Yproj`.

Yproj_max_iter The maximum number of iterations for inverting `Yproj`.

Yproj_precon The preconditioner for the iterative solver for the inversion of `Yproj`. See Section 4.10.7 below.

Yprec A part defining one part of the preconditioner as explained in equation (4.4). Maybe `NULL`, in which case no preconditioner will be applied in the outer CG-loop for inverting Schur's complement.

Yprec_solver The solver to use for inverting `Yprec`.

Yprec_max_iter The maximum number of iterations for inverting `Yprec`.

Yprec_precon The preconditioner for the iterative solver for the inversion of `Yprec`. See Section 4.10.7 below.

Yproj_frac See equation (4.4) above.

Yprec_frac See equation (4.4) above.

f The load vector for the principal unknown.

g The load vector for the linear constraint. Even in the case when the non-discrete problem is subject to a homogeneous constraint, it can be necessary to impose a slightly inhomogeneous constraint in the discrete setting. One notable example is the implementation of Dirichlet boundary conditions in the context of a divergence constraint. In this case interpolated Dirichlet boundary values will in general fail to fulfill the compatibility condition the discrete divergence constraint imposes on the discrete boundary values. Compare with the explanations for `sp_dirichlet_bound()` below.

x Storage for the principal of the solution, and initial guess for the CG-method.

y Storage for the Lagrangian multiplier, and initial guess for the CG-method for Schur's complement.

4.10.16 Datatype (`SP_CONSTRAINT`).

Description

In the multi-constraint case, each single constraint is described by a `SP_CONSTRAINT` structure, in order to reduce the number of parameters which have to be passed to the saddle-point solver. Such a structure can be obtained by a call to init_sp_constraint(), see below Section 4.10.17.

The meaning of the individual structure components is identical to the meaning of the respective component of the `OEM_SP_DATA` or parameter of the `oem_sp_solve()` function, the reader is therefore referred to Section 4.10.13 and Section 4.10.15 for a detailed discussion.

Definition

```
typedef struct sp_constraint
{
  const DOF_MATRIX     *B, *Bt;
  const DOF_SCHAR_VEC  *bound;
  OEM_MV_FCT           project;
  void                 *project_data;
  OEM_MV_FCT           precon;
  void                 *precon_data;
  REAL                 proj_factor, prec_factor;
} SP_CONSTRAINT;
```

Components

B See parameter B of oem_sp_solve().

Bt See parameter Bt of oem_sp_solve().

bound See parameter bound of oem_sp_solve().

project() See component project() of OEM_SP_DATA.

project_data See component project_data of OEM_SP_DATA.

precon() See component precon() of OEM_SP_DATA.

precon_data See component precon_data of OEM_SP_DATA.

proj_factor See parameter Yproj_frac of oem_sp_solve().

prec_factor See parameter Yprec_frac of oem_sp_solve().

4.10.17 Function (init_sp_constraint()).

Synopsis

```
constraint_data =
  init_sp_constraint(B, Bt, bound, tol, info,
                     Yproj, Yproj_solver, Yproj_max_iter,
                     Yproj_prec,
                     Yprec, Yprec_solver, Yprec_max_iter,
                     Yprec_prec,
                     Yprec_bndry, Yprec_bndry_data,
                     Yproj_frac, Yprec_frac);
```

Description

Allocate and initialize a SP_CONSTRAINT structure, for later use with oem_sp_schur_solve(), see Section 4.10.19 below. The meaning of the parameters is almost identical to the corresponding parameters to oem_sp_solve(), see Section 4.10.15 above.

Parameters

B See parameter B of oem_sp_solve().

Bt See parameter Bt of oem_sp_solve().

bound See parameter bound of oem_sp_solve().

tol The tolerance for the sub-solvers used to invert Yproj and Yprec (if present). Compare parameter tol_incr of oem_sp_solve().

info Control the amount of messages printed to the terminal the application has been started from. Compare parameter sp_info of oem_sp_solve().

Yproj See parameter Yproj of oem_sp_solve().

Yproj_solver See parameter Yproj_solver of oem_sp_solve().

Yproj_max_iter See parameter Yproj_max_iter of oem_sp_solve().

Yproj_prec See parameter Yproj_prec of oem_sp_solve().

Yprec See parameter Yprec of oem_sp_solve().

Yprec_solver See parameter Yprec_solver of oem_sp_solve().

Yprec_max_iter See parameter Yprec_max_iter of oem_sp_solve().

Yprec_prec See parameter Yprec_prec of oem_sp_solve().

Yprec_frac See parameter Yprec_frac of oem_sp_solve().

Yprec_bndry(data, r, mod_r, Cr) A callback for cases where the constraint has to fulfil special boundary conditions. Yprec_bndry may be NULL. The first argument to the call-back is the application provided Yprec_bndry_data-pointer specified by the following argument. r is the current residual which normally serves as load-vector for the preconditioner (see equation (4.4)), mod_r is a modifiable copy of r, and Cr is the preconditioned residual which is solved for when inverting Yprec.

Yprec_bndry_data See the description for Yprec_bndry() above; Yprec_bndry_data is the application-data pointer for that callback.

Yprec_frac See parameter Yprec_frac of oem_sp_solve().

Return Value

A pointer to an initialized SP_CONSTRAINT structure, which can be passed as argument to oem_sp_schur_solve() described in Section 4.10.19 below. The return structure should be deleted by a call to release_sp_constraint(), see below.

4.10.18 Function (release_sp_constraint()).

Synopsis

```
release_sp_constraint(constraint_data);
```

Description

Release the resources associated with a SP_CONSTRAINT structure as returned by init_sp_constraint().

Parameters

constraint_data A pointer to a SP_CONSTRAINT structure previously acquired by a call to init_sp_constraint(), see Section 4.10.17.

4.10.19 Function (`oem_sp_schur_solve()`).

Prototype

```
int oem_sp_schur_solve(OEM_SOLVER sp_solver,
                       REAL sp_tol, int sp_max_iter, int sp_info,
                       OEM_MV_FCT principal_inverse,
                       OEM_DATA *principal_data,
                       const DOF_REAL_VEC_D *f,
                       DOF_REAL_VEC_D *u,
                       SP_CONSTRAINT *constraint,
                       const DOF_REAL_VEC *g,
                       DOF_REAL_VEC *p,
                       ...);
```

Synopsis

```
iterations =
  oem_sp_schur_solve(sp_solver,
                     sp_tol, sp_max_iter, sp_info,
                     A_inverse, A_data, f, u,
                     constraint, g, p,
                     ...);
```

Description

Solve a saddle-point problem with possibly multiple, decoupled linear constraints by inverting the associated Schur's complement operator by means of an iterative method. Currently, only a CG-method is implemented, so the principal operator `A` has to be symmetric and positive (semi-) definite.

Parameters

`sp_solver` The solver used for the *outer* iteration. Currently, only a CG-method for a symmetric and positive (semi-) definite Schur's complement operator is implemented, so `sp_solver` must equal the symbol `CG`.

`sp_tol` The tolerance for the *outer* CG-loop.

`sp_max_iter` The maximum number of iterations for the outer CG-loop.

`sp_info` A "verbosity-level" controlling the amount of information printed to the terminal the application is running from.

`A_inverse()` Pointer to a solver-function, for instance as returned by `get_oem_solver()`.

`A_data` A pointer to a data structure needed by `A_inverse()`, the pointer is passed as first argument to `A_inverse()`. See also `init_oem_solver()` in Section 4.10.5.

`f` The load-vector for the principal equation.

`u` Storage for the principal unknown (solution), and initial guess for the CG-method.

constraint A SP_CONSTRAINT structure, for instance as generated by a call to init_sp_constraint(), see Section 4.10.17, see also release_sp_constraint(), Section 4.10.18.

g The load vector for the possibly inhomogeneous linear constraint described by the parameter constraint. Note that only *scalar* constraints are supported by this function, consequently g is a scalar DOF_REAL_VEC.

p Storage for the Lagrangian multiplier associated with constraint and initial guess for the Lagrangian multiplier in the outer CG-loop.

... More constraints may be added after the parameter p, each as a triple

```
..., constraint_data, load_vector, lagrangian_multiplier, ...
```

All constraints must be decoupled from each other. After the final constraint a NULL-pointer must be passed to oem_sp_schur_solve(), if only a single constraint is needed, then the first argument after the parameter p must already be a NULL-pointer.

Return Value

The number of iterations of the outer CG-loop for the inversion of Schur's complement.

Examples

The single-constraint oem_sp_solve() functions are implemented on top of oem_sp_schur_solve(). The interested reader may want to have a look at alberta-VERSION/alberta/src/Common/oem_sp_solve.c. See also Example 4.10.20 below.

4.10.20 Example. A brief demonstration of how oem_sp_schur_solve() could be used in the single constraint case is given below. The reader is referred to Section 4.10.7 below for the documentation of the functions related to preconditioning.

```
... /* other stuff */

A_prec = init_precon_from_type(A, NULL /* bound */, sub_info, &A_prec_type);
A_oem  = init_oem_solve(A, NULL, tol, A_prec, -1, A_miter, sub_info);

Yproj_prec = init_precon_from_type(Yproj, NULL /* bound */, sub_info,
                                   Yproj_prec_type);
Yprec_prec = init_precon_from_type(Yprec, NULL /* bound */, sub_info,
                                   Yprec_prec_type);
SP_CONSTRAINT *div_constraint =
  init_sp_constraint(B, Bt, NULL, tol / 100.0, MAX(0, info - 3),
                     Yproj, Yproj_solver, Yproj_miter, Yproj_prec,
                     Yprec, Yprec_solver, Yprec_miter, Yprec_prec,
                     nu, mu);

oem_sp_schur_solve(solver, tol, miter, info,
                   get_oem_solver(A_solver), A_oem,
                   f_h, u_h,
                   div_constraint,
                   g_h, p_h,
                   NULL);
```

```
release_sp_constraint(div_constraint);
release_oem_solve(A_oem);

... /* other stuff */
```

4.10.21 Function (sp_dirichlet_bound_[dow_scl|ds]()).

Prototype

```
REAL sp_dirichlet_bound_dow_scl(MatrixTranspose transpose,
                                const DOF_MATRIX *Bt,
                                const DOF_SCHAR_VEC *bound,
                                const DOF_REAL_VEC_D *u,
                                DOF_REAL_VEC *g);
REAL sp_dirichlet_bound_ds(MatrixTranspose transpose,
                           const DOF_MATRIX *Bt,
                           const DOF_SCHAR_VEC *bound,
                           const DOF_REAL_D_VEC *u,
                           DOF_REAL_VEC *g);
```

Synopsis

```
flux_excess = sp_dirichlet_bound_[dow_scl|ds](
  transpose, Bt, bound, u, g);
```

Description

If a flow field u is subject to a divergence constraint and has to satisfy Dirichlet boundary values h on the entire boundary of a domain Ω, and if the test-space for the Lagrangian multiplier contains the function which is constant and equal to 1 on the entire domain, then the Dirichlet boundary values have to satisfy the compatibility condition

$$0 = \int_\Omega 1 \operatorname{div} u = -\int_{\partial\Omega} u \cdot \nu = -\int_{\partial\Omega} h \cdot \nu. \tag{4.5}$$

This compatibility conditions has also to be satisfied in the discrete setting, however, if one simply uses Lagrange-interpolation to implement Dirichlet boundary values, then the discrete Dirichlet boundary values in general violate this condition, and consequently the discrete saddle point problem does not have a solution in this case. One way to cope with this problem is to solve a slightly inhomogeneous saddle-point problem, where a load-vector for the Lagrangian multiplier compensates for the "flux-excess" of the interpolated Dirichlet boundary conditions (another way would be to modify the boundary values, of course).

`sp_dirichlet_bound()` computes a load-vector for the Lagrangian multiplier by applying the B^* operator to the boundary values. Of course, this makes only sense if the discrete boundary values asymptotically approximate the compatibility condition in the limit $h \to \infty$.

Parameters

transpose If equal to Tranpose, then the following parameter Bt actually is not B^*, but B. sp_dirichlet_bound() internally uses the transposed matrix for computing the load-vector g. If the parameter Bt is actually B^*, the transpose should be set to NoTranpose.

Bt A pointer to the DOF-matrix implementing the B^* matrix from equation (4.3), or the B-matrix if transpose == Transpose.

bound A DOF_SCHAR_VEC, if bound->vec[dof] >= DIRICHLET, then the corresponding DOF belongs to a Dirichlet boundary. bound *must not* be NULL, sp_dirichlet_bound() just works on the linear algebra level and does not loop over the mesh-elements. A suitable boundary-flag vector can be obtained by a call to the function dirichlet_bound(), see also Section 4.7.7.1.

If sp_dirichlet_bound() encounters DOFs with bound->vec[dof] <= NEUMANN, then it returns immediately to the caller and does not modify the load-vector g. See also Section 3.2.4.

u The initial value for the principle unknown, sp_dirichlet_bound() expects that u already carries the Dirichlet boundary values.

g Storage for the load-vector to compensate for the flux-excess. Note that the application has to initialize g prior to calling sp_dirichlet_bound(), which works also in the case of an inhomogeneous divergence constraint. In that case the compatibility condition has to be modified in the obvious manner. Anyhow, sp_dirichlet_bound() works additive.

Return Value

The total flux excess over the boundary segments of the domain, or 0.0 if for any DOF with bound->vec[DOF] <= NEUMANN was encountered.

Examples

The interested read is referred to the program

```
alberta-VERSION-demo/src/Common/stokes.c
```

4.10.6 OEM matrix-vector functions for DOF-matrices and -vectors

The general oem_...() solvers all need pointers to matrix-vector multiplication routines which do not accept arguments of type DOF_REAL_[D_]VEC[_D] and a DOF_MATRIX but work directly on flat REAL-arrays. For the application to a scalar or vector-valued linear system described by a DOF_MATRIX (and an optional DOF_SCHAR_VEC which can be used to honour Dirichlet boundary conditions, see Section 4.7.7.1), the following routines are provided:

```
typedef int (*OEM_MV_FCT)(void *data, int dim, const REAL *rhs, REAL *u);

OEM_MV_FCT oem_init_mat_vec(void **dataptrptr,
                            MatrixTranspose transpose, const DOF_MATRIX *A,
                            const DOF_SCHAR_VEC *mask);
void exit_oem_mat_vec(void *dataptr)
```

4.10.22 Example. A short example demonstrating the function listed above. These are stripped-down versions of `init/release_oem_solve()` explained in Section 4.10.2. The interested reader is referred to `alberta-VERSION/alberta/src/Common/oem_solve.c` for the full source code.

```
OEM_DATA *simple_init_oem_solve(const DOF_MATRIX *A,
                                const DOF_SCHAR_VEC *mask,
                                REAL tol, int max_iter, int info)
{
  OEM_DATA      *oem;
  const MatrixTranspose transpose = NoTranspose;

  oem = MEM_CALLOC(1, OEM_DATA);
  oem->mat_vec   = init_oem_mat_vec(&oem->mat_vec_data, transpose, A, mask);
  oem->ws        = NULL; /* work-space,
                          * let the solvers handle this point for themselves.
                          */
  oem->tolerance = tol;
  oem->max_iter  = max_iter;
  oem->info      = MAX(0, info);

  return oem;
}

void simple_release_oem_solve(const OEM_DATA *_oem)
{
  OEM_DATA *oem = (OEM_DATA *)_oem;

  exit_oem_mat_vec(oem->mat_vec_data);
  MEM_FREE(oem, 1, OEM_DATA);
}
```

4.10.23 Function (`init_oem_mat_vec()`)**.**

Synopsis

```
mat_vec_fct =
  oem_init_mat_vec(&mv_data_ptr, transpose, A, mask);
```

Description

Return a pointer to a function implementing the matrix-vector operation of the matrix `A` with a `DOF_REAL[_D]_VEC[_D]`. Of course, a matrix-vector product between a `DIM_OF_WORLD` × `DIM_OF_WORLD` block-matrix and a scalar `DOF_REAL_VEC` does not make sense. This function is fully aware of ALBERTA's implementation of direct sums of finite element spaces, as described in Section 3.7.

Parameters

`mv_data_ptr` After calling this function, `mv_data_ptr` will point to a control structure which must be passed as first argument to the function returned by `init_oem_mat_vec()`. The application can call `exit_oem_mat_vec()` to release the memory resources allocated by `init_oem_mat_vec()`.

transpose One of `Transpose` or `NoTranspose`, indicating the matrix-vector operation should be performed with either the transposed or non-transposed matrix.

A A pointer to a `DOF_MATRIX`.

mask A pointer to a `DOF_SCHAR_VEC` which can be used to exclude DOFs from the matrix-vector product. `mask` can be `NULL`. See Section 4.7.7.1 for further explanations.

Return Value

A function pointer, pointing to the function actually implementing the matrix-vector operation. This function obeys the calling convention for the matrix-vector routines in the `OEM_DATA` structure, see Section 4.10.1 above.

Examples See Example 4.10.22.

4.10.24 Function (`exit_oem_mat_vec()`).

Synopsis

```
exit_oem_mat_vec(mv_data_ptr);
```

Description

Release the resources previously allocated by a call to `init_oem_mat_vec()`.

Parameters

mv_data_ptr The data-pointer allocated by `init_oem_mat_vec()`.

Examples See Example 4.10.22.

4.10.7 Preconditioners

4.10.25 Compatibility Note. *The* `get_XXX_precon()` *functions no longer carry a* `..._[s|d|dow|`*-suffix. This has been dropped, because the* `DOF_MATRIX` *structure now carries its own block-type, and the finite element spaces described by the* `FE_SPACE` *structure now know about the dimension of the range their elements are mapping to.*

See also Compatibility Note 4.10.1 above for further remarks.

The interface functions described in Section 4.10.2 and Section 4.10.5 which call the iterative solvers described in Section 4.10.1 and Section 4.10.4 all need a pointer to a `PRECON` structure. Such a structure can either be initialized by calls to one of the `get_XXX_precon()` functions described in the Sections 4.10.27-4.10.31:

```
const PRECON *get_diag_precon(const DOF_MATRIX *A,
                              const DOF_SCHAR_VEC *bound);
const PRECON *get_HB_precon(const DOF_MATRIX *matrix,
                            const DOF_SCHAR_VEC *bound,
                            int info);
const PRECON *get_BPX_precon(const DOF_MATRIX *matrix,
                             const DOF_SCHAR_VEC *bound,
```

```
                                    int info);
const PRECON *get_SSOR_precon(const DOF_MATRIX *A,
                              const DOF_SCHAR_VEC *bound,
                              REAL omega,
                              int n_iter);
const PRECON *get_ILUk_precon(const DOF_MATRIX *A,
                              const DOF_SCHAR_VEC *mask,
                              int ilu_level, int info);
```

These functions implement a diagonal and an SSOR preconditioner and two hierarchical basis preconditioners (classical Yserentant [28] and Bramble-Pasciak-Xu [4] types). The *ILU(k)* preconditioner is the one described in [3].

Another possibility to get access to preconditioners are calls to the following functions (see Sections 4.10.33-4.10.36), which also implement preconditioners for the block-matrices which arise in the context of direct sums of finite element spaces (see Section 3.7):

```
const PRECON *init_oem_precon(const DOF_MATRIX *A,
                              const DOF_SCHAR_VEC *mask,
                              int info, OEM_PRECON precon,
                              ... /* ssor_omega, ssor_n_iter etc. */);
const PRECON *vinit_oem_precon(const DOF_MATRIX *A,
                               const DOF_SCHAR_VEC *mask,
                               int info, OEM_PRECON,
                               va_list ap);
const PRECON *init_precon_from_type(const DOF_MATRIX *A,
                                    const DOF_SCHAR_VEC *mask,
                                    int info,
                                    const PRECON_TYPE *prec_type);
```

4.10.26 Datatype (PRECON).

Description

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 memory which was allocated during initialization, e.g. All three functions are collected in the structure

Definition

```
typedef struct precon PRECON;
struct precon
{
  void    *precon_data;

  bool    (*init_precon)(void *precon_data);
  void    (*precon)(void *precon_data, int n, REAL *vec);
  void    (*exit_precon)(void *precon_data);
};
```

Components

precon_data data for the preconditioner; always the first argument to the functions `init_precon()`, `precon()`, and `exit_precon()`.

init_precon(precon_data) pointer to a function for initializing the preconditioning method; the return value is `false` if initialization fails, otherwise `true`.

precon(precon_data) pointer to a function for executing the preconditioning method;

`precon` can be used as the entry `left_precon` or `right_precon` in an `OEM_DATA` structure together with `precon_data` as the corresponding pointer `left_precon_data` respectively `right_precon_data`.

exit_precon(precon_data) frees all data used by the preconditioning method.

4.10.27 Function (get_diag_precon()).

Prototype

```
const PRECON *get_diag_precon(const DOF_MATRIX *A,
                              const DOF_SCHAR_VEC *bound);
```

Synopsis

```
precon_ptr = get_diag_precon(A, bound);
```

Description

Initialize a PRECON structure describing a diagonal preconditioner for `A`. The application should call `precon_ptr->exit_precon(precon_ptr)` to release the resources associated with `precon_ptr` ones the preconditioner is no longer needed. But note that the solver interface-functions `oem_solve()` and `release_oem_solve()` call `exit_precon()` on their own.

Parameters

A The matrix to compute the diagonal preconditioner for.

bound A flag-vector, masking out specific DOFs, compare the explanations for the `mask` parameter to `oem_solve()`, see Section 4.10.2. `bound` may be `NULL`.

Return Value

A pointer to an initialized PRECON structure implementing the preconditioner, see Section 4.10.26.

4.10.28 Function (get_HB_precon()).

Prototype

```
const PRECON *get_HB_precon(const DOF_MATRIX *matrix,
                            const DOF_SCHAR_VEC *bound,
                            int info);
```

Synopsis

```
precon_ptr = get_HB_precon(A, bound, info);
```

Description

Initialize a PRECON structure describing a hierarchical preconditioner, as described in [28]. The application should call `precon_ptr->exit_precon(precon_ptr)` to release the resources associated with `precon_ptr` once the preconditioner is no longer needed. But note that the solver interface-functions `oem_solve()` and `release_oem_solve()` call `exit_precon()` on their own.

Parameters

A The matrix to compute the preconditioner for.

bound A flag-vector, masking out specific DOFs, compare the explanations for the `mask` parameter to `oem_solve()`, see Section 4.10.2. `bound` may be `NULL`.

info An integer controlling the amount of information printed to the terminal the application is running in (larger values mean more "noise").

Return Value

A pointer to an initialized PRECON structure implementing the preconditioner, see Section 4.10.26.

4.10.29 Function (get_BPX_precon()).

Prototype

```
const PRECON *get_BPX_precon(const DOF_MATRIX *matrix,
                             const DOF_SCHAR_VEC *bound,
                             int info);
```

Synopsis

```
precon_ptr = get_BPX_precon(A, bound, info);
```

Description

Initialize a PRECON structure describing the BPX-preconditioner, as described in [4]. The application should call `precon_ptr->exit_precon(precon_ptr)` to release the resources associated with `precon_ptr` once the preconditioner is no longer needed. But note that the solver interface-functions `oem_solve()` and `release_oem_solve()` call `exit_precon()` on their own.

Parameters

A The matrix to compute the preconditioner for.

bound A flag-vector, masking out specific DOFs, compare the explanations for the mask parameter to oem_solve(), see Section 4.10.2. bound may be NULL.

info An integer controlling the amount of information printed to the terminal the application is running in (larger values mean more "noise").

Return Value

A pointer to an initialized PRECON structure implementing the preconditioner, see Section 4.10.26.

4.10.30 Function (get_SSOR_precon()).

Prototype

```
const PRECON *get_SSOR_precon(const DOF_MATRIX *A,
                              const DOF_SCHAR_VEC *bound,
                              REAL omega,
                              int n_iter);
```

Synopsis

```
precon_ptr = get_SSOR_precon(A, bound, info);
```

Description

Initialize a PRECON structure describing an SSOR-preconditioner. The application should call precon_ptr->exit_precon(precon_ptr) to release the resources associated with precon_ptr once the preconditioner is no longer needed. But note that the solver interface-functions oem_solve() and release_oem_solve() call exit_precon() on their own.

Parameters

A The matrix to compute the preconditioner for.

bound A flag-vector, masking out specific DOFs, compare the explanations for the mask parameter to oem_solve(), see Section 4.10.2. bound may be NULL.

omega The relaxation parameter.

n_iter The number of SSOR-iterations to perform.

Return Value

A pointer to an initialized PRECON structure implementing the preconditioner, see Section 4.10.26.

4.10.31 Function (get_ILUk_precon()).

Prototype

```
const PRECON *get_ILUk_precon(const DOF_MATRIX *A,
                              const DOF_SCHAR_VEC *mask,
                              int ilu_level, int info);
```

Synopsis

```
precon_ptr = get_ILUk_precon(A, bound, info);
```

Description

Initialize a `PRECON` structure describing an $ILU(k)$-preconditioner as described [3]. This preconditioner uses a combinatorical, "level"-based strategy to control the amount of fill-in generated by the incomplete LU-factorization. The preconditioner can benefit from re-ordering the DOFs in a way that the amount of fill-in generated by a complete LU-factorization would be minimized. Currently, ALBERTA searches for a library `libgpskca` and expects that this library contains the functions of the `GPSKCA` package from www.netlib.org, [16].

Note the level-based fill-in control has the disadvantage that the generated preconditioner may not even be positive definite, even if `A` is spd. On the other hand, $ILU(k)$ may still be spd even if `A` is not.

The application should call `precon_ptr->exit_precon(precon_ptr)` to release the resources associated with `precon_ptr` once the preconditioner is no longer needed. But note that the solver interface-functions `oem_solve()` and `release_oem_solve()` call `exit_precon()` on their own.

Parameters

`A` The matrix to compute the preconditioner for.

`bound` A flag-vector, masking out specific DOFs, compare the explanations for the `mask` parameter to `oem_solve()`, see Section 4.10.2. `bound` may be `NULL`.

`level` The control parameter for the amount of fill-in, see [3].

`info` An integer controlling the amount of information printed to the terminal the application is running in (larger values mean more "noise").

Return Value

A pointer to an initialized `PRECON` structure implementing the preconditioner, see Section 4.10.26.

4.10.32 Datatype (`OEM_PRECON`).

Definition

```
typedef enum {
  PreconEnd   = -1,
  PreconRepeat = PreconEnd,
  NoPrecon    = 0,
  DiagPrecon  = 1,
  HBPrecon    = 2,
  BPXPrecon   = 3,
  SSORPrecon  = 4,
  __SSORPrecon = 5,
  ILUkPrecon  = 6,
  BlkDiagPrecon = 512,
  BlkSSORPrecon = 513,
} OEM_PRECON;
```

Symbols

PreconEnd

PreconRepeat Terminate the variable argument list of `init_oem_precon()`, see Section 4.10.33 in the context of block-matrix preconditioners for block-matrices having their origin in direct-sum structure of the underlying finite element spaces (see Section 3.7).

NoPrecon

DiagPrecon

HBPrecon

BPXPrecon Self-explanatory, select the respective preconditioner.

SSORPrecon Select an SSOR-preconditioner with `omega == 1.0` and `n_iter == 2`.

__SSORPrecon Select an SSOR-preconditioner with control over `omega` and `n_iter`.

ILUkPrecon Self explanatory.

BlkDiagPrecon Select a preconditioner which acts on a block-matrix structure induced by a finite element space with is composed of several components as a direct sum (see Section 3.7).

BlkSSORPrecon Currently not supported.

4.10.33 Function (`init_oem_precon()`).

Prototype

```
const PRECON *init_oem_precon(const DOF_MATRIX *A,
                              const DOF_SCHAR_VEC *bound,
                              int info, OEM_PRECON precon_enum,
                              ... /* ssor_omega, ssor_n_iter etc. */);
const PRECON *vinit_oem_precon(const DOF_MATRIX *A,
                               const DOF_SCHAR_VEC *bound,
                               int info, OEM_PRECON precon_enum,
                               va_list ap);
```

Synopsis

```
precon = init_oem_precon(A, bound, info, precon_enum, ...);
precon = vinit_oem_precon(A, bound, info, precon_enum, ap);
```

Description

These two function initialize a PRECON structure, based on the value of a descriptive enumeration symbol. The returned structure can then be passed to oem_solve() or init_oem_solve(), as described in Section 4.10.2. In contrast to the get_XXX_precon() functions described above these two functions support matrices with the block-matrix structure implied by using direct sums of finite element spaces, see Section 3.7 for further explanations.

For the difference between the ... "argument" and the ap argument the reader is referred to any text-book dealing with the C-programming language.

Parameters

A The matrix to compute the preconditioner for.

bound A flag-vector, masking out specific DOFs, compare the explanations for the mask parameter to oem_solve(), see Section 4.10.2. bound may be NULL.

info An integer controlling the amount of information printed to the terminal the application is running in (larger values mean more "noise").

precon_enum An enumeration value as defined by OEM_PRECON, see Section 4.10.32, selecting the respective preconditioner to use.

..., ap A variable-length argument list, providing additional parameters needed by some of the preconditioners, as explained below:

__SSORPrecon The two arguments following precon_enum must specify the relaxation parameter omega and the number of iterations n_iter to perform.

ILUkPrecon The argument following precon_enum must specify the control-parameter k controlling the amount of fill-in.

BlkDiagPrecon The parameters following precon_enum must specify the type and parameters for the preconditioners for the diagonal blocks. It is the responsibility of the calling application to ensure that enough preconditioners are defined. An example to generate a block-diagonal preconditioner for a 3×3 block-matrix (e.g. in the context of a "Crouzeix-Raviart" discretization for the Stokes-problem in 3d) would be

```
precon = init_oem_precon(A, NULL, 3 /* info */, BlkDiagPrecon,
                         __SSORPrecon, 1.5, 2, DiagPrecon,
                             DiagPrecon);
```

The symbol PreconRepeat has a special meaning: it indicates that the last specified preconditioner should also be used for all other blocks. In the 3×3 example given above, the following code-fragment would select diagonal preconditioning for all blocks¿

```
precon = init_oem_precon(A, NULL, 3 /* info */, BlkDiagPrecon,
                         DiagPrecon, PreconRepeat);
```

Return Value

A pointer to an initialized `PRECON` structure implementing the preconditioner, see Section 4.10.26.

4.10.34 Datatype (`PRECON_TYPE`).

Description

A data structure which can be use to define more complex preconditioners. The purpose of this structure is to avoid defining functions with an endless number of arguments. This "parameter-transport-structure" can be passed to `init_precon_from_type()`, instead of calling `init_oem_precon()`. The actual definition looks somewhat complicated and maybe ugly, but using this structure is more or less straight-forward, have a look at Example 4.10.35 below.

Definition

```
#define N_BLOCK_PRECON_MAX 10

struct __precon_type {
  OEM_PRECON type;
  union {
    struct {
      REAL omega;
      int n_iter;
    } __SSOR;
    struct {
      int level;
    } ILUk;
  } param;
};

typedef struct precon_type
{
  OEM_PRECON type;
  union {
    struct {
      REAL omega;
      int n_iter;
    } __SSOR;
    struct {
      int level;
    } ILUk;
    struct {
      struct __precon_type precon[N_BLOCK_PRECON_MAX];
    } BlkDiag;
    struct {
      struct __precon_type precon[N_BLOCK_PRECON_MAX];
      REAL omega;
      int  n_iter;
    } BlkSSOR;
  } param;
} PRECON_TYPE;
```

Components

type One of the symbolic constants defined by the OEM_PRECON enumeration type. See Section 4.10.32.

param If the preconditioner defined by type needs additional parameters, then the corresponding section in the param component has to be filled. The names of the structure components correspond to the parameters for the get_XXX_precon() functions described above, currently, only __SSORPrecon, ILUkPrecon and, of course, BlkDiagPrecon need additional parameters. For the latter, the param component contains an array of N_BLOCK_PRECON_MAX many struct __precon_type sub-structures for storing additional parameters possibly needed by the sub-preconditioners.

4.10.35 Example. Two short examples demonstrating the use of the PRECON_TYPE structure defined above.

- Defining an SSOR preconditioner with control over the relaxation parameter and the number of iterations:

```
PRECON_TYPE prec;
prec.type = __SSORPrecon;
prec.param.__SSOR.omega  = 1.5;
prec.param.__SSOR.n_iter = 2;
```

- Defining a preconditioner for a block-matrix resulting from using a direct sum of finite element spaces

```
PRECON_TYPE prec;
prec.type = BlkDiagPrecon;
prec.param.BlkDiag.precon[0].type = __SSOR;
prec.param.BlkDiag.param.precon[0].__SSOR.omega  = 1.0;
prec.param.BlkDiag.param.precon[0].__SSOR.n_iter = 1;
for (i = 1; i < 3; i++) {
  prec.param.BlkDiag.precon[i].type = DiagPrecon;
}
```

4.10.36 Function (init_precon_from_type()).

Prototype

```
const PRECON *init_precon_from_type(const DOF_MATRIX *A,
                                    const DOF_SCHAR_VEC *bound,
                                    int info,
                                    const PRECON_TYPE *prec_type);
```

Synopsis

```
precon = init_precon_from_type(A, bound, info, prec_type);
```

Description

Initialize a PRECON structure, based on contents of the prec_type parameter. The returned structure can then be passed to oem_solve() or init_oem_solve(), as described in Section 4.10.2. In contrast to the get_XXX_precon() functions described above these two functions support matrices with the block-matrix structure implied by using direct sums of finite element spaces, see Section 3.7 for further explanations.

Parameters

A The matrix to compute the preconditioner for.

bound A flag-vector, masking out specific DOFs, compare the explanations for the mask parameter to oem_solve(), see Section 4.10.2. bound may be NULL.

info An integer controlling the amount of information printed to the terminal the application is running in (larger values mean more "noise").

prec_type A pointer to a structure of type PRECON_TYPE, as described in Section 4.10.34 above, describing the preconditioner to generate.

Return Value

A pointer to an initialized PRECON structure implementing the preconditioner, see Section 4.10.26.

Examples

The function init_oem_precon() (see Section 4.10.33) is implemented on top of init_precon_from_type(). The interested reader is referred to the source code in alberta-VERSION/alberta/src/Common/oem_solver.c

4.10.8 Multigrid solvers

A abstract framework for multigrid solvers is available. The main data structure for the multigrid solver MG() is

```
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        (*mg_restrict)(MULTI_GRID_INFO *mg_info, int level);
  void        (*mg_prolongate)(MULTI_GRID_INFO *mg_info, int level);
  void        (*exact_solver)(MULTI_GRID_INFO *mg_info, int level);
  REAL        (*mg_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 level of information produced by the multigrid method.

init_multi_grid pointer to a function for initializing the multigrid method; may be `NULL`; if not `NULL`, `init_multi_grid(mg_info)` initializes data needed by the multigrid method, returns `true` if an error occurs.

pre_smooth pointer to a function for performing the smoothing step before coarse grid corrections;
`pre_smooth(mg_info, level, n)` performs `n` smoothing iterations on grid `level`.

in_smooth pointer to a function for performing the smoothing step between coarse grid corrections;
`in_smooth(mg_info, level, n)` performs `n` smoothing iterations on grid `level`.

post_smooth pointer to a function for performing the smoothing step after coarse grid corrections;
`post_smooth(mg_info, level, n)` performs `n` smoothing iterations on grid `level`.

mg_restrict pointer to a function for computing and restricting the residual to a coarser level;
`mg_restrict(mg_info, level)` computes and restricts the residual from grid `level` to next coarser grid (`level-1`).

mg_prolongate pointer to a function for prolongating and adding coarse grid corrections to the fine grid solution;
`mg_prolongate(mg_info, level)` prolongates and adds the coarse grid (`level-1`) correction to the fine grid solution on grid `level`.

exact_solver pointer to a function for the "exact" solver;
`exact_solver(mg_info, level)` computes the "exact" solution of the problem on grid `level` with tolerance `mg_info->exact_tolerance`.

mg_resid pointer to a function for computing the norm of the actual residual; `mg_resid(mg_info, level)` returns the norm of residual on grid `level`.

exit_multi_grid a pointer to a cleanup routine, may be `NULL`; if not `NULLexit_multi_grid(mg_info)` is called after termination of the multigrid method for freeing used data.

data pointer to application dependent data, holding information on or about different grid levels, e.g.

The abstract multigrid solver is implemented in the routine

```
int MG(MULTI_GRID_INFO *)
```

Description:

MG(mg_info) based upon information given in the data structure `mg_info`, the subroutine `MG()` iterates until the prescribed tolerance is met or the prescribed number of multigrid cycles is performed.

Main parts of the `MG()` 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->mg_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);
  }
}
```

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 *, DOF_REAL_VEC *, const DOF_REAL_VEC *,
         const DOF_SCHAR_VEC *, REAL, int, int, char *);
```

Description:

`mg_s(matrix, u, f, bound, tol, max_iter, info, prefix)` solves the linear system for a scalar valued problem by a multigrid method; the return value is the number of performed iterations;

`matrix` is a pointer to a DOF matrix storing the system matrix, `u` is a pointer to a DOF vector for the solution, holding an initial guess on input; `f` is a pointer to a DOF vector storing the right hand side and `bound` a pointer to a DOF vector with information about boundary DOFs; `bound` must not be `NULL` if Dirichlet DOFs are used;

`tol` is the tolerance for multigrid solver, `max_iter` the maximal number of multigrid iterations and `info` gives the level of information for the solver;

`prefix` is a parameter key prefix for the initialization of additional data via `GET_PARAMETER`, see Table 4.8, may be `NULL`; an SOR smoother (`mg_s_info->smoother=1`) and an SSOR smoother (`smoother=2`) are available; 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 *, const DOF_SCHAR_VEC *, int, char *);
int mg_s_solve(MG_S_INFO *, DOF_REAL_VEC *, const DOF_REAL_VEC *, REAL, int);
void mg_s_exit(MG_S_INFO *);
```

Description:

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	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 4.8: Parameters read by mg_s() and mg_s_init()

mg_s_init(matrix, bound, info, prefix) function for initializing a multigrid method for solving a scalar valued problem by mg_s_solve(); the return value is a pointer to data used by mg_s_solve() and is the first argument to this function; the structure MG_S_INFO contains matrices and vectors for linear problems on all used grid levels.
matrix is a pointer to a DOF matrix storing the system matrix, bound a pointer to a DOF vector with information about boundary DOFs; bound must not be NULL if Dirichlet DOFs are used;
info gives the level of information for mg_s_solve(); prefix is a parameter key prefix for the initialization of additional data via GET_PARAMETER, see Table 4.8, may be NULL.

mg_s_solve(mg_s_info, u, f, tol, max_iter) solves the linear system for a scalar valued problem by a multigrid method; the routine has to be initialize by mg_s_init() and the return value mg_s_info of mg_s_init() is the first argument; the return value of mg_s_solve() is the number of performed iterations;
u is a pointer to a DOF vector for the solution, holding an initial guess on input; f is a pointer to a DOF vector storing the right hand side; tol is the tolerance for multigrid solver, max_iter the maximal number of multigrid iterations;
the function may be called several times with different right hand sides f.

mg_s_exit(mg_s_info) frees data needed for the multigrid method and which is allocated by mg_s_init().

4.10.37 Remark. The multigrid solver is currently available only for Lagrange finite elements of first order (lagrange1). An implementation for higher order elements is future work.

4.10.9 Nonlinear solvers

For the solution of a nonlinear equation

$$u \in \mathbb{R}^N : \qquad F(u) = 0 \qquad \text{in } \mathbb{R}^N \tag{4.6}$$

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.

The data structure (defined in `alberta_util.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 `NULL`, 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 system 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 `NULL`, 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 `NULL`, 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 (4.6) 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 *);
int nls_newton_br(NLS_DATA *, REAL, 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_ds()` 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

$$\|DF(u_k)^{-1}F(u_k + \tau d_k)\| \leq (1 - \tfrac{1}{2}\tau)\|d_k\|$$

holds, where $\|.\|$ is the problem dependent norm, if `nls_data->norm` is not `NULL`, 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_ds()` 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_fs() 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

$$\|F(u_k + \tau d_k)\| \leq (1 - \tfrac{1}{2}\tau)\|F(u_k)\|$$

holds, where $\|.\|$ is the problem dependent norm, if nls_data->norm is not NULL, 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_fs() needs a workspace for storing 3*dim additional REALs.

nls_newton_br(nls_data, delta, dim, u0) solves a nonlinear system by a global Newton method by Bank and Rose [1]; 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. delta is a parameter with $\delta \in (0, 1 - \alpha_0)$, where $\alpha_0 = \|DF(u_0)\, u_0 + F(u_0)\|/\|F(u_0)\|$; dim is the dimension of the nonlinear system, and u0 the initial guess on input and the solution on output; nls_newton_br() 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 by the method of Bank and Rose are performed; 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_br() needs a workspace for storing 3*dim additional REALs.

4.11 Graphics output

ALBERTA provides one and two dimensional interactive graphic subroutines built on the X–Windows and GL/OpenGL interfaces, and one, two and three dimensional interactive graphics via the gltools [10]. Additionally, interfaces for post–processing data with the GRAPE visualization environment [25] as well as with the General Mesh Viewer [19] are supplied.

4.11.1 One and two dimensional graphics subroutines

A set of subroutines for opening, closing of graphic output windows, and several display routines are provided, like drawing the underlying mesh, displaying scalar finite element functions as a graph in 1d, and using iso–lines or iso–colors in 2d. For vector valued functions $\boldsymbol{v}$ similar routines are available, which display the modulus $|\boldsymbol{v}|$.

The routines use the following type definitions for window identification, color specification in [red, green, blue] coordinates, with $0 \leq$ red, green, blue ≤ 1, and standard colors

```
typedef void * GRAPH_WINDOW;

typedef float  GRAPH_RGBCOLOR[3];

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;

extern const GRAPH_RGBCOLOR rgb_albert;
extern const GRAPH_RGBCOLOR rgb_alberta;
```

The last two colors correspond to the two different colors in the ALBERTA logo.

The following graphic routines are available for one and two dimensions:

```
GRAPH_WINDOW graph_open_window(const char *, const char *, REAL *, MESH *);
void graph_close_window(GRAPH_WINDOW);
void graph_clear_window(GRAPH_WINDOW, const GRAPH_RGBCOLOR);
void graph_mesh(GRAPH_WINDOW, MESH *, const GRAPH_RGBCOLOR, FLAGS);
void graph_drv(GRAPH_WINDOW, const DOF_REAL_VEC *, REAL, REAL, int);
void graph_drv_d(GRAPH_WINDOW, const DOF_REAL_D_VEC *, REAL, REAL, int);
void graph_el_est(GRAPH_WINDOW, MESH *, REAL (*)(EL *), REAL, REAL);
void graph_line(GRAPH_WINDOW, , const REAL [2], const REAL [2],
                const GRAPH_RGBCOLOR, REAL);
void graph_point(GRAPH_WINDOW, const REAL [2], const GRAPH_RGBCOLOR, float);
void graph_points(GRAPH_WINDOW, int, REAL (*)[2], const GRAPH_RGBCOLOR, float);
```

Description:

graph_open_window(title, geometry, world, mesh) the function returns a pointer to a GRAPH_WINDOW which is opened for display; if the window could not be opened, the return value is NULL; in 1d the y-direction of the graphic window is used for displaying the graphs of functions;

title is an optional string holding a window title, if title is NULL, a default title is used; geometry is an optional string holding the window geometry in X11 format "WxH" or "WxH+X+Y", if NULL, a default geometry is used;

world is an optional pointer to an array of *world coordinates* (xmin, xmax, ymin, ymax) to specify which part of a triangulation is displayed in this window, if world is NULL and mesh is not NULL, mesh->diam is used to select a range of world coordinates; in 1d, the range of the y-direction is set to $[-1, 1]$; if both world and mesh are NULL, the unit square $[0, 1] \times [0, 1]$ is displayed in 1d and 2d.

graph_close_window(win) closes the graphic window win, previously opened by the function graph_open_window().

graph_clear_window(win, c) clears the graphic window win and sets the background color c; if c is NULL, white is used as background color.

graph_mesh(win, mesh, c, flag) displays the underlying mesh in the graphic window win; c is an optional color used for drawing lines, if c is NULL black as a default color is used; the last argument flag allows for a selection of an additional display; flag may be 0 or the bitwise OR of some of the following flags:

GRAPH_MESH_BOUNDARY only boundary edges are drawn, otherwise all edges of the triangulation are drawn; c is the display color for all edges if not NULL; otherwise the display

color for Dirichlet boundary vertices/edges is blue and for Neumann vertices/edges the color is red;

GRAPH_MESH_ELEMENT_MARK triangles marked for refinement are filled red, and triangles marked for coarsening are filled blue, unmarked triangles are filled white;

GRAPH_MESH_VERTEX_DOF the *first* DOF at each vertex is written near the vertex; currently only working in 2d when the library is not using OpenGL.

GRAPH_MESH_ELEMENT_INDEX element indices are written inside the element, only available for `EL_INDEX == 1`; currently only working in 2d when the library is not using OpenGL.

graph_drv(win, u, min, max, n_refine) displays the finite element function stored in the `DOF_REAL_VEC u` in the graphic window `win`; in 1d, the graph of `u` is plotted in black, in 2d an iso-color display of `u` is used; `min` and `max` specify a range of `u` which is displayed; if `min` $\geq$ `max`, `min` and `max` of `u` are computed by `graph_drv()`; in 2d, coloring is adjusted to the values of `min` and `max`; the display routine always uses the linear interpolant on a simplex; if `n_refine` > 0, each simplex is recursively bisected into $2^{\texttt{mesh->dim*n\_refine}}$ sub–simplices, and the linear interpolant on these sub–simplices is displayed; for `n_refine` < 0 the default value `u->admin->bas_fcts->degree-1` is used.

graph_drv_d(win, v, min, max, n_refine) displays the modulus of the vector valued finite element function stored in the `DOF_REAL_D_VEC v` in the graphic window `win`; the other arguments are the same as for `graph_drv()`.

graph_el_est(win, mesh, get_el_est) displays piecewise constant values over the triangulation `mesh`, like local error indicators, in the graphics window `win`; `get_el_est` is a pointer to a function which returns the constant value on each element; by this function the piecewise constant function is defined.

graph_line(win, p0, p1, c, lw) draws the line segment with start point `p0` and end point `p1` in (x, y) coordinates in the graphic window `win`; `c` is an optional argument and may specify the line color to be used; if `c` is `NULL` black is used; `lw` specifies the linewidth (currently only for OpenGL graphics); if `lw` ≤ 0 the default linewidth `1.0` is set.

graph_point(win, p, c, diam) draws a point at the position `p` in (x, y) coordinates in the graphic window `win`; `c` is an optional argument and may specify the color to be used; if `c` is `NULL` black is used; `diam` specifies the drawing diameter (currently only for OpenGL graphics); if `diam` ≤ 0 the default diameter `1.0` is set.

graph_points(win, np, p, c, diam) draws a `np` points at the positions `p` in (x, y) coordinates in the graphic window `win`; `c` is an optional argument and may specify the color to be used; if `c` is `NULL` black is used; `diam` specifies the drawing diameter (currently only for OpenGL graphics); if `diam` ≤ 0 the default diameter `1.0` is set.

4.11.1.1 Graphic routines for two dimensions

The following routines are specialized routines for two dimensional graphic output:

```
void graph_level_2d(GRAPH_WINDOW, const DOF_REAL_VEC *, REAL,
                    const GRAPH_RGBCOLOR, int);
void graph_levels_2d(GRAPH_WINDOW, const DOF_REAL_VEC *, int, const REAL *,
                     const GRAPH_RGBCOLOR *, int);
void graph_level_d_2d(GRAPH_WINDOW, const DOF_REAL_D_VEC *, REAL,
                      const GRAPH_RGBCOLOR, int);
```

```
void graph_levels_d_2d(GRAPH_WINDOW, const DOF_REAL_D_VEC *, int, const REAL *,
                       const GRAPH_RGBCOLOR *, int);
void graph_fvalues_2d(GRAPH_WINDOW, MESH *,
                      REAL (*)(const EL_INFO *, const REAL *),
                      FLAGS, REAL, REAL, int);
```

graph_level_2d(win, v, level, c, n_refine) draws a single selected isoline at value `level` of the scalar finite element function stored in the `DOF_REAL_VEC u` in the graphic window `win`; by the argument `c` a line color for the isoline can be specified; if `c` is `NULL`, black is used as line color; the display routine always uses the linear interpolant of `u` on a simplex; if `n_refine` > 0, each triangle is recursively bisected into $2^{2*\texttt{n\_refine}}$ sub–triangles, and the selected isoline of the linear interpolant on these sub–triangles is displayed; for `n_refine` < 0 the default value `u->admin->bas_fcts->degree-1` is used.

graph_levels_2d(win, u, n, levels, c, n_refine) draws `n` selected isolines at values `level[0]`, ..., `level[n-1]` of the scalar finite element function stored in the `DOF_REAL_VEC u` in the graphic window `win`; if `level` is `NULL`, `n` equally distant isolines between the minimum and maximum of `u` are selected; `c` is an optional vector of `n` color values for the `n` isolines, if `NULL`, then default color values are used; the argument `n_refine` again chooses a level of refinement, where iso-lines of the piecewise linear interpolant is displayed; for `n_refine` < 0 the default value `u->admin->bas_fcts->degree-1` is used.

graph_level_d_2d(win, v, level, c, n_refine) draws a single selected isoline at values `level` of the modulus of a vector valued finite element function stored in the `DOF_REAL_D_VEC v` in the graphic window `win`; the arguments are the same as for `graph_level()`.

graph_levels_d_2d(win, v, n, levels, c, n_refine) draws `n` selected isolines at values `level[0]`, ..., `level[n-1]` of the modulus of a vector valued finite element function stored in the `DOF_REAL_D_VEC v` in the graphic window `win`; the arguments are the same as for `graph_levels()`.

graph_fvalues_2d(win, mesh, f, flag, min, max, n_refine) displays the function `f` in the graphic window `win`; `f` is a pointer to a function for evaluating values on single elements; `f(el_info, lambda)` returns the value of the function on `el_info->el` at the barycentric coordinates `lambda`;

an iso-color display of `f` is used; `min` and `max` specify a range of `f` which is displayed; if `min` $\geq$ `max`, `min` and `max` of `f` are computed by `graph_fvalues_2d()`; coloring is adjusted to the values of `min` and `max`; the display routine always uses the linear interpolant of `f` on a simplex; if `n_refine` > 0, each simplex is recursively bisected into $2^{2*\texttt{n\_refine}}$ sub–simplices, and the linear interpolant on these sub–simplices is displayed.

4.11.2 gltools interface

The following interface for using the interactive gltools graphics of WIAS Berlin [10] is implemented. The gltools are freely available under the terms of the MIT license, see

http://www.wias-berlin.de/software/gltools/

The ALBERTA interface to the gltools is compatible with version `gltools-2-4`. It can be used for 1d, 2d, and 3d triangulation, but only when `mesh->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_vec(GLTOOLS_WINDOW, const DOF_REAL_D_VEC *, REAL, REAL);
void gltools_est(GLTOOLS_WINDOW, MESH *, REAL (*)(EL *), REAL, REAL);

void gltools_disp_mesh(GLTOOLS_WINDOW, MESH *, int, const DOF_REAL_VEC *);
void gltools_disp_drv(GLTOOLS_WINDOW, const DOF_REAL_VEC *, REAL, REAL,
                      const DOF_REAL_VEC *);
void gltools_disp_drv_d(GLTOOLS_WINDOW, const DOF_REAL_D_VEC *, REAL, REAL,
                      const DOF_REAL_VEC *);
void gltools_disp_vec(GLTOOLS_WINDOW, const DOF_REAL_D_VEC *, REAL, REAL,
                      const DOF_REAL_VEC *);
void gltools_disp_est(GLTOOLS_WINDOW, MESH *, REAL (*)(EL *), REAL, REAL,
                      const DOF_REAL_VEC *);
```

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 NULL; title is an optional string holding a title for the window; if title is NULL, a default is used; geometry is an optional string holding the window geometry in X11 format ("WxH" or "WxH+X+Y"), if NULL, 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, it can be used to specify which part of the mesh will be displayed in the window; if world is NULL, either mesh or the default domain $[0,1]^d$ 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 NULL, the default domain $[0,1]^d$ 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; for higher order elements it is possible to display the vector on a refined grid; the key 'P' toggles between refined and not refined mode; 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; for higher order elements it is possible to display the vector on a refined grid; the key 'P' toggles between refined and not refined mode; min and max

define the range of the modulus of discrete function for display; if `min` $\geq$ `max` this range is adjusted automatically;

`gltools_vec(win, ud, min, max)` displays the vector field given by `DOF_REAL_D_VEC ud` in the graphic window `win`; for higher order elements it is possible to display the vector on a refined grid; the key `'P'` toggles between refined and not refined mode; `min` and `max` define the range of the modulus of discrete function for display; if `min` $\geq$ `max` this range is adjusted automatically;

`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()`;

`gltools_disp_mesh(win, mesh, mark, disp)` additionally to `gltools_mesh()`, a distortion of the geometry by a displacement vector field `DOF_REAL_D_VEC disp` is shown; this can be used in solid mechanics applications, e.g.;

`gltools_disp_drv(win, u, min, max, disp)` similar to `gltools_drv()` but displayed on the distorted geometry given by `DOF_REAL_D_VEC disp`;

`gltools_disp_drv_d(win, ud, min, max, disp)` similar to `gltools_drv_d()` but displayed on the distorted geometry given by `DOF_REAL_D_VEC disp`;

`gltools_disp_vec(win, ud, min, max, disp)` similar to the function `gltools_vec()` but displayed on the distorted geometry given by `DOF_REAL_D_VEC disp`;

`gltools_disp_est(win, mesh, get_el_est, min, max, disp)` similar to the function `gltools_est()` but displayed on the distorted geometry given by `DOF_REAL_D_VEC disp`.

4.11.3 GRAPE interface

Visualization using the GRAPE library [25] is only possible as a post–processing step. 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 the GRAPE mesh interface for the visualization. We recommend using the `xdr` routines for portability of the stored binary data. The use of the GRAPE h–mesh and hp–mesh interfaces is work in progress and the description of these programs will be done in the near future. References to visualization methods used in GRAPE applying to ALBERTA can be found in [12, 21, 22].

For obtaining the GRAPE library, please see

http://www.iam.uni-bonn.de/sfb256/grape/

The distribution of ALBERTA contains source files with the implementation of GRAPE mesh interface to ALBERTA in the `add_ons/grape/` subdirectory. Having access to the GRAPE library (Version 5.4.2), this interface can be compiled and linked with the ALBERTA and GRAPE library into the executables `alberta_grape??` and `alberta_movi??`, where the two-digit suffix `??` codes for the mesh-dimension and `DIM_OF_WORLD`. Currently, however, only co-dimension 0 versions in 2d and 3d are available. The path of the GRAPE header file and library has to be specified during the installation of ALBERTA, compare Section 2.5.

The presence of the GRAPE library and header-file is determined at configure time. If it is found, then the four GRAPE-programs are compiled automatically when running `make` in the top-level directory of the ALBERTA distribution and installed below `PREFIX/bin/` running `make install`

The program `alberta_grape??` is mainly designed for displaying finite element data on a single grid, i. e. one or several scalar/vector-valued finite element functions on the corresponding mesh. `alberta_grape??` expects mesh data stored by `write_mesh[_xdr]()` and `write_dof_real[_xdr]()` or `write_dof_real_d[_xdr]()` defined on the same mesh.

```
alberta_grape22 -m mesh.xdr -s scalar.xdr -v vector.xdr
```

will display the 2d mesh stored in the file `mesh.xdr` together with the scalar finite element function stored in `scalar.xdr` and the vector valued finite element function stored in `vector.xdr`.

`alberta_grape?? --help` gives some online-help, including a short example:

```
jane_john_doe@street ~ $
jane_john_doe@street ~ $ alberta_grape33 --help
Usage: alberta_grape33 [-p PATH] [OPTIONS]
          -m  MESH [-s DRV] [-v DRDV] [[-m MESH1] [-s DRV1] [-v DRDV1] ...]

Example:
 alberta_grape33 --mesh=mymesh -s temperature --vector velocity

    where "mymesh", "temperature" and "velocity" are file-names.

If a long option shows an argument as mandatory, then it is mandatory
for the equivalent short option also.  Similarly for optional arguments.

The order of the options _is_ significant in the following cases:
‘-p PATH’ alters the search path for all following data-files.
‘-m MESH’ specifies a new mesh for all following DRVs and DRDVs (see below)

Options:
  -m, --mesh=MESH
            The file-name of an ALBERTA-mesh gnereated by the ALBERTA
            library routines ‘write_mesh()’ or ‘write_mesh_xdr()’
            ‘-x’ and ‘-b’ options below.
            This option is mandatory and may not be omitted. This option
            may be specified multiple times. All following dof-vectors
            given by the ‘-s’ and ‘-v’ options must belong to the most
            recently specified mesh.
  -b, --binary
            Expect MESH, DRV and DRDV to contain data in host dependent
            byte-order, generated by ‘write_SOMETHING()’ routines of the
            ALBERTA library (SOMETHING is ‘mesh’, ‘dof_real_vec’ etc.
  -x, --xdr
            This is the default and just mentioned here for completeness.
            Expect MESH, DRV and DRDV to contain data in network
            byte-order, generated by ‘write_SOMETHING_xdr()’ routines
            of the ALBERTA library. Per convention this means big-endian
            byte-order.
```

```
  -s, --scalar=DRV
            Load the data-file DRV which must contain a DOF_REAL_VEC
            dumped to disk by 'write_dof_real_vec[_xdr]()'.
            This option may be specified multiple times. The DOF_REAL_VECs
            must belong to the most recently specified mesh.
            See '-m' and '-b' above.
  -v, --vector=DRDV
            Load the data-file DRDV which must contain a DOF_REAL_VEC_D
            dumped to disk by 'write_dof_real_d_vec[_xdr]()'.
            This option may be specified multiple times. The vector
            must belong to the most recently specified mesh.
            See '-m' and '-b' above.
  -p, --path=PATH
            Specify a path prefix for all following data-files. This option
            may be specified multiple times. PATH is supposed to be the
            directory containing all data-files specified by the following
            '-m', '-s' and '-v' options.
  -h, --help
            Print this help.
```

The program `alberta_movi??` is designed for displaying finite element data on a sequence of grids with one or several scalar/vector-valued finite element functions. This is the standard visualization tool for post–processing data from time–dependent simulations. `alberta_movi??` expects a sequence of mesh data stored by `write_mesh[_xdr]()` and finite element data of this mesh stored by `write_dof_real[_xdr]()` or `write_dof_real_d[_xdr]()`, where the filenames for the sequence of meshes and finite element functions are generated by the function `generate_filename()`, explained in Section 3.1.6. Section 2.4.10 shows how to write such a sequence of data in a time-dependent problem.

Similar to `alberta_grape??` the command `alberta_movi?? --help` gives some online-help:

```
jane_john_doe@street ~ $
jane_john_doe@street ~ $ alberta_movi33 --help
Usage: alberta_movi33 START END [-p PATH] [OPTIONS]
          -m  MESH [-s DRV] [-v DRDV] [[-s DRV1] [-v DRDV1] ...]

Example:
  alberta_movi33 --mesh=mymesh 0 10 -i 5 -s u_h --vector v_h

  reads grid mesh000000 with scalar function u_h000000 and
  vector function v_h000000, then mesh000005 with u_h000005 and
  v_h000005, and finally mesh000010 with u_h000010 and v_h000010

If a long option shows an argument as mandatory, then it is mandatory
for the equivalent short option also.  Similarly for optional arguments.

The order of the options is not significant with the exception that the
non-option arguments START and END must come _first_.

Non-option arguments:
  START END
            Two integers specifying the start- and end-scene. The actual
```

```
            file names of the data-files are generated by appending a six
            digit number which loops between START and END.
            See also ‘-i’ below.
Options:
  -i, --increment=INC
            INC is an integers specifying the increment while reading in
            the time scenes. To read e.g. only every second time-scene
            use ‘-i 2’. INC defaults to 1
  -m, --mesh=MESH
            The file-name prefix of an ALBERTA-mesh gnereated by the ALBERTA
            library routines ‘write_mesh()’ or ‘write_mesh_xdr()’
            ‘-x’ and ‘-b’ options below. The actual file name is generated
            by appending a six digit time-scene number to MESH, unless
            the ‘-f’ option is also specified, see below.
            This option is mandatory and may not be omitted.
  -f, --fixed-mesh
            Use a single fixed mesh for all time-scenes (i.e. in the
            non-adaptive case). If ‘-f’ is used ‘-m MESH’ gives the actual
            file name of the mesh and not only the mesh-prefix. See ‘-m’
            above.
  -s, --scalar=DRV
            Load the data-files DRVXXXXXX which must contain DOF_REAL_VECs
            dumped to disk by ‘write_dof_real_vec[_xdr]()’.
            ‘XXXXXX’ stands for the time-scene number.
            This option may be specified multiple times. The DOF_REAL_VECs
            must belong to the meshes specified with the ‘-m’ option.
            See ‘-m’, ‘-b’, ‘-p’ and ‘-i’.
  -v, --vector=DRDV
            Load the data-files DRDVXXXXXX which contain DOF_REAL_VEC_Ds
            dumped to disk by ‘write_dof_real_d_vec[_xdr]()’.
            ‘XXXXXX’ stands for the time-scene number.
            This option may be specified multiple times. The vectors
            must belong to the meshes specified with the ‘-m’ option.
            See ‘-m’, ‘-b’, ‘-p’ and ‘-i’.
  -p, --path=PATH
            Specify a path prefix for all data-files. PATH is supposed to
            be the directory containing all data-files specified by the
            ‘-m’, ‘-s’ and ‘-v’ options.
  -B, --Bar
            Generate a time-progress-bar when displaying the data in GRAPE.
  -b, --binary
            Expect MESH, DRV and DRDV to contain data in host dependent
            byte-order, generated by ‘write_SOMETHING()’ routines of the
            ALBERTA library (SOMETHING is ‘mesh’, ‘dof_real_vec’ etc.
  -x, --xdr
            This is the default and just mentioned here for completeness.
            Expect MESH, DRV and DRDV to contain data in network
            byte-order, generated by ‘write_SOMETHING_xdr()’ routines
            of the ALBERTA library. Per convention this means big-endian
            byte-order.
  -h, --help
            Print this help.
```

4.11.4 Paraview interface

The Paraview interface (http://www.paraview.org/) – like the GRAPE-interface – is available as a set of separate programs which can be used to display finite element data in a post-processing step. The corresponding programs do not require any support package and are always compiled when running `make` and installed below `PREFIX/bin/` when running `make install`. The programs are named `alberta2paraview2d` and `alberta2paraview3d`. The calling convention is somewhat similar to the GRAPE support-programs, and running the programs with the `--help` command-line switch displays an online-help, including some simple examples:

```
jane_john_doe@street ~ $
jane_john_doe@street ~ $ alberta2paraview3d --help
Usage: alberta2paraview3d [-t FIRST LAST] [-i STEP] [-p PATH] [-o OUTPUT]
          -m  MESH [-s DRV] [-v DRDV] [[-m MESH1] [-s DRV1] [-v DRDV1] ...]

Example for converting stationary data:
  alberta2paraview3d \
       -r lagrange_degree --mesh mymesh -s temperature --vector velocity
    where "mymesh", "temperature" and "velocity" are file-names.

Example for converting a sequence of files resulting from a transient
problem:
  alberta2paraview3d -t 0 10 -i 5 -p PATH --mesh mymesh -s u_h --vector v_h

  reads grid mymesh000000 with scalar function u_h000000 and
  vector function v_h000000, then mesh000005 with u_h000005 and
  v_h000005, and finally mesh000010 with u_h000010 and v_h000010

If a long option shows an argument as mandatory, then it is mandatory
for the equivalent short option also.  Similarly for optional arguments.

The order of the options _is_ significant in the following cases:
‘-p PATH’ alters the search path for all following data-files.
‘-m MESH’ specifies a new mesh for all following DRVs and DRDVs (see below)
‘-b|-x’   alters the expected data-format for all following files
          (see below)

Options:
  -t, --transient FIRST LAST
            Convert a sequence of mesh- and data-files. The file-names
            must end with 6-digit decimal number. FIRST and LAST specify the
            first and last member of this sequence.
  -i, --interval SKIP
            In conjunction with ‘-t’ use only every SKIP-th frame in the
            given sequence of files.
  -m, --mesh MESH
            The file-name of an ALBERTA-mesh gnereated by the ALBERTA
            library routines ‘write_mesh()’ or ‘write_mesh_xdr()’
            ‘-x’ options below.
            This option is mandatory and may not be omitted. This option
            may be specified multiple times. All following dof-vectors
            given by the ‘-s’ and ‘-v’ options must belong to the most
```

```
          recently specified mesh.
    -a, --ascii
          Write the paraview file in ASCII format.
    -r, --refined LAGRANGE-DEGREE
          Expect Lagrange-degree (between 0 and 4) to refine the given
          MESH
          To select 'no refinement' simply do not specify '--refined',
          'Lagrange-degree = 0' is the default.
    -u, --unperforated
          For a 3d mesh refine without holes (produces a lot more
          elements). To select mesh-refine with holes simply do not
          specify '--unperforated' (refinement with holes is the default).
    -b, --binary
          Write the paraview file in binary format.
          To select ASCII OUTPUT format simply do not specify '--binary',
          because ASCII OUTPUT format is the default.
    -x, --xdr
          This is the default and just mentioned here for completeness.
          Expect MESH, DRV and DRDV to contain data in network
          byte-order, generated by 'write_SOMETHING_xdr()' routines
          of the ALBERTA library. Per convention this means big-endian
          byte-order. '-l' and '-x' may be specified multiple times.

    -l, --legacy
          Expect MESH, DRV and DRDV to contain data in ALBERTA's
          legacy file-format, generated by 'write_SOMETHING()' routines
          of the ALBERTA library. This may not work, because the format
          of those data-files is byte-order dependent and thus not portable
          across different computer architectures. '-l' and '-x' may be
          specified multiple times.
    -s, --scalar DRV
          Load the data-file DRV which must contain a DOF_REAL_VEC
          dumped to disk by 'write_dof_real_vec[_xdr]()'.
          This option may be specified multiple times. The DOF_REAL_VECs
          must belong to the most recently specified mesh.
          See '-m' above.
    -v, --vector DRDV
          Load the data-file DRDV which must contain a DOF_REAL_VEC_D
          dumped to disk by 'write_dof_real_d_vec[_xdr]()'.
          This option may be specified multiple times. The vector
          must belong to the most recently specified mesh.
          See '-m' above.
    -o, --output FILENAME
          Specify an output file-name. If this option is omitted, then the
          output file-name is"alberta".
    -d, --pvd_output FILENAME
          Specify an pvd_output file-name,in conjuncion with'-t'.
           "alberta_paraview_movi"is the default
    -p, --path PATH
          Specify a path prefix for all following data-files. This option
          may be specified multiple times. PATH is supposed to be the
          directory containing all data-files specified by the following
          '-m', '-s' and '-v' options.
```

```
-h, --help
          Print this help.
```

4.11.5 Geomview interface

Geomview (http://www.geomview.org/) is a quite ancient rendering engine, originally developed by the Geometry Center (http://www.geom.uiuc.edu/). It is easy to use, but as such not a visualization tool for finite element data, and mainly aiming at displaying 2-surfaces. Currently, Geomview is the only way to directly visualize 2d and 3d simulations with co-dimension larger than 1, respectively 0. The suite of demo-programs contains a rudimentary interface to Geomview, the use is demonstrated in the demo-programs using parametric meshes, like `src/Common/ellipt-sphere.c`.

4.11.6 GMV interface

A second possibility to visualize ALBERTA meshes and vectors as a post–processing step is to use the General Mesh Viewer (GMV) developed at the Los Alamos National Laboratories. For information on how to obtain this program see

http://www-xdiv.lanl.gov/XCM/gmv/GMVHome.html

GMV is a standalone program and support for the GMV interface in ALBERTA is always built in. At the moment, this interface is the only one which supports all of the following features: embedded meshes (`mesh->dim < DIM_OF_WORLD`), parametric meshes, generation of movie sequences, reusing meshes for several vectors to reduce disk space, and several more. The ALBERTA interface was written for GMV 4.0.

The interface contains the following functions:

```
int write_mesh_gmv(MESH *, const char *, int, int, const int,
                   DOF_REAL_VEC **, const int, DOF_REAL_D_VEC **,
                   DOF_REAL_D_VEC *, REAL)

int write_dof_vec_gmv(MESH *, const char *, const char *, int, int, const int,
                      DOF_REAL_VEC **drv_ptr, const int, DOF_REAL_D_VEC **,
                      DOF_REAL_D_VEC *, REAL);
```

Description:

`write_mesh_gmv(mesh,name,asc,ref,n_drv,drvs,n_drdv,drdvs,vel,time)` Writes an ALBERTA triangulation and DOF vectors into a file `name` readable by the GMV program. The parameter `asc`, if set to `true` directs ALBERTA to write the data in GMV ASCII format, otherwise a native binary format is used. The triangulation is stored in `mesh`. The parameters `n_drv` and `n_drdv` state the number of `DOF_REAL_VEC`s and `DOF_REAL_D_VEC`s to store in the file. These vectors are passed as arrays of pointers `drvs` and `drdvs`. At the moment there is a limit of 250 vectors of either type which may be written at once. The additional argument `vel` is used for one `DOF_REAL_D_VEC` which has the meaning of a velocity field. This results in a special treatment by GMV; GMV will

automatically create a new field storing the velocity magnitudes on reading the file. The argument `time` stores a time value for instationary simulations.

As most other visualization packages, GMV is only able to display linear data. To alleviate this problem, the parameter `ref`, if set to `true`, directs the interface to output a virtually refined triangulation to avoid loss of data when visualizing higher order Lagrange DOF vectors. This only works for Lagrange finite element spaces.

`write_dof_vec_gmv(mesh,mfile,name,asc,ref,n_drv,drvs,n_drdv,drdvs,vel,time)`

This routine works in a similar way as `write_mesh_gmv()`. The only difference is that the mesh triangulation is not output into the file. Instead, ALBERTA generates a GMV file containing a reference to another GMV file `mfile` containing the mesh. The mesh file must have been output previously using `write_mesh_gmv()`. No refinement or coarsening must occur between these calls, otherwise GMV will be unable to use the old mesh.

The advantage of this is that disk space is saved, since there is no need to repeatedly write entire mesh triangulations for instationary simulations without mesh changes. This also saves time on reading the GMV file.

4.12 Contributed "add-ons"

The ALBERTA distributions contains a sub-directory

```
alberta-VERSION/add_ons/
```

with contributed extension and code-fragments. The degree of stability varies between the different packages in the `add_ons/` sub-directory. We give just a very brief description here. Some of the "add-ons" already have been mentioned in the preceeding sections. The stand-alone programs contained in the `add_ons/` directory are compiled during the ordinary compilation cycle for the ALBERTA-distribution, and install below `PREFIX/bin/`, where prefix is the principal installation prefix for the entire package, as specified by the `--prefix`-argument to the `configure`-script.

4.12.1 `add_ons/bamg2alberta/`

Conversion from the output of the `bamg` grid-generator distributed along with the *FreeFem++* toolbox (Christian Haarhaus).

4.12.2 `add_ons/block_solve/`

A `C`-framework implementing block-matrices consisting of ordinary `DOF_MATRIX` structure (Notger Noll). The add-on comes in the shape of a library

```
PREFIX/lib/liboem_block_solve_Xd[_debug].EXTENSION
PREFIX/include/alberta/oem_block_solve.h
```

The basic data structures are a `BLOCK_DOF_VEC` for storing finite element functions, a `BLOCK_DOF_SCHAR_VEV` for storing boundary masks (compare Section 4.7.7.1), and, of course, a `BLOCK_DOF_MATRIX` for storing matrices composed from blocks of `DOF_MATRIX` structures. Finally, there is a `BLOCK_PRECON_TYPE` structure, for a purpose similar to the `PRECON_TYPE` structure described in Section 4.10.34.

The basic support functions implemented in the library are explained further below in the Sections 4.12.5-4.12.11, in particular

- get_block_dof[_schar]_vec() on page 385,
- free_block_dof[_schar]_vec() on page 385
- get_block_dof_matrix() on page 386,
- free_block_dof_matrix() on page 387,
- clear_block_dof_matrix() on page 387,
- oem_block_solve() on page 388,
- init_oem_block_precon() on page 389.

4.12.1 Datatype (BLOCK_DOF[_SCHAR]_VEC).

Definition

```
#define N_OEM_BLOCKS_MAX 10

typedef struct block_dof_vec
{
  const char *name;
  int n_components;

  DOF_REAL_VEC_D *dof_vec[N_OEM_BLOCKS_MAX];

} BLOCK_DOF_VEC;

typedef struct block_dof_schar_vec
{
  const char *name;
  int n_components;

  DOF_SCHAR_VEC *schar_vec[N_OEM_BLOCKS_MAX];

} BLOCK_DOF_SCHAR_VEC;
```

Components

These two structure are quite simple, the meaning of the components are as follows;

name A descriptive name, used for debugging and pretty-printing.

n_components The number of blocks, the restriction `n_components < N_OEM_BLOCKS_MAX` applies, of course.

dof_vec A flat array of at most `N_OEM_BLOCKS_MAX` many DOF_REAL_VEC_D components, the actual number is stored in `n_components`. Analogously for the `schar_vec` component of the `BLOCK_DOF_SCHAR_VEC`. *Note:* Though the data-type is a `DOF_REAL_VEC_D` it is (ab-)used to store also `DOF_REAL_VEC` data, compare the remarks in Section 3.3.2 concerning the `stride` respectively the `reserved` components of a `DOF_REAL_VEC_D` respectively a `DOF_REAL_VEC` structure.

4.12.2 Datatype (BLOCK_DOF_MATRIX).

Definition

```
#define N_OEM_BLOCKS_MAX 10

typedef enum { Full, Empty, Diag, Triag, Symm } MatType;

typedef struct block_dof_matrix
{
  const char       *name;
  int              n_row_components;
  int              n_col_components;

  const FE_SPACE   *row_fe_spaces[N_OEM_BLOCKS_MAX];
  const FE_SPACE   *col_fe_spaces[N_OEM_BLOCKS_MAX];

  MatType          block_type;

  DOF_MATRIX       *dof_mat[N_OEM_BLOCKS_MAX][N_OEM_BLOCKS_MAX];
  MatrixTranspose  transpose[N_OEM_BLOCKS_MAX][N_OEM_BLOCKS_MAX];
} BLOCK_DOF_MATRIX;
```

Components

Slightly more complicated than the BLOCK_DOF_VEC structure, but still straight forward, maybe with the exception of the `block_type` component.

`name` A descriptive name, for pretty-printing an debugging purposes.

`n_row_components`

`n_col_components` The number of row- and column-blocks.

`row_fe_spaces`

`col_fe_spaces` The finite element spaces, for each row and column.

`block_type` An enumeration value, describing the block-structure:

- **`Full`** An ordinary, fully filled block-matrix.
- **`Empty`** The empty, i.e. zero-matrix. This implies that all pointers in the `dof_mat[][]` component (see below) are `NULL`-pointers.
- **`Diag`** A diagonal matrix. Only the diagonal blocks in `dof_mat[][]` are non-`NULL`.
- **`Triag`** An upper triangular matrix. Only the upper-triangular blocks in `dof_mat[][]` are non-`NULL`.
- **`Symm`** A symmetric matrix, it holds `dof_mat[i][j] == dof_mat[j][i]`. The `transpose[][]` component is initialized to `Transpose` by `get_block_dof_matrix()`.

`dof_mat[][]` The data of the matrix. Not all pointers need to be non-`NULL`, see the documentation for `block_type` above.

`transpose[][]` For each component of `dof_mat` a `MatrixTranspose` flag specifying whether the matrix pointed to should operate as transposed matrix.

4.12.3 Datatype (BLOCK_PRECON_TYPE).

Definition

```
#define N_OEM_BLOCKS_MAX 10

typedef struct block_precon_type
{
  /* Block-Precon-Type */
  OEM_PRECON block_type;

  REAL block_omega;       /* for BlkSSORPrecon */
  int  block_n_iter;      /* for BlkSSORPrecon */

  PRECON_TYPE precon_type[N_OEM_BLOCKS_MAX];
} BLOCK_PRECON_TYPE;
```

Description

This is a "parameter-transport" structure understood by `init_oem_block_precon()`, see below Section 4.12.11. Compare also Section 4.10.36.

Components

`block_type` The block-type of the preconditioner. Only `BlkDiagPrecon` and – experimentally – `BlkSSORPrecon` are supported.

`block_omega`

`block_n_iter` The respective parameters when unsing `block_type == BlkSSORPrecon`.

`precon_type` For each row the type of the preconditioner, see Section 4.10.34.

4.12.4 Function (`..._print_block_...()`).

Description

Not all functions implemented in the library are explained in detail below, in particular, we just notice without detailed description that the following routines exist for pretty-printing:

Prototypes

```
void print_block_dof_vec(BLOCK_DOF_VEC *block_vec);
void print_block_dof_matrix(BLOCK_DOF_MATRIX *block_mat);
void print_block_dof_vec_maple(BLOCK_DOF_VEC *block_vec,
                               const char *block_name);
void print_block_dof_matrix_maple(BLOCK_DOF_MATRIX *block_mat,
                                  const char *block_name);
void fprint_block_dof_vec_maple(FILE *fp, BLOCK_DOF_VEC *block_vec,
                                const char *block_name);
void fprint_block_dof_matrix_maple(FILE *fp, BLOCK_DOF_MATRIX *block_mat,
                                   const char *block_name);
void file_print_block_dof_vec_maple(const char *file_name,
                                    const char fopen_options[],
                                    BLOCK_DOF_VEC *block_vec,
                                    const char *block_name);
void file_print_block_dof_matrix_maple(const char *file_name,
                                       const char fopen_options[],
                                       BLOCK_DOF_MATRIX *block_mat,
                                       const char *block_name);
```

4.12.5 Function (get_block_dof[_schar]_vec()).

Prototype

```
BLOCK_DOF_VEC *get_block_dof_vec(const char *name, int n_components,
                                 const FE_SPACE *fe_space, ...);
BLOCK_DOF_SCHAR_VEC *
get_block_dof_schar_vec(const char *name, int n_components,
                        const FE_SPACE *fe_space, ...);
```

Synopsis

```
block_vec = get_block_dof[_schar]_vec(name, n_components,
                                      first_fe_space, ...);
```

Description

Allocate and initialize a new BLOCK_DOF[_SCHAR]_VEC structure. The routine will internally place calls to get_dof_real[_d]_vec[_d]().

Parameters

name A descriptive name, useful for debugging purposes and pretty-printing. The name is duplicated by calling strdup(3).

n_components The number of blocks the vector shall consist of.

first_fe_space The finite element space for the first component.

... In generalk, n_components-1 further finite element spaces. If a NULL-pointer is encountered in the list, then the preceding finite element space will be used for all following components of the block-vector.

Return Value

A pointer to a newly allocated DOF_BLOCK[_SCHAR]_VEC structure, use free_block_dof_vec() to release the associated resources and delete the vector.

Examples

Have a look at the test program

```
alberta-VERSION/add_ons/block_solver/demo/Common/quasi-stokes.c
```

4.12.6 Function (free_block_dof[_schar]_vec()).

Prototype

```
void free_block_dof_vec(BLOCK_DOF_VEC *bvec);
void free_block_dof_schar_vec(BLOCK_DOF_VEC *bvec);
```

Synopsis

```
free_block_dof[_schar]_vec(block_vec);
```

Description

Release a vector previously allocated by a call to `get_block_dof_vec()`.

Parameters

`block_vec` The vector to destroy.

4.12.7 Function (`get_block_dof_matrix()`).

Prototype

```
BLOCK_DOF_MATRIX *get_block_dof_matrix(const char *name,
                                       int n_row_components,
                                       int n_col_components,
                                       MatType block_type,
                                       const FE_SPACE *fe_space, ...);
```

Synopsis

```
block_matrix = get_block_dof_matrix(name, n_row, n_col,
                                    block_type,
                                    first_fe_space, ...);
```

Description

Allocate a new `BLOCK_DOF_MATRIX` structure. Call `free_block_dof_matrix()` to release the associated memory.

Parameters

`name` A descriptive name, useful for debugging purposes and pretty-printing. `name` is duplicating by a call to `strdup(3)`.

`n_row`

`n_col` The number of row- and column-blocks.

`block_type` The block-type, as explained in Section 4.12.2.

`first_fe_space` The finite element spaces defining the blocks. The function expects them to be ordered alternating: first row space, first column space, second row space, second column space. If `n_cols != n_rows`, then the trailing "excess" spaces are specified one after another. If a `NULL`-pointer is encountered, then the preceding finite element space is used for all remaining rows and columns.

Return Value

A pointer to a newly allocated DOF_DOF_BLOCK_MATRIX structure, use free_block_dof_matrix() to release the associated resources and delete the matrix.

Examples

The interested reader is referred to the test program

```
alberta-VERSION/add_ons/block_solver/demo/Common/quasi-stokes.c
```

4.12.8 Function (free_block_dof_matrix()).

Prototype

```
void free_block_dof_matrix(BLOCK_DOF_MATRIX *bmatrix);
```

Synopsis

```
free_block_dof_matrix(block_matrix);
```

Description

Release a matrix previously allocated by a call to get_block_dof_matrix().

Parameters

block_matrix The matrix to destroy.

4.12.9 Function (clear_block_dof_matrix()).

Prototype

```
void clear_block_dof_matrix(BLOCK_DOF_MATRIX *bmatrix);
```

Synopsis

```
clear_block_dof_matrix(block_matrix);
```

Description

Clear the entries of a BLOCK_DOF_MATRIX.

Parameters

block_matrix The matrix to clear to 0.

4.12.10 Function (oem_block_solve()).

Prototype

```
int oem_block_solve(const BLOCK_DOF_MATRIX *A,
                    const BLOCK_DOF_SCHAR_VEC *bound,
                    const BLOCK_DOF_VEC *f, BLOCK_DOF_VEC *u,
                    OEM_SOLVER solver,
                    REAL tol,
                    const PRECON *precon,
                    int restart, int max_iter, int info);
```

Synopsis

```
iterations =
  oem_block_solve(A, bound, f, u, solver,
                  tol, precon, restart, max_iter, info);
```

Description

The reader is referred to oem_solver() for further explanations. oem_solve() and oem_block_solver() differ only in that the latter accepts block-vectors and -matrices, and the former accepts ordinary DOF-vectors and -matrices as arguments.

Parameters

A The system matrix.

bound A flag-vector to mask-out certain DOFs, e.g. to implement Dirichlet boundary conditions.

f The load vector.

u Storage for the solution and initial guess for the iterative solver.

solver Use the respective OEM-solver; see above for the available keywords.

tol Tolerance for the residual; if the norm of the residual is less or equal tol, oem_solve_[s|d|dow]() returns the actual iterate as the approximative solution of the system.

tol A pointer to a structure describing the preconditioner to use, see further below in Section 4.12.11.

restart Only used by gmres: the maximum dimension of the Krylov-space.

max_iter Maximal number of iterations to be performed by the linear solver. This can be compared with the return value – which gives the number of iterations actually performed – to determine whether the solver has achieved its goal.

info This 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.

Return Value

The number of iterations the solver needed until the norm of the residual was below `tol`, or `max_iter` if the solver was not able to reach its goal before the prescribed maximum iteration count was exhausted.

Examples

The interested reader is referred to the test program

```
alberta-VERSION/add_ons/block_solver/demo/Common/quasi-stokes.c
```

4.12.11 Function (`init_oem_block_precon()`).

Prototype

```
const PRECON *
init_oem_block_precon(const BLOCK_DOF_MATRIX *A,
                      const BLOCK_DOF_SCHAR_VEC *bound,
                      int info,
                      const BLOCK_PRECON_TYPE *prec_type);
```

Synopsis

```
precon = init_oem_block_precon(A, bound, info, prec_type);
```

Description

The reader should compare this functions with `init_precon_from_type()` on page 361.

Parameters

`A` The matrix to compute the preconditioner for.

`bound` A flag-vector, masking out specific DOFs, compare the explanations for the `mask` parameter to `oem_solve()`, see Section 4.10.2. `bound` may be `NULL`.

`info` An integer controlling the amount of information printed to the terminal the application is running in (larger values mean more "noise").

`prec_type` A pointer to a structure of type `BLOCK_PRECON_TYPE`, as described in Section 4.12.3 above, describing the preconditioner to generate.

Return Value

A pointer to an initialized `PRECON` structure implementing the preconditioner, see Section 4.10.26.

Examples

The interested reader is referred to the test program

```
alberta-VERSION/add_ons/block_solver/demo/Common/quasi-stokes.c
```

4.12.12 Function ([...]()).

Description

The remaining functions are also implemented, the reader is referred to the Section 3.7.3 for similar functions for ordinary DOF_REAL_VEC structures.

Prototypes

```
void block_dof_copy(const BLOCK_DOF_VEC *x, BLOCK_DOF_VEC *y);
void block_dof_set(REAL stotz, BLOCK_DOF_VEC *bvec);
int copy_from_block_dof_vec(REAL *x, BLOCK_DOF_VEC *bdof);
int copy_to_block_dof_vec(BLOCK_DOF_VEC *bdof, REAL *x);
int block_dof_vec_length(BLOCK_DOF_VEC *bdof);
```

4.12.3 add_ons/geomview/

A stand-alone viewer to convert simulation data as produced by ALBERTA's IO-routines (see Section 3.3.8) to OOGL-format, which is the data format understood by Geomview. See also Section 4.11.5. (Claus-Justus Heine, Carsten Eilks)

4.12.4 add_ons/gmv/

A stand-alone program to convert ALBERTA data-files (see Section 3.3.8) to GMV format, see also Section 4.11.6. The program in the add_ons/ directory is just a wrapper, calling the library functions described in Section 4.11.6 (courtesy to Daniel Köster).

4.12.5 add_ons/grape/

The Grape interface, see also Section 4.11.3 (Alfred Schmidt, Kunibert G. Siebert, Robert Klöfkorn, Claus-Justus Heine and probably others).

4.12.6 add_ons/libalbas/

A basis-function add-on, with the focus on stable discretisations of the Stokes problem (Claus-Justus Heine). The additional basis function sets are on the one hand available through ALBERTA basis-function plugin-mechanism (see Section 3.5.7), and otherwise through the following functions:

```
const BAS_FCTS *bas_fcts_init(int dim, int dow, const char *name);

const BAS_FCTS *get_null_bfcts(unsigned dim);
const BAS_FCTS *get_bubble(unsigned dim, unsigned inter_deg);
const BAS_FCTS *get_wall_bubbles(unsigned dim, unsigned inter_deg);
const BAS_FCTS *get_trace_bubble(unsigned dim, unsigned inter_deg);
const BAS_FCTS *get_raviart_thomas(unsigned dim, unsigned inter_deg);
const BAS_FCTS *get_old_mini_element(unsigned dim);

typedef struct stokes_pair STOKES_PAIR;
struct stokes_pair
{
  const BAS_FCTS *velocity;
```

```
  const BAS_FCTS *pressure;
  /* const BAS_FCTS *slip_stress; */
};
STOKES_PAIR stokes_pair(const char *name, unsigned dim, unsigned degree);
```

We document only `bas_fcts_init()` and `stokes_pair()`, the other functions are self-explanatory after reading the documentation for `bas_fcts_init()` below.

4.12.13 Function (`bas_fcts_init()`).

Prototype

```
const BAS_FCTS *bas_fcts_init(int dim, int dow, const char *name);
```

Synopsis

```
bas_fcts = bas_fcts_init(dim, DIM_OF_WORLD, name);
```

Description

The entry point when using `libalbas` as plugin-module (see Section 3.5.7), but also an ordinary library function which can be called by functions linked against `libalbas`.

Parameters

`dim` The desired dimension of the basis functions.

`dow` This should equal `DIM_OF_WORLD`. As `libalbas` can be used as a plugin which is loaded according to the value of an environment variable (see Section 3.5.7), the parameter `dow` can be used by the library for sanity checks.

`name` In ALBERTA basis functions are identified by a unique name. `bas_fcts_init()` currently implements the following basis function sets:

`"P1+bubble"` An older implementation of the velocity component of the Mini-element. This implementation does *not* use the direct sum framework (see Section 3.7).

`"Bubble[_IX][_Nd]"` A single element bubble b_T,

$$b_T(\lambda) = w(\texttt{dim}) \prod_{i=0}^{\texttt{dim}} \lambda_i,$$

where the scaling factor $w(\texttt{dim})$ is chosen such that the bubble has mean-value 1 on the reference element. The "`_Nd`" suffix is optional, if present, it is compared against the parameter `dim` as sanity check. The "`_IX`" part is optional, too. If present, it specifies the degree of a quadrature rule used for the interpolation operator. The interpolation operator uses the mean-value of the non-interpolated function as value for the single DOF per element. If the bubble-function belongs to a chain of basis functions, then the interpolation operator take the mean value

of the other components of the corresponding direct sum into account, so the resulting interpolant will have the same mean-value as the non-interpolated function on each element. The default interpolation degree is 0 (respectively 1, using the standard 1-point formula).

"WallBubbles[_IX][_Nd]" A DIM_OF_WORLD-valued basis function set which consists the face-bubbles. This basis function set comes with an per-element initializer (see Section 3.11) as it depends on the geometry of the element: the bubbles point in normal direction with respect to the faces of each element. Using a formula:

$$b_i^e = \pm\, w(\texttt{dim}) \Big(\prod_{\substack{j=0,\dots \texttt{dim} \\ j\neq i}} \lambda_j \Big)\, \nu_i,$$

where ν_i denotes the normal to the i-the face of the element. The scaling factor is chosen such that the mean-value over the faces of the reference simplex is 1 for each bubble. The sign is chosen such that the resulting finite element space consists of globally continuous functions. The current implementation does not take curved boundaries into account. The "_IX" and "_Nd" parts are optional. The X denotes the quadrature degree of a quadrature formula used for interpolation. The interpolation operator determines the local DOFs such that the flux of the interpolated function across the boundaries of each element is the same as the flux of the non-interpolated function, up to quadrature errors. The default quadrature degree is again 0 (respectively 1, see above in the explanations for the element bubble).

"TraceBubbles[_IX][_Nd]" This is the trace-space of the face-bubbles (compare with the trace_bas_fcts component in the BAS_FCTS structure, see Section 3.5).

"RaviartThomas" This is the lowest-order Raviart-Thomas element. However, the code is untested and was primarily meant as a sketch.

"...#..." Any string containing a "#" letter is first decomposed into separate tokens, separated by the "#" signs. The individual components are then generated by calls to ALBERTA's get_bas_fcts() routine, and then chained together by calls to chain_bas_fcts(), see Section 3.5.3.

Return Value

A pointer to new BAS_FCTS structure, as requested by the parameter name, or NULL in case that the request could not be serviced.

Examples

The interested reader is referred to the source-code for the stokes_pair() function in alberta-VERSION/add_ons/libalbas/src/basfcts.c

4.12.14 Function (stokes_pair()).

Prototype

```
typedef struct stokes_pair
{
  const BAS_FCTS *velocity;
  const BAS_FCTS *pressure;
  /* const BAS_FCTS *slip_stress; */
} STOKES_PAIR;

STOKES_PAIR stokes_pair(const char *name, unsigned dim, unsigned
    degree);
```

Synopsis

```
stokes_pair_struct = stokes_pair(name, dim, degree);
```

Description

Generate some of the known stable mixed discretizations for the Stokes-problem, the explanations for the parameter `name` below.

Parameters

`name` The name of the Stokes-pair. The function understands the following names:

`"Mini"` Generate the so-called "Mini element": the velocity space consists of the direct sum of a linear Lagrange element and an element bubble, and the pressure space is a linear Lagrange space. The parameter `degree` to `stokes_pair()` controls the quadrature degree for the interpolation operator, see the explanations for `bas_fcts_init()` above in Section 4.12.13.

`"TaylorHood"` The classical Taylor-Hood element. The parameter `degree` controls the degree of the velocity space in this case.

`"BernardiRaugel"` Generate the "Bernardi-Raugel" element: the velocity space is constructed as the direct sum of a linear Lagrange space and the space of face bubbles described in Section 4.12.13 above. The parameter `degree` to `stokes_pair()` controls the quadrature degree for the interpolation operator, see the explanations for `bas_fcts_init()` above in Section 4.12.13.

The pressure space for the "Bernardi-Raugel" element consists of the space of discontinuous, element-wise constant functions.

`CrouzeixRaviart` Generate the quadratic "Crouzeix-Raviart-Mansfield" element: the velocity space consists of the direct sum of a quadratic Lagrange space with an element bubble in 2d, and of a three-component direct sum in 3d, where additionally face bubbles have to be added. The pressure space is piece-wise linear and discontinuous.

The parameter `degree` controls the degree of the quadrature formula used for the interpolation operator, see see the explanations for `bas_fcts_init()` above in Section 4.12.13.

`dim` The (mesh-)dimension of the requested set of basis functions.

degree As explained above, the meaning of this parameter changes, depending on which ; Stokes-pair is requested.

Return Value

An instance of a STOKES_PAIR structure. Note that this is not a pointer, but a real instance of that structure.

4.12.7 add_ons/meshtv/

A stand-alone program to convert ALBERTA data-files (see Section 3.3.8) to SILO/MeshTV format. (Daniel Köster).

4.12.8 add_ons/paraview/

A stand-alone program to convert ALBERTA data-files (see Section 3.3.8) to Paraview format, see Section 4.11.4. (Rebecca Stotz).

4.12.9 add_ons/static_condensation/

Static-condensation for Stokes-discretizations with the Mini-element, thus reducing the dimension of the velocity space by #elements × DIM_OF_WORLD. (Rebecca Stotz)

There are versions for the "gradient" formulation as well as for the deformation-tensor formulation, including some timings, comparing time needed to solve the condensed equations (with the block-solve add-on, see Section 4.12.2 above and the SYMMLQ solver) against the time needed to solve the uncondensed equations with the CG-method for Schur's complement.

The add-on comes in the shape of a library

```
PREFIX/lib/libstatic_condensation_Xd[_debug].EXTENSION
PREFIX/include/alberta/static-condensation.h
```

The library defines the following functions:

4.12.15 Function (condense_mini_spp[_dd]()).

Prototype

```
void condense_mini_spp(const DOF_REAL_VEC_D *u_h,
                       const DOF_REAL_VEC_D *f_h,
                       const DOF_REAL_VEC *g_h,
                       BNDRY_FLAGS dirichlet_mask,
                       EL_MATRIX_INFO *A_minfo,
                       EL_MATRIX_INFO *B_minfo,
                       BLOCK_DOF_MATRIX *system_matrix,
                       BLOCK_DOF_VEC *up_h,
                       BLOCK_DOF_VEC *load_vector)

void condense_mini_spp_dd(const DOF_REAL_VEC_D *u_h,
                          const DOF_REAL_VEC_D *f_h,
                          const DOF_REAL_VEC *g_h,
                          BNDRY_FLAGS dirichlet_mask,
                          EL_MATRIX_INFO *A_minfo,
                          EL_MATRIX_INFO *B_minfo,
```

```
                                BLOCK_DOF_MATRIX *system_matrix,
                                BLOCK_DOF_VEC *up_h,
                                BLOCK_DOF_VEC *load_vector)
```

Synopsis

```
condense_mini_spp(u_h, f_h, g_h, dirichlet_mask,
                  A_minfo, B_minfo,
                  system_matrix, up_h, load_vector);

condense_mini_spp_dd(u_h, f_h, g_h, dirichlet_mask,
                     A_minfo, B_minfo,
                     system_matrix, up_h, load_vector);
```

Description

We consider the saddle point problem, with Lagrange and bubble basis-functions:

$$\begin{bmatrix} A_{11} & A_{12} & B_1 \\ A_{21} & A_{22} & B_2 \\ B_1^t & B_2^t & 0 \end{bmatrix} \cdot \begin{bmatrix} u_{h,1} \\ u_{h,2} \\ p_h \end{bmatrix} = \begin{bmatrix} f_{h,1} \\ f_{h,2} \\ g_h \end{bmatrix},$$

where, e.g. $u_{h,1}$ and $f_{h,1}$ are the Lagrange-components of the velocity field and the load-vector for the velocity and $u_{h,2}$ and $f_{h,2}$ are the bubble-components. This problem will be converted into an new system:

$$\begin{bmatrix} A_{single} & B_{single} \\ B_{single}^t & C_{single} \end{bmatrix} \cdot \begin{bmatrix} u_{h,single} \\ p_{h,single} \end{bmatrix} = \begin{bmatrix} f_{h,single} \\ g_{h,single} \end{bmatrix},$$

which is equivalent to

$$\texttt{system\_matrix} \cdot \texttt{up\_h} = \texttt{load\_vector}.$$

The function `condense_mini_spp()` converts the saddle point problem as follows

$$\begin{aligned}
u_{h,single} &= u_1 = \texttt{up_h->dof_vec[0]}, \\
p_{h,single} &= p = \texttt{up_h->dof_vec[1]}, \\
f_{h,single} &= f_{h,1} - A_{12}\ A_{22}^{-1}\ f_{h,2} = \texttt{load_vector->dof_vec[0]}, \\
g_{h,single} &= g - B_2^t\ A_{22}^{-1}\ f_2 = \texttt{load_vector->dof_vec[1]}, \\
A_{single} &= A_{11} - A_{12}\ A_{22}^{-1}\ A_{21} = \texttt{system_matrix->dof_mat[0][0]}, \\
B_{single} &= B_1 - A_{12}\ A_{22}^{-1}\ B_2 = \texttt{system_matrix->dof_mat[0][1]}, \\
B_{single}^t &= (B_{single})^{tr} = \texttt{system_matrix->dof_mat[1][0]}, \\
C_{single} &= -B_2^t\ A_{22}^{-1}\ B_2 = \texttt{system_matrix->dof_mat[1][1]}.
\end{aligned} \tag{4.7}$$

Parameters

u_h Storage of the principal unknown, and start-value for an iterative solver. In the context of Dirichlet boundary conditions (see Section 4.7.7.1) the application has to make sure that u_h already incorporates (interpolated) Dirichlet boundary conditions.

f_h Load-vector for the principal equations.

g_h Load-vector for the constraint equation.

dirichlet_mask A bit-mask describing which parts of the boundary should be treated as Dirichlet-boundary, see Section 4.7.7.1. *Note:* dirichlet_mask must not be NULL.

A_minfo Element matrix information to assemble matrix $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$. The static condensation only works if A_{22} is diagonal, which is the case for the bubble basis-functions because they "live" only on one element.

B_minfo Element matrix information to assemble the matrix $B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$.

system_matrix Storage for the new matrices of the condensed system. It is a pointer to a BLOCK_DOF_MATRIX structure, in which the matrices A_{single}, B_{single}, B^t_{single} and C_{single} are stored, as shown in equation (4.7).

up_h Storage for the condensed solution. It is a pointer to a BLOCK_DOF_VEC structure, up_h->dof_vec[0] is the storage for the lagrange components of the velocity and up_h->dof_vec[1] is the storage for the pressure.

load_vector Load-vector of the condensed system, as shown in (4.7).

Examples

In subdirectory static_condensation/demo, there are two demo programs, mini-stokes.c and mini-quasi-stokes.c as an example how to use the functions condense_mini_spp(), condense_mini_spp_dd() and expand_mini_spp(), expand_mini_spp_dd().

4.12.16 Function (expand_mini_spp[_dd]()).

Prototype

```
void expand_mini_spp(const BLOCK_DOF_VEC *up_h,
                     const DOF_REAL_VEC_D *f_h,
                     DOF_REAL_VEC_D *uh,
                     BNDRY_FLAGS dirichlet_mask,
                     EL_MATRIX_INFO *A_minfo,
                     EL_MATRIX_INFO *B_minfo)

void expand_mini_spp_dd(const BLOCK_DOF_VEC *up_h,
                        const DOF_REAL_VEC_D *f_h,
                        DOF_REAL_VEC_D *uh,
                        BNDRY_FLAGS dirichlet_mask,
                        EL_MATRIX_INFO *A_minfo,
                        EL_MATRIX_INFO *B_minfo)
```

Synopsis

```
expand_mini_spp(up_h, f_h, uh, dirichlet_mask,
                A_minfo, B_minfo);

expand_mini_spp_dd(up_h, f_h, uh, dirichlet_mask,
                   A_minfo, B_minfo);
```

Description

The functions `expand_mini_spp()` and `expand_mini_spp_dd()` reconstruct the bubble-components which were eliminated by the function `condense_mini_spp()` or `condense_mini_spp_dd()` or (see Section 4.12.15). The functions recompose the Lagrange-components and the bubble-components and store them in `uh`.

The bubble-component of u is reconstructed as follows

$$u_{h,2} = A_{22}^{-1} \, (f_{h,2} - A_{21} \, u_{h,1} - B_2 \, p_h).$$

Note that A_{22} is a diagonal matrix, so this operation is comparatively cheap.

Parameters

`up_h` The principal unknown, after solving the condensed system.

`f_h` Load-vector for the principal equations.

`uh` Storage for the recomposed solution of the principal equations.

`dirichlet_mask` A bit-mask describing which parts of the boundary should be treated as Dirichlet-boundary. *Note:* `dirichlet_mask` must not be `NULL`.

`A_minfo` Element matrix information for assembling the matrix $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$. The static condensation only works if A_{22} is diagonal, which is the case for the bubble basis-functions because they "live" only on one element.

`B_minfo` Element matrix information to assemble the matrix $B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$.

Examples

In subdirectory `static_condensation/demo`, there are two demo programs, `mini-stokes.c` and `mini-quasi-stokes.c` as an example how to use the functions `condense_mini_spp()`, `condense_mini_spp_dd()` and `expand_mini_spp()`, `expand_mini_spp_dd()`. One for

4.12.10 `add_ons/triangle2alberta/`

A converter from the mesh-generator *Triangle* to ALBERTA macro-file format (Daniel Köster).

4.12.11 `add_ons/write_mesh_fig/`

Contains a function to dump an ALBERTA-mesh in the `fig`-file-format as understood by the `xfig` CAD-tool. Daniel Köster).

Bibliography

[1] R. E. Bank and D. J. Rose, *Global approximate Newton methods.*, Numer. Math., 37 (1981), pp. 279–295.

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

[3] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*, SIAM, Philadelphia, PA, 1994.

[4] J. H. Bramble, J. E. Pasciak, and J. Xu, *Parallel multilevel preconditioners*, Math. Comput., 55 (1990), pp. 1–22.

[5] R. Cools and P. Rabinowitz, *Monomial cubature rules since "Stroud": a compilation*, J. Comput. Appl. Math., 48 (1993), pp. 309–326.

[6] 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.

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

[8] D. Dunavant, *High degree efficient symmetrical Gaussian quadrature rules for the triangle*, Int. J. Numer. Methods Eng., 21 (1985), pp. 1129–114.

[9] G. Dziuk, *An algorithm for evolutionary surfaces*, Numer. Math., 58 (1991), pp. 603 – 611.

[10] J. Fuhrmann and H. Langmach, *gltools: OpenGL based online visualization.* Software: `http://www.wias-berlin.de/software/gltools/`.

[11] K. Gatermann, *The construction of symmetric cubature formulas for the square and the triangle*, Computing, 40 (1988), pp. 229–240.

[12] B. Haasdonk, M. Ohlberger, M. Rumpf, A. Schmidt, and K. G. Siebert, *Multiresolution visualization of higher order adaptive finite element simulations*, Computing, 70 (2003), pp. 181–204.

[13] H. Kardestuncer, ed., *Finite Element Handbook*, McGraw-Hill, New York, 1987.

[14] 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.

[15] M. Lenoir, *Optimal isoparametric finite elements and error estimates for domains involving curved boundaries*, SIAM J. Numer. Anal., 23 (1986), pp. 562–580.

[16] J. G. Lewis, *Algorithm 582: The gibbs-poole-stockmeyer and gibbs-king algorithms for reordering sparse matrices*, ACM Trans. Math. Softw., 8 (1982), pp. 190–194.

[17] A. Meister, *Numerik linearer Gleichungssysteme*, Vieweg, 1999.

[18] 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.

[19] F. A. Ortega, *GMV Version 4.0 – General Mesh Viewer User's Manual*, Los Alamos National Laboratory.

[20] C. C. Paige and M. A. Saunders, *Solutions of sparse indefinite systems of linear equations*, SIAM J. Numer. Anal., 12 (1975), pp. 617–629.

[21] M. Rumpf, A. Schmidt, and K. G. Siebert, *On a unified visualization approach for data from advanced numerical methods*, in Visualization in Scientific Computing '95, R. Scateni, J. V. Wijk, and P. Zanarini, eds., Springer, 1995, pp. 35–44.

[22] ——, *Functions defining arbitrary meshes – a flexible interface between numerical data and visualization routines*, Computer Graphics Forum, 15 (1996), pp. 129–141.

[23] Y. Saad, *Iterative Methods for Sparse Linear Systems*, PWS, 1996.

[24] A. Schmidt and K. G. Siebert, *Design of adaptive finite element software. The finite element toolbox ALBERTA. With CD-ROM.*, Lecture Notes in Computational Science and Engineering 42. Berlin: Springer. xii, 315 p. EUR 64.15 , 2005.

[25] SFB 256, *GRAPE – GRAphics Programming Environment Manual, Version 5.0*, Bonn, 1995.

[26] A. H. Stroud, *Approximate calculation of multiple integrals*, Prentice-Hall, Englewood Cliffs, NJ, 1971.

[27] R. Verfürth, *A posteriori error estimates for nonlinear problems: Finite element discretization of elliptic equations*, Math. Comp., 62 (1994), pp. 445–475.

[28] H. Yserentant, *On the multi-level splitting of finite element spaces*, Numer. Math., 49 (1986), pp. 379–412.

Index

Data types, symbolic constants, functions, and macros

List of data types

List of symbolic constants

List of functions

List of macros