deal.II — a General Purpose Object Oriented Finite Element Library

W. BANGERTH
Texas A&M University
and
R. HARTMANN
German Aerospace Center (DLR)
and
G. KANSCHAT
Universität Heidelberg

An overview of the software design and data abstraction decisions chosen for deal.II, a general purpose finite element library written in C++, is given. The library uses advanced object-oriented and data encapsulation techniques to break finite element implementations into smaller blocks that can be arranged to fit users requirements. Through this approach, deal.II supports a large number of different applications covering a wide range of scientific areas, programming methodologies, and application-specific algorithms, without imposing a rigid framework into which they have to fit. A judicious use of programming techniques allows to avoid the computational costs frequently associated with abstract object-oriented class libraries.

The paper presents a detailed description of the abstractions chosen for defining geometric information of meshes and the handling of degrees of freedom associated with finite element spaces, as well as of linear algebra, input/output capabilities and of interfaces to other software, such as visualization tools. Finally, some results obtained with applications built atop deal.II are shown to demonstrate the powerful capabilities of this toolbox.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: Finite element software—*mesh handling*; *linear algebra*; G.1.8 [**Numerical Analysis**]: Partial Differential Equations—*finite element method.*

General Terms: Algorithms, Design, Documentation

Additional Key Words and Phrases: object-orientation, software design

1. INTRODUCTION

The development of libraries providing finite element infrastructure to application programs has a long tradition reaching back several decades (see, for example, [Rheinboldt

Author's addresses: W. Bangerth, Dept. of Mathematics, Texas A&M University, College Station, TX 77843, USA; R. Hartmann, Institute of Aerodynamics and Flow Technology, DLR, 38108 Braunschweig, Germany; G. Kanschat, Inst. f. Angew. Mathematik, Universität Heidelberg, 69120 Heidelberg, Germany.
Parts of this research have been supported by a graduate fellowship by the German Science Foundation through the Graduiertenkolleg “Modellierung und Wissenschaftliches Rechnen in Mathematik und Naturwissenschaften” at the IWR, Universität Heidelberg, and a postdoctoral fellowship at the Institute for Computational Engineering and Sciences (ICES), University of Texas at Austin.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0098-3500/20YY/1200-0001 $5.00

and Mesztenyi 1980; Bank 1998]), and can be witnessed by the plethora of such libraries that can be found by a search of the world wide web. However, the design of only very few of these libraries turns out to be sufficiently general to support more than the one or two applications it was written for initially, and even fewer of the available libraries are structured and documented well enough to allow their application beyond the group of its developers. Consequently, the design of successful and widely applicable finite element libraries appears to be a hard problem. In this paper, we will examine some of the abstractions and design decisions used in deal.II, an Open Source finite element library that has found significant use with scientists not affiliated with its developers, and that has enjoyed continuous growth since its inception in 1997. It can thus be considered one of the libraries that seem to have survived the test of time.

deal.II is an object-oriented class library providing all tools needed for simulations with the finite element method. Using the structuring means of C++, the different objects used in such a simulation program are well separated, allowing for a wide variation of applications without sacrificing program structure or increasing susceptibility to programming errors. In particular, the strict separation of meshes, finite element spaces and linear algebra classes allows for a very modular approach in programming applications built on deal.II, and to combine the provided functionality in many different ways, suiting the particular needs of an application.

The development of deal.II (and its predecessor DEAL) was initiated by the need for a software tool for research in novel adaptive and high-performance finite element schemes of a number of areas. Most available software tools would either be tuned to performance, but be specialized to one class of applications, while others offer flexibility and generality at a significant waste of memory and computing power. In addition, very few packages were publicly available at the time. Therefore, DEAL filled a gap when its design began in 1992 and was the software basis of the development of goal oriented error estimation and adaptive methods for partial differential equations [Becker and Rannacher 1998; Kanschat 1996; Suttmeier 1996]. However, by 1997, it became clear that some concepts of DEAL had become too cumbersome, and that important improvements could be made by building on the then recent developments in the C++ programming language [Stroustrup 1997] and compilers for it, in particular mature support for templates and the standard template library [Stepanov and Lee 1995; Plauger et al. 2000]. Therefore, a fresh start was made with the new implementation of deal.II.

deal.II has been used in a large number of projects since then, first in mathematical research on error estimation [Bangerth and Rannacher 2003] and discontinuous Galerkin methods [Hartmann 2002; Kanschat 2006, and references found there], then in other areas of academic research, applied sciences, and industrial projects, some of which are mentioned in Section 7. It is available for download from `http://www.dealii.org/` under an Open Source license since early 2000 and has gathered some popularity since then. This web site also features an extensive online documentation, encompassing a comprehensive reference manual as well as a tutorial for beginners.

The library is decisively not designed to solve certain applications. Rather, it is intended to serve as a toolbox containing whatever is needed to write a finite element application program. The main tasks of a finite element program (even for complex nonlinear problems) usually are:

(1) Setting up a mesh and the finite element space,

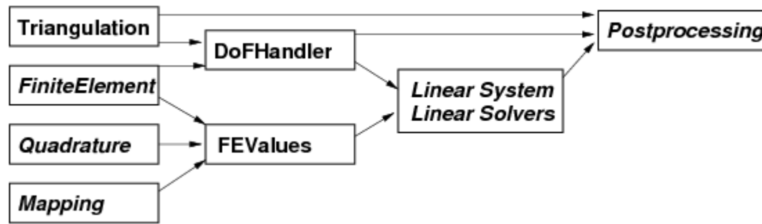

Fig. 1. An overview of the collaboration between the most important classes in deal.II. Abstract base classes and groups of classes are shown in italics. The discussion in this article roughly follows this figure from left to right.

(2) Assembling a discrete linear or linearized system of equations,
(3) Solving this linear system, and
(4) Postprocessing by numerical evaluation of the solution or visualization.

All of these tasks are supported by deal.II, either directly or (for instance in the case of visualization) by interfaces to third party software. To achieve this goal, the library strives to provide a large number of classes and tools that are typically needed for and found in finite element software. While they are designed to cooperate, they do not enforce a particular model or framework for their use, but rather allow programmers to use them in ways appropriate for the structure of a code as dictated by its application area, solution scheme, and software design strategy.

Given the size of the library of about 300,000 lines of code this article is not intended to be a comprehensive description of the library, but rather a description of the software design considerations underlying deal.II, the data abstractions influencing our choice of separating functionality into object-oriented classes, and how they interact. We will demonstrate the use of a few of the objects within the library in small code fragments that show the way of programming with deal.II.

The remainder of this article is structured roughly following the collaboration diagram shown in Fig. 1: Section 2 is devoted to the central building blocks of every finite element code, in particular meshes (represented by the `Triangulation` class), shape function spaces (class `FiniteElement`), and degrees of freedom (class `DoFHandler`). Section 3 discusses how these basic classes are used to build the linear systems arising from the discretization of partial differential equations (classes `Quadrature`, `Mapping`, and `FEValues`). The structures used for efficient linear algebra are described in Section 4. Section 5 lists the connections deal.II offers to other software packages, in particular pre- and postprocessing options available with deal.II. We give an overview over the available documentation in Section 6. Section 7 contains a brief description of some of the many applications built on the library. We conclude with Section 8.

2. MESHES, FINITE ELEMENT CLASSES, AND DEGREES OF FREEDOM

Since deal.II was designed to offer as much flexibility as possible with reasonable effort, the support of finite elements is split over several classes that can be combined in different ways. In particular, the requirement of using nearly arbitrary finite element types of variable order made it necessary to separate mesh objects from element related data. In this section, we discuss which parts of the library organize the geometric and topological structure of the finite element space as well as the auxiliary classes combining these properties

```
Triangulation<dim> triangulation;
GridGenerator::hyper_cube(triangulation);
triangulation.refine(2);
```

Listing 1. Creating a mesh of arbitrary dimension.

again.

2.1 Mesh classes and iterators

The central class in all finite element codes describes the mesh. For historical reasons, in deal.II this class is called `Triangulation`, although the cells it provides are line segments (1D), quadrilaterals (2D), and hexahedra (3D). The triangulation class, as almost the entire library, is organized such that the space dimension is chosen by a template parameter [Bangerth 2000]. Consequently, it is possible to write codes independent of the space dimension, enabling users to develop and test a program in 2D, and later run it in 3D without changes. An example of the creation of a `Triangulation` object is shown in Listing 1. Here, the dimension is deliberately left as a template parameter of the enclosing function. A hyper cube $[0,1]^{\text{dim}}$ (i.e., the unit line, unit square, or unit cube, depending on the value of the template parameter `dim`) is generated and refined uniformly twice into 4^{dim} mesh cells. Note that the dimension is a compile-time constant. Thus, the compiler can optimize on it, and there will be no run-time checks on the dimension impeding performance (unlike in many other libraries such as DiffPack [Langtangen 2003], libmesh [Kirk et al. 2006], or OOFEM [Patzák and Bittnar 2001], where the space dimension is a run-time parameter).

One of the original goals in the development of deal.II was support for adaptive meshing, including both mesh refinement and coarsening. For this purpose, triangulation objects do not only store the respective finest cells, but also their (now inactive) ancestors. Consequently, a triangulation can be thought of as a collection of trees, where the cells of the coarsest mesh form the roots and children branch off their parent cells. The present version of deal.II supports regular (bisection) refinement of cells, leading to 2, 4, or 8 children per cell in 1D, 2D, and 3D, respectively. The cells of a triangulation therefore form binary trees, quad-trees, or oct-trees, respectively [Rheinboldt and Mesztenyi 1980], where the terminal nodes correspond to the active cells, that is, the cells without children forming the mesh in the usual meaning. The non-terminal nodes correspond to inactive cells, i.e. to cells that have children and belong to the mesh hierarchy, but not to the mesh itself. A simple example of an adaptively refined 2D mesh along with its tree of cells and a more complex 3D mesh is shown in Figure 2. The individual levels of these cell trees also form a natural hierarchy for multilevel methods with their obvious importance to the solution of partial differential equations; this feature is exploited in other parts of the library.

In order to create such a triangulation we start with a coarse mesh, that can either be generated by deal.II itself in simple cases, or can be read from a file in several mesh file formats (see Section 5). The cells of this mesh are then refined recursively, either globally or individually. In typical finite element simulations, local refinement is controlled by error estimators, yielding a powerful tool for efficient simulations (see [Bangerth and Rannacher 2003]).

Since unstructured meshes do not have a natural, sequential numbering, mesh cells in

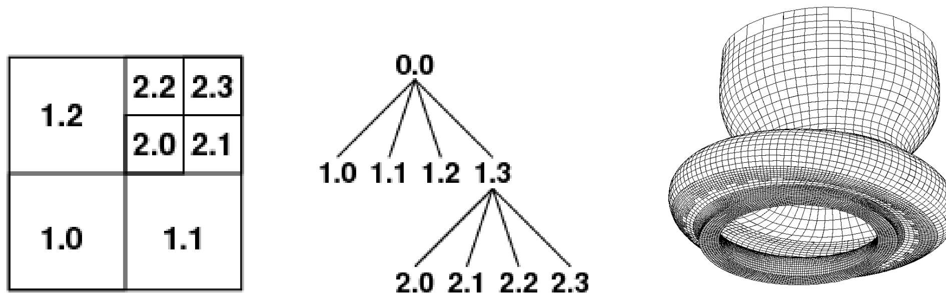

Fig. 2. A simple two-dimensional mesh with cells numbered based on their refinement level and index within a level (left) and the corresponding quad-tree of cells (middle). A complex three-dimensional mesh (right).

```
double average_mesh_size = 0;
for (Triangulation<dim>::active_cell_iterator
       cell = triangulation.begin_active();
     cell != triangulation.end(); ++cell)
  average_mesh_size += cell->diameter();
average_mesh_size /= triangulation.n_active_cells();
```

Listing 2. A loop over all active mesh cells to compute a "mean" mesh size (follow-up to Listing 1).

deal.II are accessed by iterators instead of indices. To this end, the `Triangulation` class provides STL-like iterators that "point" to objects describing cells, faces, edges, and other objects of a mesh. Separate iterator types indicate operations on all objects, only the active subset, and objects satisfying other additional properties. Using these concepts, a loop over all active cells of a triangulation can be written as in Listing 2.

As a final remark, we note that triangulations in deal.II only store geometric (the location of vertices) and topological information (how vertices are connected to edges and cells, and neighborship relations between cells). On the other hand, `Triangulation` objects do not know about finite elements, nodal data, linear systems, etc. It is therefore possible to use the same triangulation object for a number of purposes at the same time.

2.2 Finite element descriptions

In the mathematical literature, finite element spaces are usually described by a tuple $(\hat{\mathcal{P}}, \hat{\mathcal{N}})$ of a shape function space $\hat{\mathcal{P}}$ and a set of node functionals $\hat{\mathcal{N}}$ on the reference cell $\hat{K}$ [Ciarlet 1978; Brezzi and Fortin 1991; Brenner and Scott 2002]. In deal.II, the reference cell is always the hypercube $[0,1]^d$. The node functionals form a basis for the dual space $\hat{\mathcal{P}}^*$. Together with a mapping F_K from the reference cell $\hat{K}$ to the actual mesh cell K and rules for the mapping of $\hat{\mathcal{P}}$ and $\hat{\mathcal{N}}$ to $\mathcal{P}$ and $\mathcal{N}$ on K, these completely describe a parametric definition of a finite element space. Alternatively, in a non-parametric description, the spaces $\mathcal{P}$ and $\mathcal{N}$ may be defined on the mesh cell K directly in special cases.

The class structure of deal.II closely follows the definition of parametric elements. We will discuss the definition of polynomial spaces and nodal functionals in the following two subsections. Due to their close relationship, both these concepts are implemented by

classes derived from the `FiniteElement` class. We defer the discussion of mappings to Section 3.2.

2.2.1 *Finite element spaces.* At present, the following types of finite element spaces are supported by providing shape functions that span the space:

—Tensor product elements $\hat{\mathcal{P}} = \mathbb{Q}_k = \text{span}\,\{\hat{\mathbf{x}}^\alpha : 0 \le \alpha_i \le k,\ 0 \le i \le d\}$ of arbitrary polynomial degree k. In $\mathbb{R}^d$, these are polynomials of the form

$$\hat{\varphi}_j(\hat{x}_1, \dots, \hat{x}_d) = p_{j_1}(\hat{x}_1) p_{j_2}(\hat{x}_2) \cdots p_{j_d}(\hat{x}_d),$$

where the one-dimensional polynomials p_i are of maximal degree k. Both hierarchical polynomials or Lagrange interpolation polynomials for arbitrary sets of support points may be used for the one-dimensional basis. The representation above already suggests an efficient implementation in arbitrary dimensions and up to arbitrary order, without having to actually write down the form of d-dimensional shape functions.
By choosing the node functionals $\hat{\mathcal{N}}$ appropriately (see the discussion of Fig. 3 below), these elements exist both in a globally continuous form ("continuous Galerkin" or "CG elements") as well as a variant that is entirely discontinuous across cell faces ("discontinuous" or "DG elements").

—Elements $\hat{\mathcal{P}} = \mathbb{P}_k = \text{span}\,\{\hat{\mathbf{x}}^\alpha : 0 \le |\alpha| \le k\}$ of polynomial degree k. $\mathbb{P}_k$ may be represented by a monomial basis or by a Legendre basis which allows for diagonal mass matrices on parallelograms. Since this space cannot be guaranteed to be globally continuous on quadrilateral/hexahedral meshes, these elements exist only in an entirely discontinuous variant.

—The vector-valued Raviart/Thomas element $\hat{\mathcal{P}} = RT_k$ of arbitrary degree k, suitable for the mixed discretization of the Laplacian and some flow problems. This element, while not continuous at faces between mesh cells, has continuous normal components. Consequently, its divergence exists everywhere and the element is conforming with the space $H(\Omega, \text{div})$.

—The $H(\Omega, \text{curl})$-conforming, vector-valued element by Nedelec with continuous tangential components, widely used in the discretization of Maxwell's equations [Castillo et al. 2005].

—Elements where individual vector components are composed from simpler elements. For example, in the discretization of the mixed Laplacian, a common choice is the combination $\hat{\mathcal{P}} = RT_k \times DG\mathbb{Q}_k$ of Raviart-Thomas and discontinuous tensor-product elements. deal.II provides a class that is able to construct this $d+1$-dimensional element from its two building blocks. Similarly, the space $\hat{\mathcal{P}} = [\mathbb{Q}_k]^d$ of d-dimensional H^1-conforming shape functions appropriate for second order equations and systems is composed of d copies of a scalar $\mathbb{Q}_k$ element.

The part of the finite element interface pertaining to shape functions spanning the finite dimensional spaces on the reference cell is shown in Listing 3. The first two functions provide the values and gradients of shape function i at a point p on the reference cell (represented by a `dim`-dimensional point object). Similar functions exist for higher derivatives as well as for vector valued finite elements. Note that the gradient is a vector of fixed dimension `dim`, and is represented as a `dim`-dimensional tensor of rank 1 (the matrix of second derivatives would correspondingly be represented by the data type `Tensor<2,dim>`).

```
template <int dim>
class FiniteElement {
  public:
    virtual double shape_value(const unsigned int  i,
                               const Point<dim>   &p) const =0;
    virtual Tensor<1,dim> shape_grad(const unsigned int i,
                                     const Point<dim> &p) const =0;
    // ...
    virtual void fill_fe_values(InternalData &, ...) const =0;
};
```

Listing 3. The most important parts of the interface of the finite element base class as pertaining to the description of shape functions on the reference cell.

Derived classes implementing concrete finite element spaces have to override the abstract interface laid out above. It is clear that the use of virtual functions of such fine granularity is very expensive. Consequently, the library strives to use the more powerful function `fill_fe_values` provided by finite element classes that provides data for the `FEValues` objects discussed in Section 3.3 below and computes shape function values, derivatives, etc. at a number of quadrature points at once. This sort of data is transferred between the various involved classes in an object of type `InternalData`.

It is the task of the implementation of a new finite element to design `fill_fe_values` as efficiently as possible. Typical optimizations applied here are the use of Horner's scheme to compute values and derivatives of polynomials at once. Further improvements rely on the special structure of the element. The function values of tensor product elements for instance can be computed very efficiently all at once, if the values of the one-dimensional polynomials are evaluated for each space direction and then these values are multiplied.

2.2.2 *Node functionals.* Node functionals are used in two ways in the description of finite elements. The first is in order to define a basis of the finite dimensional space: Basis functions $\hat{\varphi}_i$ are chosen such that $\hat{N}_j(\hat{\varphi}_i) = \delta_{ij}$ for all $\hat{N}_j \in \hat{\mathcal{N}}$; note that this relation holds for the basis of the shape function space on each cell as well as for the finite element space as a whole, where the indices i and j range up to the number of degrees of freedom on the reference cell and in the complete mesh, respectively. This use is most frequently implicit: almost all finite element packages describe finite element spaces as the span of a given number of shape functions, rather than in some more abstract way. deal.II is no exception to this rule.

The second use is more practical: By defining node functionals, we can associate degrees of freedom with finite element solutions. To this end, a finite element needs to describe the number of degrees of freedom per vertex, edge, face, and interior for each cell. For example, in 2D, a continuous bicubic element would have one degree of freedom per vertex, two per line, and four per cell interior. Figure 3, shows the locations of degrees of freedom for a few simple elements. This kind of information is consumed by the `DoFHandler` class described in the next section.

Of particular importance are degrees of freedom on the boundary of the cell, since they establish the continuity of finite element functions on the whole mesh by being shared between cells. Vice versa, the global continuity of such a function is already determined

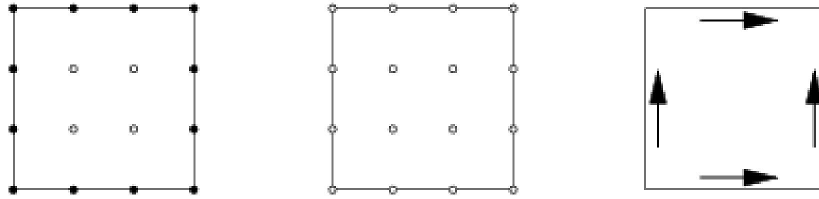

Fig. 3. Degrees of freedom of the standard (left) and discontinuous $\mathbb{Q}_3$ elements (middle), as well as lowest order Nedelec elements (right). Filled circles indicate node functionals requiring continuity across a face, open circles have no such requirements. Arrows denote continuity only of the indicated vector component.

```
template <int dim>
class FiniteElement {
  public:
    // ... as above

    const unsigned int dofs_per_vertex;
    const unsigned int dofs_per_line;
    const unsigned int dofs_per_quad;
    const unsigned int dofs_per_hex;

    const unsigned int dofs_per_cell;

    const FullMatrix<double> & interface_constraints () const;
};
```

Listing 4. The parts of the interface of the finite element base class concerning node functionals and continuity between cells.

by the distribution of degrees of freedom on the surface of the mesh cells. We call this the topology of the element. Figure 3 shows that several degrees of freedom may be located on a single geometrical item (in this case edges). Furthermore, the figure also shows that degrees of freedom, even if they denote interpolation points physically located on the boundary of a cell, may still logically be interior degrees of freedom if they don't imply continuity conditions, like in the case of discontinuous elements in the middle; for the shown discontinuous element, all 16 degrees of freedom are logically interior to the cell.

Listing 4 shows the part of the finite element base class interface that corresponds to node functionals. The constant data members are set by the constructor of the class based on information passed down by derived classes, and describe the logical location of degrees of freedom. The main use of this information is in allocating global indices for the degrees of freedom, as explained in the next section.

The frequently-used value `dofs_per_cell` is computed from the previous elements; for example, in 2D, `dofs_per_cell` = 4*`dofs_per_vertex` + 4*`dofs_per_line` + `dofs_per_quad`. Finally, the function `interface_constraints` returns interpolation constraints for hanging node constraints between cells if one cell is more refined than one of its neighbors and is used in the `ConstraintMatrix` class (see Section 3.4 below).

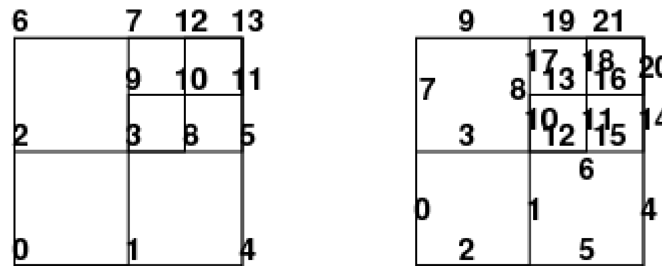

Fig. 4. Depictions of global enumerations of degrees of freedom on the mesh shown in Fig. 2 for $\mathbb{Q}_1$ elements with degrees of freedom associated with vertices (left) and lowest order Nedelec elements with degrees of freedom on cell faces (right).

2.3 Degrees of freedom handler

A global description of a finite element space is obtained by combining the geometric information on the location and connection of cells provided by a `Triangulation` with the local description and topology of degrees of freedom on a single cell introduced in Section 2.2.2, to provide a global enumeration of the degrees of freedom (DoFs) of a finite element space. Fig. 4 shows such enumerations for the simple mesh presented in Fig. 2 for two different finite elements. (Due to hanging nodes, the corresponding spaces are not conforming here; the algorithms used in deal.II to treat this are described in Section 3.4.)

Most finite element packages store the global numbers of degrees of freedom within the triangulation itself, sometimes even together with nodal data and/or matrix entries. However, such a data storage scheme severely restricts the use of these objects. For example, it does not allow to associate different finite element spaces with the same triangulation within a program, and it requires that the number of data vectors that need to be stored be hard-coded into the triangulation.

To avoid these limitations, deal.II separates this information (in a way somewhat similar to DiffPack [Langtangen 2003]). To this end, the class `DoFHandler` provides a global enumeration of degrees of freedom, given a `Triangulation` object describing the mesh and a `FiniteElement` object describing the association of degrees of freedom with vertices, faces, and cells as described above. A typical code fragment for initializing a `DoFHandler` based on the triangulation of Listing 1 is shown in Listing 5. With this global enumeration of DoFs, it is up to application programs to allocate as many vectors and matrices as necessary to solve a particular problem, and to associate the entries of matrices and vectors with the indices of degrees of freedom. It is also easily possible to associate an arbitrary number of `DoFHandler` objects with the same triangulation, possibly representing a number of different finite elements on the same mesh. (Note though, that a similar effect can be achieved using the deal.II class `FESystem` that provides a simple interface for combining simpler elements to form more complex, vector-valued elements suitable for systems of equations.)

By default, the `DoFHandler` class "distributes" degrees of freedom by walking over all cells in the order in which the triangulation presents them, and enumerates all DoFs not previously encountered. This often leads to a more or less random ordering that is sometimes undesirable, for example if ordering-dependent preconditioners (such as SSOR) or solvers (such as marching solvers for advection dominated problems, or sparse direct solvers) are

```
FE_Q<dim>       finite_element(3);
DoFHandler<dim> dof_handler(triangulation);
dof_handler.distribute_dofs(finite_element);
DoFRenumbering::Cuthill_McKee(dof_handler);
```

Listing 5. Initialization of a `DoFHandler` for $\mathbb{Q}_1$ elements as shown in Fig. 4, followed by bandwidth-reducing renumbering using the algorithm of Cuthill-McKee (follow-up to Listing 1). The argument to the finite element class indicates the polynomial degree; here, we select (bi-, tri-)cubic elements.

used. The `DoFRenumbering` namespace of deal.II therefore provides algorithms for re-sorting degrees of freedom once they have been initially distributed. Among the methods offered are the bandwidth reducing algorithms by Cuthill and McKee as well as downwind numbering techniques for advection problems.

deal.II also contains two variants to the `DoFHandler` class. The first assigns degrees of freedom on each refinement level of the mesh (i.e. on *all* cells of the cell-tree, not only the terminal ones), allowing for multilevel preconditioners as in [Bramble 1993; Hackbusch 1985] and in [Gopalakrishnan and Kanschat 2003] for discontinuous elements. These preconditioners yield optimal results with local smoothing on locally refined meshes for discontinuous finite elements (see [Kanschat 2004]), as well as for continuous elements on globally refined meshes.

A second variant of the `DoFHandler` class allows to select different finite elements on each cell of the mesh, enabling the use of the hp finite element method.

3. FROM DEGREES OF FREEDOM TO LINEAR SYSTEMS

The information discussed so far is independent of a particular application. To solve a concrete problem, we will solve one or a sequence of linear problems for which we have to obtain a matrix and right hand side vector, which then need to be passed to a solver for linear systems. In this section, we will discuss the functionality within deal.II that allows to integrate matrices and right hand side vectors. An overview of linear algebra capabilities is given in Section 4 below.

For the rest of this section, let us consider the Laplacian as an example problem. There, the bilinear form defining the system matrix is

$$a(u,v) = (\nabla u, \nabla v)_\Omega, \tag{1}$$

and matrix entries are calculated as

$$A_{ij} = (\nabla\varphi_i, \nabla\varphi_j)_\Omega = \sum_K (\nabla\varphi_i, \nabla\varphi_j)_K \qquad 0 \le i,j < N,$$

where we have split the integration into a sum of integrals over all elements K in the triangulation with N the *global* number of degrees of freedom. Rather than looping over all cells for each matrix entry, practical finite element programs usually compute local contributions

$$A^K_{ij} = (\nabla\varphi_i, \nabla\varphi_j)_K \qquad 0 \le i,j < n_K \tag{2}$$

for each cell K with n_K the number of degrees of freedom *per cell*. The global matrix is

```
const unsigned int dofs_per_cell = finite_element.dofs_per_cell;
std::vector<unsigned int> global_dof_indices(dofs_per_cell);

for (DoFHandler<dim>::active_cell_iterator
       cell = dof_handler.begin_active();
     cell != dof_handler.end(); ++cell)
  {
    // compute local_a as below

    cell->get_dof_indices(global_dof_indices);

    for (unsigned int i=0; i<dofs_per_cell; ++i)
      for (unsigned int j=0; j<dofs_per_cell; ++j)
        global_a(global_dof_indices[i], global_dof_indices[j])
          += local_a(i,j);
  }
}
```

Listing 6. Updating the global matrix with local contributions.

then assembled in a loop over all cells where we update matrix entries according to

$$A_{g(i),g(j)} = A_{g(i),g(j)} + A^K_{ij},$$

where $g(i)$ is the *global* index of the i-th degree of freedom on cell K. The implementation of assembling the global matrix from local contributions is shown in Listing 6. Note how the code abstracts from the spatial dimension as well as the number of degrees of freedom a particular finite element has on each cell. A similar technique is used for the right hand side vector. While it is possible to write application programs using exact integration (at least on parallelepipeds) for the local matrices A^K, we consider the use of quadrature formulæ the usual case and deal.II offers several classes supporting this in an efficient way. These classes will be discussed in the remainder of this section.

3.1 Numerical Quadrature

The local matrix contributions A^K_{ij} can in general not be computed as an analytic integral, but are approximated using quadrature, i.e. we compute

$$A^K_{ij} \approx \sum_q \nabla\varphi_i(\mathbf{x}_q) \cdot \nabla\varphi_j(\mathbf{x}_q) w_q \qquad 0 \le i,j < n_K, \tag{3}$$

where $\mathbf{x}_q$ are a set of quadrature points and w_q are the corresponding weights. Some bilinear forms (such as the one shown) can be integrated exactly even through quadrature if the locations and weights of quadrature points are chosen appropriately.

deal.II offers Gaussian quadrature formulæ of arbitrary order as well as closed Newton-Cotes and other formulæ by providing locations and weights of quadrature points on the reference cell. Since the reference cell is a line segment, square, or cube, quadrature formulæ are generated in 1D, and are then readily constructed in higher space dimensions by tensor products.

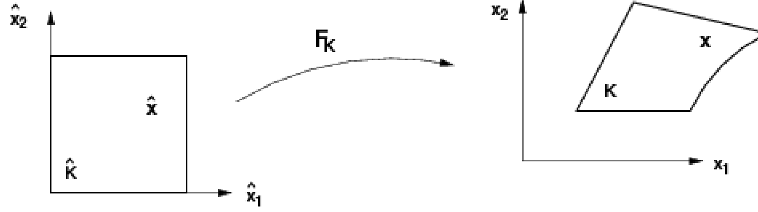

Fig. 5. Curvilinear mapping F_K of the reference element $\hat{K}$ to an element K in real space.

3.2 Mapping of the reference cell to cells in real space

In order to evaluate the integrals for the local matrices (2) using quadrature as in (3), we need to be able to evaluate functions and gradients at quadrature points in real space, rather than only on the reference cell where they are originally defined.

To this end, deal.II has a hierarchy of classes derived from a common `Mapping` base class that provides such mappings F_K from reference coordinates $\hat{x}$ to real coordinates $x \in K$ and back, see Figure 5. In particular, we support (bi-, tri-) linear mappings $F_K \in \mathbb{Q}_1$ (class `MappingQ1<dim>`), higher order polynomial mappings of degree k (class `MappingQ<dim>(k)`), C^1-continuous mappings (class `MappingC1<dim>`), as well as Eulerian mappings, where the real space coordinates are not only given by the locations of vertices of the mesh, but also by a displacement field given as a finite element function. Finally, a special mapping for Cartesian meshes exists with simple and fast transformation functions. Each mapping class provides functions to map points in both directions between the reference and real cell, as well as co- and contravariant transformations of tensors of rank 1 (i.e. vector-valued quantities such as gradients of shape functions) and 2 (e.g. second derivatives of shape functions).

Each of these mappings can be combined with any finite element space, thereby allowing sub-, iso-, super-parametric mappings; there is also a non-parametric $\mathbb{P}_k$ element. The use of iso-, or super-parametric elements is particularly important for the discontinuous Galerkin discretization of compressible flows [Hartmann 2002] as well as for other applications where high accuracy is required with curved geometries.

3.3 The `FEValues` abstraction

In order to compute local contributions as in equation (3), we have now transformed all integrals to the reference cell (assuming parametric elements). The integration of a cell matrix then reads

$$A_{ij}^K = \sum_q J^{-1}(\hat{\mathbf{x}}_q)\hat{\nabla}\varphi_i(\hat{\mathbf{x}}_q) \cdot J^{-1}(\hat{\mathbf{x}}_q)\hat{\nabla}\varphi_j(\hat{\mathbf{x}}_q)\ |\det J(\hat{\mathbf{x}}_q)| w_q, \tag{4}$$

where a hat indicates reference coordinates, and $J(\hat{\mathbf{x}}_q)$ is the Jacobian $\frac{\partial F_K(\hat{\mathbf{x}}_q)}{\partial \hat{\mathbf{x}}}$ of the mapping, evaluated at a quadrature point $\hat{\mathbf{x}}_q$ on the reference cell.

In order to evaluate such an expression in an application code, we have to access three different kinds of objects: a quadrature object that describes locations $\hat{\mathbf{x}}_q$ and weights w_q of quadrature points on the reference cell; a finite element object that describes the gradients $\hat{\nabla}\varphi_i(\hat{\mathbf{x}}_q)$ of shape functions on the unit cell; and a mapping object that provides the Jacobian as well as its determinant. Dealing with all these objects would be cumbersome

```
QGauss<dim>   quadrature(3);
MappingQ<dim> mapping (2);
FEValues<dim> fe_values(mapping, finite_element, quadrature,
                        (update_values    | update_gradients |
                         update_q_points  | update_JxW_values));
```

Listing 7. Creation of an `FEValues` object.

and error prone.

On the other hand, these three kinds of objects almost always appear together, and it is in fact very rare for deal.II application codes to do anything with quadrature, finite element, or mapping objects besides using them together. For this reason, we have introduced the `FEValues` abstraction combining information on the shape functions, the geometry of the actual mesh cell and a quadrature rule on a reference cell. Upon construction it takes one object of each of the three mentioned categories. Later, it can be "re-initialized" for a concrete grid cell and then provides mapped quadrature points and weights, mapped shape function values and derivatives as well as some properties of the transformation from the reference cell to the actual mesh cell.

Since computation of any of these values is potentially expensive (for example when using high order mappings with high order elements), the `FEValues` class only computes what is explicitly asked for. To this end, it takes a list of flags at construction time specifying which quantities should be updated each time a cell is visited. In addition, allowing further optimizations, the functions filling the data fields of `FEValues` are able to distinguish between values that have to be recomputed on each cell (for example mapped gradients) and quantities that do not change from cell to cell (for example the values of shape functions of the usual $\mathbb{Q}_k$ finite elements at the same quadrature points on different cells; this property does not hold for the shape functions of Raviart-Thomas elements, however, which must be rotated with the local cell).

The code fragment in Listing 7 shows how such an `FEValues` object can be constructed and initialized. It will evaluate the shape functions of the given finite element (created, for example, in Listing 5), mapped to the real cell using a second-degree (bi-, tri-quadratic) mapping, at the quadrature points described by the given Gaussian quadrature formula with three points in each coordinate direction. The last argument to the constructor tells the object to provide values and gradients of shape functions on each cell, along with the mapped quadrature points as well as the products of the determinant of the Jacobian matrix and the quadrature weights on the cell K ("det J times w", or `JxW` values – since this product always appears in conjunction, we provide it as a single value rather than as two separate ones).

The `FEValues` object created in Listing 7 can then be used to compute local contributions (4) to the cell matrix, as shown in Listing 8. Here, `fe_values.shape_grad(i,q)` returns the gradient of the i-th local (mapped) shape function at the q-th quadrature point, and `fe_values.JxW(q)` provides $|\det J(\hat{\mathbf{x}}_q)| w_q$.

For the implementation of discontinuous Galerkin methods as well as for error estimates, integrals over faces of grid cells must be computed. While the general structure for this is the same as for integration over cells, the mapping is different, and one frequently needs additional information such as the outward normal vector to the cell at a given quadrature

```
for (DoFHandler<dim>::active_cell_iterator
       cell = dof_handler.begin_active();
     cell != dof_handler.end(); ++cell)
  {
    fe_values.reinit(cell);

    for (unsigned int q=0; q<fe_values.n_quadrature_points; ++q)
      for (unsigned int i=0; i<fe_values.dofs_per_cell; ++i)
        for (unsigned int j=0; j<fe_values.dofs_per_cell; ++j)
          local_a(i,j) += fe_values.shape_grad (i,q) *
                          fe_values.shape_grad (j,q) *
                          fe_values.JxW (q);
    // copy local_a into the global matrix as above
  }
}
```

Listing 8. Integration of a (local) bilinear form.

point. To this end, deal.II provides an analogous class `FEFaceValues` for integration on faces.

3.4 Local refinement and "hanging nodes"

One of the initial design goals of deal.II is to fully support adaptivity: if mesh adaptation is necessary to resolve local features of the solution, the library offers the possibility of local, rather than global mesh refinement and coarsening (see Fig. 2).

After local refinement, consistency of the finite element functions between the refined and the coarse part of the mesh must be ensured. For triangles and tetrahedra, the most common technique to do so is the use of transition elements (for example "red-green refinement" [Bornemann et al. 1993]). However, such transition elements are awkward or impossible to construct for all-quadrilateral and all-hexahedral meshes. Instead, we opt for a technique known as "hanging nodes" in which additional constraints have to be added to the linear system to ensure consistency [Rheinboldt and Mesztenyi 1980]. In return, the mesh structure remains simple and refinement of cells is a local operation. The only restriction we require mostly for algorithmic reasons is that each face of a cell is divided at most once ("one-irregular" mesh). This can be ensured by refining a few additional cells in each cycle as necessary and has no significant detrimental effects on the complexity of a mesh.

3.4.1 *Elimination of "hanging nodes" for continuous elements.* After arriving at a mesh as the one shown in the left part of Fig. 2, we enforce consistency through algebraic constraints. Their derivation is based on the observation that the required continuity can only be achieved if the finite element functions of the refined side of the face are constrained to be in the coarse space on the face. (Similar observations holds for all "conforming" finite elements, including $H(\Omega, \mathrm{div})$- and $H(\Omega, \mathrm{curl})$-conforming ones.) For simplicity, let us look at the $\mathbb{Q}_1$ space with degrees of freedom as shown in the left part of Fig. 4: if we expand a finite element function as $u_h(x) = \sum_{i=0}^{13} U_i \varphi(x)$, then it will only

```
ConstraintMatrix hanging_nodes;
DoFTools::make_hanging_node_constraints(dof_handler,
                                        hanging_nodes);
hanging_nodes.close();
```

Listing 9. Initialization of “hanging nodes” constraints.

```
hanging_nodes.condense(sparsity_pattern);
hanging_nodes.condense(matrix);
hanging_nodes.condense(right_hand_side);
// Solve constrained linear system here
hanging_nodes.distribute(solution);
```

Listing 10. Application of “hanging nodes” constraints to a linear system.

be globally continuous if

$$U_8 = \frac{1}{2}U_3 + \frac{1}{2}U_5, \qquad \text{and} \qquad U_9 = \frac{1}{2}U_3 + \frac{1}{2}U_7.$$

Such constraints can be computed for all faces with hanging nodes and all conforming finite element spaces. They can then be gathered into a homogenous system $CU = 0$ of constraints, where for the simple example above the matrix C would have the form

$$C = \begin{pmatrix} 0 & 0 & 0 & -\frac{1}{2} & 0 & -\frac{1}{2} & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{2} & 0 & 0 & 0 & -\frac{1}{2} & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix},$$

and can obviously be stored in a very compact form.

The changes in user code required to deal with hanging nodes are minimal. Listing 9 shows the necessary code to let the library compute the system of constraints, where the matrix C is represented by the class `ConstraintMatrix`. After all constraints have been added to this object (possibly involving more than just the one call to the shown function), the matrix entries are sorted and stored in a more efficient format using the `close()` function.

In a second step, the constraint matrix can be used to remove constrained nodes from a linear system $AU = F$ previously assembled, a process called “condensation”. Since this adds some non-zero entries to a sparse matrix representation of A, we also have to “condense” the sparsity pattern. The code to do these operations is shown in Listing 10. After this, the condensed linear system can be solved, yielding the solution values for all non-constrained degrees of freedom. In a final step, we “distribute” the so-computed values also to constrained nodes, for example by computing U_8 from U_3 and U_5.

3.4.2 *“Hanging nodes” and discontinuous elements.* If discontinuous finite elements are used, all nodes are in the interior of mesh cells and thus no “hanging nodes” have to be eliminated. On the other hand, flux terms between involving the degrees of freedom from both sides of a face must be computed. These terms are computed using the `FEFaceValues` class mentioned in Section 3.3, or, if we have to integrate over a face between cells of different refinement level, a variant class `FESubfaceValues` object offering integration on the subset of the coarse face matching one of the children.

```
const unsigned int        n_dofs = dof_handler.n_dofs();
std::vector<unsigned int> row_lengths(n_dofs);
DoFTools::compute_row_length_vector(dof_handler, row_lengths);
SparsityPattern           sparsity(n_dofs, n_dofs, row_lengths);

DoFTools::make_sparsity_pattern(dof_handler, sparsity);
sparsity.compress();

SparseMatrix<double>      matrix(sparsity);
```

Listing 11. Initialization of a sparse matrix (follow-up to Listing 5).

4. LINEAR ALGEBRA

A major part of the solution process for discretized partial differential equations is the solution of large sparse systems of equations, where we consider systems of the size of hundreds of thousands and more unknowns as large. Usually, the rows of these systems contain only a few dozen entries, depending on the finite element space and the mesh topology. Nevertheless, these entries cannot be arranged into a narrow band along the diagonal; on homogeneously refined meshes with mesh size h, the bandwidth cannot be reduced below a number of order $\mathcal{O}(h^{-(d-1)}) = \mathcal{O}(N^{(d-1)/d})$. Direct solvers are therefore often unusable for fine meshes and in higher dimensions.

For this reason, deal.II contains a comprehensive set of linear algebra components for systems with sparse structure. Underlying sparse matrices is the `SparsityPattern` class that handles the sorted and compressed storage of locations of nonzero matrix entries. Independent from, but built atop the sparsity pattern are `SparseMatrix` objects. This separation allows multiple matrices to use the same sparsity pattern without having to generate and store it more than once.

The generation of a sparsity pattern for a given mesh and finite element is shown in Listing 11. Initialization of a sparse matrix happens in three steps: First, an object of type `SparsityPattern` is created. Here, the matrix is quadratic with as many rows and columns as there are degrees of freedom in the `DoFHandler` object. The last argument to the constructor presents an upper bound on the number of nonzero entries for each of the rows of the matrix as previously computed by the function in `DoFTools`. In the second step, the actual nonzero entries are computed and allocated, and the resulting sparsity pattern is sorted and compressed (i.e., extraneous entries are removed). Finally, this pattern is used to create the matrix, which is now ready to, for example, receive the local contributions as shown in Listings 6 and 8.

Building atop these matrices, the library then implements standard matrix-based preconditioning methods such as Jacobi, Gauss-Seidel and SSOR, as well as blocked versions thereof. Additionally, incomplete LU decomposition and Vanka-type preconditioners are included.

For the solution of linear systems, most customary iteration schemes are implemented, from the linear Richardson method to a variety of Krylov space methods including conjugate gradient, Bicgstab or GMRES. All these are implemented in an abstract way, using only the properties of the matrix and preconditioner as linear mappings, i.e. member functions of matrix classes that provide matrix-vector products. It is therefore trivial to use

these solvers also with more complex matrix objects, for example representing the Schur complement of a saddle point system, or preconditioners constructed in similar ways, as long as they offer a function implementing a matrix-vector product. A few such classes are actually provided by deal.II.

Setting out from a vector class with a complete set of arithmetic operations, an important feature is a class for block vectors, a feature apparently found in no other finite element or linear algebra package. For systems of partial differential equations, it allows the modularization of the resulting systems of equations. Together with matching block matrix classes it provides for efficient solution methods for saddle-point problems using block preconditioners [Kanschat 2005] or Schur complement formulations [Bangerth 2004].

In addition to its own linear algebra classes, deal.II has a comprehensive interface to the vectors, matrices and solvers provided by the PETSc library [Balay et al. 2004] as well as several sparse direct solvers. This allows deal.II to leverage the abilities of PETSc to solve linear systems in parallel on distributed memory machines.

5. INTERFACES TO OTHER SOFTWARE

In addition to the functionality deal.II provides itself, it also has interfaces to numerous additional programs and libraries for pre- and postprocessing as well as linear algebra. Following is a list of areas in which deal.II uses external programs:

Grid generators. The grid handling of deal.II always assumes that there is a coarse mesh approximating the geometry of the domain of computation. While the mesh refinement process is able to follow curved boundaries, the coarse mesh has to be provided either internally using functions such as the one shown in Listing 1 for a simple hypercube, or externally through mesh generators. deal.II has the capability to read several different mesh formats, among which are UCD, DBMESH, XDA, GMSH, and NetCDF.

Linear algebra. While deal.II provides an extensive suite of linear algebra classes for the iterative solution of linear systems, it does not offer direct or sparse direct solvers for large matrices. However, interfaces to the sparse direct solvers in the UMFPACK [Davis 2004] and HSL [HSL 2004] packages are provided; a copy of UMFPACK is actually part of deal.II distributions, courtesy of the author. In addition, basic support for LAPACK [Anderson et al. 1999] eigenvalue solvers exists.

Parallelization. deal.II includes genuine support for shared memory parallelization by multi-threading, a significant advantage at a time when multiprocessor and multicore machines become more common. For parallelization by message passing, an interface to PETSc [Balay et al. 2004] is provided. Since PETSc relies on MPI, it is portable to a large number of parallel systems. Interfaces to METIS [Karypis 2006] can be used to partition meshes efficiently.

Visualization. The library has output drivers for a significant number of different visualization tools as well as its own Postscript[1] output. Generation of output data is implemented as a two-step process, where the first generates an intermediate format from simulation data independent of the actual output format, and the second step them converts this, rather simple, intermediate data into one of the supported graphical formats. It is therefore quite simple to write a new driver for a missing file format. At present, the

[1] Trademark of Adobe Systems.

```
if (cell->has_children())
  for (unsigned int c=0; c<8; ++c)
    do_something_on_child_cell (cell->child(c));
```

Listing 12. Example of a loop that tries to access invalid data in 2D.

following formats are supported: VTK [Schroeder et al. 2004], OpenDX[2] in both text and binary format, UCD format for AVS Express[3], binary and text files for Tecplot[4], gnuplot, povray, encapsulated postscript, and GMV.

6. DOCUMENTATION AND ERROR DETECTION

In addition to the source code, deal.II is distributed with a wealth of documentation detailing all public and internal interfaces, as well as explaining the interplay between the different modules.

The documentation is split in two parts. The first part is a user guide written in the form of a tutorial. It presently contains 20 programs and about 450 pages if printed. It starts with very basic features of the library and explores more and more complex problems. These example programs cover a variety of partial differential equations from the Laplace equation to quasi-static large-deformation elasticity, as well as a large part of the functionality of the library. Their purpose is to show how the different parts of the library can be combined to build a finite element program; though they follow along the same and well thought-out lines they do not intend to show how such a program *should* look. These well documented programs typically serve as the starting point for new users: they take the basic structure of one of the problems and successively replace bilinear forms, geometry, or forcing functions to match their particular application. Although the library is decidedly not intended to solve any particular application, the set of equations covered in the tutorial certainly serve as examples of what is possible with the library.

The second part is a reference handbook describing *all* classes and functions of the library. It is produced directly from the source code using the documentation tool doxygen [van Heesch 2006] and provides several indices for looking up particular classes as well as providing bird's views of connections between different parts of the library. While intended as an online handbook, a printed version can be produced as well, encompassing about 3000 pages at present. This documentation has proven invaluable in teaching graduate students at the University of Heidelberg and at Texas A&M University as well as introducing researchers to deal.II at the German Aerospace Center.

An equally important part of documentation is automatic assistance in case of programming errors. To this end, the library contains some 5,500 checks where ranges and sizes of parameters, consistency of internal state, and other things are checked. If such an assertion is violated, a detailed description of the problem and a back trace is produced, immediately helping users to find places where they call library functions with invalid data. For example, Listing 12 shows a loop over the 8 children of a cell. However, if in 2D, there are only 4 children, and the library will trigger an exception when trying to access child 4 (counted

[2]Formerly IBM Visualization Data Explorer, `www.opendx.org`.
[3]AVS Express is a trademark of Advanced Visual Systems, `www.avs.com`.
[4]Tecplot is a trademark of Tecplot Inc., `www.tecplot.com`.

```
An error occurred in line <426> of file
    <deal.II/include/grid/tria_accessor.templates.h>
in function
    int TriaObjectAccessor<2, dim>::child_index(unsigned int) const
    [with int dim = 2]
The violated condition was:
    i<4
The name and call sequence of the exception was:
    ExcIndexRange(i,0,4)
Additional Information:
Index 4 is not in [0,4[

Stacktrace:
-----------
#0  ./step-1: TriaObjectAccessor<2, 2>::child_index(unsigned) const
#1  ./step-1: CellAccessor<2>::child(unsigned) const
#2  ./step-1: bogus_access()
#3  ./step-1: main
```

Listing 13. Illustration of the exception generated by the code shown in Listing 12.

from zero) rather than returning invalid data. The form of this exception is shown in Listing 13. The message and backtrace give information about what happened and where. In addition, the program is aborted, allowing easy access to the offending code in a debugger. In practice, it turns out that errors caught this way are usually fixed within a few minutes, whereas errors due to invalid data often take an order of magnitude longer to remedy. In addition, it is our experience that about 90% of programming errors are actually cases of invalid function parameters, violations of internal assumptions, and similar, almost all of which are caught with the large number of assertions in the library. The remaining 10% of errors are errors in the logic of a program (such as forgotten terms in a bilinear form) that can clearly not be found with this technique.

Obviously, this form of error checking is expensive in terms of computing time. Therefore, the library is compiled both with and without these checks. After debugging and verifying that the program runs without triggering any exceptions, it can be recompiled and linked with the optimized version of the deal.II libraries to allow tested programs to run at maximal performance. In conjunction with the use of templates to specify the space dimension while using the same source code for 1D, 2D, and 3D, this usually allows to develop and test a program in 2D where runs are comparatively cheap, and then do production runs in 3D using the same source but compiled in optimized mode.

In addition to these obvious benefits, the internal consistency checks, in conjunction with some 1,000 regression test programs run every night on a number of different machines, also ensure that the functionality we offer today is not accidentally changed by future development.

7. APPLICATIONS

Since the start of the project and its first publicly available version in early 2000, deal.II has been used successfully in the development of codes for a wide variety of problems

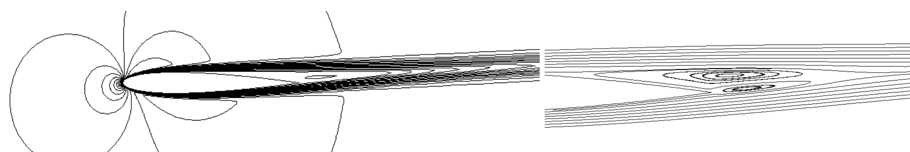

Fig. 6. Compressible flow at $M = 0.5, \mathrm{Re} = 5000, \alpha = 3^\circ$ around the NACA0012 airfoil: Mach isolines (left); streamlines near the trailing edge (right).

in academic research, applied sciences, and industrial projects. At the time of writing, the deal.II publications page[5] lists more than 60 publications presenting results obtained with the library, in fields as disparate as mathematical error analysis for finite element discretizations, incompressible and compressible fluid flow, fluid-structure interaction, magnetohydrodynamics, biomedical inverse imaging problems, fuel cell modeling, simulation of crystal growth, and many more. We show results obtained with two of these codes in the following subsections.

7.1 Compressible flows

The first two examples of applications are obtained with an aerodynamics code (some 45,000 lines of code) built atop deal.II. Based on a discontinuous Galerkin discretization of the compressible Navier-Stokes equations, cf. [Hartmann and Houston 2006a], Figure 6 shows a subsonic viscous compressible flow with separation (taken from [Hartmann and Houston 2006b]). This computation uses a piecewise quadratic boundary approximation (`MappingQ(2)`) of the profile. The nonlinear equations are solved fully implicitly using a Newton iteration where linear systems are solved with the help of PETSc. By simply replacing the finite element object `FE_DGQ<dim>(1)` by `FE_DGQ<dim>(`p`)`, $p > 1$, the same code can use higher polynomial degree shape functions to obtain higher order accurate flow solutions.

In a second example (taken from [Hartmann 2006]) we show a supersonic compressible flow and an adjoint solution related to the pressure induced drag coefficient, see Figure 7. In Figure 8 we show the kind of meshes obtained with deal.II, and compare two different refinement strategies and the accuracy of the resulting solution (with respect to the pressure induced drag coefficient c_{dp}): the left mesh in Figure 7 has been adapted according to residual-based refinement indicators, while the one at the right uses an adjoint-based refinement criterion.

7.2 Biomedical imaging

The second set of examples of applications built atop deal.II are obtained with a large (some 68,000 lines of code) program that implements fully adaptive, three-dimensional biomedical and other imaging methodologies. The code implements algorithms described in [Bangerth 2004; Bangerth et al. 2005] and is able to convert experimentally obtained optical tomography data into three-dimensional images of tumor locations, shapes, and sizes [Joshi et al. 2006; Joshi et al. 2004].

Fig. 9 shows the first step in such simulations: verifying that the (light propagation) model is correct. To this end, we use the program to predict surface light intensities com-

[5]See `http://www.dealii.org/developer/publications/toc.html`.

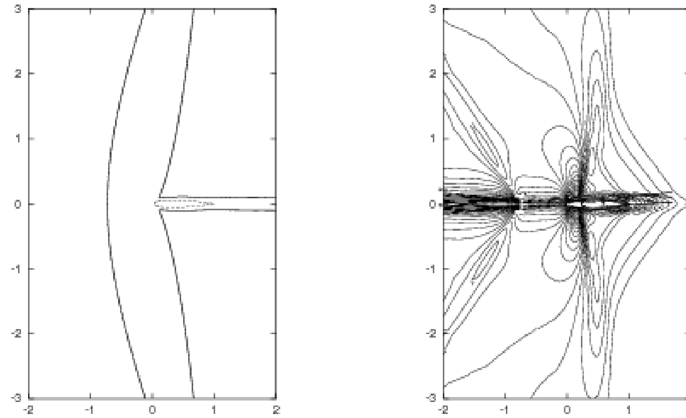

Fig. 7. Compressible flow at $M = 1.2, \mathrm{Re} = 1000, \alpha = 0°$ around the NACA0012 airfoil: Sonic isolines of the flow solution (left); isolines of the first component of the computed adjoint solution (right).

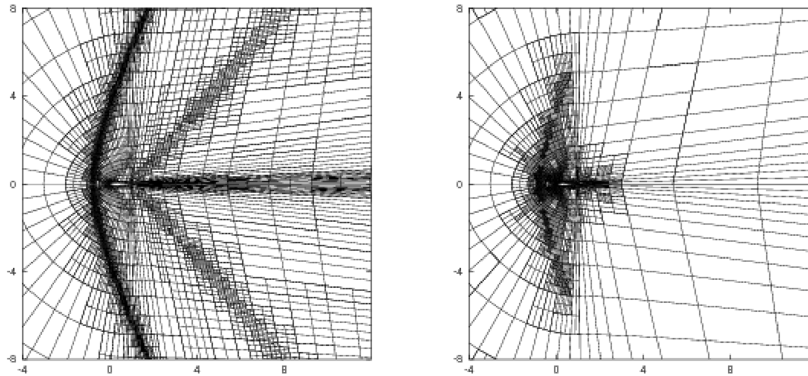

Fig. 8. Supersonic compressible flow at $M = 1.2, \mathrm{Re} = 1000, \alpha = 0°$ around the NACA0012 airfoil: Residual-based refined mesh of 17,670 elements with 282,720 degrees of freedom and $|J_{c_{\mathrm{dp}}}(\mathbf{u}) - J_{c_{\mathrm{dp}}}(\mathbf{u}_h)| = 1.9 \cdot 10^{-3}$ (left); goal-oriented (adjoint-based) refined mesh for c_{dp}: mesh of 10,038 elements with 160,608 degrees of freedom and $|J_{c_{\mathrm{dp}}}(\mathbf{u}) - J_{c_{\mathrm{dp}}}(\mathbf{u}_h)| = 1.6 \cdot 10^{-4}$ (right).

putationally for an experimental geometry resembling a human breast when illuminated by two narrow laser beams, with known tumor locations. This prediction is then compared with experimental data for the same situation. The effect of adaptivity again is clearly seen.

Fig. 10 shows results from the second step: reconstructing tumor locations from measurements. The solution of this inverse and ill-posed problem requires the solution of a nonlinear and high-dimensional optimization problem where several partial differential equations modeling light propagation form additional nonlinear constraints.

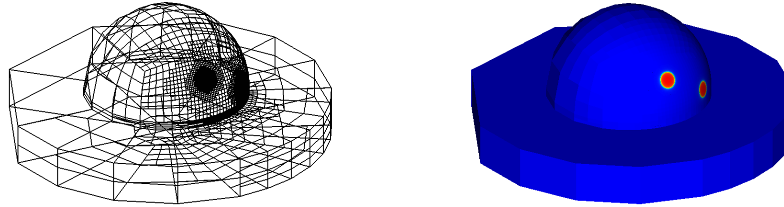

Fig. 9. Surface of a three-dimensional mesh with 66,989 cells simulating light propagation in a breast phantom geometry (left). Solution obtained on this mesh with 321,264 degrees of freedom.

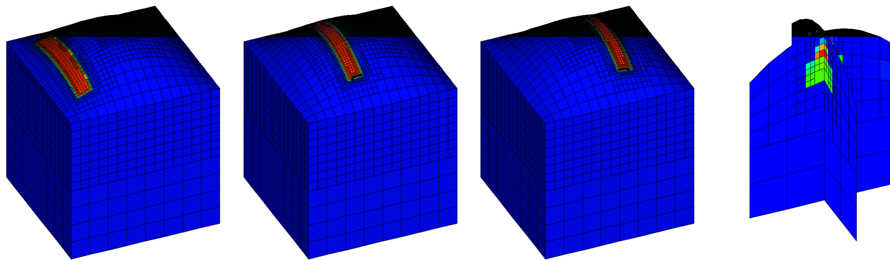

Fig. 10. Simulation of a widened laser scanning over a surface (left to center right). Cut planes through reconstructed tumor densities (right). Computations involved solving a PDE-constrained optimization problem with 640,902 unknowns.

8. CONCLUSIONS AND OUTLOOK

In this paper, we have given an overview of the software design and data abstractions chosen in the implementation of deal.II, a general purpose toolbox for building finite element software downloadable at `www.dealii.org`. We have given a detailed description of basic concepts and structures which allow the use of deal.II in a wide variety of situations, independent of particular applications. The breadth of applications built on deal.II mentioned in Section 7 indicates the success of this design.

The key in achieving this is a proper separation of concepts, such as meshes, finite element spaces, and degrees of freedom as well as the possibility to arbitrarily combine finite element spaces, numerical quadrature and mapping information. All these concepts are implemented in separate classes and class hierarchies and can be assembled in arbitrary ways to form application programs. On the other hand, attention has been paid to avoid the usual overhead incurred by making frameworks overly general, for example by using constant variables (see, e.g., the number of degrees of freedom per geometric object in Listing 4) instead of the virtual functions so ubiquitous in some other object-oriented finite element codes (see, for example, OOFEM [Pátzak and Bittnar 2001]), or by using constants in the form of template parameters that allow compilers to perform various optimizations at compile time rather than deferring these checks to run-time. The introduction of the `FEValues` class discussed in Section 3.3 is another example of an optimization to avoid

the use of overly fine-grained virtual functions, as well as a way to present an easy to use interface to users when combining arbitrary finite elements, mappings, and quadratures. Finally, deal.II adopts many accepted standard concepts, such as iterators, and offers an extensive tutorial and API documentation, which helps introducing deal.II to new users as well as in the daily work of deal.II experts.

While most of the externally visible design is frozen to provide interface stability to users, deal.II continues to grow in many directions. Since its first public release in 2000, it grows at an almost constant rate of 3000 lines of code per month. Currently, support for the hp-method is being implemented and multigrid with global coarsening is under consideration. New finite element spaces are easily added when needed; currently planned are the vector-valued elements of Brezzi, Douglas and Marini (BDM) and Arnold, Boffi and Falk (ABF), as well as plate elements. Furthermore, there are first considerations of extending deal.II from isotropic to anisotropic refinement capabilities as well as extending deal.II's hexahedral mesh representation to hybrid meshes including tetrahedra, hexahedra, prisms and pyramids. Through a publicly available Subversion repository, access to this and other current development is available to all parties interested in participation.

REFERENCES

ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, S., DEMMEL, J., DONGARRA, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., AND SORENSEN, D. 1999. *LAPACK user's guide*, 3rd ed. Software - Environments - Tools, vol. 9. SIAM.

BALAY, S., BUSCHELMAN, K., EIJKHOUT, V., GROPP, W. D., KAUSHIK, D., KNEPLEY, M. G., MCINNES, L. C., SMITH, B. F., AND ZHANG, H. 2004. PETSc users manual. Tech. Rep. ANL-95/11 - Revision 2.1.5, Argonne National Laboratory.

BANGERTH, W. 2000. Using modern features of C++ for adaptive finite element methods: Dimension-independent programming in deal.II. In *Proceedings of the 16th IMACS World Congress 2000, Lausanne, Switzerland, 2000*, M. Deville and R. Owens, Eds. IMACS – Department of Computer Science, Rutgers University, New Brunswick. Document Sessions/118-1.

BANGERTH, W. 2004. A framework for the adaptive finite element solution of large inverse problems. I. Basic techniques. Tech. Rep. 04-39, Institute for Computational Engineering and Sciences (ICES), University of Texas at Austin.

BANGERTH, W., JOSHI, A., AND SEVICK-MURACA, E. M. 2005. Adaptive finite element methods for increased resolution in fluorescence optical tomography. *Progr. Biomed. Optics Imag. 6*, 318–329.

BANGERTH, W. AND RANNACHER, R. 2003. *Adaptive Finite Element Methods for Solving Differential Equations*. Birkhäuser, Basel.

BANK, R. E. 1998. *PLTMG: a software package for solving elliptic partial differential equations*. SIAM, Philadelphia. Users' guide 8.0.

BECKER, R. AND RANNACHER, R. 1998. Weighted a posteriori error control in FE methods. In *Proc. of ENUMATH 95, in Proc. of ENUMATH 97*, H. G. Bock et al., Eds. World Scientific, Singapore.

BORNEMANN, F., ERDMANN, B., AND KORNHUBER, R. 1993. Adaptive multilevel methods in three space dimensions. *Int. J. Numer. Meth. Engrg. 36*, 3187–3203.

BRAMBLE, J. H. 1993. *Multigrid Methods*. Longman.

BRENNER, S. C. AND SCOTT, R. L. 2002. *The Mathematical Theory of Finite Elements*, second ed. Springer, Berlin-Heidelberg-New York.

BREZZI, F. AND FORTIN, M. 1991. *Mixed and Hybrid Finite Element Methods*. Springer.

CASTILLO, P., RIEBEN, R., AND WHITE, D. 2005. FEMSTER: An object-oriented class library of higher-order discrete differential forms. *ACM Trans. Math. Software 31*, 425–457.

CIARLET, P. G. 1978. *The Finite Element Method for Elliptic Problems*. North-Holland.

DAVIS, T. A. 2004. Algorithm 832: UMFPACK V4. 3 – an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw. 30,* 2, 196–199.

GOPALAKRISHNAN, J. AND KANSCHAT, G. 2003. A multilevel discontinuous Galerkin method. *Numer. Math. 95,* 3, 527–550.

HACKBUSCH, W. 1985. *Multi-grid Methods and Applications*. Springer.

HARTMANN, R. 2002. Adaptive finite element methods for the compressible euler equations. Ph.D. thesis, University of Heidelberg.

HARTMANN, R. 2006. Adaptive discontinuous Galerkin methods with shock-capturing for the compressible Navier-Stokes equations. *Int. J. Numer. Meth. Fluids*. To appear.

HARTMANN, R. AND HOUSTON, P. 2006a. Symmetric interior penalty DG methods for the compressible Navier–Stokes equations I: Method formulation. *Int. J. Num. Anal. Model. 3,* 1, 1–20.

HARTMANN, R. AND HOUSTON, P. 2006b. Symmetric interior penalty DG methods for the compressible Navier–Stokes equations II: Goal–oriented a posteriori error estimation. *Int. J. Num. Anal. Model. 3,* 2, 141–162.

HSL 2004. The Harwell Subroutine Library. http://www.cse.clrc.ac.uk/Activity/HSL.

JOSHI, A., BANGERTH, W., HWANG, K., RASMUSSEN, J. C., AND SEVICK-MURACA, E. M. 2006. Fully adaptive fem based fluorescence optical tomography from time-dependent measurements with area illumination and detection. *Med. Phys., accepted*.

JOSHI, A., BANGERTH, W., AND SEVICK-MURACA, E. M. 2004. Adaptive finite element modeling of optical fluorescence-enhanced tomography. *Optics Express 12*, 5402–5417.

KANSCHAT, G. 1996. Parallel and adaptive Galerkin methods for radiative transfer problems. Ph.D. thesis, Universität Heidelberg. Preprint SFB 359, 1996-29.

KANSCHAT, G. 2004. Multi-level methods for discontinuous Galerkin FEM on locally refined meshes. *Comput. & Struct. 82,* 28, 2437–2445.

KANSCHAT, G. 2005. Block preconditioners for LDG discretizations of linear incompressible flow problems. *J. Sci. Comput. 22,* 1, 381–394.

KANSCHAT, G. 2006. *Discontinuous Galerkin Methods for Viscous Flow Problems*. Teubner. to appear.

KARYPIS, G. 2006. METIS – Serial graph partitioning and fill-reducing matrix ordering. `glaros.dtc.umn.edu/gkhome/views/metis/index.html`.

KIRK, B., PETERSON, J. W., STOGNER, R., AND PETERSEN, S. 2006. The `libmesh` finite element library. `http://libmesh.sourceforge.net`.

LANGTANGEN, H. P. 2003. *Computational Partial Differential Equations: Numerical Methods and Diffpack Programming*. Texts in Computational Science and Engineering. Springer Verlag.

PATZÁK, B. AND BITTNAR, Z. 2001. Design of object oriented finite element code. *Advances in Engineering Software 32,* 10–11, 759–767.

PÁTZAK, B. AND BITTNAR, Z. 2001. Design of object oriented finite element code. *Adv. Engrg. Softw. 32*, 759–767.

PLAUGER, P. J., STEPANOV, A. A., LEE, M., AND MUSSER, D. R. 2000. *The C++ Standard Template Library*. Prentice Hall.

RHEINBOLDT, W. C. AND MESZTENYI, C. K. 1980. On a data structure for adaptive finite element mesh refinements. *ACM Trans. Math. Softw. 6*, 166–187.

SCHROEDER, W., MARTIN, K., AND LORENSEN, B. 2004. *The Visualization Toolkit: An Object-Oriented Approach To 3D Graphics*, 3rd ed. Kitware Inc.

STEPANOV, A. A. AND LEE, M. 1995. The standard template library. Tech. Rep. HPL-95-11, HP Labs.

STROUSTRUP, B. 1997. *The C++ Programming Language*, 3rd ed. Addison-Wesley.

SUTTMEIER, F.-T. 1996. Adaptive finite element approximation of problems in elasto-plasticity theory. Ph.D. thesis, Universität Heidelberg.

VAN HEESCH, D. 2006. Doxygen. `www.doxygen.org`.