# Concepts for Object-Oriented Finite Element Software – the deal.II Library

Wolfgang Bangerth and Guido Kanschat

Institute of Applied Mathematics
University of Heidelberg
Germany
Email: wolfgang.bangerth@iwr.uni-heidelberg.de,
kanschat@iwr.uni-heidelberg.de

**Abstract**

An overview of the deal.II library is given. This library provides the functionality needed by modern numerical software used in the finite element solution of partial differential equations, offering adaptively refined meshes, different finite element classes, multigrid solvers and support for one, two and three spatial dimensions.

We give a description of the basic design criteria used in the development of the library and how they were transformed into actual code, and some examples of the use of the library in numerical analysis.

## 1 Design and evolution of deal.II

The DEAL project, short for *D*ifferential *E*quations *A*nalysis *L*ibrary, was started to provide means for the implementation of adaptive finite element methods. In fact, the development of DEAL and adaptive methods at the Institute of Applied Mathematics in Heidelberg are closely linked. From this starting point, a finite element library was needed, that is able to handle grids with strongly varying mesh width and supports strategies for the computation of error estimates based on dual problems.

While DEAL was a library developed since 1993, this article gives an overview of the design criteria, programming models and fields of application of its successor library, deal.II.

### 1.1 Design criteria

deal.II was written with the following aims in mind:

- *Flexibility:* Our aim was to produce a library which enables us to try and test our ideas in a way as quick as possible. The library should thus be easily extendible with respect to the most common approaches in numerical analysis, i.e., different

variational formulations, different space dimensions, different finite element spaces, different linear solvers.

- *High level interfaces:* The library should be as simple to use as possible. Quite complex data structures are necessary for using different finite elements on locally refined grids. The interface is designed to hide and shield these structures from the user. This way, programmers applying the library do not have to know about the grid handling details and – more important – cannot interfere with essential data needed by the library functions.

- *Efficiency:* The computations involved in finite element calculations are often highly time-consuming. Furthermore, the memory requirements are enormous and they will fill up any machine we have. Therefore, efficiency has to be considered as well.

Obviously, some of these criteria are contradictory and must be traded against each other. In almost all cases, we traded efficiency in favor of safety and flexibility. However, some decisions were also made with the performance aspect in mind. These do not so much affect the complicated grid handling, which is not overly time-critical, but are especially found in the use of finite element objects and in the linear algebra sub-library.

Another important decision for simplicity against flexibility was the reduction to hypercube (line, quadrilateral, hexahedron) cells and their transformations to the physical space only. The previous DEAL library featured the possibility of combining simplicial (triangle, tetrahedron) and hypercube elements and in the end this additional flexibility caused very complicated data structures and violations of type safety. Since in all our experiments hypercube cells proved superior to simplicial ones, the decision was made against the latter. An immediate result of this decision is the absence of closure cells. Treatment of hanging nodes is much better done in the numerical than in the geometrical context (cf. Section 3.3).

Two other important criteria in the design of the library, unfortunately not widely followed in the academic world, were:

- *Safety:* Run-time checks of function parameters and the internal consistency of the library have proven an invaluable means of finding programming errors as early as possible. Even for experienced users of a library, at least 90% of errors are violations of constraints on parameters and variables, and similar problems. A library that does not accept such parameters during development makes programming much faster and significantly less tedious; in a non-debug mode, the error checking assertions are taken out of the code before compilation, so no performance penalty is imposed for production computations.

- *Documentation:* Within the academic sector, too many projects still fail in the long term due to insufficient documentation; at the latest when the initial developers leave the institute, i.e. usually after about five years, a program is destined to die if not properly documented.

The design goals described above need a programming language with a high level of abstraction. We chose `C++` for implementing the library. The main reasons for this were:

- *Availability:* `C++` is widely available and with `gcc` there is a compiler that runs on almost any platform, including personal computers for home use, workstations and supercomputers. Besides this, `C++` is a popular language in nearly all fields of computing, so acquiring skilled programmers is less difficult than with other programming languages.

- *Standardization:* `C++` is a standardized language, so the programmer is able to rely on language features and can write portable programs that are guaranteed to run in the future as well. Furthermore, the standard includes a large library of generic data containers and algorithms which allow to program significantly faster than using most other languages.

- *Data encapsulation:* The complex data structures required by locally refined meshes need to be encapsulated and be made visible in a more structured and readily available form than using the raw data. This is done using wrapper classes that distribute the access to the different parts of the data structures providing a uniform interface to the user. These classes have the advantage to shield the user from the actual data structures as well as from changes therein, allowing optimizations to be implemented without the need to change application programs.

- *Flexible and strong data typing:* `C++` provides, through the concept of templates, the use of generic data types for algorithms and data structures. Still, neither speed nor strong data typing is sacrificed for this. This is an important advantage over many other languages where either no genericity is available, leading to duplication of code (including more possibilities for data type errors and additional effort for coding) or where the type system is relaxed in order to allow genericity, leading to errors which are harder to track down, and slower programs. Genericity is an important part of complex numerical software since the choice of data types often involves a trade-off between accuracy and computing time or memory; this choice has often to be made for each case again, so a library should not settle on one data type (say, double precision) beforehand.

- *Speed:* As opposed to teaching focused languages like `Pascal` and languages for network and interactive applications like `Java`, `C++` was developed with the aim of allowing highly structured software that still is fast. Therefore, it offers features like templates and inline functions, that enable a good compiler to mostly eliminate structural overhead.

- *In-code documentation:* In practice, for a rather small group of developers it is impossible to keep a good technical documentation up to date unless this can happen within the source code itself. With the advent of a standard for the documentation of `Java` programs, there have also appeared several programs to extract documentation

directly from C++ source code. Using these programs, documentation is written directly at the point where modifications occur, making it much easier to keep program and documentation in a matching state.

At present, if printed the deal.II documentation comprises about 800 pages of function and class references, along with several dozens of pages of technical documentation. All information is available on the World Wide Web as well and is updated every night (cf. [2]).

These ingredients of the programming language have enabled us to use a programming model that resembles the style of the C++ standard library and makes use of templates to support several space dimensions at once. These points will be explained in detail below.

## 1.2 Programming model

**Iterators and accessors.** The standard template library (STL) introduced the notion of iterators to C++ from 1993 on. This model, abstracting pointers and, in general, elements of containers has since then gained wide support in the C++ world. For a library making heavy use of the standard container classes, it is therefore natural to offer its data structures in a similar way. In the present context, a triangulation can be considered a container holding points, lines, quadrilaterals, etc. accessible like the elements of a list.

In fact, this mode of addressing elements of a triangulation is a major abstraction, since the data elements making up one of the objects mentioned above are distributed over a number of different arrays and other containers. By providing classes called iterators, programs can be written as if data collections were arrays of data; in particular, these classes offer operators ++ and -- that move the pointer-like variable to the next or previous element, respectively, just like a pointer would behave, and as do the iterator classes of the standard library.

While pointers in the original sense pointed to actual data which is organized in a linear fashion in memory, iterators need not do so. For example iterators may point to the elements of a linked list, which need not have any special order in physical memory apart from the pointers that link the different elements.

In the deal.II library, iterators actually point to no data at all. Dereferenced, they return an object called *accessor* having no data elements itself apart from some numbers identifying the line or quadrilateral it is to represent; it is a collection of functions knowing how to obtain and manipulate data related to that object. A typical function, setting a bit for a quadrilateral in the triangulation that might be used by application programs would then look like the following extract:

```
void QuadAccessor::set_user_flag () const
{
  tria->levels[level]->quads.user_flags[index] = true;
}
```

`level` and `index` denote the address of the quadrilateral represented by this accessor object within the hierarchical triangulation. It is obvious that shielding this multiple dereferencing from the user makes the library much more robust with regard to changes in the internal data structures. Furthermore, it allows us to write programs in a much simpler way than by using the internals directly. They will be much better readable, too.

Using this concept of offering data centralized in an accessor class while still storing it decentralized in many complex data structures, it is possible to use the advantages of more flexible data structures while still having the simplicity of programming applications on top of these; in addition, the user needs not know anything about the actual representation of the data, which may thus be changed at any time as long as the accessor classes are changed accordingly. The actual knowledge of the data structures is restricted to very small parts of the library (a few hundred statements), the rest of the library and all user programs use iterators and accessors to address these data.

**Logical addressing of objects.** The most often used operation in finite element programs is looping over all quadrilaterals within a triangulation of a two dimensional domain and performing some operations on each of them; an example would be to compute the contributions of each quadrilateral to the global system matrix. From this point of view, a triangulation is composed of vertices, lines, quadrilaterals, etc.

Alternatively, it can be regarded as a collection of cells, faces, etc.; we call this the *dual topology*. This view, which depends on the dimension (a cell is a line in one space dimension, a quadrilateral in 2d and a hexahedron in 3d) is more natural to the programming of numerical software since assembling of matrices, computation of error estimators, and so on are done on cells and their faces.

A typical loop, in this case marking all cells for some future operation, therefore looks like this:

```
Triangulation<dim> triangulation;
...                            // triangulate a domain
Triangulation<dim>::cell_iterator cell;
for (cell=triangulation.begin();
     cell != triangulation.end();
     ++cell)
  {
    cell->set_user_flag ();
  };
```

Remember that if `cell` is dereferenced, we obtain an accessor which has the member function shown above to perform the wanted operation.

The `deal.II` library offers both ways of addressing elements of triangulations. While the first one centered on the dimension of an object is most often used in the interior of the library, the second way, focused on the dimension of an object relative to what a cell constitutes, is most helpful for numerical algorithms and in fact allows to write programs

and algorithms in a dimension independent way. Operations work cell by cell, choosing finite element shape functions, quadrature rules and the like according to the dimension of the cell. All the application programs presently implemented with `deal.II` exclusively use the dual topology instead of the primal one.

**Dimension independent programming.** In this approach to programming, the program is actually formed when the compiler associates a concrete data type, such as a *quadrilateral*, to an abstract data type like a *cell*. This is done using template manipulations as provided by `C++`. It allows to write a function or algorithm in a way that does not depend on the dimension by using *logical* data types like cells or faces, which are parameterized aliases (i.e. `typedef`s) to concrete data types; the mapping between these data types depends on the dimension (which is a template parameter) for which a program is presently compiled. For example, in the piece of code shown above, if the template parameter `dim` equals one, two, or three, the objects marked are lines, quadrilaterals, and cells, respectively. The code to switch these data types looks roughly like this:

```
class TriaDimensionInfo<2>
{
  typedef quad_iterator                        cell_iterator;
  typedef line_iterator                        face_iterator;
  //...
};

class TriaDimensionInfo<3>
{
  typedef hex_iterator                         cell_iterator;
  typedef quad_iterator                        face_iterator;
  //...
};


template <int dim>
class Triangulation : public TriaDimensionInfo<dim>
{
  typedef TriaDimensionInfo<dim>::cell_iterator cell_iterator;
  typedef TriaDimensionInfo<dim>::face_iterator face_iterator;
  //...
};
```

By preferring a template parameter over preprocessor variables, it is possible to retain a greater amount of type safety as well as the possibility to use parts of programs with different space dimensions together at the same time.

If an algorithm is not dimension independent, it is possible to specialize it for some or all dimensions where its form deviates from the general template. This is typically the case

in one space dimension, where algorithms often work a bit different, since no proper faces exist (we do not consider points proper objects in the same sense as lines, quadrilaterals, etc. since their lack of extension in any direction does not permit to define shape functions on them).

Being called from some dimension independent part of the code, the compiler automatically figures out which version of a function to call; dimension independent and dimension dependent code therefore work together smoothly without the intervention of the programmer. This approach has proven useful when several programs originally written for two-dimensional computations ran by mere recompilation in three space dimension as well when the three dimensional support within the library became available.

## 1.3 History

As a last note before we start with the actual description of the library, we want to give a brief historical overview of the evolution of the project. The following are some of the more important milestones that should be mentioned:

*1991-1992:* Needing finite element software for their diploma theses, Guido Kanschat and Franz-Theo Suttmeier start to develop finite element codes, first in PASCAL, later in `C++`. Working on quasi-regular solutions to variational systems, the need for grid adaption becomes obvious. A finite element code for solving two-dimensional problems in plasticity and quasi-linear elliptic systems evolves [10].

*Since 1993:* DEAL, short for *D*ifferential *E*quations *A*nalysis *L*ibrary, is developed by Guido Kanschat and Franz-Theo Suttmeier. `C++` becomes the language of choice for the reasons mentioned in the introduction and because it is the only advanced programming language portable to a T805 parallel computer.

*1995-1996:* Roland Becker joins and the implementation of the multigrid method allows fast solution of PDE problems. The concept of a posteriori error estimates based on dual solutions is developed and used in the theses of the three programmers [4, 9, 16] and the article [5]. By this time, the library had grown quite complex and no documentation was available. Also, some of the structures had turned out too complicated to allow further development.

*End of 1997:* At the time that the first author of this paper was about to start working on his thesis, it was decided that it was in time to do a major redesign of the library. The urge for other than $Q_1$ finite element spaces and the code reduplication in error estimators demanded for a more flexible structure, while mixed triangular/quadrilateral grids were not used anymore. We decided to rewrite the library from scratch, then called `deal.II`.

Design and implementation of the core of the library, i.e. handling of grids and degrees of freedom, was done entirely by W. Bangerth, applying the knowledge accumulated in the use of the predecessor library, DEAL.

*Since 1998:* Many tools for numerical simulations are added. This includes, among others, output for different visualization tools, discontinuous elements, finite elements for systems and mixed discretizations, and grid transfer operators for time-dependent problems. Multi-level algorithms are about to be finished by now.

At present, the `deal.II` library is maintained by W. Bangerth and G. Kanschat. It is used by several postdoctoral scientists and graduate students at the Institute of Applied Mathematics and several undergraduate students; it is also used in teaching at the universities of Heidelberg and Minnesota.

# 2 Grid handling

Since the stated goals of the library included adaptive grids that are easy to program and should still be fast, hierarchical grids were the only choice possible. By hierarchical we here mean that the structures of the grid are described hierarchically (points, lines, quadrilaterals, ...) and that refinement has to be made hierarchically as well, as opposed to unstructured grids. The first point, hierarchical description, serves the simplicity of programming, while the second, hierarchical refinement, allows to use fast algorithms for grid refinement and in the numerics, such as multigrid solvers. These two concepts are outlined in the following subsections.

## 2.1 Hierarchical cell representation

As has been outlined above, one of the design criteria was to focus on quadrilaterals in two, and hexahedra in three space dimensions. A triangulation is therefore made up of the objects in Figure 1. If the space dimension is lower than three, this sequence is truncated;

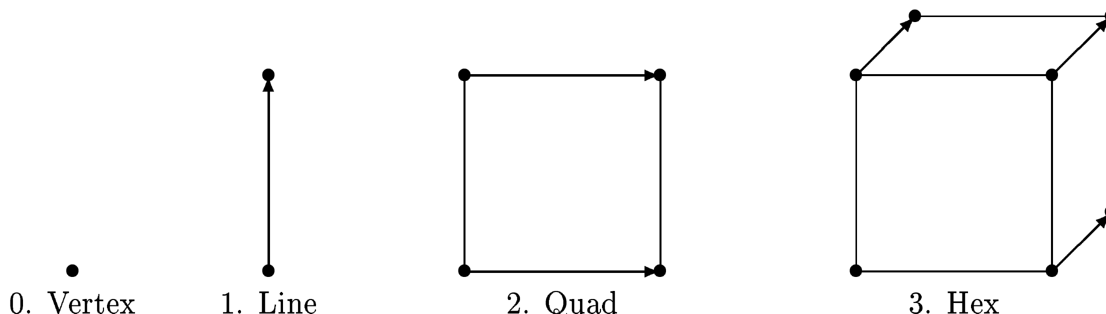


Figure 1: *Topological constituents of a mesh*

if anyone chooses to implement a finite element code in more than three space dimensions (e.g. for general relativity), the sequence may be prolonged.

Each of the objects can be made up of a number of objects of lower level. They are thus hierarchic and one only needs to store pointers to four lines to characterize a quadrilateral, which are themselves characterized by two vertices each. A hexahedron is characterized by six quadrilaterals which are themselves characterized by four lines each and so on.

Storing objects this way has advantages over the other possibility, i.e. storing the vertices only, because it is possible to do computations on faces if they exist as independent objects. This, though obviously also possible if only vertices are stored, makes the evaluation of jump terms (in error estimators or using discontinuous finite elements), the handling of hanging nodes and many more things much simpler, at the expense of a slightly increased memory consumption.

## 2.2 Hierarchical grid refinement

The other hierarchical property of grids that is applied in `deal.II` is hierarchical refinement. At present, there are two concurrent approaches to adaptive meshes:

- *Unstructured meshes:* Using this approach, after computing a refinement indicator, a mesh is created according to this criterion that is not based on the previous grid at all. This can, for example, be done by scattering points onto the domain where the density of points is determined by the error indicator or other criteria. The new grid is now generated using, for example, the Voronoi cells of the point cloud.

  It should be noted that this approach suffers from the expensive computations required for the generation of totally unrelated grids, but is able to produce meshes with smooth transitions of the mesh size parameter $h(x)$.

- *Structured meshes:* Refining those cells in an existing triangulation with the largest refinement indicator, yields meshes that are hierarchical because every cell, unless belonging to the coarsest mesh, has a mother cell and may be refined further. Usually, the refinement of a cell is done in a way that does not deteriorate the aspect ratio of the cells, i.e. refinement into four congruent triangles for triangular meshes or splitting quadrilaterals into four children. In three space dimensions, the subdivision of tetrahedra may be done using several different possibilities, while the subdivision of a hexahedron is canonical, if no anisotropic refinement is required. This way of constructing adaptive grids was considered already in [15].

  The problem with this approach is that, refining one cell, there remains an unbalanced vertex on each side if the respective neighbor is not refined as well. There are basically two strategies to handle these: Using special refinement rules for the neighbor to eliminate these hanging nodes (e.g. red-green refinement of triangles) or allowing for hanging nodes and treat them in a special way to ensure continuity of finite element functions at these points (e.g. by formulating the continuity requirement as a constraint and inserting this into the matrices and vectors).

  It should be noted that the grids produced using this approach usually have relatively steep gradients at the boundaries between two regions of different refinement depths, but refinement can be made very fast. Furthermore, transfer of data between meshes is highly accurate and fast.

Since the additional information in structured meshes can be favorably used in multigrid algorithms as well as in the fast generation of new meshes from a given one, we chose the second approach. Furthermore, because the generation of red-green-like refinement rules for quadrilaterals is relatively complex without recourse to triangles and because any of these special refinements of some elements destroys the regularity in the data structures, we chose to impose the compatibility conditions at interfaces of cells of different refinement depths as constraints to the system of equations. It should be noted that these constraints need not be incorporated using a Lagrangian multiplier (which would lead to saddle point problems), but can be inserted into the matrix and right hand side, thus retaining the properties of the matrices (cf. Section 3.3). In particular, symmetric and positive definite matrices retain these properties.

Using the described approach, all lines, quadrilaterals, etc. are refined in a uniform fashion thus allowing for fast algorithms exploiting this structure. For these, multigrid algorithms based on hierarchical bases come to mind. However, the structure is also particularly suited for time dependent problems with grids that may differ between any two time steps and where the transfer of the old solution to the new grid can only be done efficiently if the two grids are related in some way; such a relationship can be obtained by using the same coarse grid with different refinement depths on the different time steps. In that case, integration of the old solution with the new test functions can even be done exactly, leading to both higher accuracy and dramatic speed improvements compared with unstructured mesh approaches.

# 3 Finite element spaces

In finite element theory, the discrete function spaces are represented by the vector space of node values. Each node value denotes one degree of freedom in this vector space. The functionals defining the degrees of freedom, for example the interpolation points of Lagrangian elements or face integrals for some nonconforming elements, may be located on any of the basic topological objects of a mesh, as shown in Figure 1. The `FiniteElement` class describing a certain finite element states the objects on which these functionals are defined. For neighboring cells, these nodes may coincide, in which case they are identified.

In case of local refinement ($h$ or $p$), an interface between two cells might carry different node values, which we call *hanging nodes*. Then, additional compatibility conditions have to be imposed numerically, as we will explain below in Section 3.3.

## 3.1 Finite element objects

A finite element space can be viewed as the function space spanned by a number of relatively simple shape functions. These basis functions are chosen such that they have a small support, in general at most as many cells as may be adjacent at any one vertex in the triangulation. Furthermore they are usually defined on each cell within their domain of support separately, with some compatibility condition for the interfaces between these

subdomains; compatibility conditions may be continuity along a face between two cells (for $H^1$-conforming elements), equality of the mean value of the shape function on both sides of a face (Crouzeix-Raviart [8], Rannacher-Turek [13]) or its normal component (Raviart-Thomas [14], Brezzi-Douglas-Marini [7]) or some other requirements.

Due to these properties, it is convenient for the implementation to consider a finite element space as consisting of a set of shape functions defined on a cell and compatibility conditions at the boundaries of the cell; this viewpoint stresses a purely local description of the space which enables us to do computations on each cell separately, without looking around, in most cases. A further abstraction is possible in some cases, where the shape functions can be computed on the unit cell and only afterwards need to be transformed to the real space cell; this is possible for interpolation elements, but not for all elements involving line integrals, normal derivatives, etc.

With these considerations in mind, the representation of a finite element in `deal.II` is a class that provides the shape functions and its derivatives on the unit cell (if possible, otherwise on the real space cell), and the transformation from unit to real cell and its derivatives. It furthermore offers some information to the degree of freedom management class, among which is where degrees of freedom are located (i.e. on vertices, line, quadrilaterals, etc) and finally the information necessary to impose the compatibility of the different parts of a shape function on adjacent elements.

However, looking at actual finite element programs, one notes that most of the information listed above is not necessary to application programs. While compatibility and transformations are of interest to the internal functions of the library, application programs almost always are only interested in the restriction of a finite element field or shape function to a real space cell. Here, however, it is important to note that one usually is not interested in the finite element as a continuous function, but only at a specified set of points, for example in quadrature points located on a cell or a face. Since codes for finite element applications usually do not use the analytical representation of the shape functions (which can be rather complicated on real space cells), computations are done using quadrature, though in some cases with quadrature formulae which yield the same result as exact integration.

In order to make access to actual finite elements more efficient and structured, `deal.II` therefore offers an abstraction of a finite element restricted to a set of points on a cell. This interface is provided by the `FEValues` class described in the next subsection. The actual finite element class is hardly ever accessed directly by application programs and indeed by none of the existing applications built upon `deal.II`.

## 3.2 Computation of shape function values

Integrating and assembling matrices and right hand sides can consume a considerable amount of time during the execution of a finite element program. This is especially true in non-linear applications, where the process of solving the linear system is not necessarily predominant anymore. Therefore, the access to finite element shape functions is not only a question of defining a well structured interface, it is also a matter of run-time efficiency.

Analyzing finite element software, we observe that shape functions are always used in the same context: integration on a grid cell. This means, that shape functions are always used in combination with a quadrature rule. We exploit this relationship by introducing a special class `FEValues` combining quadrature and shape functions; in fact it can be considered to be the restriction of the trial space on the whole mesh to the quadrature points of a given quadrature rule applied to a single element of the triangulation. An object of this class will compute all values of the shape functions at the appropriate time and store them in arrays. These values come basically in two categories:

1. Values only depending on the quadrature point on the unit cell. These are for instance the function values of standard Lagrangian elements.

2. Values depending on the actual grid cell. In this class are for example values of derivatives, since they have to be transformed by the tangent mapping of the transformation between unit cell and actual cell, but also the actual locations of the quadrature points in real space.

It should be remarked here that the category a certain value belongs to is dependent on the type of finite element. While for standard Lagrangian elements, values of the shape functions belong to the first, in the case of Raviart-Thomas type elements they are of the second kind.

The two kinds of behavior are reflected in the fact that the tables of `FEValues` are built at different times. Tables of the first category are already filled upon construction, where `FEValues` objects obtain information about quadrature points and finite element shape functions. This is usually done before a loop over all cells starts. The second type of values is computed in a function that has to be called within the loop for each grid cell before any evaluations are done.

It is up to the interaction of `FiniteElement` and `FEValues` to do these computations as efficiently as possible. The user should give some hints though: `FEValues` takes a group of flags, telling it which fields need to be computed on each cell; if second derivatives are not used in a loop, they will not be computed on each element.

## 3.3 Hanging nodes

As briefly mentioned above, we obtain hanging nodes at interfaces between cells of differing refinement,[1] see Figure 2. The degrees of freedom on the refined side of a face (for example the left side of the face with end points $P_1$ and $P_2$ in the figure) are not all matched by degrees of freedom on the coarse side, so in order to guarantee continuity[2] of the finite element space along this face we need to impose additional constraints. In order to illustrate

[1] We get a similar problem if we have different finite elements on two adjacent elements. Since the treatment of this problem is along the same lines as for $h$-refinement, we do not make explicit reference to $p$-refinement in the following.

[2] For nonconforming finite element spaces, the continuity requirement has to be substituted by a generalized compatibility condition, cf. [11].

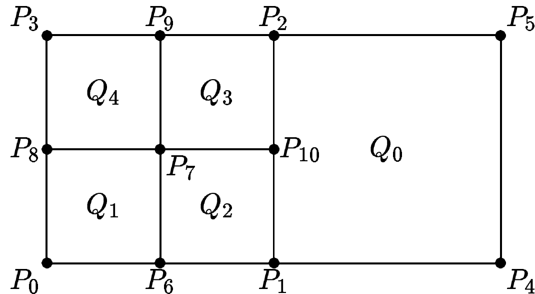

Figure 2: *A simple grid with a hanging node.*

the process, let us assume that $V$ be the original trial space including the continuity requirement on its members, while $\tilde{V}$ be a finite element space that is constructed from the union of the same shape functions as in $V$, but restricted to each cell separately; $\tilde{V}$ therefore may contain functions that are discontinuous along refined faces. While hanging nodes carry degrees of freedom in $\tilde{V}$, they do not do so in $V$.

Since doing computations with functions from $\tilde{V}$ is much simpler (because it can be done locally on each cell without taking care that there may be coarser or finer cells around, and because we would like to associate degrees of freedom to all vertices and lines, irrespective of their neighborhood), we would like to use $\tilde{V}$ as long as possible and use the constraints that $V$ has over $\tilde{V}$ as late as possible in the solution of the problem. This can be done in the following way (cf. [12, 9]; [15] also shows how to do this, but the algorithms are significantly more complex since more than one irregular node is allowed per face).

Assume we had to solve the problem: *find $u \in V$, such that for all $v \in V$*

$$a(v,u) = (v,f)$$

with a bilinear form $a(\cdot,\cdot)$ and the usual $L_2$ scalar product $(\cdot,\cdot)$. We would like to state this problem in $\tilde{V}$ instead of $V$. To do so, we note that each function in $\tilde{V}$ can be expanded into the basis functions of $\tilde{V}$, which we will denote by $\tilde{\varphi}_i$, as $\tilde{u} = \sum_i \tilde{u}_i \tilde{\varphi}_i$. Likewise, $V$ is decomposable like $u = \sum_i u_i \varphi_i$. Now, because $V$ is a subspace of $\tilde{V}$ (it is $\tilde{V}$ plus some constraints) every function $\varphi_i$ can be written as a sum of functions from $\tilde{V}$: $\varphi_i = \sum_{j=1}^{\tilde{N}} C_{ij} \tilde{\varphi}_i$, which defines a matrix $C_{ij}$. For example, for bilinear shape functions the function $\varphi_1$ associated with node $P_1$ in Figure 2 can be written as $\varphi_1 = \tilde{\varphi}_1 + \frac{1}{2}\tilde{\varphi}_{10}$; the support of these three shape functions are $\mathrm{supp}(\varphi_1) = Q_0 \cup Q_2 \cup Q_2$, $\mathrm{supp}(\tilde{\varphi}_1) = Q_0 \cup Q_2$, and $\mathrm{supp}(\tilde{\varphi}_{10}) = Q_2 \cup Q_3$, respectively. $\tilde{\varphi}_1$ is discontinuous along the line $P_1$–$P_{10}$, while $\tilde{\varphi}_{10}$ is discontinuous along the whole line $P_1$–$P_2$, because $\tilde{\varphi}_{10}|_{Q_0} \equiv 0$.

This representation of basis functions implies that a function $u \in V$ must be representable as $u = C\tilde{u}$ with some function $\tilde{u} \in \tilde{V}$, where in the latter representation we identified the function $u$ by the vector of its nodal values. For the grid in Figure 2 and bilinear elements, the matrix $C$ has the following form:

$$C \in R^{10\times 11}, \qquad C_{ii} = 1, \qquad C_{1,10} = C_{2,10} = \tfrac{1}{2}.$$

All other entries are zero. Since most of the columns (notably those belonging to non-constrained nodes) consist only of the diagonal entry, we only need store those columns that deviate from this form. Note that for simplicity we have here numbered the only constrained node such that it is the last one. This way, nodal numbers for $V$ can be chosen to be the same as for $\tilde{V}$, simply dropping the numbers of constrained nodes following those of unconstrained ones. If we had not chosen this numbering, we would need to permute the rows in $C$, which would move the entries in rows of unconstrained nodes to nondiagonal places.

We can now restate the problem as follows: *find* $\tilde{u} \in \tilde{V}$ *such that for all* $\tilde{v} \in \tilde{V}$

$$\tilde{v}_i C_{ij} a(\tilde{\varphi}_j, \tilde{\varphi}_k) C_{lk} \tilde{u}_k = \tilde{v}_i C_{ij} (\varphi_j, f),$$

which amounts to solving the linear system of equations

$$C\tilde{A}C^T \tilde{u} = C\tilde{f}.$$

It should be noted that $\tilde{A}$ is assembled as usual, i.e. cell-wise without the need to look at the neighboring cells, and that it is relatively simple to generate $C\tilde{A}C^T$ in-place, i.e. without the need for another matrix where we copy the result into. In particular, the sparsity pattern of $\tilde{A}$ can be obtained from that of $A$ by filling in some additional places that can be computed beforehand. The same holds for the right hand side. These properties result from the fact that each hanging node may only be constrained once, which however limits the difference in refinement to one level of cells in three space dimensions that are adjacent at one edge only.

In order to see how this property-conservation can be obtained, we use another matrix $\hat{C}$ instead of $C$, where we add additional rows for each constrained node such that it becomes square. In doing so, there is no need to renumber the degrees of freedom in a way as to order constrained nodes to the end. The additional rows in $\tilde{C}$ are set to zero only and the diagonal elements of all rows apart from the newly added ones are equal to one, which allows for very efficient storage mechanisms. Multiplying $\tilde{A}$ by $\hat{C}^T$ from the right, only multiplication with rows in $\hat{C}$ have to be taken care of that contain either only zeroes (resulting in a blanked out column in $\tilde{A}\hat{C}^T$ for each constrained node) or that contain entries other than only the diagonal entry, which are nodes that constrain another one (resulting in multiples of some columns being added to other columns). The multiplication with $\hat{C}$ from the left can be done along the same lines.

In practice, due to the special structure of $\hat{C}$, multiplication by $\hat{C}$ from the left and $\hat{C}^T$ from the right can be done at the same time and with writing the results into the same matrix where previously $\tilde{A}$ was stored. We call this process *condensation.* Lines and columns belonging to constrained nodes are filled with zeroes, effectively eliminating these degrees of freedom from the system of equations in the same way as if we used the original matrix $C$.

# 4 Iterative solvers

Solution methods for finite equations are quite generally iterative. This is true for non-linear equations, where we use different variations of Newton's method, and linear equations. Even time-stepping methods fit into this framework.

Since solution methods are used in many different contexts, but are usually time-critical, they should be implemented as general as is possible without loosing efficiency. Therefore, linear solvers and matrix classes are extracted into a separate library called LAC, short for *L*inear *A*lgebra *C*lasses.

## 4.1 Iterative linear solvers of LAC

Solvers in `LAC` are supposed to be basic multi-purpose tools. In particular, the requirements for matrix and vector classes involved should be as low as possible. In LAC we chose to not use abstract base classes to state these interface requirements. Instead, matrix and vector are template arguments to the solvers, e.g. `SolverCG<Matrix, Vector>`.

### 4.1.1 Requirements on template parameters

A minimal interface for a matrix class used for `LAC` Solvers is

```
class Matrix
{
  public:
    void   vmult    (Vector& dst, const Vector& src) const;

    double residual (Vector& dst, const Vector& src,
                     const Vector& right_hand_side) const;
};
```

Here, `vmult` for a matrix $A$ should perform $v_{dst} = Av_{src}$, while `residual` performs $v_{dst} = v_{rhs} - Av_{src}$ and returns the Euclidean norm of the result.

The requirements for a preconditioner are even simpler:

```
class Precondition
{
  public:
    void operator() (Vector& dst, const Vector& src) const;
};
```

Unfortunately, the interface of the vector class is more complex. To avoid a lot of unnecessary loops, there are different redundant scaled vector additions used in the iterative methods.

#### 4.1.2 Administrative classes

The LAC-solvers do not check for the stopping criterion themselves. Instead, they compute the value used by this criterion, e.g. the norm of the residual, and hand it over to an object of a class called `SolverControl`. This class now decides, whether the iteration

- failed, because for example the maximum number of iteration steps is exceeded;
- terminates successfully, because the aimed stopping criterion was reached;
- continues, until one of the first two conditions is reached.

The provided class `SolverControl` has a standard version of this virtual function `check` built in. It can be replaced by a derived class, for instance to balance convergence of the linear solver with some outer iteration.

Most solvers need auxiliary vectors. Since it might be advisable in many cases that these vectors are pre-allocated and retrieved from a special memory, the class `VectorMemory` was introduced. It is an abstract base class for a memory handler. If more sophisticated allocation methods are needed, a derived class can implement a vector pool, where vectors are not deallocated after use, e.g. to avoid fragmentation of heap memory or to speed up allocation.

# 5 Example applications

In this section, we give a brief overview of some applications already implemented with `deal.II`, to show the range of applicability and to give an idea of what our motives were to write such a library.

## 5.1 Conservation Laws

The solution of the nonlinear Burgers equation

$$u_t + uu_x = 0$$

with given initial and boundary conditions in space and time often yields discontinuous solutions, usually even if the data are smooth. Resolving these discontinuities without recourse to a globally refined grid is a major challenge, especially in view of the fact that the same problems with discontinuous solutions occur with other and more important equations as well, such as the Euler equations of inviscid, eventually transonic flow.

The examples in Figure 3 of computations with locally refined grids for these equations were done by R. Hartmann. The left and middle picture show solutions of the nonlinear Burgers equation with two merging shocks on a coarse grid and a refined one; bilinear discontinuous elements were used for these computations. On the right, the Euler equations for the shock tube problem were solved; shock, contact discontinuity and rarefaction wave can clearly be seen.

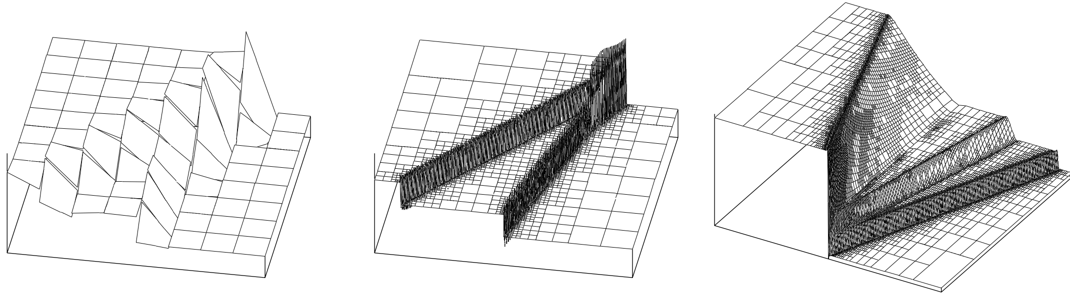

Figure 3: *Examples of solutions of conservation laws. Computation: R. Hartmann.*

## 5.2 Parameter estimation

In many applications in engineering, one wants to determine the value of a parameter from measurements. One example would be the estimation of the elasticity parameters within a body from measurements of the displacement under a specified load. Another similar problem would be the estimation of the conductivity in a porous medium from measurements of the hydraulic head. A prototypic elliptic equation would be

$$-\nabla \cdot (q \nabla u) = f$$

with some boundary conditions. The coefficient $q(x)$ is to be determined by minimizing the difference between the solution of this equation for a given coefficient, $u_q(x)$, and a measurement $u_{meas}$, by variation of $q$.

Problems of this kind usually are ill-posed, i.e. they do not possess a continuous dependence of the optimal parameter $q^*$ from the measurement $u_{meas}$, which makes them rather difficult to solve. Furthermore, they are strongly nonlinear and usually require many solutions of the forward problem (i.e. the elliptic equation for a fixed coefficient) to determine the coefficient itself, which makes the efficient solution of the forward problem indispensable.

In Figure 4 we show two examples of estimated parameters along with the exact values. In order to reduce the degree of ill-posedness and to reduce the number of unknowns, the discretization of the parameter $q$ was performed on a coarser grid than the state variable $u$. In fact, the discretization was done on the same grid but we posed additional constraints such that each patch of four cells can be written as a shape function on the next coarser grid; these constraints can then be inserted into the system matrix just as was done for hanging nodes. The discretization of the parameter was done with discontinuous elements of one degree lower than the polynomial degree with which the state variable was discretized.

The top row of the picture shows the approximation of a continuous coefficient on a square domain; the state variable was discretized with bilinear elements, while the parameter was discretized with discontinuous constant elements. In the bottom row, a discontinuous coefficient was to be estimated on a circular domain of which one quarter is shown;

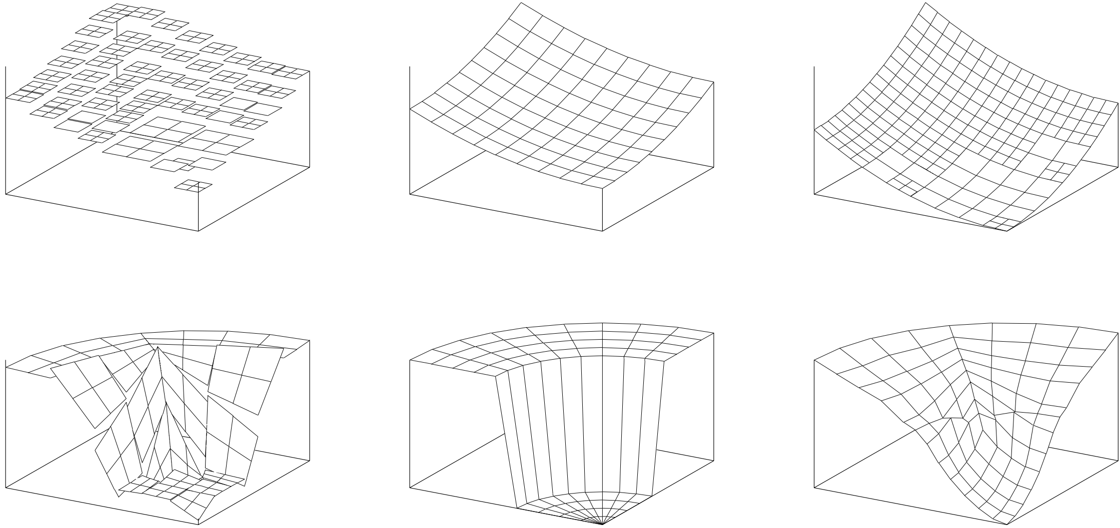

Figure 4: *Estimated (left) and true coefficients (middle) in a parameter estimation problem. The state variable approximating the measurement is shown in the right column. Top: smooth true coefficient, bottom: discontinuous true coefficient.*

state and parameter variables used bi-quadratic continuous elements and discontinuous bilinear elements, respectively. No adaptive grid refinement was performed for the second case.

## 5.3 Wave equation

The acoustic wave equation

$$\rho u_{tt} - \nabla \cdot (a \nabla u) = 0,$$

associated with suitable initial and boundary conditions occurs in a variety of places in physics, among which are water waves, acoustic sound waves in gaseous media, electromagnetics and many other fields. Usually, $u$ is the deviation from the state of rest and $\rho$ and $a$ denote density and stiffness coefficient. The solutions of this equation often share the feature that they form waves traveling through the domain. Often the region where waves presently are is significantly smaller than the whole domain, so adaptivity should be able to reduce the number of cells needed for the solution of this equation by a noticeable amount, because coarse grids can be used wherever no waves are presently.

On the other hand, if we are not interested in the whole solution but only in parts of it (say that part traveling in one direction, or an integral over part of the domain at the end time), the domain of influence of the region where we evaluate itself also often is significantly smaller than the whole region, due to the finite speed with which waves spread. Here again a reasonable reduction of the required number of cells can be obtained.

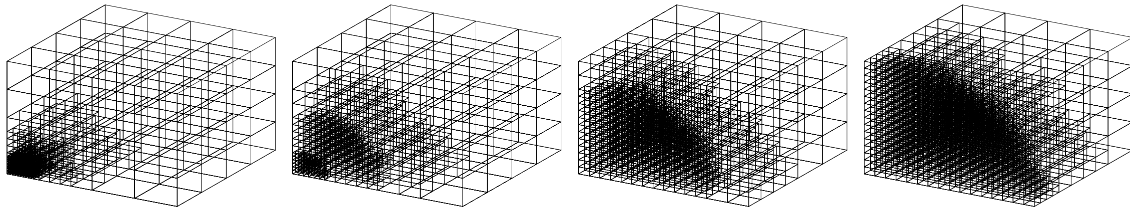

Figure 5: *Automatically generated grids tracking an acoustic wave traveling outwards in a three-dimensional domain.*

Typical grids showing this are presented in the following. They were obtained using the program described in [1, 3]. In Figure 5, a wave traveling outward from the lower left corner of the domain is shown. We used a simple error indicator to track the spreading of the wave to adjust the computational meshes accordingly; it can be seen that rather coarse cells were used where the solution is smooth.

### 5.4 Boundary approximation

In real-life applications such as air flow around a plane or scattering of waves by objects, the computational boundaries often are rather complex. In particular, a coarse grid that already mostly features the contours of the objects usually has many more cells than is acceptable for a coarse grid. One way to avoid this problem is to not approximate the boundary with the grid and pose the boundary values on the boundary of the grid, but to use a mesh that is not necessarily adapted to the physical boundaries and pose boundary values in the weak form of the equation. This allows, among other advantages, to use more regular meshes and a coarse grid with less cells, which enables us to use efficient multigrid algorithms.

Figure 6 shows examples of solutions to Poisson's equation using Neumann boundary conditions on an embedded boundary curve. The interior of the circle is not part of the domain on which the problem is posed and the solution is set to zero there.

## References


[1] W. Bangerth. Adaptive Finite-Elemente-Methoden zur Lösung der Wellengleichung mit Anwendung in der Physik der Sonne. Diplomarbeit, Institut für Angewandte Mathematik, Universität Heidelberg, 1998.

[2] W. Bangerth and G. Kanschat. ***deal.II*** *Homepage.* `http://gaia.iwr.uni-heidelberg.de/~deal/`.

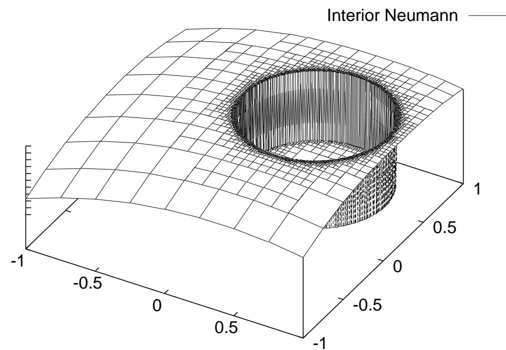


Figure 6: *Approximation of a circular domain by inclusion of boundary values into the weak formulation of the equation.*

[3] W. Bangerth and R. Rannacher. Finite element approximation of the acoustic wave equation: Error control and mesh adaptation. *East–West J. Num. Math.*, (4), 1999. submitted.

[4] R. Becker. *An Adaptive Finite Element Method for the Incompressible Navier-Stokes Equations on Time-dependent Domains.* Dissertation, Universität Heidelberg, 1995.

[5] R. Becker and R. Rannacher. Weighted a posteriori error control in FE methods. In *ENUMATH 95*, Paris, September 1995. in [6].

[6] H. G. Bock, F. Brezzi, R. Glowinsky, G. Kanschat, Y. A. Kuznetsov, J. Périaux, and R. Rannacher, editors. *ENUMATH 97, Proceedings of the 2nd Conference on Numerical Mathematics and Advanced Applications*, Singapore, 1998. World Scientific.

[7] F. Brezzi and M. Fortin. *Mixed and Hybrid Finite Element Methods.* Springer, 1991.

[8] M. Crouzeix and P. Raviart. Conforming and nonconforming finite element methods for solving the stationary stokes equations, part I. *RAIRO*, R–3:33–76, 1973.

[9] G. Kanschat. *Parallel and Adaptive Galerkin Methods for Radiative Transfer Problems.* Dissertation, Universität Heidelberg, 1996.

[10] G. Kanschat and F.-T. Suttmeier. Datenstrukturen für die Methode der finiten Elemente. unpublished, Bonn-Venusberg, 1992.

[11] G. Kanschat and F.-T. Suttmeier. A posteriori error estimates for nonconforming finite element schemes. *Calcolo*, 36(3):129–141, 1999. to appear.

[12] G. Kozlovsky. Solving partial differential equations using recursive grids. *Appl. Numer. Math.*, 14:165–181, 1994.

[13] R. Rannacher and S. Turek. Simple nonconforming quadrilateral stokes element. *Num. Meth. PDE*, 8(2):97–111, March 1992.

[14] P. Raviart and J.-M. Thomas. A mixed finite element method for second order elliptic problems. In I. Galligani and E. Magenes, editors, *Mathematical Aspects of the Finite Element Method*, pages 292–315, New York, 1977. Springer.

[15] W. C. Rheinboldt and C. K. Mesztenyi. On a data structure for adaptive finite element mesh refinements. *ACM Trans. Math. Software*, 6:166–187, 1980.

[16] F.-T. Suttmeier. *Adaptive Finite Element Approximation of Problems in Elasto-Plasticity Theory.* Dissertation, Universität Heidelberg, 1996.