Data Structures and Requirements for hp Finite Element Software

W. BANGERTH
Texas A&M University
and
O. KAYSER-HEROLD
Harvard School of Public Health

Finite element methods approximate solutions of partial differential equations by restricting the problem to a finite dimensional function space. In hp adaptive finite element methods, one defines these discrete spaces by choosing different polynomial degrees for the shape functions defined on a locally refined mesh.

Although this basic idea is quite simple, its implementation in algorithms and data structures is challenging. It has apparently not been documented in the literature in its most general form. Rather, most existing implementations appear to be for special combinations of finite elements, or for discontinuous Galerkin methods.

In this paper, we discuss generic data structures and algorithms used in the implementation of hp methods for arbitrary elements, and the complications and pitfalls one encounters. As a consequence, we list the information a description of a finite element has to provide to the generic algorithms for it to be used in an hp context. We support our claim that our reference implementation is efficient using numerical examples in 2d and 3d, and demonstrate that the hp specific parts of the program do not dominate the total computing time. This reference implementation is also made available as part of the Open Source deal.II finite element library.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: Finite element software—*data structures*; hp *finite element methods*; G.1.8 [**Numerical Analysis**]: Partial Differential Equations—*finite element method.*

General Terms: Algorithms, Design

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

1. INTRODUCTION

The hp finite element method was proposed more than two decades ago by Babuška and Guo [Babuška 1981; Guo and Babuška 1986a; 1986b] as an alternative to either (i) mesh refinement (i.e. decreasing the mesh parameter h in a finite element computation) or (ii) increasing the polynomial degree p used for shape functions. It is based on the observation that increasing the polynomial degree of the shape functions reduces the approximation error if the solution is sufficiently smooth. On the other hand, it is well known [Ciarlet 1978; Gilbarg and Trudinger 1983] that even for the generally well-behaved class of elliptic problems, higher degrees of regularity can not be guaranteed in the vicinity of boundaries,

Author's addresses: W. Bangerth, Department of Mathematics, Texas A&M University, College Station, TX 77843, USA; O. Kayser-Herold, Department of Environmental Health, Harvard School of Public Health, Boston, MA 02115, USA.

corners, or where coefficients are discontinuous; consequently, the approximation can not be improved in these areas by increasing the polynomial degree p but only by refining the mesh, i.e. by reducing the mesh size h. These differing means to reduce the error have led to the notion of hp finite elements, where the approximating finite element spaces are adapted to have a high polynomial degree p wherever the solution is sufficiently smooth, while the mesh width h is reduced at places wherever the solution lacks regularity. It was already realized in the first papers on this method that hp finite elements can be a powerful tool that can guarantee that the error is reduced not only with some negative power of the number of degrees of freedom, but in fact exponentially.

Since then, some 25 years have passed and while hp finite element methods are subject of many investigations in the mathematical literature, they are hardly ever used outside academia, and only rarely even in academic investigations on finite element methods such as on error estimates, discretization schemes, or solvers. It is a common perception that this can be attributed to two major factors: (i) There is no simple and widely accepted a posteriori indicator applicable to an already computed solution that would tell us whether we should refine any given cell of a finite element mesh or increase the polynomial degree of the shape functions defined on it. This is at least true for continuous elements, though there are certainly ideas for discontinuous elements, see [Houston et al. 2008; Houston et al. 2007; Ainsworth and Senior 1997] and in particular [Houston and Süli 2005] and the references cited therein. The major obstacle here is not the estimation of the error on this cell; rather, it is to decide whether h-refinement or p-refinement is preferable. (ii) The hp finite element method is hard to implement. In fact, a commonly heard myth in the field holds that it is "orders of magnitude harder to implement" than simple h adaptivity. This factor, in conjunction with the fact that most software used in mathematical research is homegrown, rarely passed on between generations of students, and therefore of limited complexity, has certainly contributed to the slow adoption of this method.

In order to improve the situation regarding the second point above, we have undertaken the task of thoroughly implementing support for hp finite element methods in the freely available and widely used Open Source finite element library deal.II [Bangerth et al. 2008; 2007] and to thereby making it available as a simple to use research tool to the wider scientific community. deal.II is a library that supports a wide variety of finite element types in 1d, 2d (on quadrilaterals) and 3d (on hexahedra), including the usual Lagrange elements, various discontinuous elements, Raviart-Thomas elements [Brezzi and Fortin 1991], Nedelec elements [Nedelec 1980], and combinations of these for coupled problems with several solution variables.

There are currently not many implementations of the hp finite element method that are accessible to others in some form. Of these, the codes by Leszek Demkowicz [Demkowicz 2006] and Concepts [Frauenfelder and Lage 2002] may be among the best known and in addition to most other libraries also include fully anisotropic refinement. Others, such as for example libMesh [Kirk et al. 2006] and hpGEM [Pesch et al. 2007] claim to be in the process of implementing the method, but the current state of their software appears unclear. More importantly, most of these libraries seem to focus on implementing the method for one particular family of elements, most frequently either hierarchical Lagrange elements (for continuous ansatz spaces) or for the much simpler case of discontinuous spaces.

In contrast, we wanted to implement hp support as general as possible, so that it can be applied to all the elements supported by deal.II, i.e. including continuous and discontinuous

ones, without having to change again the parts of the library that are agnostic to what finite element is currently being used. For example, the main classes in deal.II only require to know how many degrees of freedom a finite element has on each vertex, edge, face, or cell, to allocate the necessary data. Consequently, the aim of our study was to find out what additional data finite element classes have to provide to allow the element-independent code to deal with the hp situation.

This led to a certain *tour-de-force* in which we had to learn the many corner cases that one can find when implementing hp elements in 2d and 3d, using constraints to enforce the continuity requirements of any given finite element space. The current paper therefore collects what we found are the requirements the implementation of hp methods imposes on code that describes a particular finite element space. deal.II itself already has a library of such finite element space descriptions, but there are other software libraries whose sole goal is to completely describe all aspects of finite element spaces (see, e.g., [Castillo et al. 2005]). The current contribution then essentially lists what pieces of information an implementor of a finite element class would have to provide to the underlying implementation in deal.II, and show how this information is used in the mathematical description. We also comment on algorithmic and data structure questions pertaining to the necessity to implement hp algorithms in an efficient way, and will support our claims of efficiency using a set of numerical experiments solving the Laplace equation in 2d and 3d and measuring the time our implementation spends in the various parts of the overall solution scheme.

We believe that our observations are by no means specific to deal.II: Other implementations of the hp method will choose different interfaces between finite element-specific and general classes, but they will require the same information. Furthermore, although all our examples will deal with quadrilaterals and hexahedra, the same issues will clearly arise when using triangles and tetrahedra. (For lack of complexity, we will not discuss the 1d case, although of course our implementation supports it as a special case.) The algorithms and conclusions described here, as well as the results of our numerical experiments, are therefore immediately applicable to other implementations as well.

The rest of the paper is structured as follows: In Section 2, we will discuss general strategies for h, p, and hp-adaptivity and explain our choice to enforce conformity of discrete spaces through hanging nodes. In Section 3, we introduce efficient data structures to store and address global degree of freedom information on the structural objects from which a triangulation is composed, whereas Section 4 contains the central part of the paper, namely what information finite element classes have to provide to allow for hp finite element implementations. Section 5 then deals with the efficient handling of constraints. Section 6 shows practical results, and Section 7 concludes the paper.

2. HP-ADAPTIVE DISCRETIZATION STRATEGIES

Adaptive finite element methods are used to improve the relation between accuracy and the computational effort involved in solving partial differential equations. They compare favorably with the more traditional approach of using uniformly refined meshes with a fixed polynomial degree by exploiting one or both of the following observations:

—for most problems the solution is not uniformly complex throughout the domain, i.e. it may have singularities or be “rough” in some parts of the domain;

—the solution does not always need to be known very accurately everywhere if, for example, only certain local features of the solution such as point values, boundary fluxes, etc,

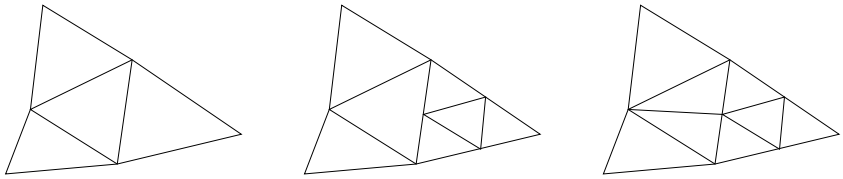

Fig. 1. *Refinement of a mesh consisting of four triangles. Left: Original mesh. Center: Mesh with rightmost cell refined. Right: The center cell has been converted to a transition cell.*

are of interest.

In either case, computations can be made more accurate and faster by choosing finer meshes or higher polynomial degrees of shape functions in parts of the domain where an "error indicator" suggests that this is necessary, whereas the mesh is kept coarse and lower degree shape functions are used in the rest of the domain.

A number of different and (at least for h-adaptivity) well-known approaches have been developed in the past to implement schemes that employ adaptivity. In the following subsections, we briefly review these strategies and explain the one we will follow in this paper as well as in the implementation of our ideas in the deal.II finite element library.

2.1 h-adaptivity

In the course of an adaptive finite element procedure, an error estimator indicates at which cells of the spatial discretization the error in the solution field is highest. These cells are then usually flagged to be refined and, in the h version of adaptivity, a new mesh is generated that is finer in the area of the flagged cells (i.e., the mesh size function $h(x)$ is adapted to the error structure). This could be achieved by generating a completely new mesh using a mesh generation program that honors prescribed node densities. However, it is more efficient to create the new mesh out of the old one by replacing the flagged cells with smaller ones, since it is then simpler to use the solution on the previous mesh as a starting guess for the solution on the new one.

This process of mesh refinement is most easily explained using a mesh consisting of triangles,[1] see Fig. 1: If the error is largest on the rightmost cell, then we refine it by replacing the original cell by the four cells that arise by connecting the vertices and edge midpoints of the original cell, as is shown in the middle of the figure.

In the finite element method shape functions are associated with the elements from which triangulations are composed. Taking the lowest-order P_1 space as an example, one would have shape functions associated with the vertices of a mesh. As can be seen in the central mesh of Fig. 1, mesh refinement results in an unbalanced vertex at the center of the face separating a refined and an unrefined cell, a so-called "hanging node". There are two widely used strategies to deal with this situation: special treatment of the degree of freedom associated with this vertex through introduction of constraints [Rheinboldt and Mesztenyi 1980; Carey 1997; Šolín et al. 2008; Šolín et al. 2003], and converting the center cell to

[1] For simplicity, we illustrate mesh refinement concepts here using triangles. However, the rest of the paper will deal with quadrilaterals and hexahedra because this is what our implementation supports. On the other hand, triangular and tetrahedral meshes pose very similar problems and the techniques developed here are applicable to them as well.

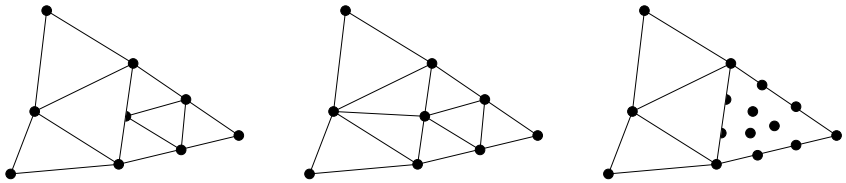

Fig. 2. *Degrees of freedom on h- and p-adaptive meshes. Left: Dots indicate degrees of freedom for P_1 (linear) elements on a mesh with a hanging node. Center: Resolution of the hanging node through introduction of transition cells. Right: A mixture of P_1 and P_3 elements on the original mesh.*

a transition cell using strategies such as *red-green refinement* [Carey 1997], as shown in the right panel of the figure. (An alternative strategy is to use Rivara's algorithm [Rivara 1984].) The left and center panel of Fig. 2 show the locations of degrees of freedom for these two cases for the common P_1 element with linear shape functions.

For pure h-refinement, both approaches have their merits, though we choose the first. If we use piecewise linear shape functions in the depicted situation, continuity of the finite element functions requires that the value associated with the hanging node is equal to the average of the value at the two adjacent vertices along the unrefined side of the interface. We will explain this in more detail in Section 4.4.

2.2 p-adaptivity

In the p version of adaptivity, we keep the mesh constant but change the polynomial degrees of shape functions associated with each cell. The right panel of Fig. 2 shows this for the situation that the rightmost cell of the original mesh is associated with a P_3 (cubic) element, whereas the other elements still use linear elements.

As is seen from the figure, we again have two "hanging nodes" in the form of the two P_3 degrees of freedom associated with the edge separating the two cells. There are again two widely used strategies to deal with this situation: introduction of constraints for the hanging nodes (explained in more detail in Section 4.3), and adding or removing degrees of freedom from one of the two adjacent cells. In the latter case, one would, for example, not use the full P_3 space on the rightmost cell, but use a reduced space that is missing the two shape functions associated with the line, and uses modified shape functions for the degrees of freedom associated with the vertices of the common face. Alternatively, one could use the full P_3 space on the rightmost cell, and augment the finite element space of the middle cell by the two P_3 shape functions defined on the common face.

2.3 hp-adaptivity

The hp version of adaptivity combines both of the approaches discussed in the previous subsections. One quickly realizes that the use of transition elements is not usually possible to avoid hanging nodes in this case, and that the only options are, again, constraints or enriched/reduced finite element spaces on the adjacent cells.

As above, in our approach we opt to use constraints to deal with hanging nodes. This is not to say that the alternative is not possible: it has in fact been successfully implemented in numerical codes, see for example [Demkowicz 2006]. However, it is our feeling that our approach is simpler in many ways: finite element codes almost always do operations such as integrating stiffness matrices and right hand side vectors on a cell-by-cell basis.

It is therefore advantageous if there is a simple description of the finite element space associated with each cell. When using constraints, it is unequivocally clear that a cell is, for example, associated with a P_1, P_2, or P_3 finite element space and there is typically a fairly small number (for example less than 10) of possible spaces. On the other hand, there is a proliferation of spaces when enriching or reducing finite element spaces to avoid hanging nodes. This is especially true in 3-d, where each of the four neighbors of a tetrahedron may or may not be refined, may or may not have a different space associated with it, etc. To make things worse, in 3-d not only the space associated with neighbor cells has to be taken into account, but also the spaces associated with any of the potentially large number of cells that only share a single edge with the present cell. If one considers the case of problems with several solution variables, one may want to use spaces $P_{k_1} \times P_{k_2} \times \cdots \times P_{k_L}$ with different indices k_l for each solution variable, and vary the indices k_l from cell to cell. In that case, the number of different enriched or reduced spaces becomes truly astronomical and may easily lead to inefficient and/or unmaintainable code.

Given this reasoning, we opt to use constraints to deal with hanging nodes. The following sections will discuss algorithms and data structures to store, generate, and use these constraints efficiently. Despite the relative simplicity of this approach, it should be noted already at this place that the generation of constraints is not always straightforward and that certain pathological cases exist, in particular in 3-d. However, we will enumerate and present solutions to all the cases we could find in our extensive use and testing of our implementation.

3. STORING GLOBAL INDICES OF DEGREES OF FREEDOM

In order to keep our implementation as general as can be achieved without unduly sacrificing performance, we have chosen to separate the concept of a `DoFHandler` from that of a triangulation and a finite element class in deal.II (see [Bangerth et al. 2007] for more details about this). A `DoFHandler` is a class that takes a triangulation and annotates it with global indices of the degrees of freedom associated with each of the cells, faces, edges and vertices of the triangulation. A `DoFHandler` object is therefore independent of a triangulation object, and several `DoFHandler` objects can be associated with the same triangulation, for example to allow programs that use different discretizations on the same mesh.

On the other hand, a `DoFHandler` object is also independent of the concept of a global finite element space, since it doesn't know anything about shape functions. It does, however, draw information from one or several finite element objects (that implement shape functions) in that it needs to know how many degrees of freedom there are per vertex, line, etc. A `DoFHandler` is therefore associated with a triangulation and a finite element object and sets up a global enumeration of all degrees of freedom on the triangulation as called for by the finite element object.

The deal.II library has several implementations of `DoFHandler` classes. The simplest, `dealii::DoFHandler` allocates degrees of freedom on a triangulation for the case that all cells use the same finite element; on the contrary, the `dealii::MGDoFHandler` class allocates degrees of freedom for a multilevel hierarchy of finite element spaces. In the context of this paper, we are interested in the data structures necessary to implement hp finite element spaces, i.e. we have to deal with the situation that different cells might be associated with different (local) finite element spaces. This concept is implemented in the

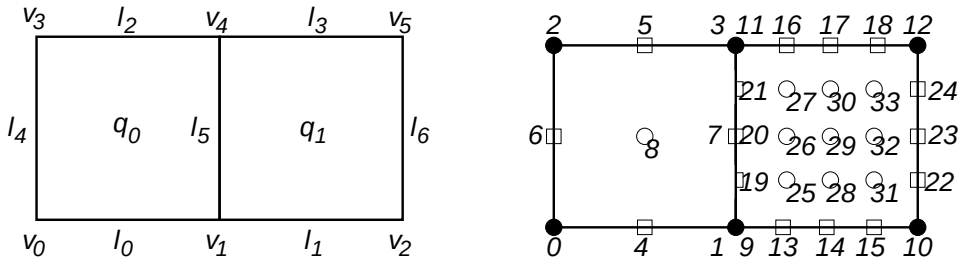

Fig. 3. *Left: A mesh consisting of two cells with a numbering of the vertices, lines, and quadrilaterals of this mesh. Right: A possible enumeration of degrees of freedom where the polynomial space on the left cell represents a Q_2 element and that on the right cell a Q_4 element. Bottom: Linked lists of degrees of freedom on each of the objects of which the triangulation consists.*

class `dealii::hp::DoFHandler`.[2]

Clearly, each cell is only associated with a single finite element, and only a single set of degrees of freedom has to be stored for each cell. However, the lower-dimensional objects (vertices, lines, and faces) that encircle a cell may be associated with multiple sets of degrees of freedom. For example, consider the situation shown in Fig. 3. There, a quadratic Q_2 element is associated with the left cell, whereas a quartic Q_4 element is associated with the one on the right. Here, the vertices v1 and v4 as well as the line l5 are all associated with both local finite element spaces. We therefore have to store the global indices of the degrees of freedom associated with both spaces for these objects.

Furthermore, it is clear that vertices in 2-d, and lines in 3-d, may be associated with as many finite element spaces as there are cells that meet at this vertex or line. This leads to our first requirement on implementations:

REQUIREMENT ON IMPLEMENTATIONS 1. *An implementation needs to store the global indices of degrees of freedom associated with each object (vertices, lines, etc.) of a triangulation. This storage scheme must be efficient both in terms of memory and in terms of fast access.*

Note that we only store the indices of degrees of freedom, not data associated with it. However, the indices can be used to look up data values in vectors and matrices.

In deal.II, we implement above requirement in the `hp::DoFHandler` class using a sort of linked list that is attached to each object of a triangulation. This list consists of one record for each finite element associated with this object, where a record consists of the number of the finite element as well as the global indices that belong to it. This is illustrated in Fig. 4 where we show these linked lists for each of the objects found in the triangulation depicted in Fig. 3. The caption also contains further explanations about the data format.

While other implementations are clearly possible, note that this storage scheme minimizes memory fragmentation. Furthermore, because in the vast majority of cases only a single element is associated with an object, access is also very fast since the linked list contains only one record.

[2]To avoid redundancy, we will drop the namespace prefix `dealii::` from here on.

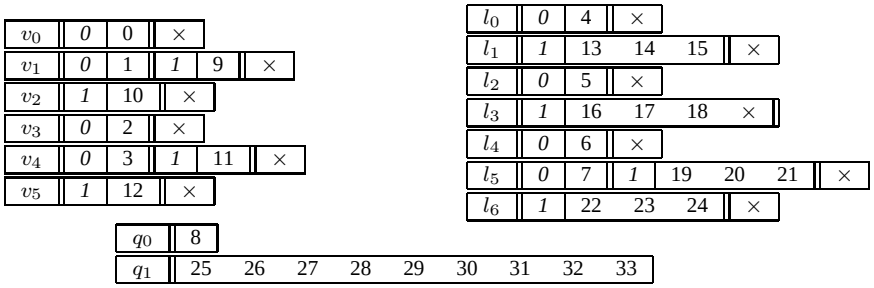

Fig. 4. *Lists of degrees of freedom associated with each of the objects identified in Fig. 3. For vertices and lines, there may be more than one finite element associated with each object, and we have to store a linked list of pairs of* `fe_index` *(printed in italics, zero indicates a Q_2 element, one indicates a Q_4 element) and the corresponding global numbers of degrees of freedom for this index; the list is terminated by an invalid index, here represented by $\times$. For quadrilaterals (i.e. cells in 2-d), only a single set of degrees of freedom can be active per object, and there is no need to store more than one data set or an* `fe_index` *that would identify the data set. Note that at this stage, each degree of freedom appears exactly once. This arrangement is later modified by the algorithm described in Section 4.2.*

4. REQUIREMENTS ON FINITE ELEMENT CLASSES

4.1 Higher order shape functions

Most importantly, finite element classes of course have to offer support for higher order shape functions to allow the use of hp finite element methods. This entails that we have an efficient way to generate them automatically for arbitrarily high polynomial degrees as well as for all relevant space dimensions. This is important since early versions of most finite element codes often implement only the lowest-order polynomials by hard-coding these functions. For example, in 2-d, the four shape functions for the Q_1 element are

$$\varphi_0(\mathbf{x}) = (1-x_1)(1-x_2), \qquad \varphi_1(\mathbf{x}) = (1-x_1)x_2,$$
$$\varphi_2(\mathbf{x}) = x_1(1-x_2), \qquad \varphi_3(\mathbf{x}) = x_1x_2.$$

These shape functions and their derivatives are obviously simple to implement directly.

On the other hand, this approach becomes rather awkward for higher order elements and in particular in 3d, for several reasons. First, these functions and their derivatives can only reliably be generated using automated code generators, for example by computing the Lagrange polynomials symbolically in Maple or Mathematica, and then generating corresponding code in the target programming language. While this leads to correct results, it is not efficient with respect to both compile and run time, since code generators are frequently not able to find efficient and stable product representations of these functions, such as for example a Horner scheme representation. Consequently, the code for these functions becomes very long, increasing both compile and run time significantly, while at the same time reducing numerical stability of the result. Secondly, the approach is not extensible at run time: only those polynomial degrees are available for which the corresponding code has been generated and compiled before.

In our experience with the deal.II library, composing shape functions from an underlying representation of the polynomial space addresses all these problems. For example, we implement the shape functions $\varphi_i^{(p)}$ of the Lagrange polynomial spaces Q_p as tensor

products of one-dimensional polynomials:

$$\varphi_i^{(p)}(\mathbf{x}) = \prod_{0\le d<dim} \psi^{(p)}_{j_d(i)}(x_d), \tag{1}$$

where $\psi_j^{(p)}(\cdot)$ are one-dimensional basis functions and $j_d(i)$ maps the dim-dimensional indices of the basis functions to one-dimensional ones; for example, a lexicographic ordering in 2-d would be represented by $j_0(i) = \lfloor i/p \rfloor$ and $j_1(i) = i \mod p$. The polynomials $\psi_j^{(p)}(\cdot)$ can be computed on the fly from the polynomial degree p using the interpolation property

$$\psi_j^{(p)}\left(\frac{n}{p+1}\right) = \delta_{nj}, \qquad 0 \le n \le p+1,$$

and are efficiently and stably encoded using the coefficients of the Horner scheme to compute polynomials. Using (1), it is also simple to obtain the gradient $\nabla\psi^{(p)}(\mathbf{x})$ and higher derivatives without much additional code. The introduction of this representation in deal.II allowed us not only to trivially add Lagrange elements of order higher than 4 in 2-d and higher than 2 in 3-d, it also allowed us to delete approximately 28,000 lines of mostly machine generated code in addition to speeding up computation of basis functions severalfold.

Basing the computation of shape functions on simple representations of the function space is even more important for more complicated function spaces like those involved in the construction of Raviart-Thomas or Nedelec elements. For example, on the reference cell, the Raviart-Thomas space on quadrilaterals is the anisotropic polynomial space $Q_{k+1,k} \times Q_{k,k+1}$ in 2-d, and $Q_{k+1,k,k} \times Q_{k,k+1,k} \times Q_{k,k,k+1}$ in 3-d (see, e.g., [Brezzi and Fortin 1991]), where indices indicate the polynomial order in each space direction individually. From such a representation, it is easy to write basis functions of this space for arbitrarily high degrees as a tensor product of one-dimensional polynomials, completely avoiding the need to implement any of them "by hand". Similar techniques as outlined above for quadrilaterals and hexahedra are likely also going to be available for triangles and tetrahedra, see for example [Šolín et al. 2003].

REQUIREMENT ON IMPLEMENTATIONS 2. *Finite element classes need to have an efficient way to generate shape functions of arbitrary order to avoid automatic code generation of high order polynomials that is usually accompanied by an explosion of code size and run time.*

4.2 Description of identities of degrees of freedom

As mentioned in Section 3, we store global indices for each degree of freedom on vertices, lines, quadrilaterals, etc, for each of the cells adjacent to these objects. For example, Fig. 3 showed this for the case of adjacent cells with Q_2 and Q_4 elements, respectively.

If one knows that for Lagrange elements, degrees of freedom represent *values* of shape functions, then it is immediately clear that for a finite element field $u(\mathbf{x}) = \sum_i u_i\varphi_i(\mathbf{x})$ to be continuous, one needs the constraints $u_1 = u_9, u_3 = u_{11}$, and $u_7 = u_{20}$, in addition to conditions linking u_{19} and u_{21} to u_1, u_3, and u_7 (these latter conditions are discussed in Section 4.3 below). In other words, for Lagrange elements, all degrees of freedom associated with the same vertex must have the same value, and the same holds for certain degrees of freedom on lines (or on quadrilaterals in 3-d). It is worth noting that this is a property of the finite element, not of degrees of freedom in themselves: one could, for

example, think of C^1 conforming elements having four degrees of freedom on each vertex representing the value, first derivatives, and the mixed second derivative of the field in the coordinate system of the *reference cell* at this location; unless the adjacent cells have a particular orientation to each other, only the values at the vertex will coincide, while the derivatives will only be related but not necessarily be identical in value.

Constraints such as $u_1 = u_9, u_3 = u_{11}$, and $u_7 = u_{20}$ could be dealt with in the same way as hanging node constraints, by adding these conditions as explicit constraints to the linear system of equations. However, that would be inefficient: it needlessly increases the number of unknowns of a linear system, costing memory and compute time.

Rather, the implementation of the `hp::DoFHandler` in deal.II requires finite element classes to provide information on *identities* of degrees of freedom. After degrees of freedom have been distributed on each cell individually, producing for example the layout referenced in Fig.s 3 and 4, the `hp::DoFHandler` goes over all objects (vertices, lines, etc.) again and tries to identify identical degrees of freedom if multiple sets of degrees of freedom are stored on this object.

To this end, the `hp::DoFHandler` would perform a call similar to the one shown in Listing 1. This code first queries whether there is more than one finite element associated with a vertex; this would be true for vertices `v1` and `v4` in Fig. 3, for example. If so, it then asks all pairs of finite elements active on this vertex to return lists of identical degrees of freedom. In the present case, the Lagrange finite element class would return a list of length 1 consisting of a single pair of zeros: the zeroth (and only) degrees of freedom associated with either of the two elements are identical. The code would then go on and set the global index of the degree of freedom associated with the second finite element to the same index as that of the first. Note that for the hypothetical C^1 element above, the returned list would also consist of a single pair of zeros, indicating that only the values, not the derivatives at a vertex must coincide; on the other hand, if the C^1 element implemented its shape functions so that the later shape functions indicate derivatives in the *global coordinate system*, then all four degrees of freedom must be the same and the finite element should return a list $\{\{0,0\},\{1,1\},\{2,2\},\{3,3\}\}$.

After this process, for the example given in Fig.s 3 and 4, degrees of freedom 9 and 11 have been removed, and the linked lists for vertices `v1` and `v4` now read as follows:

v1	*0*	1	*1*	1	$\times$
v4	*0*	3	*1*	3	$\times$

A similar process is then repeated for lines. In this case, on line `l5`, we call a function `fe[f].hp_line_dof_identities(fe[g])` which, for the pair Q_2 and Q_4 elements, would return the list $\{\{0,1\}\}$. This indicates that the first (and only) degree of freedom of the Q_2 element is identical to the second degree of freedom of the Q_4 element since they represent shape functions corresponding to identical interpolation points. A code similar to the one shown in Listing 1 would then yield the following list for this line:

l5	*0*	7	*1*	19	7	21	$\times$

Note that we need not perform any such algorithm on cells, since they can only have a single set of degrees of freedom associated with them. On the other hand, it is necessary to do so for quadrilaterals in 3-d. At the end of all these operations and after eliminating degrees of freedom 9, 11, and 20, we renumber all degrees of freedom to use a consecutive

```
if (v->n_active_fe_indices() > 1)
  for (unsigned int f=0; f<v->n_active_fe_indices(); ++f)
    for (unsigned int g=f+1; g<v->n_active_fe_indices(); ++g)
      {
        unsigned int fe_index1 = v->nth_active_fe (f),
                     fe_index2 = v->nth_active_fe (g);

        std::vector<std::pair<unsigned int, unsigned int> >
          dof_identities
          = fe[fe_index1].hp_vertex_dof_identities(fe[fe_index2]);

        for (unsigned int i=0; i<dof_identities.size(); ++i)
          v->set_dof_index (g,
                            dof_identities.second[i],
                            v->get_dof_index (f,
                                              dof_identities.first[i]));
      }
```

Listing 1. *Identifying degrees of freedom on a vertex* `v`.

range $0, \ldots, 30$.

Using this identification of degrees of freedom, we can immediately reduce the total size of linear systems by a significant fraction: in the 2d test case shown in Section 6, some 6% of degrees of freedom can be eliminated right away; in 3d, the fraction can be as high as 10-15%. This not only keeps matrices and vectors small, but also significantly reduces the number of degrees of freedom on which we later have to apply hanging node elimination as explained in the following section.

Unfortunately, a straight-forward adaptation to 3-d of the concepts discussed here is not possible, though the general idea and the basic algorithm remains the same. We will therefore come back to identifying degrees of freedom of different finite elements in Section 4.6. This notwithstanding, we need finite element implementations to provide us with the following information:

REQUIREMENT ON IMPLEMENTATIONS 3. *Finite element classes need to be able to communicate to the* `hp::DoFHandler` *which degrees of freedom located on vertices, edges, and faces of cells are identical even though they belong to finite elements of different polynomial orders or even different kinds.*

4.3 Interpolation on common faces between cells with different finite elements

The discrete functions which are represented by the finite element discretization have to satisfy certain continuity requirements across the element edges in most cases. For example, after the unification of degrees of freedom 1 and 9, 3 and 11, and 7 and 20 in Fig. 3 as discussed in the previous section, finite element functions on the left and right sides of the edge separating the two cells will only be continuous for particular values of degrees of freedom 19 and 21. In this section, we will derive the conditions on these degrees of freedom for the case that only the polynomial degree p of the ansatz spaces changes across an edge in our mesh, as well as how such constraints are efficiently implemented. The case that two neighboring cells also have different h-refinement levels is treated in the

next sections. In the presentation of these techniques, we follow to some degree the algorithms shown in [Ainsworth and Senior 1997; 1999]; however, we go beyond the material presented there in that we allow for the case of local refinement where the *coarser* cell has a higher polynomial degree, and in that we discuss some complications specific to the three-dimensional case.

It is worth noting that continuity requirements do not always enforce that a function has to be continuous across element edges. For example, discrete subspaces of $H(\mathrm{div})$ only require continuity of the normal component along the faces between cells (c.f. [Brezzi and Fortin 1991]). For the sake of simplicity, let us here only consider continuity of functions across element edges; it shall be understood that the same requirements and assumptions also hold when the normal or tangential component has to be continuous along element edges.

Let us first consider the simplest case of such constraints, i.e. the one corresponding to the situation of Fig. 3. If we forget for a moment that we have already identified certain degrees of freedom, then the continuity constraint requires that

$$\begin{aligned} &u_1\varphi_1(\mathbf{x}) + u_3\varphi_3(\mathbf{x}) + u_7\varphi_7(\mathbf{x}) \\ &\qquad = u_9\varphi_9(\mathbf{x}) + u_{11}\varphi_{11}(\mathbf{x}) + u_{19}\varphi_{19}(\mathbf{x}) + u_{20}\varphi_{20}(\mathbf{x}) + u_{21}\varphi_{21}(\mathbf{x}) \end{aligned} \quad (2)$$

for all $\mathbf{x}$ on the line `l5`. It is apparent in the present case that we need to restrict the degrees of freedom on the element with the higher polynomial degree, in order to enforce this equality. We identify this situation by saying that the Q_2 element *dominates* the Q_4 element on a shared face. This leads us to the following requirement:

REQUIREMENT ON IMPLEMENTATIONS 4. *An implementation needs to be able to determine which of two finite elements that share a face dominates the other, or if neither does.*

We will comment on the last case at the end of this section. In the present situation, and given the quadratic polynomials representing $\varphi_1, \varphi_3, \varphi_7$ and the quartic polynomials $\varphi_9, \varphi_{11}, \varphi_{19}, \varphi_{20}, \varphi_{21}$, it is readily checked that equality (2) implies

$$\begin{gathered} u_9 = u_1, \qquad u_{11} = u_3, \qquad u_{20} = u_7, \\ u_{19} = \frac{3}{8}u_1 - \frac{1}{8}u_3 + \frac{3}{4}u_7, \qquad u_{21} = -\frac{1}{8}u_1 + \frac{3}{8}u_3 + \frac{3}{4}u_7. \end{gathered}$$

Note that the first three constraints have already been taken care of through identification of the corresponding degrees of freedom as described in the previous section.

Above conditions can be written in a more compact way as follows, linking the degrees of freedom of the side of the face with the higher polynomial degree to those on the side with the lower degree:

$$\mathbf{u}|_{\text{dominated side of } \texttt{l5}} = \begin{pmatrix} u_9 \\ u_{11} \\ u_{19} \\ u_{20} \\ u_{21} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \frac{3}{8} & -\frac{1}{8} & \frac{3}{4} \\ 0 & 0 & 1 \\ -\frac{1}{8} & \frac{3}{8} & \frac{3}{4} \end{pmatrix} \begin{pmatrix} u_1 \\ u_3 \\ u_7 \end{pmatrix} = I^{\text{face}}_{Q_4 \to Q_2} \mathbf{u}|_{\text{dominating side of } \texttt{l5}}. \quad (3)$$

For Lagrange interpolation polynomials, the matrix $I^{\text{face}}_{Q_k \to Q_{k'}}$ that appears in this relation is simple to compute by evaluating the 5 basis functions of the Q_4 element on the right

at the interpolation points of the 3 basis functions of the Q_2 element on the left. However, these matrices are more complicated to compute for elements where the degrees of freedom are moments on faces or where values and not only gradients of shape functions are transformed by the mapping from reference to real cell; both these cases apply to elements such as the Raviart-Thomas element for $H(\mathrm{div})$.

REQUIREMENT ON IMPLEMENTATIONS 5. *Finite element classes need to be able to generate the constraint matrices I^{face} as in* (3) *that enforce continuity requirements between cells associated with different finite element spaces. These matrices must be available for all possible pairs of spaces appearing in a triangulation.*

We remark that an important requirement for this approach is that the functions appearing on one side of (2) must be able to exactly represent the functions on the other side. For the chosen pair of spaces, this is obvious: the restriction of Q_4 onto a line is the space of quartic polynomials that is of course able to represent the quadratic polynomials that result from restricting Q_2 to this line, i.e. we say that the Q_2 elements *dominates* the Q_4 element. However, it is not always the case that one element dominates the other. For example, consider the hypothetical situation of a mesh containing an edge where elements meet that have Q_4 and $R_{2,2}$ (the space of rational expressions with quadratic denominator and enumerator) elements: neither of the two is able to represent the restriction of the other one on a common edge, i.e. neither element dominates the other. Another, more practical example, would be that one cell is home to a $Q_2 \times Q_1$ vector-valued element, whereas its neighbor is associated with a $Q_1 \times Q_2$ element.

The solution to this case is to first identify a common subspace; in the first example this could be the space Q_2 of quadratic polynomials on this face, whereas in the second we would choose $Q_1 \times Q_1$. The second step would then be to enforce constraints that restrict finite element functions on both sides of the face to be within this subspace, and in addition to be equal along the face. In order to not restrict the global finite element space more than necessary, we should attempt to find the largest admissible subspace along the face. However, since this is a case that doesn't appear to have much application in practical finite element cases, we won't dwell on it in more detail, but will come back to a closely related case in Section 4.4.3.

4.4 Interpolation on refined faces between cells

The next case to consider is that of h-refinement. Fig. 5 shows the two cases that can appear in this situation: the left panel corresponds to the "simple" case where the element on the large side of the face (here a Q_1) dominates the one on the right (here a Q_2). This also includes the pure h-refinement case where both sides use the same element. In contrast, the right panel shows the "complex" case where the dominating element sits on the refined side of a face. We will discuss these two cases separately in the following.

4.4.1 *The simple case.* The "simple" case of h-refinement is where the dominating element is located on the unrefined side of the face, as shown in the left part of Fig. 5. In the situation shown, to guarantee continuity at the sub-face (the shared part of the face), we need the constraint

$$u_0\varphi_0(\mathbf{x}) + u_1\varphi_1(\mathbf{x}) = u_2\varphi_2(\mathbf{x}) + u_3\varphi_3(\mathbf{x}) + u_4\varphi_4(\mathbf{x}). \tag{4}$$

It is obvious that this constraint is enforceable since the restriction of the Q_1 space on the left to the sub-face is a linear function and that we can constrain the quadratic function on

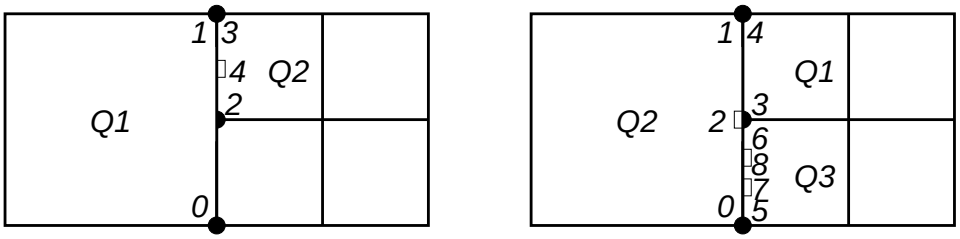

Fig. 5. *The simple (left) and complicated (right) case of h-refinement across a face between cells with different finite element spaces associated with them. Only those degrees of freedom that are relevant for the interpolation process are indicated.*

the right to equal this linear function. In particular, it is easy to show that we need

$$\mathbf{u}|_{\text{dominated side of subface}} = \begin{pmatrix} u_2 \\ u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ 0 & 1 \\ \frac{1}{4} & -\frac{3}{4} \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \end{pmatrix} = I^{\text{subface}(sf)}_{Q_2 \to Q_1} \mathbf{u}|_{\text{dominating side of subface}}, \tag{5}$$

where sf denotes the number of the subface we are presently looking at. As before, for Q_k elements, we can obtain the interpolation matrix $I^{\text{subface}(sf)}$ by evaluating the shape functions associated with the coarse side of the face at the interpolation points of the shape functions on the refined side of the face.

REQUIREMENT ON IMPLEMENTATIONS 6. *Finite element classes need to be able to generate the constraint matrices $I^{subface(sf)}$ as in* (5)*, for each of the 2^{dim-1} subfaces sf of a face. These matrices must be available for all possible pairs of spaces appearing in a triangulation.*

In deal.II, this requirement is currently implemented for elements from the same family by automatically computing the interpolation matrices at interfaces between cells.

4.4.2 *The complex case, approach 1.* The more complicated case is the one shown on the right of Fig. 5. The problem is that the linear (dominating) element is located on the refined side of the subface. As we will see in the following, this case is riddled with problems, false hopes, and traps; we will consider several cases and possible solutions that illustrate why this case is hard.

A naive approach to the situation shown in Fig. 5 is as follows: In order to force continuity of finite element functions, we need to make sure that the three degrees of freedom 0, 1, and 2 of the Q_2 element actually form a linear function, since otherwise there would be no way the linear function on the upper right Q_1 cell could match the one on the left. In general, we need to constrain the finite element space on the left of the refined face to the most dominating space on the right, here Q_1. We could do that by constraining the left face to the degrees of freedom 3 and 4 of the most dominating child face, i.e. to require

$$\mathbf{u}|_{\text{dominated side of face}} = \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} 2 & -1 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} u_3 \\ u_4 \end{pmatrix} = RI^{\text{subface}(sf)}_{Q_1 \to Q_2} \mathbf{u}|_{\text{most dominating subface}}.$$

We call the matrix $RI^{\text{subface}(sf)}_{Q_1 \to Q_2}$ the *reverse interpolation matrix*. It is not hard to see that

for this particular subface, we can write it as

$$RI^{\text{subface}(sf)}_{Q_1\to Q_2} = I^{\text{face}}_{Q_2\to Q_1}\left[I^{\text{subface}(sf)}_{Q_1\to Q_1}\right]^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix}\begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ 0 & 1 \end{pmatrix}^{-1}. \tag{6}$$

This formula can be understood as follows: Let us introduce the degrees of freedom x_0, x_1 of a virtual linear finite element space along the entire edge. The condition of continuity then means that

$$\begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} = I^{\text{face}}_{Q_2\to Q_1}\begin{pmatrix} x_0 \\ x_1 \end{pmatrix}, \qquad \begin{pmatrix} u_3 \\ u_4 \end{pmatrix} = I^{\text{subface}(sf)}_{Q_1\to Q_1}\begin{pmatrix} x_0 \\ x_1 \end{pmatrix}.$$

It is now important to realize that the matrix $I^{\text{subface}(sf)}_{Q_1\to Q_1}$ should always be invertible, even if we replace the space Q_1 by any other reasonable finite element space. This follows from the fact that the matrix intuitively describes how a finite element space on a face is restricted to the same space but on a subface. Its inverse, if it exists, then describes how a finite element function on a subface uniquely determines the function on the entire face. Since we are dealing with polynomials, the existence and uniqueness of the inverse is easy to see: polynomials are analytic functions, and its Taylor expansion on a subface therefore uniquely determines its extension to the entire face. The matrix that relates the values of this polynomial at disjoint interpolation points must consequently be invertible.

Given this, (6) is therefore a universal representation of the reverse interpolation matrix that we can obtain by removing the virtual unknowns x_0, x_1 from the equations.

Using these formulas, we have now seen how to constrain the degrees of freedom on the left Q_2 side of the face. The remaining question is what to do with the other subfaces on the right. To this end, note that we have chosen the most dominant finite element space present on this face and its children to define the virtual unknowns x_0, x_1. Consequently, all other subfaces can be constrained to it as well. For example, for the Q_3 subface shown in the figure, we would get

$$\begin{pmatrix} u_5 \\ u_6 \\ u_7 \\ u_8 \end{pmatrix} = I^{\text{face}}_{Q_3\to Q_1}\begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = I^{\text{subface}(sf')}_{Q_3\to Q_1}\left[I^{\text{subface}(sf)}_{Q_1\to Q_1}\right]^{-1}\mathbf{u}|_{\text{most dominating subface}}.$$

Here, sf' is the subface with the Q_3, and sf denotes the subface associated with the most dominating finite element space, i.e. the one with the Q_1.

So what is wrong with this approach? To see why this approach is not always successful, consider the situation shown in Fig. 6. In this case, we get constraints $u_6 = \frac{1}{2}u_0 + \frac{1}{2}u_2$, i.e. u_6 is constrained to u_0. On the other hand, we have to constrain u_0, u_7, and u_{10} to either u_0, u_{12}, or to u_{10}, u_{12}. Let us assume we chose the second alternative. In that case, we obtain the constraint $u_0 = 2u_{12} - u_{10}$. In effect, u_6 is constrained to a degree of freedom that is itself constrained.

We note that we could have avoided this here by choosing the second alternative. However, this would not have been the case if, for example, the Q_1 element characterized by degrees of freedom 12, 13, 0, and 16 were replaced by a Q_3 element, in which case the second alternative would have been forced upon us since we need to restrict to the most dominating finite element space on a face.

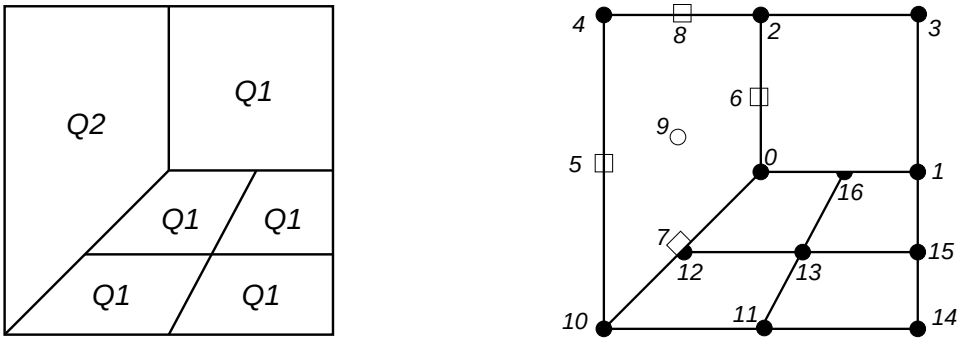

Fig. 6. *Illustration how the first approach leads to connected constraints: Geometry and associated finite element spaces (left) and degrees of freedom (right).*

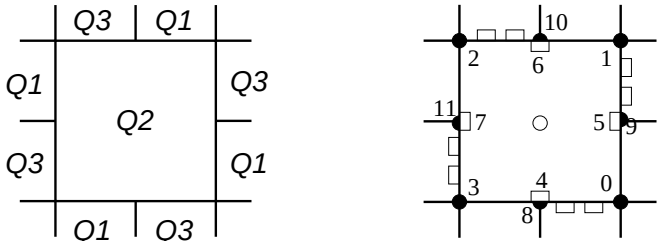

Fig. 7. *Illustration how the first approach leads to circular constraints: Geometry and associated finite element spaces (left) and those degrees of freedom relevant to the discussion (right).*

So far, the constraints are not self-contradicting: as long as we get chains of constraints $u_6 \to u_0 \to u_{10}$, the constraints form an directed acyclic graph (DAG). Although it is awkward to deal with such nested constraints in programs, it is certainly possible to deal with them.

However, we can find situations where this graph has cycles. Consider, for example, the situation shown in Fig. 7: On the bottom face, degrees of freedom 0, 3, and 8 have to be constrained to the Q_1 subface, i.e. to degrees of freedom 3 and 8. In particular, we get the constraint $u_0 = 2u_8 - u_3$. In a similar way, we get $u_1 = 2u_9 - u_0$, $u_2 = 2u_{10} - u_1$, and $u_3 = 2u_{11} - u_2$ on the other three faces of the central cell. Note how we have just created a cycle in our directed graph of constraints.

Since we are unsure how to proceed both theoretically and practically, we believe that this first approach of dealing with constraints in the "complex" case is not workable.

4.4.3 *The complex case, approach 2.* A second approach is to not constrain the degrees of freedom on the large side of the face to those of one of the subfaces, but among themselves. Going back to the right side of Fig. 5, that would mean that we impose constraints only on degrees of freedom 0, 1, and 2 to make sure the resulting function is linear. In the present case, this would require the constraint $u_2 = \frac{1}{2}u_0 + \frac{1}{2}u_1$.

In the more general case, let us again introduce virtual degrees of freedom $\mathbf{x}$ that corre-

spond to the most dominant finite element restricted to a face, which we name S. Then

$$\mathbf{u}|_{\text{coarse side of face}} = I^{\text{face}}_{S^{\text{face}}\to S}\mathbf{x},$$

where S^{face} is the finite element space on the coarse side of the face. Since we have assumed that S^{face} is dominated by S, we can divide $\mathbf{u}|_{\text{coarse side of face}}$ into a set of master and slave nodes such that

$$\mathbf{u}|_{\text{coarse side of face}} = \begin{pmatrix} \mathbf{u}|^{\text{master}}_{\text{coarse side of face}} \\ \mathbf{u}|^{\text{slave}}_{\text{coarse side of face}} \end{pmatrix} = \begin{pmatrix} I^{\text{face, master}}_{S^{\text{face}}\to S} \\ I^{\text{face, slave}}_{S^{\text{face}}\to S} \end{pmatrix} \mathbf{x}.$$

Let us assume that we can subdivide degrees of freedom into master and slave nodes such that $I^{\text{face, master}}_{S^{\text{face}}\to S}$ is an invertible square matrix (an assumption that we will discuss below), then we can conclude that

$$\mathbf{u}|^{\text{slave}}_{\text{coarse side of face}} = I^{\text{face, slave}}_{S^{\text{face}}\to S}\mathbf{x} = I^{\text{face, slave}}_{S^{\text{face}}\to S}\left[I^{\text{face, master}}_{S^{\text{face}}\to S}\right]^{-1}\mathbf{u}|^{\text{master}}_{\text{coarse side of face}}.$$

Now that we know how to deal with the degrees of freedom on the coarse side of the face, it is clear that we can deal with the subfaces as follows:

$$\mathbf{u}|_{\text{subface } sf} = I^{\text{subface}(sf)}_{S^{\text{subface}(sf)}\to S}\mathbf{x} = I^{\text{subface}(sf)}_{S^{\text{subface}(sf)}\to S}\left[I^{\text{face, master}}_{S^{\text{face}}\to S}\right]^{-1}\mathbf{u}|^{\text{master}}_{\text{coarse side of face}}.$$

Here, $S^{\text{subface}(sf)}$ is the finite element space on subface sf.

Using this second approach, one can easily construct situations where one gets chains of constraints. For example, the Q_3 degrees of freedom on the interface between the small Q_1 and Q_3 cells at the bottom of Fig. 7 (though not explicitly shown) will be constrained to the degrees of freedom located on the two vertices of the common edge. One of those is degree of freedom 8, which is itself constrained to degrees of freedom 0 and 3.

However, it is easy to prove that with this second approach there can be no cycles in the graph of constraints: resulting from the definition of dominance of spaces, each constraint is always from a degree of freedom to other ones associated with a strictly smaller embedded finite element space. The fact that this wasn't the case for the first approach illustrates immediately why that approach was prone to failure.

We conclude this subsection with a discussion of the slight complication of how to choose the subdivision of nodes on the coarse side of the face into master and slave nodes, $\mathbf{u}|^{\text{master}}_{\text{coarse side of face}}$ and $\mathbf{u}|^{\text{slave}}_{\text{coarse side of face}}$. First, let us note that the matrix $I^{\text{face}}_{S^{\text{face}}\to S}$ necessarily must have full column rank, since it is the matrix that interpolates the shape functions of the dominating space S at the support points of the dominated space S^{face}. Assuming S to be unisolvent, i.e. consisting of linearly independent shape functions, and assuming that no two interpolation points of S^{face} coincide, then the full column rank immediately follows. Consequently, we can hope that we can select certain linearly independent rows of $I^{\text{face}}_{S^{\text{face}}\to S}$ to form an invertible matrix $I^{\text{face,master}}_{S^{\text{face}}\to S}$.

The selection of these rows, however, turns out to be more involved than one would think at first. In particular, one can not simply take the first n_S of the $n_{S^{\text{face}}}$ rows, since they sometimes are linearly dependent. It is conceivable that one could devise an exact strategy for this problem, though we are not aware of any such approach and therefore chose to implement a heuristic: if $n_S^{\text{vertex}}, n_S^{\text{line}}, n_S^{\text{quad}}$ are the number of degrees of freedom associated with each vertex, line, and quad of the dominating space S, then select the first n_S^{vertex} degrees of freedom from each vertex in the dominated space S^{face} as master nodes, then the first n_S^{line} degrees of freedom from each line, and so on.

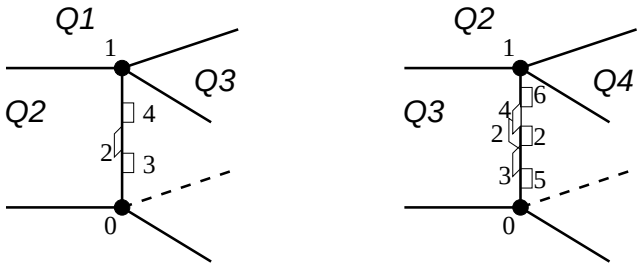

Fig. 8. *Left: Depiction of three hexahedral cells meeting at a common edge. The cell in the background is associated with a Q_1 element and has degrees of freedom 0 and 1 on the common edge. The element in the foreground is a Q_2 and carries degrees of freedom 0, 1, and 2. The element on the right is a Q_3 with degrees of freedom 0, 1, 3, and 4. Only degrees of freedom on the common edge are shown. Right: The same situation with Q_2, Q_3, and Q_4 elements meeting at a common edge.*

Using this heuristic almost always produces a matrix $I^{\text{face,master}}_{S^{\text{face}}\to S}$ that is invertible. In our experiments, the only cases where it fails are in 3-d when a $S^{\text{face}} = Q_4$ (or higher) element on a coarse cell neighbors a $S = Q_3$ (or higher) element on a set of refined cells. If we encounter such a situation, we drop the last master node to be added to the list and replace it with a slave node, until we end up with an invertible matrix. The checks for this are numerically expensive, but given the rarity of using such high polynomial degrees in 3-d and that the subdivision into master and slave nodes has to be done only once per program run, the additional effort is negligible.

4.5 Complications in 3-d: Seemingly incompatible constraints

Faces between cells, i.e. lines in 2-d and quadrilaterals in 3-d, are special in that they are shared between exactly two cells. Consequently, dealing with them is fairly straightforward, despite the length of the discussion above. On the other hand, vertices in 2-d and lines in 3-d can have arbitrarily many finite elements associated with them. While this is not much of a problem in 2-d since only slightly uncommon elements (such as C^1 continuous elements) have more than one degree of freedom associated with each vertex, it is perhaps not surprising that the situation is more complex in 3-d and generates an additional set of problems. (Vector-valued elements such as ${Q_k}^d$ of course have more than one degree of freedom per vertex, but they can be decomposed into their individual vector components, each of which is handled independently.)

Consider, for example, the situation shown in the left panel of Fig. 8. There, we have three cells associated with Q_1, Q_2, and Q_3 elements meeting at a common edge. If we, for example, first treat the face between the Q_2 and Q_3 elements, we would record the constraint

$$u_3 = \tfrac{2}{9}u_0 - \tfrac{1}{9}u_1 + \tfrac{8}{9}u_2.$$

However, if we treat the face between the Q_3 and the Q_1 elements next, we would discover the constraint

$$u_3 = \tfrac{2}{3}u_0 + \tfrac{1}{3}u_1.$$

It is important to note that these seemingly incompatible constraints on u_3 are in fact the same since we will later discover that $u_2 = \frac{1}{2}u_0 + \frac{1}{2}u_1$ when we treat the face between the

Q_2 and Q_1 elements, and we can expand the first constraint on u_3 into the second form by resolving the chain of constraints.

REQUIREMENT ON IMPLEMENTATIONS 7. *An implementation has to be able to keep track which degrees of freedom are already constrained, and simply ignore constraints generated for degrees of freedom for which other constraints have already been registered.*

While this is a simple solution, it is bothersome that one loses the ability for safety checks by just throwing away a constraint. It would be nicer if we could keep it, expand the constraint later on when u_2 is resolved, and then make sure that it equals the original constraint with which it appeared to conflict. A defensive implementation would therefore follow this latter strategy, in order to ensure that our computations are correct, and produce an error if the two constraints are not the same. We have found this strategy of pervasive and exhaustive internal consistency checks of great value in finding obscure bugs and corner cases in our implementation.

4.6 Complications in 3-d: Seemingly circular constraints

Yet another situation is shown in the right panel of Fig. 8. The situation is similar to the one discussed in the preceding subsection, but with polynomial degrees increased by one. The additional complication is introduced because we have identified the middle degree of freedom on the Q_4 side of the edge with degree of freedom 2 of the Q_2 side of the edge, using the algorithm of Section 4.2. Now, when we build constraints for the face between the Q_4 and Q_3 elements, we realize that the latter element dominates the former one, and therefore has to register constraints for degrees of freedom u_2, u_5, and u_6. In particular, we find

$$u_2 = -\tfrac{1}{16}u_0 - \tfrac{1}{16}u_1 + \tfrac{9}{16}u_3 + \tfrac{9}{16}u_4.$$

On the other hand, we find when dealing with the face between the Q_3 and Q_2 elements the constraint

$$u_3 = \tfrac{2}{9}u_0 - \tfrac{1}{9}u_1 + \tfrac{8}{9}u_2.$$

This is a circular constraint $u_2 \to u_3 \to u_2$.

We have not found a fully satisfactory solution to this problem. Ideally, one would like to exclude those degrees of freedom from identification (as described in Section 4.2) that will later create such trouble. Here, this means that the middle degree of freedom on the Q_4 edge should not have been identified with the degree of freedom 2 of the Q_2 edge. However, writing a routine that pre-scans for the potential for trouble appears complicated. Our solution is to simply not identify any degrees of freedom whenever there are three or more finite elements associated with an edge in 3-d, in contrast to the algorithm shown in Listing 1.

Note that this actually only concerns a relatively small number of degrees of freedom, since the restriction only triggers in 3-d and on edges at which cells meet that have at least three different finite elements associated with them. Most often, however, the smoothness of solutions changes gradually and the regions of the domain associated with a particular finite element form shells with interfaces where only two different finite elements meet.

5. EFFICIENT HANDLING OF CONSTRAINTS

After applying the strategies of the previous section, we now have a set of degrees of freedom many of which are constrained. (In a slight abuse of language, we will call them "constrained degrees of freedom".) In practical applications such as those shown below, up to 20% of the total number of degrees of freedom can be constrained. Their efficient handling is therefore of importance, and involves two aspects: storing the information about constraints, and applying these constraints to linear systems of equations. We will discuss these in the following.

5.1 Data structures for constraints

All the constraints we have constructed in the previous sections are homogeneous, i.e. have the form

$$c_i^T U = 0,$$

where $c_i, i = 1, \ldots, I$ is a vector of weights for the ith constraint, and U is the vector of unknowns $u_k, k = 1, \ldots, N$. The set of all constraints can therefore be written as $CU = 0$, and we call C the *constraint matrix*.

Because constraints typically only involve a small number of unknowns, c_i is a sparse vector and storing constraints as a set of full vectors is not efficient. In addition, we use that each constraint c_i corresponds to one particular degree of freedom $q(i)$ that is constrained by the values of L_i other degrees of freedom with indices $r_l(i), l = 1, \ldots, L_i$. In other words, we can normalize c_i such that $(c_i)_{q(i)} = 1$ to obtain the following form of constraint i:

$$u_{q(i)} = -\sum_{l=1}^{L_i} (c_i)_{r_l(i)} u_{r_l(i)}. \tag{7}$$

A suitable and efficient storage format for the constraint matrix is therefore a list of length I, where each entry contains first the index $q(i)$ of the constrained degree of freedom, and secondly a list of length L_i of pairs $r_l(i), (c_i)_{r_l(i)}$. This format is memory efficient and well suited to the operations involving constraint matrices described below.[3]

5.2 Applying constraints

With this definition of constraints, the problem we need to solve is $AU = F$, where A is the matrix with entries $a_{ij} = b(\varphi_i, \varphi_j)$ obtained from the bilinear form $b(\cdot, \cdot)$ of the problem involving all shape functions φ_i (corresponding to unconstrained and constrained degrees of freedom) and F the corresponding right hand side. In addition, we have to enforce our constraints $CU = 0$. In general, however, this constrained form is not particularly suitable, since, among other reasons, it is already unclear whether this set of two equations will have a solution at all. (It is easy to show the existence of a unique solution if A corresponds to a positive definite operator such as the Laplacian, but the problem becomes more complicated with indefinite operators where the solution is no longer derived through the minimization of an energy.)

[3] Alternatively, the pairs $r_l(i), (c_i)_{r_l(i)}$ could be stored in a compressed row-major (CSR) format as a matrix of size $I \times N$, with the $q(i)$ being stored as an additional vector of size I. However, building this matrix is inefficient in practice since the size and number of entries per row are not known a priori, requiring costly resizing operations on the compressed sparsity pattern.

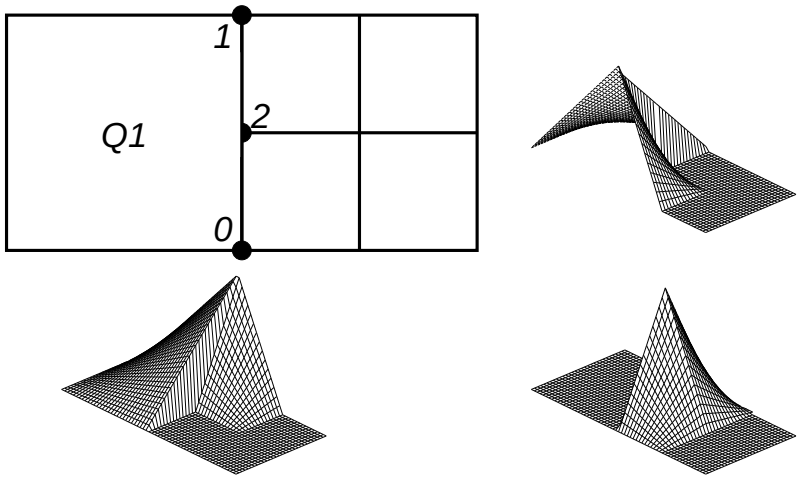

Fig. 9. *Bilinear (Q_1) shape functions on an adaptively refined mesh. Top left: Mesh and enumeration of degrees of freedom. Top right, bottom left, bottom right: Shape functions $\varphi_0, \varphi_1, \varphi_2$ as defined by mapping vertex shape functions defined on each of the cells individually to the cells to which the corresponding vertex is adjacent. Note that the shape functions are discontinuous.*

Instead, let us adopt a viewpoint dual to that considering constraints. In the approach outlined above, we obtain a system of linear equations by considering *all* shape functions φ_i and then solve it subject to constraints. We need the constraints since in general the linear combination $u_h(x) = \sum_i u_i \varphi_i(x)$ is not going to be a function that satisfies the continuity requirements of a suitable function space when hanging nodes are present. This can easily be seen on the usual Q_r shape functions, where the functions associated with hanging nodes are discontinuous and therefore not H^1-conforming; this non-conformity is shown in Fig. 9. On the other hand, the linear combination $u_h(x) = \sum_i u_i \varphi_i(x)$ is indeed a continuous function if the constraints are respected.

The alternative viewpoint is to construct a set $\tilde{\varphi}_i$ of conforming shape functions (i.e., in the case of H^1 conformity a set of continuous shape functions) from the functions φ_i that are defined locally on each cell without respect to hanging nodes. We can clearly find as many conforming shape functions $\tilde{\varphi}_i$ as there are unconstrained degrees of freedom on a mesh. For example, Fig. 10 shows the two conforming shape functions associated with the refined edge of the mesh shown in Fig. 9. For the case shown, these functions are

$$\tilde{\varphi}_0 = \varphi_0 + \frac{1}{2}\varphi_2, \qquad \tilde{\varphi}_1 = \varphi_1 + \frac{1}{2}\varphi_2.$$

In the general case, we can find the so-called "condensed" shape function $\tilde{\varphi}_i$ for each unconstrained degree of freedom as follows:

$$\tilde{\varphi}_i = \varphi_i + \sum_{j \text{ constrained DoF}} (c_j)_i \, \varphi_j,$$

where $(c_j)_i$ is the i-th component of the constraint vector corresponding to a constrained degree of freedom j. With these new, conforming shape functions, we then obtain the

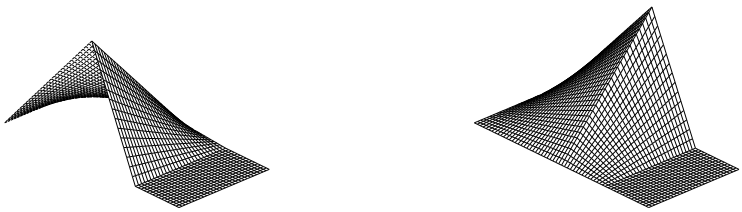

Fig. 10. *"Condensed" shape functions for the adaptive mesh shown in Fig. 9.*

"condensed" linear system $\tilde{A}U = \tilde{F}$ where

$$\tilde{a}_{ij} = \begin{cases} b(\tilde{\varphi}_i, \tilde{\varphi}_j) & \text{if degrees of freedom } i,j \text{ are both unconstrained,} \\ 1 & \text{if } i=j \text{ and degrees of freedom } i \text{ is constrained,} \\ 0 & \text{if degree of freedom } i \text{ or } j \text{ is constrained but } i \neq j. \end{cases}$$

$$\tilde{f}_i = \begin{cases} (f, \tilde{\varphi}_i) & \text{if degree of freedom } i \text{ is unconstrained,} \\ 0 & \text{if degree of freedom } i \text{ is constrained.} \end{cases}$$

The solution of this condensed linear system uniquely determines the values of those degrees of freedom that are unconstrained. The values of the constrained degrees of freedom can be obtained from the equation $CU = 0$.

The beauty of the approach lies in the fact that we can still assemble the matrix and the right hand side vectors A, F as before, i.e. using exclusively the original, possibly nonconforming shape functions that are defined on each cell without regard for the fact that they may be located on a hanging node. The condensed forms $\tilde{A}, \tilde{F}$ are then, in a second step, obtained by a condensation procedure. For example, for F, we need to take each entry F_j that belongs to a constrained degree of freedom j, then for each $0 \leq i < N$ multiply it by a factor $(c_j)_i$, and add it to row or column i. This corresponds to the operation

$$\begin{aligned} \tilde{F}_i = (f, \tilde{\varphi}_i) &= \left(f, \varphi_i + \sum_{j \text{ constrained DoF}} (c_j)_i \, \varphi_j \right) = (f, \varphi_i) + \sum_{j \text{ constrained DoF}} (c_j)_i \, (f, \varphi_j) \\ &= F_i + \sum_{j \text{ constrained DoF}} (c_j)_i \, F_j. \end{aligned}$$

Subsequently, the entries $\tilde{F}_j$ are set to zero. An algorithm to implement this is shown in Listing 2. A similar procedure can be applied to obtain $\tilde{A}$ from A, by copying and adding the rows and columns of the matrix corresponding to constrained degrees of freedom to those of the unconstrained nodes. The rows and columns are then zeroed out, and the diagonal entry is set to one to ensure regularity of the resulting matrix. At the cost of re-allocating memory and copying all entries, these rows and columns could also be eliminated from the matrix, but we do not usually do so.

Given the number $M = \mathcal{O}(N)$ of constraints in hp computations, it is important that the condensation of the matrix and right hand side vector can be performed efficiently. From Listing 2 and using that L_i is a number $\mathcal{O}(1)$ that only depends on the kind of finite elements in use and the topology of the mesh, it is clear that condensing F is an operation of complexity $\mathcal{O}(M) = \mathcal{O}(N)$.

```
for (unsigned int i=0; i<n_constrained_dofs; ++i) {
  for (unsigned int l=0; l<L_i; ++l)
    F(r_l(i)) += c_i(l) * F(q(i));
  F(q(i)) = 0;
}
```

Listing 2. *Condensing constrained degrees of freedom from a right hand side vector. The symbols* `L_i`, `c_i(l)`, *and* `q(i)` *correspond to* $L_i, (c_i)_{r_l(i)}$, *and* $q(i)$ *in equation* (7)*.*

```
for (unsigned int i=0; i<n_constrained_dofs; ++i)
  for (unsigned int j=0; j<n_dofs; ++j) {
    for (unsigned int l=0; l<L_i; ++l)
      A(r_l(i),j) += c_i(l) * A(q(i),j);
    A(q(i),j) = 0;
  }
```

Listing 3. *A naive algorithm for condensing the rows of a matrix corresponding to constrained degrees of freedom.*

The situation is more complicated when condensing matrices. First, it may be necessary to add certain elements to the sparsity pattern of the matrix. Second, care must be taken to avoid a quadratic complexity of the algorithm. Listing 3 shows a naive implementation of eliminating rows corresponding to constrained degrees of freedom. As written, the algorithms complexity is $\mathcal{O}(sMN) = \mathcal{O}(sN^2)$, where $s = \mathcal{O}(1)$ is the cost of writing to a random entry of a given row. When operating on sparse matrices, it is clear that one doesn't need to loop over all entries of a row (the j loop in the code), but only over the $\mathcal{O}(1)$ nonzero entries, reducing the complexity to $\mathcal{O}(sN)$. On the other hand, with the usual compressed row storage of sparse matrices, care must be taken to ensure that the cost s stays within reasonable bounds even for matrices with many entries per row, for example T entries per row, i.e. with $s = \mathcal{O}(\log T)$ instead of $s = \mathcal{O}(T)$.

A similar algorithm then subsequently eliminates the column that corresponds to this degree of freedom. A careful implementation of these ideas, as present in deal.II, will yield a rather complicated code that, however, runs in $\mathcal{O}(N)$ and therefore at a better complexity than most linear solvers.

6. NUMERICAL RESULTS

In this section, we present some numerical examples that demonstrate how the implementation of the ideas outlined in previous sections perform in a practical implementation. In particular, we will investigate the run-time behavior of the various steps of the hp method identified above, and implemented in the deal.II Open Source finite element library [Bangerth et al. 2008; 2007] since release 6.0. The program with which the results below are generated is a slight modification of the extensively documented step-27 tutorial program of deal.II, also included in the deal.II distribution since release 6.0. All computations were performed on a system equipped with Opteron 8216 processors and 16GB of memory.

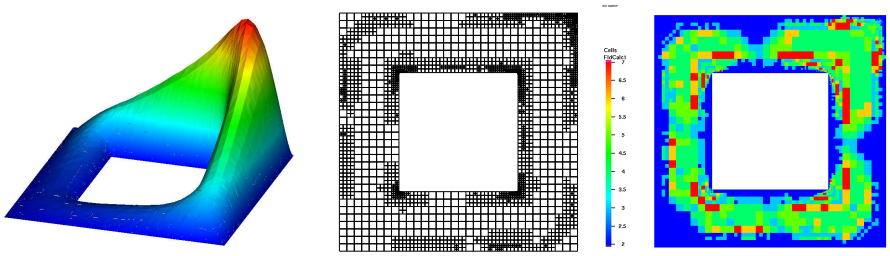

Fig. 11. *Results for example 1. Left: The solution u. Center: The mesh used for the discretization in the seventh adaptive refinement step. Right: The distribution of polynomial degrees onto cells.*

Both numerical examples solve the linear Poisson equation, $-\Delta u = f$. Since all algorithms described above are independent of the actual problem solved, it is of no further consequence that we do not solve a more complicated equation.

Example 1. In this first example, we solve on a square domain with a hole, $\Omega = [-1,1]^2 \backslash [-\frac{1}{2},\frac{1}{2}]^2$, using the right hand side $f(x,y) = (x+1)(y+1)$, and using hp finite elements Q_k with orders $2 \le k \le 8$.

The solution of this problem is shown in Fig. 11, together with the mesh after a few steps of adaptive refinement and a distribution of finite elements onto this mesh.[4] The algorithm to determine whether to refine h or increase p on a given cell uses an error indicator and a simple criterion to estimate the smoothness of the solution of this cell. Looking at the right panel of Fig. 11, we see that the polynomial degree is indeed low in the vicinity of the singularities close to re-entrant corners as well as along the boundary, and high in the interior. This corresponds to the expected smoothness properties of the solution. Whether this particular arrangement of elements is in fact optimal (which it certainly isn't) is outside the scope of this contribution: we only want to investigate how our algorithms perform for a given distribution of finite elements onto a mesh, not the optimal choice of finite elements. For details of the h refinement and p assignment algorithms, we refer to the documentation of the step-27 tutorial program [Bangerth et al. 2008].

Given this, the left panel of Fig. 12 shows the growth of the number of degrees of freedom as hp refinement iterations proceed, as well the number of constrained degrees of freedom. The latter number is roughly constant at about 20% of the total.

The right panel shows a view of where the compute time for the numerical solution of this problem is spent. The total time used on each mesh grows approximately like $\mathcal{O}(N^{1.5})$, where N is the total number of degrees of freedom; this rate can be expected for the SSOR preconditioned Conjugate Gradient solver we use in this computation. This total compute time is in fact entirely dominated by solving the linear system, which consumes more than 95% of the compute time for $N \ge 10^5$.

The rest of the time is spent on assembling the linear system (3% for N=10^5) and various other tasks. Among the hp specific activities, both allocating degrees of freedom (see the discussion in Sections 3 and 4.2) and computing constraints (see Sections 4.3–4.5 and 5.1) take negligible fractions of the total compute time, and only the elimination of constrained nodes from the linear system (see Section 5.2) is noticeable. However, even the latter

[4]The solution looks blocky since we output it as a bilinear interpolation even on cells with high polynomial degree. The actual computed solution is much more accurate than depicted.

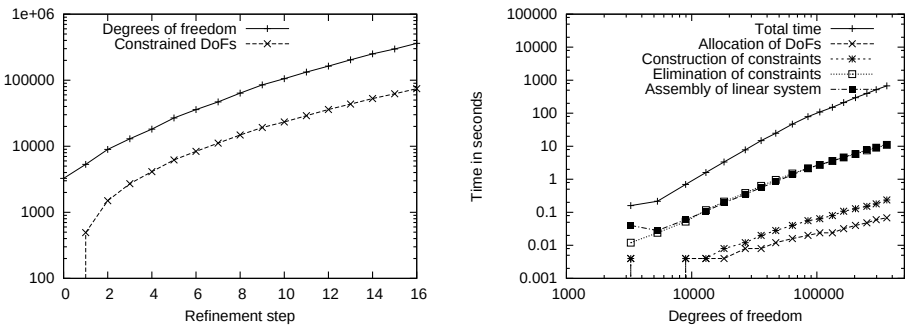

Fig. 12. *Results for example 1 for a sequence of adaptively refined meshes. Left: Growth of the total number of degrees of freedom and the number of constrained degrees of freedom as refinement iterations proceed. Right: Compute times in seconds for (i) total compute time excluding postprocessing on one mesh, (ii) allocation and identification of degrees of freedom alone, (iii) construction of constraints alone, (iv) elimination of constrained degrees of freedom from the linear system alone, and (v) assembly of the linear system alone.*

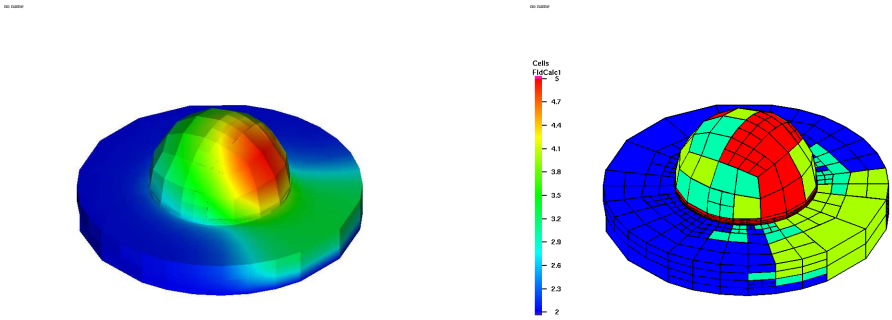

Fig. 13. *Results for example 2. Left: The solution u. Right: The mesh and distribution of polynomial degrees onto cells after five refinement steps.*

takes less than 2% of the total compute time on finer grids, and furthermore grows at a complexity of only $\mathcal{O}(N)$ and therefore slower than the overall solver process.[5]

In summary, we can conclude from this example that the hp specific algorithms do not significantly contribute to the overall compute time of the finite element solution of this problem. An obvious opportunity of improvement is clearly the simplistic linear solver, although this is outside the scope of this paper.

Example 2. In our second example, we solve on a realistic 3d domain previously already used in the simulation of breast cancer imaging [Bangerth et al. 2007; Hwang et al. 2006], see Fig. 13. As a right hand side, we use $f(x,y,z) = 1$ in the wedge $x > |y|$, and $f(x,y,z) = 0$ otherwise. We use elements Q_k with orders $2 \le k \le 5$.

Compared to the 2d example, the solver is still the most time consuming part of the program, but assembling the linear system now takes up to 22%, and eliminating con-

[5]A better fit for the data points involved is in fact $\mathcal{O}(sN)$, where $s = \log T$ with T the number of nonzero entries per row. Here T grows with refinement iterations since the average polynomial degree of shape functions on cells grows. The observation is then consistent with the estimates given in Section 5.2.

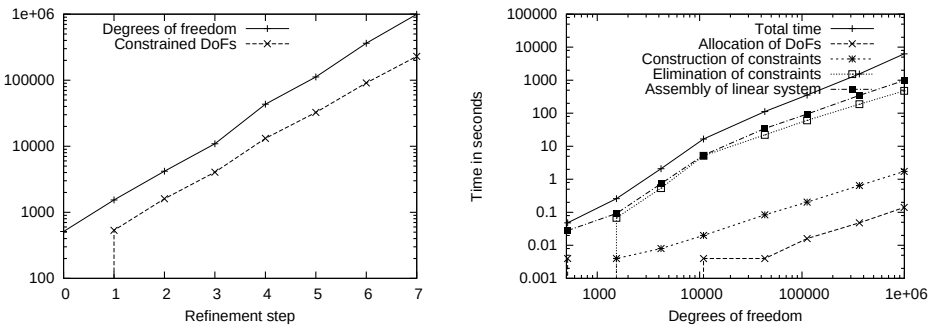

Fig. 14. *Results for example 2 for a sequence of adaptively refined meshes. Left: Growth of the total number of degrees of freedom and the number of constrained degrees of freedom as refinement iterations proceed. Right: Compute times in seconds for (i) total compute time excluding postprocessing on one mesh, (ii) allocation and identification of degrees of freedom alone, (iii) construction of constraints alone, (iv) elimination of constrained degrees of freedom from the linear system alone, and (v) assembly of the linear system alone.*

strained degrees of freedom from the matrix and the sparsity pattern takes another 11%. On the other hand, actually computing these constraints, i.e. to form the constraint matrix C discussed in Section 5.1 takes less than 0.5%, and all other tasks are also negligible.

The fact that assembling the linear system takes a significant component of the overall compute time does not come as much of a surprise, and is well known for higher-order finite elements in 3d. One of the available strategies to improve this situation is to precompute some of the matrix components, as explained for example in [Kirby and Logg 2006a; 2006b]. We were surprised, however, that eliminating constraints is so expensive. In our initial implementation, eliminating constrained degrees of freedom from the column compressed storage sparsity pattern was the largest factor in overall compute time. We consequently changed the data structures used to store an intermediate form of the sparsity pattern (see the documentation of the `CompressedSparsityPattern` and `CompressedSetSparsityPattern` classes at [Bangerth et al. 2008]) and thereby reduced the time for elimination by about a factor of 6, leading to the numbers quoted above. On the other hand, the fact that tampering with matrices and sparsity patterns is expensive should not have come entirely unexpected since in 3d a Q_5 element has 216 degrees of freedom on each cell, and a typical row in the system matrix can have more than 300 nonzero entries. Given our algorithmic improvements, it is reassuring to see that the linear solver is still the dominant part of the simulation, implying that even in 3d, hp finite elements are very much a feasible and usable technology.

7. CONCLUSIONS

The implementation of fully hp adaptive finite element methods for general classes of elements is a complicated task, sometimes rumored to be "orders of magnitude harder" than non-hp methods. While the mathematics of such methods are well described in the literature, there do not appear to be very many attempts to actually implement it beyond discontinuous Galerkin methods for which the method does not require the construction of hanging node constraints (see, however, [Ainsworth and Senior 1997; 1999]).

In the current paper, we have described the many components necessary to implement hp methods for general combinations of finite elements and both in 2d and 3d (the 1d

case is so notably absent of any particular problems that we did not discuss our implementation), and the complications and pitfalls one runs into. The techniques discussed here provide the generic algorithms that can make this implementation work not only for Q_k elements, but general combinations of elements. Actual instances of finite element classes essentially only have to describe equivalences between degrees of freedom on vertices, edges, and faces, and provide matrices that describe interpolation from one element to another on faces and subfaces between cells. Beyond that, the generic algorithms discussed can work independently of the actual elements involved. In particular, this includes $H(\text{div})$ [Brezzi and Fortin 1991] and $H(\text{curl})$ [Nedelec 1980] elements, but also immediately vector-valued elements for problems with more than one solution variable.

In the final section of this paper, we also demonstrated that our algorithms are efficient, i.e. that they are cheap compared to the expensive parts of finite element programs: assembling linear systems and solving them. This demonstrates that it is possible to implement hp finite elements efficiently, even for continuous and 3d elements. A reference implementation of our ideas, as well as the tutorial program step-27 explaining the use of hp adaptivity, is available as part of the Open Source finite element library deal.II [Bangerth et al. 2008] since release 6.0.

Finally, we can also address the question whether hp is hard to implement: we estimate that to fully address the problems discussed in this paper, we had to implement less than 20,000 lines of code on top of what deal.II already had to offer. This is comparable to probably less than one year of work for a skilled and trained individual already familiar with the internals of deal.II. This has to be compared to a total of roughly 360,000 lines of code presently in deal.II, of which maybe 100,000 are part of the low-level core that deals with meshes, degrees of freedom, and finite elements. In other words, while a significant and certainly non-trivial task, the implementation of the ideas in this paper is clearly not "orders of magnitude" more difficult than a reasonably general implementation of the finite element method.

REFERENCES

AINSWORTH, M. AND SENIOR, B. 1997. Aspects of an adaptive hp-finite element method: Adaptive strategy, conforming approximation and efficient solvers. *Comput. Methods Appl. Mech. Engrg. 150*, 65–87.

AINSWORTH, M. AND SENIOR, B. 1999. hp-finite element procedures on non-uniform geometric meshes: Adaptivity and constrained approximation. In *Grid Generation and Adaptive Algorithms*, M. Bern, J. Flaherty, and M. Luskin, Eds. Number 113 in IMA Vol. Math. Appl. Springer, 1–27.

BABUŠKA, I. 1981. Error estimates for the combined h and p version of finite element method. *Numer. Math. 37*, 252–277.

BANGERTH, W., HARTMANN, R., AND KANSCHAT, G. 2007. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw. 33*, 4, 24/1–24/27.

BANGERTH, W., HARTMANN, R., AND KANSCHAT, G. 2008. `deal.II` *Differential Equations Analysis Library, Technical Reference.* `http://www.dealii.org/`.

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

CAREY, G. F. 1997. *Computational Grids: Generation, Adaptation and Solution Strategies*. Taylor & Francis.

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*, First ed. Studies in Mathematics and its Applications, vol. 4. North-Holland, Amsterdam, New York, Oxford.

DEMKOWICZ, L. 2006. *Computing with hp-adaptive finite elements. Volume 1: One and Two Dimensional Elliptic and Maxwell Problems.* Chapman & Hall.

FRAUENFELDER, P. AND LAGE, C. 2002. Concepts – An object-oriented software package for partial differential equations. *M2AN 36*, 937–951.

GILBARG, D. AND TRUDINGER, N. S. 1983. *Elliptic Partial Differential Equations of Second Order*, Second ed. Springer, Heidelberg.

GUO, B. AND BABUŠKA, I. 1986a. The h-p version of the finite element method. Part I: The basic approximation results. *Comp. Mech. 1*, 21–41.

GUO, B. AND BABUŠKA, I. 1986b. The h-p version of the finite element method. Part II: The general results and application. *Comp. Mech. 1*, 203–220.

HOUSTON, P., SCHÖTZAU, D., AND WIHLER, T. P. 2007. Energy norm a posteriori error estimation of hp-adaptive discontinuous galerkin methods for elliptic problems. *M3AS 17*, 33–62.

HOUSTON, P. AND SÜLI, E. 2005. A note on the design of hp-adaptive finite element methods for elliptic partial differential equations. *Comp. Meth. Appl. Mech. Engrg. 194*, 229–243.

HOUSTON, P., SÜLI, E., AND WIHLER, T. P. 2008. A posteriori error analysis of hp-version discontinuous Galerkin finite element methods for second-order quasilinear elliptic problems. *IMA J. Numer. Anal., in press*.

HWANG, K., PAN, T., JOSHI, A., RASMUSSEN, J. C., BANGERTH, W., AND SEVICK-MURACA, E. M. 2006. Influence of excitation light rejection on forward model mismatch in optical tomography. *Phys. Med. Biol. 51*, 5889–5902.

KIRBY, R. AND LOGG, A. 2006a. A compiler for variational forms. *ACM Trans. Math. Softw. 32*, 417–444.

KIRBY, R. AND LOGG, A. 2006b. Optimizing the FEniCS form compiler FFC: Efficient pretabulation of integrals. *submitted to ACM Trans. Math. Softw.*.

KIRK, B., PETERSON, J. W., STOGNER, R. H., AND CAREY, G. F. 2006. `libMesh`: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations. *Engineering with Computers 22*, 3–4, 237–254.

NEDELEC, J.-C. 1980. Mixed finite elements in R^3. *Numer. Math. 35*, 315–341.

PESCH, L., BELL, A., SOLLIE, H., AMBATI, V. R., BOKHOVE, O., AND VAN DER VEGT, J. W. 2007. hpGEM – A software framework for discontinuous Galerkin finite element methods. *ACM Trans. Math. Softw. 33*, 4, 23/1–23/25.

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

RIVARA, M. C. 1984. Mesh refinement processes based on the generalized bisection of simplices. *SIAM J. Numer. Anal. 21*, 604–613.

ŠOLÍN, P., ČERVENÝ, J., AND DOLEŽEL, I. 2008. Arbitrary-level hanging nodes and automatic adaptivity in the hp-FEM. *Math. Comput. Sim. 77*, 117–132.

ŠOLÍN, P., SEGETH, K., AND DOLEŽEL, I. 2003. *Higher-Order Finite Element Methods*. Chapman & Hall/CRC.