UNIVERSITY OF WEST BOHEMIA
FACULTY OF ELECTRICAL ENGINEERING
DEPARTMENT OF THEORY OF ELECTRICAL ENGINEERING

Jakub Červený

AUTOMATIC *hp*-ADAPTIVITY FOR TIME-DEPENDENT PROBLEMS

DOCTORAL THESIS

Advisor: Prof. Ing. Ivo Doležel, CSc.
Co-advisor: RNDr. Pavel Šolín, Ph.D.

Plzeň, 2011

Acknowledgements

I would first like to thank my advisor, Prof. Ing. Ivo Doležel, CSc. for all his support, both at the Institute of Thermomechanics at the Academy of Sciences in Prague and at the KTE FEL at the University of West Bohemia in Plzeň, during the many years of my PhD. studies. Without his constant encouragement and endless patience I would never have finished this work.

Many thanks also go to my co-advisor, RNDr. Pavel Šolín, Ph.D. for giving me the opportunity to work on interesting projects in his *hp*-FEM group at the University of Texas at El Paso. Most results in this work were obtained during the three years of my stay there, which I thoroughly enjoyed. I would also like to thank Dr. Šolín for his insistence on making the results published in international journals and presented at relevant conferences. I was only able to appreciate this effort much later. Finally, Dr. Šolín wrote two excellent monographs [45, 43] which helped me learn the basics of the finite element method and introduced me to numerical analysis in general.

I would like to thank Dr. William F. Mitchell for sending me the data for the comparison graphs from his survey [35].

I am also thankful to my colleagues from the *hp*-FEM group, Martin Zítka, David Andrš, Pavel Kůs and Lenka Dubcová, for creating a friendly and inspiring working environment. Lenka deserves a special mention for her valuable contributions to Hermes2D, for always being ready to help me with mathematical issues, for being a great person and for eventually becoming my life partner. I love you :-)

This work and related code was developed solely using open-source software. I would like to use this opportunity to appreciate the work of the thousands of individuals who created, made free and continue to maintain great projects like the GNU/Linux, GCC, UMFPACK, Octave, LaTeX, Inkscape and many more, which were all indispensable during my work on this thesis.

Abstract

This dissertation focuses on the development of algorithms for automatically adaptive *hp*-FEM that can be used to solve both stationary and time-dependent partial differential equations (PDEs) in two spatial dimensions. The *hp*-FEM is an advanced version of the classical finite element method which employs elements of varying diameter h and polynomial degree p to obtain superior (exponential) convergence rates. However, the method puts high demands on its implementation and presents a number of open problems.

In this work we review the basics of 2D *hp*-FEM and then show how to extend a standard *hp*-FEM solver to support meshes with hanging nodes, which is a prerequisite for automatic *hp*-adaptation of the approximate solution. Our original algorithm and data structure enable the use of arbitrarily irregular meshes that can reduce both the size of the discrete problem and the complexity of the *hp*-adaptation algorithm. Practical implementation details and examples are included.

Next we review several existing *hp*-adaptation strategies for stationary PDEs, in particular an existing algorithm based on the reference solution approach. We design a new algorithm that is both simpler and faster, while delivering better or comparable results, as we demonstrate on two standard benchmark problems.

The next topic is the solution of systems of PDEs. We motivate the use of different meshes for different equations in the system and present an original algorithm for the assembly of the stiffness matrix of such multi-mesh systems. The goal is to save degrees of freedom and to prepare the solver for dynamic meshes in time-dependent PDEs. We test the implementation on a model problem of thermoelasticity.

Finally, we use the new multi-mesh assembling and the adaptive Rothe method to obtain computations with meshes that can change with time, in order to speed up the solution of time-dependent problems that exhibit moving features in their solution. We develop an algorithm that automatically adjusts the mesh between successive time steps and test it on two nonlinear model problems from the areas of incompressible fluid flow and combustion physics.

Anotace

Tato dizertace se zaměřuje na návrh algoritmů pro automaticky adaptivní *hp*-FEM pro účely řešení stacionárních i časově závislých parciálních diferenciálních rovnic (PDR) ve dvou prostorových rozměrech. Metoda *hp*-FEM je zdokonalená verze klasické metody konečných prvků, která využívá elementy různých poloměrů h a polynomiálních stupňů p k dosažení vynikající (exponenciální) rychlosti konvergence. Klade ovšem velké nároky na implementační stránku a přináší řadu otevřených problémů.

V této práci shrnujeme základy *hp*-FEM ve 2D a poté popisujeme rozšíření standardního řešiče *hp*-FEM o podporu sítí s visícími uzly, která je předpokladem pro automatickou *hp*-adaptaci přibližného řešení. Náš původní algoritmus a datová struktura umožňují použití libovolně neregulárních sítí, jež mohou vést ke zmenšení diskrétního problému a ke zjednodušení algoritmu pro *hp*-adaptivitu. Popisujeme i praktické implementační detaily a příklady.

Dále shrnujeme několik existujících strategií *hp*-adaptivity pro stacionární PDR, zejména existující algoritmus založený na tzv. referenčním řešení. Navrhujeme nový algoritmus, který je jednodušší a rychlejší, přičemž ale dosahuje lepších nebo srovnatelných výsledků, jak ukazujeme na dvou standardních testovacích problémech.

Dalším tématem je řešení soustav PDR. Obhajujeme možnost použití různých sítí pro různé rovnice v soustavě a přinášíme původní algoritmus pro sestavení matice tuhosti takového systému (tzv. multi-mesh assembling). Cílem je úspora stupňů volnosti a příprava řešiče na dynamické sítě u časově závislých rovnic. Implementaci testujeme na modelovém příkladu z termoelasticity.

Nový algoritmus pro multi-mesh assembling v závěru využíváme spolu s adaptivní Rotheho metodou k výpočtům na sítích, které se mohou měnit v čase, za účelem zrychlení řešení časově závislých problémů, které vykazují pohybující se úkazy v jejich řešení. Vyvíjíme algoritmus, který dokáže automaticky upravovat síť mezi jednotlivými časovými kroky a testujeme jej na dvou nelineárních modelových problémech z oblastí nestlačitelného proudění a fyziky hoření.

I hereby declare that this Ph.D. dissertation is completely my own work and that I used only the cited sources.

Prague, September 2011 Jakub Červený

Contents

Chapter 1

Introduction

Today, the development of most fields of engineering and natural sciences would be unthinkable without the use of computers. Computer simulation has become an essential part of electric, mechanical and civil engineering as well as of many areas of physics, chemistry and biology. Indeed, computational science is now regarded as the third mode of science, complementing the two classical domains of experimentation and theory.

Many natural and engineering processes can be described and modeled by partial differential equations (PDEs), and their numerical solution has been one of the first and most important applications of computers since their invention. Among the most prominent numerical methods for the solution of PDEs is the hugely popular finite element method (FEM), now widely used in countless applications. Due to the increasing power of computers and an ever-growing demand on the accuracy and scalability of the numerical methods, a number of variants of the FEM have been developed over the past decades.

This thesis is concerned with adaptive *hp*-FEM, a particular modern version of the finite element method with outstanding theoretical properties, but at the same time presenting many open problems and practical difficulties. In this thesis we review the theoretical foundations of *hp*-FEM in 2D, develop novel *hp*-adaptive algorithms for both stationary and time-dependent PDEs based on arbitrary-level hanging nodes and multi-mesh assembling, provide practical implementation details and data structures and finally present numerical examples demonstrating the performance of the methods on several model problems.

1.1 History of *hp*-FEM

The origins of the finite element method can be traced back to the 1940s to the works of Courant [14], who himself was relying on earlier developments of variational methods for PDEs by Rayleigh, Ritz and Galerkin. By the end of the 1950s the key concepts of the FEM had stabilized to the form used today and during the 1960s the method was already being routinely used in a wide variety of engineering disciplines, e.g., electromagnetism, fluid dynamics and structural analysis [56]. In the 1970s a rigorous mathematical foundation for the method was provided by Ciarlet [13] and Strang and Fix [49]. The essential characteristic of the method is the division of the computational domain into a finite number of polygonal sub-regions, called elements. The ability of the method to handle complex, non-rectangular geometries is one of the most attractive advantages over methods based on finite differences (FDM).

The accuracy of the approximate solution, within the Galerkin framework, depends solely on the choice of the finite-dimensional subspace, i.e., on the size and type of elements in the mesh. This is especially true for problems exhibiting singularities and multiscale behavior. The need for mesh adaptation and error estimation was quickly recognized and the h-adaptive version or h-FEM was born. In h-FEM problematic parts of the solution are resolved by successive subdivision of elements on which the estimated error is too large. A posteriori error estimation, an essential part of all adaptive schemes, has since become a vast research area of its own. For some of the first works, see [3, 4]. Although h-adaptivity has proved extremely useful for many problems with singularities, the approximation error can typically only be reduced with some negative power of the number of degrees of freedom.

An alternative approach to the enlargement of the finite-element subspace is based on increasing the polynomial degree of the approximation on selected elements. This scheme is referred to as p-FEM or p-adaptivity and is more successful than h-FEM for problems with a very smooth solution. However, it yields unsatisfactory convergence rates for problems with singularities. This observation led Babuška and Guo to propose the hp version of the FEM [5, 28, 29], in which a higher polynomial degree p is selected for elements where the solution is sufficiently smooth, while the element size h is reduced near oscillations and singularities. It was proved that unlike the previous methods, hp-FEM is capable of delivering exponential convergence rates, which should in theory make it the best choice for most finite element computations.

However, more than 20 years have passed and even though hp-FEM has been extensively studied in the mathematical literature, it is seldom ever used outside academia. The main reason is the lack of a reliable and widely accepted a posteriori error estimator that would provide the information whether to split an element or increase its polynomial degree. There has been a lot of research in this area (see [26, 35] and the references therein), but in practice one usually has to resort to guiding the hp-adaptivity by the costly reference solution, obtained by uniform h- and p- refinements of the mesh. Another reason why hp-FEM has not yet been adopted by the engineering community is the common belief that it is very hard to implement.

There are currently not many publicly available *hp*-FEM implementations. Among the most well known are the Fortran codes 2Dhp90 and 3Dhp90[1] by L. Demkowicz and his students, the C++ library Concepts[2] by P. Frauenfelder and C. Lage and the package deal.II[3] by W. Bangerth and coworkers. Finally, let us mention the projects Hermes2D[4] and Hermes3D by the group of P. Šolín, portions of which are the result of this work.

1.2 Objectives of This Work

The following is a list of tasks this thesis focuses on. Each task is covered in one of the following chapters.

1. Review the fundamentals of the theory of both standard FEM and higher-order FEM, nodal and hierarchic, with implementation details.
2. Review existing algorithms for the treatment of meshes with hanging nodes and extend them to the case of meshes with arbitrary-level hanging nodes. Provide practical information and data structures.
3. Summarize existing *hp*-adaptivity algorithms for stationary problems and design a simpler and faster alternative. Assess the performance of the new algorithm.
4. Develop a new assembling method enabling the use of different meshes for different components of the solution in a system of PDEs. This is important for efficient solution of coupled problems and will allow the solution of time-dependent problems on dynamic meshes.
5. Develop a new automatic *hp*-adaptive algorithm for time-dependent problems on dynamically changing meshes.
6. Demonstrate the validity and performance of the presented algorithms and methods on several numerical examples.

Original results of the author are presented in Chapters 3, 4, 5 and 6. The main concepts were developed during the author's stay at the University of Texas at El Paso and at the Institute of Thermomechanics at the Academy of Sciences of the Czech Republic.

[1] `http://users.ices.utexas.edu/~leszek/projects.html`
[2] `http://www.concepts.math.ethz.ch/`
[3] `http://www.dealii.org/`
[4] `http://hpfem.org/hermes2d/`

Chapter 2

Higher-Order FEM

This chapter presents a brief overview of the theoretical fundamentals of the finite element method, starting with classical piecewise-linear elements and later introducing both the nodal and hierarchic approaches to higher-order elements. The chapter concludes with several implementation remarks. For those seriously interested in the study of higher-order FEM we recommend two monographs [45] and [43], upon which this chapter is based.

2.1 Weak Formulation of a Model Problem

Let us start with a model equation representing a range of standard stationary problems. Let Ω be an open connected set in $\boldsymbol{R}^d$. Consider the equation

$$-\nabla(a_1\nabla u) + a_0 u = f \qquad \text{in } \Omega. \tag{2.1}$$

For the existence and uniqueness of solution we add the assumptions

$$a_1(\boldsymbol{x}) \geq C_{min} > 0 \quad \text{and} \quad a_0(\boldsymbol{x}) \geq 0 \quad \text{in } \Omega.$$

Let (2.1) be endowed with *homogeneous Dirichlet* boundary conditions

$$u(\boldsymbol{x}) = 0 \qquad \text{on } \partial\Omega. \tag{2.2}$$

Classical solution to the problem (2.1), (2.2) is a function $u \in C^2(\Omega) \cap C(\overline{\Omega})$ satisfying the equation (2.1) everywhere in Ω and fulfilling the boundary condition (2.2) at every $\boldsymbol{x} \in \partial\Omega$. Naturally, one has to assume that $f \in C(\Omega)$. However, neither this nor even stronger requirement $f \in C(\overline{\Omega})$ guarantees the solvability of the problem, for which still stronger smoothness of f is required.

In order to reduce the above-mentioned regularity restrictions, we introduce the *weak formulation* of the problem (2.1), (2.2). The derivation of the weak formulation of (2.1) consists of the following four steps:

1. Multiply (2.1) with a *test function* $v \in C_0^\infty(\Omega)$,
$$-\nabla(a_1\nabla u)v + a_0uv = fv.$$
2. Integrate over Ω,
$$-\int_\Omega \nabla(a_1\nabla u)v\,\mathrm{d}\boldsymbol{x} + \int_\Omega a_0uv\,\mathrm{d}\boldsymbol{x} = \int_\Omega fv\,\mathrm{d}\boldsymbol{x}.$$
3. Use the Green's formula to reduce the maximum order of the partial derivatives present in the equation. The fact that v vanishes on the boundary $\partial\Omega$ removes the boundary term, and we have
$$\int_\Omega a_1\nabla u\cdot\nabla v\,\mathrm{d}\boldsymbol{x} + \int_\Omega a_0uv\,\mathrm{d}\boldsymbol{x} = \int_\Omega fv\,\mathrm{d}\boldsymbol{x}. \tag{2.3}$$
4. Find the largest possible function spaces for u, v and other functions in (2.3) so that all integrals are still defined. The identity (2.3) was derived under very strong regularity assumptions $u \in C^2(\Omega)\cap C(\overline{\Omega})$ and $v \in C_0^\infty(\Omega)$. Notice that all integrals in (2.3) still make sense if these assumptions are weakened to
$$u, v \in H_0^1(\Omega), \qquad f \in L^2(\Omega),$$
where $H_0^1(\Omega)$ is the Sobolev space $W_0^{1,2}(\Omega)$. Similarly the regularity assumptions for the coefficients a_1 and a_0 can be reduced to
$$a_1, a_0 \in L^\infty(\Omega).$$

The weak form of the problem (2.1), (2.2) is stated as follows: Given $f \in L^2(\Omega)$, find a function $u \in H_0^1(\Omega)$ such that
$$\int_\Omega a_1\nabla u\cdot\nabla v + a_0uv\,\mathrm{d}\boldsymbol{x} = \int_\Omega fv\,\mathrm{d}\boldsymbol{x} \quad \text{for all } v \in H_0^1(\Omega). \tag{2.4}$$
Obviously the classical solution to the problem (2.1), (2.2) also solves the weak formulation (2.4). Conversely, if the weak solution of (2.4) is sufficiently regular, which in this case means $u \in C^2(\Omega)\cup C(\overline{\Omega})$, it also satisfies the classical formulation (2.1), (2.2).

Let $V = H_0^1(\Omega)$. In the language of linear forms, we define a bilinear form $a(\cdot,\cdot) : V \times V \to \boldsymbol{R}$,
$$a(u,v) = \int_\Omega (a_1\nabla u\cdot\nabla v + a_0uv)\,\mathrm{d}\boldsymbol{x}, \tag{2.5}$$
and a linear form $l \in V'$,
$$l(v) = \int_\Omega fv\,\mathrm{d}\boldsymbol{x}.$$
Then the weak formulation of the problem (2.1), (2.2) reads: Find a function $u \in V$ such that
$$a(u,v) = l(v) \qquad \text{for all } v \in V. \tag{2.6}$$

Definition 2.1 *Let V be a real Hilbert space and $a : V \times V \to \boldsymbol{R}$ a bilinear form. We say that*

1. *a is* bounded *if there exists a constant $C_a > 0$ such that $|a(u,v)| \leq C_a\|u\|\|v\|$ for all $u, v \in V$,*
2. *a is V-*elliptic (coercive) *if there exists a constant $\tilde{C}_a > 0$ such that $a(v,v) \geq \tilde{C}_a\|v\|^2$ for all $v \in V$.*

The following important result asserts the existence and uniqueness of the solution to the weak formulation of our problem.

Theorem 2.1 (Lax-Milgram lemma) *Let V be a Hilbert space, $a : V \times V \to \boldsymbol{R}$ a bounded V-elliptic bilinear form and $l \in V'$. Then there exists a unique solution to the problem*

$$a(u,v) = l(v) \qquad \textit{for all } v \in V.$$

Proof See [43].

2.2 The Galerkin Method

The problem (2.6) is stated in an infinitely-dimensional space, and therefore in most cases it is not possible to find its exact solution. The Galerkin method constructs a sequence of finite-dimensional subspaces $\{V_n\}_{n=1}^{\infty} \subset V$, $V_n \subset V_{n+1}$, that fill the space V in the limit. In each finite-dimensional space V_n the problem (2.6) is solved exactly. It can be shown that under suitable assumptions the sequence of the *approximate solutions* $\{u_n\}_{n=1}^{\infty}$, $u_n \in V_n$, converges to the exact solution of the problem (2.6).

The Galerkin approximate problem is usually called *discrete problem* and it reads: find a solution $u_n \in V_n$, satisfying

$$a(u_n, v) = l(v) \quad \text{for all } v \in V_n. \tag{2.7}$$

The solution $u_n \in V_n$ to the discrete problem (2.7) can be found explicitly thanks to the fact that the space V_n has a finite basis $\{v_n\}_{n=1}^{N_n}$. The solution u_n can be written as a linear combination of these basis functions with unknown coefficients,

$$u_n = \sum_{j=1}^{N_n} y_j v_j. \tag{2.8}$$

Substituting (2.8) into (2.7), one obtains

$$a\left(\sum_{j=1}^{n} y_j v_j, v\right) = l(v) \quad \text{for all } v \in V_n.$$

The linearity of $a(\cdot,\cdot)$ in its first component yields

$$\sum_{j=1}^{N_n} a\left(v_j, v\right) y_j = l(v) \quad \text{for all } v \in V_n. \tag{2.9}$$

Substituting the basis functions $v_1, v_2, \ldots, v_{N_n}$ for v in (2.9), we obtain

$$\sum_{j=1}^{N_n} a\left(v_j, v_i\right) y_j = l(v_i) \quad i = 1, 2, \ldots, N_n. \tag{2.10}$$

Let us define the stiffness matrix

$$\boldsymbol{S}_n = \{s_{ij}\}_{i,j=1}^{N_n}, \quad s_{ij} = a(v_j, v_i), \tag{2.11}$$

the load vector

$$\boldsymbol{F}_n = \{f_i\}_{i=1}^{N_n}, \quad f_i = l(v_i), \tag{2.12}$$

and the unknown coefficient vector

$$\boldsymbol{Y}_n = \{y_i\}_{i=1}^{N_n}.$$

Then the system of linear algebraic equations (2.10) can be written in matrix form:

$$\boldsymbol{S}_n \boldsymbol{Y}_n = \boldsymbol{F}_n.$$

2.3 Standard Linear Elements

Consider the model problem (2.1), (2.2). The domain Ω is first approximated by a polygonal domain Ω_h. This is one of the so-called *variational crimes* – departures from the "mathematically clean" variational framework – since $\Omega_h \not\subset \Omega$ and the solution or other functions from the weak formulation (2.6) are not defined where they are to be approximated or evaluated. In practice these crimes are often hard to avoid. The next step consists of defining the finite element mesh:

Definition 2.2 (Finite element mesh) Finite element mesh $\mathcal{T}_h = \{K_1, K_2, \ldots, K_M\}$ *over a domain* $\Omega_h \in \boldsymbol{R}^d$ *with a piecewise-polynomial boundary is a geometrical division of* Ω_h *into a finite number of nonoverlapping open polygonal cells* K_i *such that*

$$\Omega_h = \bigcup_{i=1}^{M} \overline{K}_i$$

A standard requirement, which greatly facilitates the discretization procedure of non-adaptive FEM, is mesh regularity. In this chapter we only work with regular meshes.

Definition 2.3 (Regular mesh) *The mesh is called* regular *if for any two elements K_i and K_j, $i \neq j$ only one of the following alternatives holds:*

- *$\overline{K}_i \cap \overline{K}_j$ is empty*
- *$\overline{K}_i \cap \overline{K}_j$ is a single common vertex*
- *$\overline{K}_i \cap \overline{K}_j$ is a single (whole) common edge*

According to the geometrical approximation $\Omega_h \approx \Omega$, the space $V(\Omega)$ is approximated by a piecewise-linear polynomial space $V_h(\Omega_h)$,

$$V_h = \{v \in C(\Omega_h);\ v|_{\partial\Omega} = 0;\ v|_{K_i} \in \text{span}\{1, x, y\}\ \}$$

The basis functions of V_h have the form of "pyramids", as shown in Figure 2.1.

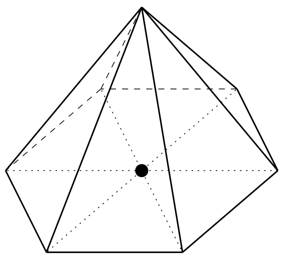


Figure 2.1: Basis function of the space V_h.

These functions are defined as follows: assume a vertex patch $S(i)$ consisting of all mesh triangles sharing the vertex $\boldsymbol{x}_i$,

$$S(i) = \bigcup_{k \in N(i)} \overline{K}_k,$$

where the index set $N(i)$ is defined as

$$N(i) = \{k;\ K_k \in \mathcal{T}_h,\ \boldsymbol{x}_i \text{ is a vertex of } K_k\}.$$

The basis function v_i is defined to be zero in $\Omega_h \setminus S(i)$, and in $S(i)$ it has the form

$$v_i(\boldsymbol{x})|_{K_k} = (\varphi^{v_r} \circ \boldsymbol{x}_{K_k}^{-1})(\boldsymbol{x})$$

Here for every element $K_k \in S(i)$, φ^{v_r} is the unique vertex *shape function* on the reference domain K_t, such that $\varphi^{v_r}(\boldsymbol{x}_{K_k}^{-1}(\boldsymbol{x}_i)) = 1$ (see Figure 2.2). The reference domain K_t and reference domain map $\boldsymbol{x}_{K_k}$ will be defined in the following section.

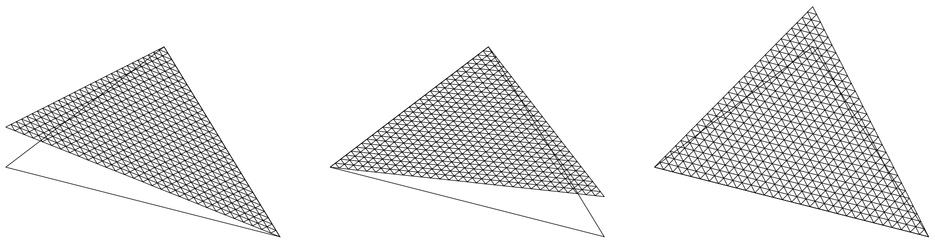


Figure 2.2: Vertex shape functions φ^{v_1}, φ^{v_2}, φ^{v_3}.

2.4 Affine Concept of the FEM

The basic idea of the affine concept is to transform the weak formulation from each mesh element K_i to a triangular reference domain K_t, where all the computation takes place. This approach is also efficient from the point of view of computer memory, since the numerical quadrature data is stored on the reference domain only. The shape functions and their partial derivatives can be stored via their values at integration points in the reference domain.

The standard reference domain K_t is shown in Figure 2.3.

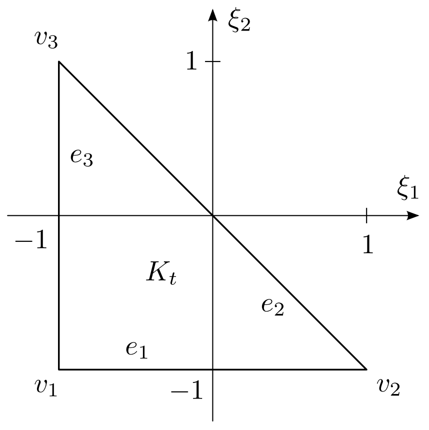


Figure 2.3: Reference triangle K_t.

Consider now an arbitrary triangular domain $K \subset \boldsymbol{R}^2$ with the vertices $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, $\boldsymbol{x}_3$. The isoparametric reference map $\boldsymbol{x}_K : K_t \to K$ is defined as

$$\boldsymbol{x}_K(\boldsymbol{\xi}) = \sum_{i=1}^{3} \boldsymbol{x}_i \varphi^{v_i}(\boldsymbol{\xi}),$$

where φ^{v_i} are the vertex shape functions shown in Figure 2.2 and defined as

$$\varphi^{v_1}(\boldsymbol{\xi}) \quad = \quad -\frac{\xi_1 + \xi_2}{2},$$

$$\begin{aligned}\varphi^{v_2}(\boldsymbol{\xi}) &= \frac{1+\xi_1}{2},\\ \varphi^{v_3}(\boldsymbol{\xi}) &= \frac{1+\xi_2}{2}.\end{aligned}$$

All integrals in (2.11), (2.12) are first transformed to the reference domain K_t and then evaluated using Gaussian quadrature (see Section 2.9). Transformation of function values is straightforward. Let $w(\boldsymbol{x}) \in C(K), 1 \le m \le M$, then its transformed values are

$$\tilde{w}(\boldsymbol{\xi}) = (w \circ \boldsymbol{x}_K)(\boldsymbol{\xi}) = w(x_{K,1}(\boldsymbol{\xi}), x_{K,2}(\boldsymbol{\xi})).$$

One has to be more careful when transforming derivatives. Using the chain rule of differentiation we obtain

$$\begin{pmatrix} \dfrac{\partial \tilde{w}}{\partial \xi_1} \\ \dfrac{\partial \tilde{w}}{\partial \xi_2} \end{pmatrix} = \begin{pmatrix} \dfrac{\partial x_{K,1}}{\partial \xi_1} & \dfrac{\partial x_{K,2}}{\partial \xi_1} \\ \dfrac{\partial x_{K,1}}{\partial \xi_2} & \dfrac{\partial x_{K,2}}{\partial \xi_2} \end{pmatrix} \begin{pmatrix} \dfrac{\partial w}{\partial x_1} \\ \dfrac{\partial w}{\partial x_2} \end{pmatrix} = \left(\frac{D\boldsymbol{x}_K}{D\boldsymbol{\xi}}\right)^T \begin{pmatrix} \dfrac{\partial w}{\partial x_1} \\ \dfrac{\partial w}{\partial x_2} \end{pmatrix},$$

where $D\boldsymbol{x}_K/D\boldsymbol{\xi}$ stands for the Jacobi matrix of the map $\boldsymbol{x}_K$. Since nonsingular matrices satisfy $(\boldsymbol{A}^T)^{-1} = (\boldsymbol{A}^{-1})^T = \boldsymbol{A}^{-T}$, the gradient $\nabla w(\boldsymbol{x})$ at an arbitrary point $\boldsymbol{x} \in K$ is transformed to the point $\boldsymbol{\xi} = \boldsymbol{x}_K^{-1}(\boldsymbol{x}) \in K_t$ by

$$\nabla w(\boldsymbol{x}) = \left(\frac{D\boldsymbol{x}_K}{D\boldsymbol{\xi}}\right)^{-T} \nabla \tilde{w}(\boldsymbol{\xi}).$$

By applying the substitution theorem, the integral in (2.11) for the model problem is then transformed as

$$\begin{aligned}&\int_K \left(a_1(\boldsymbol{x})\nabla v_j(\boldsymbol{x}) \cdot \nabla v_i(\boldsymbol{x}) + a_0(\boldsymbol{x}) v_j(\boldsymbol{x}) v_i(\boldsymbol{x})\right) \mathrm{d}\boldsymbol{x}\\ &= \int_{K_t} J_K \left[\tilde{a}_1(\boldsymbol{\xi}) \left(\frac{D\boldsymbol{x}_K}{D\boldsymbol{\xi}}\right)^{-T} \nabla \tilde{v}_j(\boldsymbol{\xi})\right] \cdot \left[\left(\frac{D\boldsymbol{x}_K}{D\boldsymbol{\xi}}\right)^{-T} \nabla \tilde{v}_i(\boldsymbol{\xi})\right] \mathrm{d}\boldsymbol{\xi}\\ &\quad + \int_{K_t} \left[J_K \tilde{a}_0(\boldsymbol{\xi}) \tilde{v}_j(\boldsymbol{\xi}) \tilde{v}_i(\boldsymbol{\xi})\right] \mathrm{d}\boldsymbol{\xi},\end{aligned}$$

where

$$\begin{gathered}\tilde{v}_i(\boldsymbol{\xi}) = (v_i \circ \boldsymbol{x}_K)(\boldsymbol{\xi}), \quad \tilde{v}_j(\boldsymbol{\xi}) = (v_j \circ \boldsymbol{x}_K)(\boldsymbol{\xi}),\\ \tilde{a}_0(\boldsymbol{\xi}) = (a_0 \circ \boldsymbol{x}_K)(\boldsymbol{\xi}), \quad \tilde{a}_1(\boldsymbol{\xi}) = (a_1 \circ \boldsymbol{x}_K)(\boldsymbol{\xi}),\\ J_K = \det\left(\frac{D\boldsymbol{x}_K}{D\boldsymbol{\xi}}\right).\end{gathered}$$

2.5 Higher-Order Finite Element Space

In Section 2.3 we constructed the Galerkin sequence $V_1 \subset V_2 \ldots$ in the Sobolev space V by refining the mesh of linear elements (h-refinement). Sometimes, much faster convergence can be achieved by increasing the polynomial degree of the elements instead (p-refinement). Such approach is usually more efficient for elements where the solution is smooth.

Let the domain Ω_h be covered with a mesh $\mathcal{T}_{h,p} = \{K_1,\, K_2, \ldots, K_M\}$ where the elements K_m carry arbitrary polynomial degrees $1 \leq p_m$, $m = 1, 2, \ldots, M$. The space $V(\Omega)$ is now approximated by a picewise-polynomial space $V_{h,p}(\Omega_h)$,

$$V_{h,p} = \{v \in C(\Omega_h);\ v|_{K_m} \in P^{p_m}(K_m) \text{ for all } 1 \leq m \leq M\}$$

where P^p is defined as

$$P^p = \text{span}\{1, x, y, x^2y, xy^2, \ldots, x^my^n, \ldots, x^py^p\} \text{ for all } 0 \leq m, n \leq p$$

In addition to *vertex basis functions* (Figure 2.1), the piece-wise polynomial space $V_{h,p}$ will be generated on the mesh edges by *edge basis functions* and on element interiors by *bubble basis functions.* There are two major approaches to the construction of these basis functions: *nodal* and *hierarchic.*

2.6 Nodal Shape Functions

The nodal basis is a generalization of the classical linear basis (see Section 2.3), where the expansion coefficients of the basis or shape functions are obtained by taking the value of the function to be approximated at the vertices of an element. A more general definition is via linear forms:

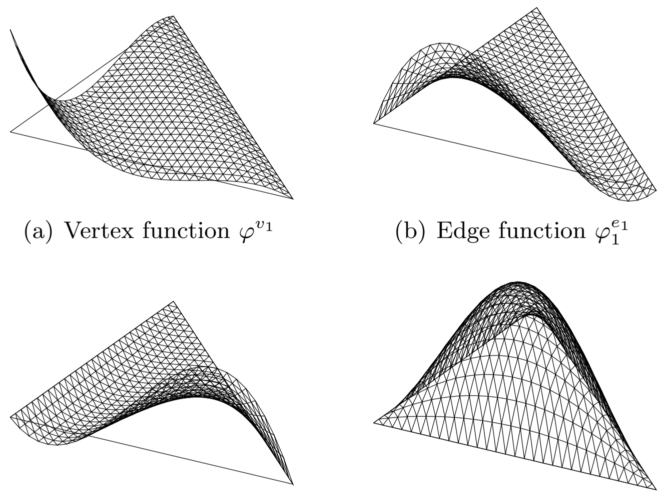
Definition 2.4 (Nodal finite element) Nodal finite element *is a triad* $\mathcal{K} = (K, P, \Sigma)$*, where*

- *K is a bounded domain in $\boldsymbol{R}^d$ with a Lipschitz-continuous boundary,*
- *P is a space of polynomials on K of the dimension dim$(P) = N_P$,*
- *$\Sigma = \{L_1, L_2, \ldots, L_{N_P}\}$ is a set of linear forms*

$$L_i : P \to \boldsymbol{R}, \quad i = 1, 2, \ldots, N_P.$$

The elements of Σ are called degrees of freedom *(DOF).*

Any polynomial $g \in P^p(K)$ must be uniquely identified by a set of N_P given numbers. This property is called unisolvency:


(a) Vertex function φ^{v_1} (b) Edge function $\varphi_1^{e_1}$

(c) Edge function $\varphi_2^{e_1}$ (d) Bubble function φ_1^{b}

Figure 2.4: Nodal basis of the P^3 element (only 4 of total 10 shape functions are depicted).

Definition 2.5 (Unisolvency) *A nodal finite element (K, P, Σ) is said to be* unisolvent *if for every polynomial $g \in P$ it holds*

$$L_1(g) = L_2(g) = \ldots = L_{N_P}(g) = 0 \quad \Rightarrow \quad g = 0.$$

Lemma 2.1 *Let (K, P, Σ) be a unisolvent nodal element. Given any set of numbers $\{g_1, g_2, \ldots, g_{N_P}\} \in \boldsymbol{R}^{N_P}$, where $N_P = dim(P)$, there exists a unique polynomial $g \in P$ such that*

$$L_1(g) = g_1, \; L_2(g) = g_2, \ldots, L_{N_P}(g) = g_{N_P}.$$

Definition 2.6 (Nodal basis) *Let (K, P, Σ), $dim(P) = N_P$, be a nodal finite element. We say that a set of functions $\mathcal{B} = \{\theta_1, \theta_2, \ldots, \theta_{N_P}\} \subset P$ is a* nodal basis *of P if it satisfies*

$$L_i(\theta_j) = \delta_{ij} \quad \textit{for all } 1 \le i, j \le N_P.$$

The functions θ_i are called nodal shape functions.

Theorem 2.2 (Characterization of unisolvency) *Consider a nodal finite element (K, P, Σ), $dim(P) = N_P$. The finite element is unisolvent if and only if there exists a unique nodal basis $\mathcal{B} = \{\theta_1, \theta_2, \ldots, \theta_{N_P}\} \subset P$.*

The proof of Theorem 2.2 (see [43]) provides an algorithm for the construction of the nodal shape functions. Figure 2.4 is an example of the nodal basis for a cubic element. Note that the functions φ^{v_2}, φ^{v_3}, $\varphi_1^{e_2}$, $\varphi_1^{e_3}$, $\varphi_2^{e_2}$, $\varphi_2^{e_3}$ are not shown and can be obtained from φ^{v_1}, $\varphi_1^{e_1}$ and $\varphi_2^{e_1}$ by rotation.

Nodal shape functions are advantageous in the sense that the corresponding unknown coefficients obtained from the solution of the discrete problem directly represent the value of the approximate solution $u_{h,p}$ at the nodal points. On the other hand they are not hierarchic and thus one has to replace the whole set of shape functions when increasing the polynomial order of elements. This means that it is difficult to combine nodal elements with various polynomial orders in the mesh and therefore they are not suitable for *p*- and *hp*-adaptivity. Furthermore, with simple choices of nodal points these shape functions yield ill-conditioned stiffness matrices.

2.7 Hierarchic Shape Functions

Hierarchic shape functions are constructed in such a way that the basis $\mathcal{B}^{p+1}$ of the polynomial space $P^{p+1}(K)$ is obtained from the basis $\mathcal{B}^p$ of the polynomial space $P^p(K)$ by adding new shape functions only. This is essential for *p*- and *hp*-adaptive finite element codes since one does not have to change his shape functions completely when increasing the order of polynomial approximation. In this section we will describe the popular *Lobatto-based* hierarchic shape functions.

Similarly to Section 2.6 we will define three types of shape functions:

- *vertex functions* $\varphi^{v_1}, \ldots, \varphi^{v_3}$ associated with the vertices $v_1, \ldots, v_3$. Each vertex function φ^{v_i} will be equal to one at v_i and will vanish at the remaining two vertices.
- *edge functions* $\varphi_j^{e_i}$, $i = 1, \ldots, 3, j = 2, \ldots, p^{e_i}$ associated with the edges $e_1, \ldots, e_3$. Each edge function $\varphi_j^{e_i}$ will concide with the Lobatto shape functions $l_2, l_3, \ldots$ (see below) on the edge e_i and will vanish on all remaining edges.
- *bubble functions* φ_j^b that vanish entirely on the element boundary.

Definition 2.7 (Affine coordinates) Affine coordinates *on the triangular reference domain K_t (Figure 2.3) are defined as*

$$\lambda_1(\xi_1, \xi_2) = \frac{\xi_2 + 1}{2}, \ \lambda_2(\xi_1, \xi_2) = -\frac{\xi_1 + \xi_2}{2}, \ \lambda_3(\xi_1, \xi_2) = \frac{\xi_1 + 1}{2}$$

Vertex functions $\varphi^{v_1}, \ldots, \varphi^{v_3}$ are chosen simply as these affine coordinates (we have already seen these shape functions in Figure 2.2).

$$\varphi^{v_1}(\xi_1, \xi_2) \quad = \quad \lambda_2(\xi_1, \xi_2) \tag{2.13}$$

$$\begin{aligned}\varphi^{v_2}(\xi_1,\xi_2) &= \lambda_3(\xi_1,\xi_2)\\ \varphi^{v_3}(\xi_1,\xi_2) &= \lambda_1(\xi_1,\xi_2)\end{aligned}$$

Definition 2.8 (Legendre polynomials) *We define the* Legendre polynomials *of degree k as*

$$\mathrm{L}_k(x) = \frac{1}{2^k k!}\frac{\mathrm{d}^k}{\mathrm{d}x^k}(x^2-1)^k, \quad k = 0,1,2,\ldots$$

The set of Legendre polynomials forms an orthonormal basis of the space $L^2(-1,1)$.

Definition 2.9 (Lobatto functions and kernel functions) *Let us define the functions*

$$\begin{aligned} l_0(x) &= \frac{1-x}{2} \\ l_1(x) &= \frac{x+1}{2} \\ l_k(x) &= \frac{1}{||\mathrm{L}_{k-1}||_2}\int_{-1}^{x}\mathrm{L}_{k-1}(\xi)\,\mathrm{d}\xi, \quad k\geq 2 \end{aligned} \tag{2.14}$$

Since all functions $l_k(x)$, $k\geq 2$ vanish at ±1, we can define the kernel functions ϕ_j, *$j=0,1,2,\ldots$, by decomposing l_k into*

$$l_k(x) = l_0(x)l_1(x)\phi_{k-2}(x), \quad k\geq 2$$

The edge functions $\varphi_j^{e_i}$ can now be written in the form

$$\begin{aligned} \varphi_j^{e_1} &= \lambda_2\lambda_3\phi_{j-2}(\lambda_3-\lambda_2), \quad 2\leq j\leq p^{e_1} \\ \varphi_j^{e_2} &= \lambda_3\lambda_1\phi_{j-2}(\lambda_1-\lambda_3), \quad 2\leq j\leq p^{e_2} \\ \varphi_j^{e_3} &= \lambda_1\lambda_2\phi_{j-2}(\lambda_2-\lambda_1), \quad 2\leq j\leq p^{e_3} \end{aligned} \tag{2.15}$$

Figure 2.5 is an example of a quadratic, cubic and a fourth-order edge function on the edge e_1.

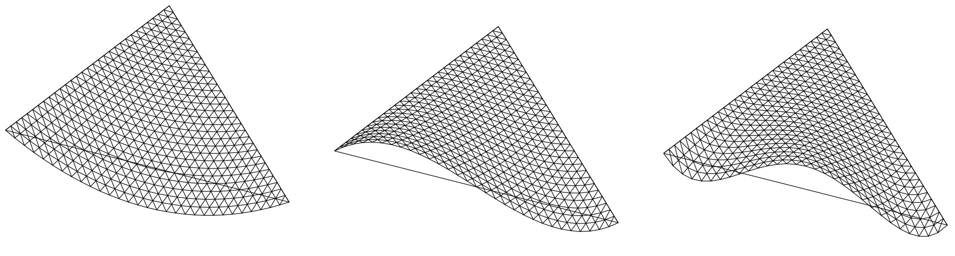


Figure 2.5: Edge shape functions $\varphi_2^{e_1}$, $\varphi_3^{e_1}$, $\varphi_4^{e_1}$.

The numbers $p^{e_1},\ldots,p^{e_3}$ in (2.15) are edge polynomial degrees, as determined by the *minimum rule*:

Definition 2.10 (Minimum rule) *Let K_i, K_j, $1 \le i,j \le M$ be two elements sharing a common edge e_k and let p_i, p_j be their associated polynomial degrees. The polynomial degree p^{e_k} of the edge e_k is $p^{e_k} = \min\{\, p_i,\, p_j\,\}$.*

The minimum rule guarantees that the resulting piece-wise polynomial space $V_{h,p}$ will be independent of a concrete choice of the shape functions.

The hierarchic basis will be completed by defining the bubble functions. A standard approach is to combine affine coordinates with varying powers,

$$\varphi^b_{n_1,n_2} = \lambda_1(\lambda_2)^{n_1}(\lambda_3)^{n_2}, \quad 1 \le n_1, n_2;\; n_1 + n_2 \le p_m - 1 \tag{2.16}$$

However, conditioning properties of these functions are relatively bad. This motivates us to define a different set of bubble functions by incorporating the kernel functions:

$$\varphi^b_{n_1,n_2} = \lambda_1\lambda_2\lambda_3\phi_{n_1-1}(\lambda_3 - \lambda_2)\phi_{n_2-1}(\lambda_2 - \lambda_1), \tag{2.17}$$

where n_1 and n_2 satisfy the same conditions as in (2.16). Figures 2.6 and 2.7 are examples of such shape functions.

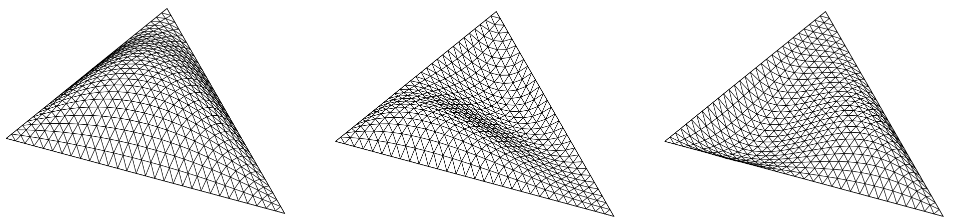


Figure 2.6: Cubic bubble function $\varphi^b_{1,1}$ and fourth-order functions $\varphi^b_{1,2}$, $\varphi^b_{2,1}$.

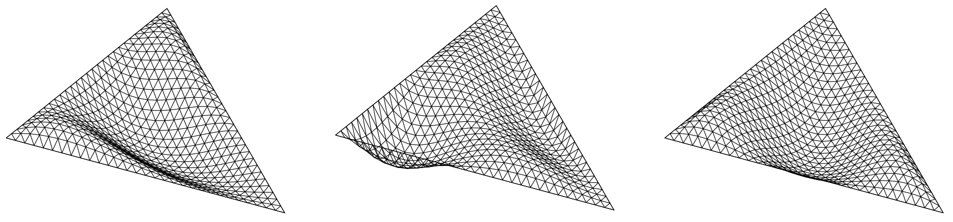


Figure 2.7: Fifth-order bubble functions $\varphi^b_{1,3}$, $\varphi^b_{3,1}$ and $\varphi^b_{2,2}$.

Proposition 2.1 *The shape functions (2.13), (2.15) and (2.17) constitute a hierarchic basis of the space $P^{p_m}(K_t)$*

Proof The complete proof can be found in [45]. Briefly, the following must be verified:

1. Are all the shape functions linearly independent?
2. Do they all belong to the space $P^{p_m}(K_t)$?
3. Matches their number the dimension of the space $P^{p_m}(K_t)$?

As has been mentioned, the choice of the shape functions has a strong influence on the conditioning of the stiffness matrix associated with the discrete problem. The described bubble functions can still be improved, e.g., by their orthogonalization. Even though the orthogonality is not preserved after the shape functions are transformed from the reference domain to the actual element, experiments show that the condition number of the stiffness matrix is significantly lower nevertheless (see [55]). In our software we have been using these orthonormal bubble functions.

Let us conclude this section with Table 2.1, showing the numbers of shape functions depending on the polynomial degree p^m of the element.

Shape type	**Existence**	**Number of shape functions**
Vertex	always	3
Edge	$p^{e_i} \geq 2$	$\sum_{i=1}^{3}(p^{e_i} - 1)$
Bubble	$p^m \geq 3$	$(p^m - 1)(p^m - 2)/2$

Table 2.1: Numbers of shape functions.

2.8 Nodes and Connectivity Arrays

In standard linear FEM, mesh vertices are simply numbered, ie. assigned vertex basis function indices. Due to the existence of edge and bubble basis functions in higher-order FEM, a more elaborate bookkeeping is required. Data structures that link the shape functions $\varphi_1, \varphi_2, \varphi_3, \ldots$ on an element to the basis functions $v_i, v_j, v_k, \ldots \subset V_{h,p}$ are called *connectivity arrays.* These arrays ensure that:

- vertex shape functions are joined properly to form vertex basis functions, ie. that the appropriate vertex shape functions on elements sharing a common vertex are assigned the same vertex basis function index;
- edge shape functions are joined properly to form edge basis functions, ie. that the appropriate edge shape functions on elements sharing a common edge are assigned the same edge basis function index.

This is usually achieved by data structures called *nodes.* Three types of nodes are defined: *vertex nodes*, associated with vertices, *edge nodes*, associated with edges and *bubble nodes*, associated with element interiors. Each node holds one or more basis function indices. Each element is linked with seven nodes corresponding to its vertices, edges and interior during initialization.

2.9 Higher-Order Numerical Quadrature

Most commonly, the integrals in (2.11), (2.12) are evaluated numerically by the *Gaussian quadrature*. The k-point Gaussian quadrature rule on the domain K_t has the form

$$\int_{K_t} g(\xi)\mathrm{d}\xi \approx \sum_{i=1}^{k} w_{k,i} g(\xi_{k,i}) \tag{2.18}$$

where g is a bounded continuous function, $\xi_{k,i} \in K_t, i = 1, 2, \ldots, k$ are the integration points and $w_{k,i} \in \boldsymbol{R}$ are the integration weights. The sum of the integration weights must be equal to the area of K_t, so that the rule (2.18) is exact for constants. If the points and weights are chosen carefully, the formula (2.18) can be exact for polynomials up to a certain degree $q > 0$.

In 1D the integration points are roots of the Legendre polynomials. Also in 2D it is quite easy to find the integration points and weights for low-order polynomials ocurring in traditional linear FEM. For higher-order polynomials, however, the task of finding optimal Gaussian quadrature rules presents a complex non-linear optimization problem with many open questions left. Optimal integration points and weights are known on K_t for polynomials up to degree 10. Suboptimal (with more points than necessary) rules have been found for polynomials up to degree 20. Complete integration tables along with more information on this subject can be found in [45].

2.10 Assembling Procedure

Recall from Section 2.2 that after converting the PDE (2.1) to the weak form (2.6), we can use the Galerkin method to obtain the discrete problem

$$\boldsymbol{SY} = \boldsymbol{F},$$

where $\boldsymbol{Y}$ is the vector of uknowns, and the stiffness matrix $\boldsymbol{S}$ and the load vector $\boldsymbol{F}$ are defined as

$$\boldsymbol{S} = \{\ a(v_j, v_i)\ \}_{i,j=1}^{N},$$

$$\boldsymbol{F} = \{\ l(v_i)\ \}_{i=1}^{N}.$$

Here $v_1, v_2, \ldots, v_N$ are the basis function of the finite-dimensional space $V(\Omega)$, $\dim(V(\Omega)) = N$ and $a(u, v)$, $l(v)$ are the bilinear and linear forms, respectively. The construction of $\boldsymbol{S}$ and $\boldsymbol{F}$ is called *assembling*.

A naive implementation of the assembling procedure might read as follows:

```
$\boldsymbol{S} = \mathbf{0}_{N\times N}$
**for** $i = 1 \dots N$ **do**
    **for** $j = 1 \dots N$ **do**
        **for** $k = 1 \dots M$ **do**
            $\boldsymbol{S}_{ij} = \boldsymbol{S}_{ij} + a(v_j, v_i)|K_k$
```

The innermost statement is summing contributions of $a(v_j, v_i)$ restricted to each element K_k, M is the total number of elements. The first flaw of this algorithm is that it ignores the *sparsity* of $\boldsymbol{S}$, since $\boldsymbol{S}$ only contains $O(N)$ nonzero elements. This is due to the fact that the supports of v_i, $1 \le i \le N$,

$$\mathrm{supp}(v_i) = \overline{\{\boldsymbol{x} \in \Omega;\ v_i(\boldsymbol{x}) \neq 0\}} \subset \Omega$$

are small and mostly disjoint. The element $\boldsymbol{S}_{ij} = a(v_j, v_i)$ is nonzero if and only if

$$\mathrm{meas}_{2D}(\mathrm{supp}(v_i) \cap \mathrm{supp}(v_j)) > 0.$$

This is easy to see, since the bilinear form $a(u, v)$ is zero whenever u or v is zero. The zero elements of $\boldsymbol{S}$ should not be stored at all to avoid $O(N^2)$ memory complexity and only the nonzero elements should be evaluated:

```
**for** $i = 1 \dots N$ **do**
    **for** $j = 1 \dots N$ **do**
        **for** $k = 1 \dots M$ **do**
            **if** $\mathrm{meas}_{2D}(\mathrm{supp}(v_i) \cap K_k) > 0$ **and** $\mathrm{meas}_{2D}(\mathrm{supp}(v_j) \cap K_k) > 0$
                $\boldsymbol{S}_{ij} = \boldsymbol{S}_{ij} + a(v_j, v_i)|K_k$
```

However, this algorithm still has about $O(MN^2)$ time complexity. This is easily remedied by rearranging the cycles and utilizing the element connectivity arrays (Section 2.8), which gives us the final Algorithm 2.1, called the *element-by-element assembling procedure*. The size of the list L is $O(p^2)$, where p is the polynomial degree of the elements (see Table 2.1), therefore the total time required to assemble $\boldsymbol{S}$ is $O(Mp^4)$.

Algorithm 2.1: Element-by-element assembling procedure.

```
**for** $k = 1 \dots M$ **do**
    $L$ = list of basis function indices whose $\mathrm{supp}(v_i) \cap K_k \neq \emptyset$ ;
    **foreach** $i \in L$ **do**
        **foreach** $j \in L$ **do**
            $\boldsymbol{S}_{ij} = \boldsymbol{S}_{ij} + a(v_j, v_i)|K_k$
```

Chapter 3

Constrained Approximation

In Chapter 2 we have assumed that an optimal finite element mesh is given a priori and moreover that it is regular. In practice, however, an optimal mesh has to be obtained by an adaptive process, in which elements whose approximation error is too high are replaced with smaller elements. This concept is know as h-adaptivity and is central to most adaptive schemes (more details on h-, p-, and other schemes will be given in Chapter 4).

This chapter motivates the use of irregular meshes for hp-FEM, gives a survey of existing constrained approximation techniques and presents a detailed description of our new algorithm and data structures allowing the use of arbitrary-level hanging nodes in a finite element computation.

3.1 Hanging Nodes and Mesh Regularity

In the design of most h-adaptive schemes one is faced with the problem of maintaining mesh regularity in the course of the adaptive process. A class of algorithms known as bisection methods [37, 27] is based on dividing triangles across the longest edge followed by recursive restoration of the regularity by additional bisections. The recursion can be shown to always finish provided the initial mesh satisfies certain properties. Due to their inherent limitation to simplicial meshes and their requirements on the initial mesh, these methods will not be discussed here.

Another widely used approach is the red-green refinement strategy [9, 10], illustrated in Figure 3.1. After being selected for refinement, some of the elements have been subdivided into four smaller elements ("red" refinement), which gives rise to vertices coinciding with an edge, often called hanging nodes. Most finite element codes, especially those based on standard piecewise-linear elements, tend to eliminate hanging nodes by forcing additional ("green") element refinements. In this way the modified mesh is regular again and ready for the finite element solver.

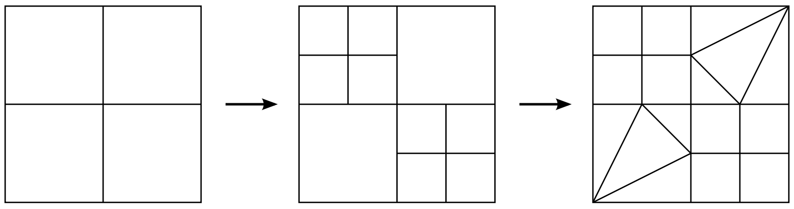


Figure 3.1: Red-green refinement strategy.

Figure 3.2 lists all hanging-node cases that need to be taken into account when implementing this refinement strategy. These patterns are sometimes called transition elements, since they provide transition layers between different levels of refinement of the mesh. It should be noted that all transition elements must only exist temporarily and that they need to be removed from the mesh in the subsequent adaptive iteration, otherwise a deterioration of mesh quality will occur. Only after the true mesh elements have been refined again the transition layers are reintroduced to allow recalculation of the finite element solution.

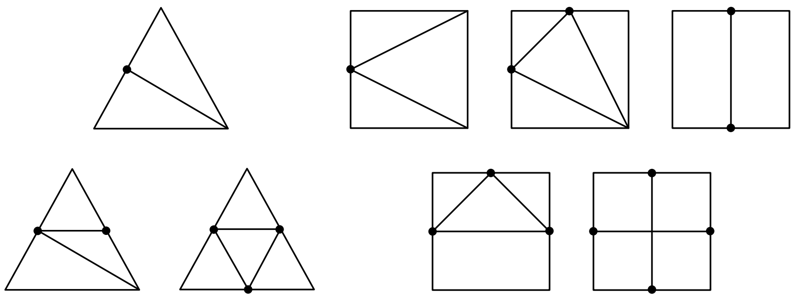


Figure 3.2: Transition elements.

The use of the transition elements is not limited to one hanging node per edge only, as they can be applied repeatedly (recursively) to themselves. However, this quickly leads to elements with sharp angles, unsuitable for finite element analysis. The solution is to make sure first that the mesh is 1-irregular, i.e., that it only contains at most one hanging node per edge. This is achieved by additional "red" refinements, as shown in Figure 3.3. Once the mesh is 1-irregular, the transition elements can safely be applied.

While the red-green strategy is simple enough to implement and gives satisfactory results for linear finite elements, the following problems arise in higher-order FEM as soon as the polynomial order of the approximation is at least two:

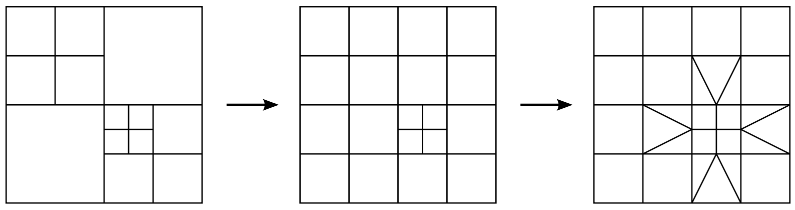


Figure 3.3: 1-irregularity enforcement.

- It is unclear what polynomial orders should be assigned to transition elements replacing standard higher-order elements. Together with the fact that quadrilaterals often have to be replaced by transition triangles, it is difficult to maintain approximation properties of the original elements.

- Where linear transition elements only add degrees of freedom to the hanging nodes, higher-order transition elements also contain internal degrees of freedom which may be redundant. Experience shows (see [22], p. 83) that enforcing 1-irregularity and then complete regularity by transition elements makes the discrete problem substantially larger in *hp*-FEM.

For these reasons, most *hp*-adaptive codes (but also some standard, *h*-adaptive codes) drop the regularity requirement and allow meshes with hanging nodes, even though typically only 1-irregular meshes are supported for simplicity [16, 18]. However, global continuity of the solution must be preserved, which means that algebraic constraints have to be imposed on the irregular approximation. The rest of this chapter deals with the various techniques of enforcing such constraints, which can roughly be divided into two groups:

- Explicit constraints, in which the constraining relations are explicitly constructed and enforced during the solution of the linear system. These methods are suitable for nodal higher-order elements.

- Implicit constraints, on the other hand, are enforced in the process of the construction of the higher-order finite element space, so that already the basis functions satisfy the constraints. This way the solution satisfies the constraints automatically and implicitly. This class of methods is more suited for hierarchic higher-order elements.

3.2 Explicit Constraints

Since the emphasis of this work is on hierarchic elements, let us discuss the explicit constraints only briefly. Consider the simplest situation with one

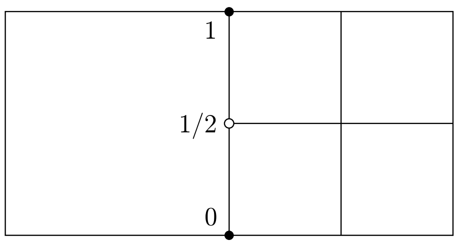


Figure 3.4: Linear nodal elements with one constrained node.

hanging node in a mesh of linear nodal elements, as depicted in Figure 3.4. The nodes we are interested in are marked by their position along the constraining edge $\overline{01}$. It is clear that in order to keep the solution continuous, the value $y_{1/2}$ corresponding to node 1/2 in the solution vector $\boldsymbol{y}$ must in this case satisfy the constraint

$$y_{1/2} = (y_0 + y_1)/2.$$

In general we introduce a constraining matrix $\boldsymbol{C}$ and require that $\boldsymbol{y} = \boldsymbol{C}\boldsymbol{y}$. The constraining matrix acts as identity on regular, unconstrained nodes and ensures that correct values are assigned to constrained nodes. In our simple example $\boldsymbol{C}$ would have the form

$$\boldsymbol{C} = \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & \frac{1}{2} & 0 & \frac{1}{2} & & \\ & & & & \ddots & \\ & & & & & 1 \end{pmatrix}.$$

The higher-order case will be illustrated on the same mesh with quadratic nodal elements, as shown in Figure 3.5. Here, the nodes 0, 1/2 and 1 are unconstrained, as they form a quadratic solution along the edge $\overline{01}$ belonging to the large element. Nodes 1/4 and 3/4 are constrained and their values need to be expressed in terms of the values y_0, $y_{1/2}$ and y_1.

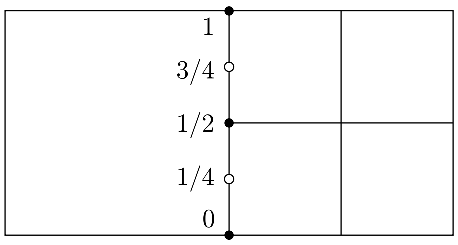


Figure 3.5: Quadratic nodal elements with two constrained nodes.

This is easily accomplished by noticing that the coefficients (a, b, c) of the unknown quadratic function along $\overline{01}$ can be obtained from the linear system

$$(y_0,\, y_{1/2},\, y_1)^T = \boldsymbol{V} \cdot (a, b, c)^T,$$

where

$$\boldsymbol{V} = \{(x_i)^{2-j}\}_{ij}, \quad 0 \leq i, j \leq 2,$$

is the Vandermonde matrix for $(x_0,\, x_1,\, x_2) = (0,\, 1/2,\, 1)$. Knowing the coefficients of the quadratic polynomial we can express its value at any point x along the edge:

$$y_x = (x^2,\, x,\, 1) \cdot \boldsymbol{V}^{-1} \cdot (y_0,\, y_{1/2},\, y_1)^T.$$

This way the following constraints are obtained:

$$\begin{aligned} y_{1/4} &= \frac{3}{8} y_0 + \frac{3}{4} y_{1/2} - \frac{1}{8} y_1, \\ y_{3/4} &= -\frac{1}{8} y_0 + \frac{3}{4} y_{1/2} + \frac{3}{8} y_1. \end{aligned}$$

The constraining matrix $\boldsymbol{C}$ then has the form

$$\boldsymbol{C} = \begin{pmatrix} 1 & & & & \\ 3/8 & 0 & 3/4 & 0 & -1/8 \\ & & 1 & & \\ -1/8 & 0 & 3/4 & 0 & 3/8 \\ & & & & 1 \end{pmatrix}.$$

An analogous procedure can be carried out for nodal elements of any polynomial degree and conceivably even for hanging nodes of higher levels, if indirect constraints, i.e., constrained nodes depending on other constrained nodes, are eliminated from $\boldsymbol{C}$ by raising it to a sufficiently large power.

Given the constraining matrix $\boldsymbol{C}$, the question now is how to solve the discrete problem $\boldsymbol{Ay} = \boldsymbol{f}$ so that at the same time $\boldsymbol{y} = \boldsymbol{Cy}$ is satisfied.

3.2.1 Solver-enforced Constraints

The simplest method [50] requires that the discrete problem can be solved by minimizing the energy functional

$$\Phi(\boldsymbol{y}) = \frac{1}{2} \boldsymbol{y}^T \boldsymbol{A} \boldsymbol{y} - \boldsymbol{y}^T \boldsymbol{f},$$

which is the case for symmetric elliptic problems. Moreover, the method assumes that an iterative solver such as CG is used for the solution of $\boldsymbol{Ay} = \boldsymbol{f}$, that the solver only accesses the matrix $\boldsymbol{A}$ via a matrix-vector product $\boldsymbol{Ax}$ and finally that you have a chance of modifying the iterative solver.

The constrained minimization problem

$$\Phi(\boldsymbol{y}) = \frac{1}{2}\boldsymbol{y}^T\boldsymbol{A}\boldsymbol{y} - \boldsymbol{y}^T\boldsymbol{f} = \min,$$
$$\boldsymbol{y} = \boldsymbol{C}\boldsymbol{y}$$

is simply rewritten as

$$\Phi(\boldsymbol{y}) = \frac{1}{2}\boldsymbol{y}^T(\boldsymbol{C}^T\boldsymbol{A}\boldsymbol{C})\boldsymbol{y} - \boldsymbol{y}^T(\boldsymbol{C}^T\boldsymbol{f}) = \min.$$

However, the product $\boldsymbol{C}^T\boldsymbol{A}\boldsymbol{C}$ is never formed explicitly. Instead, the iterative solver is modified to calculate $\boldsymbol{C}^T(\boldsymbol{A}(\boldsymbol{C}\boldsymbol{x}))$ (i.e., a series of matrix-vector products) in place of the standard product $\boldsymbol{A}\boldsymbol{x}$, where $\boldsymbol{x}$ is the solution iterate. The operation $\boldsymbol{C}\boldsymbol{x}$ forces all iterates and eventually also the solution to conform to the constraints. A more rigorous justification is given in Section 3.2.3.

This method was used in the software DEAL [15]. Its disadvantages coincide with its assumptions, the cheif disadvantage being the need to use a modified iterative solver.

3.2.2 Lagrange Multipliers

A more standard technique that does not require a special solver is the method of Lagrange multipliers. We will use a particular version of this general method to solve the system $\boldsymbol{A}\boldsymbol{y} = \boldsymbol{f}$ subject to the set of constraints $\boldsymbol{D}\boldsymbol{y} = 0$. In our case the matrix $\boldsymbol{D}$ is obtained by deleting zero rows from $(\boldsymbol{C} - \boldsymbol{I})$. Let us again assume that $\boldsymbol{A}$ is symmetric and positive definite (SPD).

The method proceeds by minimizing the Lagrangian, a specially constructed functional of the form

$$\Lambda(\boldsymbol{y}, \boldsymbol{\lambda}) = \frac{1}{2}\boldsymbol{y}^T\boldsymbol{A}\boldsymbol{y} - \boldsymbol{y}^T\boldsymbol{f} + \boldsymbol{\lambda}^T\boldsymbol{D}\boldsymbol{y}.$$

Here, $\boldsymbol{\lambda}$ is a vector of M Lagrange multipliers, $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, ..., \lambda_M)^T$, where M is the number of rows of $\boldsymbol{D}$, i.e., the number of constraints. In order to find the minimum of the Lagrangian, we require that

$$\nabla\Lambda(\boldsymbol{y}, \boldsymbol{\lambda}) = 0. \tag{3.1}$$

In particular, by writing all derivatives by y_1, y_2, ..., y_N and all derivatives by λ_1, λ_2, ..., λ_M separately, the gradient in (3.1) reads

$$\nabla_y \Lambda(\boldsymbol{y}, \boldsymbol{\lambda}) = \boldsymbol{A}\boldsymbol{y} - \boldsymbol{f} + \boldsymbol{D}^T\boldsymbol{\lambda},$$

$$\nabla_\lambda \Lambda(\boldsymbol{y}, \boldsymbol{\lambda}) = \boldsymbol{D}\boldsymbol{y}.$$

We obtain an augmented system which solves the original problem:

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{D}^T \\ \boldsymbol{D} & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{y} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \boldsymbol{f} \\ 0 \end{bmatrix}.$$

The main disadvantage of this method is the fact that the augmented matrix is larger and no longer SPD due to the zero block, which means that many common iterative solvers cannot be applied. The method may be useful for enforcing more complex constraints than those induced by hanging nodes, for example in the context of mortar or mesh tying methods.

3.2.3 Matrix Condensation Method

Section 3.2.1 hints at the possibility of directly solving $\tilde{\boldsymbol{A}}\boldsymbol{y} = \tilde{\boldsymbol{f}}$, where

$$\begin{aligned} \tilde{\boldsymbol{A}} &= \boldsymbol{C}^T\boldsymbol{A}\boldsymbol{C}, \\ \tilde{\boldsymbol{f}} &= \boldsymbol{C}^T\boldsymbol{f}. \end{aligned}$$

This is indeed possible[1], provided that one is willing to calculate the product $\boldsymbol{C}^T\boldsymbol{A}\boldsymbol{C}$. This may not be trivial for sparse matrices, because the calculation should be done in $O(N)$ operations. Let us assume this is feasible. We will show what is the meaning of $\tilde{\boldsymbol{A}}$ and $\tilde{\boldsymbol{f}}$, to justify both this method and that of Section 3.2.1.

Recall from Chapter 2 that the elements of $\boldsymbol{A}$ and $\boldsymbol{f}$ are

$$\begin{aligned} A_{ij} &= a(\varphi_j, \varphi_i), \\ f_i &= l(\varphi_i), \end{aligned}$$

where $a(u,v)$ is the bilinear form, $l(v)$ is the linear form and φ_1, φ_2, …, φ_N are the basis functions. An element of $\tilde{\boldsymbol{f}}$ is then

$$\tilde{f}_i = \sum_m C_{mi}\, f_m = \sum_m C_{mi}\, l(\varphi_m) = l\left(\sum_m C_{mi}\, \varphi_m\right) = l(\tilde{\varphi}_i).$$

Similarly for one element of $\tilde{\boldsymbol{A}}$:

$$\begin{aligned} \tilde{A}_{ij} &= \sum_m \sum_n C_{mi} A_{mn} C_{nj} = \sum_m \sum_n C_{mi}\, a(\varphi_n, \varphi_m)\, C_{nj} = \\ &= a\left(\sum_n C_{nj}\varphi_n,\ \sum_m C_{mi}\varphi_m\right) = a(\tilde{\varphi}_j, \tilde{\varphi}_i). \end{aligned}$$

We see that $\tilde{\boldsymbol{A}}$ and $\tilde{\boldsymbol{f}}$ are the stiffness matrix and load vector corresponding to a new basis $\tilde{\varphi}_1$, $\tilde{\varphi}_2$, …, $\tilde{\varphi}_N$, where

$$\tilde{\varphi}_i = \sum_m C_{mi}\, \varphi_m.$$

[1]Zero rows and columns of $\tilde{\boldsymbol{A}}$ need to be disregarded, or all zero $\tilde{A}_{ii}$ replaced by ones.

In other words, each $\tilde{\varphi}_i$ is a linear combination corresponding to the i-th column of $\boldsymbol{C}$. All of the new $\tilde{\varphi}_i$ are continuous functions, since the constraining matrix $\boldsymbol{C}$ was constructed in such a way that $\boldsymbol{Cy}$ results in a continuous approximation for any $\boldsymbol{y}$. This means we can take $\boldsymbol{y} = \boldsymbol{e}_i = (0, \ldots, 0, 1, 0, \ldots, 0)^T$ and obtain a continuous function which is exactly the new basis function $\tilde{\varphi}_i$. Any linear combination of the new basis functions will be a continuous function and we conclude that $\tilde{\boldsymbol{A}}\tilde{\boldsymbol{y}} = \tilde{\boldsymbol{f}}$ and $\boldsymbol{y} = \boldsymbol{C}\tilde{\boldsymbol{y}}$ solves the original constrained problem.

Referring back to the situation in Figure 3.5 on page 22, let us illustrate how a continuous basis function $\tilde{\varphi}_0$ corresponding to node 0 is formed. The first column of the matrix $\boldsymbol{C}$ says that on the right-hand side of the edge $\overline{01}$ three standard nodal functions φ_0, $\varphi_{1/4}$ and $\varphi_{3/4}$ should be combined to match the unconstrained vertex function on the left side of $\overline{01}$, as shown in Figure 3.6.

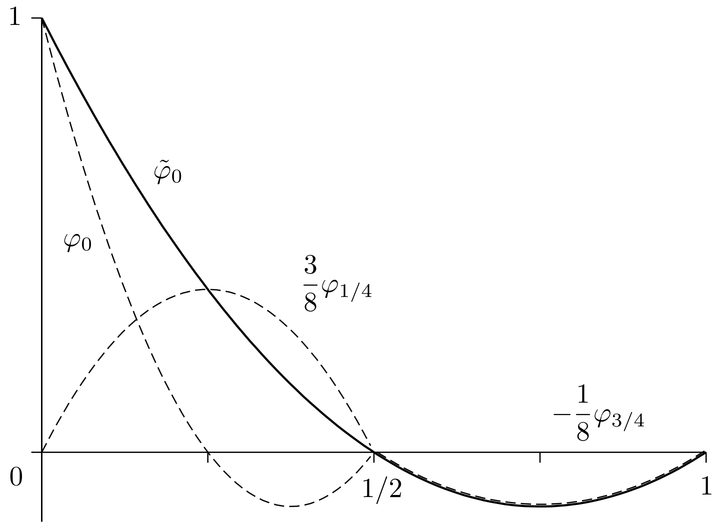


Figure 3.6: Combining φ_0, $\varphi_{1/4}$ and $\varphi_{3/4}$ to form $\tilde{\varphi}_0$.

This method is used in the deal.II finite element library [7] and has several advantages over the older method based on iterative solvers: an arbitrary, unmodified solver can be used, the method preserves positive definiteness of the stiffness matrix in elliptic problems, but works also for problems that are not elliptic. The only disadvantage of the method is the need to manipulate the stiffness matrix to form the product $\boldsymbol{C}^T\boldsymbol{AC}$. Care must especially be taken to avoid quadratic complexity of the condensation algorithm. In [6] the authors claim that a code running in $O(N)$ is complicated, but possible. We believe that this problem could be avoided completely by applying portions of $\boldsymbol{C}$ and $\boldsymbol{C}^T$ to the local element matrices, before they are inserted into the global matrix, thus eliminating the need for complex sparse matrix operations.

3.3 Implicit Constraints

Let us now turn our attention to hierachic higher-order elements, which are more suitable for *hp* discretizations. Unlike in the case of nodal elements, there is no direct relation between the values of the solution vector $\boldsymbol{y}$ and the values of the approximate solution and thus the constraining matrix $\boldsymbol{C}$ cannot be obtained easily. However, inspired by the previous section, we may be able to directly construct continuous hierarchic basis functions so that the solution satisfies the continuity constraints implicitly.

3.3.1 One-Level Hanging Nodes

Most codes are limited to 1-irregular meshes in their implementation of hanging nodes [16, 18], since especially in 3 dimensions this simplifies the code substantially. In this section we will also resort to this simplification in order to start with the simplest possible example of hanging nodes for hierarchic elements.

Consider a triangular mesh of 5 quadratic elements with one hanging node, as shown in Figure 3.7. The elements are numbered ABC, AED, EGD, DGB and EFG. Assuming no Dirichlet boundary conditions, 21 basis functions can be constructed on this quadratic mesh:

- 6 vertex functions associated with the vertices A, B, C, E, F, G, numbered $(d^A, d^B, d^C, d^E, d^F, d^G) = (1, 2, 3, 4, 5, 6)$,
- 10 quadratic edge functions associated with the edges AB, BC, CA, AE, ED, EG, DG, GB, EF, FG, numbered $(d^{AB}, d^{BC}, \dots, d^{FG}) = (7, 8, \dots, 16)$,
- 5 quadratic bubble functions associated with the elements K_1, K_2, K_3, K_4, K_5 numbered $(d^{K_1}, d^{K_2}, \dots, d^{K_5}) = (17, 18, \dots, 21)$.

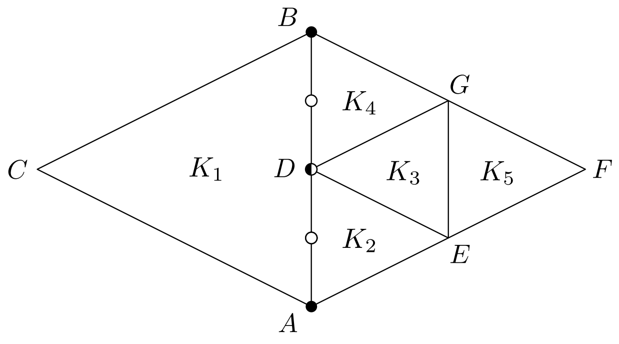


Figure 3.7: 1-irregular triangular mesh.

The vertex node D and the edge nodes corresponding to edges AD and DB are constrained and carry no degrees of freedom, i.e., no basis functions are associated with them. In fact, these nodes belong to basis functions coinciding with the edge AB containing the hanging node. These basis functions are special and in this case there are three of them:

1. Vertex function φ_A with value 1 in node A, linear along AB and zero in all other nodes except in node D where it has to attain the value 1/2.
2. Similarly, vertex function φ_B with value 1 in node B, 1/2 in node D and zero in all other vertex nodes.
3. Quadratic edge function φ_{AB} with value y_D in node D and zero in all other vertex nodes.

The simplest continuous functions that meet these requirements are shown in Figure 3.8 (φ_A was omitted, since it is a mirror image of φ_B). These basis functions, along with the other 18 standard basis functions, generate the proper (P^2) polynomial spaces on all the elements and at the same time ensure a continuous FE solution on top of the irregular mesh in Figure 3.7.

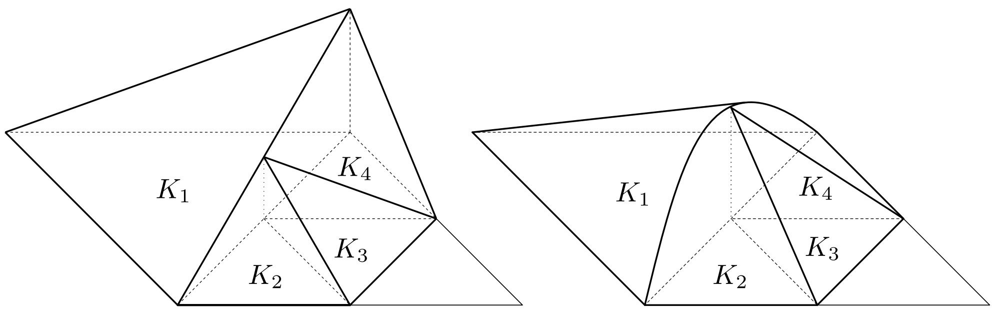


Figure 3.8: Basis functions φ_B (left) and φ_{AB} (right).

The vertex basis functions φ_A and φ_B are constructed from standard vertex shape functions φ^{v_1}, φ^{v_2}, φ^{v_3} (see Figure 3.9), with the exception that shape functions corresponding to the hanging node D need to be multiplied by the value 1/2. Figure 3.10 shows the construction of the edge function φ_{AB}. Here, the vertex functions corresponding to the node D need to be multiplied by y_D, the mid-edge value of the constraining quadratic edge function $\varphi_2^{e_1}$. Moreover, two quadratic functions φ^{c_1} and φ^{c_2} must be added on K_2 and K_4 to make the basis function continuous. These functions, called constrained edge functions, are added to the standard set of hierarchic shape functions and are defined as linear combinations of regular edge functions,

$$\varphi_p^{c_j^k} = \sum_{i=2}^{p} \alpha_{p,i}^k \, \varphi_i^{e_j}, \qquad k = 1, 2.$$

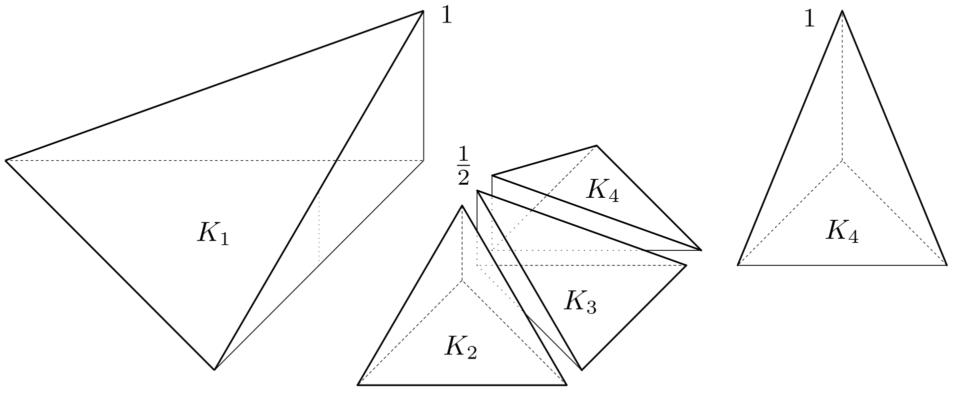


Figure 3.9: Shape functions constituting the vertex basis function φ_B.

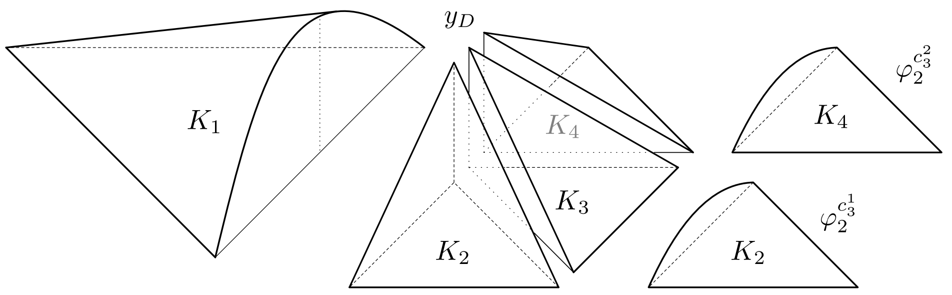


Figure 3.10: Shape functions constituting the edge basis function φ_{AB}.

The coefficients $\alpha^k_{p,i}$ are precalculated for both halves of the edge $(k = 1, 2)$ so that except for the linear part the constrained edge functions exactly fit portions of the regular edge functions. In the quadratic case, the constrained edge functions are just the regular edge functions multiplied by a constant.

Before we show how the assembling procedure needs to be modified, recall first from Chapter 2 how the usual element-by-element assembling procedure works in the unconstrained case, for example on the element K_1. The following two vectors are obtained from the connectivity arrays:

$$\begin{aligned}\boldsymbol{\varphi}^{K_1} &= (\ \varphi^{v_1},\ \varphi^{v_2},\ \varphi^{v_3},\ \varphi_2^{e_1},\ \varphi_2^{e_2},\ \varphi_2^{e_3},\ \varphi^{b}\ ),\\ \boldsymbol{d}^{K_1} &= (\ d^A,\ d^B,\ d^C,\ d^{AB},\ d^{BC},\ d^{CA},\ d^{K_1}\ ).\end{aligned}$$

The vector $\boldsymbol{\varphi}$ lists all shape functions for the element in question and the vector $\boldsymbol{d}$ connects the local degrees of freedom to the global ones. A local stiffness matrix $\boldsymbol{L}$ is calculated,

$$L^{K_m}_{ij} = a(\varphi^{K_m}_j, \varphi^{K_m}_i),$$

and the result is then distributed to the global stiffness matrix $\boldsymbol{A}$,

$$A_{d^{K_m}_i, d^{K_m}_j} = A_{d^{K_m}_i, d^{K_m}_j} + L^{K_m}_{ij}.$$

Only a few modifications are necessary to add support for hanging nodes. First, the vector $\boldsymbol{\varphi}^{K_m}$ can contain multiple instances of the vertex shape functions φ^{v_i}, since these can now be shared by multiple basis functions, as is the case for φ_A, φ_B and φ_{AB}. Second, each shape function can be multiplied by a constant. For this reason we introduce a third vector, $\boldsymbol{c}^{K_m} = (c_1^{K_m}, c_2^{K_m}, \ldots)$, storing a coefficient for each of the shapes $\varphi_i^{K_m}$. The coefficients only affect the handling of the local stiffness matrix $\boldsymbol{L}$, since

$$L_{ij}^{K_m} = a(c_j^{K_m}\varphi_j^{K_m}, c_i^{K_m}\varphi_i^{K_m}) = c_j^{K_m}c_i^{K_m}\, a(\varphi_j^{K_m}, \varphi_i^{K_m}).$$

For the lack of a better term, we call the vectors $\boldsymbol{\varphi}^{K_m}$, $\boldsymbol{d}^{K_m}$ and $\boldsymbol{c}^{K_m}$ the "assembly lists". To conclude the one-level quadratic example, we include the complete assembly lists for the elements K_2, K_3 and K_4:

$$\begin{aligned}
\boldsymbol{\varphi}^{K_2} &= (\ \varphi^{v_1},\ \varphi^{v_2},\ \varphi^{v_3},\ \varphi^{v_3},\ \varphi^{v_3},\ \varphi_2^{e_1},\ \varphi_2^{e_2},\ \varphi_2^{c_3^1},\ \varphi^{b}\ ),\\
\boldsymbol{d}^{K_2} &= (\ d^A,\ d^E,\ d^A,\ d^B,\ d^{AB},\ d^{AE},\ d^{ED},\ d^{AB},\ d^{K_2}\ ),\\
\boldsymbol{c}^{K_2} &= (\ 1,\ 1,\ \tfrac{1}{2},\ \tfrac{1}{2},\ y_D,\ 1,\ 1,\ 1,\ 1\ ),\\[1ex]
\boldsymbol{\varphi}^{K_3} &= (\ \varphi^{v_1},\ \varphi^{v_2},\ \varphi^{v_3},\ \varphi^{v_3},\ \varphi^{v_3},\ \varphi^{e_1},\ \varphi^{e_2},\ \varphi^{e_3},\ \varphi^{b}\ ),\\
\boldsymbol{d}^{K_3} &= (\ d^E,\ d^G,\ d^A,\ d^B,\ d^{AB},\ d^{EG},\ d^{GD},\ d^{DE},\ d^{K_3}\ ),\\
\boldsymbol{c}^{K_3} &= (\ 1,\ 1,\ \tfrac{1}{2},\ \tfrac{1}{2},\ y_D,\ 1,\ 1,\ 1,\ 1\ ),\\[1ex]
\boldsymbol{\varphi}^{K_4} &= (\ \varphi^{v_1},\ \varphi^{v_1},\ \varphi^{v_1},\ \varphi^{v_2},\ \varphi^{v_3},\ \varphi_2^{e_1},\ \varphi_2^{e_2},\ \varphi_2^{c_3^2},\ \varphi^{b}\ ),\\
\boldsymbol{d}^{K_4} &= (\ d^A,\ d^B,\ d^{AB},\ d^G,\ d^B,\ d^{DG},\ d^{GB},\ d^{AB},\ d^{K_4}\ ),\\
\boldsymbol{c}^{K_4} &= (\ \tfrac{1}{2},\ \tfrac{1}{2},\ y_D,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1\ ).
\end{aligned}$$

3.3.2 Arbitrary-Level Hanging Nodes

While one-level hanging nodes may be quite useful, they still introduce unneeded refinements since the mesh has to be kept 1-irregular. Depending on the problem being solved, this can increase the number of degrees of freedom unnecessarily. However, especially in 2D, the ideas of the previous section can be easily extended to arbitrarily irregular meshes. No further modifications to the assembling procedure are necessary, as it is sufficient to properly set up the assembly lists.

In the following we shall consider a simple 2-irregular mesh[2] shown in Figure 3.11. The construction of vertex basis functions (such as φ_B) and edge basis functions (e.g., φ_{AB}) will be described.

Vertex Functions

Every vertex basis function on an irregular mesh must attain the value of 1 in a certain vertex, drop linearly to zero along the immediate edges and also must

[2]Please note that vertex G is now constrained.

be continuous in all constrained vertices. Before presenting the algorithm to construct such functions, let us introduce *partial assembly lists* associated with vertex nodes to store information on basis functions. A partial assembly list is a pair of vectors $\{\boldsymbol{d}^V, \boldsymbol{c}^V\}$, where $\boldsymbol{d}^V$ lists the numbers of all basis functions that are nonzero in vertex V and $\boldsymbol{c}^V$ contains their values in V. The partial assembly lists are then collected on each element from the vertex nodes (and also supplemented by information from the edge and bubble nodes) to form the final assembly lists we have seen in Section 3.3.1.

The following algorithm assigns all partial vertex assembly lists, which when collected form continuous vertex basis functions on an irregular mesh:

1. Assign partial assembly lists to all unconstrained vertices in the mesh. The assembly lists have the simple form
$$\{\boldsymbol{d}^V, \boldsymbol{c}^V\} = \{(d^V), (1)\}.$$
2. Repeat until finished:
   (a) Select an unprocessed constrained vertex node C whose both parent nodes (constrained or unconstrained) already have partial assembly lists; denote the lists $\{\boldsymbol{d}^A, \boldsymbol{c}^A\}$ and $\{\boldsymbol{d}^B, \boldsymbol{c}^B\}$.
   (b) Merge the parent assembly lists to form $\{\boldsymbol{d}^C, \boldsymbol{c}^C\}$. The vector $\boldsymbol{d}^C$ contains all elements from $\boldsymbol{d}^A$ and $\boldsymbol{d}^B$,
$$d_i^C \in \boldsymbol{d}^C \;\Leftrightarrow\; d_i^C \in \boldsymbol{d}^A \text{ or } d_i^C \in \boldsymbol{d}^B \quad \text{for some } i,$$
   and moreover the elements are sorted[3] and unique,
$$d_1^C < d_2^C < \ldots < d_n^C.$$
   The coefficient vector $\boldsymbol{c}^C$ contains either halves or averages of the coefficients in $\boldsymbol{c}^A$ and $\boldsymbol{c}^B$:
$$c_k^C = (c_i^A + c_j^B)/2, \text{ if both } d_k^C = d_i^A \text{ and } d_k^C = d_j^B,$$
$$c_k^C = c_i^A/2, \text{ if only } d_k^C = d_i^A,$$
$$c_k^C = c_j^B/2, \text{ if only } d_k^C = d_j^B.$$
   For a good example of merging two partial assembly lists, see $\{\boldsymbol{d}^L, \boldsymbol{c}^L\}$ below, which is created from $\{\boldsymbol{d}^D, \boldsymbol{c}^D\}$ and $\{\boldsymbol{d}^G, \boldsymbol{c}^G\}$.

The algorithm constructs all vertex basis functions on the mesh at once. Indirect constraints (constrained nodes depending on other constrained nodes) are

[3]This makes the implementation simpler and also more efficient, since then two partial assembly lists can be merged in linear time.

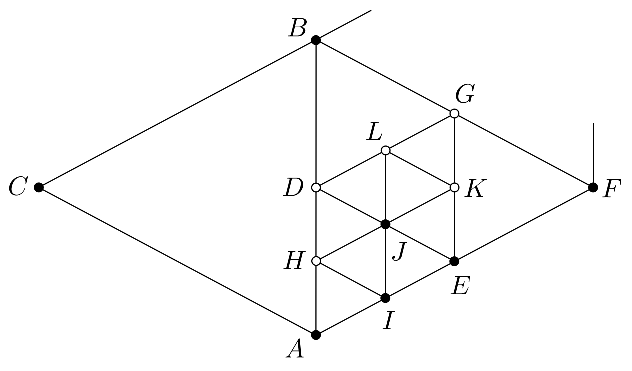


Figure 3.11: 2-irregular triangular mesh.

accounted for in step (a) above, since a parent assembly list can easily belong to a previously assigned constrained node, which is the case for nodes L and K in our example. The following are partial assembly lists, as obtained by the algorithm, for the constrained nodes D, G (assigned first) and H, L, K:

$$\begin{aligned} \boldsymbol{d}^D &= (\ d^A,\ d^B\ ), & \boldsymbol{d}^G &= (\ d^B,\ d^F\ ), & \boldsymbol{d}^H &= (\ d^A,\ d^B\ ),\\ \boldsymbol{c}^D &= (\ \tfrac{1}{2},\ \tfrac{1}{2}\ ), & \boldsymbol{c}^G &= (\ \tfrac{1}{2},\ \tfrac{1}{2}\ ), & \boldsymbol{c}^H &= (\ \tfrac{3}{4},\ \tfrac{1}{4}\ ),\end{aligned}$$

$$\begin{aligned} \boldsymbol{d}^L &= (\ d^A,\ d^B,\ d^F\ ), & \boldsymbol{d}^K &= (\ d^B,\ d^E,\ d^F\ ),\\ \boldsymbol{c}^L &= (\ \tfrac{1}{4},\ \tfrac{1}{2},\ \tfrac{1}{4}\ ), & \boldsymbol{c}^K &= (\ \tfrac{1}{4},\ \tfrac{1}{2},\ \tfrac{1}{4}\ ).\end{aligned}$$

Assuming that the elements are linear, the following is then a complete assembly list for the element JKL, for instance:

$$\begin{aligned} \boldsymbol{\varphi}^{JKL} &= (\ \varphi^{v_1},\ \varphi^{v_2},\ \varphi^{v_2},\ \varphi^{v_2},\ \varphi^{v_3},\ \varphi^{v_3},\ \varphi^{v_3}\ ),\\ \boldsymbol{d}^{JKL} &= (\ d^J,\ d^B,\ d^E,\ d^F,\ d^A,\ d^B,\ d^F\ ),\\ \boldsymbol{c}^{JKL} &= (\ 1,\ \tfrac{1}{4},\ \tfrac{1}{2},\ \tfrac{1}{4},\ \tfrac{1}{4},\ \tfrac{1}{2},\ \tfrac{1}{4}\ ).\end{aligned}$$

Figure 3.12 shows one of the basis functions, φ_B.

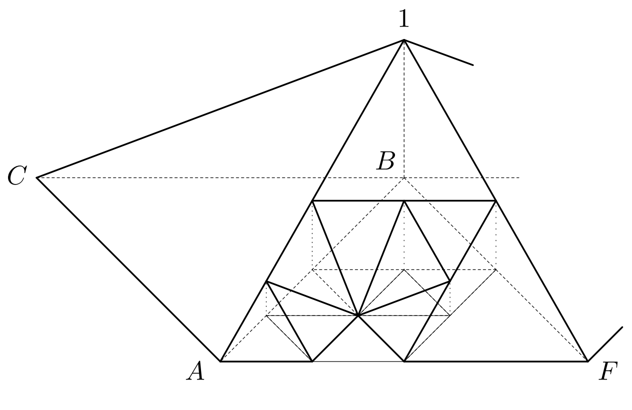


Figure 3.12: Basis function φ_B on the 2-irregular mesh.

Edge Functions

As we have seen in the one-level example, edge basis functions consist of (1) a regular edge shape function on the unconstrained side of the edge in question, (2) a part formed by vertex functions on the other side (dropping linearly to zero as quickly as possible) and finally, (3) constrained edge functions on elements sharing an edge with the large, constraining element. This is illustrated in Figure 3.13.

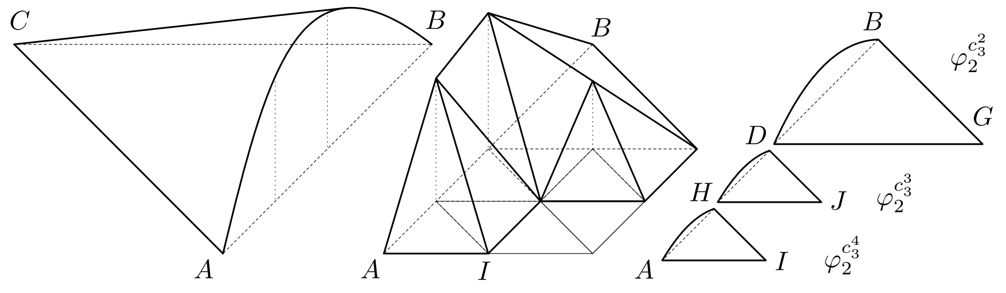


Figure 3.13: Construction of the edge function φ_{AB} on the 2-irregular mesh.

The vertex part is easily generated by a minor addition to the vertex algorithm on page 31. When processing nodes D and H, the DOF numbers and values of the constraining edge function are appended to their partial assembly lists:

$$\begin{array}{llll} \boldsymbol{d}^D = (\ d^A,\ d^B,\ d^{AB}\ ), & \qquad \boldsymbol{d}^H = (\ d^A,\ d^B,\ d^{AB}\ ), \\ \boldsymbol{c}^D = (\ \frac{1}{2},\ \frac{1}{2},\ y^D\ ), & \qquad \boldsymbol{c}^H = (\ \frac{3}{4},\ \frac{1}{4},\ y^H\ ), \end{array}$$

Indirectly constrained nodes, such as L, receive the correct values (e.g., $y^D/2$) automatically in the successive steps of the algorithm.

The basis function is completed by adding the constrained edge shapes $\varphi_p^{c_j^k}$. In the case of arbitrarily-level hanging nodes, there may be a large number of them and for an easier bookkeeping of their generating coefficients $\alpha_{p,i}^k$ we find it useful to introduce a numbering scheme for the edge position intervals

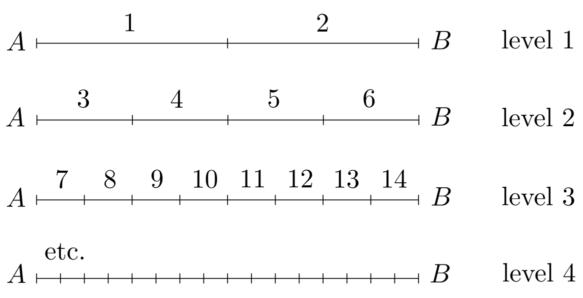


Figure 3.14: Edge interval numbering.

k, illustrated in Figure 3.14. This arrangement has the property that if k_0 is an interval number then the numbers of its immediate sub-intervals are $k_1 = 2k_0 + 1$ and $k_2 = 2k_0 + 2$. Each interval number k uniquely identifies the relative endpoints a, b of the interval within the edge AB, $-1 \le a < b \le 1$.

The constrained edge function coefficients $\alpha^k_{p,i}$ are obtained by solving the following system of $p - 1$ linear equations,

$$\sum_{i=2}^{p} \alpha^k_{p,i}\, \varphi^{e_j}_i(y^p_m) = \psi(y^p_m) - g(y^p_m), \qquad 2 \le m \le p,$$

where ψ is the constraining edge function whose section we want to match, g is a linear function such that $g(a) = \psi(a)$ and $g(b) = \psi(b)$, and y^p_m are $p - 1$ interior Chebyshev points of degree p.

Let us conclude this section with a remark on the minimum rule regarding edges with hanging nodes. In theory, the constraining edge should carry the minimum polynomial degree of all elements touching the edge. This means that a tiny constrained element with a low degree can influence many other elements with higher polynomial degrees, including the constraining element itself. According to our experience, this has a negative effect on the quality of the approximation, especially in the arbitrarily irregular case. We therefore opt to violate the minimum rule on constrained edges, always assigning the degree of the constraining element to the the constraining edge.

3.3.3 Mesh Data Structures

An indivisible part of the implementation of an adaptive FEM solver with hanging nodes is the design of the supporting mesh data structure. In traditional codes mesh management is trivial: it is sufficient to store a fixed array of vertices and a fixed array of elements (i.e., triples or quadruples of vertex indices). In an adaptive and/or higher-order code the following problems must be considered:

1. Element hierarchy must be stored, since refinement and possibly also coarsening of the mesh is necessary.
2. Vertex and edge nodes (and their hierarchy) need to be stored and tracked in higher-order FEM.
3. Standard techniques such as element neighbor lists may no longer work, since we are dealing with irregular meshes.
4. Element and node arrays must be dynamically sized.

In the following we present an original design of data structures for adaptive irregular meshes which meets the above requirements yet retains extreme simplicity.

Node and Element Structures

It should be noted first that we have departed from traditional mesh data structures storing complete information about the finite element problem. Our meshes contain geometrical information only – the remaining data, including basis function indices, boundary conditions, polynomial degrees, etc., are stored in separate classes describing concrete finite element spaces (H^1, $\boldsymbol{H}(\mathrm{curl})$, ...) and are accessible via the `id` numbers of nodes and elements. This was done to allow multiple spaces and multiple element types to coexist on the same mesh, which is vital for solving multiphysics and coupled problems.

The entire mesh is defined by two arrays of the following C structures. The first structure stores all nodes:

```
struct Node
{
  // int id; (implied)
  unsigned ref:30;
  unsigned used:1;
  unsigned type:1;

  union {
    struct { // vertex node data
      double x, y;
    };
    struct { // edge node data
      int marker;
      Element* elem[2];
    };
  };

  int p1_id, p2_id; // parent node ids
  Node* next_hash;
};
```

The `Node` structure defines both vertex and edge nodes by utilizing the `union` construct. The standard vertex positions `x`, `y`, while typically stored separately, were placed in the vertex variant of the `Node` structure for simplicity. The edge variant contains an edge marker (used for identifying different boundaries of the computational domain) and pointers to the two elements sharing the edge node (useful, e.g., when enforcing the minimum rule for edge polynomial degree).

The `id` number, pointing to extra node data in one or more FE space tables, is implied from the position of the structure in the array. Note that the size of the structure is 32 bytes, thus the division involved in the calculation of `id` can be performed by a bit shift.

The members `type` and `used` determine the variant of the structure and whether the particular item of the node array is used, respectively. The remaining members will be described later.

Elements are defined by the second structure:

```
struct Element
{
  int id;
  unsigned marker : 29;
  unsigned active : 1;
  unsigned used : 1;
  unsigned type : 1;

  Node* vn[4]; // vertex nodes
  union {
    Node* en[4]; // edge nodes
    Element* sons[4]; // son elements
  };
};
```

An element can be either active or inactive, hence the one-bit variable `active`. Active elements store pointers to the appropriate vertex and edge nodes. Inactive elements are those which have been refined and are excluded from the computation. Their purpose is to hold pointers to the descendant elements. The constraint update algorithm requires the inactive elements to preserve the vertex node pointers, which is why the array `vn` is outside the union.

Triangular and quadrilateral elements share the same structure and are distinguished by the member `type`. The fourth vertex node pointer is unused for triangles, but this is worth the simpler code that results from the shared structure. The rest of the variables are analogous to the `Node` structure.

Eliminating Neighbor Search by Hashing

To properly initialize edge node pointers after reading a mesh file, one has to construct neighbor lists for all elements and use them in such a way that only one node is created for each edge. Further problems arise when certain elements are refined after mesh adaptation. Unless hanging nodes are removed by extra refinements, no longer is each edge shared by at most two elements. Standard neighbor lists fail to fully capture such situations and thus more complex data structures, e.g., trees [17], have to be introduced.

We have avoided all of these problems by introducing a function which, given the `id` numbers of two nodes, returns a pointer to the node halfway between them. If no such node exists, it is created first. The task of translating two numbers to a node pointer is accomplished using a hash table.

We are maintaining two independent layers of nodes: the first layer contains all vertex nodes, the second all edge nodes. The following two functions can be called:

```
Node* get_vertex_node(int id1, int id2);
Node* get_edge_node(int id1, int id2);
```

All nodes, apart from being accessible by their `id` number, can be reached using these functions by passing the `id`s of their "parent" nodes. Top-level vertex nodes (those loaded from the mesh file) are stored at the beginning of the node array and can be accessed directly without hashing. Mesh initialization then becomes trivial:

```
nodes = // read all top-level vertex nodes from a file
for (all e in elements) {
  vv[3] = // read element vertex id numbers
  for (i = 0; i < 3; i++) {
    e->vn[i] = &nodes[vv[i]];
    e->en[i] = get_edge_node(vv[i], vv[(i+1)%3]);
  }
}
```

Element refinement is also very straightforward. No care must be taken of the neighboring elements, regardless of their refinement level or even existence:

```
Element* create_triangle(Node* v0, Node* v1, Node* v2)
{
  Element* e = new Element;
  e->active = 1; e->type = 0; // etc.
  e->vn[0] = v0; e->vn[1] = v1; e->vn[2] = v2;
  e->en[0] = get_edge_node(v0->id, v1->id);
  e->en[1] = get_edge_node(v1->id, v2->id);
  e->en[2] = get_edge_node(v2->id, v0->id);
  // reference all nodes of the new element
  return e;
}

void refine_element(Element* e)
{
  Node* x0 = get_vertex_node(e->vn[0]->id, e->vn[1]->id);
  Node* x1 = get_vertex_node(e->vn[1]->id, e->vn[2]->id);
  Node* x2 = get_vertex_node(e->vn[2]->id, e->vn[0]->id);
  e->sons[0] = create_triangle(e->vn[0], x0, x2);
  e->sons[1] = create_triangle(x0, e->vn[1], x1);
  e->sons[2] = create_triangle(x2, x1, e->vn[2]);
  e->sons[3] = create_triangle(x0, x1, x2);
  e->active = 0;
  // un-reference all nodes of e (to be explained)
}
```

Each hash table is implemented as an array of linked lists of hash synonyms (*open hash*). This hash table organization has the advantage of simple node removal, which is required if a node is no longer needed by any element. Synonym nodes are linked by the pointer `next_hash` and are distinguished by the parent node numbers `p1_id` and `p2_id`. To ensure that `get_*_node(id1, id2)` gives the same result as `get_*_node(id2, id1)`, the smaller parent `id` is always stored in `p1_id` and each query is modified accordingly as well.

There are two important parameters in the design of any hash table: the size of the table and the choice of the hash function. To prevent the table from becoming overfilled we set its size to about one half of the expected number of nodes when loading the mesh. The size is not required to be a prime number for this type of hash table. We always choose it to be a power of two to avoid the modulo operation in the hash function. A satisfactory spreading of the table items is achieved by the following hash function:

```
int hash(int id1, int id2)
  { return (A*id1 + B*id2) & (table_size-1); }
```

where `A` and `B` are large integer constants. The number of synonyms in each non-empty list is then two on average and very scarcely greater than four, which outperforms most tree representations (tested on a mesh with around one million nodes).

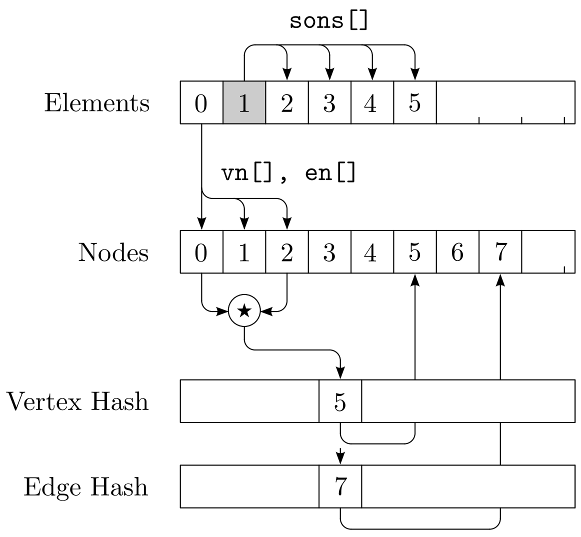


Figure 3.15: Data structures for mesh storage.

Figure 3.15 provides a graphical summary of the data structures. Initially there are just two elements in the mesh, each of them containing pointers to its vertex and edge nodes. When element number 1 is refined, it is deactivated and pointers to newly created elements 2, 3, 4 and 5 are stored in the array `sons` of element 1. We also see, thanks to the hash tables, that the edge 0-2 has vertex node 5 in its middle as well as edge node number 7.

Determining Node Type

Figure 3.16 shows a simple mesh with all its nodes displayed. Five types of nodes can be identified: standard vertex nodes (A), standard edge nodes (B), constrained vertex nodes (C), constrained edge nodes (D) and finally standard edge nodes and constrained vertex nodes at the same place (E). The last case is the reason for having two layers of nodes in two separate hash tables.

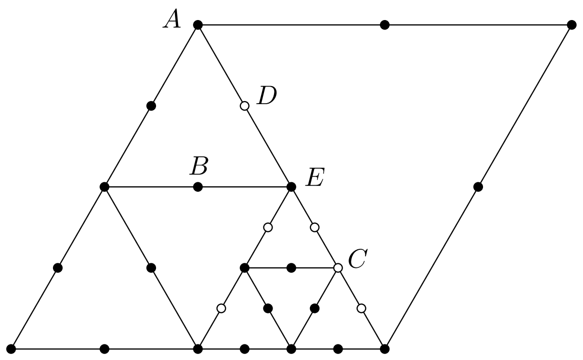


Figure 3.16: Types of nodes in an irregular mesh.

As the elements in the mesh get refined (or un-refined), some nodes are created, others are removed and some change their types. One has to be able to quickly recognize whether a node is constrained or unconstrained without searching in the element hierarchy. This is achieved through the member `ref` of the structure `Node`. At all times this variable holds the number of elements pointing to the node. This is the reason why all nodes of an element must be *referenced* (`ref` increased) when creating the element and *un-referenced* (`ref` decreased) when the element is being refined or destroyed, as shown in Section 3.3.3.

The cases A and C can be distinguished just by looking at `ref`. If `ref` = 3, the vertex node is constrained, if `ref` > 3 it is not constrained. Top-level vertex nodes have `ref` artificially increased to a large number, which ensures that they are always treated as unconstrained. Similarly, if `ref` = 1 for an edge node, it is constrained D; if `ref` = 2 it is unconstrained D.

The case E can be detected by calling the function `peek_*_node`, which works the same way as `get_*_node`, with the exception that the node is not created if it does not exist. The case E is important, since it is a starting point for the constraint update algorithm.

A node is destroyed as soon as its `ref` reaches zero.

Chapter 4

Automatic *hp*-Adaptivity for Stationary Problems

Having described a higher-order solver capable of handling irregular meshes that arise during the adaptive process, we can now focus on the algorithm responsible for the generation of the sequence of optimal *hp* meshes.

This chapter briefly introduces the types of adaptive strategies and presents a survey of existing *hp*-adaptive algorithms. An existing algorithm based on the so-called reference solution is described in more detail, since it is a basis for our two improved algorithms, presented in the remainder of the chapter.

4.1 Introduction

While it is possible in some cases to design an optimal finite element mesh for a given problem *a priori*, in practice this is usually not the case. Most of the time the exact behavior of the solution to a PDE is not known ahead, and a mesh giving a good approximation must be arrived to by successive improvements (refinements) of the mesh. This process is known as adaptive mesh refinement and has been studied extensively since the 1970s. Several basic approaches to mesh adaptation exist, the most common being the following:

- h-adaptivity (h-FEM): Elements with large error estimate are subdivided, thus reducing the mesh diameter, usually denoted as h, in problematic parts of the domain. This is by far the most popular adaptive strategy and is suitable for problems exhibiting singularities and oscillations and for problems with multiple spatial scales. The approximation error is typically reduced with a negative power of the number of degrees of freedom.

- *p*-adaptivity (*p*-FEM): Instead of splitting elements, their polynomial degrees (denoted *p*) are increased. This approach is advantageous if the solution to the PDEs is smooth, in which case exponential convergence of the approximation error may be achieved [51]. However, for non-smooth solutions the method yields unsatisfactory results and performs worse than *h*-adaptivity.

- *r*-adaptivity: Vertices of the mesh are merely relocated, for the mesh to better fit the solution. The goal is to make the mesh denser in problematic areas or to align it with the anisotropy of the solution. This method will not be discussed here.

An indivisible part of all adaptive methods is an error estimator that can be used to judge the quality of the approximation. A vast amount of literature exists on *a posteriori* error estimation, see for example [1, 53]. Some estimators (such as the Zienkiewicz-Zhu estimator) post-process the solution to obtain a smoother approximation which is then subtracted from the solution to give a rough estimate of the error function. Other estimators are based on the computation of local residual problems, each of which produces an estimated measure of error on each element (error indicator). Most estimators are designed for a specific class of PDEs – typically for the most well-behaved class of elliptic PDEs. In any case these analytic estimates produce an error indicator in the form of one number per element, which is then used to select elements to be refined by one of the methods above.

In the mid-1980s Szabó and Babuška [28, 29, 51] introduced the *hp* version of the finite element method, which combines the best from both *h*-FEM and *p*-FEM, and proved (in 1D) that *hp*-FEM is capable of exponential convergence rates. This is achieved by performing *h*-refinements where the solution is not regular and where *p*-FEM would fail, but retaining the possibility of using higher-order polynomials where the solution is sufficiently smooth.

The difficulty is that standard error estimates (if at all available for the given problem) only provide the information which elements to refine, but fail to determine whether one should perform an *h*- or a *p*-refinement. An algorithm which attempts to do that is called an *hp*-adaptive strategy.

4.2 Existing *hp*-Adaptive Strategies

An excellent survey of 15 different *hp*-adaptive strategies is conveniently available in [35]. Here we briefly describe some of them and will later compare the performance of methods developed in this chapter with several of these existing algorithms.

A priori knowledge A simple hp-adaptive strategy that uses *a priori* knowledge about the behavior of elliptic PDEs was presented in [2]. It is known that singularities in linear elliptic PDEs with piecewise-smooth coefficients and boundary data only have singularities in reentrant corners of the domain and in points where the coefficients and boundary conditions change. The user *a priori* marks potentially problematic vertices and the adaptive strategy then always selects h-refinements for elements containing these vertices, otherwise it performs p-refinements.

Type parameter A strategy presented by Babuška and Gui [30] uses the comparison of two local Neumann error estimates η_{i,p_i} and η_{i,p_i-1}, evaluated with different precisions, to assess the perceived smoothness $R_{K_i} = \frac{\eta_{i,p_i}}{\eta_{i,p_i-1}}$ of the solution on element K_i. The so-called type parameter γ specified by the user, $0 \leq \gamma \leq \infty$, then determines the type of refinement. If $R_{K_i} \leq \gamma$, an h-refinement is performed, otherwise a p-refinement is selected.

Legendre coefficient decay Several hp-adaptive strategies are based on examining the decay of coefficients in an expansion of the solution u in Legendre polynomials. In 1D, the approximate solution u_i on element K_i with degree p_i can be written as $u_i(x) = \sum_{j=0}^{p_i} y_j \mathrm{L}_j(x)$, where L_j is a Legendre polynomial of degree j scaled to the interval of element K_i. Mavriplis [33] estimates the decay rate of the coefficients by a least squares fit of the last four coefficients y_j to $Ce^{-\sigma j}$. Elements are p-refined where $\sigma > 1$ and h-refined where $\sigma \leq 1$. Several variations on this approach were described and the technique was later extended into 2D.

Nonlinear programming Each step of hp-adaptivity can also be posed as a nonlinear optimization problem [36]. The current mesh with elements $\{K_i\}$ is viewed as parametrized by element degrees $\{p_i\}$ and h-refinement levels $\{l_i\}$. The objective is to determine new mesh parameters $\{\hat{p}_i\}$ and $\{\hat{l}_i\}$ so that the total estimated error $\sum \hat{\eta}_i^2$ is less than some prescribed error, the number of degrees of freedom is minimal and at the same time a number of constraints on mesh compatibility are satisfied. Since the $\hat{l}_i$ and $\hat{p}_i$ are integers, the optimization problem is NP-hard. To make the problem tractable, the integer requirement is dropped before applying standard optimization software, and then reintroduced.

Smoothness prediction A strategy proposed by Melenk and Wohlmuth [34] uses the theory of linear elliptic PDEs to predict what the error should be following a refinement, assuming the solution is smooth. When performing an h-refinement, the expected error of the four children should in theory be $2^{-p_i} e_i \gamma_h$, where e_i and p_i are the error and polynomial degree of the parent element, respectively, and γ_h is an additional user specified correction. If a p-refinement is performed, exponential convergence is expected, so the predicted error estimate is $\gamma_p e_i$, where $\gamma_p \in (0,1)$ is another user specified parameter. When the actual analytic error estimate η_j of a child becomes available, it is compared to the predicted error estimate. If the estimate is less than or equal to the predicted error estimate, then p-refinement is indicated for the child. Otherwise, h-refinement is indicated since presumably the assumption of smoothness was wrong.

Reference solution strategies Demkowicz et al. [18, 16] developed an hp-adaptive strategy based on projection-based interpolation of the reference solution, a greatly enriched version of the current (coarse) solution. The difference between the coarse and the reference solution replaces the standard error estimator. As this strategy is the basis for our method, it is described in more detail in the following section.

4.3 Projection-Based-Interpolation Algorithm

Since most analytic error estimators are limited to elliptic problems and they do not provide enough information for hp-adaptivity anyway, Demkowicz [18] resorts to estimating the approximation error by subtracting the coarse solution $u_{h,p}$ from the reference solution $u_{h/2,p+1}$. The reference solution is obtained by solving the problem on a mesh uniformly refined in both h and p. Obviously, such approach is very costly because the reference problem is much larger. However, there are several aspects that justify this method:

- The reference solution works for virtually any PDE, not only for the simplest ones, as we will see in Chapter 6.
- Unlike standard error estimators, it provides the actual shape of the error, which can be used to decide between many refinement options, not just p or h. This results in fewer iterations of the whole process.
- The reference solution is not discarded after the computation – in fact it can be used as the final, very accurate result, which means that the coarse solution can aim at somewhat less strict error levels than those actually required.
- Each reference solution $u^n_{h/2,p+1}$ at step n can (and should) be reused to calculate the reference solution $u^{n+1}_{h/2,p+1}$ at the next adaptive step $n+1$, using multigrid or similar techniques.

The general outline of an hp-adaptive algorithm based on the reference solution is the following:

1. Solve the problem on the current mesh to obtain $u_{h,p}$.
2. Create a temporary copy of the current mesh and refine it uniformly in h and then in p.

3. Solve the problem on the refined mesh to obtain the reference solution $u_{h/2,p+1}$ and the error estimate $e_{h,p} = u_{h,p} - u_{h/2,p+1}$.
4. If $||e_{h,p}|| < tol$ then stop, $u_{h/2,p+1}$ is the result.
5. Use the reference solution to determine optimal refinement of the current mesh.
6. Repeat from step 1.

The heart of the algorithm is step 5. Demkowicz's main idea is to refine the current mesh in such a way that the coarse solution *interpolates* the reference solution as well as possible while using as few degrees of freedom as possible. Let us first describe the 1D version of the algorithm, since the 2D version is built on top of it.

4.3.1 1D Algorithm

The algorithm relies on projection-based interpolation. To obtain the best interpolant $g_{h,p} = \Pi w$ of a function $w(x)$ in some finite element space V_{hp} one typically employs the classical orthogonal projection, which is equivalent to solving a system of linear equations of the form

$$\textstyle\sum_{j=1}^{N} y_j (v_j, v_i)_{V_{hp}} = (w, v_i)_{V_{hp}} \qquad \text{for all } i = 1, 2, \dots, N.$$

The best interpolant is then

$$\Pi^{\mathrm{OG}}\, w = \textstyle\sum_{j=1}^{N} y_j v_j.$$

However, this method is computationally expensive. A slightly less accurate but much faster algorithm is the projection-based interpolation, which combines nodal interpolation of vertex values with local orthogonal projections on element interiors. The projection-based interpolant

$$g_{h,p} = \Pi^{\mathrm{PB}}\, w = g^{v}_{h,p} + g^{b}_{h,p}$$

is now sought as a sum of two parts. The vertex part is defined as a piecewise-linear function which satisfies $g^{v}_{h,p}(x_i) = w(x_i)$ where x_i are the coordinates of mesh vertices. The bubble (interior) interpolant $g^{b}_{h,p}$ is calculated by projecting the residual $w - g^{v}_{h,p}$ to the spaces formed by element bubble functions. This can be done locally in each element thanks to the fact that the residual vanishes at all mesh points x_i. The solution of a series of local problems is cheaper in terms of CPU time than one large classical Π^{OG} projection problem.

The 1D *hp*-adaptive algorithm has three steps, shortly described below. For more details we refer to the book [16], page 95.

Step 1: determine element error decrease rates. For each element K, several refinement options (candidates) are tested. One possible refinement is to increase the polynomial degree p of the element to $p+1$. Other choices are the so-called competitive h-refinements. If the element is subdivided, two new elements with degrees p_1 and p_2 are created (remember we are in 1D), and we need to choose these degrees. In order for the h-refinements to be comparable with the p-refinement (which has $p+1$ degrees of freedom), only such combinations of p_1 and p_2 which satisfy $p_1 + p_2 = p + 1$ are considered.

For each refinement candidate, the element error decrease rate is calculated,

$$\mathrm{rate}_K = \frac{||u_{h/2,p+1} - \Pi^{\mathrm{PB}}_{hp}\, u_{h/2,p+1}||^2_{H^1} - ||u_{h/2,p+1} - \Pi^{\mathrm{PB}}_{cand}\, u_{h/2,p+1}||^2_{H^1}}{N_{cand} - N_{hp}},$$

where $\Pi^{\mathrm{PB}}_{hp}\, u_{h/2,p+1}$ is the projection-based interpolant of the reference solution restricted to element K, similarly $\Pi^{\mathrm{PB}}_{cand}\, u_{h/2,p+1}$ is the projection onto the candidate element(s) and finally N_{hp} and N_{cand} are the dimensions (degrees of freedom) of the respective finite element spaces.

In this way we obtain a list of refinement candidates (one p-refinement and p different competitive h-refinements) along with a measure of their "profitability". However, this list could potentially contain more h-refinement candidates, for all combinations of p_1 and p_2 (if we could afford the CPU costs), because some of them may have better rates than the competitive h-refinements. As a compromise, Demkowicz adds several more candidates that are based on the best competitive h-refinement. The interpolation errors on its two son elements are examined and the son element which contributes more than 70% of the total error is assigned one higher polynomial degree. Thus a new h-refinement candidate is added and the process is repeated, with the newly added candidate serving as a new starting point. This process is referred to as following the biggest subelement error path.

Step 2: determine which elements to refine. Among the best element error decrease rates found in Step 1 the maximum rate is found, denoted rate_{max}. All elements that have a rate of at least $1/3 \cdot \mathrm{rate}_{max}$ are marked for refinement. The factor $1/3$ is somewhat arbitrary but a justification related to an integer version of the method of steepest descent is given in [16].

Step 3: refine elements optimally. All selected elements are now refined. If for some element the p-refinement won, the element is simply p-refined. However, if the best refinement is an h-refinement, we want to maximize its p_1 and p_2 degrees. We follow again the biggest subelement error refinement path, but this time only as long as the error decrease rate is above $1/3 \cdot \mathrm{rate}_{max}$, to remain consistent with Step 2.

4.3.2 2D Algorithm

The 2D algorithm is based on the 1D version we have just described. In essence, the 1D algorithm is first used for all edges of the mesh and the resulting edge refinements then determine element h-refinements. After that, optimal refinements of element interiors are determined. We again refer to the book [16], page 227, for some of the more technical details.

Step 1: determine edge error decrease rates. This step is identical to the first step of the 1D algorithm, with the only exception that a different norm is used in the projection problems and the corresponding evaluation of the projection errors. Ideally, the $H^{\frac{1}{2}}$ seminorm should be used, however in practice a weighted H^1 seminorm is used instead, because it is easier to work with and because it scales in the same way with edge length as the $H^{\frac{1}{2}}$ seminorm. The output of this step are optimal edge refinements with the associated error decrease rates.

Step 2: determine which edges to refine. As in the 1D algorithm, all edges whose error decrease rate is at least 1/3 of the maximum of all edge rates are selected for refinement.

Step 3: determine h-refinement of elements The edge refinements have direct influence on element h-refinements. If any of an element's edges is to be split, the whole element is also marked for h-refinement. There are two additional issues that need to be taken care of: anisotropic refinements and maintaining 1-irregularity.

To be able to handle boundary layers and other phenomena where the solution behaves anisotropically, the algorithm allows quadrilateral elements to be only h-refined along one of the reference domain axes, producing just two new quadrilaterals. If the edge refinements suggest such a split, the derivatives of the error function are examined to check whether the solution is really anisotropic inside the element. A conservative threshold is used in order to allow anisotropic refinements only if the solution exhibits strong 1D behavior, otherwise the element is flagged for standard four-way h-refinement.

1-irregularity is enforced through additional checks. Whenever an edge that is only shared by one element needs to be h-refined, the neighboring larger element is forcibly marked for h-refinement as well, in order to keep the mesh 1-irregular. After all the element refinement flags have been determined, all refinements are actually performed and thus the new mesh topology is finally established.

Step 4: determine optimal degrees for refined edges. The same strategy as in 1D is followed when assigning final edge degrees: the selected edges are either p-refined or the biggest subelement error path is used to assign the degrees on h-refined edges. Again the only difference is in the choice of the weighted H^1 seminorm.

Step 5: determine element error decrease rates. At this point of the algorithm the situation is as follows: the topology of the mesh is final, all degrees of the original edges are assigned and the degrees of interior edges in h-refined elements are implied by the minimum rule from the (yet unknown) polynomial degrees of the h-refined elements. What remains is to calculate the error decrease rates for the 2D p-refinement and h-refinement candidates. Similarly as in Step 1 of the 1D algorithm, we follow the maximum subelement error path, but since we are now dealing with 2D elements, there are more options when following the path. Both triangles and quadrilaterals can have up to four sons, moreover all quadrilaterals can be assigned two different polynomial degrees if the solution is anisotropic. The directional structure of the error function is examined again and if 1D behavior is detected, the degree of approximation is raised in one direction only. See [16] for many more details.

Step 6: determine optimal degrees for element interiors. Finally, the internal polynomial degrees are assigned analogously to Step 3 of the 1D algorithm, except that the subelement error path is cut off at $1/3 \cdot \text{rate}_{max}$ where the maximum rate is now chosen as

$$\text{rate}_{max} = \max \{ \text{element rate}_{max},\ \text{edge rate}_{max} \}.$$

4.4 Our *hp*-Adaptivity Algorithm

Demkowicz's algorithm is exceedingly difficult to implement – to our best knowledge, no one has reimplemented it in their own code so far. We also felt that similar results should be possible to achieve with a simpler algorithm, such as the one presented in this section.

We are building on the same idea of using the reference solution to estimate the error and our algorithm has an identical outer loop as the algorithm described in Section 4.3. However, the determination of optimal refinements, given the coarse and reference solutions (step 5), is substantially simpler, both to implement and to describe.

The main concept of selecting good refinements is again that the coarse solution should interpolate the reference solution as well as possible. Unlike the 2D version of Demkowicz's algorithm, however, our algorithm is based on element refinements, not edge refinements. For each element we simply try out several refinement options (candidates), assess the interpolation properties of each candidate and select the best performing candidate. Some of the refinement candidates are shown in Figure 4.1.

In detail, the simplified algorithm consists of the following three steps:

Step 1: determine which elements to refine. The coarse solution is subtracted from the reference solution on each element K_i and the local error estimate e_i is calculated:

$$e_i = ||u_{h/2,p+1} - u_{h,p}||_{H^1(K_i)}.$$

The elements are sorted by e_i in descending order (this will be utilized in Step 3). The maximum element error $e_{max} = \max\{e_i\}$ is determined and all elements for which

$$e_i > k \cdot e_{max}$$

are marked for refinement. The coefficient k controls the size of the batch of elements that is refined in each iteration. It is specified by the user, typical values are between 0.2 and 0.7. Higher values lead to better convergence curves at the cost of many iterations, lower values speed up the algorithm but the convergence may be somewhat worse.

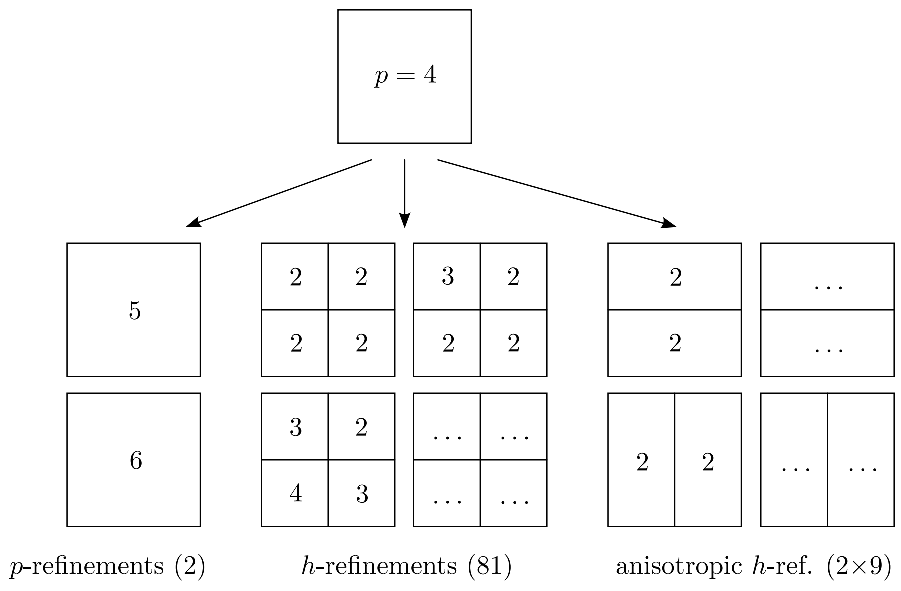


Figure 4.1: *hp*-refinement candidates.

Step 2: assess refinement candidates. For each of the selected elements a list containing the following refinement candidates is created:

- Two *p*-refinement candidates, $p + 1$ and $p + 2$.
- 81 *h*-refinement candidates consisting of four $h/2$ elements with all combinations of degrees varying between $\lceil p/2 \rceil$ and $\lceil p/2 \rceil + 2$.
- If dealing with quadrilaterals, 2×9 anisotropic *h*-refinement candidates are added with similar combinations of degrees as for the isotropic *h*-refinements.

- For the purpose of evaluation of other candidates, an artificial "no refinement" candidate with degree p is added to the list.

All candidates whose polynomial degrees exceed the maximum degree (currently 10 in our code) are discarded. We also do not allow the degrees in h-refinements to reach $p+1$, which is the degree of the reference solution.

For each candidate in the list a temporary mesh is created along with a temporary finite element space which corresponds to the polynomial degrees of the candidate's elements. No boundary conditions are prescribed on the boundaries of the temporary domain – all nodes are left unconstrained. The corresponding region of the reference solution $u_{h/2,p+1}$ is then projected onto the candidate space and the projection error

$$e_{cand} = ||u_{h/2,p+1} - \Pi^{\text{OG}}_{cand}\, u_{h/2,p+1}||_{H^1(K_i)}$$

is evaluated. Standard H^1 orthogonal projection is used, because it is easier to implement than projection-based interpolation and because all projection problems are small (local to the element K_i being refined).

Since the number of candidates is relatively high, we keep the projection problems assembled and LU-decomposed and use them repeatedly for different elements K_i by just replacing the right-hand sides. We assume that the shape of the coarse mesh elements K_i does not influence the projections too much. All temporary meshes for the candidates are therefore created in the region $[-1,1]^2$ and they do not coincide with the actual elements. The storage and reuse of all projection problems that are encountered has a very positive effect on the speed of the algorithm, but unfortunately it is still not enough. We will address this problem in Section 4.5.

Another assumption that we make is that the neighboring elements of K_i have little or no influence on the refinement of K_i. We can thus consider each element separately when making the projections and when selecting the best candidates. Conversely, any refinement of K_i does not immediately influence its neighbors thanks to our use of arbitrarily irregular meshes.

Step 3: select best refinements. For each element selected for refinement, we now take its candidate list and the corresponding projection errors and select an above-average candidate with the steepest error decrease.

The error decrease for a candidate with projection error e_{cand} and degrees of freedom N_{cand} is taken relative to the artificial no-refinement candidate with error e_0 and degrees of freedom N_0,

$$\text{rate}_{cand} = \frac{\ln e_0 - \ln e_{cand}}{N_{cand} - N_0}.$$

In order to discard candidates that may have a high rate_{cand} but are too close to the no-refinement candidate in terms of degrees of freedom (i.e, the candidate represents a very small change), we only consider such candidates for which

$$\ln e_{cand} < \bar{e} + \sigma,$$

where $\bar{e}$ and σ are the mean and standard deviation of the logarithms of all candidate errors:

$$\bar{e} = \frac{1}{n} \sum_{cand} \ln e_{cand},$$

$$\sigma^2 = \frac{1}{n} \sum_{cand} (\ln e_{cand})^2 - \bar{e}^2.$$

We also immediately reject all candidates whose error is higher than e_0. Figure 4.2 illustrates these concepts on an example of a degree 2 element. Many refinement candidates were discarded because either their error was worse than e_0 (for elements of higher degrees this is less common) or because their error was not significantly better than e_0. From the viable candidates an h-refinement with degrees (2,1,1,1) and a p-refinement with $p = 3$ are the best ones. The h-refinement is the eventual winner because it has the steepest error decrease.

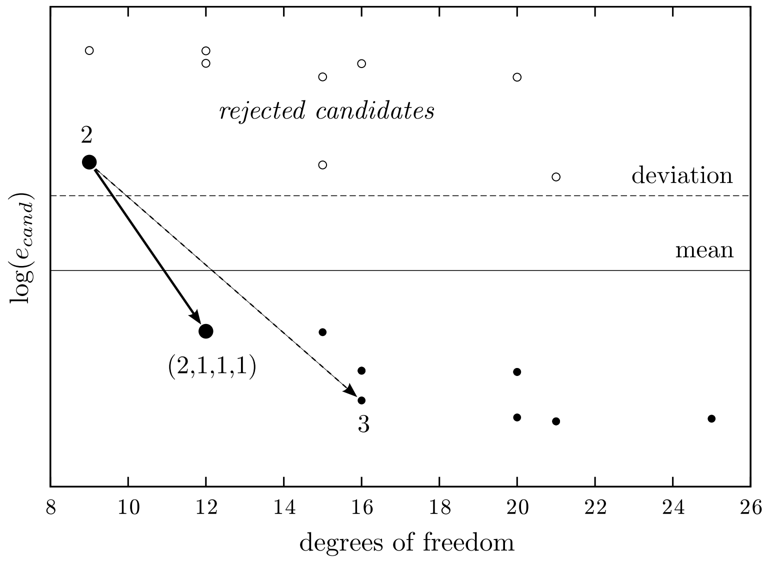


Figure 4.2: Selection of the best refinement candidate.

It should be noted that the mechanism of rejecting candidates with error above $\bar{e} + \sigma$ is somewhat arbitrary and was implemented only to eliminate pathological cases and to speed up the adaptive process.

On the other hand, the use of logarithms of candidate errors in all of the final calculations is not arbitrary. One explanation for this is that we are aiming for the steepest convergence curve in the final graph, which has a logarithmic scale for the error. This means we should also work in a logarithmic setting when assessing the candidates, just as in Figure 4.2. Another justification is that the convergence of *hp*-FEM should be exponential, so for good refinement candidates it should also hold that

$$e_0 \approx e^{-\alpha N_0},$$

$$e_{cand} \approx e^{-\alpha N_{cand}}.$$

Dividing these two equations and taking the logarithm we obtain

$$\ln e_0 - \ln e_{cand} = -\alpha(N_0 - N_{cand}),$$

$$\alpha = \frac{\ln e_0 - \ln e_{cand}}{N_{cand} - N_0},$$

which is our formula for rate_{cand} that we are trying to maximize.

Unlike in the Demkowicz's algorithm, we do not worry about investing too many degrees of freedom in one step, since we are striving for a long uninterrupted convergence, and not for a particular error level. In other words, if we invest too much in some element in one step, the element will just not be selected for refinement in the next step. We have not encountered any problems with this approach.

Once the best refinement candidate is found for element K_i, the refinement is actually performed and the algorithm moves on to the next element selected for refinement in Step 1. The elements are processed in decreasing order of approximation error, i.e., the worst elements are processed first. The total number of DOFs added to the coarse mesh is monitored during this process and in addition to the parameter k in Step 1 there is another constant D specified by the user, which controls the maximum number of DOFs that can be added on one adaptive iteration. If the number of new DOFs exceeds D the rest of the elements that should have been refined are skipped.

This concludes Step 3 of our *hp*-adaptive algorithm. Its performance will be assessed in Section 4.6.

4.5 Fast Algorithm with Orthonormal Bases

The main problem of our *hp*-adaptive algorithm described in the previous section is that it checks too many refinement candidates. Despite the caching and reuse of the local projection problems, the algorithm is still quite slow in 2D and completely infeasible in 3D. In his algorithm, Demkowicz avoids having too many candidates by only testing several basic ones plus a small number of candidates that lie on the largest sub-element error path. However, this approach is just a heuristic which is not guaranteed not to miss good candidates. We have therefore implemented a different approach that retains the large number of tested candidates but which instead makes the projections extremely cheap.

The idea is to always try to project to an orthonormal basis when evaluating the refinement candidates. If the basis functions of the candidate space are orthonormalized (in a preprocessing step), subsequent projections do not require the solution of a linear system, since the matrix of the problem is an identity. All that is needed to obtain the expansion coefficients of the projected function is to evaluate the right-hand side, i.e.,

$$y_i = (u_{ref},\, \tilde{\varphi}_i)_{H^1} \qquad \text{for all } i = 1, 2, \ldots, N,$$

where we have denoted $u_{h/2,p+1} \equiv u_{ref}$ for convenience. For p-refinement candidates, the functions $\tilde{\varphi}_i$ are obtained by applying the standard Gramm-Schmidt orthogonalization process to the set of hierarchic shape functions φ_i of an element of the corresponding degree,

$$\bar{\varphi}_i = \varphi_i - \sum_{j=1}^{i} (\varphi_i, \bar{\varphi}_j)\bar{\varphi}_j,$$

$$\tilde{\varphi}_i = \frac{\bar{\varphi}_i}{||\bar{\varphi}_i||}.$$

Although any set of linearly independent functions could be used as the input of the orthogonalization, we use the shape functions φ_i for their hierarchic properties. When starting the G-S process with linear functions and proceeding in the order of increasing polynomial degree of the shape functions, the hierarchy is preserved also in $\tilde{\varphi}_i$. In other words, if $P^p(K_q)$ and $P^{p+1}(K_q)$ are the polynomial spaces for p-refinement candidates with degrees p and $p+1$, respectively, on the reference domain K_q (or K_t in case of triangular elements), and $d_1 = \dim(P^p)$ and $d_2 = \dim(P^{p+1})$ are their numbers of degrees of freedom, then the degree p candidate will have the basis $\tilde{\mathcal{B}}^p = \{\tilde{\varphi}_1, \tilde{\varphi}_2, \ldots, \tilde{\varphi}_{d_1}\}$ and the degree $p+1$ candidate will have the basis $\tilde{\mathcal{B}}^{p+1} = \{\tilde{\varphi}_1, \tilde{\varphi}_2, \ldots, \tilde{\varphi}_{d_2}\}$. That is, $\tilde{\mathcal{B}}^p \subset \tilde{\mathcal{B}}^{p+1}$.

This allows us to speed up the evaluation of p-candidates even further. The candidates are evaluated from the lowest degree to the highest. If for degree p candidate the projected function is

$$\Pi_p^{\mathrm{OG}}\, u_{ref} = \sum_{i=1}^{d_1} (u_{ref},\, \tilde{\varphi}_i)\, \tilde{\varphi}_i,$$

then to obtain the projection for the $p+1$ candidate we just augment the previous result to

$$\Pi_{p+1}^{\mathrm{OG}}\, u_{ref} = \Pi_p^{\mathrm{OG}}\, u_{ref} + \sum_{i=d_1+1}^{d_2} (u_{ref},\, \tilde{\varphi}_i)\, \tilde{\varphi}_i.$$

We follow a similar approach for h-candidates. However, in order to create a hierarchic set of orthonormal basis functions we must commit a "crime". It is not possible to directly apply the previous ideas to the bases of h-candidate spaces, such as the space of piecewise-quadratic continuous functions depicted in Figure 4.3 (a). There is no ordering of the basis functions that would preserve their hierarchic properties after orthonormalization. The only possibility is to "break" the space into four independent parts, as shown in Figure 4.3 (b), producing a space of functions discontinuous along the internal edges (c).

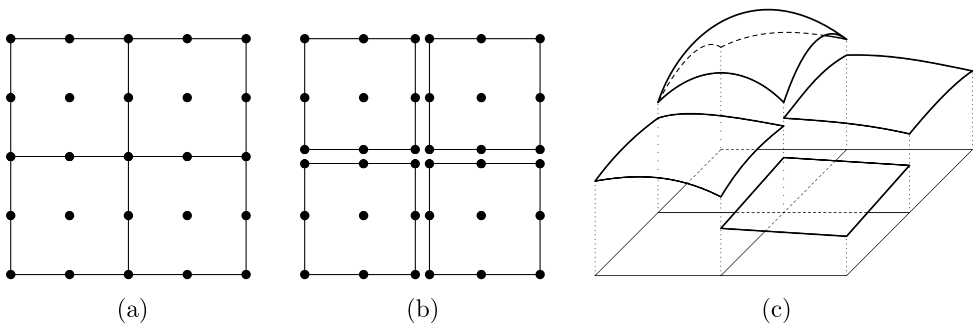


Figure 4.3: (a) Nodes for a standard space of piecewise-quadratic continuous functions, (b) nodes for a "broken" space of piecewise-quadratic functions, (c) example of a discontinuous piecewise-quadratic function.

By enlarging the function spaces in this way we can expect the projection errors to be somewhat smaller for h-candidates than if continuous spaces were used. However, we will see in the following sections that this does not have a substantial effect on the quality of the selection of refinement candidates.

The procedure for evaluating the errors of h-candidates is then as follows. First, an array $h_{i,j}$ of "partial" projection errors is calculated, $1 \le i \le \lceil p/2 \rceil + 2$ and $1 \le j \le 4$,

$$\begin{aligned} h_{i,1} &= ||u_{ref} - \Pi_i^{\mathrm{OG}} u_{ref}||_{K_{q_1}}, & K_{q1} &= (-1,0) \times (-1,0), \\ h_{i,2} &= ||u_{ref} - \Pi_i^{\mathrm{OG}} u_{ref}||_{K_{q_2}}, & K_{q2} &= (0,1) \times (-1,0), \\ h_{i,3} &= ||u_{ref} - \Pi_i^{\mathrm{OG}} u_{ref}||_{K_{q_3}}, & K_{q3} &= (0,1) \times (0,1), \\ h_{i,4} &= ||u_{ref} - \Pi_i^{\mathrm{OG}} u_{ref}||_{K_{q_4}}, & K_{q4} &= (-1,0) \times (0,1). \end{aligned}$$

Again, as for the p-candidates, all projections are calculated in an efficient way by reusing previous (lower-degree) projections. All computations are done on (parts of) the reference domain $K_q = (-1,1)^2$, which means there are some technical issues with derivatives of u_{ref} that are in effect used without transformation to the physical domain of the current element.

The second step is to evaluate the h-candidates, which is very easy. The projected functions do not even have to be explicitly constructed anymore.

If (p_1, p_2, p_3, p_4) are the polynomial degrees of the four elements of some h-candidate, its error is simply

$$e_{(p_1,p_2,p_3,p_4)} = h_{p_1,1} + h_{p_2,2} + h_{p_3,3} + h_{p_4,4}.$$

In this way we can evaluate a practically unlimited number of h-refinement candidates, which is especially important in 3D, where thousands of combinations may exist. In 2D, this algorithm is about 40 times faster than the algorithm described in Section 4.4.

Anisotropic h-refinements are handled as additional h-refinement candidates. This is both simpler and more robust than checking the shape of the error up front as in the algorithm by Demkowicz. In our implementation anisotropic refinements compete naturally with all other isotropic refinements. This is affordable thanks to the fast evaluation of projection errors, all that is needed is to extend the array of the partial projection errors:

$$\begin{aligned}
h_{i,5} &= ||u_{ref} - \Pi_i^{\mathrm{OG}}\, u_{ref}||_{K_{q_5}}, \qquad K_{q_5} = (-1,1)\times(-1,0),\\
h_{i,6} &= ||u_{ref} - \Pi_i^{\mathrm{OG}}\, u_{ref}||_{K_{q_6}}, \qquad K_{q_6} = (-1,1)\times(0,1),\\
h_{i,7} &= ||u_{ref} - \Pi_i^{\mathrm{OG}}\, u_{ref}||_{K_{q_7}}, \qquad K_{q_7} = (-1,0)\times(-1,1),\\
h_{i,8} &= ||u_{ref} - \Pi_i^{\mathrm{OG}}\, u_{ref}||_{K_{q_8}}, \qquad K_{q_8} = (0,1)\times(-1,1).
\end{aligned}$$

4.6 Benchmarks

In this section we compare our slow projection algorithm with the fast orthogonal version described in the previous section, to verify the validity of the fast approach. We also compare both of them with Dekmowicz's algorithm, as well as with several most successful algorithms from [35] that are not based on the reference solution. The benchmarks used are two synthetic elliptic problems that are often used for this purpose.

4.6.1 L-shaped Domain Problem

The first benchmark is a prototypical elliptic problem invented by Babuška, whose solution exhibits a singularity at the reentrant corner of an L-shaped domain (see Figure 4.4). The exact solution in polar coordinates is

$$u(r,\theta) = r^{2/3}\sin(2\theta/3).$$

The same function is used to prescribe Dirichlet boundary conditions on $\partial\Omega$. The equation to be solved is the Laplace equation $\Delta u = 0$. The domain is defined as $\Omega = (-1,1)^2 \setminus ((0,1)\times(-1,0))$.

We performed two calculations with different initial meshes. The first initial mesh consisted of six triangular elements of degree $p = 2$, the second initial

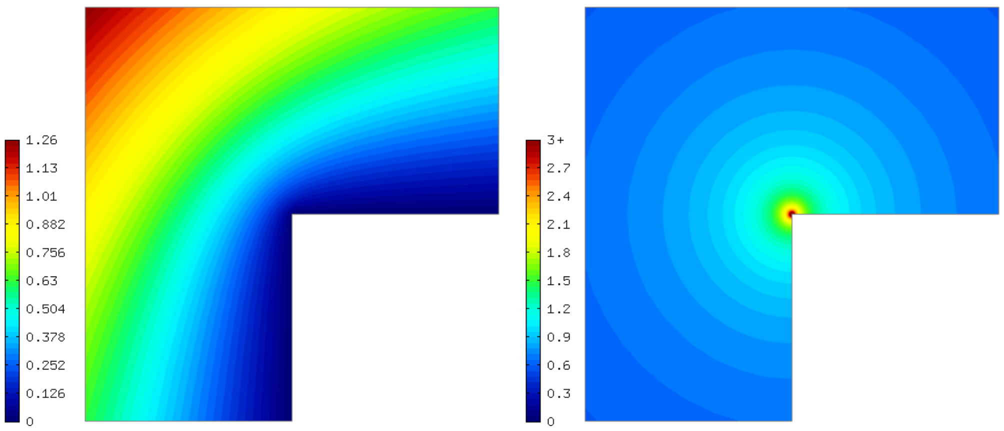


Figure 4.4: Exact solution of the L-shaped domain problem (left) and the magnitude of its gradient (right, singularity truncated).

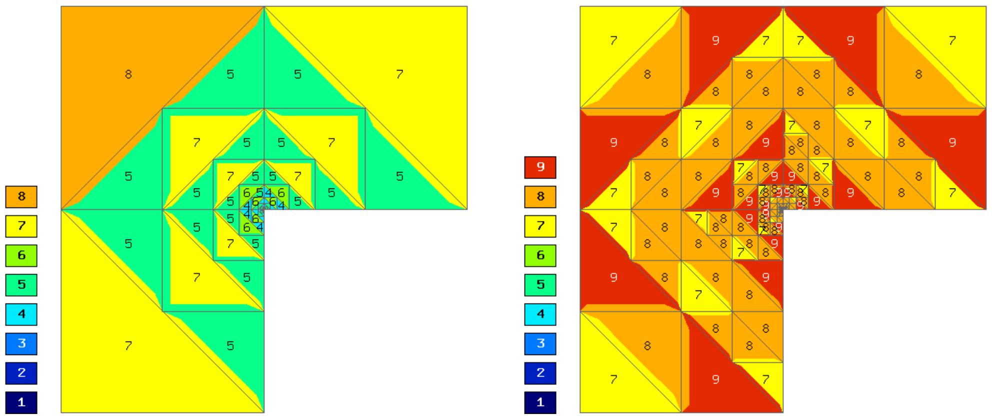


Figure 4.5: Triangular meshes with 971 (left) and 8359 (right) DOFs.

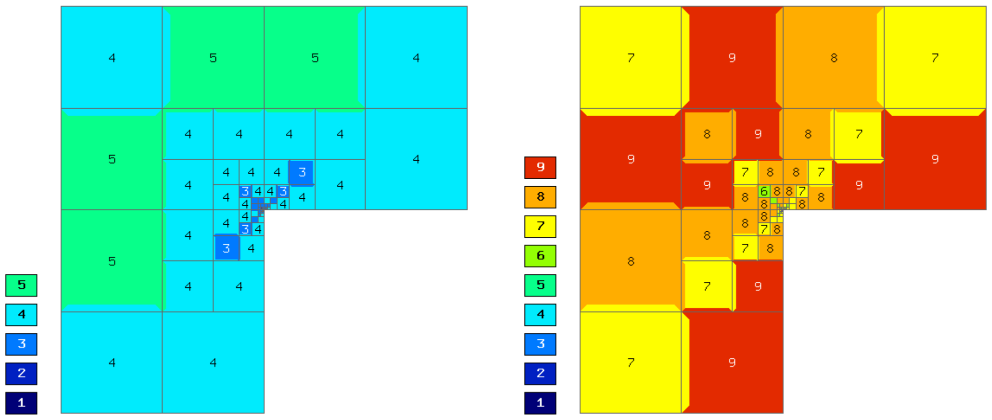


Figure 4.6: Quadrilateral meshes with 1026 (left) and 6778 (right) DOFs.

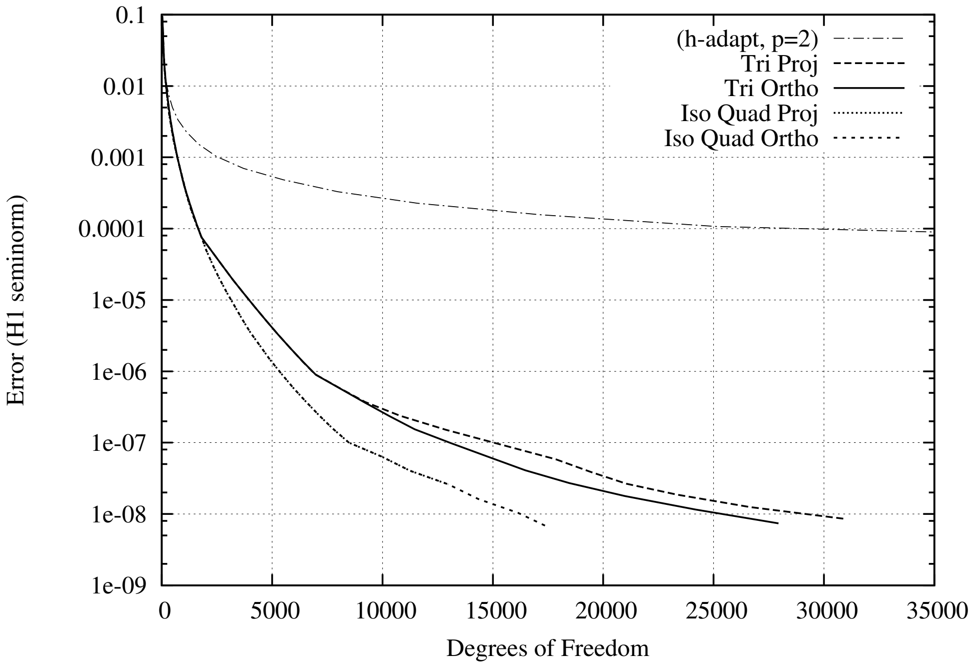


Figure 4.7: Convergence of our methods for the L-shaped domain problem.

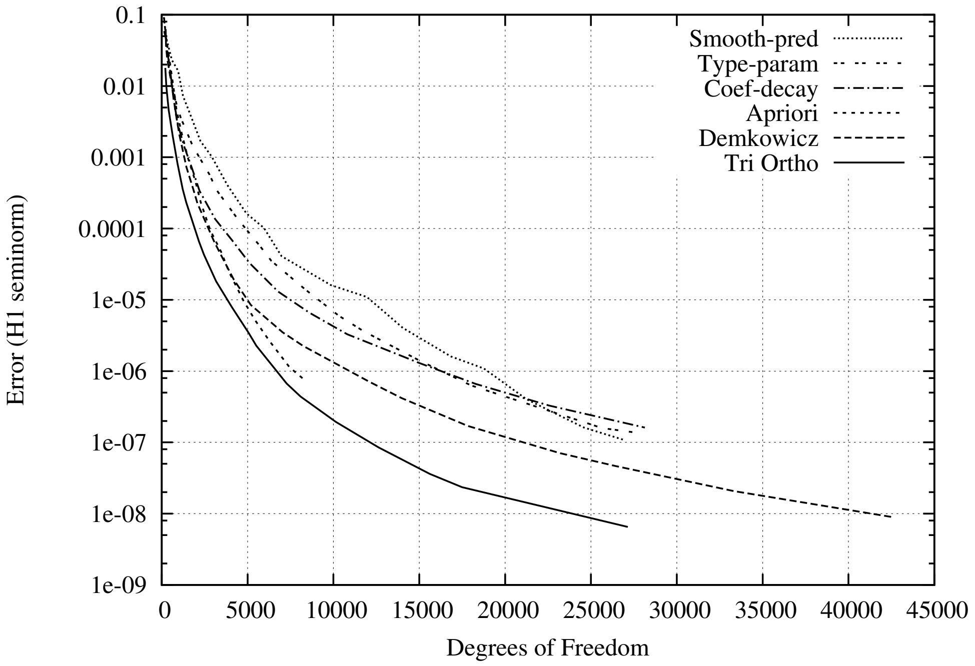


Figure 4.8: Various algorithms on the L-shaped domain problem.

mesh contained three quadrilaterals also of degree $p = 2$. Figure 4.5 shows the automatically obtained triangular meshes at two selected steps of the *hp*-adaptation process. Figure 4.6 shows results from the second computation with quadrilateral elements, at two iterations with roughly the same number of DOFs as in the first computation. In both cases the *hp*-adaptive strategy correctly selects large high-order elements for smooth parts of the solution, whereas towards the singularity smaller and lower-order elements are used.

In addition, each of the two computations was performed with our standard *hp*-adaptive algorithm (described in Section 4.4) and with its fast version with orthonormal bases. The convergence curves for the four computations are shown in Figure 4.7. All *hp* calculations achieved very good, almost exponential convergence rates – in later stages of the convergence, we can see that to reduce the error by one order of magnitude, *hp*-FEM only needs to add a similar number of degrees of freedom that were needed to cross the previous order of magnitude of the error. The reason why the curves are not completely straight (as they ought to be if the convergence was really exponential) is that we are limited in the maximum polynomial degree of the elements to $p = 9$ ($p = 10$ is reserved for the reference solution).

In case of quadrilaterals the fast algorithm (marked Ortho) produced almost exactly the same convergence curve as the standard algorithm (marked Proj). In the computation on triangles the fast version was even better.

For comparison, we have included a fifth computation, marked *h*-adapt in Figure 4.7, which is a traditional *h*-adaptation on quadratic elements. The difference between *h*-FEM and *hp*-FEM is obvious: the former has no chance of reaching the error levels of the *hp*-FEM computations using a reasonable number of DOFs.

Figure 4.8 compares the case Tri Ortho from the previous graph with the Demkowicz algorithm and the Smoothness prediction and Apriori knowledge approaches described in Section 4.2. At all error levels our algorithm produced meshes with fewer degrees of freedom than the other strategies.

In all cases the approximation error was measured against the exact solution in H^1 seminorm and integrated with the highest available integration rule (degree 20). The H^1 seminorm also happens to be the energy norm for the problem and was used to guide the adaptive strategy.

4.6.2 Inner Layer Problem

The second benchmark problem, taken from [35], is the Poisson equation $-\Delta u = f$ on the unit square $\Omega = (0, 1)^2$ with the right-hand side f and Dirichlet boundary conditions chosen in such a way that the exact solution is

$$u(x, z) = \tan^{-1}\left(\alpha\left(\sqrt{(x - x_c)^2 + (y - y_c)^2} - r_0\right)\right).$$

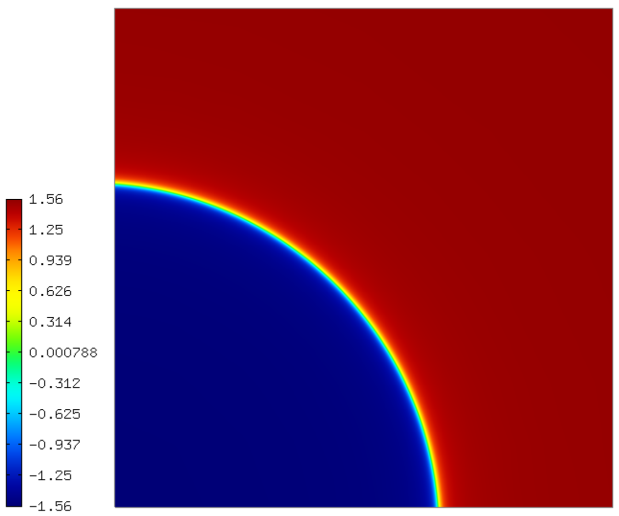


Figure 4.9: Exact solution of the inner layer problem.

The solution has a sharp circular wave front of radius r_0 centered at (x_c, y_c), as shown in Figure 4.9. The constant α determines the slope of the wave front. A similar problem appeared already in [18] with a smoother wave front but here we use the more difficult setting from [35]:

$$\alpha = 200, \; (x_c, y_c) = (-0.05, -0.05), \; r_0 = 0.7.$$

Again we performed several versions of the computation. The first computation starts on a mesh consisting of eight triangular elements with degree $p = 2$. The second starts with a single quadrilateral (again $p = 2$) however only isotropic h-refinements were allowed. The third computation starts with the same quadrilateral mesh but with anisotropic refinements turned on. Snapshots of two selected iterations from each of three computations are shown in Figures 4.10– 4.12.

The six convergence curves are compared in Figure 4.13. Again the curves are nearly exponential in later (asymptotic) stages of the convergence and by a large margin better than simple h-adaptivity. Calculations on quadrilaterals were better than on triangles, especially when anisotropic h-refinements were allowed, which confirms the correctness of our approach with anisotropic refinement candidates. Interestingly, the fast algorithm (Ortho) seems to have better properties even though it should in theory be worse than the slow but "correct" algorithm (Proj). Our hypothesis is that the fast version tends to prefer h-refinements, which may be beneficial for this particular problem (but not necessarily for other problems).

Figure 4.14 compares our computation Tri Ortho with the Demkowicz algorithm and other rival algorithms. This time our result is merely comparable

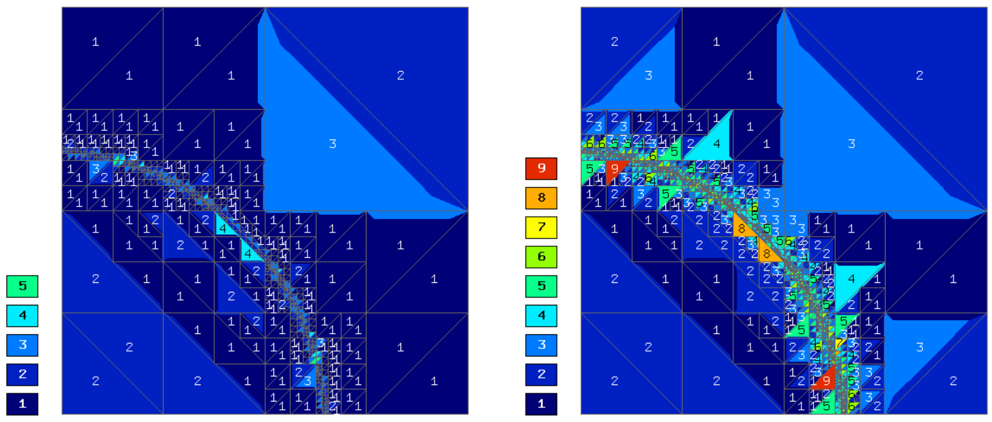


Figure 4.10: Triangular meshes with 1148 (left) and 9075 (right) DOFs.

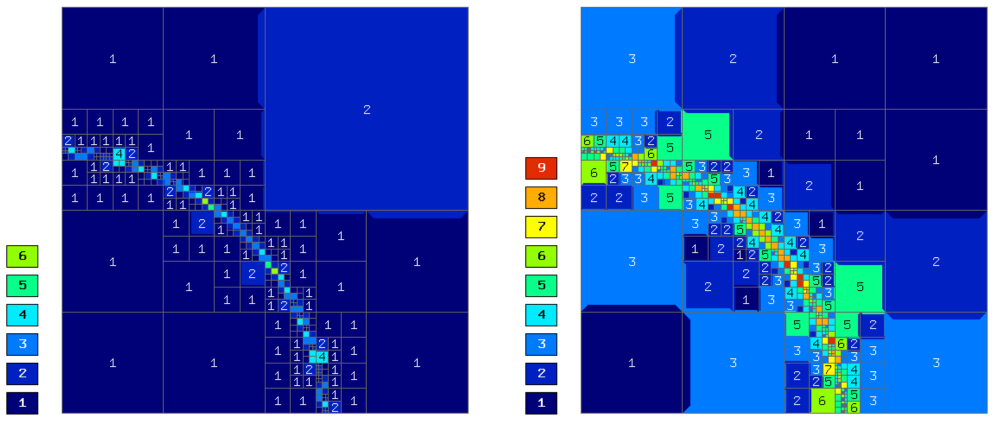


Figure 4.11: Isotropic quad meshes with 903 (left) and 9274 (right) DOFs.

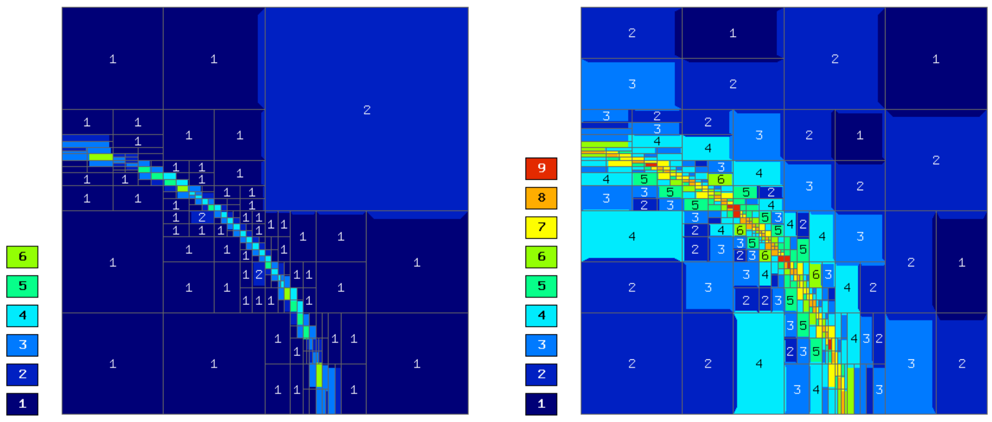


Figure 4.12: Anisotropic quad meshes with 980 (left) and 9389 (right) DOFs.

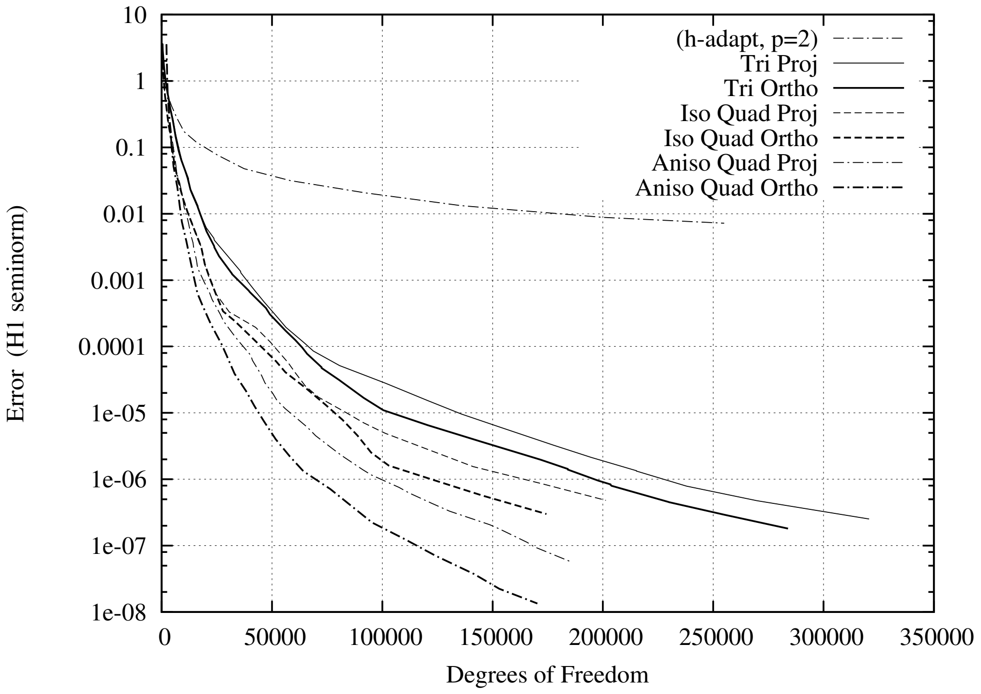


Figure 4.13: Convergence of our methods for the inner layer problem.

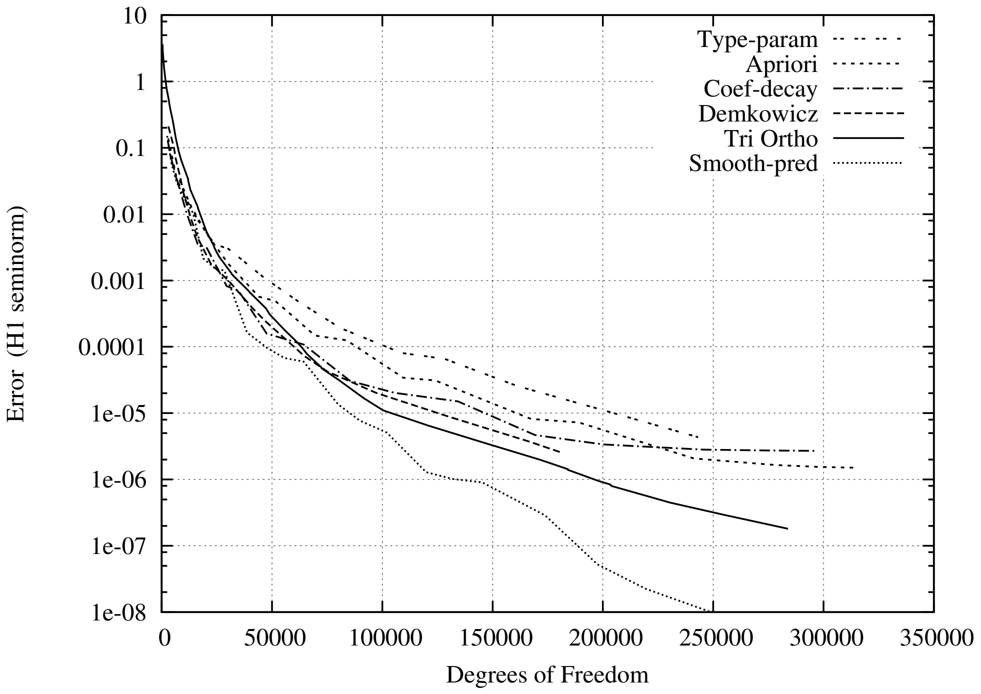


Figure 4.14: Various algorithms on the inner layer problem.

with Demkowicz's convergence. It is remarkable however, that one of the algorithms not based on reference solution, the Smoothness prediction, was performing extremely well on this problem. We have no explanation for this, as the reference solution-based algorithms should be much better since they have more information available. The graph is based on data from [35], kindly provided by Dr. William F. Mitchell of NIST. In all cases the error is measured in the energy norm of the problem, which coincides with the H^1 seminorm.

4.7 Conclusion and Future Work

We have shown that our *hp*-adaptive strategy is very competitive, which we consider a success given the relative simplicity of the approach, especially of the "fast" version of the algorithm. The biggest weakness of the method is currently the time needed to assemble and solve the reference solution, because at every iteration this has to be done again from scratch. Figure 4.15 shows the total CPU time needed to reach an error level in the L-shaped domain problem. The (fast) adaptation itself takes very little time but the reference solution takes about 90% of the total time. Out of that, matrix assembly and UMFPACK solution take about the same time, however the solution part tends to take much more time in bigger problems or when an iterative solver is used.

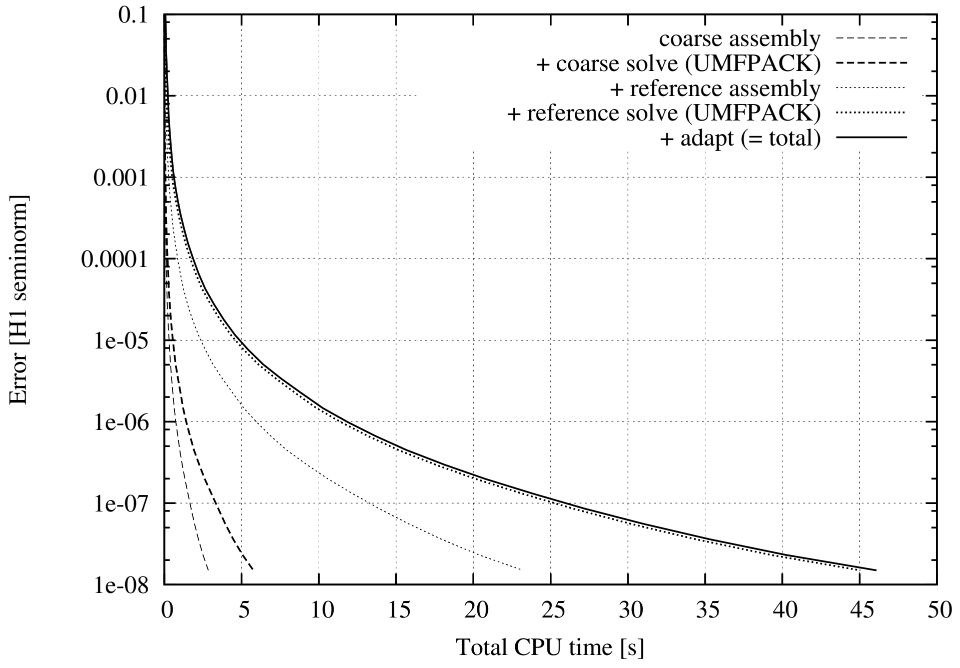


Figure 4.15: CPU time breakdown for the L-shaped domain problem.

An obvious way to speed up the solution of the reference problem u^n_{ref} is to try to reuse the previous reference solution u^{n-1}_{ref} (we stress that previous *reference* solution has to be used, not the current coarse solution, which does not contain enough information). This is conceivable with iterative solvers, where the previous solution vector (after some modifications to account for new DOFs) could be used as the initial approximation. Moreover, only partially converged reference solutions may be usable to guide the adaptation [16].

Direct solvers are more important from our point of view, because of their larger applicability (non-SPD problems in, e.g., fluid flow). Our group has been studying the possibility of extending the basis functions in the *hp*-mesh hierarchically during both *h*- and *p*-refinements [44]. In such case any refinements would only result in new rows and columns in the stiffness matrix, and a specialized direct solver based on LU decomposition could be written to reuse (update) the previous LU decompositions. However such approach presents considerable implementation challenges and we have not yet been able to realize it.

Although the assembling performance is not as big an issue as solution performance, there is definitely room for improvement as well. High-order elements tend to be expensive to assemble and any technique to optimize the integration of the weak forms is important. Precalculating some terms of the forms before the reference transformation is applied has very good results, however this method does not combine well with forms where non-constant coefficients occur. We also currently do not reuse local matrices from previous adaptive iterations, partly because of multi-mesh assembling (see following chapter). In short, optimizations are possible, but at the cost of making the implementation much more complex.

Finally, one thing that we did not discuss which the fast adaptation algorithm still needs to be extended for is a support for anisotropic *p*-refinements on quadrilaterals. Every higher-order quadrilateral can be equipped with a different polynomial degree in the ξ_1 and ξ_2 directions of the reference domain. In the slow version this is easily handled by just including more *p*-refinement candidates, but in the orthonormalized shape function set this would break the degree hierarchy. The solution is to replace the list of N orthonormal functions ordered by the isotropic degree by a matrix of $N \times N$ functions where each row would represent a new set for a fixed anisotropic degree in ξ_1 and the rest of the row would be ordered by the ξ_2 degree. This is an implementation complication that we have not had the time to handle so far.

Chapter 5

Systems of PDEs and Multi-Mesh Assembling

So far we have only been concerned with the solution of a single PDE. This chapter describes the solution of systems of linear elliptic PDEs and introduces our multi-mesh method, which goes one step further by allowing individual components in the system to be equipped with different meshes.

5.1 Introduction

As we will see in the following section, it is relatively easy to extend a single-equation finite element solver to support systems of PDEs. In the traditional approach, all equations in the system must share the same mesh. However, when performing h-adaptivity, some components (equations) of the system may require certain elements to be refined while other components may require different or no refinements at all. Typically, an element of the mesh is refined when at least one component's error indicator demands the refinement. This may however lead to an unnecessary increase in the number of DOFs because in other components the refinement is forced.

For example, consider a problem with an electric field and an x and y velocity equations. The optimal meshes for these components might be the meshes (a), (b), (c), respectively, in Figure 5.1. In practice, one is usually forced to use mesh (d) for all components, as it contains the union of all refinements.

It is therefore quite natural to require that each equation in the system can be refined independently of all others. Multi-mesh approaches similar to ours were developed by Schmidt and Ruo Li already in [42, 38, 39, 40] in the context of phase-field models of solidification, where the temperature component is much smoother than the second component containing the material phase interface. Very recently, Voigt [54] presented a study which unlike the previous

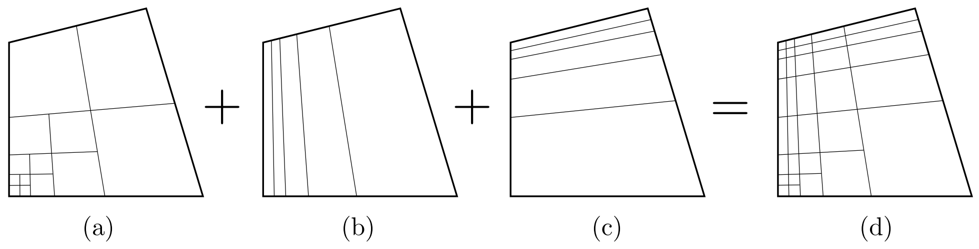


Figure 5.1: Three different example meshes and their union.

publications contains also a detailed derivation of the method and implementation details, although only in a simpler context of constant degree (non-hp) higher-order FEM with simplicial elements. Our approach that we published first in [24] and which is described in this chapter produces similar stiffness matrices but is more general, since it allows anisotropic refinements and the coupling of components with arbitrary polynomial degrees.

It should be noted that the multi-mesh method is not identical to operator splitting, which is often the method of choice in multi-physics scenarios where it is necessary to couple different finite element codes. Usually, this requires the transfer of data from one mesh to another by interpolation, which may cause degradation of the quality of the computation. In contrast, our method is able to discretize systems of independent meshes that can even support finite elements of different types (H^1, $\boldsymbol{H}(\mathrm{curl})$, L^2). As a result, the coupling is fully monolithic, with no need for operator splitting. The only limitation of our approach is that all meshes must be derived from one common predecessor, called the master mesh. This requirement is usually easily met in practice.

In this chapter we first derive the weak formulation of a system of linear elliptic PDEs and show how the element-by-element assembling procedure needs to be modified to handle such systems. We then introduce further modifications that allow us to assemble multi-mesh systems.

5.2 Discretization of Systems of PDEs

Consider a system of r linear PDEs. The unknown weak solution $\boldsymbol{u}$ has r components that belong to different function spaces $V_1, V_2, \ldots, V_r$,

$$\boldsymbol{u} = (u_1, u_2, \ldots, u_r)^T \in W = V_1 \times V_2 \times \ldots \times V_r.$$

The weak formulation for this problem has the form

$$\mathcal{A}(\boldsymbol{u}, \boldsymbol{v}) = \mathcal{L}(\boldsymbol{v})$$

where

$$\mathcal{A}(\boldsymbol{u}, \boldsymbol{v}) \;\; = \;\; (A^{(1)},\, A^{(2)},\, \ldots,\, A^{(r)})^T(\boldsymbol{u}, \boldsymbol{v}),$$

$$\mathcal{L}(\boldsymbol{v}) \;=\; (L^{(1)},\, L^{(2)},\, \ldots,\, L^{(r)})^T(\boldsymbol{v}).$$

The test functions are also vector-valued,

$$\boldsymbol{v} \;=\; (v^{(1)}, v^{(2)}, \ldots, v^{(r)})^T \in W.$$

A sufficiently general expression for the multilinear forms $A^{(m)} : W \times W \to R$ and $L^{(m)} : W \to R,\ 1 \le m \le r,$ can be written as

$$\begin{aligned} A^{(m)}(\boldsymbol{u}, \boldsymbol{v}) &= \sum_{n=1}^{r}\sum_{k=1}^{r} a^{(mnk)}(u^{(n)}, v^{(k)}), && (5.1)\\ L^{(m)}(\boldsymbol{v}) &= \sum_{k=1}^{r} l^{\,(mk)}(v^{(k)}). && (5.2)\end{aligned}$$

However, in practice the test functions can always be chosen to have only one nonzero component:

$$\begin{aligned} \boldsymbol{v}_i &= (v_i^{(1)},\, 0,\, 0,\, \ldots,\, 0)^T, & v_i^{(1)} \in V_1,\ \ 0 \le i < N_1,\\ \boldsymbol{v}_{N_1+i} &= (0,\, v_i^{(2)},\, 0,\, \ldots,\, 0)^T, & v_i^{(2)} \in V_2,\ \ 0 \le i < N_2,\\ &\ \,\vdots & \\ \boldsymbol{v}_{N_1+\ldots+N_{r-1}+i} &= (0,\, 0,\, 0,\, \ldots,\, v_i^{(r)})^T, & v_i^{(r)} \in V_r,\ \ 0 \le i < N_r, \end{aligned}$$

where $N_m = \dim(V_m)$. Then we can denote $a^{(mn)} \equiv a^{(mnm)}$, $l^{(m)} \equiv l^{(mm)}$, and the equations (5.1) and (5.2) simplify to

$$\begin{aligned} A^{(m)}(\boldsymbol{u},\, (0,\ldots,0,v^{(m)},0,\ldots,0)^T) &= \sum_{n=1}^{r} a^{(mn)}(u^{(n)}, v^{(m)}), && (5.3)\\ L^{(m)}((0,\ldots,0,v^{(m)},0,\ldots,0)^T) &= l^{\,(m)}(v^{(m)}). && \end{aligned}$$

From (5.3) it is apparent that the stiffness matrix $\boldsymbol{S}$ and the load vector $\boldsymbol{F}$ now have the block form

$$\mathcal{S} = \begin{pmatrix} \boldsymbol{S}^{(11)} & \boldsymbol{S}^{(12)} & \cdots & \boldsymbol{S}^{(1r)}\\ \boldsymbol{S}^{(21)} & \boldsymbol{S}^{(22)} & \cdots & \boldsymbol{S}^{(2r)}\\ \vdots & \vdots & \ddots & \vdots\\ \boldsymbol{S}^{(r1)} & \boldsymbol{S}^{(r2)} & \cdots & \boldsymbol{S}^{(rr)} \end{pmatrix}, \quad \mathcal{F} = \begin{pmatrix} \boldsymbol{F}^{(1)}\\ \boldsymbol{F}^{(2)}\\ \vdots\\ \boldsymbol{F}^{(r)} \end{pmatrix},$$

where

$$\begin{aligned} \boldsymbol{S}_{ij}^{(mn)} &= a^{(mn)}(u_j^{(n)},\, v_i^{(m)}), & 0 \le i < N_m,\\ \boldsymbol{F}_i^{(m)} &= l^{\,(m)}(v_i^{(m)}), & 0 \le j < N_n. \end{aligned}$$

Recall the single-PDE assembling procedure, Algorithm 2.1 on page 18. For a system of PDEs, we need to modify the procedure by adding two cycles over the $r \times r$ equation blocks. This is shown in Algorithm 5.1, which represents the standard element-by-element assembling procedure for systems of PDEs.

Algorithm 5.1: Element-by-element assembling procedure for a system of r PDEs.

```
for $k = 1 \ldots M$ do
    for $m = 1 \ldots r$ do
        $L_m$ = list of basis fn. indices in space $m$ whose $\mathrm{supp}(v_i) \cap K_k \neq \emptyset$
    for $m = 1 \ldots r$ do
        for $n = 1 \ldots r$ do
            foreach $i \in L_m$ do
                foreach $j \in L_n$ do
                    $\boldsymbol{S}_{ij} = \boldsymbol{S}_{ij} + a_{mn}(v_j, v_i)|K_k$
```

5.3 Assembling on Multiple Meshes

Algorithm 5.1 assumes that all equations in the system share the same mesh containing elements $K_1 \ldots K_M$. This is however just a convenience, as the Galerkin method requires no such thing. In fact, the Galerkin method is only concerned with basis functions and does not require any mesh at all (as is the case in meshless methods). We can thus relax the assumption and only require that all meshes in the system are derived from a single, very coarse predecessor, which we call the *master mesh.* In our example in Figure 5.1, the master mesh would consist of a single quadrilateral element, which can however be refined independently in each of the three components (a) – (c).

To assemble the stiffness matrix of such a multi-mesh system, we no longer loop over the physical elements but instead we traverse the *union mesh* shown in Figure 5.2 (d) and its (virtual) elements $Q_1 \ldots Q_{M'}$. In doing so we still cover the whole domain Ω so all integrals are evaluated correctly, only the integration may be done in more steps and also in some components we need to restrict the integration and evaluation of shape functions to sub-domains of the standard reference domain $(-1, 1)^2$. For example, when evaluating the weak forms on element Q_k, in component (a) in Figure 5.2 we need to integrate in $(0, 1) \times (-1, 0)$, in component (b) in $(-1, 1) \times (0, \frac{1}{2})$ and in component (c) in $(-\frac{1}{2}, 0) \times (-1, 1)$, see also Figure 5.3. These are areas of the reference domain

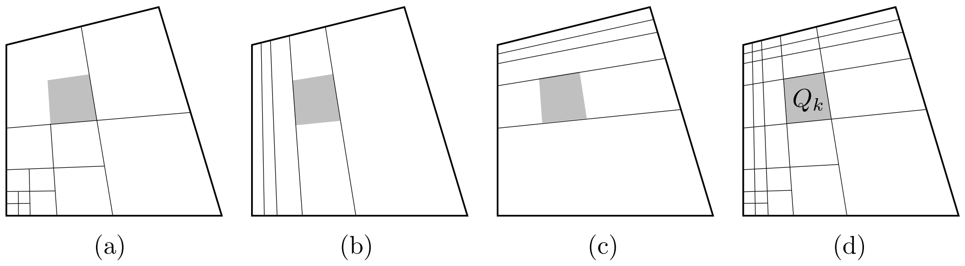


Figure 5.2: Restriction to the union mesh element Q_k.

Algorithm 5.2: Multi-mesh assembling procedure for a system of PDEs.

```
foreach union mesh element $Q_k$ do
    for $m = 1 \dots r$ do
        $L_m$ = list of basis fn. indices in space $m$ whose $\mathrm{supp}(v_i) \cap Q_k \neq \emptyset$
    for $m = 1 \dots r$ do
        for $n = 1 \dots r$ do
            foreach $i \in L_m$ do
                foreach $j \in L_n$ do
                    $\boldsymbol{S}_{ij} = \boldsymbol{S}_{ij} + a_{mn}(v_j, v_i)|Q_k$
```

which correspond to Q_k within the physical elements of meshes (a, b, c), or *sub-elements*. The multi-mesh assembling procedure is shown in Algorithm 5.2. Formally, there are not many modifications compared to Algorithm 5.1.

Of course, the union mesh does not have to be explicitly constructed at all. A simple recursive algorithm for the traversal of a virtual union mesh is described in Section 5.3.2. Also, it should be noted that assembling over different meshes is not more efficient than standard assembling over the real union mesh in all components. The time savings are expected in the solution of the resulting linear system, which should be smaller and possibly better conditioned.

5.3.1 Integration Over Sub-elements

In order to integrate the bilinear forms over the virtual element Q_k, we need to extend the affine concept from Section 2.4 (page 9). In Figure 5.2 we see that the shaded areas in each mesh correspond to those depicted in Figure 5.3 in the reference domains.

Since we are only able to integrate over the whole reference domain (where the integration points are defined), we need to introduce the mapping $\boldsymbol{r} : K_q \to K_q$, in addition to the standard reference mapping $\boldsymbol{x}_K : K_q \to K$. The situation is illustrated in Figure 5.4.

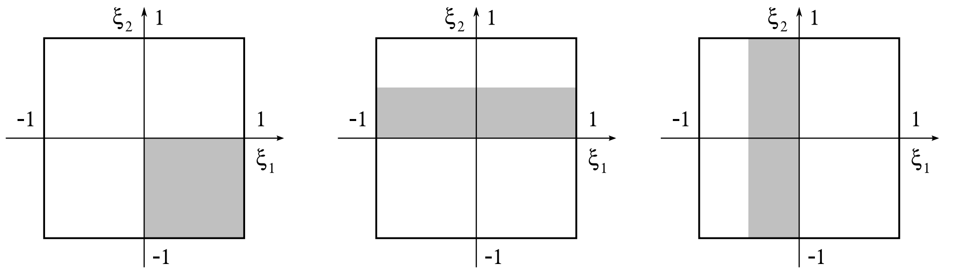


Figure 5.3: Areas of the ref. domain corresponding to Q_k in each component.

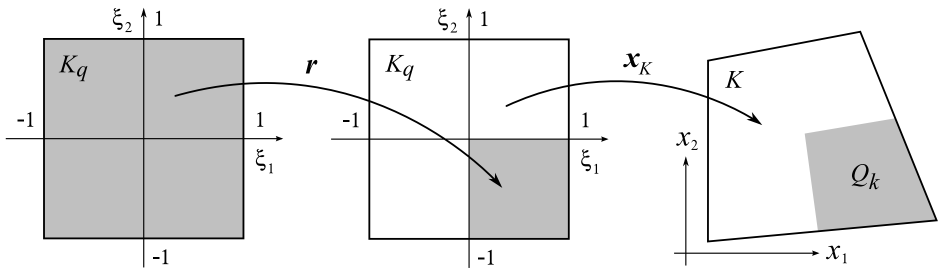


Figure 5.4: The mappings $\boldsymbol{r}$ and $\boldsymbol{x}_K$.

The mapping $\boldsymbol{r}$ is a simple affine transformation of the form

$$\boldsymbol{r}(\boldsymbol{\xi}) = \boldsymbol{R}\boldsymbol{\xi} + \boldsymbol{t}. \tag{5.4}$$

For example, in Figure 5.4 the mapping is

$$\boldsymbol{r}(\boldsymbol{\xi}) = \begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix} \boldsymbol{\xi} + \begin{pmatrix} 0.5 \\ -0.5 \end{pmatrix}$$

Taking now a model weak form and applying the procedure of Section 2.4, we have

$$a(v_j, v_i) = \int_{Q_k} \nabla v_j(\boldsymbol{x}) \cdot \nabla v_i(\boldsymbol{x}) + v_j(\boldsymbol{x}) v_i(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} =$$

$$= \int_{K_q} J \left[\left(\frac{D\boldsymbol{r}_j}{D\boldsymbol{\xi}} \right)^{-T} \left(\frac{D\boldsymbol{x}_{K_j}}{D\boldsymbol{\xi}} \right)^{-T} \nabla \tilde{v}_j(\boldsymbol{\xi}) \right] \cdot \left[\left(\frac{D\boldsymbol{r}_j}{D\boldsymbol{\xi}} \right)^{-T} \left(\frac{D\boldsymbol{x}_{K_i}}{D\boldsymbol{\xi}} \right)^{-T} \nabla \tilde{v}_i(\boldsymbol{\xi}) \right] \mathrm{d}\boldsymbol{\xi}$$

$$+ \int_{K_q} J \, \tilde{v}_j(\boldsymbol{\xi}) \tilde{v}_i(\boldsymbol{\xi}) \, \mathrm{d}\boldsymbol{\xi},$$

where

$$\tilde{v}_i(\boldsymbol{\xi}) = (v_i \circ \boldsymbol{x}_{K_i} \circ \boldsymbol{r}_i)(\boldsymbol{\xi}),$$

$$\tilde{v}_j(\boldsymbol{\xi}) = (v_j \circ \boldsymbol{x}_{K_j} \circ \boldsymbol{r}_j)(\boldsymbol{\xi}),$$

$$J = \det \left(\frac{D\boldsymbol{x}_{K_i}}{D\boldsymbol{\xi}} \right) \det \left(\frac{D\boldsymbol{r}_i}{D\boldsymbol{\xi}} \right) = \det \left(\frac{D\boldsymbol{x}_{K_j}}{D\boldsymbol{\xi}} \right) \det \left(\frac{D\boldsymbol{r}_j}{D\boldsymbol{\xi}} \right).$$

The extended reference mapping itself should not hinder the performance of the code, as the Jacobian and the inverse Jacobi matrices can be precalculated as usual. The only concern might be the need to precalculate the shape functions at the transformed integration points, as discussed in Section 5.3.3.

5.3.2 Union Mesh Traversal

As mentioned above, the union mesh is never constructed explicitly. We only need to enumerate all its elements Q_k, together with the appropriate sub-element transformations $\boldsymbol{r}$ for each mesh. Our algorithm relies on the fact that all component meshes are obtained by refining a single common master mesh. This allows us to describe all possible transformations $\boldsymbol{r}$ as a composition of four predefined basic transformations on triangles and eight basic transformations on quadrilaterals, as shown in Figure 5.5. Suppose $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$ are transformations of the form (5.4). Their composition is simply

$$(\boldsymbol{r}_2 \circ \boldsymbol{r}_1)(\boldsymbol{\xi}) = \boldsymbol{R}_2(\boldsymbol{R}_1\boldsymbol{\xi} + \boldsymbol{t}_1) + \boldsymbol{t}_2 = (\boldsymbol{R}_2\boldsymbol{R}_1)\boldsymbol{\xi} + (\boldsymbol{R}_2\boldsymbol{t}_1 + \boldsymbol{t}_2)$$

In the implementation we often need to identify a concrete transformation by a simple integer code, for example to be able to store (cache) precalculated shape function values in some transformed integration points. The basic sub-element transformations are numbered as in Figure 5.5. If $\boldsymbol{r}_2$ is a basic transformation then the composition $\boldsymbol{r}_2 \circ \boldsymbol{r}_1$ is assigned a code in the following way:

$$\mathrm{code}(\boldsymbol{r}_2 \circ \boldsymbol{r}_1) = 8 * \mathrm{code}(\boldsymbol{r}_1) + \mathrm{code}(\boldsymbol{r}_2) + 1.$$

For example, the transformation from the reference quadrilateral $(-1,-1)^2$ to the area $(0, \frac{1}{2})^2$ formed by composing $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$, where $\mathrm{code}(\boldsymbol{r}_1) = 2$ and $\mathrm{code}(\boldsymbol{r}_2) = 0$, has code 17.

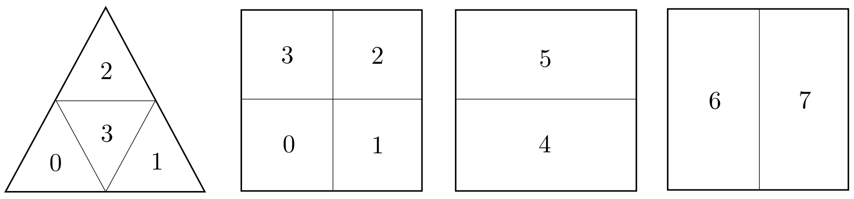


Figure 5.5: Numbering of predefined sub-element transformations.

Before describing the union mesh traversal algorithm, let us demonstrate the process on a simple example. In Figure 5.6 we have two meshes that were refined independently from a common predecessor, a single triangular element. The refinement trees (see Section 3.3.3) and element `id` numbers are also shown. The union mesh (only visible as dotted lines in Figure 5.6) has ten elements, $Q_1 \dots Q_{10}$. The traversal begins with element 0 in both meshes, but this element is refined in both of them so the procedure goes down one level to element 1. This element is active in both meshes so it can be assembled as usual, without any transformation of integration points. Next is element 2, which is refined in mesh (a) but active in mesh (b). The algorithm needs to go deeper in mesh (a) but this is not possible in mesh (b) and as a compensation we descend to a sub-element using a transformation. In Table 5.1 we can see that while the algorithm is visiting elements 5–8 in mesh (a), the current element is still 2 in mesh (b) with transformations varying from 1 to 4. A similar, but reversed situation occurs with element 3 in mesh (a) and elements 5–8 in mesh (b). Finally, element 4 is active in both meshes and can be assembled normally.

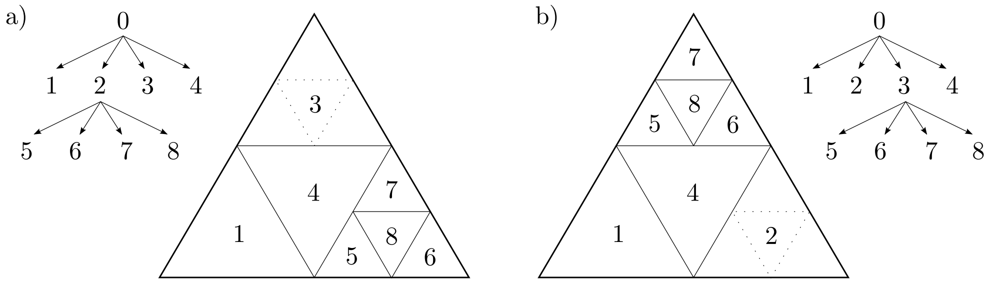


Figure 5.6: Two triangular meshes and their refinement trees.

Q_k	K_1	code($\boldsymbol{r}_1$)	K_2	code($\boldsymbol{r}_2$)
1	1	0	1	0
2	5	0	2	1
3	6	0	2	2
4	7	0	2	3
5	8	0	2	4
6	3	1	5	0
7	3	2	6	0
8	3	3	7	0
9	3	4	8	0
10	4	0	4	0

Table 5.1: Example of union mesh traversal.

The traversal algorithm begins with a main loop visiting each element of the master mesh. For these elements it invokes one of the two recursive procedures which traverse the refinement trees in all component meshes. For simplicity, let us assume we only have two different component meshes. For triangles, the recursive procedure is called `traverse_tri` and its pseudocode is shown on page 71. As it is apparent from the above example, the main idea is as follows: if it is not possible to go deeper in the element hierarchy in a certain component, its current transformation (which is initially an identity) is replaced with a composition of itself and one of the predefined sub-element transformations.

A similar recursive function is invoked in case the current master mesh element is a quadrilateral. Its principle is identical to `traverse_tri`, but the implementation is more complex due to anisotropic splits. The encoding of transformations and their compositions is not unique for quadrilaterals, because

Procedure `traverse_tri`(K_1, $\boldsymbol{r}_1$, K_2, $\boldsymbol{r}_2$)

```
if active($K_1$) && active($K_2$) then
    assemble($K_1$, $K_2$, $\boldsymbol{r}_1$, $\boldsymbol{r}_2$)
else
    for $s = 0 \dots 3$ do // loop over the four son elements
        for $i = 1 \dots 2$ do
            if active($K_i$) then // i.e., $K_i$ has no sons
                $K_i^{new} = K_i$;
                $\boldsymbol{r}_i^{new} = \boldsymbol{r}_i \circ$ predef[$s$];
            else
                $K_i^{new} = K_i$->sons[$s$];
                $\boldsymbol{r}_i^{new} = \boldsymbol{r}_i$;
        traverse_tri($K_1^{new}$, $\boldsymbol{r}_1^{new}$, $K_2^{new}$, $\boldsymbol{r}_2^{new}$);
```

Procedure `traverse_quad`(K_1, $rect_1$, K_2, $rect_2$, cur)

```
if active($K_1$) && active($K_2$) then
    $\boldsymbol{r}_1$ = get_transform($cur$, $rect_1$);
    $\boldsymbol{r}_2$ = get_transform($cur$, $rect_2$);
    assemble($K_1$, $K_2$, $\boldsymbol{r}_1$, $\boldsymbol{r}_2$)
else
    if $cur$ is split both ways by $K_1$ or $K_2$ then
        for $s = 0 \dots 3$ do // loop over the four son elements
            $cur^{new}$ = move_to_son(s, $cur$);
            for $i = 1 \dots 2$ do
                if active($K_i$) then // i.e., $K_i$ has no sons
                    $K_i^{new} = K_i$;
                    $rect_i^{new}$ = move_to_son(s, $rect_i$)
                else
                    $K_i^{new} = K_i$->sons[$s$];
                    $rect_i^{new} = rect_i$
            traverse_quad($K_1^{new}$, $rect_1^{new}$, $K_2^{new}$, $rect_2^{new}$, $cur^{new}$);

    else // $cur$ is split anisotropically by $K_1$ or $K_2$
        for $s = 0 \dots 1$ do // loop just over the two son elements
            ...;
            // analogous to the above;
            ...;
            traverse_quad($K_1^{new}$, $rect_1^{new}$, $K_2^{new}$, $rect_2^{new}$, $cur^{new}$);
```

successive anisotropic splits may give the same transformation as an isotropic split, but in our encoding scheme these transformations will have different numbers (codes). This is why the current state of traversal in `traverse_quad` is based on rectangles within the reference domain $(-1,1)^2$. The recursion is controlled by the way the current rectangle is being split by one or more of the meshes. When a union element is reached, the transformations are inferred from the relation of the current rectangle *cur* and the element rectangles $rect_i$ using the function `get_transform()`. The other auxiliary function, `move_to_son()`, adjusts the position of the specified rectangle according to the son number.

In the actual implementations we have replaced the recursion by a state stack, so that the assembling algorithm can still be written in the form given in the previous sections. In the outer loop, instead of iterating over physical elements, the assembling procedure calls the main traversal function which returns the next traversal state. The traversal state represents one of the elements of the virtual union mesh and consists of the `id` numbers of physical elements in the actual meshes along with the codes of their sub-element transforms.

5.3.3 Evaluation of Coupling Forms

Let us briefly describe the implementation of the core of the multi-mesh assembling procedure, which roughly corresponds to the two inner-most cycles in Algorithm 5.2. The interesting case is when $m \neq n$, that is, when weak forms coupling two different equations m and n need to be assembled, each defined on its own mesh, $\mathcal{T}_m$ and $\mathcal{T}_n$. Recall from Section 5.2 that the block $\boldsymbol{S}^{(mn)}$ of the stiffness matrix has the form

$$\boldsymbol{S}^{(mn)}_{ij} = a^{(mn)}(v_j,\, v_i), \qquad 0 \le i \le N_m,\, 0 \le j \le N_n,$$

where the bilinear form $a^{(mn)}(\cdot,\cdot)$ is the part of the weak formulation that contains the coupling terms and $v_i \in V_m$ and $v_j \in V_n$ are basis functions. The block $\boldsymbol{S}^{(mn)}$ is assembled from local element matrices $\boldsymbol{L}^{[rs]}$ corresponding to the elements Q_k of the union of meshes $\mathcal{T}_m$ and $\mathcal{T}_n$. For each virtual element Q_k the traversal procedure determines the sub-element transformations $\boldsymbol{r}(\boldsymbol{\xi})$, $\boldsymbol{s}(\boldsymbol{\xi})$, with codes r, s (see Section 5.3.2). The local stiffness matrix for Q_k is then calculated as

$$\boldsymbol{L}^{[rs]}_{i'j'} = \tilde{a}^{(mn)}(\tilde{v}^s_j,\, \tilde{v}^r_i), \qquad 0 \le i' \le |L_m|,\; 0 \le j \le |L_n|,$$

where for simplicity we have denoted $\tilde{a}^{(mn)}(\cdot\,,\cdot)$ to be the bilinear form defined on the reference domain that includes all the necessary Jacobians and transformations of derivatives (see Section 2.4), such that

$$\tilde{a}^{(mn)}(\tilde{v}_{j'}(\boldsymbol{\xi}), \tilde{v}_{i'}(\boldsymbol{\xi})) = a^{(mn)}(v_j(\boldsymbol{x}), v_i(\boldsymbol{x})).$$

Here and in the previous, $\tilde{v}_i$, $\tilde{v}_j$ are shape functions defined on the reference domain and $\tilde{v}_i^r$, $\tilde{v}_j^s$ are their sub-element parts stretched again to the whole reference domain by the mappings $\boldsymbol{r}(\boldsymbol{\xi})$ and $\boldsymbol{s}(\boldsymbol{\xi})$:

$$\tilde{v}_i^r(\boldsymbol{\xi}) = \tilde{v}_i(\boldsymbol{r}(\boldsymbol{\xi})),$$
$$\tilde{v}_j^s(\boldsymbol{\xi}) = \tilde{v}_j(\boldsymbol{s}(\boldsymbol{\xi})).$$

In traditional FEM, the values of all shape functions $\tilde{v}_i$ are precalculated in integration points and stored in tables for fast use by the assembling procedure. In multi-mesh FEM this is also desirable but more complicated for the various r, s, and an efficient storage of the precalculated tables is crucial for the success of the multi-mesh method. Because there are many possible sub-element transformations $\boldsymbol{r}(\boldsymbol{\xi})$ the amount of RAM required becomes a concern. A full-blown cache needs to be implemented in which the tables are calculated on-demand and in which least-recently used items are discarded when running out of the alotted space. However letting the table cache grow too large may lead to slow fetching of the tables because the data set is so large that the CPU L2 cache becomes inefficient, which is what we observed in Hermes2D on problems featuring meshes with extreme differences in refinements. In hindsight, it might have been better to limit the number of precalculated tables to a fixed number of sub-element levels so that all tables fit in the L2 cache, and keep recalculating all finer levels during the assembling.

Voigt and Witkowski [54] present an alternative implementation which we have also been considering. Instead of caching values of shape function cut-outs for all r, s, they utilize the linearity of $\tilde{a}^{(mn)}(\cdot\,,\cdot)$ and construct and store matrices which transform the local matrix $\boldsymbol{L}$ calculated for standard (untransformed) shape functions $\tilde{v}_i$ into the desired matrix $\boldsymbol{L}^{[rs]}$. This is possible thanks to the fact that the shape functions $\tilde{v}_i$ form a basis of the appropriate polynomial space on the reference element and this means that the cut-outs $\tilde{v}_i^r(\boldsymbol{\xi})$ and $\tilde{v}_j^s(\boldsymbol{\xi})$ can be expressed as a linear combination of the functions $\tilde{v}_i$:

$$\tilde{v}_i^r(\boldsymbol{\xi}) = \sum_k \alpha_{ik}^r\, \tilde{v}_k(\boldsymbol{\xi}),$$
$$\tilde{v}_j^s(\boldsymbol{\xi}) = \sum_l \alpha_{jl}^s\, \tilde{v}_l(\boldsymbol{\xi}).$$

By linearity of $\tilde{a}^{(mn)}(\cdot\,,\cdot)$ we have

$$\tilde{a}^{(mn)}(\tilde{v}_j^s,\, \tilde{v}_i^r) = \tilde{a}^{(mn)}(\sum_l \alpha_{jl}^s\, \tilde{v}_l,\, \sum_k \alpha_{ik}^r\, \tilde{v}_k,) = \sum_k \sum_l \alpha_{jl}^s\, \tilde{a}^{(mn)}(\tilde{v}_l, \tilde{v}_k)\, \alpha_{ik}^r.$$

Denoting $\boldsymbol{A}_{ik}^{[r]} = \alpha_{ik}^r$, $\boldsymbol{A}_{jl}^{[s]} = \alpha_{jl}^s$ we can write in matrix form

$$\boldsymbol{L}^{[rs]} = \boldsymbol{A}^{[s]}\boldsymbol{L}\boldsymbol{A}^{[r]}.$$

The transformation matrices $\boldsymbol{A}^{[\cdot]}$ can be calculated on the fly during assembling but in practice they also need to be cached, as the authors admit in [54], which leads to similar problems that we observed.

5.3.4 Small Elements vs. High Order Elements

Apart from the L2 cache thrashing issue, which can be solved by careful implementation, we have encountered a more fundamental problem with multi-mesh assembling on *hp* meshes, related to the integration of weak forms. Consider a system of two equations, each solved on its own mesh, with integrals in the weak formulation similar to

$$\int_{Q_k} v_i^1 v_j^2 \, \mathrm{d}\boldsymbol{x},$$

where v_i^1 is a basis function on the first mesh and v_i^2 is a basis function on the other mesh (the weak form could also contain derivatives of v_i^1, v_i^2, this is not important here). Assume that the first mesh was refined many times and contains tiny elements of a low polynomial degree p. The second mesh consists of just one large element with a high polynomial degree q (for instance, let $p = 1$, $q = 8$). To integrate the product $v_i^1 v_j^2$, we need to use a quadrature rule of degree at least $p + q$. Because the multi-mesh assembling procedure operates on the (virtual) union mesh, there are many virtual elements Q_k where this high-degree quadrature rule needs to be used, which makes the assembling process more CPU expensive. In other words, due to the nature of the assembling algorithm, having one large high-order element in one mesh and many small elements in the other mesh is similar to having many high-order elements in the first mesh, a situation that should normally be avoided because high-order elements take a long time to assemble.

The problem can be partially side-stepped in the implementation by grouping the weak forms by the combinations of meshes which they require and by performing the assembling in several passes, thus minimizing the number of fragments Q_k each element needs to be broken to. For example, if the coupling terms where the above-mentioned problem arises only occur on the right-hand side of the weak formulation, the left-hand side terms can be assembled on a per-equation basis, without the multi-mesh approach. The same applies to diagonal left-hand side terms, which can always be assembled by iterating over the single corresponding mesh. The multi-mesh traversal can be applied just to the coupling terms where this is really necessary and such terms will be given their own assembling pass. This optimization is now an integral part of the assembling procedure of Hermes2D.

If there really are left-hand side terms coupling very different meshes, the problem can still be avoided by reformulating the equations and approximating one of the operands as a constant (e.q., as a solution from the previous time step), thus moving the term to the right-hand side. If this is not possible or desirable, the only way to avoid prolonged assembling time is to try to reduce the number q in the total quadrature degree $p + q$, because typically only a small portion of the high-order polynomial is being integrated, a portion that could be regarded as a lower-degree polynomial. We have performed an experiment which confirmed that sufficiently small sub-elements can be integrated with slightly reduced degree rules without losing too much precision, we were however not able to establish a general scheme which would guarantee error bounds on such approximation. This approach is thus not currently used in our numerical code.

5.4 Model Problem: Thermoelasticity

We demonstrate our multi-mesh implementation on a model problem of thermoelasticity, where three independent meshes are used for the three solution components: x and y displacement and temperature.

Governing Equations The plane-strain model of linear thermoelasticity inherits basic simplifying assumptions from the plane-strain elasticity model,

$$\varepsilon_{33} = \varepsilon_{13} = \varepsilon_{23} = 0. \tag{5.5}$$

Moreover, it assumes temperature-dependent strains in the form

$$\varepsilon_{ii} = \frac{\partial u_i}{\partial x_i} = \varepsilon_{ii,E} + \varepsilon_{ii,T} = \varepsilon_{ii,E} + \alpha(T - T_0), \quad 1 \le i \le 3 \tag{5.6}$$

(repeated indices do not imply Einstein summation). Here $\varepsilon_{ii,E}$, $\varepsilon_{ii,T}$, and ε_{ii} stand for the elastic, thermal, and total strains in the x_i-direction, respectively. By u_i we denote the displacement component in the x_i-direction, α is the thermal expansion coefficient, T the temperature, and T_0 a constant temperature corresponding to a stress-free initial configuration. The material is assumed to be isotropic. Recall that the stress component σ_{33} is nonzero in general.

Substituting assumptions (5.5), (5.6) into the basic stress-strain relation

$$\sigma_{ij} = \frac{E}{1+\nu}\varepsilon_{ij,E} + \frac{E\nu \sum_{k=1}^{3} \varepsilon_{kk,E}}{(1-2\nu)(1+\nu)}\delta_{ij}, \quad 1 \le i,j \le 3, \tag{5.7}$$

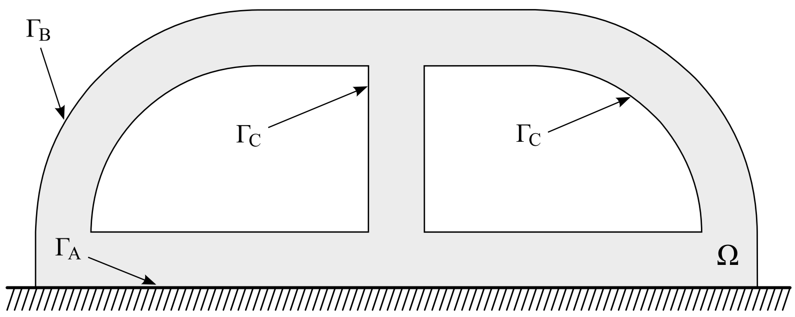


Figure 5.7: Transversal cross-section of the massive winding.

we obtain a system of equations for the stress components,

$$\left.\begin{aligned}
\sigma_{11} &= \frac{E}{(1-2\nu)(1+\nu)}\left[(1-\nu)\varepsilon_{11,E}+\nu(\varepsilon_{22,E}+\varepsilon_{33,E})\right] \\
&= \frac{E}{(1-2\nu)(1+\nu)}\left[(1-\nu)\frac{\partial u_1}{\partial x_1}+\nu\frac{\partial u_2}{\partial x_2}\right]-\frac{E\alpha(T-T_0)}{1-2\nu}\ , \\
\sigma_{22} &= \frac{E}{(1-2\nu)(1+\nu)}\left[(1-\nu)\varepsilon_{22,E}+\nu(\varepsilon_{11,E}+\varepsilon_{33,E})\right] \\
&= \frac{E}{(1-2\nu)(1+\nu)}\left[(1-\nu)\frac{\partial u_2}{\partial x_2}+\nu\frac{\partial u_1}{\partial x_1}\right]-\frac{E\alpha(T-T_0)}{1-2\nu}\ , \\
\sigma_{33} &= \frac{E}{(1-2\nu)(1+\nu)}\left[(1-\nu)\varepsilon_{33,E}+\nu(\varepsilon_{22,E}+\varepsilon_{11,E})\right], \\
&= \frac{E\nu}{(1-2\nu)(1+\nu)}\left(\frac{\partial u_1}{\partial x_1}+\frac{\partial u_2}{\partial x_2}\right)-\frac{E\alpha(T-T_0)}{1-2\nu}\ , \\
\sigma_{12} &= \frac{E}{2(1+\nu)}\varepsilon_{12,E}=\frac{E}{2(1+\nu)}\left(\frac{\partial u_1}{\partial x_2}+\frac{\partial u_2}{\partial x_1}\right).
\end{aligned}\right\} \quad (5.8)$$

Here, E and ν stand for the Young modulus and Poisson number, respectively. Substituting the stresses σ_1, σ_2, and σ_{12} from (5.8) into the equilibrium equations

$$\left.\begin{aligned}
\frac{\partial\sigma_{11}}{\partial x_1}+\frac{\partial\sigma_{12}}{\partial x_2}+f_1 &= 0, \\
\frac{\partial\sigma_{12}}{\partial x_1}+\frac{\partial\sigma_{22}}{\partial x_2}+f_2 &= 0, \\
\frac{\partial\sigma_{33}}{\partial x_3}+f_3 &= 0,
\end{aligned}\right\} \quad (5.9)$$

one obtains a system of second-order PDEs for the fields u_1, u_2 and T. In (5.9), the only nonzero component of the volume force is $f_2 = -\varrho g$. The symbols ϱ, g represent the material density and the gravitational constant, respectively. If all quantities are constant in the x_3-direction (as in our case), then the last equation in (5.9) is satisfied automatically.

In addition to the equilibrium equations (5.9), we consider the stationary heat transfer equation

$$-\nabla\cdot(a\nabla T)=0, \quad (5.10)$$

where the thermal conductivity a is a nonzero constant. These equations are assumed in a bounded polygonal domain $\Omega \subset R^2$.

Boundary Conditions Let the boundary $\partial\Omega$ have nonempty open subsets $\Gamma_0, \Gamma_1, \Gamma_2$ and Γ_3, such that $\Gamma_0, \Gamma_1, \Gamma_2$ are disjunct. Equations (5.9), (5.10) are equipped with boundary conditions of the form

$$\left.\begin{array}{rcl} \sum_{j=1}^{2}\sigma_{ij}\nu_j &=& g_i^* \quad \text{on } \Gamma_0, \quad i=1,2, \\ u_1 &=& u_1^* \quad \text{on } \Gamma_1, \\ u_2 &=& u_2^* \quad \text{on } \Gamma_2, \\ T &=& T^* \quad \text{on } \Gamma_3, \\ \dfrac{\partial T}{\partial \nu} &=& T_N^* \quad \text{on } \partial\Omega \setminus \Gamma_3. \end{array}\right\} \tag{5.11}$$

Here $g_i^*, u_i^*, T^*, T_N^* \in L^2(\partial\Omega)$ are prescribed boundary force components, displacement components, temperature, and temperature flux, respectively (other standard types of boundary conditions may be used as well). The symbol ν stands for the unit outer normal vector to $\partial\Omega$.

Problem Domain We consider the cross-section of the winding of a massive coil with two cooling channels, as shown in Fig. 5.7. The material is heated by a current flowing through the winding and cooled by fluid running through the channels, whose temperature has stabilized at the value T_C. This causes a nonuniform temperature distribution in the winding and consequently thermoelastic deformations.

The transversal outer dimensions of the winding are $13L \times 5L$ and the cavities measure $5L \times 3L$. We prescribe zero displacement on Γ_A and zero external forces on the remaining part of the boundary $\Gamma_B \cup \Gamma_C$:

$$\begin{array}{rcl} (u_1, u_2) &=& 0 \quad \text{on } \Gamma_A \\ \displaystyle\sum_{j=1}^{2}\sigma_{ij}n_j &=& 0 \quad \text{on } \Gamma_B \cup \Gamma_C, \quad 1 \le i \le 2. \end{array}$$

Here, $\boldsymbol{n} = (n_1, n_2)$ stands for the unit outer normal vector to the boundary $\partial\Omega$. For the thermal part, we prescribe a fixed temperature T_C on the face

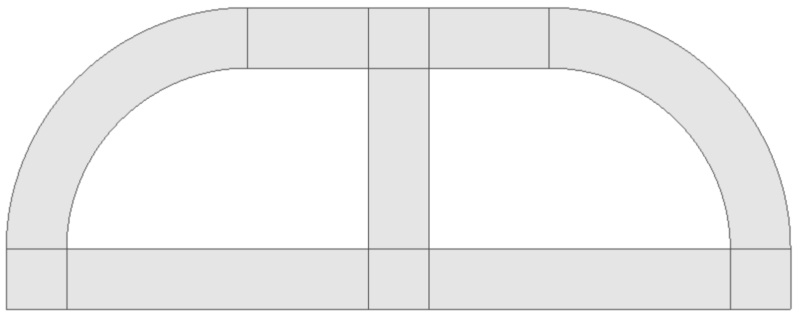


Figure 5.8: Initial (master) mesh for the computation.

Γ_C, and a negative heat flux ϕ_B on the winding-air interface Γ_B:

$$\begin{aligned} T &= T_C \quad \text{on } \Gamma_C, \\ \frac{\partial T}{\partial \boldsymbol{n}} &= \phi_B < 0 \quad \text{on } \Gamma_B, \end{aligned}$$

In the computation we used the values $L = 0.1$, $T_C = 50$, $\phi_B = -50$, $\varrho = 8000$, $g = 9.81$, $a = 1.3 \cdot 10^{-5}$, $E = 200$ GPa, $\nu = 0.3$.

Results The problem was first solved using standard, single-mesh adaptive *hp*-FEM that starts from the initial mesh shown in Figure 5.8. The solution is shown in Figure 5.9, where the displacements were used to calculate the Von Mises stress, and in Figure 5.10, which shows the temperature. Figure 5.11 shows the *hp*-mesh, shared by all three equations, after 12 adaptive steps, with approximately 2400 DOFs in each component (exact numbers vary slightly due to the different boundary conditions). The energy norm of the problem was used to guide the adaptive algorithm.

For the multi-mesh computation, mesh in Figure 5.8 served both as an initial mesh for adaptivity and a master-mesh for multi-mesh assembling. Figures 5.12, 5.13 and 5.14 show the meshes for u_1, u_2 and T, respectively, after several adaptive steps, at an error level roughly comparable to the single-mesh result in Figure 5.11. The numbers of DOFs are 2452 for u_1, 1932 for u_2 and 1005 for T. We can see that in case of T there is significantly fewer DOFs than in the single-mesh computation.

Graph in Figure 5.15 shows the convergence curves for the two computations. For all error levels, the multi-mesh computation used about 20% fewer DOFs than the single-mesh computation. In total CPU time however, the difference was smaller or not present at all, due to the higher cost of multi-mesh assembling (Figure 5.16).

5.5 Conclusion

We have shown that the multi-mesh algorithm works. It however requires careful implementation, in order for the time savings in the linear solver which result from the reduced number of DOFs not to be outweighed by the increased CPU cost of assembling. Our implementation could still be improved, but we believe that for problems where the individual equations exhibit very different behavior the method can be very successful. A good candidate for such a problem may be e.g. the dendritic growth example in [54].

However, as we will see in the following chapter, multi-mesh assembling can be used not only to save degrees of freedom, but also to enable dynamic meshes in time-dependent problems.

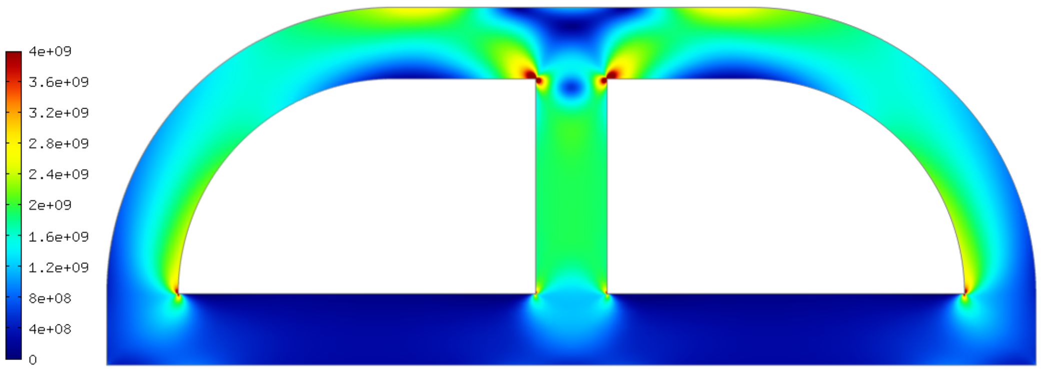


Figure 5.9: Distribution of Von Mises stress [Pa].

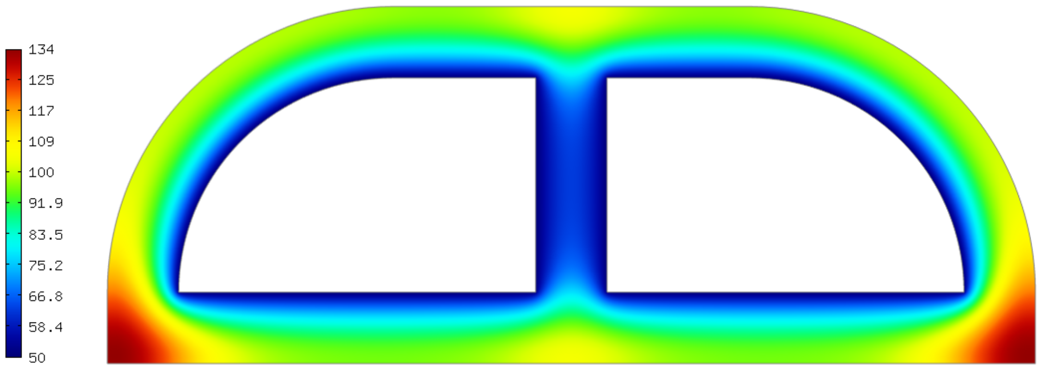


Figure 5.10: Distribution of temperature [°C].

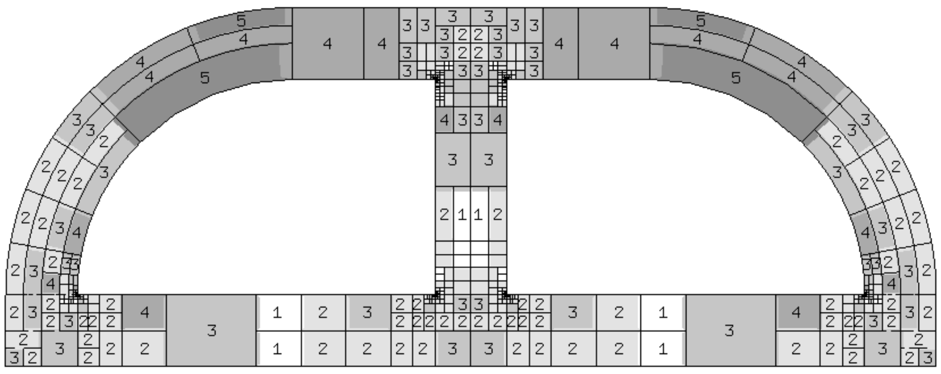


Figure 5.11: Standard *hp*-FEM mesh after 12 refinement steps (≈ 2400 DOFs).

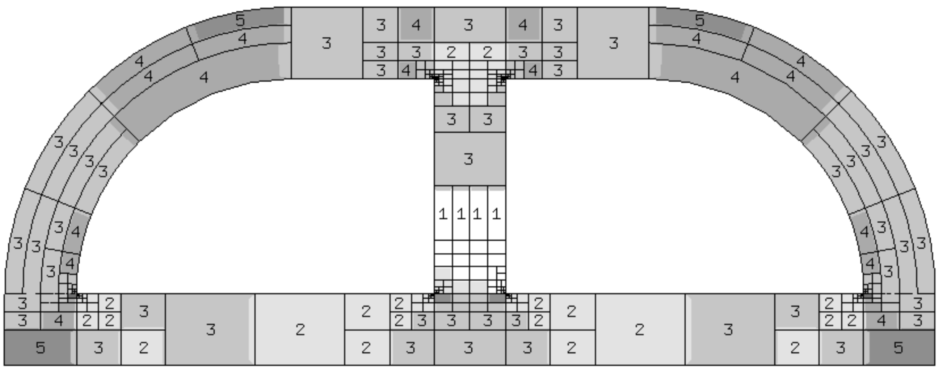


Figure 5.12: Component mesh for u_1 (2452 DOFs).

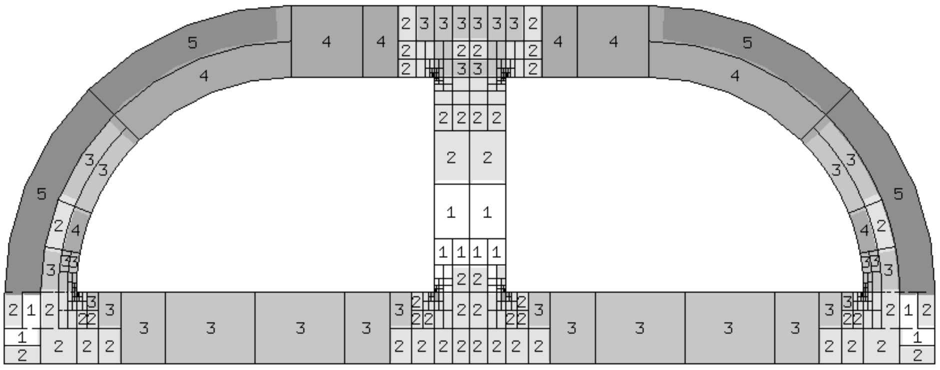


Figure 5.13: Component mesh for u_2 (1932 DOFs).

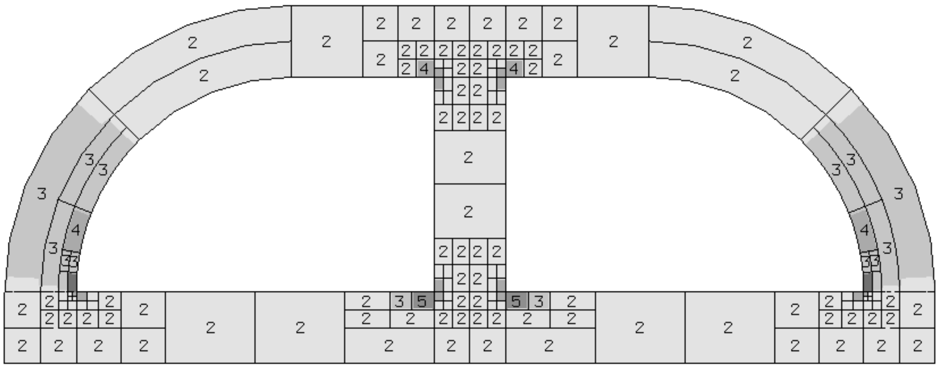


Figure 5.14: Component mesh for T (1005 DOFs).

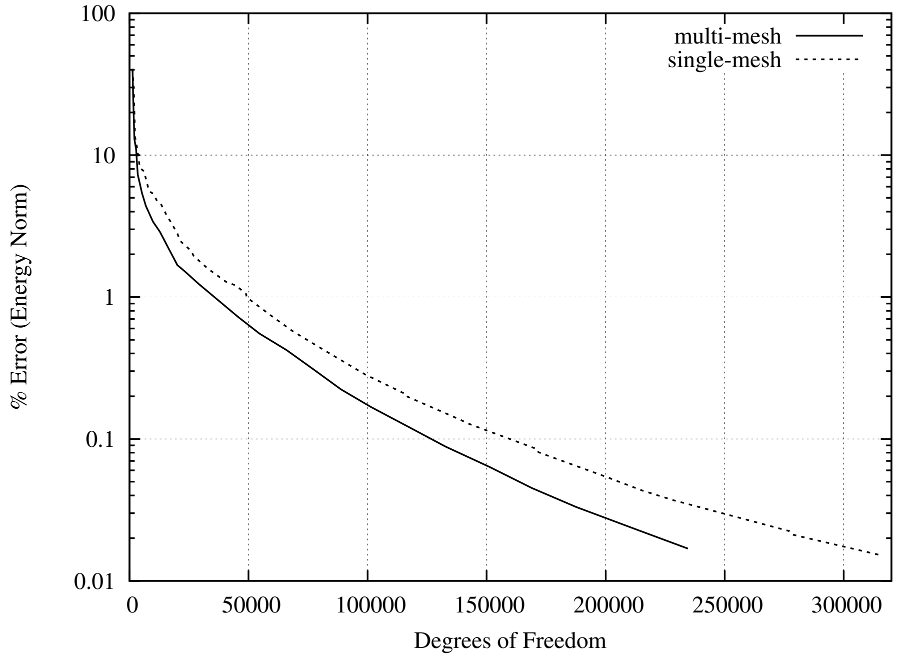


Figure 5.15: Convergence comparison of standard and multi-mesh *hp*-FEM.

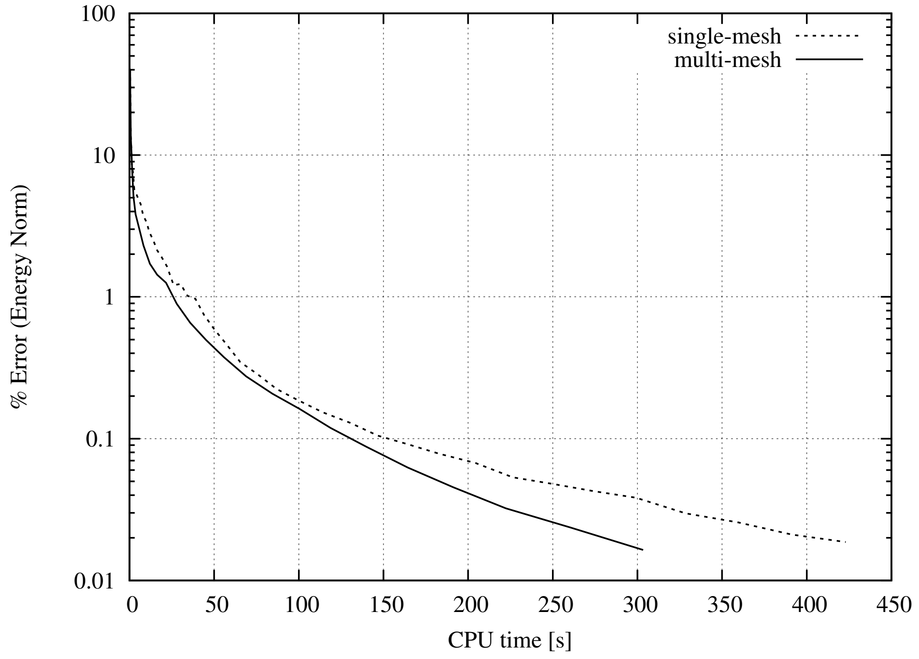


Figure 5.16: CPU time comparison of standard and multi-mesh *hp*-FEM.

Chapter 6

Time-Dependent Problems and *hp*-Adaptivity

In this final chapter we turn our attention to automatic h- and hp-adaptivity for time-dependent problems. We combine our multi-mesh assembling algorithm and the Rothe method to obtain computations with meshes that dynamically adapt in time to changing features of the solution. Our algorithms are demonstrated on two non-linear model problems.

6.1 Introduction

Time-dependent problems often exhibit localized transient phenomena such as sharp moving fronts or other problematic effects which require a finely resolved mesh for their accurate representation. This presents a fundamental difficulty for traditional approaches which only use a single mesh for all time steps. If the phenomenon we need to capture is changing its location in time, a mesh finely resolved across a wide area of the computational domain has to be used. This may greatly increase the total computational time as typically a large number of time steps must be calculated on the fine mesh.

In theory it would be possible to formulate and solve the problem in a full space-time setting, performing an adaptation in $d+1$ dimensions. On average, the number of degrees of freedom per time step should be smaller than in the single-mesh case, however the extreme storage demands necessary to hold the whole space-time cylinder makes this approach impractical, not to mention the effort required to develop, in the general case, a 4D solver. A more realistic approach (e.g., [31, 19]) is to retain the discrete time steps $t_1 \dots t_N$ and operate on space-time slabs between successive time steps t_i and t_{i+1}. We will however not follow this path in our thesis.

A less fancy way to obtain meshes that adapt both in space and in time is to occasionally re-mesh the problem and restart the chosen time-stepping method (such as the method of lines described in the next section). This approach has been routinely used in codes for fluid dynamics where moving domains are appearing frequently and where complete re-meshing is necessary anyway when the moving mesh deteriorates. Apart from its inherent cost, re-meshing entails the transfer of values from the old mesh to the new one, which is a source of error as typically some form of interpolation is used instead of a full (and expensive) projection.

Yet another approach, that we are going to pursue in this work, is to replace the usual method of lines (MOL) by its natural counterpart, the Rothe method, in which one is free to use a different mesh from one time step to another. To avoid the transfer of solutions between meshes, we conveniently utilize our multi-mesh assembling algorithm. In this chapter we first propose an algorithm which produces dynamic meshes with constant polynomial degrees and with h-refinements only. Later we extend the algorithm to perform full dynamic hp-adaptivity, something which, according to our knowledge, has not been attempted yet.

6.2 Method of Lines vs. the Rothe Method

Assume the general second-order parabolic equation,

$$\frac{\partial u}{\partial t} + Lu = f, \tag{6.1}$$

where L is a second-order elliptic operator. The basic idea of the method of lines is to discretize the spatial part $Lu = f$ analogously to time-independent problems (see Chapter 2) while keeping the temporal variable continuous. This technique is called semidiscretization in space [43].

Similarly to Section 2.1, we derive the weak formulation of (6.1) to obtain

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}t} \int_\Omega (u(t))(\boldsymbol{x})\, v(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} + a(u(t), v) &= l(v) \quad \text{for all } v \in V, \\ u(0) &= u_0, \end{aligned}$$

where $u(t) \in V = H_0^1(\Omega)$ is the sought solution $u(\boldsymbol{x}, t)$ at a time instant t. We construct the piecewise-polynomial space $V_{h,p} \subset V$ and a suitable basis

$$\{v_1, v_2, \ldots, v_N\} \subset V_{h,p}.$$

The sought function $u_{h,p}$ is expressed as a linear combination of these basis functions with time-dependent coefficients $y_j(t)$,

$$u_{h,p} = \sum_{j=1}^{N} y_j(t) v_j(\boldsymbol{x}) \tag{6.2}$$

(compare with equation (2.8) on page 6). By substituting (6.2) into the weak formulation we obtain

$$\sum_{j=1}^{N} \dot{y}_j(t) \underbrace{\int_\Omega v_j(\boldsymbol{x}) v_i(\boldsymbol{x}) \mathrm{d}\boldsymbol{x}}_{m_{ij}} + \sum_{j=1}^{N} y_j(t) \underbrace{a(v_j, v_i)}_{s_{ij}} = l(v_i),$$

$i = 1, 2, \ldots, N$. In matrix form this equation reads

$$\boldsymbol{M}\dot{\boldsymbol{Y}}(t) + \boldsymbol{S}\boldsymbol{Y}(t) = \boldsymbol{F}(t).$$

We have obtained a system of ordinary differential equations for the unknown time-dependent coefficients $\boldsymbol{Y}(t) = \{y_j(t)\}_{j=1}^{N}$. This is a standard system that can be readily solved by one of the many ODE solver packages available, thus arriving to the approximate solution $u_{h,p}(\boldsymbol{x}, t)$.

We see that in the method of lines the finite element mesh is inherently fixed for all time instants $t \in (0, T)$ because the spatial FE discretization is done before the temporal discretization within the ODE solver, where the mesh cannot be modified anymore. The natural counterpart of the method of lines is the Rothe method, in which the temporal variable is discretized first, and the spatial variable is formally left continuous. Each time step is then a separate elliptic boundary problem that can be solved (and spatially adapted) independently of other time steps. For example, we can employ the first-order backward difference formula

$$\frac{\partial f(t)}{\partial t} = \frac{f(t) - f(t - \Delta t)}{\Delta t} + O(\Delta t) \tag{6.3}$$

to approximate (6.1) as

$$\frac{u^{n+1} - u^n}{\Delta t} + L u^{n+1} = f. \tag{6.4}$$

Here u^n and u^{n+1} are the solutions to two independent boundary value problems corresponding to two successive time steps t^n and t^{n+1}. When solving for u^{n+1}, the function u^n from the previous time step poses as a constant and u^{n+1} can be resolved on a completely different mesh. This approach is sometimes called adaptive Rothe method and appeared e.g. in [8].

6.3 Basic Idea and Algorithm

In this chapter we use the adaptive Rothe method to produce computations on meshes whose spatial refinements are changing in time. The basic idea is to take the mesh adaptation algorithms we developed in Chapter 4 for stationary problems and apply them repeatedly to the isolated problems that arise in the Rothe method for each time step.

For simplicity let us assume that a one-step method such as (6.3) is used to discretize the time derivative. Denote by 'solve' the algorithm which finds u^{n+1} in equation (6.4), given the previous solution u^n and time step Δt. The solutions u^n, u^{n+1} are defined on meshes $\mathcal{T}^n, \mathcal{T}^{n+1}$, respectively. Note that the two meshes need not be identical, we only require them to have a common ancestor (master mesh, see Section 5.3), so that we can utilize the multi-mesh assembling. Formally, we have

$$u^{n+1} = \text{solve}(\Delta t, u^n, \mathcal{T}^n, \mathcal{T}^{n+1}).$$

The algorithm to solve the time-dependent problem (6.1) can then be written as Algorithm 6.1. Again, for error estimation we use the reference solution approach with all its advantages and disadvantages, as discussed in Section 4.3.

Algorithm 6.1: Adaptive Rothe method for dynamic meshes.

```
$n = 0$ ;
$\mathcal{T}^0 = \mathcal{T}_{master}$ ;
$u^0 = u_0$ // initial condition ;
$t = 0$ ;
repeat // outer iterations over time steps $t^n$
    $i = 0$ ;
    $\mathcal{T}_0^{n+1} = \mathcal{T}_{master}$ ;
    repeat // inner iterations to obtain $u^{n+1}$, time $t$ frozen
        $u_i^{n+1} = \text{solve}(\Delta t, u^n, \mathcal{T}^n, \mathcal{T}_i^{n+1})$ ;
        $\mathcal{T}_{ref} = \text{refine}(\mathcal{T}_i^{n+1})$ ;
        $u_{ref} = \text{solve}(\Delta t, u^n, \mathcal{T}^n, \mathcal{T}_{ref})$ ;
        $\mathcal{T}_{i+1}^{n+1} = \text{adapt}(\mathcal{T}_i^{n+1}, u_i^{n+1}, u_{ref})$ ;
        $err = \frac{||u_i^{n+1} - u_{ref}||}{||u_{ref}||}$ ;
        $i = i + 1$ ;
    until $err > tol$;
    $u^{n+1} = u_{ref}$ ;
    $\mathcal{T}^{n+1} = \mathcal{T}_{ref}$ ;
    $t = t + \Delta t$ // possibly adapt $\Delta t$ using ODE techniques ;
    $n = n + 1$ ;
until $t > T$;
```

The algorithm performs a full *hp* adaptation for each time step, starting with the coarsest mesh every time. Note that when calculating both the coarse and the reference solution, as the previous solution u^n we use the previous reference solution in each case. This is important because it prevents the coarse and the reference solutions from diverging with time, which would lead the algorithm to a halt due to the increasing error between the solutions.

6.4 Simple Dynamic h-Adaptivity

An obvious weakness of Algorithm 6.1 is that for each time step t^n a full adaptation is run from scratch. The results of previous time steps are always discarded, even though there is usually a great deal of similarity between successive time steps in terms of mesh refinement. We therefore view Algorithm 6.1 just as a basis on which we will try to improve.

In this section, for simplicity, we limit the adaptation to h-refinements only. However, we will attempt to reduce the number of iterations of the inner loop of the algorithm by reusing the coarse mesh from the previous time step. Instead of initializing the mesh for the inner loop as

$$\mathcal{T}_0^{n+1} = \mathcal{T}_{master},$$

we will write

$$\mathcal{T}_0^{n+1} = \text{unrefine}(\mathcal{T}_{prev}),$$

where $\mathcal{T}_{prev}$ is the last coarse mesh from the previous time iteration, stored when the inner loop finishes, and 'unrefine' is a function which removes the finest level of refinements everywhere in the mesh. We test this modified algorithm on a model CFD problem.

6.4.1 Model Problem: 2D Incompressible Viscous Flow

Consider a simple problem of external fluid flow past an infinite obstacle with a square cross-section, as shown in Figure 6.1. The flow is governed by the incompressible Navier-Stokes equations, which can be written in dimensionless form as

$$\frac{\partial \boldsymbol{u}}{\partial t} + \frac{1}{\text{Re}}\Delta \boldsymbol{u} + (\boldsymbol{u}\cdot\nabla)\boldsymbol{u} - \nabla p = 0 \qquad (6.5)$$

$$\nabla\cdot\boldsymbol{u} = 0. \qquad (6.6)$$

Here, $\boldsymbol{u} = (u_1, u_2)$ is the velocity, p is the kinematic pressure and Re denotes the Reynolds number, a dimensionless parameter which characterizes the flow behavior. We equip the equations with the following boundary conditions:

- $\boldsymbol{u} = (1, 0)$ on the inlet Γ_I,
- $\boldsymbol{u} = (0, 0)$ on the obstacle Γ_W,
- the *do-nothing* condition on the outlet Γ_O.

The initial condition is $\boldsymbol{u}(\boldsymbol{x}, 0) = 0$. The size of the obstacle is 1. After discretizing the time derivative and linearizing the convective term $(\boldsymbol{u}\cdot\nabla)\boldsymbol{u}$ using

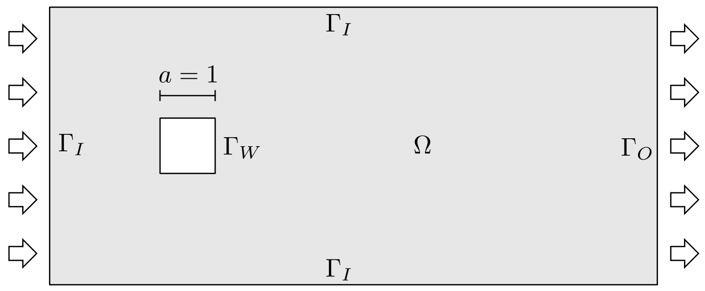


Figure 6.1: Setting for the flow problem.

the solution on the previous time level as $(\boldsymbol{u}^n\cdot\nabla)\boldsymbol{u}^{n+1}$, the weak formulation of the system (6.5)–(6.6) to be solved on each time level reads:

$$\begin{aligned}
\int \frac{u_1^{n+1}\, v_1}{\Delta t} + \frac{1}{\mathrm{Re}} \int \nabla u_1^{n+1}\cdot\nabla v_1 + \int (\boldsymbol{u}^n\cdot\nabla)\, u_1^{n+1}\, v_1 - \int p^{n+1}\frac{\partial v_1}{\partial x} &= \int \frac{u_1^n v_1}{\Delta t},\\
\int \frac{u_2^{n+1}\, v_2}{\Delta t} + \frac{1}{\mathrm{Re}} \int \nabla u_2^{n+1}\cdot\nabla v_2 + \int (\boldsymbol{u}^n\cdot\nabla)\, u_2^{n+1}\, v_2 - \int p^{n+1}\frac{\partial v_2}{\partial y} &= \int \frac{u_2^n v_2}{\Delta t},\\
\int \frac{\partial u_1^n}{\partial x} q + \int \frac{\partial u_2^n}{\partial y} q &= 0,
\end{aligned}$$

where v_1, v_2 are test functions for the velocity (applied to equation (6.5)) and q is a test function for the pressure (applied to equation (6.6)). For the spatial discretization we used quadratic continuous elements for the velocity and linear continuous elements for the pressure.

We employed Algorithm 6.1 with the mentioned modification of mesh initialization to perform three calculations with $\mathrm{Re} = 4\cdot 10^3$, $\mathrm{Re} = 20\cdot 10^3$ and $\mathrm{Re} = 100\cdot 10^3$. The time step was 0.05, 0.04 and 0.02, respectively. Snapshots of the coarse mesh at three time instants (from the calculation with $\mathrm{Re} = 20\cdot 10^3$) are shown in Figure 6.2. All meshes were obtained fully automatically from a master mesh that contains only 54 quadrilateral elements. The mesh adapts nicely in time to the changing features of the flow[1]. The adaptation in the inner loop was performed until the L^2 norm of the error fell below 0.8%, 0.6%, 0.5%, respectively, for the different Re. The reference mesh was h-refined only, no increase in the polynomial degrees was necessary in this case. For simplicity, we used the same mesh for all three solution components, i.e., multi-mesh assembling was utilized only to handle the different refinements of u^n and u^{n+1}, although independent dynamic meshes for all components are in theory also possible with Hermes2D.

It is interesting to note that thanks to the accurately resolved meshes for all time levels the calculations were quite stable even for the relatively high Reynolds numbers, which produce complex flow patterns (see Figure 6.3). Nor-

[1]See video at `http://www.youtube.com/watch?v=BAL8ilbvXkU`

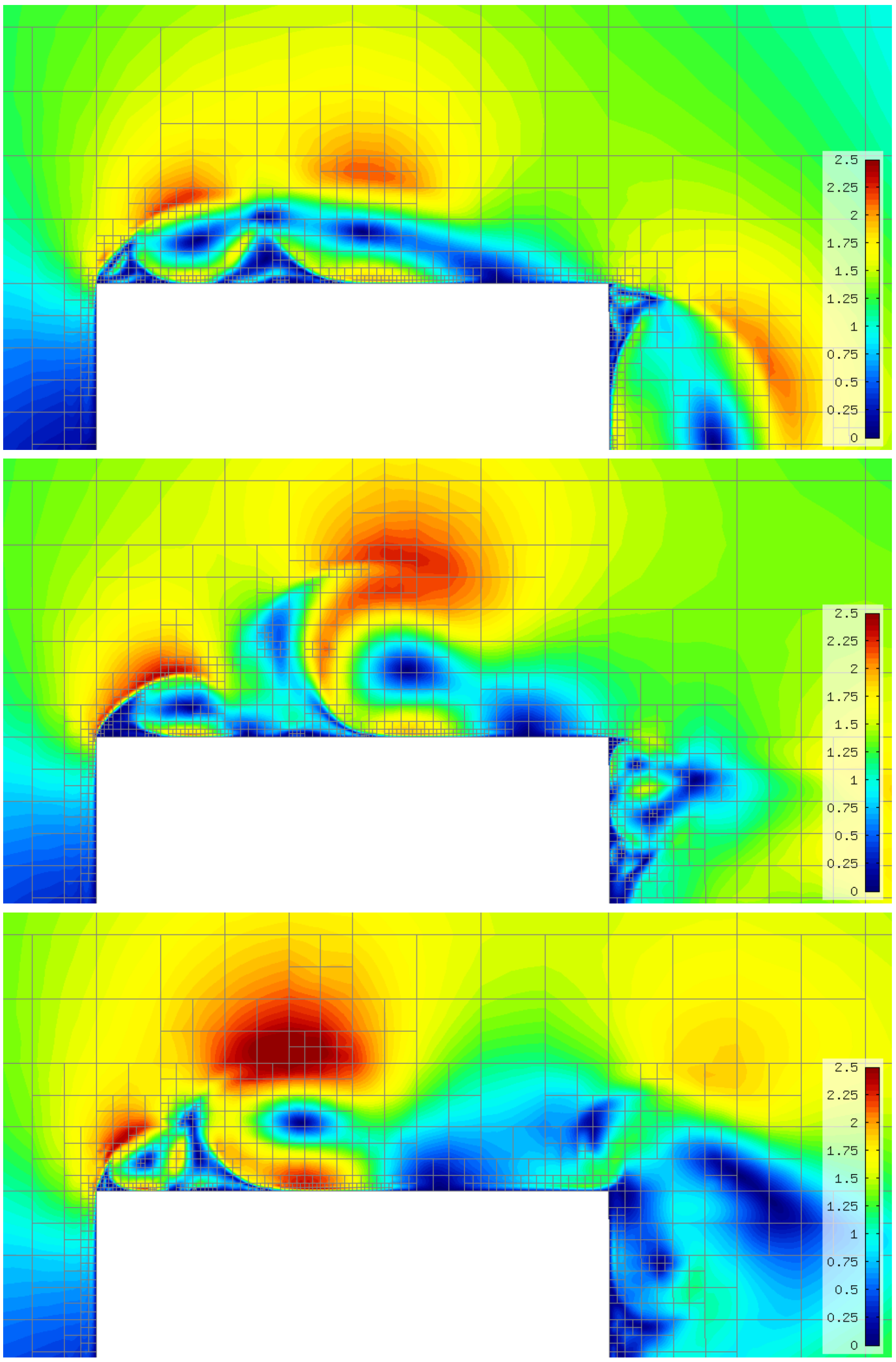
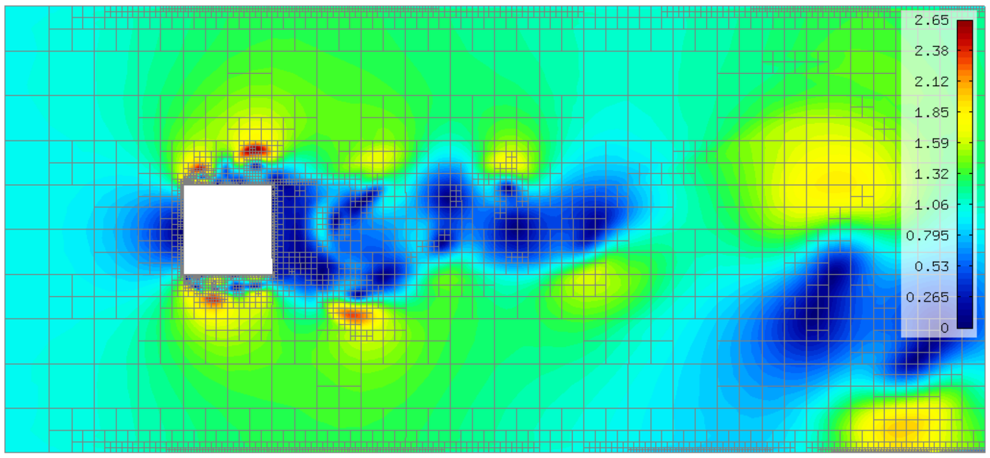


Figure 6.2: Evolving mesh for Re $= 20 \cdot 10^3$ at times $t = 13$ (top), $t = 14$ (middle) and $t = 15$ (bottom). Detail of the upper part of the obstacle is shown. Colors represent the magnitude of velocity.



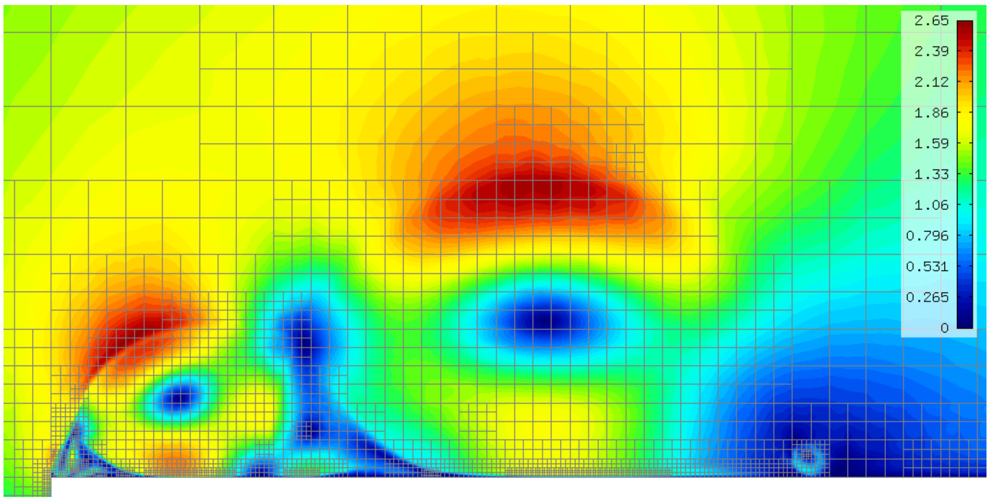


Figure 6.3: Computation with $\mathrm{Re} = 100 \cdot 10^3$. Overview of the whole domain at $t = 16$ (top). Detail of the boundary layer and complex flow features near the obstacle at $t = 7$ (bottom, reference solution shown).

mally either extremely well-designed meshes or advanced stabilization techniques are necessary in similar finite-element solvers.

Figure 6.4 shows the number of degrees of freedom of the coarse problem at the end of the inner loop, for all time levels. The following table lists the average numbers of inner (adaptive) loop iterations and the total CPU time on an Intel Q9550 CPU[2], for the three computations:

Reynolds number	$4 \cdot 10^3$	$20 \cdot 10^3$	$100 \cdot 10^3$
Average # inner iterations	6.5	9	9
Total outer iterations	400	500	1000
Total CPU time [hours]	2.2	8.8	22

[2]Our program is serial, i.e., single-core only.

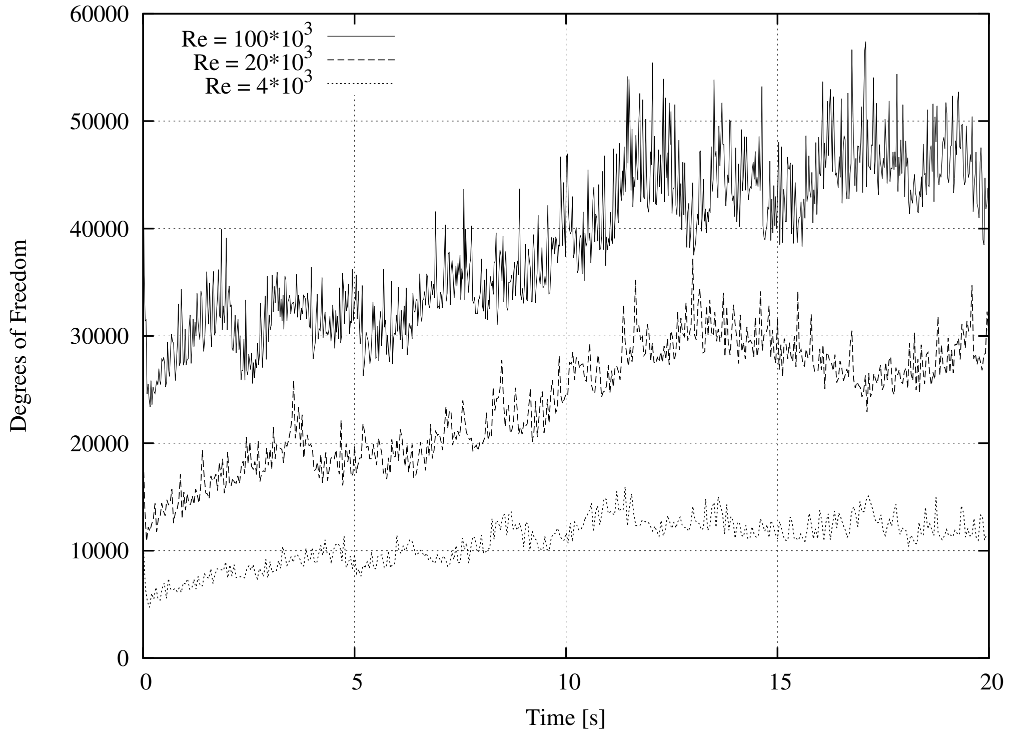


Figure 6.4: Number of DOFs of the coarse mesh as a function of time.

We see that the simple approach with 1-level coarsening works, but obviously its performance could be improved. Removing one level of refinements in each step still causes a lot of work to be lost every time, which is apparent from the numbers of inner loop iterations. Ideally, we would like this number to be about 2 on average, that would however require a more sophisticated unrefinement algorithm. Our inner loop is simply taken from the stationary algorithm in Chapter 4 and tends to over-refine the solution instead of aiming at an exact error level (hence the fuzziness of the graphs in Figure 6.4). The global stopping criterion should be replaced by local error tests of individual elements. This would in turn allow more careful element coarsening. We have however not succeeded realizing this approach for the flow problem.

A lot more work would also be required to obtain a proper flow solver. Oseen or Newton iterations would have to be used, stabilization would preferably be included to reduce the number of DOFs and the reference solution could be replaced by a specialized error estimator. However this is beyond the scope of this thesis and would lead to a loss of generality of the simple approach. We view this model problem merely as a prototype that leads us to dynamic *hp*-adaptivity, which we explore in the following section.

6.5 Full Dynamic hp-Adaptivity

In this section we improve the refinement and coarsening steps of the dynamic mesh algorithm and test full hp refinements on a model problem of combustion.

6.5.1 Improved Refinement Criterion

Recall from Section 4.4 on page 47 that in each mesh adaptation step we refine all elements e_i whose error is larger than some fraction of the error of the worst element (k is a user-specified constant),

$$e_i > k \cdot e_{max}.$$

This simple criterion works well for stationary problems, where we aim to reduce the global error as much as possible. If however we need to arrive at some specific error level, as in time-dependent problems, this method tends to over-shoot the required error and refine too many elements. We therefore introduce an additional test, which may stop the refinements early, i.e., before the standard criterion is met.

The idea is to keep track of the total error of elements that were already refined. We (very roughly) assume that after an element is refined, its error is reduced by some constant factor c, $0 < c < 1$. Remember that our elements e_i are sorted by decreasing error with increasing i. We thus keep refining the worst elements and estimate the decrease of the global error. Once the estimated error falls below the target error level, i.e, when

$$(1-c)\sum_{i=0}^{M} e_i > (\mathrm{err}_{current} - \mathrm{err}_{target})$$

for some M, we stop the adaptation. This new criterion, even though not perfect, leads to less aggressive refinements as the solution error approaches the target level.

6.5.2 Coarsening Based on Super-Coarse Solutions

Ideally, we would like the inner (adaptation) loop in Algorithm 6.1 to seldom repeat more than once or twice. For most time steps, as long as the error estimate is within the prescribed tolerance, the mesh should change only slightly or not at all since it can be assumed that there is sufficient time coherence between successive solutions u^n and u^{n+1}. In such an ideal case the cost of managing the dynamic mesh should be rather small and the whole process should resemble a standard time-stepping computation (not counting the reference solution-based error estimation). To achieve this, a better coarsening algorithm needs to be designed. Removing one layer of finest elements (where possible) after every time step is a waste of resources because the number of DOFs decreases by as much as one half and most of the coarsened elements get refined again in the next time step.

To determine which elements will not get refined immediately after being coarsened, we introduce an additional reference solution, this time however coarser than the regular coarse solution, hence the name, “super-coarse” solution. By comparing the super-coarse solution with the standard reference solution we can determine which elements can be coarsened without increasing the error too much. Since our goal is dynamic *hp*-adaptivity, we in fact need to introduce two super-coarse solutions, one for *h*-unrefinements and one for *p*-unrefinements. We can easily afford that, though, because the super-coarse solutions are extremely cheap, even in comparison with the regular coarse solution.

After a time step is finished, we take the current coarse mesh and calculate the two super-coarse solutions, to determine which elements are no longer needed. First we (temporarily) lower the polynomial degrees on all elements by one and calculate the first super-coarse solution. By comparing the result with the standard reference solution we obtain new error measures on all elements. We permanently lower the degree of elements whose error is less than $q \cdot e_{max}$, where $q \geq k$ is another user-specified parameter, representing the opposite threshold to the refinement parameter k. Elements above the $q \cdot e_{max}$ threshold are unlikely to be refined again in the next time step.

The second super-coarse solution is obtained by unrefining all elements which have exactly four active sons in a temporary copy of the coarse mesh, i.e., the same process of removing one layer of refinements as in Section 6.4. Since this time we are dealing *hp* meshes, the question is what polynomial degree to select for the *h*-unrefined elements. We choose to set the parent element's degree to the maximum of the degrees of the four descendant elements. After the super-coarse solution is calculated and subtracted from the reference solution, similarly to *p*-unrefinements, we again unrefine those elements whose error did not increase dramatically.

After the two coarsening steps the mesh is prepared for the next time step. Obviously the key to success is the right choice of the value of the parameter q, which is the most sensitive out of the three constants k, c, q. Setting it too high leads to unnecessary unrefinements and conversely a low value causes fine elements to be left in the domain.

6.5.3 Model Problem: Propagation of Laminar Flame

We demonstrate the improved dynamic *hp*-adaptivity algorithm on a nonlinear parabolic problem which appeared in [41]. The problem is a simplified model of combustion inside a closed chamber, assuming the low Mach number hypothesis. This assumption permits us to neglect the motion of the combustible fluid and focus on just two of the quantities of the combustion process, the temperature θ and fuel concentration Y.

Problem definition Assuming constant diffusion coefficients, the dimensionless equations describing the combustion process are

$$\frac{\partial \theta}{\partial t} - \Delta\theta = \omega(\theta, Y), \tag{6.7}$$

$$\frac{\partial Y}{\partial t} - \frac{1}{\mathrm{Le}}\Delta Y = -\omega(\theta, Y), \tag{6.8}$$

where Le is the ratio of diffusivity of heat and diffusivity of mass (Lewis number), and ω is the reaction rate governed by the Arrhenius law,

$$\omega(\theta, Y) = \frac{\beta^2}{2\mathrm{Le}} Y e^{\frac{\beta(\theta-1)}{1+\alpha(\theta-1)}}, \tag{6.9}$$

where α and β are two parameters. We model a freely propagating laminar flame described by (6.7)–(6.9) inside a chamber containing two cooling rods with rectangular cross-section (see Figure 6.5). The computational domain has height $H = 16$ and length $L = 60$. The cooling rods take half of the height and have length $L/4$. The cooling is modeled by Robin boundary conditions on Γ_R with heat loss parameter k. On the left boundary of the domain, Dirichlet boundary conditions corresponding to the burned state are prescribed, while the remaining boundary conditions are of the homogeneous Neumann type. In summary, the boundary conditions are

$$\begin{array}{lll} \theta = 1, & Y = 0 & \text{on } \Gamma_D, \\ \dfrac{\partial \theta}{\partial n} = 0, & \dfrac{\partial Y}{\partial n} = 0 & \text{on } \Gamma_N, \\ \dfrac{\partial \theta}{\partial n} = -k\theta, & \dfrac{\partial Y}{\partial n} = 0 & \text{on } \Gamma_R. \end{array}$$

The initial condition, chosen as

$$\theta(\boldsymbol{x}, 0) = \begin{cases} 1 & \text{for } x \le x_0, \\ e^{x_0 - x} & \text{for } x > x_0, \end{cases}$$

$$Y(\boldsymbol{x}, 0) = \begin{cases} 0 & \text{for } x \le x_0, \\ 1 - e^{\mathrm{Le}(x_0 - x)} & \text{for } x > x_0, \end{cases}$$

represents a right-traveling flame located left of the cooling rods. For this computation, the values of the various parameters are set to

$$\mathrm{Le} = 1, \ \alpha = 0.8, \ \beta = 10, \ k = 0.1, \ x_0 = 9.$$

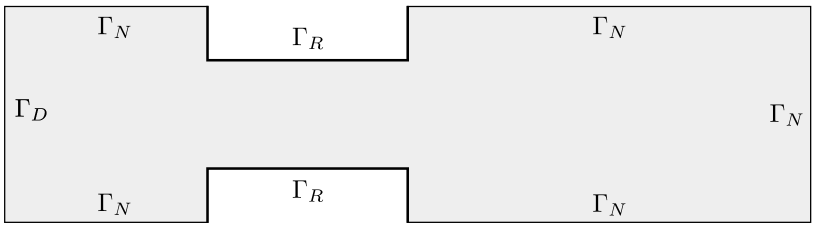


Figure 6.5: Domain for the flame propagation problem.

Results In the combustion process, as the fuel is consumed, the concentration of the reactive species Y decreases in the domain from 1 to 0, producing an increase in the temperature θ of the burned mass from 0 to 1. The points where this reaction takes place (i.e., where the flame is located) are those with high $\omega(\theta, Y)$. Instead of showing the actual solution θ, Y to the system 6.7–6.8, we display the reaction rate $\omega(\theta, Y)$, as seen in Figure 6.6, which shows the laminar flame[3] at times $t = 1.37, 19, 47.4, 59$.

Since Y is almost a constant zero left of the flame and almost a constant one right of the flame (and vice versa for θ), the only place in the domain which needs fine discretization is where the reaction rate is high. We therefore use $\omega(\theta, Y)$ as the error measure for the purpose of mesh adaptation:

$$\text{err} = \frac{||\omega(\theta, Y) - \omega(\theta_{ref}, Y_{ref})||_{H^1}}{||\omega(\theta_{ref}, Y_{ref})||_{H^1}}.$$

Meshes corresponding to Figure 6.6 obtained using the improved algorithm are shown in Figure 6.7. Error tolerance for each time step was set to 0.5% (i.e., $100 \cdot \text{err} < 0.5$). The refinement parameters were chosen $k = 0.3$, $q = 0.4$. Graph of the history of the number of DOFs for all time steps is shown in Figure 6.9 (solid line).

Besides the full *hp*-FEM computation, for comparison we also performed an *h*-adaptive computation on quadratic elements, with the same precision. One of the low-order meshes is shown in Figure 6.8 and the DOF history is shown in Figure 6.9 as a dotted line. We see that low-order FEM needed 4–5 times more degrees of freedom than *hp*-FEM, with a similar increase in CPU time.

The following table represents a histogram of the number of inner loop iterations. Thanks to the improved coarsening, the mesh needed adaptation only in every fourth time step and in the remaining steps the inner loop was executed only once:

# of inner iterations	1	2	3	4+
Percent of time steps	75%	22.5%	2%	0.3%

[3]See also a video at `http://www.youtube.com/watch?v=h1aVopx6RYA`

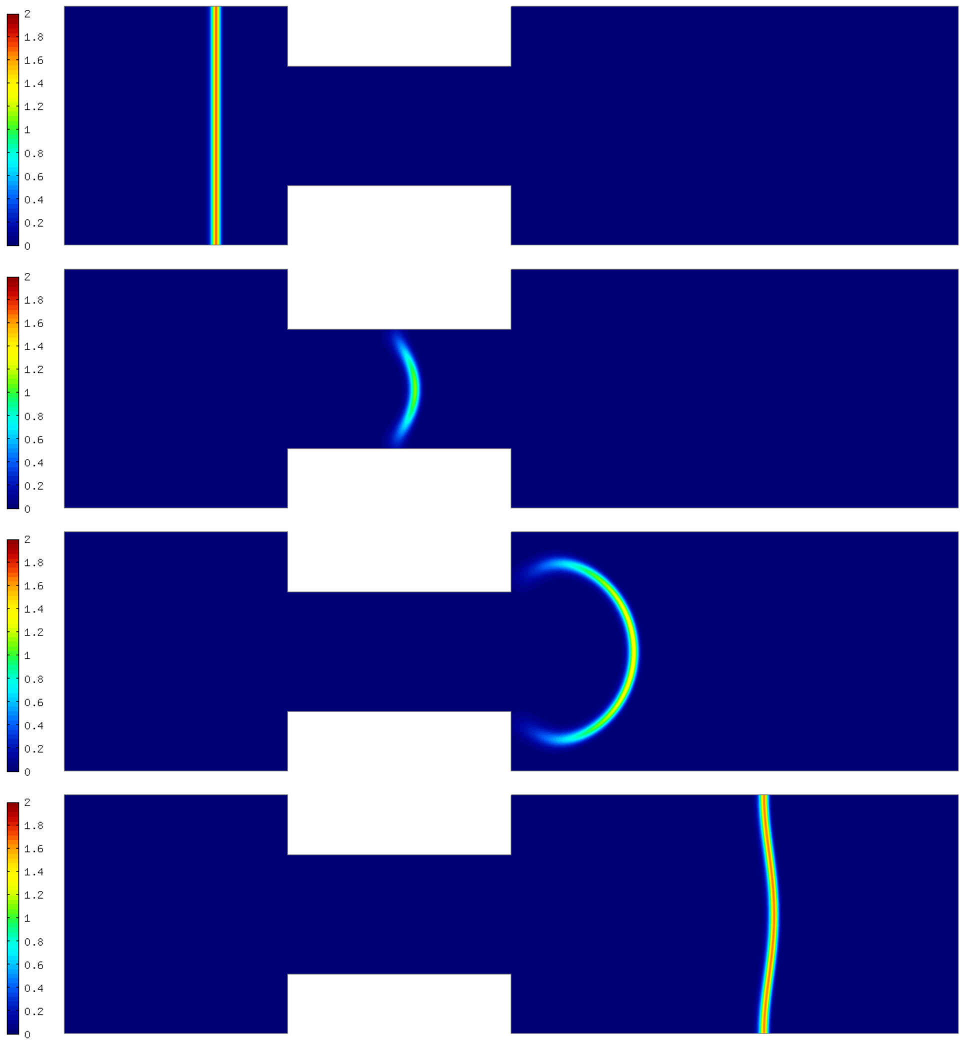


Figure 6.6: Reaction rate $\omega(\theta, Y)$ at times $t = 1.37, 19, 47.4, 59$.

Although this is largely due to the relatively small time step $\Delta t_0 = 0.01$, the number of steps with no adaptation is better than we expected and it had a positive effect on the total CPU time, which was 2.5 hours on an Intel Q9550 CPU. It is worth mentioning that performing the computation at the same accuracy on a fixed mesh (finely resolved across the whole domain) would take many times longer, even with no reference solutions.

Time-step control To further speed up the computation, we implemented a simple and fast time step adaptation proposed in [52], based on a PID controller. Even though this technique does not control the time discretization error, it works well for adapting the time step Δt so that a suitable indicator,

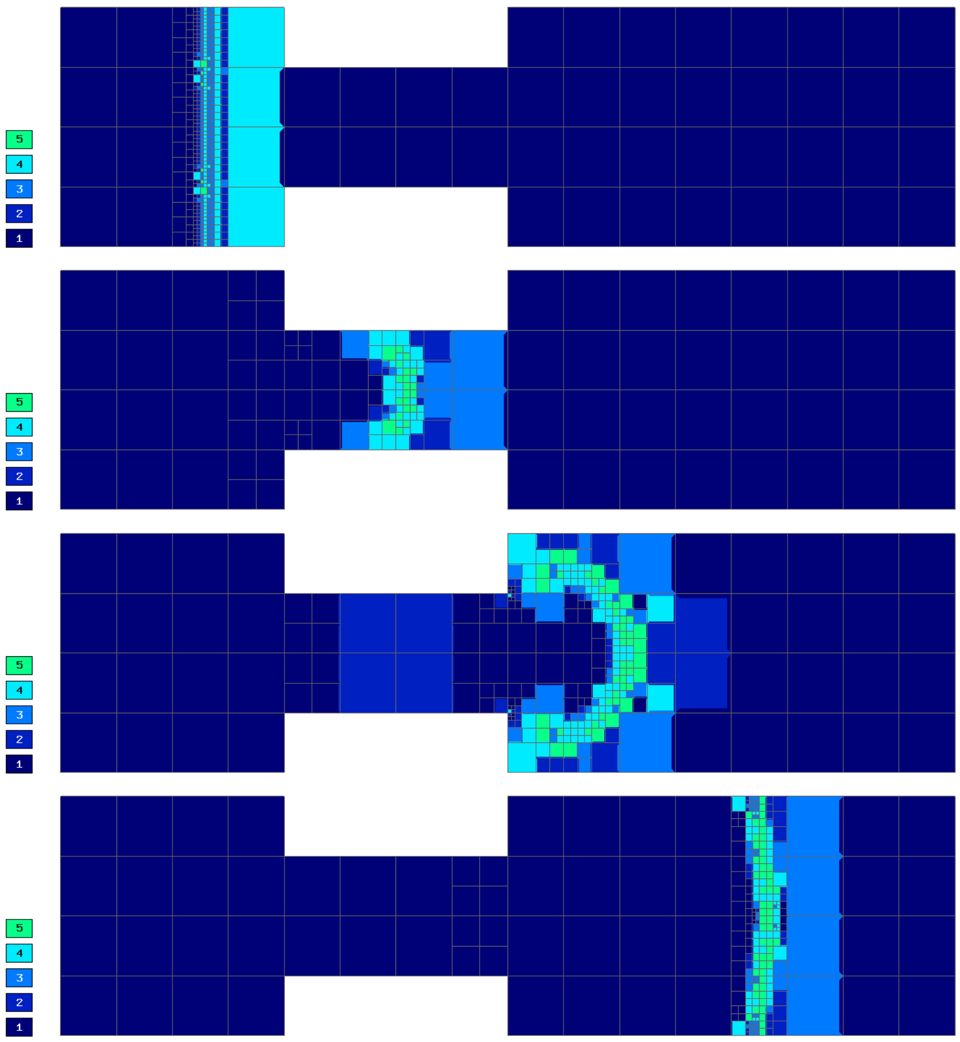


Figure 6.7: *hp*-meshes at time levels $t = 1.37,\ 19,\ 47.4,\ 59$ (top-down).

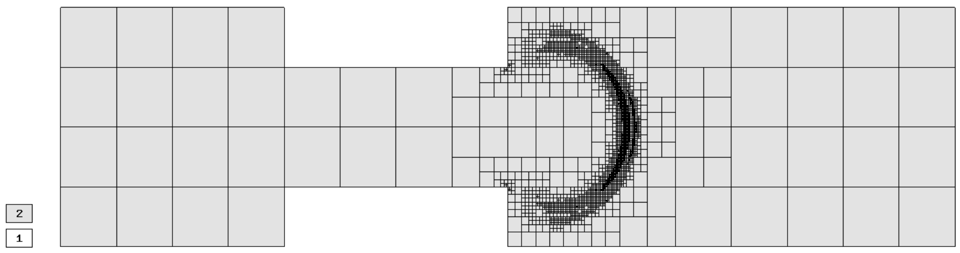


Figure 6.8: Low-order (quadratic) mesh at $t = 47.4$.

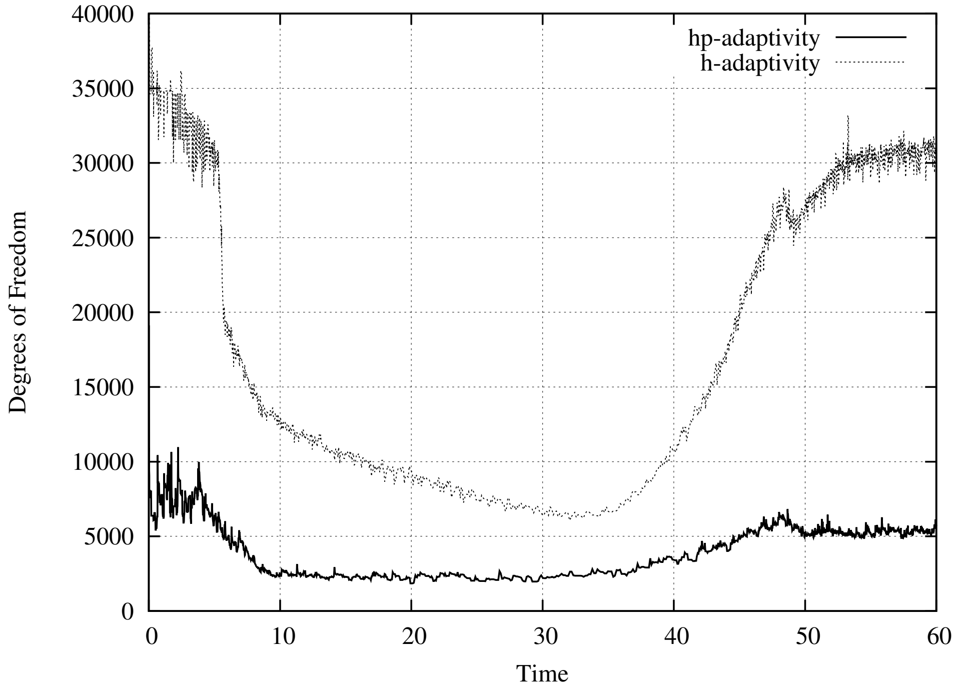


Figure 6.9: History of the number of DOFs for the combustion problem.

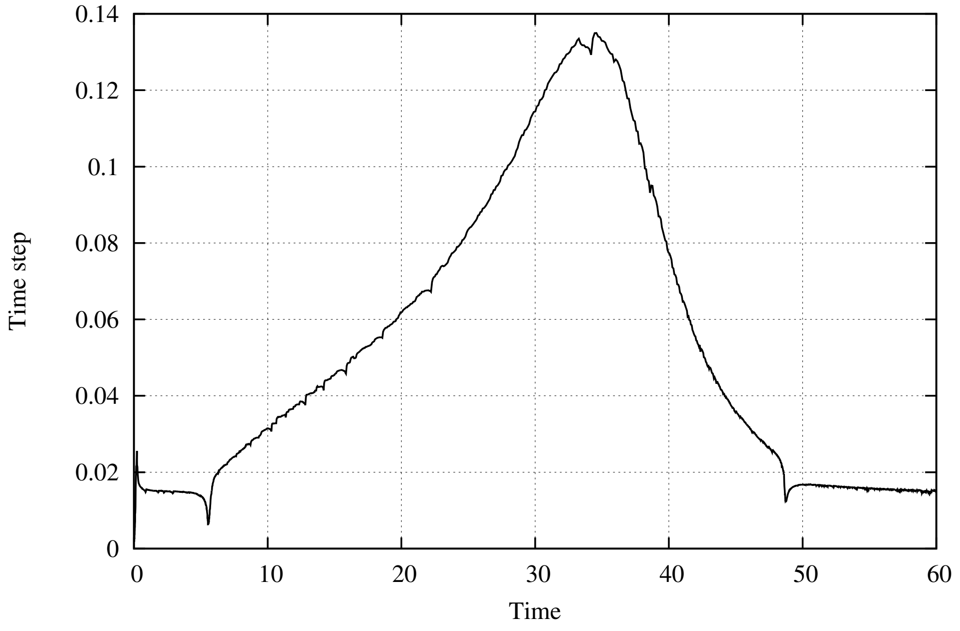


Figure 6.10: History of the time step size in the *hp*-adaptive computation.

such as

$$\epsilon_n = \frac{||\omega(\theta^{n+1}, Y^{n+1}) - \omega(\theta^n, Y^n)||}{||\omega(\theta^{n+1}, Y^{n+1})||},$$

is kept within prescribed bounds. If ϵ_n is too large ($\epsilon_n >$ tol), the solution θ^{n+1}, Y^{n+1} is discarded and recomputed using

$$\Delta t_* = \frac{\text{tol}}{\epsilon_n} \Delta t_n.$$

Otherwise the time step is adjusted smoothly using the PID formula

$$\Delta t_{n+1} = \left(\frac{\epsilon_{n-1}}{\epsilon_n} \right)^{k_P} \left(\frac{\text{tol}}{\epsilon_n} \right)^{k_I} \left(\frac{\epsilon_{n-1}^2}{\epsilon_n \epsilon_{n-2}} \right)^{k_D} \Delta t_n,$$

where the recommended values of the exponents are $k_P = 0.075$, $k_I = 0.175$, $k_D = 0.01$. The time step sizes obtained using this method are shown in Figure 6.10. The cost of this algorithm is very small since the solutions are only rarely discarded and no extra solutions (e.g., at $\Delta t/2$) are required for each time step as in other approaches. The PID controller in fact improved the total CPU time by about 40%, due to the increased time step sizes when the solution only changes slowly (about $10 \leq t \leq 45$ in the combustion problem).

6.6 Conclusions and Future Work

We have demonstrated a working approach to the solution of time-dependent problems using *hp*-FEM with dynamically changing meshes, utilizing our multi-mesh assembling algorithm. The method is superior to both dynamic *h*-FEM and traditional methods for problems exhibiting moving features in the solution and requiring accurate resolution.

Our algorithm only supports isotropic refinements. We have attempted (but not succeeded so far) to extend it for anisotropic refinements, which could further improve its efficiency. While certainly possible, this extension has proved difficult to implement. All anisotropic refinements need to be checked in every time step, even for inactive elements (those higher in the refinement hierarchy), otherwise the anisotropic refinements may become invalid. Our current unrefinement algorithm will not detect and remove these cases and they continue to exist in the mesh. Similarly, several neighboring isotropic refinement may become eligible for replacement by a single anisotropic refinement. This problem cannot be solved by a direct extension of the current algorithm and remains open.

Chapter 7

Conclusion

In this thesis we endeavored to develop several new methods and algorithms for automatically adaptive *hp*-FEM for both stationary and time-dependent problems. Here we summarize the objectives of the thesis and try to assess how they were met.

After briefly reviewing the theory of higher-order FEM in Chapter 2, we devoted Chapter 3 to constrained approximation, which enables finite element computations on irregular meshes, important for the development of adaptive higher-order solvers. We collected information on approaches used in existing nodal-element software and proposed our own solution for solvers using hierarchic bases. In addition, our method handles arbitrarily irregular meshes, which is a unique feature of our solver. We included detailed description of the implementation, with examples. Also included is an original and simple design of the underlying data structures.

In Chapter 4 we gave an overview of *hp*-adaptive strategies for stationary problems currently available in the literature. In detail we described the existing projection-based-interpolation algorithm. We then introduced our design of a simpler and faster alternative, and demonstrated on two benchmark problems that our algorithm is comparable or superior to the existing algorithms. The results were already independently verified in [35].

In Chapter 5 we described how systems of PDEs are handled in FEM solvers and motivated the use of different meshes for different equations in the system. We developed an algorithm for the assembly of the stiffness matrix of such multi-mesh systems and we provided implementation details. We tested the implementation on a model problem of thermoelasticity. Even though the results in terms of solution speed-up are not conclusive for the model problem, multi-mesh assembling has a wider applicability.

We showed the usefulness of multi-mesh assembling in the final Chapter 6, where we experimented with time-dependent problems and the adaptive Rothe method to design an algorithm which produces automatically adapted *hp*-meshes that moreover change in time and can accurately resolve transient phenomena. A simpler version with *h*-refinements only was first tested on a model problem of incompressible fluid flow. The full algorithm with improved coarsening was then demonstrated on a model problem of low-Mach number combustion. To our knowledge, a method to produce automatically adapted dynamic meshes in the context of *hp*-FEM has not yet been described in the literature.

Last but not least, an indivisible part of the thesis is the numerical software Hermes2D[1], which the author worked on during the three years of his stay at the research group of Dr. Šolín at El Paso, Texas. The project is a collaborative effort, however the author of this thesis made several fundamental contributions, including: 1. a complete modular re-design of the code, 2. the ability to handle (arbitrarily) irregular meshes, 3. two *hp*-adaptation algorithms, including the fast orthonormal version, 4. support for systems of equations and the implementation of multi-mesh assembling, 5. built-in OpenGL visualization. The code, distributed under the GNU GPL license, continues to be used by a new generation of students and a new version is underway. Thanks to its flexible design, the code has already been applied to a number of challenging problems, such as interface tracking, compressible fluid flow or complex systems coupling the equations of fluid flow, electromagnetism and heat transfer [23]. A user-friendly graphical front-end Agros2D[2] that aims to compete with commercial engineering applications was written by Pavel Karban.

7.1 Future Work

Some ideas for possible future enhancements were already mentioned in Sections 4.7 and 6.6. Those that could be immediately worked on, given more time, are the support for anisotropic *p*-refinements on quadrilaterals in the fast orthonormal *hp*-adaptive algorithm and a support for anisotropic *h*-refinements in the dynamic mesh algorithm.

The biggest problem of our methods remains the CPU time needed to obtain the reference solution. The problem is even more pronounced in 3D, where the reference solution is another order of magnitude larger than in 2D. Possible directions to fight this problem were described in Section 4.7. For selected (elliptic) problems it can make sense to use one of the alternative *hp*-adaptive strategies listed in Section 4.2.

It should not be forgotten that the ultimate goal is to extend all methods in this work to 3D. For the fast adaptive strategy this is possible, as was already discussed. The arbitrarily-irregular constrained approximation is by

[1]`http://hpfem.org/hermes2d/`
[2]`http://hpfem.org/agros2d/`

several orders more complex in 3D than it is in 2D, but the implementation was already managed in [32]. Dynamic meshes were not yet tested in 3D.

7.2 Publications and Citations

Original results of the author are presented in Chapters 3, 4, 5, 6 and were published in journal articles [47] (impact factor 0.930), [48] (i.f. 1.048), [11], in part in journal articles [25] (i.f. 0.688), [20] (i.f. 0.959), [44] (i.f. 1.048) and in conference proceedings [46], [12], [21].

Journal article [47] is cited in:

- A. Gerstenberger, W. A. Wall: Enhancement of fixed-grid methods towards complex fluid-structure interaction applications. International Journal for Numerical Methods in Fluids, vol. 57, no. 9, pp. 1227–1248, 2008.
- [6] W. Bangerth, O. Kayser-Herold: Data Structures and Requirements for *hp* Finite Element Software. ACM Transactions on Mathematical Software, Vol. 36, No. 4, March 2009.
- [35] W. F. Mitchell, M. A. McClain: A Survey of hp-Adaptive Strategies for Elliptic Partial Differential Equations. Recent Advances in Computational and Applied Mathematics (T. E. Simos, ed.), Springer, pp. 227–258, 2011.

Journal article [48] is cited in:

- C. Vokas, M. Kasper: Adaptation in coupled problems. COMPEL – The International Journal for Computation and Mathematics in Electrical and Electronic Engineering, 29 (6), pp. 1626–1641, 2010.
- R. W. Johnson, G. Hansen, C. Newman: The role of data transfer on the selection of a single vs. multiple mesh architecture for tightly coupled multiphysics applications. Applied Mathematics and Computation, vol. 217, no. 22, pp. 8943–8962, 2011.

Research report [24] is cited in:

- [54] A. Voigt, T. Witkowski: A multi-mesh finite element method for lagrange elements of arbitrary degree. Submitted, 2010.

References

[1] M. Ainsworth, J. T. Oden: A posteriori error estimation in finite element analysis. John Wiley & Sons, New York, 2000.

[2] M. Ainsworth, B. Senior: An adaptive refinement strategy for *h-p* finite element computations. Appl. Numer. Math. 26, no. 1-2, 165–178, 1997.

[3] I. Babuška, W. C. Rheinboldt: Error Estimates for Adaptive Finite Element Computations. SIAM J. Numer. Anal., 15, 736–754, 1978.

[4] I. Babuška, W. C. Rheinboldt: A-posteriori error estimates for the finite element method. Internat. J. Numer. Methods Engrg., 12, 1597–1615, 1979.

[5] I. Babuška: Error estimates for the combined h and p version of the finite element method. Numer. Math. 37, 252–277, 1981.

[6] W. Bangerth, O. Kayser-Herold: Data Structures and Requirements for *hp* Finite Element Software. ACM Transactions on Mathematical Software, Vol. 36, No. 4, March 2009.

[7] W. Bangerth, R. Hartmann and G. Kanschat: deal.II Differential Equations Analysis Library, Technical Reference. To be found on the project webpage `http://www.dealii.org`

[8] F. Bornemann: An adaptive multilevel approach to parabolic equations I. General theory and 1D-implementation. IMPACT of Comput. Sci. Engrg., 2:279–317, 1990.

[9] F. Bornemann, B. Erdmann and R. Kornhuber: Adaptive multilevel methods in three space dimensions. Int. J. Numer. Meth. Engrg. 36, 3187–3203, 1993.

[10] G. F. Carey: Computational Grids: Generation, Adaptation and Solution Strategies. Taylor & Francis, 1997.

[11] J. Červený: On constrained approximation in higher-order finite element methods in 2D. Acta Technica ČSAV, Vol. 54, No. 2, pp. 199–221, 2009.

[12] J. Červený, I. Doležel, L. Dubcová, P. Karban, P. Šolín: Higher-Order Finite Element Modeling of Electromagnetically Driven Flow of Molten Metal in a Pipe. Proc. ICEMS 2009, Tokyo, Japan, November 2009, CD-ROM. Included in IEEExplore.

[13] P. G. Ciarlet: The Finite Element Method for Elliptic Problems. North-Holland, Amsterdam, 1979.

[14] R. L. Courant: Variational Methods for the Solution of Problems of Equilibrium and Vibrations. Bulletin of the American Mathematical Society, 49, 1–23, 1943.

[15] DEAL: Differential Equations Analysis Library. Available from `http://www.math.uni-siegen.de/suttmeier/deal/deal.html`, 1995.

[16] L. Demkowicz: Computing with *hp*-Adaptive Finite Elements, Vol. 1: One and Two Dimensional Elliptic and Maxwell Problems. Chapman & Hall/CRC, 2006.

[17] L. Demkowicz, J.T.Oden, W.Rachowicz, O. Hardy: Toward a universal *hp*-adaptive finite element strategy. Part 1: constrained approximation and data structure. Comput. Methods Appl. Math. Engrg. 77:79 - 112, 1989.

[18] L. Demkowicz, W. Rachowicz, P. Devloo: A Fully Automatic *hp*-Adaptivity. TICAM Report No. 01-28, University of Texas at Austin, 2001.

[19] M. Vlasák, V. Dolejší, J. Hájek: A priori error estimates of an extrapolated space-time discontinuous Galerkin method for nonlinear convection-diffusion problems. Numer. Meth. Partial Differ. Eq., pp. N/A, 2010.

[20] I. Doležel, L. Dubcová, P. Karban, J. Červený, P. Šolín: Inductively Heated Incompressible Flow of Electrically Conductive Liquid in Pipe. IEEE Transactions on Magnetics 46, 2010.

[21] I. Doležel, L. Dubcová, P. Karban, J. Červený, P. Šolín: Inductively Heated Flow of Electromagnetically Conductive Liquid in Pipe. Proc. COMPUMAG 2009, Florianopolis, Brasil, November 2009, CD ROM. Submitted to IEEE Trans. Mag.

[22] L. Dubcová: Novel Self-Adaptive Higher-Order Finite Element Methods for the Maxwell's Equations of Electromagnetics. Master's Thesis, University of Texas at El Paso, 2008.

[23] L. Dubcová: *hp*-FEM for Coupled Problems in Fluid Dynamics. Doctoral Thesis, Charles University in Prague, Prague, 2010.

[24] L. Dubcová, P. Šolín, J. Červený: Adaptive multi-mesh *hp*-FEM for linear thermoelasticity. Research Report No. 2007-08, The University of Texas at El Paso, 2007.

[25] L. Dubcová, P. Šolín, J. Červený, P. Kůs: Space and Time Adaptive Two-Mesh hp-FEM for Transient Microwave Heating Problems. Electromagnetics, in press, 2009.

[26] T. Eibner, J. M. Melenk: An adaptive strategy for *hp*-FEM based on testing for analyticity. Comp. Mech. 39, 575–595, 2007.

[27] M. S. Gockenbach: Understanding and Implementing the Finite Element Method. SIAM, 2006.

[28] B. Guo, I. Babuška: The *h-p* version of the finite element method. Part I: The basic approximation results. Comp. Mech. 1, 21–41, 1986.

[29] B. Guo, I. Babuška: The *h-p* version of the finite element method. Part II: The general results and application. Comp. Mech. 1, 203–220, 1986.

[30] W. Gui, I. Babuška: The *h*, *p* and *h-p* versions of the finite element method in 1 dimension. Part 3: The adaptive *h-p* version. Numer. Math. 49, 659–683, 1986.

[31] T. J. R. Hughes, J. R. Stewart: A space-time formulation for multiscale phenomena. Journal of Comp. and Appl. Math. 74, 217-229, 1996.

[32] P. Kůs: Automatic *hp*-Adaptivity on Meshes with Arbitrary-Level Hanging Nodes in 3D. Doctoral Thesis, Institute of Mathematics, Academy of Sciences of Czech Republic, Prague, 2011.

[33] C. Mavriplis: Adaptive mesh strategies for the spectral element method. Comput. Methods Appl. Mech. Engrg. 116, 77–86, 1994.

[34] J. M. Melenk, B. I. Wohlmuth: On residual-based a-posteriori error estimation in *hp*-FEM. Adv. Comput. Math. 15, 311–331, 2001.

[35] W. F. Mitchell, M. A. McClain: A Survey of hp-Adaptive Strategies for Elliptic Partial Differential Equations. Recent Advances in Computational and Applied Mathematics (T. E. Simos, ed.), Springer, pp. 227–258, 2011.

[36] A. Patra, A. Gupta: A systematic strategy for simultaneous adaptive *hp* finite element mesh modification using nonlinear programming. Comput. Methods Appl. Mech. Engrg., 190, 3797–3818, 2001.

[37] M. C. Rivara: Mesh refinement processes based on the generalized bisection of simplices. SIAM J. Numer. Anal. 21, 604–613, 1984.

[38] Ruo Li: On multi-mesh *h*-adaptive methods. J. Sci. Comput., 24(3):321–341, 2005.

[39] Yana Di, Ruo Li: Computation of dendritic growth with level set model using a multi-mesh adaptive finite element method. J. Sci. Comput., 39(3):441–453, 2009.

[40] Ruo Li, X. Hu and Tao Tang: A multi-mesh adaptive finite element approximation to phase field models. Commun. Comput. Phys, 5:1012–1029, 2009.

[41] M. Schmich, B. Vexler: Adaptivity with Dynamic Meshes for Space-Time Finite Element Discretizations of Parabolic Equations. SIAM J. Sci. Comput., Vol. 30, No. 1, pp. 369–393, 2008.

[42] A. Schmidt: A multi-mesh finite element method for phase field simulations. Lecture Notes in Computational Science and Engineering, 32:208–217, 2003.

[43] P. Šolín: Partial Differential Equations and the Finite Element Method. J. Wiley & Sons, 2005.

[44] P. Šolín, D. Andrš, J. Červený, M. Šimko: PDE-Independent Adaptive *hp*-FEM Based on Hierarchic Extension of Finite Element Spaces. J. Comput. Appl. Math. 233, 3086-3094, 2010.

[45] P. Šolín, K. Segeth and I. Doležel: Higher-Order Finite Element Methods. Chapman & Hall/CRC, 2003.

[46] P. Šolín, J. Červený, L. Dubcová, I. Doležel: Multi-Mesh *hp*-FEM for Thermally Conductive Incompressible Flow. In: Proceedings of ECCOMAS Conference COUPLED PROBLEMS 2007, CIMNE, Barcelona, pp. 677–680.

[47] P. Šolín, J. Červený, I. Doležel: Arbitrary-Level Hanging Nodes and Automatic Adaptivity in the *hp*-FEM. Math. Comput. Simul. 77, 117 - 132, 2008.

[48] P. Šolín, J. Červený, L. Dubcová, D. Andrš: Monolithic Discretization of Linear Thermoelasticity Problems via Adaptive Multimesh *hp*-FEM. J. Comput. Appl. Math, in press, 2009.

[49] G. Strang, G. J. Fix: An Analysis of the Finite Element Method. Prentice-Hall, Englewood Cliffs, NJ, 1973.

[50] F. T. Suttmeier: On concepts of PDE software: The cellwise oriented approach in DEAL. International Mathematical Forum, No. 1, 1–20, 2007.

[51] B. Szabó, I. Babuška: Finite Element Analysis. John Wiley & Sons, New York, 1991.

[52] A. M. P. Valli, G. F. Carey and A. L. G. A. Coutinho: Control strategies for timestep selection in simulation of coupled viscous flow and heat transfer. Commun. Numer. Methods Eng. 18, No. 2, 131–139, 2002.

[53] R. Verfürth: A review of a posteriori error estimation and adaptive mesh refinement techniques. Wiley Teubner, Chichester Stuttgart, 1996.

[54] A. Voigt, T. Witkowski: A multi-mesh finite element method for lagrange elements of arbitrary degree. Submitted, 2010.

[55] M. Zítka, P. Šolín, T. Vejchodský, F. Avila: Imposing Orthogonality to Hierarchic Higher-Order Finite Elements. Math. Comput. Simul. 76, 211–217, 2007.

[56] O. C. Zienkiewicz, Y. K. Cheung: The Finite Element Method in Structural and Continuum Mechanics. McGraw-Hill, 1967.