TICAM Report 02-06

2D *hp*-ADAPTIVE FINITE ELEMENT PACKAGE (2Dhp90) VERSION 2.0

L. Demkowicz

Texas Institute for Computational and Applied Mathematics
The University of Texas at Austin
Austin, TX 78712

Abstract

This document provides a description of the second version of *2Dhp90* - a package supporting 2D Finite Element *hp*-approximations for the solution of various boundary-value problems, written in FORTRAN 90. The discretization is defined on a hybrid grid consisting of both triangles and quads and allows for both h- and p-refinements of the mesh. The new version has been equipped with automatic h and hp refinements routines.

Acknowledgment

The work has been partially supported by Air Force under Contract F49620-98-1-0255. The computations reported in this work were done through the National Science Foundation's National Partnership for Advanced Computational Infrastructure. The code represents a multi-year effort and collaboration with my colleagues and students: Waldek Rachowicz, Andrzej Karafiat, Krzysiek Banaś, Yao Chang-Chang, Klaus Gerdes, Leon Vardapetyan, Andrzej Bajer, Timothy Walsh and David Pardo.

Contents

1 Introduction **3**

2 Fundamentals **5**

2.1 A Model Problem: Poisson Equation in 2D 5

2.2 Variational formulation 6

2.3 Galerkin method 9

3 The *hp* Finite Element Method on Regular Meshes **11**

3.1 One-dimensional *hp* discretization 11

3.1.1 A 1D master element of order p 11

3.1.2 A 1D parametric element of arbitrary order 13

3.1.3 1D *hp* finite element space 15

3.2 Quadrilateral and triangular master elements with variable order of approximation 17

3.2.1 Quadrilateral master element 17

3.2.2 Triangular master element 20

3.3 Parametric element 22

3.4 Finite element space. Construction of basis functions 24

3.5 Data structure in FORTRAN 90 27

3.6 The element routine 28

3.7 Modified element. Imposing Dirichlet boundary conditions. 30

3.8 Interface with a frontal solver 34

3.9 Initial mesh generator. Interfacing with the Geometric Modeling Package (GMP) 35

3.10 p-adaptivity 37

4 The *hp* Finite Element Method on h-Refined Meshes **39**

4.1 Introduction. The h-refinements 39

4.2 1-Irregular Mesh Refinement Algorithm 40

4.3 Data structure in Fortran 90 - continued. 43

4.4 Constrained approximation for C^0 discretizations 46

4.5 Reconstructing element nodal connectivities. 49
4.6 Determining neighbors for mid-edge nodes 52
4.7 Additional comments 53

5 Organization of the code 55
5.1 Graphics interface 56
5.2 Graphics package 56
5.3 Files 57
5.4 How to prepare the data ? 58
5.5 Running the code in the complex mode. 59

6 Automatic *hp*-Adaptivity 60
6.1 The main idea 60
6.2 Discussion of one step of the algorithm 60
6.3 Example: L-shape domain problem. 64
6.4 Additional remarks 65

7 Concluding Remarks 68

1 Introduction

The purpose of these notes is twofold:

- to introduce a reader to the concept of *hp*-adaptive Finite Element Methods, in particular to the h-refinements and constrained approximation;
- to provide a minimal user information for the package.

Consequently, we have decided to abandon the traditional structure of a FE code manual (theory, user and example manuals) and write this document in a format of lecture notes. Indeed, the code has been used in the EM 394F (Finite Element Methods) class taught in the ASE/EM Department at the University of Texas at Austin.

For a short history of *hp* FE methods, we refer to [19].

Compared with the first implementations [6, 8], the following essential changes have been introduced.

1. Data structure has been significantly modified. The new version consists of three arrays of user defined (derived) objects. The first array contains information about the initial mesh elements. Contrary to all previous implementations, no information is stored for elements that result from h-refinements! Two other arrays contain information for vertex and non-vertex (higher order) nodes. The information stored for vertex nodes is in several aspects different from that for mid-edge and middle (mid-quad and mid-triangle) nodes, and we have found it convenient to deal with these two classes of nodes separately.
2. The information about h-refinements is supported by 'growing' family trees but not for the elements as in the previous versions, but the nodes instead. Trees for the middle nodes can be identified with those for elements, but additional trees are 'grown' for mid-edge nodes now. The difference is much more pronounced in 3D [12] where separate trees are constructed for mid-edge, mid-face and middle nodes.
3. The entire information for elements resulted from h-refinements is reconstructed from the nodal family trees. This includes not only finding neighbors for elements or nodes but, first of all, the connectivity information for all non-initial mesh elements.
4. The fact that we do not store the nodal connectivities for elements, has allowed for a drastic simplification of routines supporting h-refinements. As a result of it, we have been able to introduce three possible, *instantaneous* refinements for a quad element that now can be broken in two ways into two elements, or directly into four. This is an

essential difference with the first implementation, where the refinement into four element sons was executed through three consecutive $h2$-refinements. Again, the changes are much more significant for the 3D implementation.

5. Reconstruction of nodal connectivities for an element, is accompanied by a simultaneous creation of a local data base for the element constrained nodes. This facilitates and simplifies the logic of constrained approximation routines.

6. Only single constrained nodes are allowed. An element is not broken until all its nodes are regular. This simplifies the constrained approximation as well.

7. The code has been equipped with a package performing automatic hp-refinements. The hp strategy is based on an interplay between two meshes: the current *coarse* mesh, and a *fine mesh* obtained from the coarse mesh through a global hp-refinement. The problem is solved on both meshes and the next, optimal coarse mesh is constructed by minimizing the hp-interpolation error for the fine grid solution interpolated on the (next) coarse mesh. For details, see [11]. In the present version, the fine mesh solution is still obtained using the frontal solver which makes the code slow and severely limits the size of problems that can be solved using the strategy. In version 2.1 of the code that we expect to release within months, the fine grid solution is obtained using a two-grid solver. Additionally, our current experience indicates that the solution on the fine grid can be replaced just with a couple of smoothing operations.

8. For possible comparisons on different adaptive strategies, the code has also been equipped with a package performing automatic h-refinements only.

9. The entire code, including automatic hp-adaptive package, has been set up not only for a single equation but for an arbitrary system of (elliptic) PDE's in two space variables. With only a few minimal changes, the code can also be used to solve boundary-value problems involving complex-valued functions.

2 Fundamentals

In this section we shall use a model PDE (Partial Differential Equation) problem to discuss the fundamentals: formulation of a boundary-value problem, variational (weak) formulation, abstract variational formulation, and Galerkin method.

2.1 A Model Problem: Poisson Equation in 2D

We shall restrict ourselves to the following model boundary-value problem, illustrated in Fig. 1.

$$\begin{cases} \text{Find } u(\boldsymbol{x}), \boldsymbol{x} \in \bar{\Omega}, & \text{such that:} \\ -\Delta u = f & \text{in } \Omega \\ u = u_0 & \text{on } \Gamma_1 \\ \dfrac{\partial u}{\partial n} = g & \text{on } \Gamma_2 \end{cases} \tag{2.1}$$

Here Ω is a bounded domain (i.e. open and connected set) in $\mathbb{R}^2$ with boundary Γ consisting of two disjoint parts Γ_1 and Γ_2. We shall assume that Γ_1 has a positive measure (length).[1]. The data include:

- the right-hand side of the PDE, function $f(\boldsymbol{x}), \boldsymbol{x} \in \Omega$, specified in the whole domain Ω;
- the Dirichlet boundary condition data, function $u_0(\boldsymbol{x}), \boldsymbol{x} \in \Gamma_1$, specified on *Dirichlet boundary* Γ_1;
- the Neumann boundary condition data, function $g(\boldsymbol{x}), \boldsymbol{x} \in \Gamma_2$, specified on *Neumann boundary* Γ_2.

Finally, Δu and $\frac{\partial u}{\partial n}$ denote the Laplace operator and the normal derivative which, in a Cartesian system of coordinates, take on the following form:

$$\begin{aligned} \Delta u &= \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} \\ \frac{\partial u}{\partial n} &= \sum_{i=1}^{2} \frac{\partial u}{\partial x_i} n_i, \end{aligned} \tag{2.2}$$

with n_i denoting components of the outward normal unit vector to Neumann boundary Γ_2.

[1] We shall comment on the case $\Gamma_1 = \emptyset$ in the last section of this manual

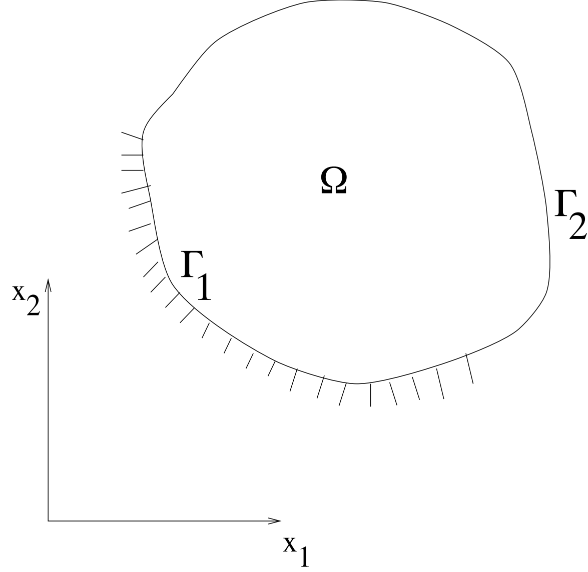


Figure 1: Model boundary-value problem

2.2 Variational formulation

We shall discuss first the case of *homogeneous* Dirichlet boundary condition, i.e. assume that $u_0 = 0$. Assume that u is a solution to the model boundary-value problem. Let $v = v(\boldsymbol{x})$ be a *test function*, *vanishing* on Dirichlet boundary Γ_1,

$$v = 0 \text{ on } \Gamma_1. \tag{2.3}$$

Multiplying equation $(2.1)_1$ by the test function v and integrating over domain Ω, we obtain

$$\int_\Omega -\Delta u \, v = \int_\Omega f v \tag{2.4}$$

and, upon integrating the left-hand side by parts,

$$\int_\Omega \boldsymbol{\nabla} u \boldsymbol{\nabla} v - \int_{\Gamma_2} \frac{\partial u}{\partial n} v = \int_\Omega f v \tag{2.5}$$

where $\boldsymbol{\nabla} u$ denotes the gradient of function u. Note that the boundary integral has been reduced *only* to the Neumann part, due to the assumption that the test function v vanishes on the Dirichlet boundary. Using the Neumann boundary condition $(2.1)_3$, we conclude that the solution u satisfies the following integral identity, known as the *variational (weak) formulation* of the boundary-value problem.

$$\left\{ \begin{array}{l} \text{Find } u(\boldsymbol{x}), \text{ vanishing on } \Gamma_1, \text{ such that} \\ \displaystyle\int_\Omega \boldsymbol{\nabla} u \boldsymbol{\nabla} v = \int_\Omega f v + \int_{\Gamma_2} g v, \\ \quad \text{for all test functions } v \text{ vanishing on } \Gamma_1 \end{array} \right. \tag{2.6}$$

One can show that, conversely, any sufficiently regular solution[2] to the variational problem satisfies the classical formulation as well [1, 15]. Thus, up to the regularity of the solution, the two formulations are equivalent.

It is convenient to identify more precisely the algebraic structure of the weak formulation. Towards this end we introduce the *space of test functions*, i.e. functions that satisfy the homogeneous Dirichlet BC's,

$$V = \{v(\boldsymbol{x}) \ : \ v = 0 \text{ on } \Gamma_1\}, \tag{2.7}$$

and identify the left-, and right-hand sides of the variational formulation (2.6) as *bilinear form* $b(u, v)$,

$$b(u, v) = \int_\Omega \boldsymbol{\nabla} u \boldsymbol{\nabla} v \tag{2.8}$$

and *linear form* $l(v)$,

$$l(v) = \int_\Omega f v \ + \int_{\Gamma_2} g v \tag{2.9}$$

This leads to the *abstract variational boundary-value problem*

$$\begin{cases} \text{Find } u \in V, \text{ such that:} \\ b(u, v) = l(v), \quad \forall v \in V \end{cases} \tag{2.10}$$

A precise mathematical setting involves the theory of Sobolev spaces and the notion of distributional derivatives [15]. Without going into details, we mention only that the space V is assumed to be a subspace of Sobolev space of first order $H^1(\Omega)$,

$$V = \{u \in H^1(\Omega) \ : \ u = 0 \text{ on } \Gamma_1\} \tag{2.11}$$

and the boundary values of functions u are understood in the sense of *traces*. Typical regularity assumptions on data, within that theory, are that functions f and g are square-integrable on Ω and Γ_2, respectively. With these assumptions, one can show that the model problem has a unique solution that depends continuously on the data. We say that the problem is *well-posed*.

Non-homogeneous Dirichlet data

We turn now to the case of $u_0 \neq 0$. We shall assume that the Dirichlet data u_0 has been selected in such a way that it admits an extension (a *lift*), denoted by $\tilde{u}_0$, defined on whole domain Ω with the same regularity as that anticipated for the solution. Mathematically

[2] It is the regularity of the solution that distinguishes the classical and variational formulations of the boundary value problems, e.g., the variational formulation is valid for only piecewice smooth data f and g, while the classical formulation is not.

speaking, the extension $\tilde{u}_0$ must live in the Sobolev space $H^1(\Omega)$. This excludes, in particular, the possibility of *discontinuous* Dirichlet data u_0. Note that the extension is not unique as it is determined up to an arbitrary function satisfying the homogeneous Dirichlet boundary condition.

Derivation of the variational boundary-value problem is now identical as for the homogeneous case. We end up with the following formulation.

$$\left\{\begin{array}{l} \text{Find } u(\boldsymbol{x}), u = u_0 \text{ on } \Gamma_1, \text{ such that} \\ \displaystyle\int_\Omega \boldsymbol{\nabla} u \boldsymbol{\nabla} v = \int_\Omega f v + \int_{\Gamma_2} g v, \\ \quad \text{for all test functions } v = 0 \text{ on } \Gamma_1 \end{array}\right. \tag{2.12}$$

Note the different Dirichlet boundary conditions for the solution and the test functions. This difference is reflected in the algebraic structure of the problem isolated in the abstract variational formulation:

$$\left\{\begin{array}{l} \text{Find } u \in \tilde{u}_0 + V, \text{ such that:} \\ b(u,v) = l(v), \quad \forall v \in V. \end{array}\right. \tag{2.13}$$

The set $\tilde{u}_0 + V$ has the structure of an *affine subspace* (of $H^1(\Omega)$), and it is defined as the collection of all sums $\tilde{u}_0 + v$ where $\tilde{u}_0 \in H^1(\Omega)$ is the lift of the boundary data, and v is an arbitrary function from $H^1(\Omega)$ satisfying the homogeneous Dirichlet boundary conditions:

$$\tilde{u}_0 + V = \{\tilde{u}_0 + v \ : \ v \in V\}. \tag{2.14}$$

Notice that the affine space is independent of the choice of the lift of the Dirichlet data as long as that lift is in $H^1(\Omega)$. Simply speaking, once we have found a particular function $\tilde{u}_0 \in H^1(\Omega)$ that satisfies the non-homogeneous Dirichlet data, we can make then the substitution $u = \tilde{u}_0 + w$ where $w \in V$ satisfies the homogeneous Dirichlet boundary conditions, and set to determine the perturbation w. The corresponding abstract formulation is then as follows.

$$\left\{\begin{array}{l} \text{Find } w \in V, \text{ such that:} \\ b(w,v) = l(v) - b(\tilde{u}_0, v), \quad \forall v \in V \end{array}\right. \tag{2.15}$$

Thus, solution of the non-homogeneous case reduces thus to the homogeneous one, provided we can find the lift and modify next the right-hand side according to the formula above. Note that, with $\tilde{u}_0$ fixed, the right-hand side of (2.15) defines a *modified* linear form

$$l_{mod}(v) = l(v) - b(\tilde{u}_0, v) \tag{2.16}$$

Again, with appropriate regularity assumptions, one can show that the problem above is well defined, i.e. it has a unique solution that depends continuously upon the data for the problem [15].

2.3 Galerkin method

We shall restrict ourselves to the Bubnov-Galerkin method only. Given a *finite-dimensional* subspace V_h of V, we look for the solution[3] $u_h \in V_h$ of the following *approximate variational boundary-value problem.*

$$\left\{ \begin{array}{l} \text{Find } u_h \in \tilde{u}_0 + V_h, \text{ such that:} \\ b(u_h, v_h) = l(v_h), \quad \forall v_h \in V_h \end{array} \right. \tag{2.17}$$

Introducing a basis in space V_h,

$$e_{hi}, \; i = 1, \ldots, N_h \tag{2.18}$$

where $N_h = \dim V_h$ is the dimension of the approximate space, we look for the approximate solution in the form:

$$u_h(\boldsymbol{x}) = \sum_{k=1}^{N_h} u_{hk} e_{hk}(\boldsymbol{x}) \tag{2.19}$$

The unknown coefficients $u_{hk}, k = 1, \ldots, N_h$ are known as the *global degrees of freedom* (d.o.f.). Substituting the linear combination into the variational boundary-value problem (2.17) and setting the test functions to the basis functions $v = e_{hl}, l = 1, \ldots, N_h$, we arrive at the algebraic system of equations:

$$\left\{ \begin{array}{l} \text{Find } u_{hk}, k = 1, \ldots, N_h, \text{ such that:} \\ b(u_0 + \sum_{k=1}^{N_h} u_{hk} e_{hk}, e_{hl}) = l(e_{hl}), \quad l = 1, \ldots, N_h \,. \end{array} \right. \tag{2.20}$$

Conversely, multiplying equations (2.20) by arbitrary coefficients $v_{hl}, l = 1, \ldots, N_h$, and summing up in l, we use the linearity of $b(u, v)$ in v and the linearity of $l(v)$ to show that this system of algebraic equations is, in fact, *equivalent* to approximate problem (2.17).

In principle, the approximate solution depends only upon the space V_h and it is independent of the basis functions e_{hk}. In practice, however, the choice of the basis functions affects the conditioning of the final discrete system of equations and, due to the round-off error effects, may significantly influence the actual approximate solution.

In order to simplify the notation, we shall drop now the *approximate space (mesh) index* h, remembering of course, that all quantities related to the approximate problem depend upon the index h.

Finally, using the linearity of the bilinear form $b(u, v)$ in u [4], we are led to the following

[3]We shall call u_h the *approximate solution*

[4]The considered problem is *linear*.

system of linear equations.

$$\begin{cases} \text{Find } u_k, k = 1, \ldots, N, \text{ such that:} \\ \sum_{k=1}^{N} u_k S_{kl} = L_l^{mod}, \quad l = 1, \ldots, N \end{cases} \tag{2.21}$$

Here S_{kl} denotes the *global stiffness matrix*

$$S_{kl} = b(e_k, e_l) = \int_\Omega \boldsymbol{\nabla} e_k \boldsymbol{\nabla} e_l \tag{2.22}$$

and L_l^{mod} stands for the *modified load vector*

$$\begin{aligned} L_l^{mod} &= l(e_l) - b(\tilde{u}_0, e_l) \\ &= \int_\Omega f e_l + \int_{\Gamma_2} g e_l - \int_\Omega \boldsymbol{\nabla} \tilde{u}_0 \boldsymbol{\nabla} e_l \end{aligned} \tag{2.23}$$

The array:

$$L_l = l(e_l) = \int_\Omega f e_l + \int_{\Gamma_2} g e_l \tag{2.24}$$

is called the (original) *load vector.*

Summing up, in order to determine the approximate solution, we have to:

- Select the lift $\tilde{u}_0$,
- Calculate the global stiffness matrix and the modified load vector,
- Solve system (2.21).

3 The hp Finite Element Method on Regular Meshes

The Finite Element Method is a special case of the Galerkin method and differs from other methods in the way the basis functions are constructed. Domain Ω is partitioned into disjoint subdomains called *finite elements*. In practice, the 2D elements have shapes of triangles or quadrilaterals (quads), possibly with curvilinear edges. Next, for each element K, we introduce the corresponding *shape functions* ϕ_K which eventually are *glued* into the globally defined basis functions e_k in the Galerkin method.[5] It is the construction of the basis functions that distinguishes the FEM from other Galerkin approximations.

3.1 One-dimensional hp discretization

We begin our presentation with a short summary of the corresponding 1D case first, for a detailed discussions see [7]. This will give us a chance to review the fundamental notions of the master element, the isoparametric element, and the finite element space in the simpler, one-dimensional, setting first. We shall recall the construction of the Galerkin basis functions through the element shape functions and, finally, introduce the notion of a projection-based interpolation. As a 1D equivalent of our model problem, we shall consider the two-point boundary-value problem:

$$\left\{\begin{array}{ll} \text{Find } u(x), x \in (a,b) & \text{such that} \\ -\dfrac{d^2u}{dx^2} = f & \text{in } (a,b) \\ u = u_0 & \text{at } x = a \\ \dfrac{du}{dx} = g & \text{at } x = b\,, \end{array}\right. \tag{3.1}$$

with the corresponding variational formulation:

$$\left\{\begin{array}{l} \text{Find } u(x), u(a) = u_0 \text{ such that} \\ \displaystyle\int_a^b \frac{du}{dx}\frac{dv}{dx} = \int_a^b fv \, + gv(b) \\ \quad \text{for every } v, v(a) = 0\,. \end{array}\right. \tag{3.2}$$

3.1.1 A 1D master element of order p

Geometrically, the 1D master (reference) element $\hat{K}$ coincides with the unit interval (0,1). The *element space of shape functions* $X(\hat{K})$ is identified simply as polynomials of order p,

[5] Mathematically speaking, the basis functions are *unions* of contributing element shape functions and zero function elsewhere.

i.e.

$$X(\hat{K}) = \mathcal{P}^p(\hat{K}) \tag{3.3}$$

Obviously, one can introduce many particular bases that span polynomials of order p. In the present implementation, we have selected a simple set of *hierarchical shape functions* defined as follows

$$\begin{aligned}
\hat{\chi}_1(\xi) &= 1 - \xi \\
\hat{\chi}_2(\xi) &= \xi \\
\hat{\chi}_3(\xi) &= (1-\xi)\xi \\
\hat{\chi}_l(\xi) &= (1-\xi)\xi(2\xi-1)^{l-3},\ l = 4,\dots,p+1
\end{aligned} \tag{3.4}$$

The functions admit a simple recursive formula:

$$\begin{aligned}
\hat{\chi}_1(\xi) &= 1 - \xi \\
\hat{\chi}_2(\xi) &= \xi \\
\hat{\chi}_3(\xi) &= \hat{\chi}_1(\xi) * \hat{\chi}_2(\xi) \\
\hat{\chi}_l(\xi) &= \hat{\chi}_{l-1}(\xi) * (\hat{\chi}_2(\xi) - \hat{\chi}_1(\xi)),\ l = 4,\dots,p+1
\end{aligned} \tag{3.5}$$

Note that, except for the first two linear functions, the remaining shape functions vanish at the element endpoints. For that reason they are frequently called the *bubble functions.*

Projection-based interpolation. Suppose we are given a particular scalar product $(\hat{u}, \hat{v})$ (defined on space $H^1_0(0,1)$ of functions from $H^1(0,1)$ vanishing at the end points), with the corresponding norm $\|\hat{u}\| = (\hat{u}, \hat{u})^{\frac{1}{2}}$. Let $\hat{u} \in H^1(0,1)$ be a given function. The *projection based interpolant* $\hat{u}_{hp}$ of function $\hat{u}$, is defined as the solution of the constrained minimization problem,

$$\begin{cases} \hat{u}_{hp}(0) = \hat{u}(0), \quad \hat{u}_{hp}(1) = \hat{u}(1) \\ \|\hat{u} - \hat{u}_{hp}\| \to \min\ . \end{cases} \tag{3.6}$$

Here u_{hp} is an arbitrary element from the *element space of shape functions.* In other words, we force the interpolant to coincide with the interpolated function at the end points, and then we do the best we can, in terms of getting the closest approximation in the sense of the selected norm. We call this an *interpolation* rather than projection, since the procedure is *local*, i.e. everything is done on one element only. The constrained minimization is equivalent to the linear problem,

$$\begin{cases} \hat{u}_{hp}(0) = \hat{0}, \quad \hat{u}_{hp}(1) = \hat{1} \\ (\hat{u} - \hat{u}_{hp}, \hat{v}_{hp}) = 0\ , \\ \qquad \text{for every} v_{hp} \text{ such that } v_{hp}(0) = v_{hp}(1) = 0\ . \end{cases} \tag{3.7}$$

The practical determination of the interpolant is as follows. We represent FE function $\hat{u}_{hp}$ as the sum of two contributions,

$$\hat{u}_{hp} = \hat{u}^1_{hp} + \hat{u}^2_{hp} \tag{3.8}$$

where linear interpolant $\hat{u}^1_{hp}$ is determined by the values of $\hat{u}$ at the end points,

$$\hat{u}^1_{hp}(\xi) = \hat{u}(0)\hat{\chi}_1(\xi) + \hat{u}(1)\hat{\chi}_2(\xi) \tag{3.9}$$

and the span of the middle-node shape functions,

$$\hat{u}^2_{hp}(\xi) = \sum_{j=3}^{p+1} u^2_j \hat{\chi}_j(\xi)\,, \tag{3.10}$$

is determined by solving a linear system of equations,

$$\sum_{j=3}^{p+1} u^2_j\,(\hat{\chi}_j, \hat{\chi}_k) = \Big((\hat{u} - \hat{u}^1_{hp}), \hat{\chi}_k\Big)\,, \quad k = 3, \ldots, p+1\,. \tag{3.11}$$

If we select the H^1_0 product,

$$(u, v)_{H^1_0} = \int_0^1 \frac{\partial u}{\partial \xi}\frac{\partial v}{\partial \xi}\, d\xi\,, \tag{3.12}$$

the linear system takes the following form.

$$\sum_{j=3}^{p+1} u^2_j \int_0^1 \frac{d\hat{\chi}_j}{d\xi}\frac{d\hat{\chi}_k}{d\xi}\, d\xi = \int_0^1 \frac{d(\hat{u} - \hat{u}^1_{hp})}{d\xi}\frac{d\hat{\chi}_k}{d\xi}\, d\xi, \quad k = 3, \ldots, p+1 \tag{3.13}$$

For that particular choice, we coined the term *hp interpolation* [14, 9].

3.1.2 A 1D parametric element of arbitrary order

We consider now an arbitrary closed interval $K = [x_l, x_r] \subset [a, b]$ and assume that K is the image of the master element through some map x_K:

$$\hat{K} = [0, 1] \ni \xi \to x = x_K(\xi) \in K \tag{3.14}$$

The simplest choice is an affine map which may be conveniently defined through the master element linear shape functions:

$$\begin{aligned} x_K(\xi) &= x_l\hat{\chi}(\xi) + x_r\hat{\chi}(\xi) \\ &= x_l(1 - \xi) + x_r\xi \\ &= x_l + \xi(x_r - x_l) \\ &= x_l + \xi h_K \end{aligned} \tag{3.15}$$

where $h_K = x_r - x_l$ is the element length. We assume that the map is invertible with inverse x_K^{-1} and define the *element space of shape functions* $X(K)$ as the space of compositions of inverse x_K^{-1} and functions defined on the master element.

$$\begin{aligned} X(K) &= \{\hat{u} \circ x_K^{-1},\ \hat{u} \in X(\hat{K})\} \\ &= \{u(x) = \hat{u}(\xi) \text{ where } x_K(\xi) = x \text{ and } \hat{u} \in X(\hat{K})\ \} \end{aligned} \tag{3.16}$$

Consequently, the element *shape functions* are defined as:

$$\chi_k(x) = \hat{\chi}_k(\xi) \text{ where } x_K(\xi) = x, \quad k = 1, \ldots, p+1\,. \tag{3.17}$$

Note that, in general, the shape functions are no longer polynomials, unless the map x_K is an affine map. In such a case we speak about an *affine element*. For practical reasons, it is convenient that map x_K is specified using the master element shape functions, i.e. it is a polynomial of order p:

$$x_K(\xi) = \sum_{j=1}^{p+1} x_{Kj} \hat{\chi}(\xi) \tag{3.18}$$

In such a case we talk about an *isoparametric element*. Coefficients x_{Kj} will be identified as the *geometry degrees of freedom (g.d.o.f.)*. Note that only the first two have the interpretation of coordinates of the endpoints of element K.

Projection-based interpolation. We proceed the same way as with the master element. For simplicity, we shall restrict ourselves, to the H_0^1-product only. Given function $u \in H^1(K)$, we determine its projection-based interpolant $u_{hp} \in X(K)$ by solving the following linear problem.

$$\left\{ \begin{aligned} & u_{hp}(x_l) = u(x_l), \quad u_{hp}(x_r) = u(x_r) \\ & \int_K \frac{d(u - u_{hp})}{dx} \frac{dv_{hp}}{dx}\, dx = 0, \\ & \qquad \forall v_{hp} \in X(K), v_{hp}(x_l) = v_{hp}(x_r) = 0 \end{aligned} \right. \tag{3.19}$$

Switching to the master element (change of variables + chain rule), we obtain

$$\left\{ \begin{aligned} & \hat{u_{hp}}(0) = \hat{u}(0), \quad \hat{u_{hp}}(1) = \hat{u}(1) \\ & \int_0^1 \frac{d(\hat{u} - \hat{u_{hp}})}{d\xi} \frac{d\hat{v}_{hp}}{d\xi} (\frac{dx}{d\xi})^{-1}\, d\xi = 0, \\ & \qquad \forall \hat{v}_{hp} \in X(K), \hat{v}_{hp}(0) = \hat{v}_{hp}(1) = 0 \end{aligned} \right. \tag{3.20}$$

The hat notation indicates again the composition with the element map, i.e. $\hat{u}(\xi) = u(x_K(\xi))$. Note that, compared with the *hp*-interpolation defined for the master element, we have now the extra weight $d\xi/dx$ under the integral. Therefore, only when the weight is

constant, i.e. for an affine element, the projection-based interpolation commutes with the parametric transformation, i.e.

$$\hat{u_{hp}} = \hat{u}_{hp} \,. \tag{3.21}$$

Of course, in 1D we use only affine elements, and this is not an issue, but in the multidimensional case we have to be aware of the difference.

Alternatively, we can enforce the commutativity property by assuming that the projection-based interpolation is always going to be done on the master element only. Given a function $u(x), x \in K$ defined over the element K, we compose it with the map transforming the master element into element K,

$$\hat{u}(\xi) = (u \circ x_K)(\xi) = u(x_K(\xi)), \tag{3.22}$$

and find the corresponding hp-interpolant $\hat{u}_{hp}(\xi)$ defined on master element,

$$\hat{u}_{hp}(\xi) = \sum_{j=1}^{p+1} u_j \hat{\chi}_j(\xi) \tag{3.23}$$

The final hp-interpolant over element K is defined as the composition of the master element interpolant $\hat{u}_{hp}$ with the inverse of the element map x_K,

$$u_{hp}(x) = \sum_{j=1}^{p+1} u_j \chi_j(x) \tag{3.24}$$

On the practical side, this implies that the stiffness matrix corresponding to the master element projection is (for a fixed order p) the same, and its inverse could be precomputed and stored to accelerate the computations.

Again, for the 1D case, the difference between the two discussed procedures reduces to replacing the weight $d\xi/dx$ in (3.20) with a constant.

3.1.3 1D *hp* finite element space

Let now interval (a, b) be covered with a FE mesh consisting of disjoint elements K. With each element K we associate a possibly different order of approximation $p = p_K$ and element length $h = h_K$. The element endpoints with coordinates $a = x_0 < x_1 < \ldots < x_N < x_{N+1} = b$ will be called the *vertex nodes*. We define the 1D *hp finite element space* X_h [6] as the collection of all functions that are globally continuous and whose restrictions to element K live in the element space of shape functions.

$$X_h = \{u_h(x) \,:\, u \text{ is continuous and } u|_K \in X(K) \text{ for every element } K\} \tag{3.25}$$

[6] One should really use a symbol X_{hp} as the discretization depends upon the element size $h = h_K$ and order of approximation $p = p_K$

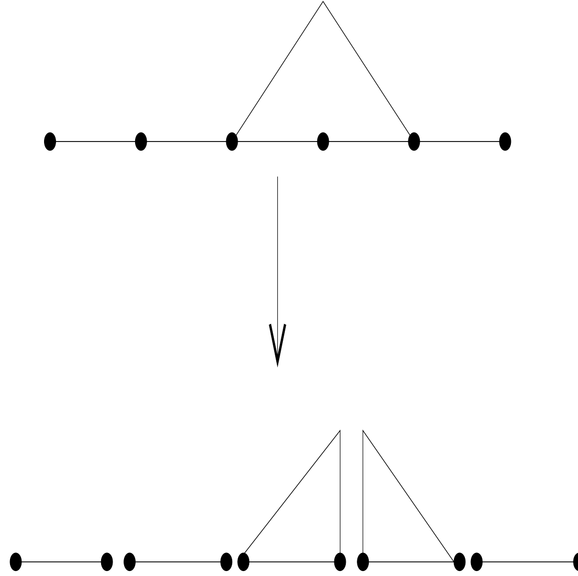


Figure 2: Construction of the vertex nodes basis functions

The global basis functions are classified into two groups:

- the vertex nodes basis functions,
- the middle nodes ('bubble') basis functions.

The basis function corresponding to a vertex node x_k is defined as the union of the two adjacent element shape functions corresponding to the common vertex and zero elsewhere. The construction is illustrated in Fig 2. The construction of the middle nodes basis functions is much easier. As the element middle nodes shape functions vanish at the element endpoints, we need simply to extend them only by the zero function elsewhere. The *support*[7] of a vertex node basis function extends over the two adjacent elements, whereas for a bubble function it is restricted just to one element.

Finally, the continuity of the approximation at the vertex nodes allows us to introduce the concept of the global *hp*-interpolation. Given a continuous function $u(x), x \in [a, b]$, we define its *hp*-interpolant as the union of the contributing elements *hp*-interpolants:

$$u_{hp}(x) = u_{hp}^K(x) \text{ where } x \in K \tag{3.26}$$

where u_{hp}^K denotes the *hp*-interpolant over element K. For linear (first order) elements, the *hp* interpolation reduces to the standard Lagrange interpolation. We emphasize that the interpolation is done *locally*, separately over each element.

[7] The support of a function is defined as the closure of a set over which the function takes on values different from zero.

3.2 Quadrilateral and triangular master elements with variable order of approximation

3.2.1 Quadrilateral master element

The element occupies the standard reference square, $\hat{K} = [0,1]^2$. The element space of shape functions $X(\hat{K})$ is a subspace of $Q^{(p_h,p_v)}$, i.e. polynomials that are of order p_h in ξ_1 and order p_v with respect ξ_2. In order to be able to match in mesh elements of different orders, we associate with each of the element edges a possibly different order of approximation $p_i, i = 1,\ldots,4$, with the assumption that

$$p_1, p_3 \le p_h \text{ and } p_2, p_4 \le p_v. \tag{3.27}$$

The element space of shape functions is now identified as the subspace of $Q^{(p_h,p_v)}$, consisting of functions whose restrictions to edge $\hat{e}_i$ reduce to polynomials of (smaller) degree p_i,

$$X(\hat{K}) = \{\hat{u} \in Q^{(p_h,p_v)} \ : \ \hat{u}|_{\hat{e}_i} \in \mathcal{P}^{p_i}(\hat{e}_i)\} \tag{3.28}$$

The element shape functions are constructed as *tensor products* of the 1D shape functions defined above. It is convenient to group them into subsets associated with the element vertices, edges and the element interior. Shape functions belonging to the same group have the same *connectivity*, i.e. the corresponding basis functions are built using the same logic. For instance, all basis functions associated with an interelement edge "consist of" two shape functions corresponding to the two adjacent elements and the common edge, whereas basis functions corresponding to an interior of an element "consist" of just one contributing shape function.

To facilitate the communication, we introduce the notion of abstract *nodes* that we associate with the element, see Fig. 3.

- four vertex nodes: $\hat{\boldsymbol{a}}_j, j = 1,\ldots,4$,
- four mid-edge nodes: $\hat{\boldsymbol{a}}_j, j = 5,\ldots,8$,
- the middle node: $\hat{\boldsymbol{a}}_9$.

With each of the nodes, we associate the corresponding order of approximation, $p = 1$ for the vertex nodes, the edge order p_i for the i-th mid-edge node, and the *anisotropic* order (p_h, p_v) for the middle node. The corresponding shape functions are now defined as tensor products of the 1D shape functions: $\hat{\chi}_i(\xi)$

- one bilinear shape function for each of the vertex nodes,

$$\begin{aligned}
\hat{\phi}_1(\xi_1,\xi_2) &= \hat{\chi}_1(\xi_1)\hat{\chi}_1(\xi_2) \\
&= (1-\xi_1)(1-\xi_2) \\
\hat{\phi}_2(\xi_1,\xi_2) &= \hat{\chi}_2(\xi_1)\hat{\chi}_1(\xi_2) \\
&= \xi_1(1-\xi_2) \\
\hat{\phi}_3(\xi_1,\xi_2) &= \hat{\chi}_2(\xi_1)\hat{\chi}_2(\xi_2) \\
&= \xi_1\xi_2 \\
\hat{\phi}_4(\xi_1,\xi_2) &= \hat{\chi}_1(\xi_1)\hat{\chi}_2(\xi_2) \\
&= (1-\xi_1)\xi_2 \, ,
\end{aligned} \tag{3.29}$$

- $p_i - 1$ shape functions for each of the mid-edge nodes,

$$\begin{aligned}
\hat{\phi}_{5,j}(\xi_1,\xi_2) &= \hat{\chi}_{2+j}(\xi_1)\hat{\chi}_1(\xi_2) \\
&= (1-\xi_1)\xi_1(2\xi_1-1)^{j-1}(1-\xi_2), \quad j=1,\ldots,p_1-1 \\
\hat{\phi}_{6,j}(\xi_1,\xi_2) &= \hat{\chi}_2(\xi_1)\hat{\chi}_{2+j}(\xi_2) \\
&= \xi_1(1-\xi_2)\xi_2(2\xi_2-1)^{j-1}, \quad j=1,\ldots,p_2-1 \\
\hat{\phi}_{7,j}(\xi_1,\xi_2) &= \hat{\chi}_{2+j}(1-\xi_1)\hat{\chi}_2(\xi_2) \\
&= (1-\xi_1)\xi_1(1-2\xi_1)^{j-1}\xi_2, \quad j=1,\ldots,p_3-1 \\
\hat{\phi}_{8,j}(\xi_1,\xi_2) &= \hat{\chi}_1(\xi_1)\hat{\chi}_{2+j}(1-\xi_2) \\
&= (1-\xi_1)(1-\xi_2)\xi_2(1-2\xi_2)^{j-1}, \quad j=1,\ldots,p_4-1 \, ,
\end{aligned} \tag{3.30}$$

- $(p_h - 1)(p_v - 1)$ *bubble shape functions* for the middle node,

$$\begin{aligned}
\hat{\phi}_{9,ij}(\xi_1,\xi_2) &= \hat{\chi}_{2+i}(\xi_1)\hat{\chi}_{2+j}(\xi_2) \\
&= (1-\xi_1)\xi_1(2\xi_1-1)^{i-1}(1-\xi_2)\xi_2(2\xi_2-1)^{j-1} \\
&\quad i=1,\ldots,p_h-1, \; j=1,\ldots,p_c-1 \, .
\end{aligned} \tag{3.31}$$

REMARK 1 Please note the difference between the first and third, and the second and fourth edges. When restricted to an element edge, the 2D shape functions reduce to 1D shape functions, with the local coordinate ξ oriented counterclockwise. Thus, the parametrization

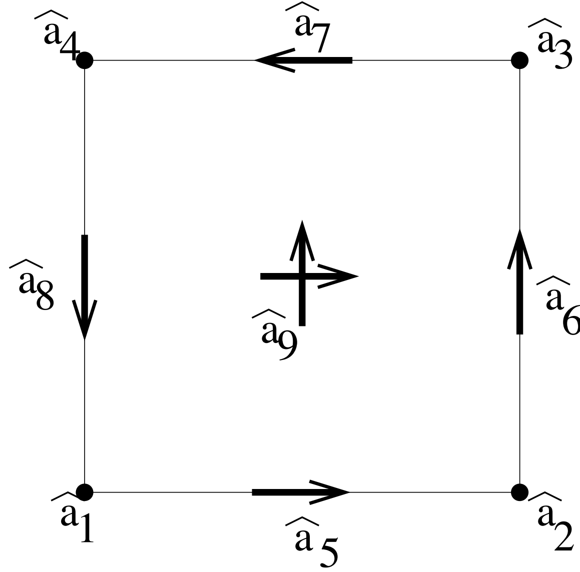


Figure 3: Quadrilateral master element

of the four element edges is:

$$
\begin{array}{ll}
\xi_1 = \xi & \xi_2 = 0 \\
\xi_1 = 1 & \xi_2 = \xi \\
\xi_1 = 1 - \xi & \xi_2 = 1 \\
\xi_1 = 0 & \xi_2 = 1 - \xi
\end{array}
\tag{3.32}
$$

∎

We emphasize the abstract character of the "nodes" introduced above. They are merely an abstraction for the element vertices, edges, and its interior, and should not be confused with the classical notion of the Lagrange or Hermite nodes.

Finally, we introduce the notion of the *hp-interpolation procedure*. The idea is a direct generalization of the 1D *hp*-interpolation defined in the previous subsection. Given a continuous function $\hat{u}(\xi_1, \xi_2)$ defined over the master element, we define its *hp*-interpolant as the sum of three contributions:

$$
\hat{u}_{hp} = \hat{u}^1_{hp} + \hat{u}^2_{hp} + \hat{u}^3_{hp}
\tag{3.33}
$$

where

- $\hat{u}^1_{hp}$ is the standard bilinear interpolant corresponding to the vertex nodes:

$$
\hat{u}^1_{hp}(\boldsymbol{\xi}) = \sum_{i=1}^{4} \hat{u}(\hat{\boldsymbol{a}}_i)\hat{\phi}_i(\boldsymbol{\xi}) \, ,
\tag{3.34}
$$

- $\hat{u}^2_{hp}$ is obtained by projecting the difference of the original function and its bilinear interpolant onto the span of shape functions associated with the mid-edge nodes. More precisely,

$$\hat{u}^2_{hp} = \sum_{i=1}^{4} \hat{u}^{2i}_{hp} \tag{3.35}$$

where

$$\hat{u}^{2i}_{hp}(\boldsymbol{\xi}) = \sum_{j=1}^{p_i-1} u_{ij}\hat{\phi}_{4+i,j}(\boldsymbol{\xi}) \tag{3.36}$$

and the coefficients u_{ij} are determined solving the system of equations:

$$\sum_{j=1}^{p_i-1} u_{ij} \int_{\hat{e}_i} \frac{d\hat{\phi}_{4+i,j}}{ds}\frac{d\hat{\phi}_{4+i,k}}{ds}\, ds = \int_{\hat{e}_i} (\frac{d\hat{u}}{ds} - \frac{d\hat{u}^1_{hp}}{ds})\frac{d\hat{\phi}_{4+i,k}}{ds}\, ds\,, \tag{3.37}$$

- $\hat{u}^3_{hp}$ is obtained by projecting the difference $\hat{u} - \hat{u}^1_{hp} - \hat{u}^2_{hp}$ onto the span of bubble shape functions associated with the middle node,

$$\begin{cases} \hat{u}^3_{hp} = \sum_{i=1}^{p_h-1}\sum_{j=1}^{p_v-1} u_{ij}\hat{\phi}_{9,ij} \\ \sum_{i=1}^{p_h-1}\sum_{j=1}^{p_v-1} u_{ij}\int_{\hat{K}} \boldsymbol{\nabla}\hat{\phi}_{9,ij}\boldsymbol{\nabla}\hat{\phi}_{9,kl}\, d\boldsymbol{\xi} = \int_{\hat{K}} \boldsymbol{\nabla}(\hat{u} - \hat{u}^1_{hp} - \hat{u}^2_{hp})\boldsymbol{\nabla}\hat{\phi}_{9,kl}\, d\boldsymbol{\xi} \\ \quad k = 1,\ldots,p_h-1,\, l = 1,\ldots,p_v-1\,. \end{cases} \tag{3.38}$$

The hp-interpolant can be viewed as a Galerkin solution of the Poisson equation with the Dirichlet boundary data consisting of the one-dimensional hp-interpolant of the original function.

As in the 1D case, the products for the edges and element interior could be selected in a different way, resulting in a different projection-based interpolation procedure.

3.2.2 Triangular master element

The element occupies the standard unit right triangle illustrated in Fig. 4. It has seven nodes: three vertex nodes $\hat{\boldsymbol{a}}_i, i = 1,2,3$, three mid-edge nodes $\hat{\boldsymbol{a}}_i, i = 4,5,6$ and the middle node $\hat{\boldsymbol{a}}_7$. The element space of shape functions consists of polynomials of order p whose restrictions to element edges $\hat{e}_i$ reduce to polynomials of order p_i,

$$X(\hat{K}) = \{\hat{u} \in \mathcal{P}^p(\hat{K}) \;:\; \hat{u}|_{\hat{e}_i} \in \mathcal{P}^{p_i}(\hat{e}_i), \quad i = 1,2,3\} \tag{3.39}$$

where, as in the quad element, we assume that

$$p_1, p_2, p_3 \le p. \tag{3.40}$$

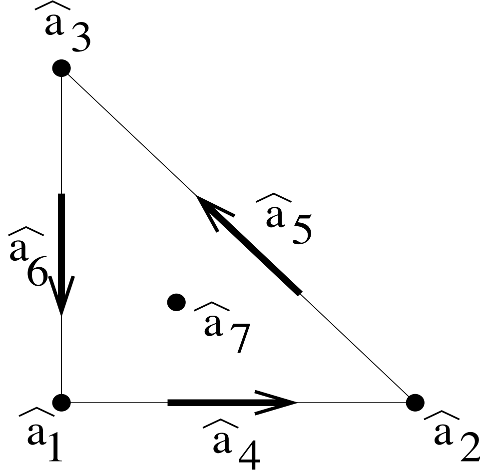


Figure 4: Triangular master element

The element shape functions are defined using area (affine, barycentric) coordinates:

$$\lambda_1 = 1 - \xi_1 - \xi_2,\ \lambda_2 = \xi_1,\ \lambda_3 = \xi_2. \tag{3.41}$$

We introduce:

- the standard linear shape functions associate with vertex nodes,

$$\hat{\phi}_i = \lambda_i, \quad i = 1, 2, 3\ , \tag{3.42}$$

- $p_i - 1$ shape functions associated with each mid-edge node,[8]

$$\hat{\phi}_{i,j} = \lambda_i \lambda_{i+1} (\lambda_{i+1} - \lambda_i)^{j-1}, \quad j = 1, \ldots, p_i - 1\ , \tag{3.43}$$

- $(p-2)(p-1)/2$ bubble shape fucntions associated with the middle node,

$$\begin{aligned} &\lambda_1 \lambda_2 \lambda_3 \\ &\lambda_1 \lambda_2^2 \lambda_3, \lambda_1, \lambda_2 \lambda_3^2 \\ &\ldots \\ &\lambda_1 \lambda_2^{p-2} \lambda_3, \lambda_1 \lambda_2^{p-3} \lambda_3^2, \ldots, \lambda_1 \lambda_2 \lambda_3^{p-2}. \end{aligned} \tag{3.44}$$

The hp-interpolation procedure is identical with that for the quad element. The hp-interpolant of a continuous function $\hat{u}$ is again defined as the sum of three contributions,

$$\hat{u}_{hp} = \hat{u}_{hp}^1 + \hat{u}_{hp}^2 + \hat{u}_{hp}^3\ , \tag{3.45}$$

[8]Mid-edge nodes are denumerated modulo 3, i.e., for $i = 3$, $i + 1 = 1$.

where $\hat{u}^1_{hp}$ is the standard linear interpolant using the vertex values, $\hat{u}^2_{hp}$ is obtained by projecting the difference of the original function and its linear interpolant onto the span of mid-edge nodes shape functions, and $\hat{u}^3_{hp}$ is defined as the H^1_0-projection of the difference $\hat{u} - \hat{u}^1_{hp} - \hat{u}^2_{hp}$ onto the span of the middle node (bubble) shape functions.

3.3 Parametric element

We use the standard procedure to define parametric (deformed) quads and triangles. Given a bijective map

$$\boldsymbol{x}_K : \hat{K} \to K \tag{3.46}$$

from either master quad or triangle onto an element K, we define the element space of shape functions as the collection of compositions of inverse $\boldsymbol{x}_K^{-1}$ and the master element shape functions,

$$X(K) = \{u = \hat{u} \circ \boldsymbol{x}_K^{-1} \ : \ \hat{u} \in X(\hat{K})\}. \tag{3.47}$$

Accordingly, the element shape functions are defined as:

$$\phi_i(\boldsymbol{x}) = \hat{\phi}_i(\boldsymbol{\xi}), \text{ where } \boldsymbol{x}_K(\boldsymbol{\xi}) = \boldsymbol{x}. \tag{3.48}$$

For each element K, we shall speak about its vertex, mid-edge, and middle nodes, understood again simply as an abstraction for the element vertices, edges and interior.

Projection based interpolation. For each of master element edges, we introduce the corresponding parametrization $\boldsymbol{\xi}(\xi)$, consistent with the *local* element edge orientation. For the master square element, these are:

$$\begin{cases} \xi_1 = \xi \\ \xi_2 = 0 \end{cases} \quad \begin{cases} \xi_1 = 1 \\ \xi_2 = \xi \end{cases} \quad \begin{cases} \xi_1 = 1 - \xi \\ \xi_2 = 1 \end{cases} \quad \begin{cases} \xi_1 = 0 \\ \xi_2 = 1 - \xi \end{cases} , \tag{3.49}$$

and for the triangular master element, we have:

$$\begin{cases} \xi_1 = \xi \\ \xi_2 = 0 \end{cases} \quad \begin{cases} \xi_1 = 1 - \xi \\ \xi_2 = \xi \end{cases} \quad \begin{cases} \xi_1 = 0 \\ \xi_2 = 1 - \xi \end{cases} . \tag{3.50}$$

The corresponding parametrizations for the parametric element edges are now defined as compositions of the parametrizations for the master element and the parametric maps,

$$\xi \to \boldsymbol{x}_K(\boldsymbol{\xi}(\xi)) . \tag{3.51}$$

The H^1_0 product for an element edge e transforms then as follows.

$$\int_e \frac{du}{ds}\frac{dv}{ds}\, ds = \int_0^1 \frac{d\hat{u}}{d\xi}\frac{d\hat{v}}{d\xi}\left(\frac{ds}{d\xi}\right)^{-1} d\xi \tag{3.52}$$

where

$$\frac{ds}{d\xi} = \sqrt{\left(\frac{dx_1}{d\xi}\right)^2 + \left(\frac{dx_2}{d\xi}\right)^2} . \tag{3.53}$$

Thus again, unless $ds/d\xi$ is constant, i.e. the edge is rectilinear, H_0^1-projection done on the deformed element edge will yield different result from the projection done on the master element edge.

The difference becomes even more pronounced in the transformation rule for the H_0^1 product done over the whole element [9],

$$\int_K \frac{\partial u}{\partial x_k}\frac{\partial v}{\partial x_k}\, d\boldsymbol{x} = \int_0^1\int_0^1 g_{ij}(\boldsymbol{\xi})\frac{\partial \hat{u}}{\partial \xi_i}\frac{\partial \hat{v}}{\partial \xi_j}\, d\xi_1 d\xi_2 \tag{3.54}$$

where metric g_{ij} is given by:

$$a_{ij} = \frac{\partial \xi_i}{\partial x_k}\frac{\partial \xi_j}{\partial x_k} j , \tag{3.55}$$

with j detoting the jacobian of the parametric element,

$$j = \det\left(\frac{\partial x_i}{\partial \xi_j}\right) . \tag{3.56}$$

This time, metric g_{ij} reduces to a constant (diagonal matrix) only, if the parametric element is a square itself. Even for a rectangular element with edges parallel to coordinate axes x_i, we get different weights in the master element coordinates.

Consequently, for general meshes, projection-based interpolation done on master and deformed elements yields different results. If we choose to work with the projections on the deformed elements, we still perform the projections (interpolation) on the master 1D and 2D elements, but with edge and element dependent metrics $dxi/ds, g_{ij}$.

We restrict ourselves to the *isoparametric* deformations only, i.e. we assume that the map $\boldsymbol{x}_K$ lives in the corresponding space of shape functions for the master element. More precisely,

$$\boldsymbol{x}_K(\boldsymbol{\xi}) = \sum_j \boldsymbol{x}_{Kj}\hat{\phi}_j(\boldsymbol{\xi}) \tag{3.57}$$

Here j is the index denumerating shape functions of the master element, and $\boldsymbol{x}_{Kj}$ denotes the corresponding *vector-valued* geometry degrees of freedom, of dimension two for planar problems, or dimension three for meshes in $\mathbb{R}^3$ [10]. Note that only the degrees of freedom corresponding to the vertex nodes have the interpretation of the vertex nodes coordinates. If the map $\boldsymbol{x}_K$ is affine (i.e. all geometry dof except the vertex nodes coordinates are zero), the parametric element is called *affine element*, and the corresponding shape functions are polynomials.

[9] We use the summation rule, i.e. repeated indices indicate summation over them

[10] For instance, in Boundary Element Methods calculations

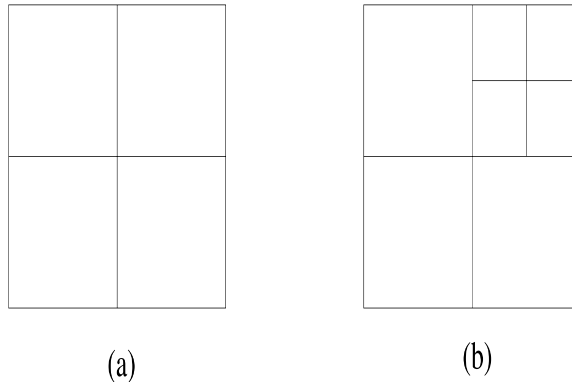


Figure 5: Examples of (a) regular and (b) irregular meshes

3.4 Finite element space. Construction of basis functions

Let domain Ω be partitioned into triangular and quadrilateral elements. We shall assume that no approximation of the boundary is necessary, i.e. that the boundary is at most polynomial and it can be represented exactly with isoparametric elements of sufficiently high order. We shall also assume that the mesh is *regular* in the sense that the intersection of any two elements is either empty, or it consists of a single vertex node or a (whole) common edge. Otherwise the mesh will be called *irregular*. Irregular meshes appear naturally as a result of h refinements discussed in the next section. The difference between regular and irregular meshes is illustrated in Fig. 5. We now define the finite element space X_h as the collection of continuous functions whose restrictions to elements K live in the elements spaces of shape functions,

$$X_h = \{u \, : \, u \text{ is continuous an } u|_K \in X(K), \text{ for every element } K \text{ in the mesh}\} \qquad (3.58)$$

The global basis functions will be obtained by "gluing together" element shape functions. The procedure is illustrated in Fig. 6. The restriction of a basis function to a contributing element K reduces to one of the element shape functions, premultiplied possibly with a *sign factor* $c_{i,K}$,

$$e_i|_K(\boldsymbol{x}) = c_{i,K}\phi_{k,K}(\boldsymbol{x}). \qquad (3.59)$$

The *connectivity* $i = i(k, K)$ equals the number of the global basis function corresponding to element K and its k-th shape function. The necessity of introducing the sign factor results from the fact that we have decided to parametrize locally element sides *always counterclockwise*. Consequently, the edge shape functions of odd degree corresponding to two neighboring elements *do not match each other*, and one of them has to be premultiplied by a -1 factor.

The sign factor issue is handled by introducing the notion of orientation for the element edges (mid-edge nodes). We assume that each edge comes with its *global orientation*. Now,

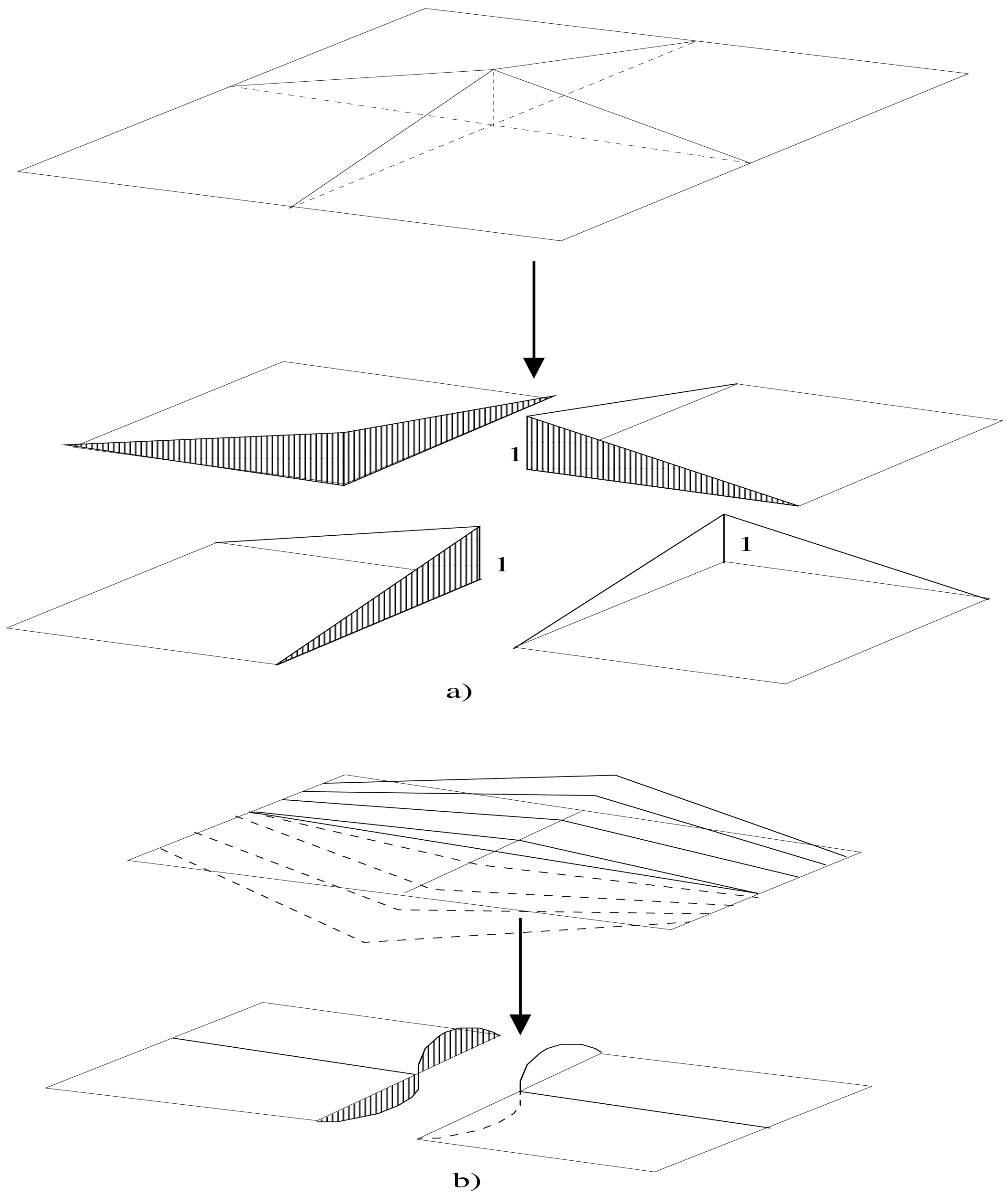


Figure 6: Construction of basis functions on regular meshes for (a) a vertex node, (b) a mid-edge node. Notice the necessity of introducing the sign factor for the mid-edge node basis function.

each of element edges has also its own *local orientation*, determined by the numbering of the nodes (element local system of coordinates), and the counterclockwise rule. If the local orientation of an element edge is now consistent with the global one, the corresponding sign factor will be 1, otherwise it will be -1. These sign factors for element edges are part of the connectivity information and, for initial mesh elements, we store them as a separate item in the data structure arrays. In the code, we set the global orientation for initial mesh element edges by following the standard rule, and assume that the *global orientation of an edge* follows from the numbering of its vertex endpoints nodes - the edge will always be oriented from the vertex node with a smaller number to the vertex node with a bigger number.

The sign factors are necessary only for the mid-edge shape functions [11], for the vertex and the middle nodes shape functions they are always equal to one.

The calculation of the global stiffness matrix (2.22) looks now as follows:

$$S_{ij} = b(e_i, e_j) = \int_\Omega \boldsymbol{\nabla} e_i \boldsymbol{\nabla} e_j = \sum_K c_{i,K} c_{j,K} b_K(\psi_k, \psi_l) \tag{3.60}$$

where the sum extends over all contributing elements, $i = i(k, K), j = j(l, K)$ are the connectivities, and

$$S_{kl,K} = b_K(\psi_k, \psi_l) = \int_K \boldsymbol{\nabla}\psi_k \boldsymbol{\nabla}\psi_l \, d\boldsymbol{x} \tag{3.61}$$

is the *element stiffness matrix.*

The global load vector (2.24) is evaluated using the same strategy,

$$L_i = l(e_i) = \int_\Omega f e_i \, d\boldsymbol{x} + \int_{\Gamma_2} g e_i \, ds = \sum_K c_{i,K} l_K(\psi_k) \tag{3.62}$$

where

$$L_{k,K} = l_K(\psi_k) = \int_K f\psi_k \, d\boldsymbol{x} + \int_{\partial K \cap \Gamma_2} g\psi_k \, ds \tag{3.63}$$

is the *element load vector.*

The actual algorithm of assembling the global matrices is as follows.

initiate global matrices S_{ij} and L_i with zeros
for each element K in the mesh
    calculate element stiffnes matrix $S_{kl,K}$ and load vector $L_{k,K}$
    for each local d.o.f. k
        determine connectivity $i = i(k, K)$
        accumulate for the global load vector:
        $L_i = L_i + c(i, K) * L_{k,K}$

[11] Actually only for those of an odd degree

for each local d.o.f. l

determine connectivity $j = j(l, K)$

accumulate for the global stiffness matrix:

$S_{ij} = S_{ij} + c(i,K) * c(j,K) * S_{kl,K}$

end of the second loop through element local d.o.f.

end of the first loop through element local d.o.f.

end of the loop through elements

3.5 Data structure in FORTRAN 90

We introduce three *user-defined structures* [13], p.353 (see *module/data_structure2D*):

- type *initial mesh element*.
- type *vertex*,
- type *node*,

The attributes of an initial mesh element are:

- element type (a character variable indicating triangle or a quad),
- integer array *nodes* containing the seven (triangles) or nine (quads) node numbers for the element, listed in the order: vertices, mid-edge nodes, and the middle node,
- integer orient containing orientations for the element edges, packed into one integer,
- integer array *neig* containing numbers of three (triangles) or four (quads) neighbors of the element,
- integer *bcond* encoding boundary conditions flags for the element edges, e.g. for a quad element we have:

$$bcond = bc4 * 1000 + bc3 * 100 + bc2 * 10 + bc1 \tag{3.64}$$

  where $bc1, \ldots, bc4$ are one-digit flags for the element edges.

- an extra integer *geom_interf* for interfacing with the Geometrical Modeling Package (GMP).

The attributes of a vertex node include: a boundary condition flag, a GMP interface flag, and two arrays *coord*, and *zdofs*, containing geometry and actual d.o.f.

The attributes of a node include: node type (a character indicating whether the node is a mid-edge, a triangle, or a quad middle node), integer order of approximation, integer boundary condition flag, and two real arrays, *coord*, containing geometrical degrees of freedom, and $zdofs$, containing the "actual" degrees of freedom. Both the geometry and actual d.o.f. are allocated dynamically, dependent upon the order of approximation for the node. Note the following details:

- the integer specifying order for a quad middle node is actually a nickname defined as

$$order = p_h * 10 + p_v \tag{3.65}$$

  where p_h, p_v are the *horizontal* and the *vertical* orders of approximation for the element;

- both *coord* and $zdofs$ are defined as arrays with two indices. The first index of *coord* corresponds to the number of coordinates and varies between 1 and $NDIMEN = 2, 3$. The first index of array $zdofs$ indicates a component of the solution, and for the Laplace equation being discussed, it is always equal one.

The remaining attributes of higher order nodes (father and sons), and vertex nodes (father), are related to h-refinements, and will be discussed in the next section.

The entire information about a (initial) mesh is now stored in three allocatable arrays, ELEMS, NVERS and NODES, as declared in the data structure module. The module also includes a declaration for a number of integer attributes of the mesh like number of elements in the initial mesh, current mesh, number of vertex and non-vertex nodes, etc.

3.6 The element routine

The purpose of the element routine is to evaluate the element stiffness matrix and the element load vector. We use the standard algorithm:

initiate element matrices $S_{ij} = S_{ij,K}$ and $L_i = L_{i,K}$ with zeros
get element geometry degrees of freedom $xnod_{ik}$,
Step1: evaluate the element integrals:
for each integration point $\boldsymbol{\xi} = \boldsymbol{\xi}_l$,
    evaluate shape functions $\hat{\phi}_k$ and their derivatives wrt master element coordinates $\frac{\partial \hat{\phi}_k}{\partial \xi_j}$
    initiate the physical coordinates of the integration point x_i
    and their derivatives with respect to the master element coordinates $\frac{\partial x_i}{\partial \xi_j}$ with zeros,
    for each shape function k,
        accumulate for x_i and $\frac{\partial x_i}{\partial \xi_j}$:

$x_i = x_i + xnod_{ik} * \hat{\phi}_k(\xi)$
$\frac{\partial x_i}{\partial \xi_j} = \frac{\partial x_i}{\partial \xi_j} + xnod_{ik} * \frac{\partial \hat{\psi}_k}{\partial \xi_j}(\xi)$
end of loop through the element shape functions
evaluate jacobian $\det(\frac{\partial x_i}{\partial \xi_j})$ and inverse derivatives $\frac{\partial \xi_j}{\partial x_i}$
use the chain formula to evaluate the derivatives of the shape functions wrt the physical coordinates:

$$\frac{\partial \phi_k}{\partial x_i} = \sum_{j=1}^{2} \frac{\partial \hat{\phi}_k}{\partial \xi_j} \frac{\partial \xi_j}{\partial x_i}$$

evaluate the weight w_l (actual Gaussian weight times the jacobian)
get the right-hand side of the equation $f(\boldsymbol{x})$
for each shape function ϕ_{k2},
accumulate for the load vector:
$L_{k2} = L_{k2} + f(\boldsymbol{x}) * \phi_{k2}(\boldsymbol{x}) * w_l$
for each shape function ϕ_{k1},
accumulate for the stiffness matrix:

$$S_{k1,k2} = S_{k1,k2} + \sum_{i=1}^{2} \frac{\partial \phi_{k1}}{\partial x_i} \frac{\partial \phi_{k2}}{\partial x_i} * w_l$$

end of the second loop through the element shape functions
end of the first loop through the element shape functions
end of loop through the integration points
Step 2: evaluate the boundary integrals:
for each element edge e on the Neumann boundary
establish a parametrization $\xi_j = \xi_j(\xi)$
for each integration point $\xi = \xi_l$,
evaluate shape functions $\hat{\phi}_k$ and their derivatives wrt the master element coordinates
evaluate physical coordinates x_i and their derivatives wrt master element coordinates $\frac{\partial x_i}{\partial \xi_j}$ (see the algorithm above)
use the chain formula to evaluate the derivatives wrt parameter ξ:

$$\frac{dx_i}{d\xi} = \sum_{j=1}^{2} \frac{\partial x_i}{\partial \xi_j} \frac{d\xi_j}{d\xi}$$

evaluate the line integral jacobian:

$$\frac{d\boldsymbol{x}}{d\xi} = \sqrt{\sum_{i=1}^{2} (\frac{dx_i}{d\xi})^2}$$

evaluate the weight w_l (actual 1D Gaussian weight times the jacobian above),
get the Neumann data $g(\boldsymbol{x})$,
for each shape function ϕ_k,
accumulate for the load vector:
$L_k = L_k + g(\boldsymbol{x}) * \phi_k(\boldsymbol{x}) * w_l$

end of loop through the element shape functions
end of loop through the integration points
end of loop through the element edges

3.7 Modified element. Imposing Dirichlet boundary conditions.

As discussed earlier, assembling of element matrices into the global matrices is accompanied for some shape functions by a change of sign. It is convenient to separate these two operations, the sign change, and the assembling procedure, from each other, by introducing the notion of a *modified element*.[12]

The information about the modified element includes:

- a list of element vertex nodes: $Nod1(i), i = 1, \ldots, Nvert$,
- a list of element non-vertex (mid-edge and middle) nodes: $Nod2(i), i = 1, \ldots, Nrnod$,
- the corresponding number of shape functions (d.o.f) per node: $Ndof2(i), i = 1, \ldots, Nrnod$,
- the modified element load vector: $Bload(k), k = 1, \ldots, Nrdof$,
- the modified element stiffness matrix: $Astiff(k,l),\ k,l = 1, \ldots, Nrdof$.

The shape functions are ordered following the order of nodes and the corresponding order of shape functions for each of the nodes. The total number of shape functions (degrees of freedom) for the element, $Nrdof$, is obtained by summing up the numbers of shape functions for each node:

$$Ndof = Nvert + \sum_{i=1}^{Nrnod} Ndof2(i) \tag{3.66}$$

For regular meshes discussed in this section, the lists of vertex and non-vertex nodes are organized in the same order as they are listed in the data structure array $ELEMS(nel)\%nodes$. The corresponding matrices are obtained by multiplying the local matrices by the sign factors discussed earlier. Let ϕ_k denote k-th shape function of element K, and let e_i be the corresponding basis function. Recall the definition of the sign factor:

$$c(k) = \begin{cases} 1, & \text{if } e_i|_K = \phi_k \\ -1, & \text{if } e_i|_K = -\phi_k \end{cases} \tag{3.67}$$

The calculation of the modified element matrices is done in routine *constrs/celem* following the algorithm:

[12] The need for this separation will be more clear for h-adapted meshes discussed in the next section.

```
initiate element shape function (d.o.f) counter k = 0
for each vertex node
    k = k + 1
    c(k) = 1
end of loop through vertex nodes
for each mid-edge node i
    for each corresponding shape function j
        k = k + 1
        if j is odd then
            c(k) = 1
        else
            check for the edge vertex nodes numbers nvert1, nvert2
            if nvert1 > nvert2 then
                c(k) = 1
            else
                c(k) = -1
            endif
        endif
    end of loop through the mid-edge node shape functions
end of loop through the mid-edge nodes
for each middle node shape function (d.o.f)
    k = k + 1
    c(k) = 1
end of loop through the middle node shape functions
for each shape function k2
    Bload(k2) = Bloc(k2) * c(k2)
    for each shape function k1
        Astiff(k1, k2) = Aloc(k1, k2) * c(k1) * c(k2)
    end of the second loop through the element shape functions
end of the first loop through the element shape functions
```

Here $Bloc, Aloc$ denote the element (local) load vector and stiffness matrix.

Imposing Dirichlet boundary conditions. We impose the Dirichlet boundary conditions *always* on the modified element matrices. We assume that the data u_0 has been represented exactly by restrictions (traces) of FE basis functions to Dirichlet boundary Γ_1. More precisely, if e_i, $i = 1, \ldots, N_1$, denote all basis functions corresponding to nodes that lie

on the Dirichlet boundary Γ_1, we assume that

$$u_0 = \sum_{i}^{N_1} u_0^i e_i|_{\Gamma_1} \, . \tag{3.68}$$

The linear combination of the same basis functions (defined on the whole domain Ω) provides then a natural extension (lift) of the boundary data to the whole domain and it is used to calculate the *modified element load vectors* discussed in the previous section.[13] In practice, we do not eliminate the Dirichlet d.o.f. from calculations but rather modify the modified element stiffness matrix and load vector to accommodate for the boundary conditions. The modification is done in routine *laplace/bcmod* and follows the standard algorithm.

```
for each (modified) element shape function j
    skip if not on the Dirichlet boundary
    modify the load vector:
    for each k
        if k = j then
            Bload(k) = u_0^j
        else
            Bload(k) = Bload(k) - Astiff(k,j) * u_0^j
        endif
    end of loop through the element d.o.f.
    modify the stiffness matrix:
    for each k2
        for each k1
            if k1 = k2 then
                Astif(k1,k2) = 1
            else
                Astif(k1,k2) = 0
            endif
        end of the second loop through the element d.o.f.
    end of the first loop through the element d.o.f.
end of loop through the element shape functions
```

The proposed modification does not change the logic of elements adjacent to the Dirichlet boundary (they are treated just the same way as other elements in the global assembling procedure), and does not destroy the symmetry of the element stiffness matrix. Note that a solver will simply reset the Dirichlet d.o.f. to the original values.

[13]More precisely, we are modifying the modified element load vector, the nomenclature is rather confusing.

Assembling of global matrices. We first have to establish a global denumeration for all basis functions. In principle, one could follow the numbering of the nodes and then the numbering of the corresponding nodal shape functions. However, in general, such a denumeration may not be optimal from the point of view of minimizing the bandwidth. Besides, as a result of refinements/unrefinements of the mesh, nodes may no longer be numbered using consecutive integers. We shall adopt the philosophy that we are always given an *order of elements*. One such order, called the *natural order of elements* is provided by routine $datstrs/nelcon$ and will be discussed in the next section. Given the order of elements and an order of nodes for each element, we can define the *natural order of nodes*. Finally, following the order of shape functions (d.o.f.) for each of the nodes, we can define the *natural order of d.o.f.*.

On the practical level, we may introduce an extra attribute for each node *nod* in the mesh, say $NPOINT(nod)$ equal to the number of the first corresponding d.o.f. in the *global*, natural order of d.o.f. The pointers are determined in the following way:

initiate array $NPOINT$ with zeros
initiate d.o.f. counter $np = 0$
for each element nel
    for each element node i
        get the global node number: $nod = ELEMS(nel)\%nodes(i)$
        skip if $NPOINT(nod) \neq 0$, i.e. the node has already been visited
        set the counter for the first d.o.f. of the node: $NPOINT(nod) = np + 1$
        determine the number of d.o.f. $ndof$ corresponding to the node
        update the counter: $np = np + ndof$
    end of loop through element nodes
end of loop through elements

Once the bijection between the local d.o.f. and the global denumeration of d.o.f. (the connectivities) has been established, the assembling procedure follows the standard algorithm.

for each element nel in the mesh
    calculate element *local* matrices $Bloc, Aloc$
    calculate the modified element matrices $Bload, Astiff$
    impose the Dirichlet boundary conditions
    establish element d.o.f. connectivities:
    initiate the element d.o.f. counter: $k = 0$
    for each element node i

get the global node number: $nod = ELEMS(nel)\%nodes(i)$
for each nodal d.o.f. j
update the element d.o.f. counter: $k = k + 1$
establish the connectivity: $NCON(k) = NPOINT(nod) + j - 1$
end of loop through the nodal d.o.f.
end of loop through the element nodes
for each (modified) element d.o.f. (shape function) $k2$
determine the connectivity: $i2 = NCON(k2)$
accumulate for the global load vector:
$LOAD(i2) = LOAD(i2) + Bload(k2)$,
for each element d.o.f. (shape function) $k1$
determine the connectivity: $i1 = NCON(k1)$
accumulate for the global stiffness matrix:
$STIFF(i1, i2) = STIFF(i1, i2) + Aload(k1, k2)$
end of the second loop through element d.o.f.
end of the first loop through the element d.o.f.
end of the loop through the elements

3.8 Interface with a frontal solver

The so-called frontal solver has become a popular choice among direct solvers for finite element codes due to its natural implementation in an 'element by element' scheme. In this method, a 'front' sweeps through the mesh, one element at a time, assembling the element stiffness matrices into a global matrix. The distinction from the standard assembling procedure is that, as soon as all of the contributions for a given dof have been accumulated, that dof is eliminated from the system of equations using standard Gaussian operations. Thus, in the frontal solver approach the operations of assembling and elimination occur *simultaneously*. The global stiffness matrix never needs to be fully assembled, and this leads to the significant savings in memory that has given the frontal solver its popularity.

Here we will only describe the interface with the frontal solver, not the solver itself. The interface is constructed via four routines, all located in the $solver1$ directory: $solve1.f$,$solin1.f$, $solin2.f$, and $solout.f$. We will now give an overview of these routines. For coding details we refer to the source codes in $solver1$.

The frontal solution consists of two steps: prefront, and elimination. The prefront requires two arrays on input: in and $iawork$. For each element, in contains the number of nodes associated with the corresponding modified element, and $iawork$ contains a listing of nicknames for the nodes of the modified element. The nicknames are defined as follows: for

a given node 'j',

$$nick_j = j * 1000 + NREQNS \tag{3.69}$$

where $NREQNS$ is the number of equations being solved. With this information, the prefront produces the destination vectors which, for a given element, denote at what stage of the frontal solution each of its nodes can be eliminated. Once this information is constructed, the elimination phase can begin. $Solve1.f$ prepares arrays in and $iawork$, calls the prefront routines, and then calls the main elimination routines. Thus, this routine is seen to be the primary driver of the frontal solver.

The other interface routines are simply for auxiliary purposes. For a given element, $solin1.f$ returns a listing of the destination vectors of the associated modified element, $solin2.f$ returns the modified element stiffness matrix and load vector, and $solout.f$ takes the solution values returned from the frontal solver and inserts them into the data structure (for a given node, the values of the corresponding dof must be placed into the $NODES(nod)\%dof$ entry).

3.9 Initial mesh generator. Interfacing with the Geometric Modeling Package (GMP)

As a part of the package, we provide a simple, multiblock mesh generator based on a Geometric Modeling Package. Referring to all details on the modeling to the GMP manual [5], we review here shortly the main assumptions of the code. It is assumed that a 2D manifold, i.e. either a domain in $\mathbb{R}^2$ or a collection of surfaces in $\mathbb{R}^3$, is described as a union of disjoint images of standard reference triangle $\tilde{T}$ or standard reference rectangle $\tilde{R}$, through some maps (parametrizations) specified by the user,

$$\Omega = \bigcup_i T_i \cup \bigcup_j R_j \tag{3.70}$$

where

$$T_i = \boldsymbol{x}_{T_i}(\tilde{T}), \quad R_j = \boldsymbol{x}_{R_j}(\tilde{R}) \tag{3.71}$$

The partition of the manifold into the figures must be *regular*, in the same sense as for FE meshes, i.e. a common part of two figures is either a vertex or a common (whole) edge, or is empty. Moreover, the parametrizations are *compatible* in the sense that for two neigboring figures, the corresponding restrictions of the figure parametrizations to the common edge yield an identical parametrization. Intuitively speaking, if the two parametrizations are used to generate an *hp* FE mesh along the common edge, the two meshes will be identical.

The idea of the mesh generation is now as follows. For each of the triangles T in the manifold, the user specifies the corresponding number of subdivisions $nsub_T$ and the order

of approximation p_T. For a rectangle R, we need the number of subdivisions $nsubh_R, nsubv_R$ for the *horizontal* and *vertical* edges. The terms *horizontal* and *vertical* have nothing to do with the actual geometrical position of the edges and refer merely to their local ordering, i.e. the first and the third sides of a rectangle are *horizontal* while the second and the fourth are *vertical.* Similarly, for each of the rectangles in the mesh, we specify the corresponding *horizontal* p_{hR} and *vertical* p_{vR} orders of approximation. Again, the numbers of subdivisions for the figures must be *compatible* with each other in the sense that the produced mesh will be regular. The order of approximation along an edge common for two figures is fixed, consistently with the definition of *hp* elements, using the *minimum rule.*

Next, for each triangle T and rectangle R, the corresponding reference figure is covered with a *uniform* mesh with the number of subdivisions specified by the user. Triangles are divided into triangular elements and rectangles are divided into rectangular elements. The geometry d.o.f. are then generated using the *hp* interpolation procedure. The concept is illustrated in Fig. 7. Given an element $\tilde{K}$ from the mesh covering a reference figure, we identify the corresponding affine map $\boldsymbol{\eta}_{\tilde{K}}$ mapping the master element $\hat{K}$ onto the element $\tilde{K}$, and consider the corresponding composition of map $\boldsymbol{\eta}_{\tilde{K}}$ and figure parametrization $\boldsymbol{x}_F(\boldsymbol{\eta})$,

$$\boldsymbol{x}(\boldsymbol{\xi}) = \boldsymbol{x}_F(\boldsymbol{\eta}_{\tilde{K}}(\boldsymbol{\xi})) \tag{3.72}$$

In principle, the map could be used directly for element computations.[14] Instead, we choose to replace the map with its *hp*-interpolant. This is where the order of approximation for the elements in the mesh comes into the picture. We emphasize the *locality* of the *hp*-interpolation procedure, all operations are done elementwise only.

The described initial mesh generation is done in routine *meshgen/hp2gen*. The routine generates nodes and elements, i.e. allocates the corresponding data structure arrays $ELEMS$, $NVERS$, and $NODES$, and initiates the geometry d.o.f. $NVERS(*)\%coord$, $NODES(*)\%coord$ with the relevant values coming from *hp*-interpolation routines stored in directory *hp_interp*. Free entries in arrays $ELEMS$, $NVERS$, and $NODES$ are linked, and pointers to the first free entries are stored in the data structure module.

REMARK 2 Please note that, independently of order of approximation specified for elements, all nodes are generated. For instance, for a linear approximation we need only vertex nodes. We still generate the mid-edge, and the middle nodes as well. This simplifies the logic of the code and makes its customization to other types of elements easier. The actual d.o.f. for non-vertex nodes, $NODES(*)\%dofs$, however, are allocated dynamically. If the node is not used in computations, the array is not allocated. ▮

[14] Such a possibility is always an option, it would require a modification of the element routine only.

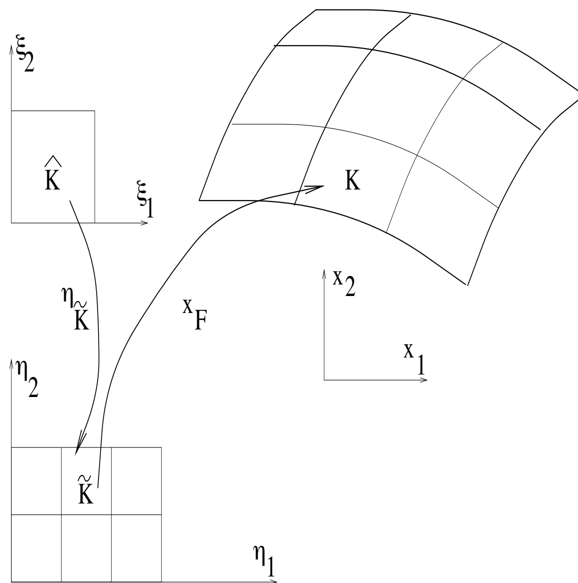


Figure 7: Initial mesh generation

Sample input files, specifying input for GMP and the mesh generator, can be found in *files/inputs*.

REMARK 3 As mentioned above, rectangles are covered with quads, and triangles with triangular elements. Thus, in order to generate a mesh consisting of triangular elements in a rectangular domain, one has to divide the rectangle into two triangles. This, unfortunately, changes the corresponding parametrizations and produces a different mesh. ▮

3.10 *p*-adaptivity

Finally, we discuss the possibility of *p*-refinements (unrefinements). Since the modifications do not introduce new or delete existing nodes, the corresponding changes in the data structure arrays are minimal. Order of approximation for an element *nel* can be modified (increased or decreased) by invoking routine *meshmods/enrich*. The routine enforces the *minimum rule*, i.e. the order for an edge common for two elements is fixed to the minimum of the orders for the elements. For rectangular elements, the appropriate horizontal or vertical order is considered. The actual changes in data structure arrays are executed in routine *meshmods/nodmod*. Please try to modify the order for an element using the interactive routine for mesh modifications *meshmods/mesh* in order to verify the discussed rules.

When the order of approximation of a node is modified, the corresponding geometry d.o.f. should be regenerated using the same procedure as that in the initial mesh generation.

For that reason, an interface between GMP and 2Dhp90 must be maintained. The interface includes two arrays in common *commons/cnsubf* storing number of subdivisions for each triangle or rectangle corresponding to the GMP representation, and a nickname stored for each (initial mesh) element *nel*:

$$ELEMS(nel)\%father = -(lnel + nf * 10000) \tag{3.73}$$

where nf is the number of the corresponding triangle or rectangle, and *lnel* is the *local* element number for the figure, This information plus the number of subdivisions stored in the common block discussed above allows for a simple reconstruction of the element position with respect to the underlying geometrical figure.

REMARK 4 It is not easy to notice that a change of a mid-edge node geometry dof requires a modification of the geometry d.o.f. for the middle nodes of both adjacent elements. This is related to the *hp*-discretization procedure where a change of mid-edge node d.o.f. affects values of the middle node d.o.f.. In order to simplify the procedure of updating the geometry d.o.f., we have eliminated in the present version the instantaneous update done in the original version. Instead, the user can execute routine *hp_interp/update_gdof* to update *all* geometry d.o.f. in the mesh at the same time. A similar routine *hp_interp/update_DirichletBC* updates the interpolant of Dirichlet data. ▮

4 The hp Finite Element Method on h-Refined Meshes

4.1 Introduction. The h-refinements

Dividing a *father* element into smaller element *sons* is known as an h-refinement. The code supports the following h refinements:

- $h4$-refinement of triangles,
- vertical or horizontal $h2$-refinement of quads,
- $h4$-refinement of quads.

The words 'vertical' or 'horizontal' refer to the local coordinates for the quad element. If the element is broken across the first axis, the refinement is vertical, if the element is broken across the second axis, the refinement is horizontal. Throughout the code, we use an array $kref(2)$ to indicate the requested kind of refinement for a quad element. If the element is broken across the i-th axis, $kref(i) = 1$, otherwise $kref(i) = 0$. Thus, $kref = (1,1)$ indicates the $h4$- refinement, $kref = (1,0)$ the vertical $h2$-refinement, and $kref = (0,1)$ the horizontal $h2$-refinement. Consequently, quad elements can be refined into four element sons in different ways, by either executing a single $h4$-refinement, or a sequence of $h2$-refinements of the element, and then its sons.

During the h-refinements, new elements are created and some old ones disappear, new nodes are generated and some existing nodes are deleted. Routines that record the corresponding changes in the data structure arrays have been collected in directory *meshmods*. Contrary to the initial mesh, for which all necessary information is stored explicitly, the information for elements that have resulted from h-refinements, will be recovered from the data structure arrays through special algorithms. The corresponding routines have been collected in directories *datstrs* and *constrs*.

Perhaps the most profound effect of the h-refinements is the introduction of *constrained (hanging) nodes*. For 2D meshes, we can have constrained vertex or mid-edge nodes, middle nodes are always unconstrained. A typical situation is illustrated in Fig. 8. As the result of a refinement, a *big* element, across one of its edges, has two smaller *neighbors*. The vertex node that lies in the interior of the big element edge, and the two mid-edge nodes lying on the small element edges, are constrained nodes. The nodes lying on the big element edge are identified as the *parent nodes* of the constrained nodes. We shall guarantee that parent nodes of constrained nodes are themselves *always* unconstrained, by using a special *1-irregular meshes refinement algorithm* discussed below.

A mesh with constrained nodes is called an *irregular* mesh. Construction of basis functions for irregular meshes is more difficult than for the regular ones. A shape function for

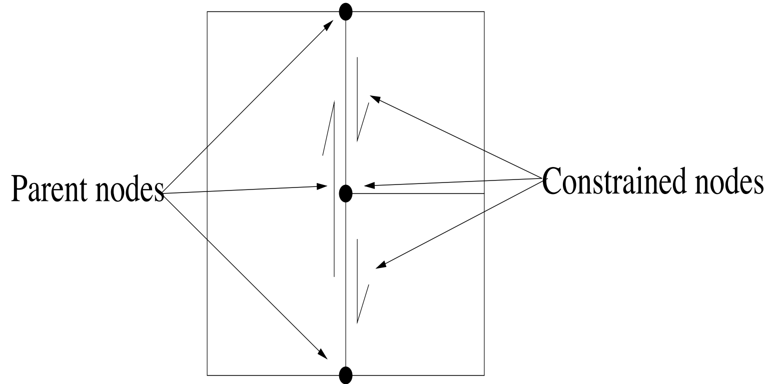


Figure 8: Constrained nodes

a big element cannot be matched with single shape functions from neighboring small elements. Instead, one has to use a linear combination of the small element shape functions. The situation is illustrated for the simplest case of a mesh of bilinear elements in Fig. 9.

4.2 1-Irregular Mesh Refinement Algorithm

It may happen that parent nodes of a constrained node are themselves constrained. In such a case the original node is said to be *multiply constrained*. It is desirable for many reasons [15] to avoid multiple constrained nodes and limit ourselves to 1-irregular meshes only. In order to avoid the multiple constrained nodes, we prohibit any h-refinement of an element, unless *all* its nodes are unconstrained. The following algorithm is a generalization of an original algorithm of Rheinboldt et al. for $h4$-refinements of one-irregular rectangular meshes [16]. The procedure uses the standard idea of a "shelf" (a waiting list).

Arguments in:

  an element number $mdle$ and requested kind of refinement $kref$

```
     put element mdle and requested refinement kref on the shelf
10   stop if the shelf is empty
     pick the last mdle and kref from the shelf
     for each of element mdle mid-edge nodes
          if the node is constrained then
               put mdle and kref back on the shelf
               determine father mid-edge node medg of the constrained mid-edge node
               determine the big element neighbor mdlen of medg
```

[15] See e.g. the discussion in [3].

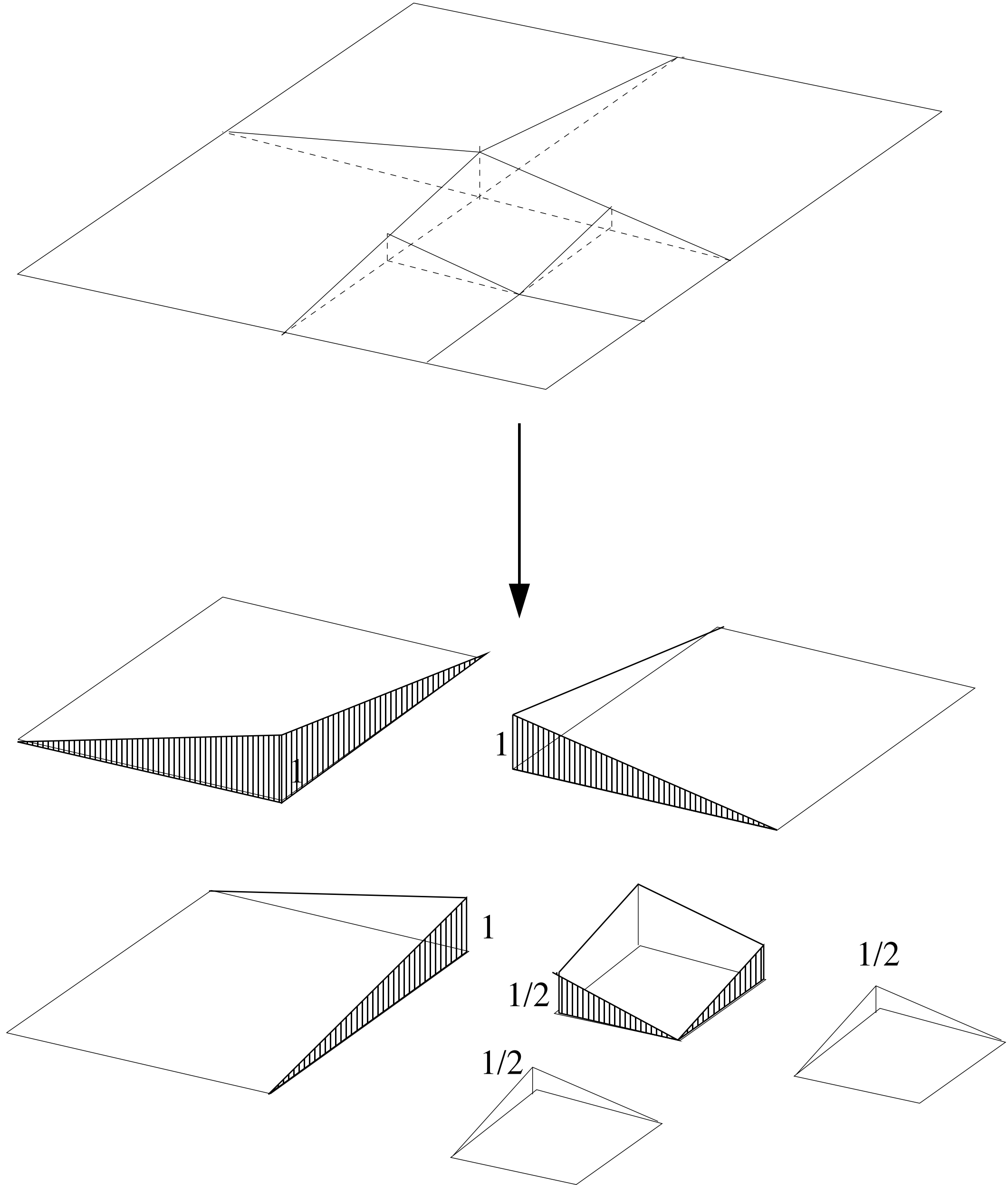


Figure 9: Construction of basis functions for an irregular mesh

add $mdlen$ and $kref = (1, 1)$ to the shelf
go to 10
elseif $kref$ does not call for breaking the edge (valid for quads only)
determine the neighbors of the mid-edge node across the element edge
if there are two small neighbors then
set $kref = (1, 1)$ and reshelf $mdle$ and $kref$
go to 10
endif
endif
end of loop through the mid-edge nodes
for each of element $mdle$ vertex nodes
if the node is constrained then
put $mdle$ and $kref$ back on the shelf
determine father mid-edge node $medg$ of the constrained vertex node
determine the big element neighbor $mdlen$ of $medg$
add $mdlen$ and $kref = (1, 1)$ to the shelf
go to 10
endif
end of loop through the vertex nodes
break the element according to the info in $kref$
go to 10

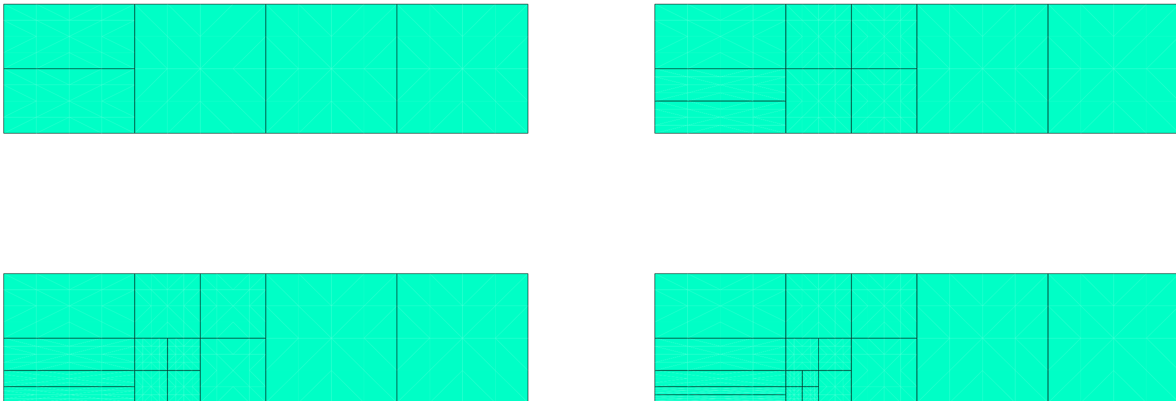


Figure 10: Terminating a 'boundary layer' with $h4$-refinements

A number of comments:

1. As mentioned earlier, elements in the code are identified with their middle node numbers. Thus, whenever we refer to an element number, we mean its middle node number.

2. The algorithm eliminates constrained nodes by enforcing $h4$-refinements of 'big' neighbors, even when the original element is to be refined only anisotropically. We anticipate using the anisotropic refinements only in special cases like boundary layers. Enforcing $h4$-refinements of the 'big' neighbors allows then for a nice termination of the refinement, contrary to enforcing only $h2$-refinements of the neighbors. The situation is illustrated in Figures 10,11. We begin with a mesh of four elements and we continue refining the (current) element in the left lower corner in the horizontal direction. In Fig. 10 we enforce $h4$-refinement of the big neighbor, in Fig. 11, only the minimal $h2$-refinement, necessary for eliminating the constrained node. The first strategy results in a nice termination of the 'boundary layer', the second causes the refinements to spread.

3. We modify an anisotropic refinement of a quad element to fit refinements of neighbors. This eliminates incidental anisotropic refinements which are intended only to capture solutions that depend only upon one of the two element local coordinates (e.g. in the case of boundary layers and meshes aligned with the boundary).

4. For quadrilateral meshes, eliminating constrained mid-edge nodes implies the elimination of constrained vertex nodes, for triangular meshes, it does not. Fig. 12 shows a simple triangular mesh, where the element in the middle has no unconstrained mid-edge nodes, but all its vertex nodes are constrained.

5. The algorithm requires the use of the following information.

   - Given an element (its middle node), return its mid-edge and vertex nodes. In the case of a constrained node, return its parent nodes. This is a standard *nodal connectivities* information.
   - Given a mid-edge node, return its neighbors.

   Note that, contrary to earlier implementations, the algorithm does not require determining (at least directly) element neighbors for a given element.

4.3 Data structure in Fortran 90 - continued.

The genealogical information for nodes. During h-refinement, the content of the data structure must be updated. Contrary to previous implementations, the new data structure

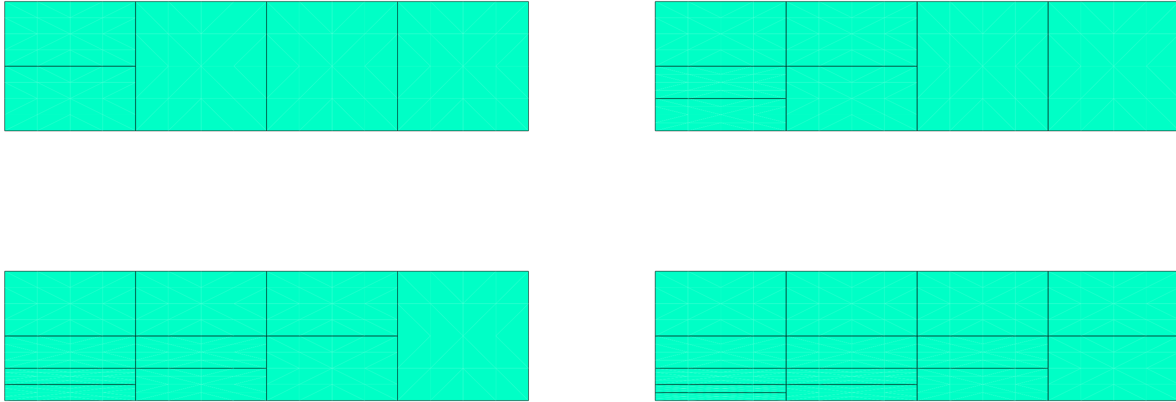


Figure 11: Terminating a 'boundary layer' with $h2$-refinements

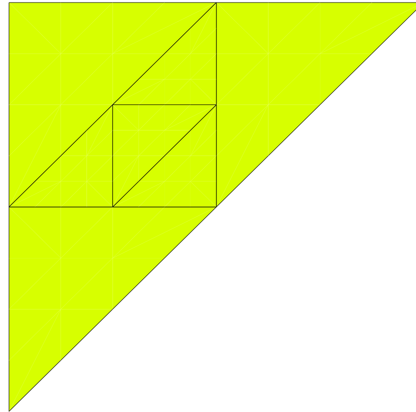


Figure 12: Example of a triangle with unconstrained edges and constrained vertices

records only a very minimal information, and only for nodes. *No information for the new elements, resulting from h-refinements is stored whatsoever !*. The information stored for nodes supports the genealogical information (trees). When a *father* node is refined (broken), we store the information about its *nodal sons*. A mid-edge node has three sons: two mid-edge nodes and one vertex node. An ($h4$-)refined middle node for a triangle has seven sons: four middle nodes, and three mid-edge nodes. An $h2$-refined middle node of a quad has three

sons: two middle nodes, and one mid-edge node. Finally, an $h4$-refined middle node for a quad, has nine sons: four middle nodes, four mid-edge nodes, and one vertex node. The numbering of the sons and the orientation of the new mid-edge nodes is depicted in Fig. 13. The numbers of the new nodes are stored in array *NODES(nod)%sons*, whose length depends on the type of the node and the refinement kind and, for that reason, the array is allocated dynamically. The executed *refinement kind* is stored in *NODES(nod)%ref_kind*. Notice that vertex nodes are never refined and, therefore, this information is relevant only for the non-vertex nodes. For the newly generated nodes we store their father in either *NVERS(nod)%father* or *NODES(nod)%father*. Finally, *NODES(nod)%connect* is a pointer to an entry in an extra, temporary array storing the *extended parent information* for the new nodes. When a mid-edge node is refined, the extended parents include its vertex nodes, for a middle node, it includes element mid-edge and vertex nodes. This extra information is necessary when constructing *extension operators* for multigrid operations. We shall return to this subject in version 2.1 of this code.

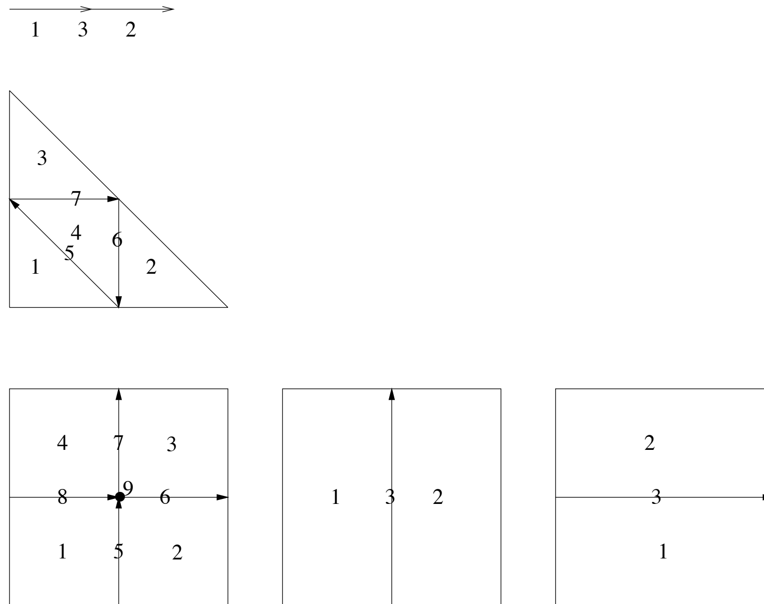


Figure 13: Numbering of sons for broken father mid-edge, triangle and quad middle nodes

The natural order of elements As elements are identified through their middle nodes, the genealogical information for the middle nodes includes the genealogical information for elements. This supports the so called *natural ordering of elements*. The ordering follows the numbering of elements in the initial mesh (introduced by the mesh generator) and the tree structure of elements following the *leaves* of the tree (*active* elements). Figure 14 shows a typical binary tree and the resulting natural order of elements.

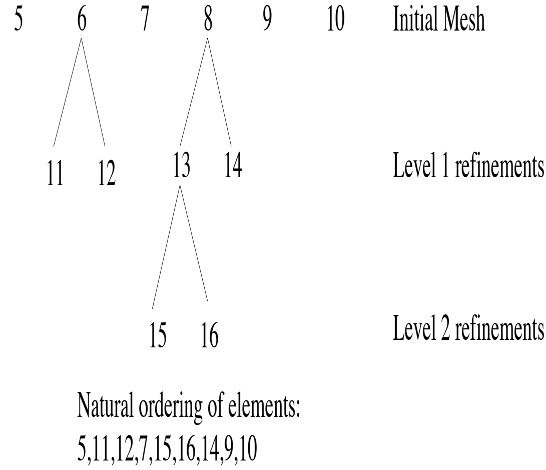


Figure 14: The natural order of elements.

The algorithm is executed by routine */datstrs/nelcon*. A typical loop through all *active* elements in the mesh then looks as follows

```
mdle = 0
do iel = 1, NRELES
        call nelcon(mdle, mdle)
        ⋮
enddo
```

4.4 Constrained approximation for C^0 discretizations

As illustrated in Fig 9, the construction of basis functions for irregular meshes is more complicated than that for regular ones discussed in the previous section. In general, restriction of a basis function to a contributing element does not reduce to (plus/minus) the element shape function. Instead, we have to assume that the basis function must be represented as a linear combination of the element shape functions:

$$e_i|_K = \sum_k c_{ik,K} \phi_{k,K}. \tag{4.74}$$

The calculation of the global stiffness is now more complicated that for regular meshes (comp. (3.60)):

$$S_{ij} = b(e_i, e_j) = \int_\Omega \boldsymbol{\nabla} e_i \boldsymbol{\nabla} e_j = \sum_K \sum_k \sum_l c_{ik,K} c_{jl,K} b_K(\psi_k, \psi_l) \tag{4.75}$$

where the element stiffness matrix (3.61) is the same as for regular meshes. Similarly, the global load vector is evaluated using the strategy:

$$L_i = l(e_i) = \int_\Omega f e_i \, d\boldsymbol{x} + \int_{\Gamma_2} g e_i \, ds = \sum_K \sum_k c_{ik,K} l_K(\psi_k) \tag{4.76}$$

with the element load vector defined by (3.63). The difference between (3.60) and (4.75), and (3.62) and (4.76) lies in the extra summations over the element shape shape functions and the connectivity coefficients $c_{ik,K}$. In practice, we sum up only over those indices for which the coefficients are non-zero. The calculations for regular meshes are then just a special case of those for irregular meshes with the summation extending over one index only and the connectivity coefficient reduced to the sign factor.

The actual algorithm of assembling the global matrices is not done from the point of the view of the receiver (the global matrices) but contributor (the element matrices) and it looks as follows.

initiate global matrices S_{ij} and L_i with zeros,
for each element K in the mesh,
    calculate element stiffness matrix $S_{kl,K}$ and load vector $L_{k,K}$
    for each local d.o.f. k
        for each connected global d.o.f. i
            determine connectivity $i = i(k,K)$ and the connectivity coefficient $c_{ik,K}$
            accumulate for the global load vector:
            $L_i = L_i + c_{ik,K} * L_{k,K}$
            for each local d.o.f. l
                for each connected global d.o.f. j
                    determine connectivity $j = j(l,K)$ and coefficient $c_{jl,K}$
                    accumulate for the global stiffness matrix:
                    $S_{ij} = S_{ij} + c_{ik,K} * c_{jl,K} * S_{kl,K}$
                end of the second loop through the connected global d.o.f.
            end of the second loop through local d.o.f.
        end of the first loop through the connected global d.o.f.
    end of the first loop through the local d.o.f.
end of loop through elements

Modified element. Even more than for regular meshes, it is convenient to separate the extra summation over the connected d.o.f. from the classical assembling procedure by introducing the definition of the *modified element*. The information about the modified element is identical with that for regular meshes and it includes:

- a list of element vertex nodes: $Nod1(i), i = 1, \ldots, Nvert$,
- a list of element non-vertex nodes: $Nod2(i), i = 1, \ldots, Nrnod$,
- the corresponding number of shape functions (d.o.f) per node: $Ndof2(i), i = 1, \ldots, Nrnod$,
- the modified element load vector: $Bload(k), k = 1, \ldots, Nrdof$,
- the modified element stiffness matrix: $Astiff(k, l),\ k, l = 1, \ldots, Nrdof$.

The list of vertex nodes $Nod1$ contains numbers of all vertex nodes connected to element vertices. This includes the element unconstrained vertex nodes and parent vertex nodes of the constrained vertices. Notice that the two sets (unconstrained nodes and parents of constrained nodes) need not be disjoint. Similarly, the list of non-vertex nodes $Nod2$ contains both the element unconstrained mid-edge and middle nodes and parent mid-edge nodes of element constrained nodes. The interpretation of the other quantities is the same as for regular meshes. Once the lists of the modified vertex and non-vertex nodes are created, the corresponding modified element d.o.f. are numbered following the order of nodes on the lists and then the order of d.o.f. corresponding to each of the nodes. Next, connectivities of the local d.o.f. k to the modified element d.o.f. $NAC(j, k), j = 1, \ldots, NRCON(k)$ and the corresponding connectivity coefficients $CONSTR(j, k), j = 1, \ldots, NRCON(k)$ are determined. The calculation of the modified element matrices is then done using the same algorithm as above:

initiate $Bload$ and $Astiff$ with zeros
for each local d.o.f. $k2$
    for each connected modified element d.o.f. $j2 = 1, \ldots, NRCON(k2)$
        $i2 = NAC(j2, k2)$
        accumulate for the modified load vector:
        $Bload(i2) = Bload(i2) + CONSTR(j2, k2) * Bloc(k2)$
        for each shape function $k1$
            for each connected modified element d.o.f. $j1 = 1, \ldots, NRCON(k1)$
                $i1 = NAC(j1, k1)$
                accumulate for the modified stiffness matrix:
                $Astiff(i1, i2) = Astiff(i1, i2)$
                $+CONSTR(j1, k1) * CONSTR(j2, k2) * Aloc(k1, k2)$
            end of the second loop through connected d.o.f.
        end of the second loop through the element d.o.f.
    end of the first loop through connected d.o.f.
end of the first loop through the element d.o.f.

Here $Bloc, Aloc$ denote the element (local) load vector and stiffness matrix.

At this point it should become clear that the whole difficulty in implementing the constrained approximation is in constructing the arrays $NRCON, NAC, CONSTR$. The implementation presented in this document is already the fourth iteration that the first author has been involved with. In our first code [3], the entire information about the constraints was reconstructed from information stored for *active elements* ONLY. In the second implementation [4] we went to other extreme and stored the information about the constraints explicitly in the data structure arrays. This was very convenient for the constrained approximation but had made the mesh modification routines very complicated. In the last implementation [8], we reconstructed the information about the constraints for an element from the nodal connectivities stored for the elements and its father (and possibly grandfather). In the presented (ultimate?) implementation, arrays $NRCON, NAC, CONSTR$ are constructed from a local data base on constraints generated during the reconstruction of element nodal connectivities. This combines the simplicity of implementation in [4] with no need for storing directly any information for constrained nodes. The algorithm described next is absolutely crucial for this new implementation.

4.5 Reconstructing element nodal connectivities.

Instead of presenting a formal algorithm, we shall illustrate it with an example presented in Fig. 15. The first picture represents a mesh that has resulted from a sequence of $h4$-refinements of an initial mesh of just four elements. Suppose we want to find the nodes of the element marked in red in the first picture. The corresponding denumeration of elements, i.e. their middle nodes numbers are shown in the second picture. We are interested in the element 115. We begin by ascending the tree of elements up [16] all the way to an initial mesh element, in this case the left corner element 13 of the initial mesh shown in the third picture. When ascending the tree, we remember the way back to the original element. For the initial mesh ancestor, the corresponding nodes are stored in data structure array *ELEMS(nel)%nodes* where *nel* is the element number of the initial mesh ancestor (using consecutive numbering, in this case $nel = 2$). Given the nodes of the initial mesh element, we descend back to the original element, reconstructing the nodes of the ancestors from the nodal trees. The fourth picture shows the reconstruction of nodes for element 19. We then turn to element 59, and reconstruct nodes for its element sons, shown in the fifth picture.

Only in the last stage, shown in the sixth picture, we encounter nodes (of the original element father) that have not been refined. For those nodes, we construct the necessary information about the corresponding constraints and store it a local data base in modul

[16] Computer trees grow downward...

module/tree. The negative numbers for the element nodes indicate constrained nodes and serve as pointers to a local data base for the constrained nodes.

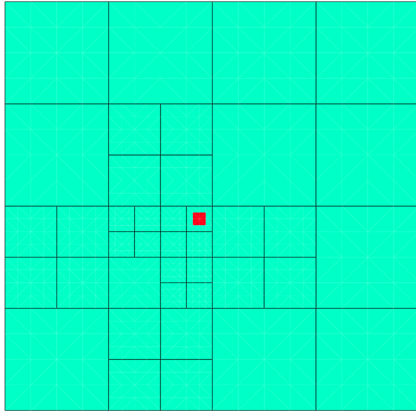
The reconstruction is done in routine *constrs/history*. Besides the information on constraints, the reconstructed information stored in modul *module/tree* contains not only the element nodes, but also their order (relevant for non vertex nodes), their orientation (relevant for mid-edge nodes) and the element refinement history, including the initial mesh ancestor and the refinement path. With a call made to routine *history*, the content of modul *module/tree* is first checked, and the routine is executed only if the information stored there does not correspond to the element on the input. That way we avoid an unnecessary multiple execution of the algorithm. A necessary, precomputed logical information used by routine *constrs/history*, is stored in module *module/refinements* and *module/edges*.

Definition of the modified element. The lists of nodes for the modified elements and the corresponding constrained approximation coefficients are determined by routine *constrs/logic*, which calls specialized versions *constrs/logict* for triangular, and *constrs/logicq* for quad elements. The lists of nodes for the modified elements are constructed in auxiliary routines *constrs/logict_nodes* and *constrs/logicq_nodes*, respectively. For the example discussed above, and element 115, the corresponding vertex nodes are determined as follows

- regular vertex node 34,
- vertex parent nodes 25,5 of the second, constrained element vertex node marked with -13,
- the third, regular vertex node 5 is already on the list,
- vertex parent node 30 of the fourth, constrained vertex node, marked with -23; the first parent vertex node 5, is already on the list.

The list of non-vertex nodes for the modified element is assembled simultaneously with the list of vertex nodes. After the loop through the vertex nodes, the list contains already mid-edge parent node 90 (of the second vertex node), and mid-edge parent node 100 of the fourth vertex node. We proceed now by looping through mid-edge nodes, adding to the list of non-vertex nodes of the modified element:

- regular (first) mid-edge node 118,
- regular (fourth) mid-edge node 119.



46 45 68 67
94 93
43 91 92 65 66
124 123 116 115
104 103 84 83
121 122 113 114 31
146 145
101 102 57 81 82
143 144
134 133
17 29 30
131 132

7 5 8 8 6 9
10 15 11 11 16 12
4 3 5 5 4 6
4 3 5 5 4 6
7 13 8 8 14 9
1 2 2 2 3

4 51 18 18 52 5
28 20 23 23 19 42
12 24 10 10 22 16
12 24 10 10 22 16
27 17 21 21 18 41
25 11 11 26 2

18 99 30 30 100 5
110 60 63 63 59 90
32 64 21 21 62 28
32 64 21 21 62 28
109 57 61 61 58 89
-11 -13 -13 -12 16

30 -12 -13 -23 -21 5
130 116 119 119 115 -12
36 120 34 34 118 -13
36 120 34 34 118 -13
129 113 117 117 114 -11
151 41 41 152 28

Figure 15: Reconstructing nodal connectivities for an element

Finally, we add to the list, (always regular) middle node 115. Once the lists of the modified element nodes have been determined, we determine the corresponding constrained approximation coefficients using the information in the local data base for the element constrained nodes. This is done in routine *constrs/logic2*.

We encourage the user to study the code to get a better idea how these things are organized. For a k-th d.o.f that belongs to an unconstrained node, the constrained approximation matrices reduce to those discussed in the previous section, i.e. $NRCON(k) = 1$, $NAC(1,k)$ contains the corresponding location of the d.o.f. on the list of the modified element d.o.f. (with the order induced by the order of its nodes), and $CONSTR(1,k)$ equals the sign factor, depending upon the orientation.

4.6 Determining neighbors for mid-edge nodes

Given a mid-edge node, we set to determine its element (middle nodes) neighbors on both sides. The neighbor information is used in the 1-irregular meshes refinement algorithm, in routines performing refinements, in graphics and (not presented here) in a-posteriori error estimation. It is an important piece of information about the mesh.

Routine $datstrs/find_medg_neig$ determines the *youngest equal size* neighbors for a mid-edge node that exists in the data structure array, active or not. If the edge is on the boundary, one neighbor is returned, for internal edges we get two. The overall strategy is very similar to that for determining element connectivities.

Step 1: Go up the tree of refinements for the mid-edge node until you reach a middle node or an initial mesh mid-edge node. A mid-edge node must have a father mid-edge node, a father middle node, or belong to the initial mesh. In the father is another mid-edge node, we replace the node with its father, and store the information about the path (first or second son, going left or right). If the father of the currently processed mid-edge node is a middle node, we know its neighbors from the way we denumerate sons of a middle node, comp. Fig. 13. For instance, for an $h11$ refined quad middle node, neighbors of the first edge-son (son 5 altogether) are sons number 1 and 2. If the edge belongs to the initial mesh, we store in place of its father a link to one of the neighboring initial mesh elements. The information about the element neighbors is then used to recover the second neighbor for the edge. In the end of the first step, we know the path to the mid-edge ancestor and the middle node neighbors (one or two) of the ancestor.

Step 2: For each of the middle node neighbors of the ancestor edge, descend the refinement tree for the middle node, following the stored path information, until you reach the

youngest element, adjacent to the original mid-edge node, that occupies the whole edge, comp. routine *datstrs/descend*.

In summary, we ascend the mid-edge nodes trees, jump to the middle nodes neighbors, and descend the middle nodes trees.

Routine *datstrs/neig_edge* determines neighbors for an *active mid-edge node*. There may be one big or two small neighbors on each of the mid-edge node sides. The routine starts by determining the equal size neighbors, and then descends further the neighbor middle nodes trees until it reaches unrefined middle nodes (elements) adjacent to the edge.

4.7 Additional comments

Assembling of global matrices. Interfacing with solvers. The global assembling procedure is identical with that for regular meshes discussed in the previous section. Once the modified elements matrices are calculated, the regular and irregular meshes are treated in an identical way. This is especially convenient when it comes to writing an interface with a linear equations solver. In particular, the interface with the frontal solver discussed in the last section remains exactly the same. In principle, the constrained approximation has been entirely hidden from the user in the constrained approximation routines. The user is free to modify the element routine (in particular to accommodate different boundary-value problems) and the constrained approximation routines will automatically produce the modified element arrays allowing then again for a standard interface with a solver.

Evaluation of local d.o.f. Routines *constrs/nodcor* and *constrs/solelm* return for an element *local* geometry and actual d.o.f. necessary to evaluate the geometry or the solution (e.g. for postprocessing). The evaluation is based again on the constrained coefficients stored in arrays $NRCON, NAC$ and $CONSTR$. First, the corresponding modified element d.o.f. are copied from the data structure arrays into a local vector $dofmod$ and then the local d.o.f. vector dof is evaluated using the algorithm:

initiate dof with zeros
for each local d.o.f. k
    for each connected modified element d.o.f. $j = 1, \ldots, NRCON(k)$
        $i = NAC(j,k)$
        accumulate for the local d.o.f.:
        $dof(k) = dof(k) + CONSTR(j,k) * dofmod(i)$
    end of loop through connected d.o.f.
end of loop through the element d.o.f.

In this way, the conversion of the global d.o.f. to the local ones is again hidden from the user who needs only to know how to call the two routines.

p-refinements. The p-refinement for an element is executed in routine *meshmods/modord*. Given new orders for an element nodes, the routine modifies the nodes initiating new d.o.f. and performing the necessary changes in data structure arary *NODES*. The new d.o.f. are initiated using the old d.o.f. by complementing them with zeros (if the polynomial order increases) or by simply deleting the redundant d.o.f. [17]. In the case of a constrained mid-edge node, we identify and modify its father. The *minimum rule* is enforced, i.e. the order for the egdes never exceeds orders of the neighboring middle nodes. The actual modification of the nodes is done in routine *meshmods/nodmod*.

Finally, routine *meshmods/enrich* requires on input only the order of approximation for the element middle node, enforcing the same order for the element edges.

We invite again the user to play with the interactive routine for mesh modifications in order to verify the discussed rules.

[17] This ought to be replaced with a projection-based interpolation...

5 Organization of the code

The code is organized in the following subdirectories:

- *adapt* - automatic *hp* refinements,
- *blas* - basic linear algebra routines,
- *commons* - system common blocks,
- *constrs* - constrained approximation routines,
- *constrs_util* - constrained approximation utilities,
- *datstrs* - data structure routines,
- *elem_util* - element utilities,
- *ffld* - free field reader routines,
- *files* - system files, sample input files,
- *frontsol* - frontal solver routines,
- *frontsolz* - frontal solver routines (complex version),
- *gcommons* - graphics common blocks,
- *geometry* - Geometry Modeling Package routines,
- *graph_2Dhp* - actual graphics routines for the code,
- *graph_geom* - graphics routines for the Geometric Modeling Package,
- *graph_interf* - graphics interface routines,
- *graph_util* - graphics utilities,
- *hp_interp* - *hp*-interpolation utilities,
- *laplace* - user routines for the Laplace equation and more general elliptic equations or systems of such,
- *main* - driver for the code,
- *meshgen* - initial mesh generation routines,

- *meshmods* - mesh modification routines,
- *module*,*module_complex*,*module_real* - data structure moduli,
- *solver*1 - interface with frontal solver,
- *utilities* - general utilities.

Included in the main directory are also:

- *makefile*,
- *tarall* - a macro to *tar* the code into a single file,
- *readme* - info how to convert the code into the complex mode.

Upon compiling, the code is executed by typing *a.out*, entering the main menu (main program *main/main*).

Most of the essential packages have already been commented on in the previous chapters. All routines in the code are fully commented and (most of the time) follow the coding protocol. We shall continue now with a brief description of selected directories which have not been dicussed yet.

5.1 Graphics interface

In order to run the code, besides a FORTRAN 90 compiler the user will need an interface with the computer graphics. The routines from directory *graph_interf* provide such an interface with standard X-Windows graphics. The interface allows additionally for producing a PS copy of any picture that appears on the computer screen.

5.2 Graphics package

The graphics driver routine *graph_2Dhp/graph* contains no arguments and can be called from any place in the code. This makes it in particular a convenient debugging tool. The routine allocates a memory, initiates the code windows system, and enters a menu that includes calls to:

- *graph_2Dhp/grsurf* - a surface graphics routine; plots the current mesh or a contour map for the solution (components),
- *graph_2Dhp/mesh* - an interactive mesh modification routine,

- *graph_2Dhp/rates* - a routine plotting convergence rates (history).

The main idea of displaying the mesh is simple. The actual elements are broken into small triangles, and the information relevant for the triangle (geometry, color, border lines, a node number to write out) are encoded into a (large) array of such triangles in routine *graph_2Dhp/lsvisids*. The array is then manipulated by routine *graph_util/dpvisids* that displays the whole array or parts of it when we zoom on a portion of the mesh. In $\mathbb{R}^3$, the triangles are displayed according to their distance from a point of view (from the farthest to the nearest), which allows for a simple hidden line effect.

A similar algorithm is used to construct contour maps. Over each of the small element subtriangles the solution is approximated with a linear function, and a family of (overlapping) triangles is defined corresponding to areas bounded by corresponding contour lines of the (linear) solution. Each of the subtriangles is then stored as a separate element in the graphics array discussed above.

Obviously, the graphics routines represent a bare minimum intended for research only.

5.3 Files

All files needed or produced by the code are stored in directory *files*. The standard files include:

- *input* - contains data for GMP and mesh generator,
- *output*,
- *result* - contains data for plotting convergence rates (output from automatic refinements routines or constructed by the user),
- *dump* - contains a dump of the data structure array,
- *mesh* - contains a dump with an intermediate mesh produced by the automatic refinements algorithm.

Except for the dump files that are open in routines *datstrs/dumpin* and *datstrs/dumpout*, the remaining files are open in routine *utilities/opfil* that assigns files numbers stored in common *commons/cinout* and it is executed in the very beginning of the main program. The i/o commands refer then to the files through the file numbers assigned in the opening routine. If an additional file is needed, the opening routine and the common block should be modified accordingly.

Sample input files for simple domains are stored in *files/inputs*.

5.4 How to prepare the data ?

Step 1: Set parameters $NDIMEN$ and $MAXEQNS$ in data structure module *module/ data_structure2D*. If the automatic refinements are to be run, $MAXEQNS$ has to be set to twice the number of equations to solve.

Step 2: Prepare the input file following GMP manual and sample input files. This includes setting the geometry, data for the mesh generator and boundary condition flags. Only Dirichlet (flag=1) and Neumann (flag=2) boundary conditions have been coded.

Step 3: If the solution is known, and the code is only used to study finite elements, code:

- the exact solution and its derivatives into routine *laplace/exact*,
- the coefficients of the differential operator in routine *laplace/getmat*.

The code is ready to be executed. The exact solution will be automatically used by the code to reproduce the corresponding right hand side of PDE, Dirichlet and Neumann BC data. This allows for a quick change of the exact solution and a study of the FE convergence. Several sample exact solutions can be found in routine *laplace/exact*.

In the case of an actual problem, the preparation of the data is more extensive. Routine calculating the exact solution has to be faked, and routines providing material as well as the *load data* for the problem:

- right-hand side f in *laplace/getf*,
- Dirichlet data in *laplace/dirichlet*,
- Neumann data in *laplace/getg*,

have to be provided by the user. If the geometry is simple, routines with a bunch of $if - then - else$ statements will do, otherwise the user has to develop his/her own data base for storing the material and load data. Creating a separate module is encouraged with the possibility of using GMP triangles and rectangles to identify subdomains with different, most of the time, piecewise constant data. Consult the interface with GMP.

Pure Neumann boundary conditions. In the case of pure Neumann boundary conditions, the solution *may be* determined only to *(linearized) rigid body modes* [18]. For the Laplace equation this reduces to a constant mode. The undetermined rigid body modes have to be eliminated setting up additional scalar conditions, as many as the number of

[18] Terminology coming from linear elasticity

linearly independent rigid body modes. This usually resembles setting up a Dirichlet BC at a vertex node but should not be confused with the Dirichlet BC's. If no scaling conditions are set, the corresponding stiffness matrix becomes singular. The frontal solver may return a solution with purely incidental rigid body modes which may be very confusing.

5.5 Running the code in the complex mode.

The following changes are required when switching from the real to the complex mode.

- Rename subdirectory *module_complex* to *module*.
- Enter subdirectory *commons* and rename *syscom.blk.complex* to *syscom.blk*
- Enter subdirectory *solver1* and rename routine *solve1.f.complex* to *solve1.f*
- Be sure to use option *laplz* when executing the 'make' command. The complex version of the frontal solver would be automatically linked.

Be sure to save the corresponding real mode files if you anticipate using the real mode again.

In the real version, the code uses only double precision reals and long integers. The corresponding implicit statement is stored in *commons/syscom* and included in all routines in the code. Except for the moduli, *no explicit type declarations are used in the code* ! The implicit statement allows for an easy switch to the complex version as all variables that correspond to complex-valued d.o.f. and complex numbers, *start with letter z*. One needs only then to change the *commons/syscom* to switch to the new implicit statement identifying all variables starting with z as complex numbers.

An example of a complex-valued problem is provided in subdirectory EM.

6 Automatic *hp*-Adaptivity

6.1 The main idea

We shall present only the main steps of the algorithm, coded in routines collected in directory *adapt*. For a detailed discussion, we refer to [11].

The main idea is based on minimizing a projection-based interpolation error, or the *hp*-interpolation error, as we have called it. There are two main points.

- The projection-based interpolation delivers optimal h- convergence, and *almost optimal* [19] p-convergence rates [10]. Thus, minimizing the interpolation error should deliver almost optimal meshes.
- Contrary to the actual approximation error, global in its nature, the interpolation error is determined *locally*, over one element at a time. This makes the minimization algorithm feasible.

Now, we need something to interpolate. In place of the unknown exact solution, the algorithm uses the solution (or its approximation) on a *globally hp-refined grid*. For practical, large problems, the global *hp*-refinement produces a huge mesh and the use of a direct solver on such a mesh is prohibitively expensive. The solution is obtained then using a two-grid or a multi-grid solver. We do not include the two-grid solver in this version of the code and, for now, the code is using the frontal solver only. We hope to deliver the two-grid solver in a forthcoming Version 2.1 of the code. Additionally, our experiments indicate that the solution on the fine grid may be replaced with a couple of smoothing operations only.

In summary, the mesh optimization algorithm produces a sequence of *coarse meshes* and an accompanying sequence of *fine meshes*. The error of the coarse grid solution is estimated simply by evaluating (the norm of) the difference between the fine and coarse mesh solutions.

We shall proceed now with a short discussion of a typical step of the algorithm performed in routine *adapt/hp_strategy*.

6.2 Discussion of one step of the algorithm

Suppose now that we are given an existing FE *coarse hp* mesh. We begin by performing a global *hp*-refinement, i.e. we split every element into four element sons, and we raise the order of approximation by one, uniformly throughout the whole mesh. This gives us the *fine mesh*. We solve then the problem of interest on the fine mesh.

[19] Up to a logarithmic factor

The main idea is now as follows. Given the fine mesh solution, we determine optimal refinement of the coarse grid, by minimizing the hp-interpolation error for the fine mesh solution interpolated on the next, optimal coarse mesh. The hp-interpolation is understood here in a generalized sense explained next.

Step 1: Linear (bilinear) interpolant. We interpolate fine mesh solution $u_{h/2,p+1}$ at coarse grid nodes, and subtract resulting linear (bilinear for quads) interpolant $u^1_{h,p}$ from the fine mesh solution.

Step 2: Optimal refinement of edges. We begin by computing for each edge the corresponding *edge interpolant*, obtained by projecting difference of $u_{h/2,p+1} - u^1_{h,p}$ on the space of edge shape functions $V_{hp}(e)$, vanishing at the edge endpoints,

$$\left\{ \begin{array}{l} w^e_{hp} \in V_{hp}(e) \\ \|w - w^1_{hp} - w^e_{hp}\| \rightarrow \min \ . \end{array} \right. \tag{6.77}$$

Ideally, we should use here the $H^{\frac{1}{2}}_{00}(e)$-norm. Instead, in the practical implementation, we have replaced it with a weighted H^1_0 norm,

$$\|u\|^2_e = \int_e \left(\frac{\partial u}{\partial s}\right)^2 \left(\frac{\partial s}{\partial \xi}\right)^{-1} ds = \int_0^1 \left(\frac{\partial u}{\partial \xi}\right)^2 d\xi \ . \tag{6.78}$$

Here $\boldsymbol{x} = \boldsymbol{x}(\xi)$, $\xi \in (0,1)$ is the FE parametrization for the edge, and

$$\frac{\partial s}{\partial \xi} = \sqrt{\sum_{i=1}^{2} \left(\frac{\partial x_i}{\partial \xi}\right)^2} \ . \tag{6.79}$$

Note that this weighted H^1_0 norm scales with the length of the edge the same way as the $H^{\frac{1}{2}}_{00}$ norm.

The projection problem is equivalent to solving a system of equations,

$$\left\{ \begin{array}{l} u^e_{hp} \in V_{hp}(e) \\ (u^e_{hp}, v_{hp})_e = ((u_{h/2,p+1} - u^1_{hp}), v_{hp})_e \quad \forall v_{hp} \in V_{hp}(e) \ , \end{array} \right. \tag{6.80}$$

where inner product $(\cdot,\cdot)_e$ corresponds to the norm discussed above.

Having solved the projection problem, we compute the corresponding projection error (squared),

$$\eta_e = \|u_{h/2,p+1} - u^1_{hp} - u^e_{hp}\|^2_e \ . \tag{6.81}$$

Next, we compute the corresponding *edge interpolants* corresponding to a p-refined edge, and a sequence of *competitive* h-refinements. By the competitive h refinement we understand here breaking the edge, and assigning such degrees of polynomial shape functions for the sons that the total number of d.o.f. for the h-refined element is the same as for the p-refined element. Thus, if the original order of approximation for the edge is p_e, then the new orders of approximation for the broken edge sons p_e^1, p_e^2, must satisfy the condition

$$p_e^1 + p_e^2 = p_e + 1 \, . \tag{6.82}$$

For example, for a linear edge element, the choice is between the quadratic element and two linear elements, for a quadratic edge, the choice is between the cubic edge and two refinements resulting in quadratic/linear of linear/quadratic approximations for the edge. In general, one has to solve $p_e + 1$ projection problems.

Once the optimal refinements for the preselected edges have been determined, we compute the corresponding projection (interpolation) errors and define the edge error indicators as the rates with which the interpolation errors decrease,

$$\eta_e = \|u_{h/2,p+1} - u_{hp}^1 - u_{hp}^e\|_e^2 - \|u_{h/2,p+1} - u_{hp}^1 - u_{opt}^e\|_e^2 \, . \tag{6.83}$$

Here u_{opt}^e denotes the edge interpolant corresponding to the optimal refinement of the edge, producing the smallest projection error. We determine next the maximum error $\eta_{max} = \max_e \eta_e$, and identify all edges e for which

$$\eta_e \geq \frac{1}{3}\eta_{max} \, . \tag{6.84}$$

These are the edges marked for a refinement. At this point, we know their desired optimal refinement and the corresponding interpolant (projection) u_{opt}^e.

Step 3: Breaking elements. Elements adjacent to edges marked for h-refinement are next h-refined. Triangular elements are just broken into four congruent element sons. In the case of quadrilaterals, we determine first an *isotropy flag* for each element by analyzing the difference

$$u_{h/2,p+1} - u_{hp} \, . \tag{6.85}$$

Roughly speaking, anisotropic refinements are selected only if the error function exhibits essentially one-dimensional behavior. This allows for an optimal refinement along boundary layers and edge singularities (3D), see [11] for details.

The 1-irregularity rule for meshes implies some involuntary h refinements of edges. This may apply to edges marked for p-refinement or marked for no refinement at all. We have

to revisit those edges and determine optimal orders of approximation for the resulted edge elements. Please see again [11] for details.

In summary, the decision on breaking elements is made exclusively by determining optimal refinements of edges and, for quads, analyzing the possible anisotropy of the error function. If none of element edges is selected for h-refinement, the element may only be p-refined.

Step 4: Determining optimal orders of approximation for refined elements. After the third step we know the topology of the mesh and optimal order of approximation for all element edges that overlap with edges of the coarse mesh. Given this information we set to determine optimal orders of approximation for all involved elements. The algorithm is again local - we consider one element of the coarse grid at the time. If none of the element edges has been changed, we do not change the approximation. If any of the element edges has been modified in any way, the element may have been left without any h-refinement, or it may have been h-refined in one (quads only) or two directions. In either of the cases, we need to determine new order of approximation for the element or its sons. We do it by monitoring the *interpolation error decrease rates*, defined as

$$\eta_K = \frac{|u_{h/2,p+1} - u^1_{hp} - u^2_{opt} - u^3_{hp}|^2_{H^1(K)} - |u_{h/2,p+1} - u^1_{hp} - u^2_{opt} - u^3_{opt}|^2_{H^1(K)}}{\Delta \text{nrdof}} \,. \tag{6.86}$$

Here

- $u^2_{opt} = \hat{u}^e_{opt}$ denotes a lift of edge interpolants corresponding to the optimal edge refinements determined in the previous step.
- u^3_{hp} denotes the projection of difference $u_{h/2,p+1} - u^1_{hp} - u^2_{opt}$ onto the space of element shape functions vanishing on element edges, $\mathcal{P}^p_{-1}(K)$,

$$\begin{aligned} &u^3_{hp} \in \mathcal{P}^p_{-1}(K) \\ &|u_{h/2,p+1} - u^1_{hp} - u^2_{opt} - u^3_{hp}|_{H^1(K)} \to \min \,. \end{aligned} \tag{6.87}$$

  This is again equivalent to solving a system of equations,

$$\left\{ \begin{aligned} &u^3_{hp} \in \mathcal{P}^p_{-1}(K) \\ &\int_K \boldsymbol{\nabla} u^3_{hp} \boldsymbol{\nabla} v_{hp} \, d\boldsymbol{x} = \int_K \boldsymbol{\nabla}(u_{h/2,p+1} - u^1_{hp} - u^2_{opt}) \boldsymbol{\nabla} v_{hp} \, d\boldsymbol{x} \quad \forall v_{hp} \in \mathcal{P}^p_{-1}(K) \,. \end{aligned} \right. \tag{6.88}$$

  Projection u^3_{hp} serves for a reference *element interior interpolant* with respect which the error decrease rate is defined.

- Δnrdof denotes the increase in the number of the element interior d.o.f., compared with the coarse mesh.

Finally, u^3_{opt} is determined by solving again the coarse element interior projection problem with $\mathcal{P}^p_{-1}(K)$ replaced with *space of element bubble functions* $V_{hp}(K)$ defined as the span of all (new coarse grid) basis functions with support in $\bar{K}$. This is done recursively, starting with the order of approximation for the element(s) interior(s) determined by orders for the edges and the *minimum rule*. The projection problem is solved, and local contributions to the projection error analyzed with the order p increased accordingly. [20] We monitor interpolation error decrease rate η_K defined above, and stop the iteration when two conditions are satisfied:

- The error corresponding to the new mesh must not exceed the error corresponding to the existing mesh.
- The rate is greater or equal than a prespecified minimum rate,

$$\eta_K \geq \eta_{min} \; . \tag{6.89}$$

In global terms, this is an investment problem. We invest with refinements only in those elements that produce sufficiently good error decrease rates. The minimum 'investment rate' η_{min} is determined by identifying the element for which the edge optimization has resulted in the largest error decrease. We then *presolve* the p-optimization problem for this element, increasing order p all the way to the order of the fine grid, and determining the maximum rate for the element. The minimum rate is then set to 1/3 of the maximum rate for the element.

The stopping criterion. In the end of the fourth step, we have determined the new coarse mesh and we continue the process until a stopping criterion is met. For the stopping criterion we use simply the error between the coarse and fine grid solutions measured in the H^1_0 seminorm, relative to the H^1_0-seminorm of the fine grid solution,

$$\frac{|u_{h/2,p+1} - u_{hp}|_{H^1(\Omega)}}{|u_{h/2,p+1}|_{H^1(\Omega)}} \leq \text{admissible error} \; . \tag{6.90}$$

6.3 Example: L-shape domain problem.

As an illustrating example, we present a solution of the classical L-shape domain problem. The Laplace equation was solved with Dirichlet boundary conditions imposed using the exact

[20] This is really a miniature p-adaptivity problem. Given an approximation on the boundary, determine the best p in the interior...

solution. The initial mesh consisted of four triangles and one quad, all of second order, and the mesh optimization procedure was continued until an error tolerance of 0.001 (.1 percent) was reached. The corresponding optimal *hp* mesh is shown in Fig. 16. Fig. 17 presents the corresponding convergence history, compared with the convergence history for just h-adaptivity using quadratic elements only. The scale on the horizontal axis corresponds to the optimal, exponential convergence rates predicted by the theory of *hp* discretizations [19], and confirmed by the algorithm.

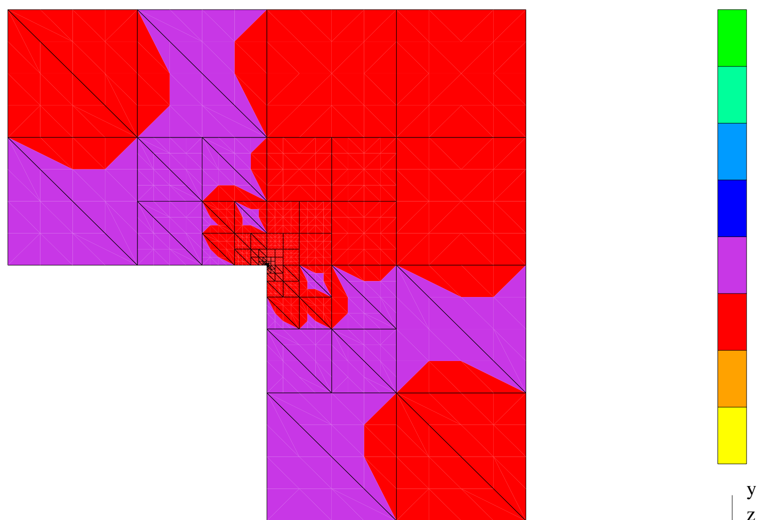


Figure 16: L-shape domain example: final, optimal *hp* mesh

Note that, in the preasymptotic range, the pure h-adaptive strategy can deliver slightly better results. This is related to the fact that our *hp* algorithm starts seeing the need for h-refinements only after a sufficient (spectral) information about the solution is available. In simple words, order p has to go up first ($p = 3$ minimum in this example) before the edges converging at the singular corner are broken. Asymptotically, however, *hp*-adaptivity always wins, as the fixed order discretization can only deliver algebraic rates of convergence. In problems with smoother solutions, the *hp* strategy wins earlier, see [11] for more examples.

6.4 Additional remarks

Is the exact solution known? If the exact solution is known, and you are using the code to study the convergence only, set parameter $nexact = 1$ in *adapt/hp_strategy* (do not forget to recompile the routine...). The exact solution will be used then to evaluate the H^1_0-error

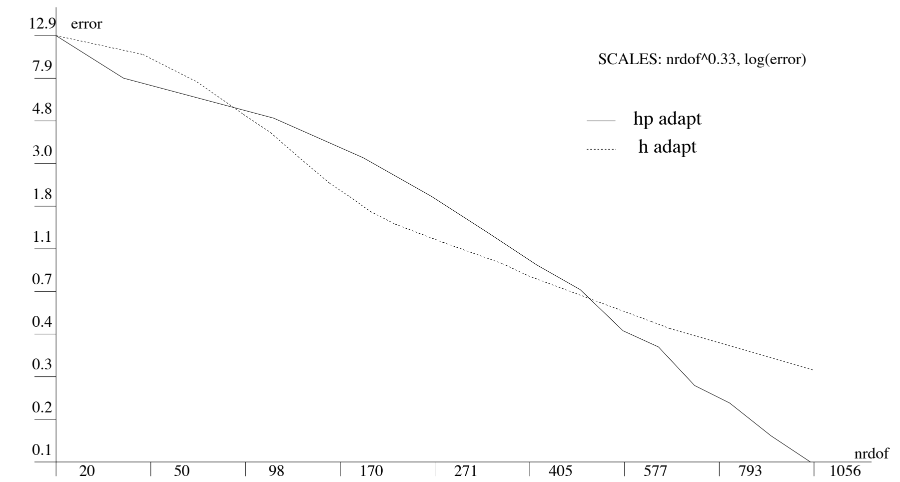


Figure 17: L-shape domain example: Convergence history

and *the exact errors* will be reported in the convergence history. Watch then for the *global effectivity index*,

$$\text{effectivity index} = \frac{\text{error estimate}}{\text{actual error}} \tag{6.91}$$

reported by the code.

If the exact solution is unknown, set $nexact = 0$. The error estimate will then be used in place of the actual error in the convergence graphs.

Maximum number of iterations, debugging. When invoking the automatic *hp* refinements routine, the user is asked to specify the maximum number of iterations and the admissible error level measured in percent of the norm of the solution. After each step of the mesh optimization, the current mesh is saved in file $files/dump$, and the problem can be restarted from the last mesh. This makes it convenient for debugging. A number of printing flags, including graphics control is available in the driver routine $adapt/hp_strategy$, and we invite the user to study the mesh optimization procedure, by displaying all consecutive meshes after various stages of the refinement.

Systems of equations and complex-valued problems. The automatic *hp*-adaptivity routines share the overall generality of the code. In the case of a system of equations, each quantity discussed in this section is defined by summing over the components of the solution.

We remind that all components of the solution are being approximated using the same mesh and no mixed discretizations are supported. In the case of complex-valued problems, all operations are simply done on complex numbers.

Automatic h-adaptivity. For a comparison, a simple h-adaptive strategy has been coded in routine *adapt/h_adapt*. The error estimation and adaptivity is again based on the interaction between the coarse and a fine mesh, this time obtained by performing only a global h-refinement. The element contributions to the global error,

$$\eta_K = \|u_{h/2} - u_h\|_{H_0^1(K)} , \tag{6.92}$$

are used as error indicator to pick the elements to be refined. The h-adaptivity done on meshes of quadratic elements, frequently delivers slightly better meshes in the preasymptotic range (see the discussion above), and provides itself a valuable tool for solving practical meshes.

7 Concluding Remarks

Compared with the first edition of the package [8], the code has undergone significant changes.

- The data structure has been significantly modified. In principle, the presented data structure supporting hp-adaptivity *can be superimposed on any existing classical finite element code!*. This should make the concept attractive for more professional implementations.
- The code supports a *unique* automatic hp-adaptivity algorithm that has been tested on a number of non-trivial examples, delivering optimal, exponential convergence rates. Most importantly, the algorithm delivers competitive meshes in the pre-asymptotic mesh, compared with h-adaptivity and quadratic elements.
- The code supports solving systems of equations and complex- valued problems.

What to expect in Version 2.1 ? We are in process of documenting the use of the code for several examples including besides the standard model Laplace equation, in-homogeneous isotropic and anisotropic heat conduction, elasticity, and axisymmetric electromagnetics radiation (antenna) problems.

The next version will include the use of a two-grid solver in the mesh optimization algorithm.

We intend to integrate the current implementation with our 2D electromagnetics code using the hp edge elements [18].

Wider perspective. Over the past years, the code has served a double purpose, both as a training and a research tool. The changes introduced in the code reflect to a large extent our struggle with 3D implementations and parallelization efforts on memory distributed machines.

User registration. The code represents an academic work and it is far from a professional implementation. It definitely contains many errors, and cannot be used as a 'black box' only. This is precisely, why we share the source code with all potential users of the code, and researchers interested in hp-adaptivity. All comments and 'bugs' detected in the code, will be greatly appreciated.

We do not update the code on a regular basis, but we do maintain a list of detected errors and changes made in the code. If you want to be informed about them and would

like to share your experience with other users, please send your name, E-mail address and affiliation to

Leszek Demkowicz <leszek@ticam.utexas.edu>

References

[1] E.B. Becker, G.F. Carey, and J.T. Oden. *Finite Elements: An Introduction.* Prentice-Hall, Inc. 1981.

[2] L. Demkowicz, J.T. Oden, and Ph. Devloo, "On h-Type Mesh Refinement Strategy Based on a Minimization of Interpolation Error", *Computer Methods in Applied Mechanics and Engineering*, **53**, 67-89, 1985.

[3] L. Demkowicz, L. J.T. Oden, and W. Rachowicz, "Toward a Universal *hp* Adaptive Finite Element Strategy, Part 1. Constrained Approximation and Data Structure", *Computer Methods in Applied Mechanics and Engineering*, **77**, 79-112, 1989.

[4] L. Demkowicz, A. Karafiat, and J.T. Oden, "Solution of Elastic Scattering Problems in Linear Acoustics Using *hp* Boundary Element Method", *Computer Methods in Applied Mechanics and Engineering*, **101**, 251-282, (Proceedings of Second Workshop on Reliability and Adaptive Methods in Computational Mechanics, Cracow, October 1991, eds. L. Demkowicz, J.T. Oden and I.Babuska).

[5] L. Demkowicz, A. Bajer, and K. Banas, "Geometrical Modeling Package", *TICOM REPORT 92-06*, August 1992, The University of Texas at Austin, Austin TX 78712.

[6] L. Demkowicz, K. Gerdes, C. Schwab, A. Bajer, and T. Walsh, "HP90: A General and Flexible Fortran 90 *hp*-FE Code", *Computing and Visualization in Science*, **1**, 145-163, 1998.

[7] L. Demkowicz, Chang-wan Kim, "1D hp-Adaptive Finite Element Package. Fortran 90 Implementation (1Dhp90)", *TICAM Report 99-38*

[8] L. Demkowicz, T. Walsh, K. Gerdes, and A. Bajer, "2D hp-Adaptive Finite Element Package. Fortran 90 Implementation (2Dhp90)", TICAM Report 98-14, June 1998.

[9] L. Demkowicz, P. Monk, L. Vardapetyan, and W. Rachowicz. "De Rham Diagram for hp Finite Element Spaces", *TICAM Report 99-06*, *Mathematics and Computers with Applications*, **39**, 7-8, 29-38, 2000.

[10] L. Demkowicz and I. Babuška, "Optimal p Interpolation Error Estimates for Edge Finite Elements of Variable Order in 2D", *TICAM Report 01-11*, submitted to *SIAM Journal on Numerical Analysis*, 2001.

[11] L. Demkowicz, W. Rachowicz, and Ph. Devloo, "A Fully Automatic *hp* Adaptivity", *TICAM Report 01-28*, accepted to *Journal of Scientific Computing.*

[12] L. Demkowicz, D. Pardo, "The Ultimate Data Structure for Three Dimensional, Anisotropic *hp* Refinements', *TICAM Report*, in preparation.

[13] L.P. Meissner, *Fortran 90*, PWS Publishing Company, Boston 1995.

[14] J. T. Oden, L. Demkowicz, W. Rachowicz and T. A. Westermann, "Towards a Universal *h-p* Adaptive Finite Element Strategy, Part 2. *A Posteriori* Error Estimation," *Comp. Meth. Appl. Meth. Eng.*, **77**, 113-180, 1989.

[15] J.T. Oden, and L.F. Demkowicz, *Applied Functional Analysis for Science and Engineering*, CRC Press, Boca Raton, 1996.

[16] Rheinboldt, W.C., and Mesztenyi, C.K., "On a Data Structure for Adaptive Finite Element Mesh Refinements", *ACM Transactions on Mathematical Software* , Vol.6, No. 2, 166-187, June 1980.

[17] W. Rachowicz, J.T. Oden, and L. Demkowicz, "Toward a Universal h-p Adaptive Finite Element Strategy, Part 3. Design of h-p Meshes," *Computer Methods in Applied Mechanics and Engineering*, **77**, 181-212, 1989.

[18] W. Rachowicz and L. Demkowicz, "A Two-Dimensional hp-Adaptive Finite Element Package for Electromagnetics", *TICAM Report 98-15*, July 1998, *Computer Methods in Applied Mechanics and Engineering*, **187**, 1-2, 307-337, 2000.

[19] Ch. Schwab, *p and hp-Finite Element Methods*, Clarendon Press, Oxford 1998.