1D *hp*-ADAPTIVE FINITE ELEMENT PACKAGE FORTRAN 90 IMPLEMENTATION (1Dhp90)

Leszek Demkowicz and Chang-Wan Kim

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

Abstract

This manual provides a description of a 1D *hp*-adaptive finite element code for the solution of a class of two-point boundary-value problems. The implementation is based on hierarchial shape functions and allows for both $h-$ and $p-$ refinements of the mesh, based on *a posteriori* error estimation. The code includes an automatic *hp*-adaptive strategy. Numerical results for several test problems illustrate the method.

Contents

1 Introduction

The finite element method is a general technique for constructing approximate solutions to boundary value problems. The method involves dividing the domain of the solution into a finite number of simple subdomains, the finite elements, and using variational concepts to construct an approximation of the solution over the collection of finite elements. Because of the generality and richness of the idea underlying the method, it has been used with a remarkable success in solving a wide range of problems in virtual all areas of engineering and mathematical physics.

The goals of this manual are :

- to give a brief introduction to fundamental ideas of variational formulation, Galerkin method, *a posteriori* error estimation, in context of a one dimensional finite element method,
- to introduce the reader to the concept of *hp*-adaptive Finite Element Methods,
- to provide a minimal user information for the package.

Consequently, we have decided to write this document in a format of lecture notes. Indeed, it is our intention to use the notes in the ASE384P/EM 394F/CAM394F(Finite Element Methods) [4] class taught in the ASE/EM Department and the CAM programs at the University of Texas at Austin.

2 A General Two-Point Boundary Value Problem

2.1 Strong, or classical, form of the problem

We begin by considering a two-point boundary value problem of finding a function $u = u(x), x \in [0, l]$. The strong form of the boundary-value problem consists of a second order, ordinary differential equation,

$$-(a(x)u(x)')' + b(x)u'(x) + c(x)u(x) = f(x) \quad x \in (0, l), \tag{2.1}$$

accompanied at each of the endpoints $x = 0$ or $x = l$ with one of three possible boundary conditions:

- Dirichlet boundary condition,

$$u(0) = \gamma_0 \quad or \quad u(l) = \gamma_l, \tag{2.2}$$

- Neumann boundary condition,

$$-a(0)u'(0) = \gamma_0 \quad or \quad a(l)u'(l) = \gamma_l, \tag{2.3}$$

- Cauchy boundary condition

$$-a(0)u'(0) + \beta_0 u(0) = \gamma_0 \quad or \quad a(l)u'(l) + \beta_l u(l) = \gamma_l. \tag{2.4}$$

The Neumann boundary condition is just a special case of Cauchy boundary condition with constant $\beta = 0$. For discussion of other possible boundary conditions including periodic boundary conditions, see [4].

The data for the problem consist thus of :

- coefficients of the differential operator $a(x), b(x), c(x)$ (the *material constants*),
- right-hand side $f(x)$ (the *load*),
- boundary conditions data β, γ.

2.2 Weak (variational) formulation of the problem

The weak formulation is a reformulation of the strong form and it is from this weak form that the FE approach is established. Whenever a smooth classical solution to a problem exists, it is also the solution of the weak problem. To establish the weak form of the strong form given by equation (2.1), multiply (2.1) by an arbitary, so called, test function $v(x)$, and integrate over interval $(0, l)$. We obtain

$$\int_0^l \left[-(a(x)u'(x))' + b(x)u'(x) + c(x)u(x)\right] v(x)\, dx = \int_0^l f(x)v(x)\, dx. \tag{2.5}$$

Next, we integrate the first term by parts,

$$\int_0^l au'v'\, dx - au'v\,|_0^l + \int_0^l bu'v\, dx + \int_0^l cuv\, dx = \int_0^l f(x)v\, dx\,. \tag{2.6}$$

At this point, further derivation depends upon the kind of the boundary conditions being used. In the case of Dirichlet boundary condition, we eliminate the boundary term by restricting ourselves to only those test functions that vanish at that point. For example, if we assume Dirichlet boundary condition at $x = 0$,

$$u(0) = \gamma(0), \tag{2.7}$$

we have to assume that $v(0) = 0$.

In the case of Cauchy or Neumann boundary condition, we *build the boundary condition into the formulation* by using it to calculate the derivative in terms of the boundary data, and (in the case of Cauchy boundary condition) still unknown solution at that point. Thus in the case of Dirichlet boundary condition at $x = 0$ and Cauchy boundary condition at $x = l$, we get

$$\begin{cases} \text{Find } u(x), u(0) = \gamma_0 \\ \int_0^l (au'v' + bu'v + cuv)\, dx + \beta_l u(l)v(l) = \int_0^l f(x)v\, dx + \gamma_l v(l) \\ \quad \text{for every test function } v(x) \text{ such that } v(0) = 0. \end{cases} \tag{2.8}$$

In the case of Neumann boundary condition, at $x = l$, constant $\beta_l = 0$, and the boundary term simply vanishes.

Notice that we have kept all terms involving solution $u(x)$ on the left-hand side, and the terms involving the test function v only, have been moved to the right-hand side of the equation.

The weak (variational) formulation can be shown to be *equivalent* [1] to the classical form of the boundary-value problem. We integrate back by parts and use the Fourier lemma argument to *recover* the differential equation and Cauchy (Neumann) boundary condition at $x = l$. Notice that there is no need to recover Dirichlet boundary condition, as it has been simply rewritten into the weak formulation.

A precise mathematical setting is obtained by introducing the *Sobolev space* of the first order $H^1(0,l)$, consisting of functions that are, together with their derivatives, square integrable,

$$H^1(0,l) = \{v(x) \ : \ \int_0^l v^2 \, dx, \int_0^l (v')^2 \, dx < \infty\}. \tag{2.9}$$

Next, we identify the subspace of *kinematically admissible functions* V, satisfying the *homogeneous* Dirichlet boundary condition,

$$V = \{v \in H^1(0,l) \ : \ v(0) = 0\}. \tag{2.10}$$

The *set* of functions satisfying the *nonhomogeneous* Dirichlet boundary condition has a more complicated algebraic structure of an *affine subspace* and can be identified as the collection of all sums $u_0 + v$ where $u_0 \in H^1(0,l)$ is a *lift* of the boundary data, and v is an arbitrary function from $H^1(0,l)$ satisfying the homogeneous Dirichlet boundary conditions:

$$\{u \in H^1(0,l) \ : \ u(0) = \gamma_0\} = u_0 + V := \{u_0 + v \ : \ v \in V\}. \tag{2.11}$$

The variational formulation can now be written in the form of the *abstract variational boundary-value problem*

$$\begin{cases} \text{Find } u \in u_0 + V, \text{ such that} \\ b(u,v) = l(v), \quad \forall v \in V. \end{cases} \tag{2.12}$$

Here,

$$b(u,v) = \int_0^l (au'v' + bu'v + cuv) \, dx + \beta_l u(l) v(l) \tag{2.13}$$

is a *bilinear form* of arguments u and v, and

$$l(v) = \int_0^l f(x) v \, dx + \gamma_l v(l) \tag{2.14}$$

is a *linear form* of test function v.

Equivalently speaking, once we have found a particular function $u_0 \in H^1(0,l)$ that satisfies the nonhomogeneous Dirichlet data, we can make the substitution $u = u_0 + w$ where

[1] up to regularity assumptions on the solution

$w \in V$ satisfies the homogeneous Dirichlet boundary conditions, and try to determine the perturbation w. The corresponding abstract formulation is then as follows.

$$\begin{cases} \text{Find } w \in V, \text{ such that:} \\ b(w, v) = l(v) - b(u_0, v), \quad \forall v \in V \end{cases} \tag{2.15}$$

For the sake of simplicity of presentation, we shall use the particular choice of the boundary conditions throughout the rest of these notes. All other cases can be treated in a completely analogous way.

3 Finite Element Method

3.1 Galerkin approximation

With these preliminaries behind us, we are ready to consider Galerkin's method for constructing approximate solutions to the variational boundary-value problem. Galerkin's method consists of seeking an approximate solution to variational boundary-value problem in a finite-dimensional subspace V_h of space V.

This procedure leads to the following *approximate variational boundary-value problem.*

$$\begin{cases} \text{Find } u_h \in u_0 + V_h, \text{ such that:} \\ b(u_h, v_h) = l(v_h), \quad \forall v_h \in V_h \end{cases} \tag{3.16}$$

We introduce a finite set of basis functions $e_{h1}, e_{h2}, \ldots, e_{hN}$ in V that span a finite dimensional subspace of test functions V_h in V. We then seek the approximate function $u_h \in u_0 + V_h$ in the form:

$$u_h(x) = u_0 + \sum_{k=1}^{N_h} u_{hk} e_{hk} \, . \tag{3.17}$$

The unknown coefficients $u_{hk}, k = 1, 2, ..., N_h$ are called *global degrees of freedom.* Substituting the linear combination into the variational boundary-value problem (3.16), and setting the test functions to the basis functions $v = e_{hl}, l = 1, 2, ..., N_h$, we arrive at the algebraic system of equations.

$$\begin{cases} \text{Find } u_{hk}, k = 1, 2, ..., N_h, \text{ such that:} \\ b(u_0 + \sum_{k=1}^{N_h} u_{hk} e_{hk}, , e_{hl}) = l(e_{hl}), \quad l = 1, 2, ..., N_h \end{cases} \tag{3.18}$$

The approximate solution depends only on the space V_h and is independent of the basis functions e_{hk}. In order to simplify the notation, we shall drop now the *approximate space (mesh) index h* remembering 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, we are led to the following *system of linear equations.*

$$\begin{cases} \text{Find } u_k, k = 1, \ldots, N, \text{ such that:} \\ \sum_{k=1}^{N} u_k S_{kl} = L_l^{mod}, \quad l = 1, \ldots, N \end{cases} \tag{3.19}$$

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

$$S_{kl} = b(e_k, e_l) = \int_0^l (a\frac{de_k}{dx}\frac{de_l}{dx}dx + b\frac{de_k}{dx}e_l + ce_k e_l)\,dx \tag{3.20}$$

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

$$\begin{aligned} L_l^{mod} &= l(e_l) - b(u_0, e_l) \\ &= \int_0^l f e_l\,dx + \gamma_l e_l(l)) - \int_0^l (a\frac{du_0}{dx}\frac{de_l}{dx}dx + b\frac{du_0}{dx}e_l + cu_0 e_l)\,dx \end{aligned} \tag{3.21}$$

The array:

$$L_l = l(e_l) = \int_0^l f e_l\,dx + \gamma_l e_l(l) \tag{3.22}$$

is called the (original) *load vector*. Notice the two different meanings of letter l.

3.2 Finite Element Method

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 $(0, l)$ is partitioned into disjoint subdomains called *finite elements*. Next, for each element K, we introduce the corresponding *shape functions* χ_K which eventually are *glued* together into the globally defined basis functions e_k in the Galerkin method.[2] It is the construction of the basis functions that distinguishes the FEM from other Galerkin approximations. We begin our presentation with a discussion of the fundamental notions of the *master element*, the *isoparametric element*, and the *finite element space*. We shall recall the construction of the Galerkin basis functions through *the element shape functions* and, finally, introduce the notion of the *hp interpolation*.

3.2.1 1D master element of an arbitary order

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

$$X(\hat{K}) = \mathcal{P}^p(\hat{K})\,. \tag{3.23}$$

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

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

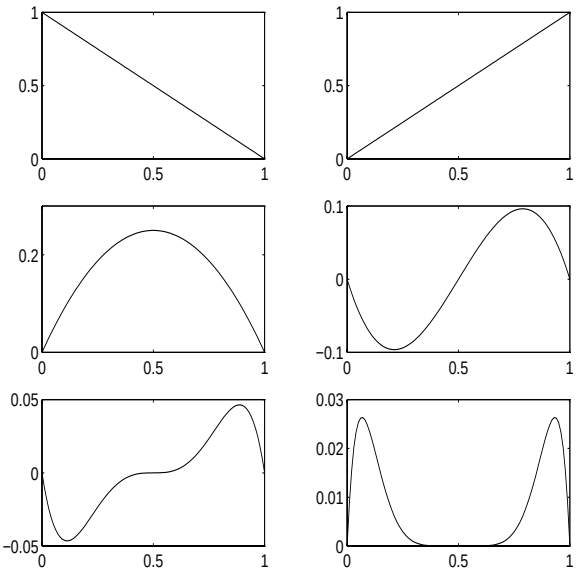

Figure 1: 1-D Hierarchical shape function

defined as follows.

$$\begin{aligned}
\hat{\chi}_1(\xi) &= 1-\xi \\
\hat{\chi}_2(\xi) &= \xi \\
\hat{\chi}_3(\xi) &= (1-\xi)\xi \\
\hat{\chi}_l(\xi) &= (1-\xi)\xi(2\xi-1)^{l-3},\ l=4,\ldots,p+1
\end{aligned} \tag{3.24}$$

The functions admit a simple recursive formula:

$$\begin{aligned}
\hat{\chi}_1(\xi) &= 1-\xi \\
\hat{\chi}_2(\xi) &= \xi \\
\hat{\chi}_3(\xi) &= \hat{\chi}_1(\xi) * \hat{\chi}_2(\xi) \\
\hat{\chi}_l(\xi) &= \hat{\chi}_{l-1}(\xi) * (\hat{\chi}_2(\xi) - \hat{\chi}_1(\xi)),\ l=4,\ldots,p+1
\end{aligned} \tag{3.25}$$

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*, see Fig 1.

Finally, we introduce the notion of the $hp-interpolation$ over the master element. Given a continuous function $\hat{u}(\xi), \xi \in [0,1]$, we define its hp *-interpolant* $\hat{u}_{hp}$ as the sum of the standard linear interpolant $\hat{u}^1_{hp}$ coinciding with function $\hat{u}$ at the endpoints, and a H^1_0-projection of the difference $\hat{u}-\hat{u}^1_{hp}$ onto the span of the bubble functions introduced above. More precisely:

$$\hat{u}_{hp} = \hat{u}^1_{hp} + \hat{u}^2_{hp} \tag{3.26}$$

where

$$\hat{u}^1_{hp}(\xi) = \hat{u}(0)\hat{\chi}_1(\xi) + \hat{u}(1)\hat{\chi}_2(\xi) \tag{3.27}$$

and

$$\hat{u}^2_{hp} = \sum_{j=3}^{p+1} u^2_j \hat{\chi}_j(\xi) \tag{3.28}$$

where the coefficients u^2_j are determined by solving the following system of equations.

$$\sum_{j=3}^{p+1} u^2_j \int_0^1 \frac{d\hat{\chi}_j}{d\xi}\frac{d\hat{\chi}_k}{d\xi} = \int_0^1 \frac{d(\hat{u} - \hat{u}^1_{hp})}{d\xi}\frac{d\hat{\chi}_k}{d\xi}, \quad k = 3, \ldots, p+1 \tag{3.29}$$

3.2.2 A 1D parametric element of arbitrary order

We consider now an arbitrary closed interval $K = [x_L, x_R] \subset [0, l]$ and assume that K is the image of the master element through some map x_K:

$$\hat{K} = [0,1] \ni \xi \to x = x_K(\xi) \in K \tag{3.30}$$

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

$$\begin{aligned} x_K(\xi) &= x_L\hat{\chi}(\xi) + x_R\hat{\chi}(\xi) \\ &= x_L(1-\xi) + x_R\xi \\ &= x_L + \xi(x_R - x_L) \\ &= x_L + \xi h_K \end{aligned} \tag{3.31}$$

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.

$$\begin{aligned} X(K) &= \{\hat{u} \circ x_K^{-1},\ \hat{u} \in X(\hat{K})\} \\ &= \{u(x) = \hat{u}(\xi) \text{ where } x_K(\xi) = x \text{ and } \hat{u} \in X(\hat{K})\} \end{aligned} \tag{3.32}$$

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

$$\chi_k(x) = \hat{\chi}_k(\xi) \text{ where } x_K(\xi) = x, \quad k = 1, \ldots, p+1 \,. \tag{3.33}$$

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, most of the time, it is convenient that map x_K is specified using the master element shape functions, i.e. it is a polynomial of order p:

$$x_K(\xi) = \sum_{j=1}^{p+1} x_{Kj} \hat{\chi}(\xi) \tag{3.34}$$

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.

In our implementation, we have restricted ourselves to the affine elements only. Consequently, throughout the rest of this presentation, we shall assume that the summation in 3.34 extends over the linear shape functions only.

The *hp*-interpolation operator can now be generalized to an arbitrary element K. The idea is to perform the interpolation procedure *always* on the master element. Given a continuous function $u(x), x \in K$, defined over the element K, we compose it with the map transforming the master element into element K,

$$\hat{u}(\xi) = (u \circ x_K)(\xi) = u(x_K(\xi)), \tag{3.35}$$

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

$$\hat{u}_{hp}(\xi) = \sum_{j=1}^{p+1} u_j \hat{\chi}_j(\xi) \,. \tag{3.36}$$

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,

$$u_{hp}(x) = \sum_{j=1}^{p+1} u_j \chi_j(x) \,. \tag{3.37}$$

Practically that means only that the coefficients u_j must be determined by solving the appropriate system of equations on the master element.

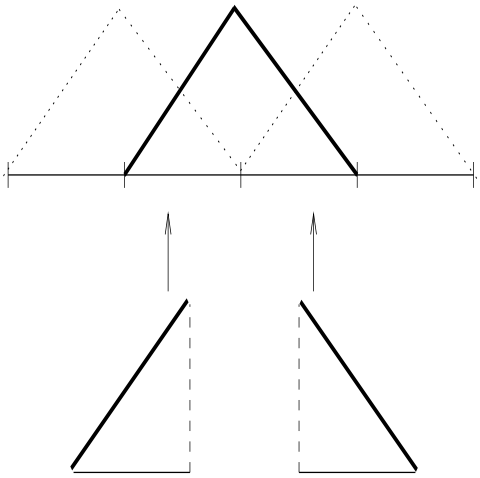

Figure 2: Construction of the vertex nodes basis functions

3.2.3 1D *hp* finite element space.

Let now interval $(0, l)$ 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 $0 = x_0 < x_1 < \ldots < x_N < x_{N+1} = l$ will be called the *vertex nodes*. We define the 1D *hp finite element space* X_h [3], as the collection of all functions that are globally continuous, and whose restrictions to element K live in the element space of shape functions.

$$X_h = \{u_h(x) \ : \ u \text{ is continuous and } u|_K \in X(K), \text{ for every element } K \} \tag{3.38}$$

The global basis functions are classified into two groups:

- the vertex nodes basis functions, and
- the 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 bubble basis functions is much easier. As the element bubble shape functions vanish at the element endpoints, we need simply to extend them only by

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

the zero function elsewhere. The *support*[4] 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:

$$u_{hp}(x) = u_{hp}^K(x) \quad \text{where } x \in K \tag{3.39}$$

with u_{hp}^K denoting 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.

3.2.4 Element stiffness matrix and load vector

Having selected an appropriate set of shape functions, we calculate the *element stiffness matrix* and *load vector*.

$$\begin{aligned} S_{ij}^K &:= \int_{x_L}^{x_R} (a\frac{d\chi_i}{dx}\frac{d\chi_j}{dx}dx + b\frac{d\chi_i}{dx}\chi_j + c\chi_i\chi_j)\,dx \\ L_j^K &:= \int_{x_L}^{x_R} f\chi_j\,dx \end{aligned} \tag{3.40}$$

The local (master element) coordinate system proves to be more convenient in the derivation of the element matrices. The matrices are calculated in the local coordinate system by using the chain rule:

$$\begin{aligned} dx &= \frac{dx}{d\xi}d\xi \\ \frac{d\chi_k}{dx} &= \frac{d\chi_k}{d\xi}\frac{d\xi}{dx}\,. \end{aligned} \tag{3.41}$$

After switching to the master element coordinate ξ, we obtain

$$\begin{aligned} S_{ij}^K &= \int_0^1 (\hat{a}\frac{d\hat{\chi}_i}{d\xi}\frac{d\xi}{dx}\frac{d\hat{\chi}_j}{d\xi}\frac{d\xi}{dx} + \hat{b}\frac{d\hat{\chi}_i}{d\xi}\hat{\chi}_j + \hat{c}\hat{\chi}_i\hat{\chi}_j)\frac{dx}{d\xi}\,d\xi \\ L_j^K &= \int_0^1 \hat{f}\hat{\chi}_j\,d\xi\,. \end{aligned} \tag{3.42}$$

As before, symbol $\hat{}$ indicates the composition with the element map, e.g.,

$$\hat{a}(\xi) = a(x_K(\xi)). \tag{3.43}$$

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

The evaluation of the integrals in (3.42) is performed using *numerical integration.* In most finite element calculations, *Gauss quadrature* rules are used. We note that the Gauss rule of order N will integrate exactly polynomials of degree $2N-1$. The explicit formulas for evaluation (3.42) using Gauss quadrature are

$$\begin{aligned} S_{ij}^K &= \sum_{l=1}^{N_l} \left\{ a(x_l) \frac{d\hat{e}_k}{d\xi}(\xi_l) \frac{d\hat{e}_l}{d\xi}(\xi_l) \frac{d\xi}{dx}\frac{d\xi}{dx} + b(x_l) \frac{d\hat{e}_k}{d\xi}(\xi_l) \hat{e}_l(\xi_l) \frac{d\xi}{dx} + c(x_l) \hat{e}_k(\xi_l) \hat{e}_l(\xi_l) \right\} \frac{dx}{d\xi} w_l \\ L_i^K &= \sum_{l=1}^{N_l} f(x_l) \hat{e}_l(\xi_l) \frac{dx}{d\xi} w_l \end{aligned} \tag{3.44}$$

where, $x_l = x_K(\xi_l)$ is the value of element map x_K calculated at integration point ξ_l. Notice that for an affine element K, $\frac{dx}{d\xi} = h_K$ and $\frac{d\xi}{dx} = h_K^{-1}$, are *independent* of integration point ξ_l.

3.2.5 Taking into account the boundary conditions. Modified element matrices

In the case of the first and the last element, element stiffness matrix and load vector must be modified to incorporate changes due to the boundary conditions.

Dirichlet boundary condition at $x = 0$. We use the first element linear shape function ξ_1, premultiplied by γ_0, to construct the lift of the boundary conditions data. Instead of eliminating the first shape *test* function, we rewrite the first row of the stiffness matrix and the load vector in such a form that would *implicitly* enforce condition $u(0) = \gamma_0$. The original element matrices,

$$\begin{bmatrix} S_{11} & S_{12} & \cdots & S_{1n} \\ S_{21} & S_{22} & \cdots & S_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ S_{n1} & S_{n2} & \cdots & S_{nn} \end{bmatrix} \tag{3.45}$$

$$\begin{bmatrix} L_1 \\ L_2 \\ \vdots \\ L_n \end{bmatrix} \tag{3.46}$$

get replaced with the *modified matrices* of the form:

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & S_{22} & \cdots & S_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & S_{n2} & \cdots & S_{nn} \end{bmatrix} \tag{3.47}$$

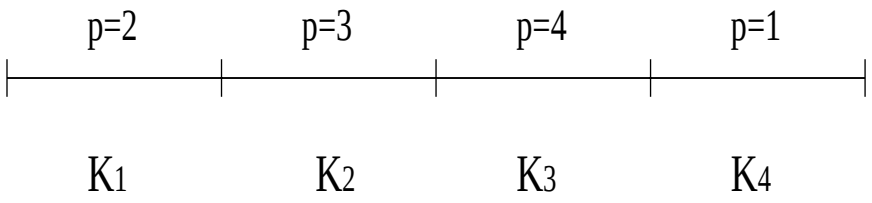

Figure 3: Finite element mesh

$$\begin{bmatrix} \gamma_0 \\ L_2 - \gamma_0 S_{21} \\ \vdots \\ L_n - \gamma_0 S_{n1} \end{bmatrix} \tag{3.48}$$

Here $n = p + 1$ is the number of the *element degrees of freedom.*

Cauchy (Neumann) boundary condition at $x = l$. Addition of the boundary terms to the bilinear and linear forms results in the following modified element matrices.

$$\begin{bmatrix} S_{11} & S_{12} & \cdots & S_{1n} \\ S_{21} & S_{22} + \beta & \cdots & S_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ S_{n1} & S_{n2} & \cdots & S_{nn} \end{bmatrix} \tag{3.49}$$

$$\begin{bmatrix} L_1 \\ L_2 + \gamma_l \\ \vdots \\ L_n \end{bmatrix} \tag{3.50}$$

3.2.6 Global stiffness matrix and load vector, the assembling procedure

As global basis functions are constructed by gluing together element shape functions, the additivity of integrals [4] implies that the entries of the global matrices are calculated by accumulating the corresponding contributions from element matrices. This *assembling procedure* is pivotal in the Finite Element Method. We shall illustrate it with a simple mesh consisting of four elements shown in Fig 3. The mesh consists of 4 elements denumerated from the left to the right, K_1, K_2, K_3 and K_4. This *ordering of elements* induces the so called *natural order of nodes.* We begin by listing all nodes of the first element, then continue with those nodes of the second element that have not been listed yet, and so on. The order for

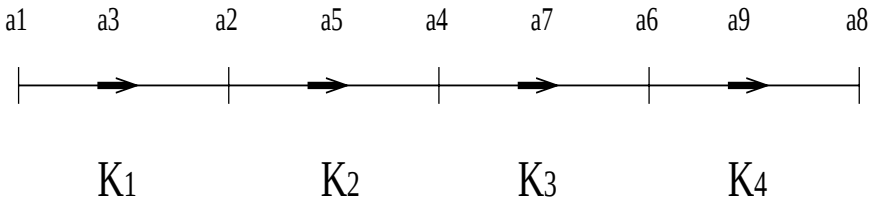

Figure 4: Natural order of nodes

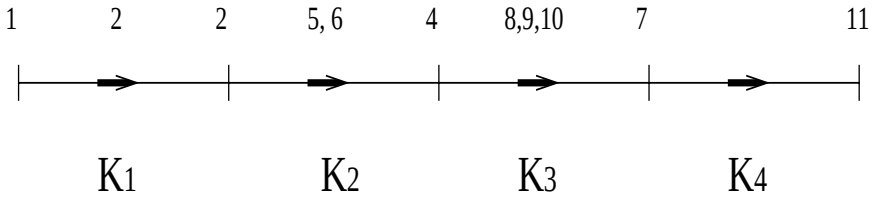

Figure 5: Natural order of global d.o.f.

nodes is depicted in Fig 4. Notice that, as the right vertex node of the element is listed *before* its middle node, the natural order of nodes is *not equivalent* to denumerating simply nodes from the left to the right. Finally, the natural order of nodes implies the corresponding *natural order of global degrees of freedom* obtained by listing the nodal d.o.f., node after node. The ordering is depicted in Fig 5. Notice the variable number of degrees of freedom associated with the middle nodes and, in particular, no degrees of freedom associated with the last middle node at all.

We are now ready to discuss the assembling procedure. We proceed with one element at a time. The first *element connectivites* indicating the global d.o.f. numbers assigned to the local ones are $1, 2, 3$. Consequently, after assembling the first element matrices, the global matrices will look as follows.

$$\begin{bmatrix} S^1_{11} & S^1_{12} & S^1_{13} & . & . & . & . & . & . & . & . \\ S^1_{21} & S^1_{22} & S^1_{23} & . & . & . & . & . & . & . & . \\ S^1_{31} & S^1_{32} & S^1_{33} & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . \end{bmatrix} \tag{3.51}$$

$$\begin{bmatrix} L_1^1 \\ L_2^1 \\ L_3^1 \\ . \\ . \\ . \\ . \\ . \\ . \\ . \\ . \\ . \end{bmatrix} \tag{3.52}$$

Dotted entries indicate zeros. Similarly, the second element connectivites are 2,4,5, and 6. After assembling the second element matrices, we obtain

$$\begin{bmatrix} S_{11}^1 & S_{12}^1 & S_{13}^1 & . & . & . & . & . & . & . & . \\ S_{21}^1 & S_{22}^1 + S_{11}^2 & S_{23}^1 & S_{12}^2 & S_{13}^2 & S_{14}^2 & . & . & . & . & . \\ S_{31}^1 & S_{32}^1 & S_{33}^1 & . & . & . & . & . & . & . & . \\ . & S_{21}^2 & . & S_{22}^2 & S_{23}^2 & S_{24}^2 & . & . & . & . & . \\ . & S_{31}^2 & . & S_{32}^2 & S_{33}^2 & S_{34}^2 & . & . & . & . & . \\ . & S_{41}^2 & . & S_{42}^2 & S_{43}^2 & S_{44}^2 & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . \end{bmatrix} \tag{3.53}$$

$$\begin{bmatrix} L_1^1 \\ L_2^1 + L_1^2 \\ L_3^1 \\ L_2^2 \\ L_3^2 \\ L_4^2 \\ \vdots \end{bmatrix} \tag{3.54}$$

The (3.55) and (3.56) show the situation after assembling the third and the fourth element contributions.

$$
\begin{bmatrix}
S_{11}^1 & S_{12}^1 & S_{13}^1 & . & . & . & . & . & . & . & . \\
S_{21}^1 & S_{22}^1 + S_{11}^2 & S_{23}^1 & S_{12}^2 & S_{13}^2 & S_{14}^2 & . & . & . & . & . \\
S_{31}^1 & S_{32}^1 & S_{33}^1 & . & . & . & . & . & . & . & . \\
. & S_{21}^2 & . & S_{22}^2 + S_{11}^3 & S_{23}^2 & S_{24}^2 & S_{12}^3 & S_{13}^3 & S_{14}^3 & S_{15}^3 & . \\
. & S_{31}^2 & . & S_{32}^2 & S_{33}^2 & S_{34}^2 & . & . & . & . & . \\
. & S_{41}^2 & . & S_{42}^2 & S_{43}^2 & S_{44}^2 & . & . & . & . & . \\
. & . & . & S_{21}^3 & . & . & S_{22}^3 + S_{11}^4 & S_{23}^3 & S_{24}^3 & S_{25}^3 & S_{12}^4 \\
. & . & . & S_{31}^3 & . & . & S_{32}^3 & S_{33}^3 & S_{34}^3 & S_{35}^3 & . \\
. & . & . & S_{41}^3 & . & . & S_{42}^3 & S_{43}^3 & S_{44}^3 & S_{45}^3 & . \\
. & . & . & S_{51}^3 & . & . & S_{52}^3 & S_{53}^3 & S_{54}^3 & S_{55}^3 & . \\
. & . & . & . & . & . & S_{21}^4 & . & . & . & S_{22}^4
\end{bmatrix} \tag{3.55}
$$

$$
\begin{bmatrix}
L_1^1 \\
L_2^1 + L_1^2 \\
L_3^1 \\
L_2^2 + L_1^3 \\
L_3^3 \\
L_4^4 \\
L_2^3 + L_1^4 \\
L_3^3 \\
L_4^3 \\
L_5^3 \\
L_2^4
\end{bmatrix} \tag{3.56}
$$

3.3 Structure of a classical FE code

We are now in a position to sketch the *computer flow diagram of a classical finite element code* as in Fig 6.

A sequence of input data is read or generated in the *preprocessor* part. This includes geometry and finite element mesh data such as number of elements, element connectivity information, node coordinates, initial order of approximation. This preprocessor may perform an automatic division of the domain into elements following some rules specified by user, and it may also provide graphical information on elements and nodes. The *processor* part includes generation of the element matrices, S_{ij}^K and L_i^K using numerical integration, imposition of the boundary conditions, assembly of global matrices, and solution of the equations for the degrees of freedom. In the *postprocessor* part, the output data is processed in a desired format for a printout or plotting, and secondary variables that are derivable from

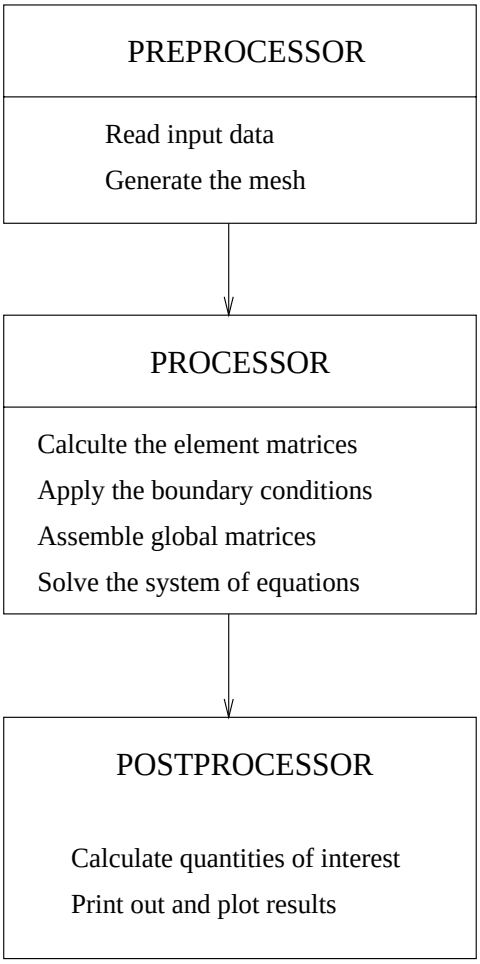

Figure 6: Computer flow diagram of a classical finite element code

the solution are computed and printed. The results may also be output in a graphical form, e.g. using contour plots.

4 Error estimation

4.1 *A priori* error estimation

The error introduced into the finite element solution u_h, because of the approximation of the dependent variable u in an element, is inherent to any problem,

$$u \approx u_h = \sum_{k=1}^{Nh} u_{hk} e_{hk}. \tag{4.57}$$

Here u_h is the finite element solution and u is the exact solution over the domain. We wish to know how the error $e = u - u_h$, measured in a meaningful way, behaves as the number of elements and/or their order of approximation in the mesh is increased. There are several ways in which one can measure the difference between any two functions u and u_h. More generally used measures of the difference of the two functions are *energy norm*, L^2-*norm* and *maximum norm*.

$$\|e\|_E = \left\{ \int_0^l \left[a(e')^2 + ce^2 \right] dx \right\}^{1/2} \tag{4.58}$$

$$\|e\|_{L^2} = \left\{ \int_0^l e^2 dx \right\}^{1/2} \tag{4.59}$$

$$\|e\|_\infty = \max_{0 \le x \le l} | e(x) | \tag{4.60}$$

The norms listed above are *global*, i.e. thay apply to th whole domain $(0, l)$. Analogous definitions hold for any element K:

$$\|e\|_{E,K} = \left\{ \int_K \left[a(e')^2 + ce^2 \right] dx \right\}^{1/2} \tag{4.61}$$

$$\|e\|_{L^2,K} = \left\{ \int_K e^2 dx \right\}^{1/2} \tag{4.62}$$

$$\|e\|_{\infty,K} = \max_{x \in K} | e(x) | \ . \tag{4.63}$$

As we refine the mesh by increasing h or p, we wish to know bounds on the error, measured in the foregoing norms. Ordinarily, for a single element K, these estimates will be

of the form

$$\|e\|_K \leq C \frac{h_K^r}{p_K^s} \tag{4.64}$$

where C is a constant depending on the data of the problem (regularity of the solution), h_K stands for the element size (length) and p_K denotes its order of approximation. The exponents r and s are the measure of *the rate of convergence* of the method with respect to the particular choice of the norm $\|\cdot\|$ and either $h-$ or $p-$ refinements. In our hp-refinment strategy discussed in section 6, we shall use the fact that the h-convergence rate is limited by two factors:

- order of approximation p_K, and
- local regularity of the solution expressed in some appropriate norm expressed in terms of derivatives of $s+1$ order.

More precisely,

$$r = \min\{p, s\} \tag{4.65}$$

A convergence rate r *lower* than order of approximation p indicates a low regularity of the solution.

Estimates of this type require no information from the actual finite element solution. They are known prior to the construction of the solution, and called *a priori* estimates. The *a priori* estimation of errors in numerical methods has long been an enterprise of numerical analysts. Such estimates give information on the convergence and stability of various solvers and can give rough information on the asymptotic behavior of errors in calculations as mesh parameters are appropriately varied.

4.2 *A posteriori* error estimation

In some cases, a more detailed estimate of accuracy can be based on information obtained from the finite element solution itself. It is called an *a posteriori error* estimate and it can be calculated only after the finite element solution has been obtained. Interest in *a posteriori* estimation for finite element methods for elliptic boundary value problems began with the pioneering work of Babuska and Rheinboldt [3]. In this note we will present the *element implicit residual method* introduced in [5, 7, 1] and applied to a variety of problems in mechanics and physics.

4.2.1 The element implicit residual method

We introduce the idea of the error estimation using the standard abstract variational formulation,

$$\begin{cases} u \in u_0 + V \\ b(u,v) = l(v) \quad \forall v \in V \, . \end{cases} \tag{4.66}$$

Introducing a finite element space $V_h \subset V$, we calculate the corresponding finite element solution u by solving the approximate problem :

$$\begin{cases} u_h \in u_0 + V_h \\ b(u_h, v_h) = l(v_h), \quad \forall v_h \in V_h \end{cases} \tag{4.67}$$

The goal is to estimate the residual defined as

$$\|r\|_{V'} := \sup_{v \in V} \frac{\mid b(u_h, v) - l(v) \mid}{\|v\|_E} \tag{4.68}$$

where $\| \cdot \|_E$ denotes the energy norm,

$$\|u\|_E^2 := a(u,u) \, . \tag{4.69}$$

Here $a(u,v)$ is identified as a *symmetric part* of $b(u,v)$ that defines a norm. For the particular example discussed in the previous section, se may select

$$a(u,v) = \int_0^l a(u')^2 \, dx + \beta_l u(l)^2 \, . \tag{4.70}$$

We have to assume then that $a(x) \geq 0$ and $\beta_l \geq 0$. If, additionally, coefficient $c(x) \geq 0$, we can include it in the definition of the energy norm as well,

$$a(u,v) = \int_0^l \left[a(u')^2 c u^2 \right] \, dx + \beta_l u(l)^2 \, . \tag{4.71}$$

REMARK 1 Mathematically speaking, the residual is a linear functional acting on space V and it must be measured using the dual norm. Its choice depends upon the choice of the norm used in the denominator. It can be shown [8] that for a class of self-adjoint problems where $b(u,v) = a(u,v)$, the residual is *equal* to the error measured in the energy norm.

∎

We shall represent the residual in the form

$$\begin{aligned} r(u_h, v) &= b(u_h, v) - l(v) \\ &= \sum_K \{b_K(u_h, v) - l_K(v) - \lambda_K(v)\} \\ &= \sum_K r_K(u_h, v) \end{aligned} \tag{4.72}$$

where, $b_K(u, v)$ and $l_K(v)$ are contributions to the bilinear and linear forms from element K respectively, and $\lambda_K(v)$ is an *element flux functional*. We will postulate the following two main assumptions :

- the element residuals $r_K(u_h, v)$ are in equilibrium with respect to the finite element space,
$$r_K(u_h, v) = b_K(u_h, v) - l_K(v) - \lambda_K(v) = 0 \quad \forall v \in V_h(K)\,, \tag{4.73}$$
- Consistency condition,
$$\sum_K \lambda_K(v) = 0 \quad \forall v \in V\,. \tag{4.74}$$

Here $V_h(K)$ denotes the space of element K shape functions, possibly incorporating Dirichlet boundary conditions if element K is adjacent to the Dirichlet boundary. Next we introduce the local element Neumann problems :

$$\begin{cases} \text{Find } \phi_K \in V(K) \\ a_K(\phi_K, \psi) = r_K(u_h, \psi) \quad \forall \psi \in V(K)\,. \end{cases} \tag{4.75}$$

Here $V(K) = H^1(K)$, except for the element adjacent to Dirichlet boundary at $x = 0$, for which

$$V(K) = \{v \in H^1(K) \,:\, v(0) = 0\}\,. \tag{4.76}$$

We can express now the mesh residual in terms of the element error indicator function ϕ_K ,

$$\begin{aligned} |r(u_h, v)| &= |\sum_K a_K(\phi_K, v)| \\ &\leq (\sum_K \|\phi_K\|_K^2)^{1/2} \|v\|_E \end{aligned} \tag{4.77}$$

This leads to the final estimate,

$$\|r\|_{V'} \leq (\sum_K \|\phi_{h,K}\|_K^2)^{1/2}\,. \tag{4.78}$$

5 User Manual

5.1 Data structure in 1Dhp90

We introduce two *user-defined structures* (*module_pack/data_structure1D*):

- type *node*,
- type *element*.

The attributes of a node include: node type (a character indicating whether the node is a vertex or middle one), integer order of approximation, integer boundary condition flag, a real array *coord*, containing geometrical degrees of freedom (node coordinate), and real array *dofs*, containig the "actual" degrees of freedom. Both the geometry and the actual d.o.f. are allocated dynamically, dependently upon the order of approximation for the node. The entire information about a mesh is now stored in two allocatable arrays, ELEMS and NODES, as declared in the data structure module. The module also includes a declaration for a number of integer attributes of the whole mesh.

The following parameters are relevant at the moment:
NRELIS - number of elements in the initial mesh,
NRELES - total number of elements in the mesh,
NRNODS - total number of nodes in the mesh,
MAXEQNS - maximum number of equations to be solved,
MAXNODS - maximum number of nodes,
MAXELES - maximum number of elements,
NREQNS - actual number of equations to be solved.

Parameters *MAXEQNS* and *NREQNS* are placed into the code in anticipation of using the code for the solution of *systems* of equations. In the code both parameters are set to one.

5.2 Organization of the code

The code is organized in the following subdirectories:

- *blas_pack* - basic linear algebra routines,
- *commons* - system common blocks,
- *data_pack* - exact solution and material data,

- *datstrs_pack* - data structure routines,
- *errest_pack* - error estimation routines,
- *elem_pack* - element routines,
- *files* - system files, sample input files,
- *frontsol_pack* - frontal solver routines,
- *gcommons* - graphics common blocks,
- *graph_1D* - actual graphics routines for the code,
- *graph_util* - graphics utilities,
- *graph_interf* - graphics interface routines,
- *main_program* - driver for the code,
- *meshgen_pack* - initial mesh generation routines,
- *meshmods_pack* - mesh modification routines,
- *module_pack* - data structure moduli,
- *solver1_pack* - interface with frontal solver,
- *utilities_pack* - general utilities.

5.3 Mesh generation and postprocessing routines

Mesh generation (*datastrs_pack/meshgen.f*). A sequence of input data is read from (*files/input*). These include:

- *MAXELES,MAXNODS* - maximum number of elements and nodes in the mesh (the corresponding memory is allocated dynamically),
- *NRELIS* - number of (uniform size) elements in the initial mesh,
- *norder* - (uniform) order of approximation for the initial mesh elements,
- *XL* - length of interval $(0, l)$ to set up the boundary-value problem,
- *ibc1, ibc2* - boundary conditions flags.

A sample input file *input* can be found in directory *files*.

Printing out content of data structure arrays (*datastrs_pack/result.f*)**.** The routine prints out the current content of the data structure arrays inlcuding the complete information on elements and nodes. It can be conveniently used for debugging the code.

Graphical output (*graph_1D/graph1D.f*)**.** Th routine displays a graphical representation of the current mesh and plots the corresponding exact and numerical solutions. The scale on the right prescribes the color code for different orders of approximation $p = 1, \ldots, 8$.

5.4 Processing algorithms

We discuss quickly a number of algorithms pivotal in the implementation of any Finite Element Method: the calculation of element matrices, modification due to the boundary conditions, and the assembling procedure. Please consult the corresponding routines for details.

5.4.1 Evaluation of element matrices (*elem_pack/elem.f*)

In : element number *Nel*
Out : element stiffness matrix S_{k_1,k_2} and load vector L_{k_1}

determine the element order of approximation and select the corresponding
Gauss quadrature,
determine the element vertex nodes, x_L, x_R (geometry d.o.f.), $h = x_R - x_L$,
initiate element stiffness matrix S_{k_1,k_2} and load vector L_{k_1},
for each integration point ξ_l,
    evaluate the physical coordinate, $x_l = x_L + \xi_l h$,
    evaluate master element shape functions $\hat{\chi}_k$ and their derivatives with respect to
    the master element coordinate $\frac{d\hat{\chi}_k}{d\xi}$
    evaluate the derivatives of the shape functions with respect to the physical coordinate:
    $\frac{d\chi_k}{dx} = \frac{d\hat{\chi}_k}{d\xi}\frac{1}{h}$
    determine the weight, $w = w_l * h$,
    get load $f = f(x_l)$
    get material constants,$a = a(x_l), b = b(x_l), c = c(x_l)$
    for each d.o.f. k_1,
        accumulate for the load vector entries:
        $L_{k_1} = L_{k_1} + f\chi_{k_1} w$
        for each d.o.f. k_2,

accumulate for the stiffness matrix entries:

$$S_{k_1,k_2} = S_{k_1,k_2} + \ (a * \frac{d\chi_{k_2}}{dx}\frac{d\chi_{k_1}}{dx} + b * \frac{d\chi_{k_2}}{dx}\chi_{k_1} + c * \chi_{k_2}\chi_{k_1}) * w,$$

end of the second loop through the d.o.f.,

end of the first loop through the d.o.f.,

end of loop through integration points.

5.4.2 Modification of the element matrices due to boundary conditions

In : element number Nel, element stiffness matrix S_{k_1,k_2}, and load vector L_{k_1}
Out : S_{k_1,k_2}, L_{k_1} after BC modification

get BC flags for the element vertex nodes
for each vertex node, $i = 1, 2$,

CASE : Dirichlet boundary
get BC data γ,
for each d.o.f. k,

if (k=i) then

$L_k = \gamma$
zero out the $i-$the row of stiffness matrix,
$S_{kk} = 1$,
$L_k = L_k - S_{ik} * \gamma$,
$S_{ik} = 0$.

endif

end of loop through d.o.f.,
CASE : Neumann boundary
get BC data β, γ,
accumulate for the stiffness matrix and load vector:
$S_{ii} = S_{ii} + \beta$,
$L_i = L_i + \gamma$.
end of loop through vertex nodes

5.4.3 Assembling 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 consequtive 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_pack/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.*, see the discussion in the previous section. On the practical level, we may introduce an extra attribute for each node nod in the mesh, say $nbij(nod)$ 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 $nbij$ with zeros
initiate d.o.f. counter $idof = 0$
for each element nel
    for each element node i
        get the global node number: $nod = ELEMS(nel)\%nodes(i)$
        skip if $nbij(nod) \neq 0$, i.e. the node has already been visited
        set the counter for the first d.o.f. of the node: $nbij(nod) = idof + 1$
        determine the number of d.o.f. $ndof$ corresponding to the node
        update the counter: $idof = idof + ndof$
    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 $Aloc, Bloc$
    for each nodal d.o.f. i
        establish element d.o.f. connectivities:$loc\_con(nel, i) = nbij(nod) + i - 1$
    end of loop through nodal d.o.f.
    for each element d.o.f. i
        determine the connectivity: $k = loc\_con(nel, i)$
        accumulate for the global load vector:
        $Bglob(k) = Bglob(k) + Bloc(i)$
        for each element d.o.f. j

determine the connectivity: $l = loc\_con(nel, i)$
accumulate for the global stiffness matrix:
$Aglob(k, l) = Aglob(k, l) + Aloc(i, j)$
end of the second loop through element d.o.f.
end of the first loop through element d.o.f.
end of loop through elements

5.4.4 Solver

Routines assembling the global matrices, and solving the corresponding system of equations are to be provided by the user. For the sake of presenting an operational code, we include in the code an interface with a (more complicated) frontal solver discussed shortly in the Appendix.

5.4.5 Setting up data. Files

As the main goal of this 1D code is rather academic and focuses more on studying the finite element method than solving practical problems, we shall accept a rather unusual way of inputing data. Namely, we shall assume that we *do know* the exact solution together with its first and second derivatives. The purpose of our finite element computations will be just to compute the finite element approximation and study the error. Consequently, we shall assume that the data to the problem : load $f(x)$ and boundary condition data γ will always be calculated using the exact solution (by calling routine *data_pack/exact.f*). That way we can minimize the number of changes in the code when we want to study a different solution. The material data (operator coefficients $a(x), b(x), c(x)$ and Cauchy boundary data β) have to be set independently in routines *data_pack/getmat.f* and *getc.f*.

Files. All files are placed in directory *files*. Besides the *input* file containing the data for the initial mesh generation, discussed earlier, the code opens an *output* file (used e.g. by routine *datstrs_pack/result*), and an additional file *result* to be discussed in the next section. Both files *output* and *result* are automatically opened by the code and need no user's action.

Running the code

Step 1: Define a boundary-value problem, and select an exact solution that you want to reproduce with the FE code. Modify routines *data_pack/getmat.f, data_pack/getc.f*, and *data_pack/exact.f* accordingly.

Step 2: Prepare the input file.

Step 3: Use the provided *makefile* to compile and link the code.

Step 4: Type *a.out* to execute the code. If the input file is correct, and the initial mesh has been generated successfully, the code will display the main menu that includes the possibility of solving the problem, printing out the content of the data structure arrays, and displaying the mesh with the corresponding exact and approximate solutions. The forth option, automatic *hp*-refinements, will be discussed in the next section. Please disregard the call to a testing routine which has been used for debugging.

6 Adaptivity

6.1 *p*-refinement/unrefinement(*meshmods_pack/modord.f*)

Modifying order of approximation for an element is easy as it affects only its middle node. The memory allocated for the middle node d.o.f. has to be either expanded or shortened depending upon the new order of approximation. If the order is increased then the new d.o.f. are initiated with zeros.

6.2 *h*-refinement (*meshmods_pack/break.f*)

Breaking an element involves creating two new entries in data structure array ELEMS for the element sons, and creating one new vertex node and two new middle nodes in array NODES. The sons have the same order of approximation as their father, their d.o.f. are initiated in routine *meshmods_pack/inidofh.f* in such a way that the new representation of the solution will *exactly* match the one corresponding to the single father element. We do not delete the middle node of the father.

During the *h*-refinement, the information about the *family tree* is stored. This includes storing the information on elements' fathers and sons. The concept is illustrated in Fig. 7.

6.3 Natural order of elements (*datastrs_pack/nelcon.f*)

The numbering of elements in the initial mesh (from the left to the right) and the family tree structure induce the corresponding *natural order of elements*. The idea is to follow the *leaves* of the tree and the ordering of elements in the initial mesh, compare Fig. 7.

6.4 *h*-unrefinement(*meshmods_pack/cluster.f*)

A refined element can be back unrefined. The corresponding entries for its element sons and their nodes are deleted from the data structure arrays. We admit a situation in which the unrefined element will have a *greater* order of approximation than its sons. The new d.o.f. for the unrefined element are evaluated in routine *elem_pack/project.f* by performing *hp*-interpolation of the old representation of the solution using the clustered element shape functions. Having done the interpolation (projection), we evaluate the corresponding interpolation error using the energy norm.

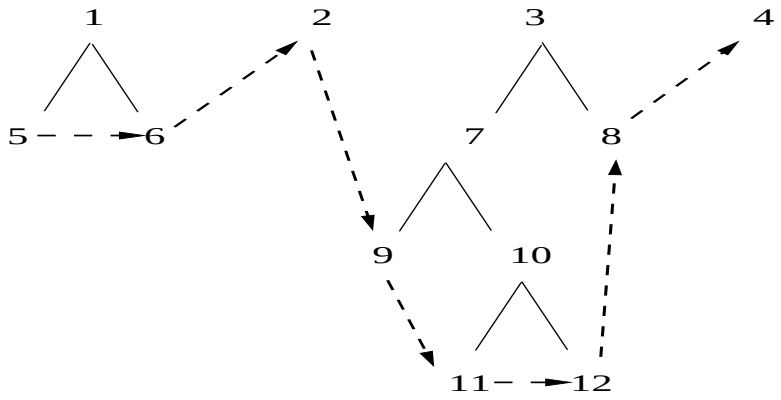

Figure 7: The family tree for a sequence of h-refinements of elements 1,3,7,10. The dotted line indicates the *natural order of elements*

6.5 The h-refinements strategy

Having calculated the element residuals in routine *errest_pack/errest.f* , we break (h-refine) elements with the biggest residuals. More precisely, given a percentage *perc* of the total residual (squared) (set to *perc*=60 in the code), we reorder elements according to their residuals, and refine the first M elements from the list that contribute with *perc* percentage to the global residual. The operation is performed in routine *meshmods_pack/refine.f*.

6.6 Trading h-refinements for p-refinements

When refining the mesh, we do not attempt to choose between h- and p-refinements . Instead, after the problem has been solved on the h-refined mesh, we try to *trade* the h-refined elements for p-refinements. Towards this goal we first solve the problem on the h-refined mesh. Next we loop through all just h-refined elements and compute for each of them the corresponding *local* (numerical) h-convergence rate of the residual. If the rate is optimal, i.e. it is equal to, or it exceeds the corresponding order p of approximation for the element, we *unrefine* (cluster) the element, and *trade* the h-refinement for a p-refinement, i.e. we increase p to $p+1$. A rate of convergence below the order of approximation p, indicates a lower regularity, compare estimate (4.64), and in such a case, we leave the h-refined element unchanged.

The formal algorithm looks as follows. Note the tollerance factor .9.

for each just h-refined element K
  if element order $p_K \leq 7$ then
    estimate the new element residual by summing up the residuals for its sons

```
        compute the numerical convergence rate r_K
        if r_K ≥ 0.9 * p_K then
           cluster back the element
           increase element order from p_K to p_K + 1
        endif
     endif
  end of loop through refined elements
```

Consult the *meshmods_pack/trade.f* routine for details.

6.7 Interactive refinements

In many problems, we may use our experience on the problem at hand to begin with a better than uniform initial mesh produced by the mesh generator. The *graph_1D/graph1D.f* routine that displays a graphical representation of the mesh and the current solution, allows also for an interactive mesh modification using the mouse.

6.8 The final *hp*-adaptive algorithm

Fig. 8 presents the final, automatic *hp*-refinements algorithm. The algorithm can be invoked from the main program menu by selecting the automatic *hp*-refinements option. Parameter *tol* - acceptable error tollerance in per-cent of the energy norm of the solution has to be input from the keyboard. After each refinement, the corresponding number of d.o.f. in the mesh and the computed FE error are written to file *files/result*, automatically open by the program. The data can later be used to visualize the corresponding convergence rates by selecting the option *rates* from the graphics program.

6.9 Examples of *hp*-adaptive solutions

Problem

$$\left\{ \begin{array}{rl} -u'' & = f(x) \quad x \in (0,1) \\ u(0) & = 0 \\ u'(1) & = g(x) \end{array} \right. \tag{6.79}$$

All convergence rates are represented using the log-log scale, in terms of the total number of degrees-of-freedom.

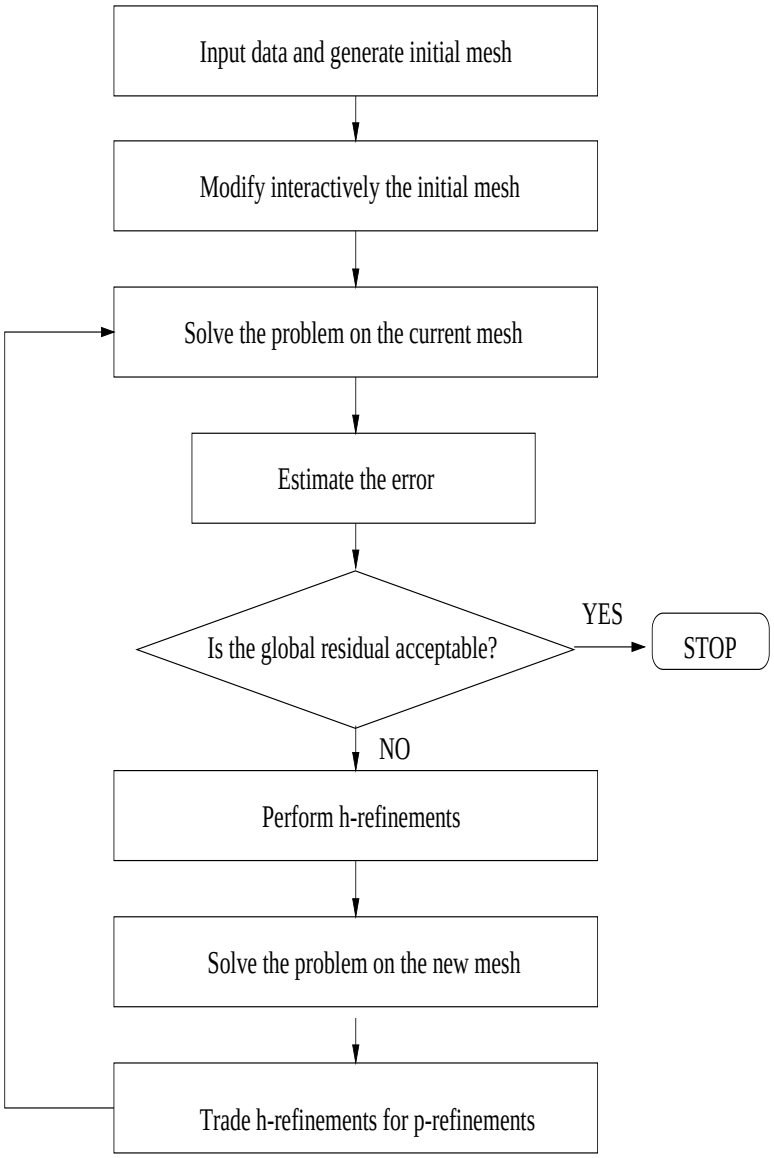

Figure 8: Flow chart for the *hp*-adaptive FE code

Example 1: A smooth solution. $u_{exact}(x) = sin(x)$. Error tolerance $tol = 0.01$ per-cent of the energy norm of the solution.

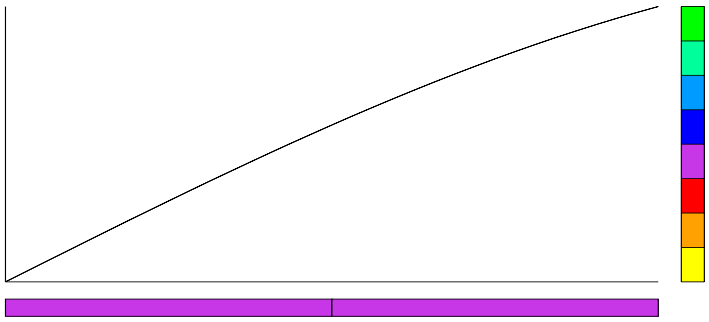

Figure 9: Example 1: Final *hp* mesh with the corresponding (indistinguishable) numerical and exact solutions

Figure 10: Example 1: Rates of convergence

This is a very smooth solution. The algorithm chooses from start to use p-refinements only. In this case, the *hp*-method reduces just to the p-method. Decreasing the error tollerance reveals also that, as the number of degrees-of-freedom grows, the order of approximation p is increased *uniformly*. This is consistent with the *hp* approximation theory, see e.g. [2].

Example 2: A singular solution. $u_{exact}(x) = x^{0.6}$. Error tolerance $tol = 1$ per-cent of the energy norm of the solution.

Please use routine *result* to verify that the mesh is geometrically graded towards the singularity at $x = 0$, with the order of approximation p increasing linearly from $p = 1$ at the

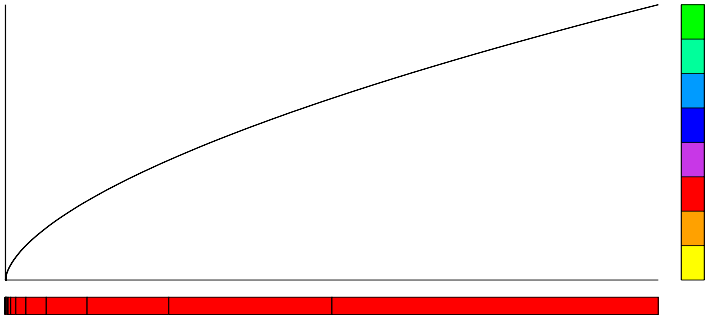

Figure 11: Example 2: Final *hp* mesh with the corresponding (indistinguishable) numerical and exact solutions

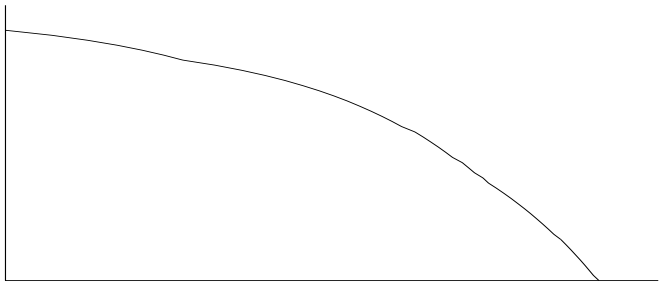

Figure 12: Example 2: Rates of convergence

singularity to $p = 3$ away from it. This is consistent with the well known result of Babuska and Gui, see e.g. [2].

Example 3: A solution with an internal layer. $u_{exact}(x) = \text{atan}60. * (x - .5)$. Error tolerance $tol = 1$ per-cent of the energy norm of the solution.

When the error tollerance is decreased, the algorithm continues to choose exclusively the p-refinements only. Moreover, the order of approximation p stays essentially uniform. Thus, we might say that the initial h-refinements help to resolve the scale and to construct an optimal initial mesh only. Once the mesh is determined, the uniform p-refinements again turn out to be *asymptotically* optimal. Of course, the point is that, in practical computations, we always work in the preasymptotic range only.

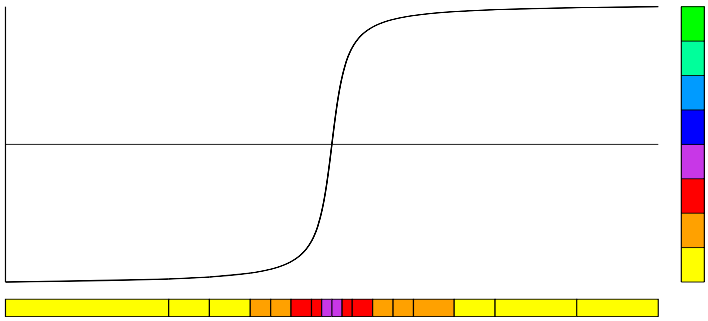

Figure 13: Example 3: Final *hp* mesh with the corresponding (indistinguishable) numerical and exact solutions

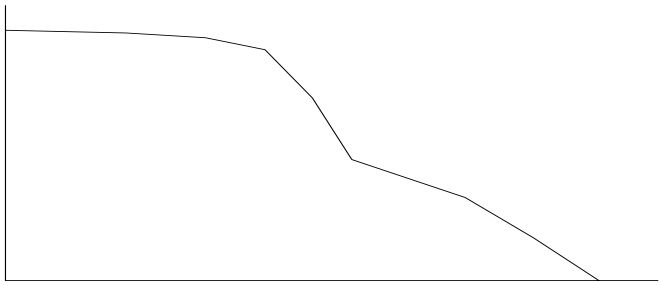

Figure 14: Example 3: Rates of convergence

Acknowledgements: The authors are much indebted to Professors Ivo Babuška and J. Tinsley Oden for numerous discussions regarding the subject of *hp* mesh optimization.

References

[1] M. Ainsworth,J.T. Oden, "A Posteriori Error Estimation in Finite Element Analysis", *Comput. Meth. Appl. Mech. Engrg.*, **142**, 1-88, 1997.

[2] I. Babuška and B. Q. Guo, "Approximation Properties of the *hp* Version of the Finite Element Method", *Computer Methods in Applied Mechanics and Engineering, Special Issue on p and hp- Methods*, eds. I Babuška and J. T. Oden, **133**, 319-346, 1996.

[3] I.Babuska and W.C.Rheinboldt, "A Posteriori Error Estimates for the Finite Element Method", *Int. J. Numer. Meth. Engng.* **12**, 1597-1615, 1978.

[4] E. Becker, J.T. Oden and G. Carey, *An Introduction to Finite Elements*, Prentice Hall, 1985.

[5] L. Demkowicz, L. J.T. Oden, and T.Strouboulis, "Adaptive Finite Elements for Flow Problems with Moving Boundaries.Part 1: Variational Principles and A Posteriori Error Estimates", *Comput. Meth. Appl. Mech. Engrg.*, **46**, 217-251, 1984.

[6] L.Demkowicz,J.T.Oden, W. Rachowicz and O.Hardy, "Toward a Universal *hp*-Adaptive Finite Element Strategy, Part 1. Constrained Approximation and Data Structure", *Comput. Meth. Appl. Mech. Engrg.* **77** , 79-112, 1989.

[7] J.T.Oden, L.Demkowicz, T.Strouboulis, and P.Devloo, Adaptive Method for Problems in Solid and Fluid Mechanics, Accuracy Estimates and Adaptive Refinements in Finite Element Computations, John Wiley-Sons Ltd. , 249-280, 1986.

[8] J.T.Oden, L.Demkowicz, W. Rachowicz and T.A.Westermann, "Toward a Universal *hp*-Adaptive Finite Element Strategy, Part 2. A Posteriori Error Estimation", *Comput. Meth. Appl. Mech. Engrg.* **77** , 113-180, 1989.

[9] W. Rachowicz, J.T.Oden and L.Demkowicz, "Toward a Universal *hp*-Adaptive Finite Element Strategy, Part 3. Design of *hp* Meshes", *Comput. Meth. Appl. Mech. Engrg.* **77** , 181-212, 1989.

A Interface with a frontal solver

The frontal solver is 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 $solver1\_pack$ 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 $solver1\_pack$.

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',

$$nick_j = j * 1000 + ndof \tag{A.80}$$

where $ndof$ is the number of degrees of freedom associated with the node, i.e. is equal 1 for a vertex node, and $p-1$ for a middle node of order p. 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 auxilliary 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).