Data structures and implementation of an adaptive *hp* finite element method

SUBMITTED TO THE UNIVERSITY OF LEICESTER FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

August 1998

By

Bill Senior

Department of Mathematics and Computer Science

Abstract

For a fully adaptive *hp* finite element programme to be implemented it is necessary to implement n–irregular meshes efficiently. This requires a sufficiently flexible data structure to be implemented. Because such flexibility is required, the traditional array based approach cannot be used because of its limited applicability. In this thesis this traditional approach has been replaced by an object orientated design and implementation. This leads to an implementation that can be extended easily and safely to include other problems for which it was not originally designed.

The problems with maintaining continuity on such a diverse variety of meshes and how continuity is maintained are discussed. Then the main structure of the mesh is described in the form of domain, subdomains and elements. These are used in conjunction with constraint mappings to give a conforming approximation even with the most irregular of meshes.

There are several varieties of matrix generated by the method each with its own problems of storage. Sparse matrices, with perhaps more than 95% of zero entries, need to be used along side dense matrices. In this thesis an object orientated matrix library is implemented that enables this variety of matrices to be used.

An *hp* finite element algorithm is then implemented using the data structures, and is tested on a range of test problems. The method is shown to be effective on these problems.

Acknowledgements

I would like to thank my supervisors Dr. Mark Ainsworth and Mr. Derek Andrews for their advice and guidance over the past few years. I would also like to thank the Department of Mathematics and Computer Science at the University of Leicester for their support throughout both my undergraduate and postgraduate studies. Thanks are also given to the Engineering and Physical Sciences Research Council for supporting me through my research studentship. I would also like to thank Pat Coggins for his help during the writing of this thesis.

Contents

List of Figures

Chapter 1

Introduction

Most partial differential equations are of a form that cannot be solved analytically. However, an approximate numerical solution can be found using several methods, the most common of which are:

- finite difference; and
- finite volume; and
- spectral methods; and
- finite element (whose implementation is the topic of this thesis).

The finite difference method [29, 44] is a method that approximates the derivatives in an attempt to approximate the solution. It is difficult to approximate general shaped domains using this method. The method can also be unstable, due to inaccuracies in the approximation of the derivative. In the finite element method [13, 14, 16, 29, 60], we divide up the domain into a number of smaller regions called elements. On these elements we have functions, usually polynomials, that approximate the solution. The finite volume method [48] approximates the solution by breaking the domain up into volumes, each of which surrounds a point. It is very similar to a piecewise constant finite element method. It is used frequently in computational fluid dynamics. Finite difference, finite volume and finite element methods are based on local representations of functions; this is usually done with polynomials of a low degree. In contrast, spectral methods [45] make use of global representations, usually high–order polynomials or Fourier series over a single element.

1.1 The finite element method

Throughout this thesis we will discuss only two dimensional elements, but all of the ideas can be modified for use in three dimensional problems quite easily. For the elements in a regular mesh we have that:

1. each element is either a triangle or a quadrilateral; and
2. for any pair of elements k and j then the intersection $k \cap j$ is either
    - the empty set; or
    - a single common vertex; or
    - a single common edge; or (in three dimensions)
    - a single common face.

Sometimes the final approximation is not as accurate as we would like and so to improve the accuracy of the approximation we add in more degrees of freedom; these new degrees of freedom can be added in a variety of ways, [60]:

1. elements are locally refined (this is called the $h-$version); or
2. the elements may have the degree of their approximation polynomial enriched (this is called the $p-$version); or
3. a combination of the above two methods (called $hp-$version).

After this refinement process has taken place we must assemble a new approximation, using the new partition. Each of these versions has its own advantages and disadvantages as will be seen in what follows.

1.2 Types of refinement

1.2.1 $h-$type refinement

This is the standard refinement used in the finite element method. Figure 1.1 attempts to show the idea of h-refinement [25] in 1 dimension. When a greater accuracy to the solution is required, the elements are divided to create more elements, while at the same time keeping the polynomial degree on the elements the same. The polynomial degree is usually low, typically linear, quadratic or cubic on each of the elements. For a uniform $h-$refinement we have that $h_k \approx h_j$ for all elements k, j in the domain, where h is the size of the element. In uniform refinement all of the elements are refined together at the same time in the same way. This will usually lead to inefficiencies in computation because there will be many elements in areas of the domain where refinement is not necessary. This leads to the idea of nonuniform $h-$refinement. In this we only refine the domain in the areas where the approximation is relatively inaccurate. This can lead to problems with irregular meshes, which in turn can cause problems in implementation.

1.2.2 $p-$type refinement

In this version the refinement [19, 60] increases the degree of the approximating basis functions but leaves the partitioning alone. Figure 1.2 attempts to show the idea in 1 dimension. Again, as in the case of the $h-$version, one can refine uniformly in all of the elements, that is, increase the degree of the approximating functions in all of the elements. This is known as the uniform $p-$version. As in the case of the uniform $h-$version, this could be computationally inefficient as the polynomial degree will also be unnecessarily increased in the areas of low error. It is possible to increase the degree of the approximating functions in only those elements where the error is largest; this is more computationally efficient but can lead to problems in implementation due to what is known as *hanging degrees of freedom*; these are discussed later.

1.2.3 $hp-$type refinement

In this version we combine both the $h-$ and $p-$versions [46] so that the best attributes of each version can be utilised. This again can lead to problems with irregular meshes. However, this time the situation is more complex since we might not only have to think about elements of different sizes being adjacent to each other but also the possibility of different degrees for their basis functions. Maintaining the continuity of the approximation across interelement boundaries can therefore be a problem with the $hp-$version. The problem now is that we will require very elaborate data structures to encode all of the information required by the method.

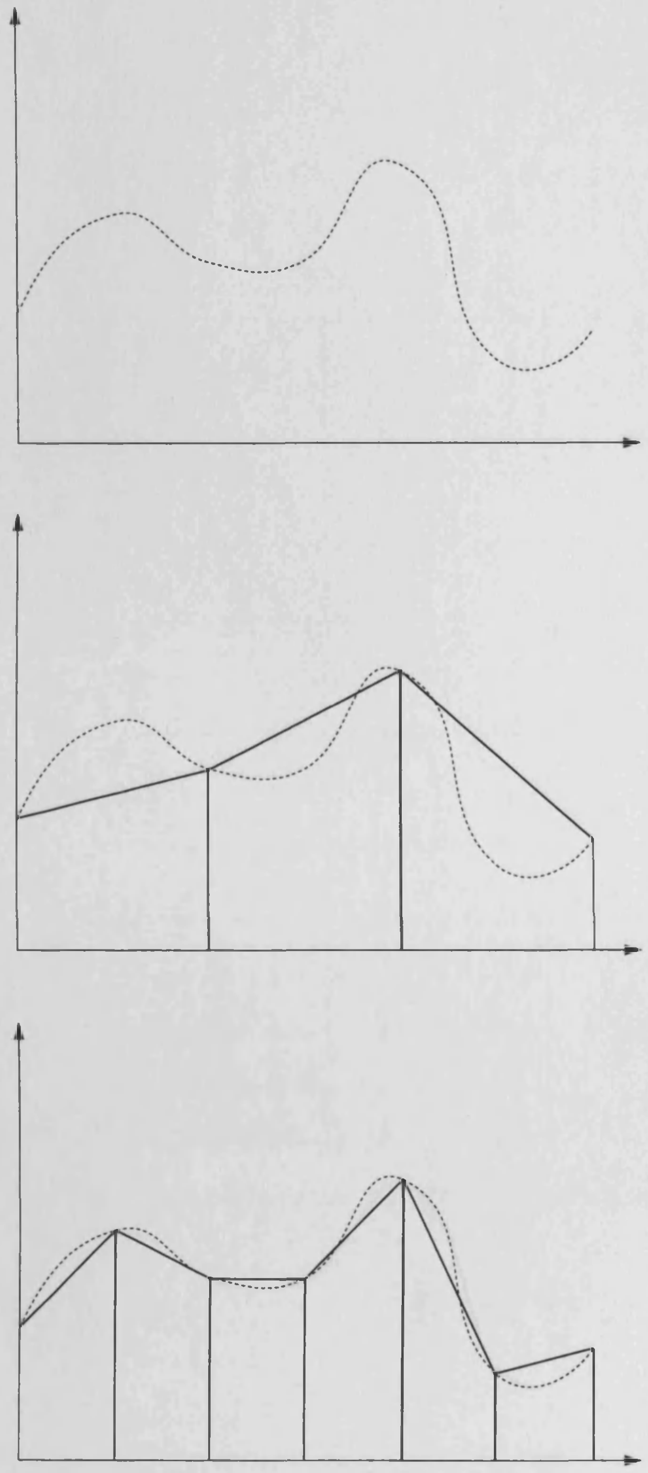


Figure 1.1: One dimensional h–refinement

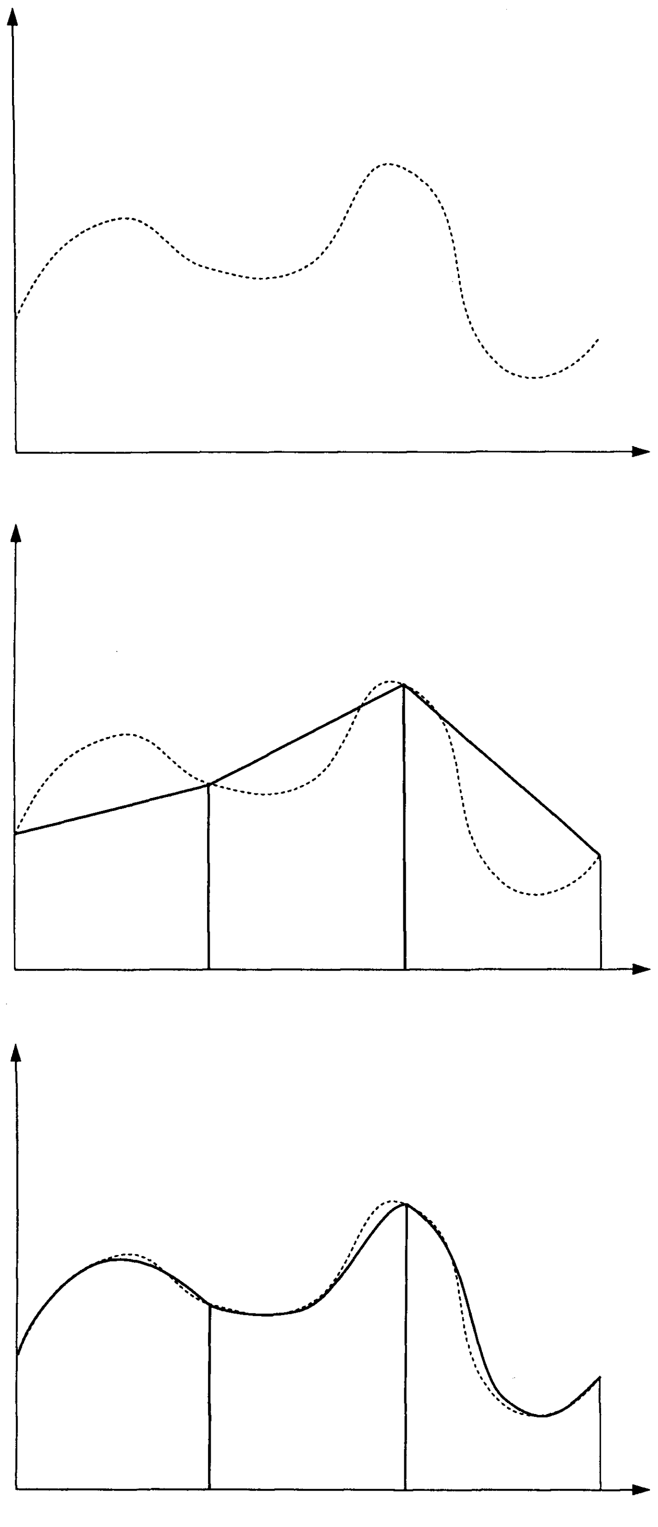


Figure 1.2: One dimensional p–refinement

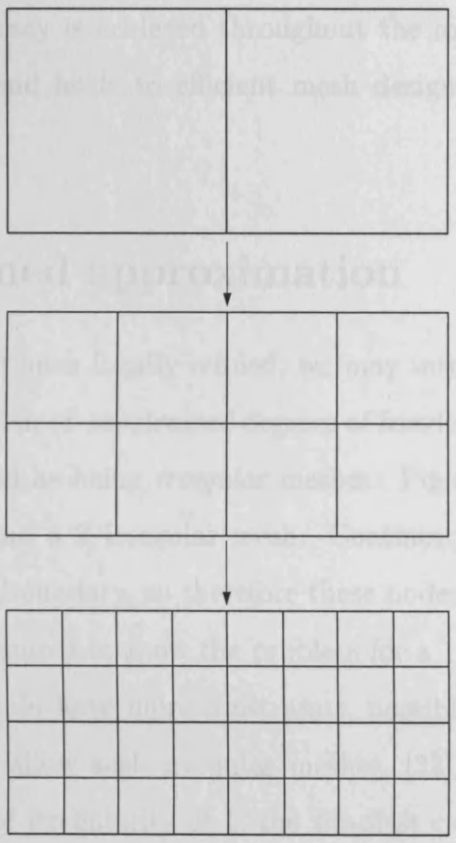


Figure 1.3: A sequence of uniformly h-refined meshes

1.2.4 Adaptivity

In general it is not known beforehand which mesh is the best one for the particular problem being solved. In an adaptive strategy, error estimators are used to identify areas in the mesh where the error is greatest, and at these places some refinement is performed to reduce the error. The process that results in either an h–refinement or p–refinement is discussed in [10, 12]. This can be repeated until the required accuracy is achieved throughout the mesh. This is known as an adaptive strategy, and leads to efficient mesh design, as long as the error estimators are effective.

1.3 Constrained approximation

After the elements have been locally refined, we may introduce what are called *hanging* degrees of freedom or *constrained* degrees of freedom. These meshes with such nodes are described as being *irregular* meshes. Figure 1.4 shows two such meshes, a 1–irregular and a 2–irregular mesh. Continuity must be maintained across the interelement boundary, so therefore these nodes must be fixed or constrained. Figure 1.5 attempts to show the problem for a 1–irregular mesh. For a 2–irregular mesh we would have more constraints, possibly up to four elements. Many implementations allow such irregular meshes, [32], but they are usually restricted to an index of irregularity of 1, the simplest case. If they do allow a higher index it is usually implemented on a case by case basis, each needing to be separately implemented, which increases the chances of programming errors occurring. If we change the basis function type, then these procedures must all be rewritten. Therefore in subsequent chapters we will develop data structures that will allow us to maintain continuity across interfaces that are n–irregular, and which for a general n with a simple change of a matrix (or set of polynomials) allow the basis functions to be changed. This is one of the novelties in this thesis.

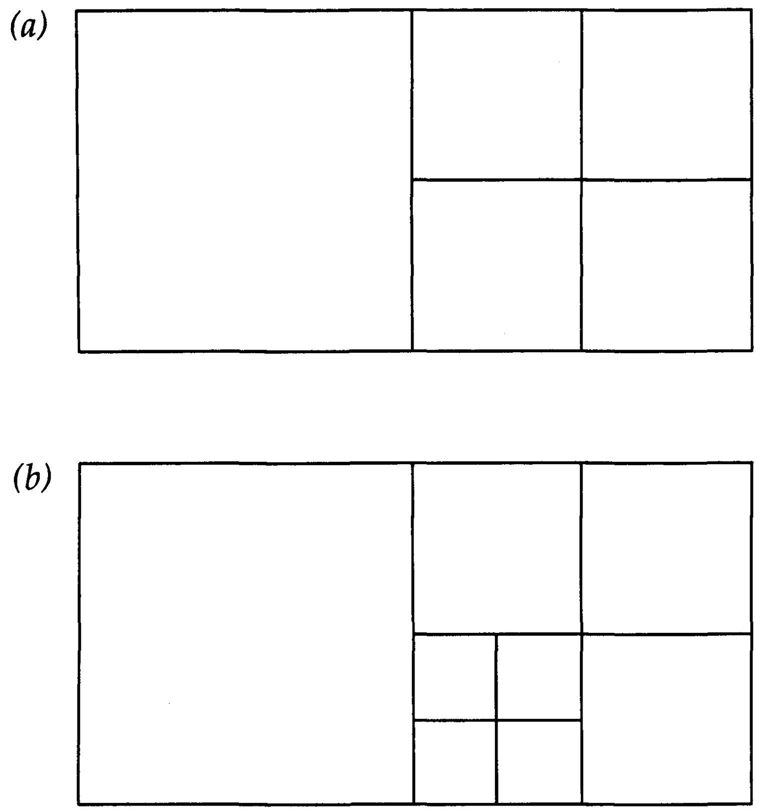


Figure 1.4: (a) shows a 1–irregular mesh, (b) a 2–irregular mesh

Allowing n–irregular meshes gives us more flexibility in our choice of mesh and allows fewer restrictions on any adaptive refinement algorithm. We will also modify the data structures to allow us to constrain the hanging nodes in meshes such as those shown in Figure 1.6 and Figure 1.7. In these meshes the elements are no longer refined equally but by a different ratio, possibly varying on different edges depending on the strength of singularity that may be present.

1.4 Systems of equations

The majority of problems encountered are *not* scalar problems, but problems with two or more components, such as linear elasticity, whose equations are shown

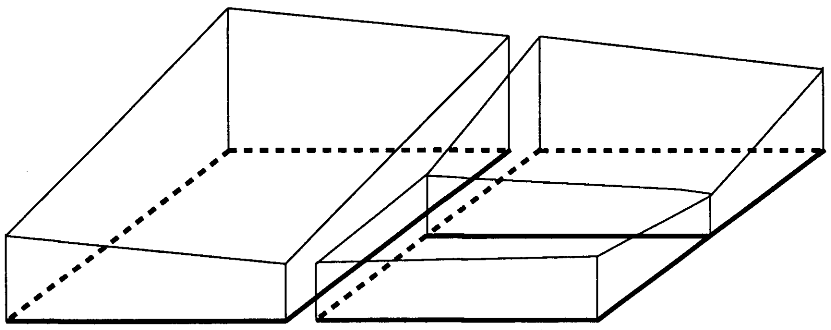


Figure 1.5: Example of unconstrained approximation

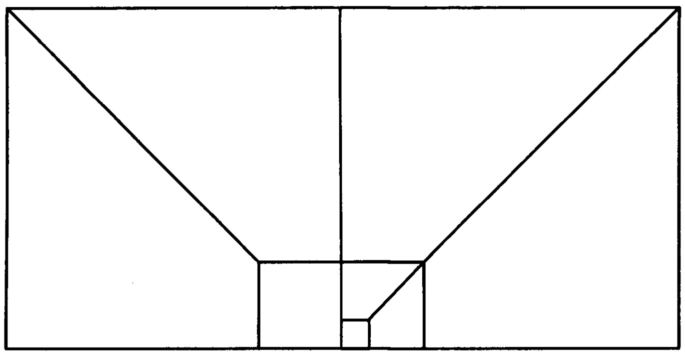


Figure 1.6: Domain with non-standard unconstrained degrees of freedom

below:

$$\frac{E(1-\nu^2)}{1-2\nu}\begin{bmatrix} \dfrac{\partial^2 u}{\partial x^2}+\dfrac{1-2\nu}{2(1-\nu)}\dfrac{\partial^2 u}{\partial y^2}+\dfrac{1}{2(1-\nu)}\dfrac{\partial^2 v}{\partial x \partial y} \\ \dfrac{1}{2(1-\nu)}\dfrac{\partial^2 u}{\partial x \partial y}+\dfrac{1-2\nu}{2(1-\nu)}\dfrac{\partial^2 v}{\partial x^2}+\dfrac{\partial^2 v}{\partial y^2} \end{bmatrix} = \begin{bmatrix} -F_x \\ -F_y \end{bmatrix} \tag{1.1}$$

where u and v represent the $x-$ and $y-$displacements respectively, and E is Young's modulus and ν is Poisson's ratio.

This presents another set of problems as we must ensure that continuity across the interelement boundary is still maintained for each of the components in the system. The matrices generated for such problems are usually much larger because, for each component in the system, we have as many degrees of freedom as in the scalar case. Therefore, because we have two components, we will have twice as many degrees of freedom, implying that the matrix will also be twice as large.

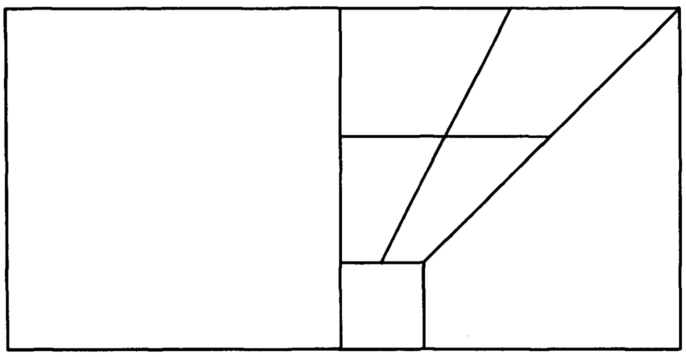


Figure 1.7: Domain that has both standard and nonstandard refinements

A data structure is needed that can handle not only any of the possible refinements, with or without any h or p constraints, but also a wide range of situations such as scalar problems, systems of equations, nonlinear problems, and also be able to assemble and solve these problems in parallel, if required.

1.5 Existing work

Many commercial partial differential equation solving tools already exist, such as Matlab PDEToolbox [1], MSC/NASTRAN [3], NAG library [2], ProPHLEX [4, 47]. In these finite element tools certain meshes may lead to difficulties in maintaining the continuity of the approximation across interelement boundaries. Normally, there would be two options to the implementer: either not to allow irregular meshes, or programme each case individually, and there is potentially an infinite number of cases each of which could cause difficulties. This leads to restrictions in their applicability. If an adaptive refinement strategy is adopted then meshes that are highly irregular might be introduced. This means that current finite element codes would not be able to simply adopt the refinements suggested by the adaptive algorithm, and thus a sub-optimal mesh will be used. Therefore it is of practical importance to address this problem.

In [32] this problem is addressed in a restricted way by implementing the most straightforward case of a 1–irregular mesh. However, the problems of n–irregular meshes are not discussed in [32], and the problem remains of how to implement general irregular meshes. Chapters 2 and 3 of this thesis describe how this problem was overcome, where *constrained nodes* were successfully implemented; thus meshes of a more irregular nature can be used. The effectiveness of this new flexibility will be tested by solving various problems, both scalar and systems of partial differential equations. The implementation of a parallel *hp*–version finite element code has been looked at in [17], here they concluded that the computational costs of the *hp*–version are dominated by the local computations, i.e., the construction of the local stiffness matrices and subsequent elimination of the interior degrees of freedom (the results shown in [17] are for a shared memory parallel architecture).

1.6 Object orientated approach

An important feature of the code is the object orientated approach, [26, 38, 67, 68]. This enables the code to be far more flexible, in that once the objects are implemented then most of the code remains the same for a wide variety of problems.

1.6.1 Elements

These are fundamental to finite element implementations, as they provide the interface to the problem functions in the form of the element contributions, i.e. the element stiffness matrix and load vector.

1.6.2 Subdomains

These are patches, or groups of elements. They are sometimes referred to as *superelements.* They are usually made up from elements, or subdomains that have been *h*–refined. These are at the intermediate level in the hierarchy, between the domain at the top and the elements at the bottom. They provide the connectivity and enable a simple h-refinement of the elements. This simplicity however, can lead to a wide variety of refinements, catering for most, if not all, of the refinements that will be required.

1.6.3 The domain

This is the main data structure that the user interfaces with the main part of the code. Elements are added to the domain then a sequence of *assemble; solve; refine* steps are performed.

1.6.4 Interfaces

These provide the connectivity information for the domains and the subdomains. They also construct the connectivity mapping for the interfaces at a particular level.

1.6.5 Connectivity mapping

The classical finite element assembly procedure may be described using *Boolean transformations* representing the mappings between the local and the global degrees of freedom. While such transformations are implemented in every finite element code, their presence has become obscured by the compact data representation, and its associated efficient implementation. Generalisations of these basic Boolean transformations will form one of the main ideas behind our treatment of hp–finite element approximations in this thesis.

1.7 Modular solution design

A simple element class enables a wide range of problems to be tackled with minimal change. Usually, the only changes that need to be made are to the subprogrammes *Assemble_Stiffness_Matrix* and *Assemble_Load_Vector* plus a copying operation. We added an element for nonlinear problems [42, 43] and a simple implementation of elements for dimensional reduction to the subsystem without changing any of the subdomain or domain programme code or most of the element code.

1.8 Matrices and solvers

Matrices and solvers are an important part of any finite method implementation as it is the resulting matrix equation that we are required to solve. This can be a very big problem, not only from the size of the matrix, perhaps many hundreds of thousands square, but also because it is almost always very ill-conditioned as a result of using very small elements or a high degree for the approximating polynomial basis function. There are a number of solvers used; these usually fall into two types:

- direct solvers, [29, 37, 44]; and
- iterative solvers, [16, 24, 40, 56].

1.8.1 Direct solvers

These can be used on any general matrix, provided, of course, the matrix is non-singular. The main methods are

- factorise;
- inverse (elementary row operations);
- Gaussian elimination.

Once the matrix has been factorised (or inverted) then this can be used on any number of *right hand sides* without having to refactorise. But direct solvers do suffer from *fill-in*, [16]; this is where most, if not all, of the zero entries in the original matrix are now not zero. This can be a problem for large matrices, for example a matrix that is about 100,000 square will require about 80 GB just to store. So in many cases it is not feasible to use a direct method to solve.

There are some direct solvers for such sparse systems though [37], with these it is beneficial to have a narrow band width as possible so as to minimise the fill–in. Numerical stability for a symmetric positive definite matrix is not usually a problem, however for the unsymmetric case Gaussian elimination for such a system can be unstable, especially for large systems.

1.8.2 Iterative solvers

Iterative solvers usually work better if the matrix is sparse, because the work done to perform a matrix vector product is much less than that for a dense matrix. Therefore if the number of iterations taken to solve is small then the amount of work used to solve will be much less than with a direct solver. Also as the matrix vector product is usually the most computationally expensive step in this type of solver, if this step is parallelised then the solver as a whole gains considerably. We do not usually want the solution of the system exactly (to machine accuracy), but to some predetermined tolerance.

If ξ_i is are the solution after the current iteration then we would want to stop the solving process when we have, for example,

$$\frac{||A\xi_i - b||}{||b||} < 10^{-3}$$

This saves on the amount of work required, because the exact solution is not required. Iterative methods do have problems sometimes though, these usually related to the *condition number*, κ, of the matrix.

$$\kappa = ||K|| \cdot ||K^{-1}|| \geq 1$$

for some matrix norm, usually the 2 norm. If our matrix is symmetric then this can be shown to be the ratio of the minimum and maximum eigenvalues of the matrix. The closer to 1 that this number is then the better conditioned the matrix is. Many iterative solvers exist now for solving a wide variety of matrices, such as Gauss Seidel, conjugate gradient and generalised minimum residual. Conjugate gradient is commonly used as the matrices that are generated by the finite element method are of a form that this method can be used nicely. We would use generalised minimum residual if we had to solve a non-symmetric matrix iteratively.

1.8.3 Sparse matrices

These will normally be used at the domain and subdomain level, because there will not be any interaction between some of the basis functions on the components (whether they are elements or further subdomains) that go to make up any particular domain or subdomain if they are only supported on one of these components.

1.8.4 Dense matrices

These will normally be used at the element level because every degree of freedom interacts with all of the others, and in a general quadrilateral element there will not be any orthogonality between the basis functions. If, on the other hand, we are required to perform static condensation on the matrix that is assembled in a subdomain, then we must also use a dense matrix here because of the fill-in that this process creates.

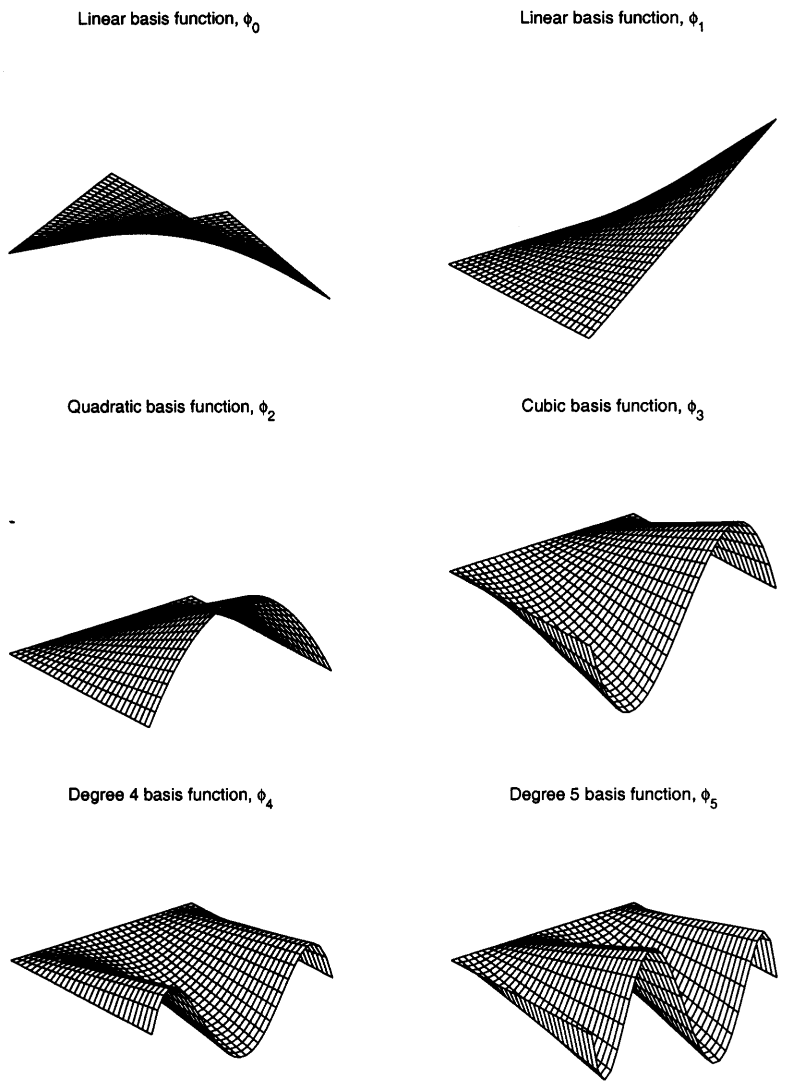
1.8.5 The matrix library

Both types of matrix have been implemented for this implementation, some with several types; they are described more fully in Chapter 4:

- dense matrix;
  - fortran matrix; and
  - dense row matrix;
- sparse matrix;

We will discuss another type of matrix, a distributed matrix, in the summary. The main operation, apart from any of the access operations, for any of the above matrix types is that of a matrix-vector product, as it is this operation that is *called* when we use an iterative solver.

1.9 Hierarchic basis functions



Linear basis function, ϕ_0
Linear basis function, ϕ_1
Quadratic basis function, ϕ_2
Cubic basis function, ϕ_3
Degree 4 basis function, ϕ_4
Degree 5 basis function, ϕ_5

Figure 1.8: Some of the hierarchic basis functions for an edge

In Figure 1.8 we show the basis functions up to degree 5 polynomials. The figure only shows the basis functions from a single edge, it does not show those from the other edges or the interior. The approximation along the edge of the element is made up of a sum of these basis functions, such as:

$$\Phi(x) = \sum_{i=0}^{5} \alpha_i \phi_i(x)$$

The coefficients α_i are from the solution of the stiffness matrix equation. When a p–refinement is required then it is simple just to add another, higher–degree polynomial to the element.

Chapter 2

Main data structures

In order to implement the $n-$irregular meshes, the choice of data structure to model the subdomain and domain is an important, if not crucial, feature of any code. In this chapter details are given of the design of the data structure and its implementation.

Some finite element codes number the elements and the degrees of freedom in such a way so as to minimise the bandwidth of the resulting stiffness matrix, [16, 60]. This is mainly because of the use of outdated programming languages, such as Fortran 77. However, using modern programming languages, e.g. Ada [21, 22, 30, 62], C++ [59], Fortran 90, more flexible data structures are able to be developed [38, 41, 52, 53]. This is good, as we are now able to eliminate this numbering, enabling a more flexible scheme to be used. So we can now number the degrees of freedom so as to make the construction of the connectivity mapping easier. The minimisation of the bandwidth was performed because of the fixed nature of the data structures used in the matrix, and to help reduce the fill-in if a direct solver is to be used. Although all the elements in all of the pictures are numbered, usually in a systematic way, in the code this is not the case — no such element numbering exists.

2.1 Structure of the domain

From a user's perspective, the most critical aspect of the code is the partition – this consists of the domain and the elements which make up the domain. The domain is the key data structure that the user interfaces with the problem, as it describes the geometry on which the problem to be approximated is solved. This is the data structure that *contains* all the elements in the domain, plus all connectivity information. In many implementations, the domain is implemented as an array, or several arrays, containing element information, connectivity mappings, etc. This array based approach is too restrictive for the amount of flexibility that we require, as it makes the adding of elements to the domain by refinement difficult as the maximum number of elements is fixed. Therefore another approach is needed — one possible approach with sufficient flexibility is a graph. For implementation of a graph (and other abstract data types) in Ada see [27]

2.1.1 Implementation of graphs

A general implementation of a graph is just a set of *nodes* and a set of *arcs* connecting them. The Figure 2.1 shows a graph with five nodes and six arcs. In our

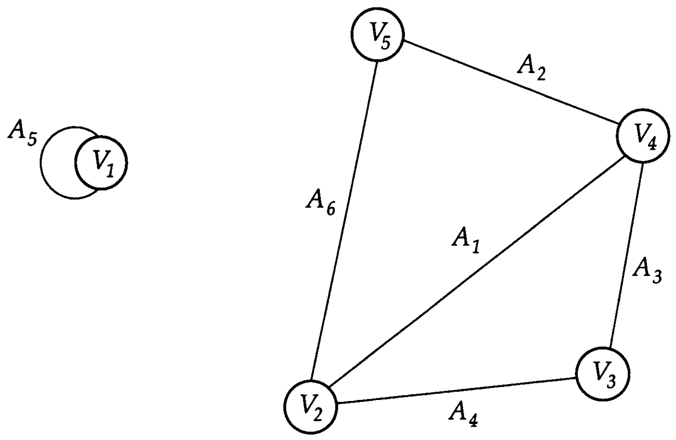


Figure 2.1: A simple graph

code it was decided to disallow the initial domain level from having constrained nodes, since the extra programming effort does not justify their inclusion. This restriction means we are able to simplify the general graph data structure by taking advantage of the knowledge that the nodes will have only three or four edges connected to them. This is because our nodes are the elements, and the only element shapes allowed are triangles and quadrilaterals. This is not too restrictive because, if we require a particular mesh to begin with, then we can refine the initial mesh appropriately.

While graphs are the most flexible data structure for the domain, allowing *any* domain to be represented, the graph data structure in this case can be greatly simplified as some sort of regularity exists:

- we will always have a *simple* graph, i.e. there will be no parallel arcs;
- the maximum *degree* of any node will be fixed, three for triangles and four for quadrilaterals;
- no *self loops*;
- the operations allowed can be greatly simplified, because we will not need to export many of them. The only operations where we will need to visit all the nodes or arcs will be hidden; and
- no need to find paths between nodes.

This regularity exists because of the *shape* of each of the vertices in the graph. The most difficult stage for the user is constructing the domain. The elements must be added to the domain with all the appropriate connectivity information embedded, so that the correct graph can be generated. The difficulty is in connecting the correct elements (nodes) together. For this we will make use of the global node numbers and the boundary conditions data. After a correct graph has been generated the user then has the option of doing some refinements, both

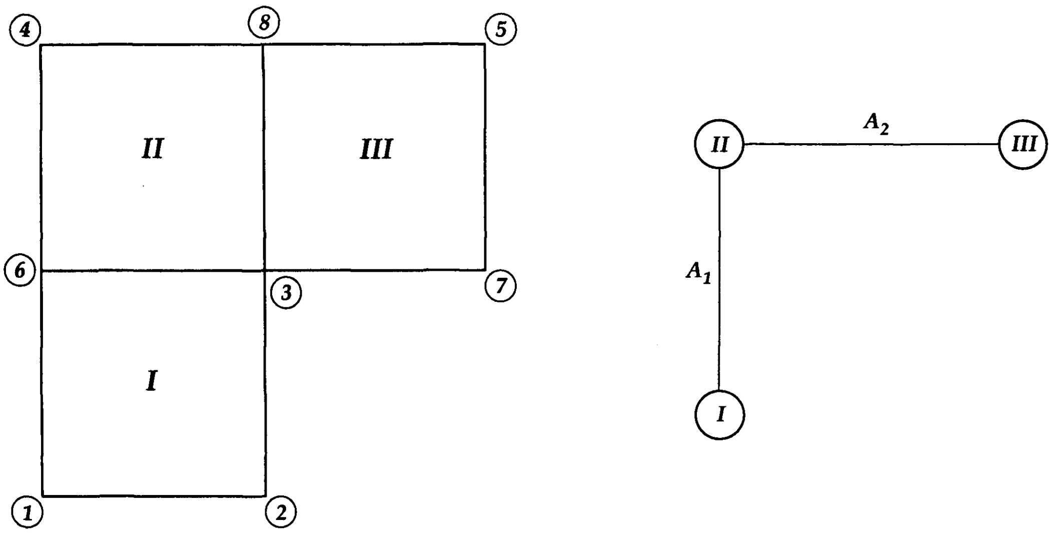


Figure 2.2: Simple domain with its graph representation

h and p, to get to the initial mesh for approximation. The domain in Figure 2.2 has three nodes and two arcs. Notice that, because we have a modified data structure for the graph, the graph representation in Figure 2.2 is not the same graph representation as that shown in Figure 2.3, which also has three nodes and

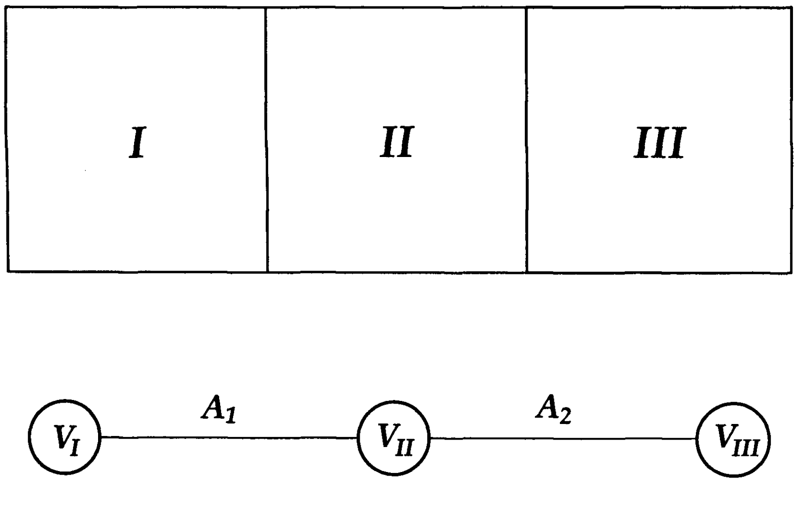


Figure 2.3: Different graph representation and the domain that it represents

two arcs, with each arc connecting the same node, implying that mathematically they are equal. Although the arcs connect the same nodes, the nodes are not connected along the same boundaries of the node; therefore each arc must also

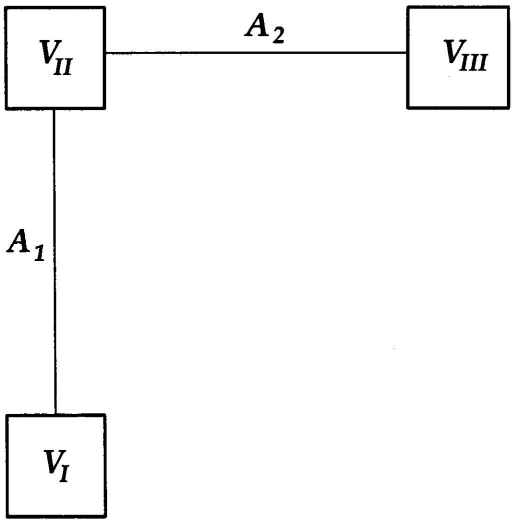


Figure 2.4: Modified graph representation

contain an additional piece of information, that of which boundary of the node it is connected to. This may sound like a complication of the graph, but actually it simplifies the construction a great deal, though the problem now is that the user must add information about the global node numbers (only the numbers for the linear degrees of freedom will be required, such as the numbers in circles shown in Figure 2.2) so that elements can have their proper edges matched. Below we have a code fragment taken from the programme that was used to construct the domain shown in Figure 2.2:

```
...
Degree := ((2, 2));
Element_Geometry := ((−1.0, 0.0), (0.0, 0.0),
                     ( 0.0, 1.0), (−1.0, 1.0));
Node_Numbers := (1, 2, 3, 6);
Construct (Element, Element_Geometry,
           Node_Numbers, Nodes, Boundary, Degree);
Add_Element_To_Domain (Domain, Element);
...
```

Since the domain level graph in Figure 2.2 remains the same even if the domain has been refined, because we change the element into a subdomain, all we need to do is to construct the stiffness matrices for the *super elements* or *subdomains* and their corresponding connectivity mappings. This is described in Section 2.2. Therefore each *graph vertex* can be assembled *independently* of any of the others. Hence we can assemble each of the contributions to the stiffness matrix concurrently but care must be taken when applying the connectivity mappings to avoid overwrites. This was achieved simply by performing this operation sequentially, which is fine because the time taken to perform this operation is negligible compared to the assembly and solving operations.

2.2 Structure of the subdomains

The subdomains are at the intermediate level in the hierarchy, between the domain and the element levels. They are required to assemble the stiffness matrix and perform any required static condensation from its component parts and then give the resulting matrix to the domain level for further accumulation and condensation, ready for the solving operation. The function of the subdomain is similar to that of the domain, except that the data structure does not need to be as flexible.

2.2.1 Structure of the subdomains

A standard refinement technique to refine a mesh is a regular refinement. For individual elements, in our case quadrilaterals, each edge is divided into half and

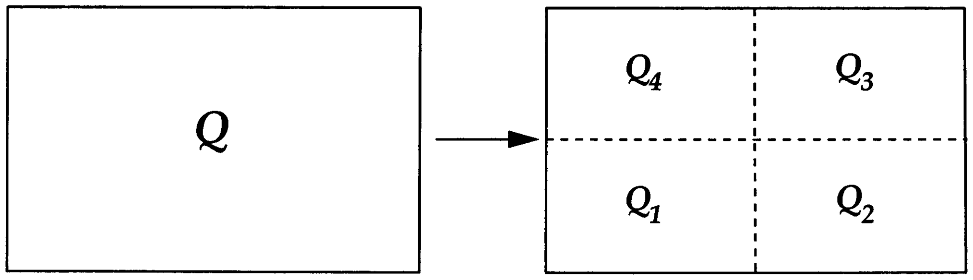


Figure 2.5: Refined element

then these midpoints are joined producing four new quadrilateral elements Q_1,

..., Q_4. This can be further refined in many ways. For example, in Figure 2.6 we

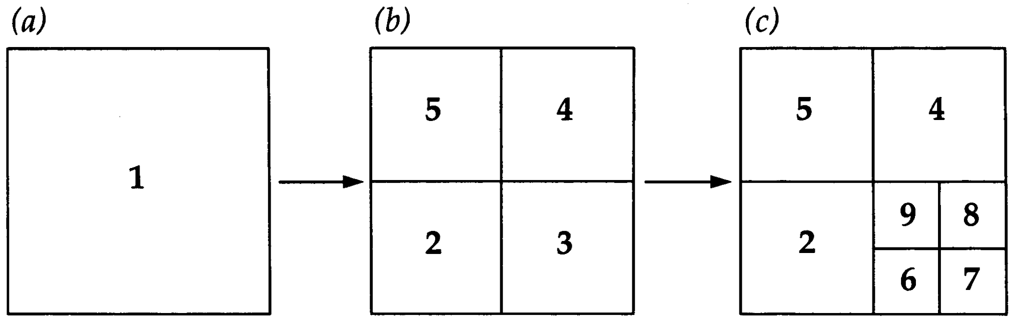


Figure 2.6: Multiple levels of refinement

have some kind of nested data structure, whereby one level is *embedded* within another. This is represented in the data structure as a tree, see Figure 2.7, and Figure 2.8, a quad tree in this case. These show how the tree structure represents the subdomain refinement within the code. This tree type data-structure opens up a whole new set of possibilities. Because static condensation, see [16, 17] and Chapter 4, is such a valuable part of the solving stage, we would obviously like to condense as much as we can. However, it is a very expensive process, therefore there exists a *trade-off* between the assembly–condensation time and the solve time. Using the tree data structure allows us to perform as much or as little static condensation as we require, giving a much finer control over the characteristics of the matrix system produced.

The refinement steps would go something like this: firstly locate the element that requires refinement, typically chosen because the approximation is not accurate enough in this element. Secondly we *change* this element into a subdomain, then finally allocate four new elements for the subdomain. This process can be repeated recursively until the required refinement is reached giving some thing like that shown in Figure 2.8.

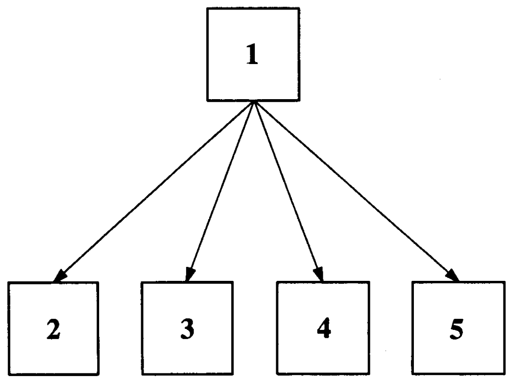


Figure 2.7: Quad tree representation for one level of refinement, represents (b) in Figure 2.6.

2.2.2 Subdomains as a graph

As with the domain, to construct the connectivity mapping for the subdomain we need some of the connectivity information between the structures in the subdomain. So, as before, we represent the subdomain with its elements as a graph. This time, though, we require less flexibility as there are only four structures in the subdomain; once again we use the interfaces to perform this task. Figure 2.9 shows the simple interconnection for the subdomain graph. In this figure we have the structures, whether they are elements or further subdomains, labelled as S_i, where $i \in \{1, 2, 3, 4\}$ are the vertexes of the graph. Each of these is connected by an arc, the interface, labelled E_i, $i \in \{1, 2, 3, 4\}$.

2.2.3 Parallelism within the subdomain

Exploitation of the inherent parallelism is achieved easily, where each level in the tree could be run as a separate process. This is because there is no interaction between the different branches in the tree during the construction phase. The only time that interaction occurs is when we want to accumulate the stiffness matrix from each of the stiffness matrices obtained from each of the branches. Indeed each level could be distributed onto another machine.

The global stiffness matrix can be computed from the local element stiffness

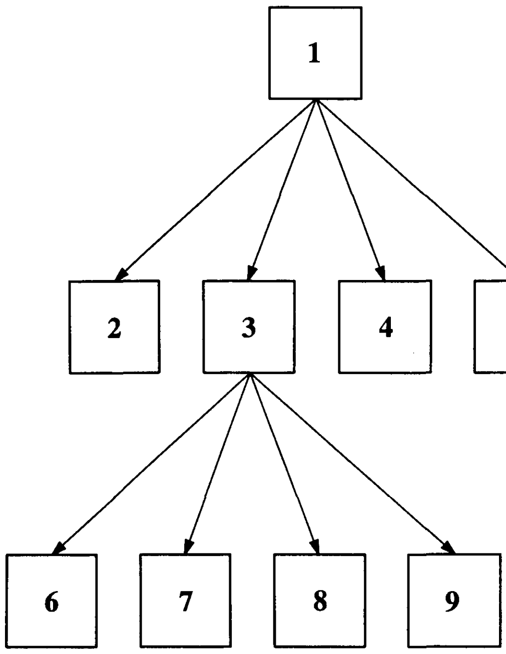


Figure 2.8: Quad tree representation for a non-uniform refinement, represents (c) in Figure 2.6

matrices as shown in (2.1), this will be developed in Chapter 3:

$$K = \sum_{e \in \Omega} \Lambda_e K_e \Lambda_e^t \tag{2.1}$$

From this we can see that the order of assembly is *not* important, therefore any of the stiffness matrices can be assembled and put into the global matrix. Therefore they can be assembled at the same time and then added to the global matrix. Care must be taken with this step so as to not allow any overwrites. The time taken to perform this step is very small compared to the time taken for the assembly of the local matrix contributions. Hence this step can be performed sequentially. We can also perform the static condensation step to each of the local matrix contributions *before* they are added to the global matrix, therefore we can also do this to each of the matrices at the parallel stage. Therefore we have a partial parallel direct solve.

Equation (2.1) can be used to develop a distributed parallel matrix–vector

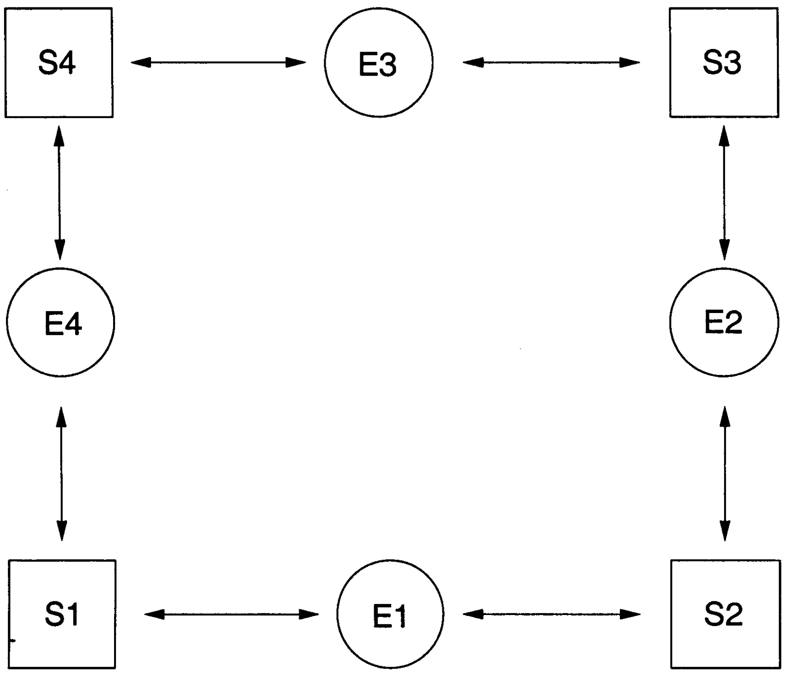


Figure 2.9: A subdomain represented as a graph

product quite simply. Consider:

$$\begin{aligned}
K\xi &= \sum_{e\in\Omega} \Lambda_e K_e \Lambda_e^t \xi \\
&= \sum_{e\in\Omega} \Lambda_e K_e (\Lambda_e^t \xi) \\
&= \sum_{e\in\Omega} \Lambda_e K_e \xi_e \\
&= \sum_{e\in\Omega} \Lambda_e (K_e \xi_e)
\end{aligned}$$

where $\xi_e = \Lambda_e^t \xi$. This enables a parallel matrix–vector product. The matrices need not be on the same machine, they could be distributed on to several machines. If we were using an iterative solver we could make use of this step, called an *element by element* method, see [23, 63], as the most expensive step is usually the matrix–vector product step. These can be generalised easily to a *subdomain by subdomain* method. If we keep the subdomain matrix at the subdomain level using a distributed matrix type this gives us more control over the amount of parallelism within the solving step.

2.2.4 Other types of refinement

We might be required to perform types of refinement that differ from the standard uniform refinement. For example, Figure 2.10 shows the optimal refinement around a singularity, see Chapter 5 for some numerical examples using the kind of refinement. Here we will examine only the the $h-$refinement, as the $p-$refinement does not affect the structure of the subdomain. The graph structure of the sub-

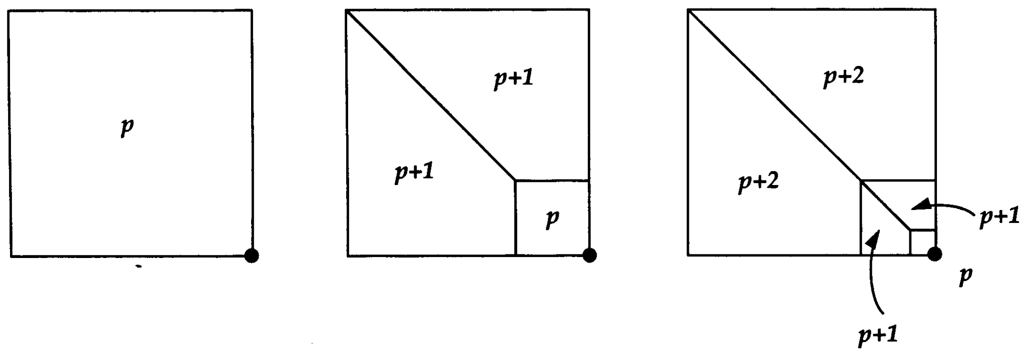


Figure 2.10: Refinement around a singularity, indicated by a ●

domain can be modified to accommodate this type of refinement. The modification is in the construction of the connectivity of the graph. Notice that we now do not have a quad tree, but a 3–tree. Another type of refinement, *anisotropic refinement*, leads to a binary tree. Here we require only a simple modification of the graph structure, again this modification is in the construction of the graph connectivity. So for all of these different types of refinement the only modification that is required is to the graph connectivity.

2.3 Structure of the interfaces

The interface between elements is represented by an arc in the graph, with additional functionality, such as assembling the connectivity mapping for the degrees of freedom along an interface.

2.3.1 How to construct the connectivity mapping

For regular meshes, or at least regular edges, this is just a simple matter of numbering the degrees of freedom along the edge. The order of this numbering will not be important for this case, just as long as we are consistent along both sides of the interface. Also the degree of the approximating polynomial is not important, just the number of degrees of freedom. So, for this case at least, there is no real implementation detail. However, not all meshes will be regular. Some, such as those pictured in Figures 2.11 and 2.14, will contain some hanging nodes, so construction of the connectivity mapping for these poses a greater difficulty. To make use of the process that is described in Chapter 3 for the construction of the connectivity mapping, we need to know the internal structure from both sides of the interface.

Use can be made of the hierarchical structure of the subdomains quite simply. The construction of the subdomains has been shown, and is a simple n-tree, where $n = 2, 3, 4$. But along each of the exterior boundaries of the subdomain (for a single refinement) there is only ever 1 or 2 elements. This will also be true for more levels of refinement, but instead of just elements there would be subdomains or elements depending on the refinement and our level in the tree. For constructing the constraint mapping we would like some form of recursive type data structure, so that the constraint mapping can be generated simply for any number of levels of refinement. The most obvious and simplest data structure to use is a *binary tree*. Each level in the tree may be either an element or a subdomain. If the item at a node is an element then we are at a leaf node otherwise we are not.

2.3.2 The interface tree

We can make use of the tree-like form of the subdomains in construction of the constraint mapping. The *normal* connectivity mapping is straight forward enough; however, consider the domain shown in Figure 2.11. If we isolate the

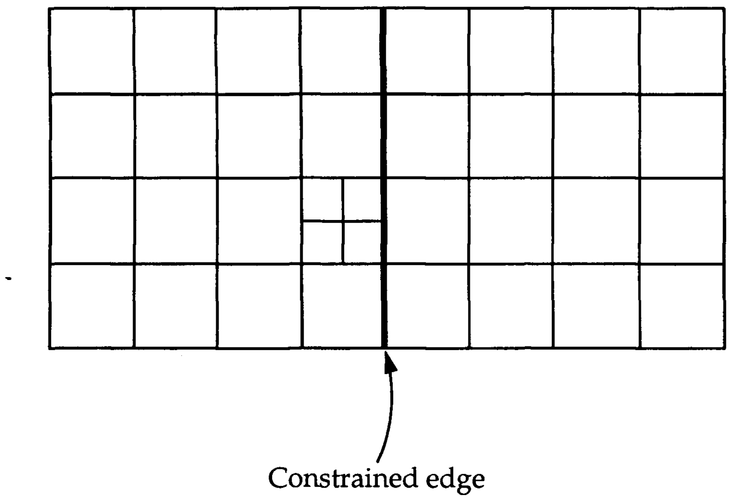


Figure 2.11: A highly refined domain

indicated interface and concentrate on the left side we get the situation shown in Figure 2.12. This interface is built up from a series of refinement steps and is shown in Figure 2.13, together with the interface tree. This binary tree structure

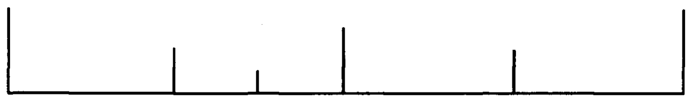


Figure 2.12: A nonuniform interface

is vital for the simple construction of the connectivity mappings in the case when we have constraints, as we need to make use the recursive nature of the binary tree for the method of construction of the connectivity mapping described in Chapter 3.

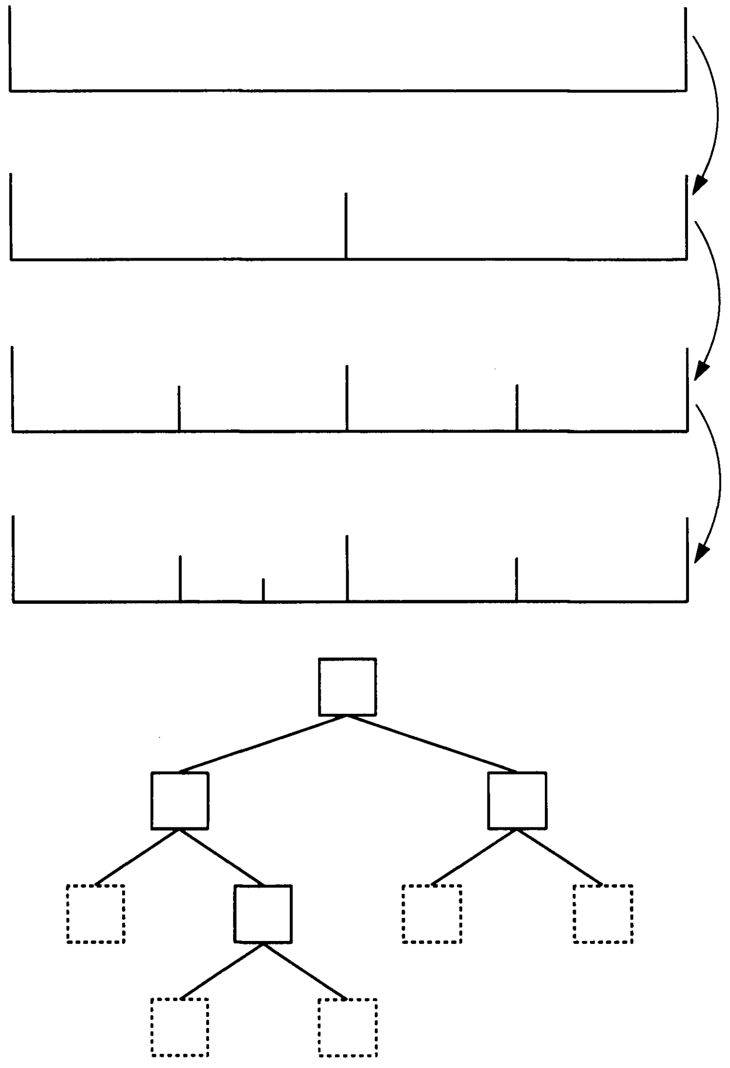


Figure 2.13: Showing where the edge tree came from

We may not always have a traditional type of refinement: again isolating the interface indicated with the coarser line we have Figure 2.14. The sequence of refinements that gave rise to this mesh is shown in Figure 2.16, so again we have a binary tree, although it does not appear to be as balanced (in some sense) as the previous example. But again using the recursive nature of the binary tree we can construct the connectivity mappings simply.

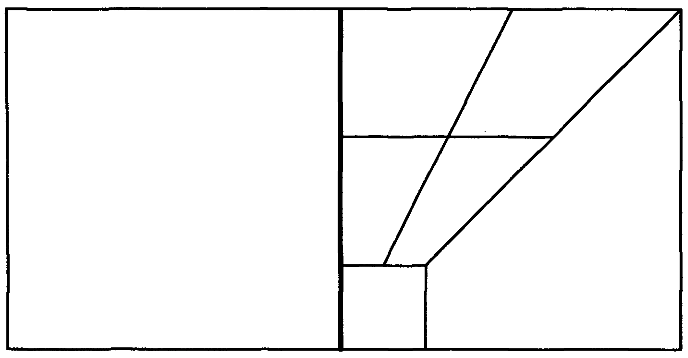


Figure 2.14: A mesh with both traditional, $\sigma = 0.5$, type refinements and $\sigma = 0.15$ type refinements

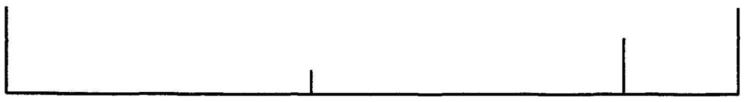
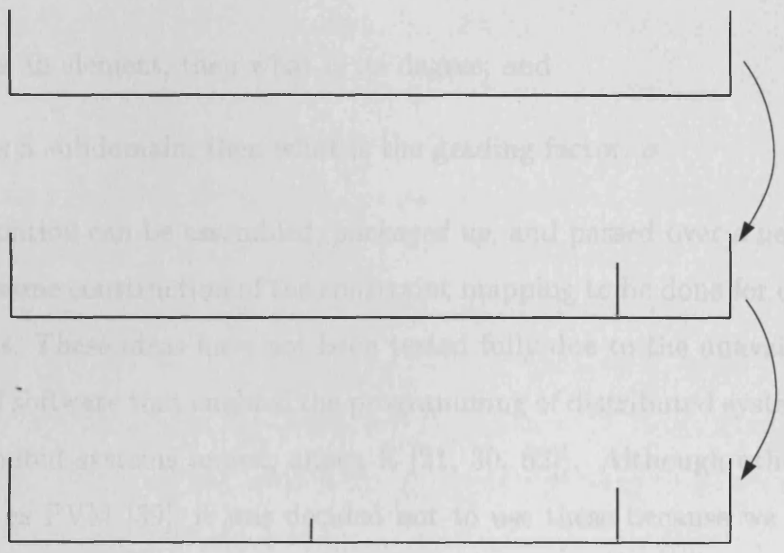


Figure 2.15: The isolated interface of the domain with both 0.15 and 0.5 type refinements



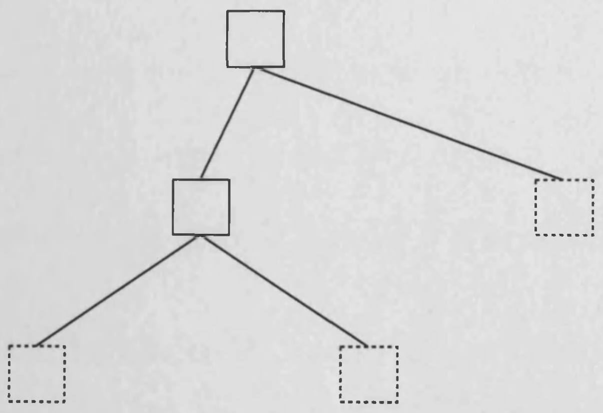


Figure 2.16: Interface with tree

2.3.3 Distribution

During the process for assembling the constraint mapping, the only information that is required is:

- whether the current item an element or subdomain;
- if it is an element, then what is its degree; and
- if it is a subdomain, then what is the grading factor, σ.

This information can be assembled, *packaged up*, and passed over a network, enabling the same construction of the constraint mapping to be done for distributed subdomains. These ideas have not been tested fully due to the unavailability, at the time, of software that enabled the programming of distributed systems in Ada (the distributed systems annex, annex E [21, 30, 62]). Although other systems exist, such as PVM [39], it was decided not to use these because we wanted to use a single language for as much of the system as was possible.

2.4 Structure of the elements

Elements are fundamental to all finite element programs. Contained within them are the properties of the problem that are to be approximated such as: boundary conditions, problem data etc. It was decided not to include any global degree of freedom information within the element structure. This was so that a minimum amount of information was stored at the element level, thus eliminating the possibility of non-local information being mistakenly altered, and to assist the parallelism by not requiring any global data accesses. This leads to a more distributed hierarchy of objects. The element's primary role now is just performing the most fundamental operations in any finite element code: that is, the construction of the element contributions (such as element stiffness matrix, load vector etc.), application of any boundary conditions and performing any static condensation. So now at any level in the mesh we just have to follow a small set of basic instructions:

1. assemble local system; then
2. perform any static condensation that is required; then
3. pass the assembled local system to next level above.

This simple *algorithm* is used at all levels, whether it is an element, a subdomain or a domain.

A number of other, globally visible, operations must also be defined, such as those that return information about the number of degrees of freedom in an element. This is to enable a simple construction of a conforming mesh.

Elements will also hold information about the approximation, i.e. the coefficients of the basis functions. This has to be done as the elements do not possess the ability to retrieve this data for themselves. This is a consequence of not having the global degree of freedom data defined at the element level.

2.4.1 Element requirements

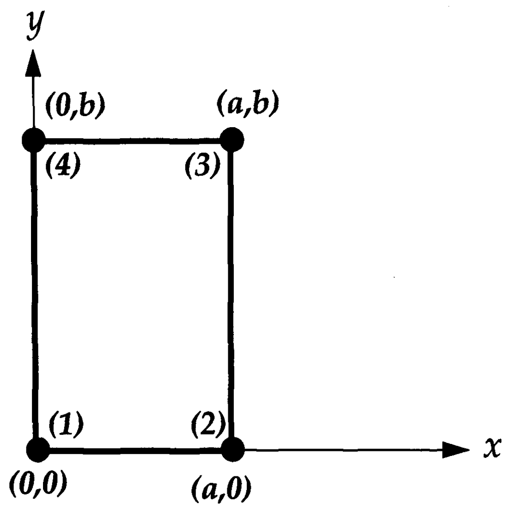


Figure 2.17: A rectangular element

Figure 2.17 shows a quadrilateral element (we will discuss only quadrilateral elements here, but the ideas also transfer directly to triangular elements). This has the following shape functions, or *geometry basis* functions:

$$\phi_1 = \left(1 - \frac{x}{a}\right)\left(1 - \frac{y}{b}\right) \tag{2.2}$$

$$\phi_2 = \left(1 - \frac{x}{a}\right)\frac{y}{b} \tag{2.3}$$

$$\phi_3 = \frac{x}{a}\frac{y}{b} \tag{2.4}$$

$$\phi_4 = \frac{x}{a}\left(1 - \frac{y}{b}\right) \tag{2.5}$$

The geometry basis functions for a more general element, such as that shown in Figure 2.18, are much more complicated. On this we have the approximation basis; these are the functions that we will use to approximate the solution to the partial differential equation. Some of the interior basis functions are shown in Figure 2.20. These are defined on $[-1, 1] \times [-1, 1]$, as this will enable a simple integration operation. So, rather than generating the approximation basis for each of the elements, we perform a mapping from the *reference element* [16, 25],

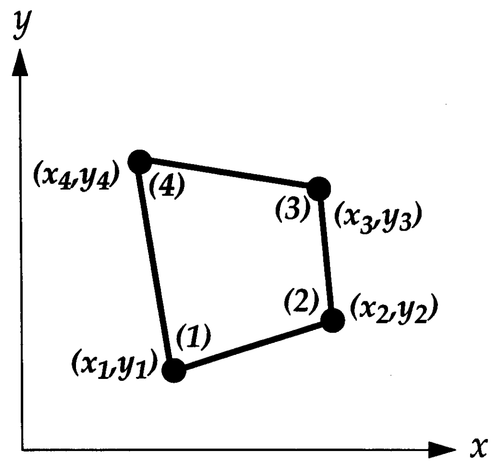


Figure 2.18: A general quadrilateral element

which is on $[-1,1] \times [-1,1]$ (for a quadrilateral), to the actual element, shown in Figure 2.19. Therefore we must modify the geometry basis, we now get:

$$\tilde{\phi}_1 = \frac{1}{4}(1-\xi)(1-\eta), \tag{2.6}$$
$$\tilde{\phi}_2 = \frac{1}{4}(1+\xi)(1-\eta), \tag{2.7}$$
$$\tilde{\phi}_3 = \frac{1}{4}(1+\xi)(1+\eta), \tag{2.8}$$
$$\tilde{\phi}_4 = \frac{1}{4}(1-\xi)(1+\eta). \tag{2.9}$$

Where the $\tilde{\phi}_i$ and ϕ_i are the basis functions on the reference element and the basis functions on the actual element respectively. The mapping from the reference element to the actual element is then:

$$\begin{aligned} x &= x_1\tilde{\phi}_1 + x_2\tilde{\phi}_2 + x_3\tilde{\phi}_3 + x_4\tilde{\phi}_4 \\ &= \boldsymbol{x} \cdot \tilde{\boldsymbol{\phi}} \\ y &= y_1\tilde{\phi}_1 + y_2\tilde{\phi}_2 + y_3\tilde{\phi}_3 + y_4\tilde{\phi}_4 \\ &= \boldsymbol{y} \cdot \tilde{\boldsymbol{\phi}} \end{aligned}$$

or put in programming terms:

```
Function_Evaluation_Point := Geometry_Matrix * Geometry_Basis;
```

This returns both the y and the x values that make up the coordinate. Here *Geometry_Matrix* is a matrix made up of the set of coordinates of the element, so we have:

$$\textit{Geometry\_Matrix} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \end{bmatrix},$$

and *Geometry_Basis* is each of the geometry basis functions: ϕ_i, evaluated at a particular point in the reference element:

$$\textit{Geometry\_Basis} = \begin{bmatrix} \tilde{\phi}_1 & \tilde{\phi}_2 & \tilde{\phi}_3 & \tilde{\phi}_4 \end{bmatrix}^t.$$

This mapping can be performed for curvilinear elements as well, [16, 25, 60].

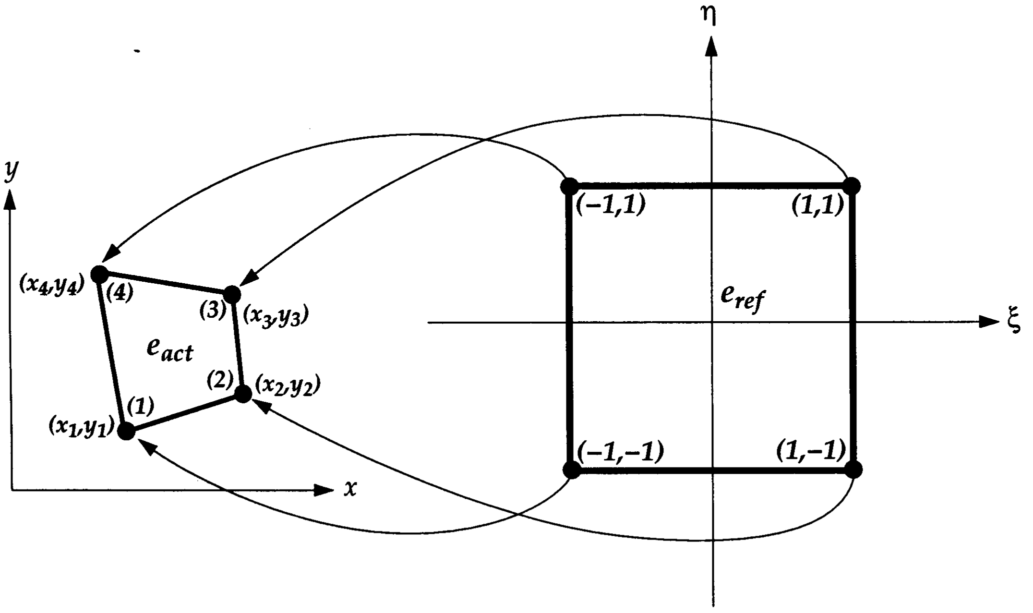


Figure 2.19: Mapping from the reference element to the actual element

We must transform the derivatives of the equation on the actual element to the corresponding derivatives on the reference element. This is achieved by making use of the chain rule for derivatives:

$$\begin{aligned} \frac{\partial}{\partial \xi} &= \frac{\partial x}{\partial \xi}\frac{\partial}{\partial x} + \frac{\partial y}{\partial \xi}\frac{\partial}{\partial y}, \\ \frac{\partial}{\partial \eta} &= \frac{\partial x}{\partial \eta}\frac{\partial}{\partial x} + \frac{\partial y}{\partial \eta}\frac{\partial}{\partial y}, \end{aligned}$$

or, put in matrix form:

$$
\begin{aligned}
\begin{bmatrix} \dfrac{\partial}{\partial \xi} \\ \dfrac{\partial}{\partial \eta} \end{bmatrix} &= \begin{bmatrix} \dfrac{\partial x}{\partial \xi} & \dfrac{\partial y}{\partial \xi} \\ \dfrac{\partial y}{\partial \eta} & \dfrac{\partial y}{\partial \eta} \end{bmatrix} \begin{bmatrix} \dfrac{\partial}{\partial x} \\ \dfrac{\partial}{\partial y} \end{bmatrix} \\
&= J \begin{bmatrix} \dfrac{\partial}{\partial x} \\ \dfrac{\partial}{\partial y} \end{bmatrix},
\end{aligned}
$$

where J is the Jacobian. This implies that, [57, 60]:

$$
\begin{aligned}
&\int_e \nabla\phi_i(x,y) \cdot \nabla\phi_j(x,y)\; dx\, dy = \\
&\int_{-1}^{1}\int_{-1}^{1} \left(\nabla\tilde{\phi}_i(\xi,\eta)J^{-1}(\xi,\eta)\right) \cdot \left(\nabla\tilde{\phi}_j(\xi,\eta)J^{-1}(\xi,\eta)\right) |J|\; d\xi\, d\eta \qquad (2.10)
\end{aligned}
$$

2.4.2 Numerical integration

We will use a Gaussian quadrature rule for the integration, [29, 44, 58]. The 1-dimensional integration rule is

$$\int_{-1}^{1} f(x)dx = \sum_{i=1}^{n} w_i f(x_i),$$

this integration rule is exact, for polynomials of up to degree $2n-1$. Some of the integration points that are used are shown in Table (2.1); many more can be found in [58]. For the 2 dimensional integrals over a quadrilateral that we require, we just form the tensor product of the 1 dimensional points, giving

$$\int_{-1}^{1}\int_{-1}^{1} f(x,y)dx\, dy = \sum_{i=1}^{n}\sum_{j=1}^{m} w_i w_j f(x_i, y_j). \qquad (2.11)$$

Again this would integrate polynomials with a maximum degree of $x^{2n-1}y^{2m-1}$ exactly.

n	x_i	w_i
1	0	2
2	0.57735 02691 89625	1
	-0.57735 02691 89625	1
3	0.77459 66692 41483	0.55555 55555 55555
	0	0.88888 88888 88888
	-0.77459 66692 41483	0.55555 55555 55555

Table 2.1: Gauss quadrature points and weights for up to $n = 3$

2.4.3 Putting this together

We can now apply all the previous work to assemble the element stiffness matrix, K, say for a Poisson type problem, with each entry, $k_{i,j}$, being: .

$$\begin{aligned} k_{i,j} &= \int_{-1}^{1}\int_{-1}^{1} \left(\nabla\tilde{\phi}_i(\xi,\eta)J^{-1}(\xi,\eta)\right)\cdot\left(\nabla\tilde{\phi}_j(\xi,\eta)J^{-1}(\xi,\eta)\right)|J|\, d\xi\, d\eta \quad (2.12)\\ & i,j = 1,\ldots,n. \end{aligned}$$

So for our linear quadrilateral element, with $n = 4$, we have:

$$K = \begin{bmatrix} k_{1,1} & k_{1,2} & k_{1,3} & k_{1,4} \\ k_{2,1} & k_{2,2} & k_{2,3} & k_{2,4} \\ k_{3,1} & k_{3,2} & k_{3,3} & k_{3,4} \\ k_{4,1} & k_{4,2} & k_{4,3} & k_{4,4} \end{bmatrix}, \qquad (2.13)$$

this is called the *stiffness matrix* [16, 42, 57, 60]. The entries in the matrix correspond to the integrations of the basis functions in equations (2.6) to (2.9) over the reference element. The following code fragment, taken from an actual program, assembles this matrix:

```
for I in Index range X_Integration_Points'Range loop
  Integration_Point(X_Coordinate) := X_Integration_Points(I);

  for J in Index range Y_Integration_Points'Range loop
    Integration_Point(Y_Coordinate) := Y_Integration_Points(J);
    Integration_Weight := X_Integration_Weights(I) * Y_Integration_Weights(J);

    Set_Geometry_Basis(Element, Geometry_Basis, Integration_Point);
    Set_Grad_Geometry_Basis(Element, Grad_Geometry_Basis, Integration_Point);
    Set_Grad_Approximation_Basis(Element, Grad_Basis, Integration_Point);
```

```
        Jacobian           := Geometry_Matrix * Grad_Geometry_Basis;
        Integration_Scale  := Integration_Weight * Determinant (Jacobian);
        Element_Basis      := Grad_Basis * Inverse (Jacobian);

        Stiffness_Matrix   := Stiffness_Matrix + Integration_Scale *
                              Element_Basis * Transpose (Element_Basis);
      end loop;

   end loop;
```

This sets the element stiffness matrix. Notice that we have two nested loops: this is because we have a two dimensional integration, shown in equation (2.11). Also notice that we have an *X_Integration_Points* and a *Y_Integration_Points.* This is because we might have different degrees of approximation in each of the variables. Thus, by not using the maximum degree integration rule, we have saved some of the computational work.

2.4.4 Approximation basis functions

An important feature of the finite element method are the functions used to approximate the solution, the *approximation basis.* We have used integrated Legendre polynomials for the quadratic and higher degree polynomials and Lagrange for the linear basis functions. These are defined as:

$$\phi_0(s) = \frac{1}{2}(1-s) \qquad \phi_1(s) = \frac{1}{2}(1+s)$$

and

$$\phi_k(s) = \int_{-1}^{s} P_{k-1}(t)\,dt; \quad k = 2, 3, \ldots$$

where $P_k(t)$ is the k^{th} Legendre polynomial. Figures 2.20 and 2.21 show some of the interior and boundary approximation basis functions.

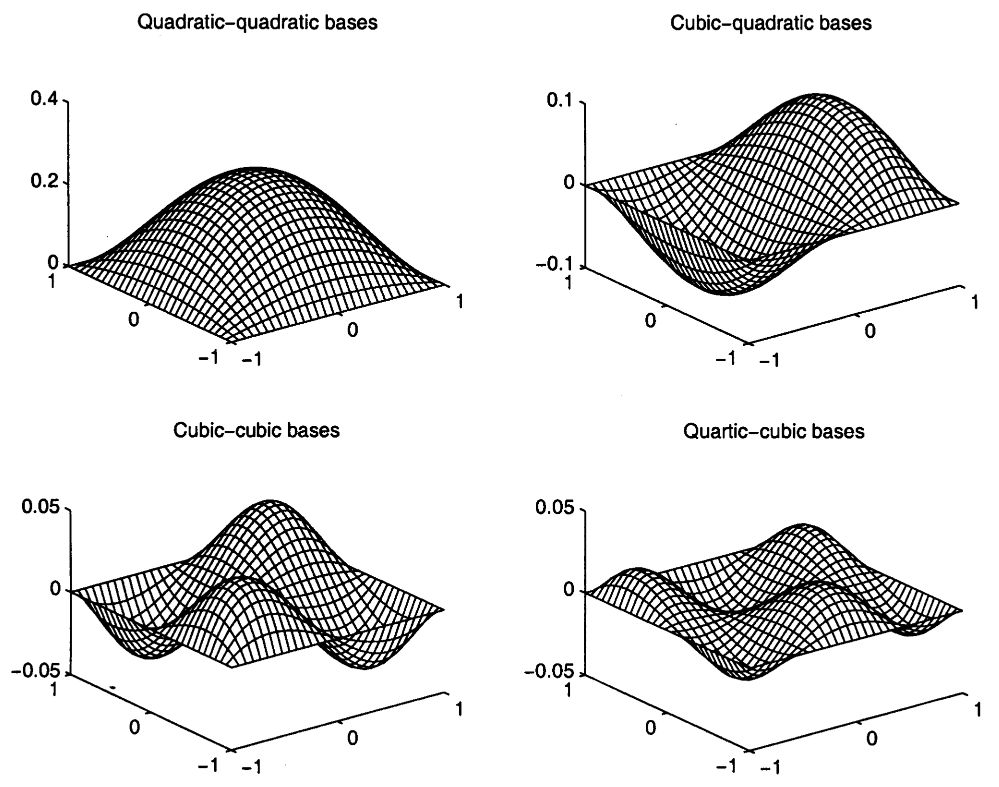


Figure 2.20: Some interior basis functions

2.4.5 General Laplacian elements

These are almost identical to the element described previously, with the exception that we might have to integrate the multiplying function, $a(x)$:

$$-\nabla \cdot (a(x)\nabla u) = f. \tag{2.14}$$

To do this we must change the construction of the stiffness matrix to include the following:

```
...
Function_Evaluation_Point := Geometry_Matrix * Geometry_Basis;
Integration_Scale         := Integration_Weight * Determinant (Jacobian) *
                               Function_A(Function_Evaluation_Point);
...
```

The rest of the process remains the same.

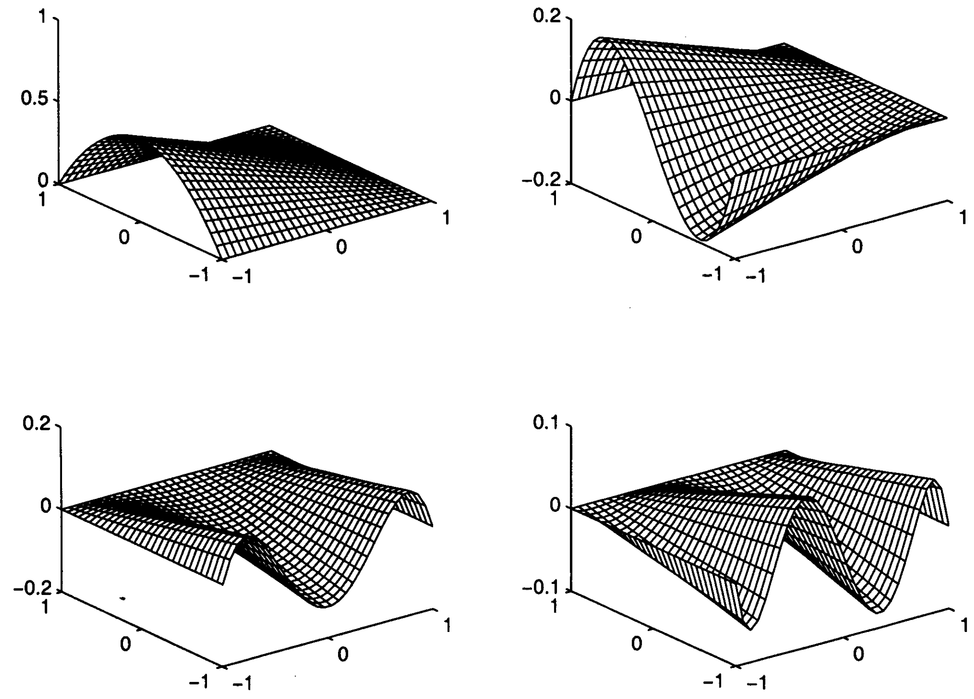


Figure 2.21: Some edge basis functions

2.4.6 Linear elasticity elements

The equations for linear elasticity are, see [57, 60]:

$$\frac{E(1-\nu^2)}{1-2\nu}\begin{bmatrix} \dfrac{\partial^2 u}{\partial x^2} + \dfrac{1-2\nu}{2(1-\nu)}\dfrac{\partial^2 u}{\partial y^2} + \dfrac{1}{2(1-\nu)}\dfrac{\partial^2 v}{\partial x \partial y} \\ \dfrac{1}{2(1-\nu)}\dfrac{\partial^2 u}{\partial x \partial y} + \dfrac{1-2\nu}{2(1-\nu)}\dfrac{\partial^2 v}{\partial x^2} + \dfrac{\partial^2 v}{\partial y^2} \end{bmatrix} = \begin{bmatrix} -F_x \\ -F_y \end{bmatrix}, \tag{2.15}$$

where u and v represent the x– and y–displacements, and E is Young's modulus and ν is Poisson's ratio. Notice this time we have a system of coupled equations to solve. The discreatisation leads to the the element stiffness matrix, B, defined to be:

$$B = \frac{E(1-\nu^2)}{1-2\nu}\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \tag{2.16}$$

where

$$B_{11} = \int\int \left(\frac{\partial \phi_i}{\partial x}\frac{\partial \phi_j}{\partial x} + \frac{1-2\nu}{2(1-\nu)}\frac{\partial \phi_i}{\partial y}\frac{\partial \phi_j}{\partial y} \right) dx\, dy, \tag{2.17}$$

$$
\begin{aligned}
B_{12} &= \iint \left(\frac{1}{2(1-\nu)} \frac{\partial \phi_i}{\partial x} \frac{\partial \phi_j}{\partial y} + \frac{1-2\nu}{2(1-\nu)} \frac{\partial \phi_i}{\partial y} \frac{\partial \phi_j}{\partial x} \right) dx\, dy, &(2.18)\\
B_{21} &= B_{12}^t, &(2.19)\\
B_{22} &= \iint \left(\frac{\partial \phi_i}{\partial y} \frac{\partial \phi_j}{\partial y} + \frac{1-2\nu}{2(1-\nu)} \frac{\partial \phi_i}{\partial x} \frac{\partial \phi_j}{\partial x} \right) dx\, dy. &(2.20)
\end{aligned}
$$

This can be assembled using the transformations described previously, though this time we must scale the basis differently, to take account of $\frac{1-2\nu}{2(1-\nu)}$ and $\frac{1}{2(1-\nu)}$ from equation (2.15) and equations (2.17) to (2.20). For a full description of linear elasticity see [60, 61].

2.4.7 Maintaining continuity

Although we only concern ourselves with the construction of the element contributions, we must still think, to some extent, about the overall global picture — especially the matching of the element bases across the interelement boundaries. Particular care must be taken to ensure that the basis function match. The problem can be eliminated with linear and quadratic basis functions because they can

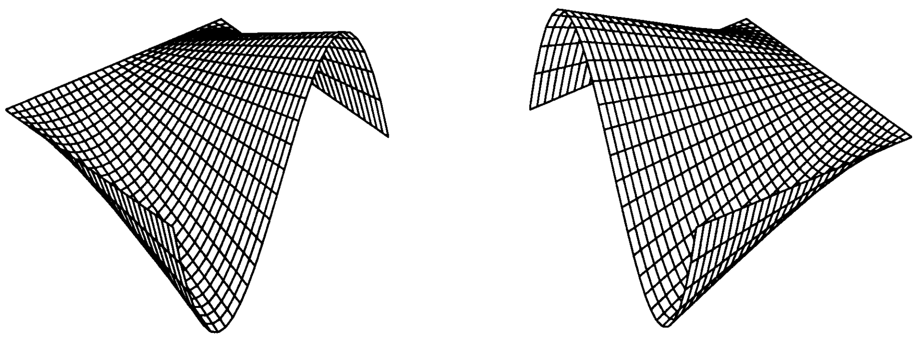


Figure 2.22: Matching element basis functions

be defined to be always positive (or negative). Cubics are the first of the bases to display the problem. Figure 2.22 shows cubics that match, but in Figure 2.23 we can see that we must get the basis functions to match on both sides of the interelement boundary. This is easily achieved by simply scaling one of the basis functions by -1. This problem leads to the introduction of the *basis direction* attribute, and all elements will need this. It contains the appropriate values so that the tensor product of the one dimensional bases leads to correct continuous basis functions.

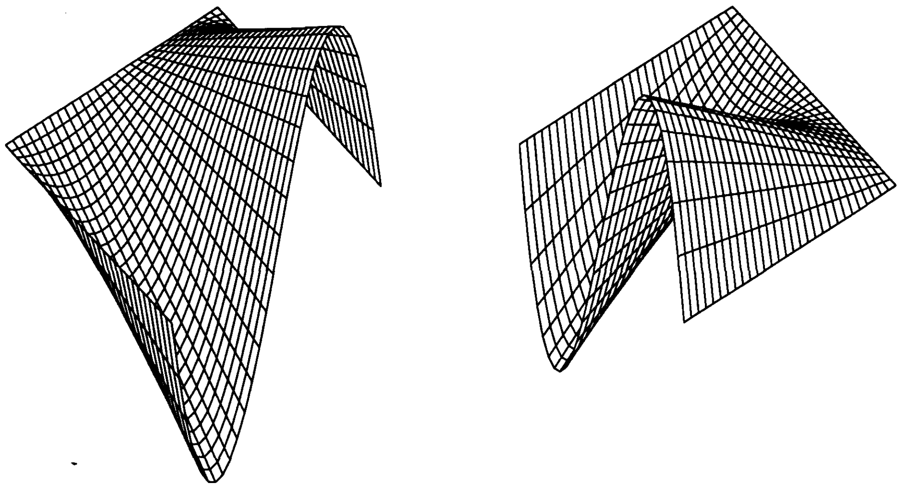


Figure 2.23: Unmatching element basis functions

2.5 Structure of the connectivity mappings

The connectivity mappings are used to map the local contributions to the more global ones. As they are used at every level in the subdomain, we would ideally like them to be as efficient as possible. They were designed using an object orientated design methodology. This approach was useful in identifying the operations required for each of the different refinement methods used throughout this implementation. We can see that on any one interface we might have any of, or possibly a combination of, the following refinement types:

- a uniform mesh, leading to a Boolean mapping; or
- h–constraint, leading to constrained nodes; or
- p–constraint, again leading to constrained nodes.

Having identified the three possibilities for any constraint mapping, we realise that there are only two operations required after the mapping has been assembled: accumulate the element contributions and restriction of the global contribution, usually the global approximation vector, to the more local level, whether that be a subdomain or element level. So therefore we have three types:

1. Boolean Mapping;

2. H Constraint Mapping; and

3. P Constraint Mapping.

Each degree of freedom will have an associated entry in the connectivity mapping that will be one of the above three. We now have a problem: in many partitions there will be many thousands or hundreds of thousands of degrees of freedom. This will mean that there will be millions of dispatching operations, which could potentially slow the progress of the programme greatly. Therefore another implementation technique must be looked at.

2.5.1 Variant programming

We decided to implement the connectivity mappings using *variant programming* techniques, see [22]. After the basic design of the types and their associated operations was done using the object orientated approach, we took these and implemented the mapping using the variant programming.

This also helped us reduce the overall resource usage from this part of the implementation, because now each of the entries in the mapping will not have a *dispatch table* or the associated access value of one for the particular type, so saving many millions of bytes of memory. An example, in Ada 95, of variant programming is given with the representation of the connectivity mappings used in the implementation:

```
type Mapping_Type is
      (Boolean_Mapping,
       H_Constraint_Mapping,
       P_Constraint_Mapping);

type Generalised_DOF (Map : Mapping_Type := P_Constraint_Mapping) is

  record

    case Map is
      when Boolean_Mapping       => Global_DOF_Number : Integer;
      when H_Constraint_Mapping  => Constraints       : Constraint_List;
      when P_Constraint_Mapping  => null;
    end case;

  end record;

type Connectivity_Mapping is array (Integer range <>) of Generalised_DOF;
```

The *discriminant*, *Map* in this case, can have any of the three values defined by the enumeration type *Mapping_Type*, the components of the record depend on this value. So, for example, if the discriminant has the value *Boolean_Mapping* then the component will be *Global_DOF_Number* which is of type *Integer*, only accessing this component is correct any other will result in a run–time error. For the case when *Map* has the value *H_Constraint_Mapping*, we have the component *Constraints* (of type *Constraint_List*), this is a set of $(\mathbb{N}, \mathbb{R})$ pairs that represent the constraints for the particular degree of freedom.

Notice the default of *P_Constraint_Mapping*, this is because the *p*–constraint is the simplest to assemble as it is just a null operation. This is a programme optimisation, enabling us to skip doing anything when we are assembling the connectivity mapping for the *p*–constraints. So in the operations we now have to have a simple *if–statement* to check for each of the three possibilities; this has taken the place of the dispatching operation.

2.6 Conclusions

The use of a graph for the domain enables an infinite number of domains to be represented, even those which consist of many distinct, nonconnecting, regions.

It is obvious from all this that the tree structure for the subdomains and from this the binary tree for the interfaces is a very important part of the implementation. It enables the simple construction of the connectivity mappings to make the approximation continuous. Also the subdomain's tree structure enables a very effective parallel computation to be used.

The element data structure forms a simple way of interfacing with the problem, without having to be concerned with the connectivity mappings, static condensation, parallelism, etc. The user need only concern themselves with the construction of the element contributions to the problem, the stiffness matrix and load vector. There are problems that may require a different approximation or geometry basis; these too can be simply added. The addition of a basis function type could be possible, and would be worth looking at in the future. This would enable us to just *plug in* the required basis function type. This is especially true for the geometry basis, as the approximation basis is unlikely to change from element to element. A problem with changing the approximation basis is that the constraint constants polynomials must also be changed. A better approach for this would be achieved using generics and the *signature* approach, see [21].

The interfaces make use of the known structure of the subdomains to help construct the connectivity mappings. Their representation could be enhanced by making use of the *incidence* model for the arcs in a graph. This would separate out the connectivity of the graph from the tree structure that is used to construct the connectivity mapping. It might also help the implementation of the distributed subdomains. The incidence is shown in Figure 2.24. S_i represent the subdomains, E_i the edges (arcs in the graph) and I_i represent the incidences. (Compare it with Figure 2.9.) These would contain the interface trees that are now contained within the edges themselves. However, what has been implemented is only a short step away from the incidence model as we have just moved the graph incidence into the arc itself.

The connectivity mappings are an important feature of any finite element implementation, never more so than in a fully adaptive *hp*-version, as there is a wide variety of meshes that we could come across. So an efficient implementation is very important. We believe that the object orientated design and subsequent implementation as a variant record has given them this efficiency.

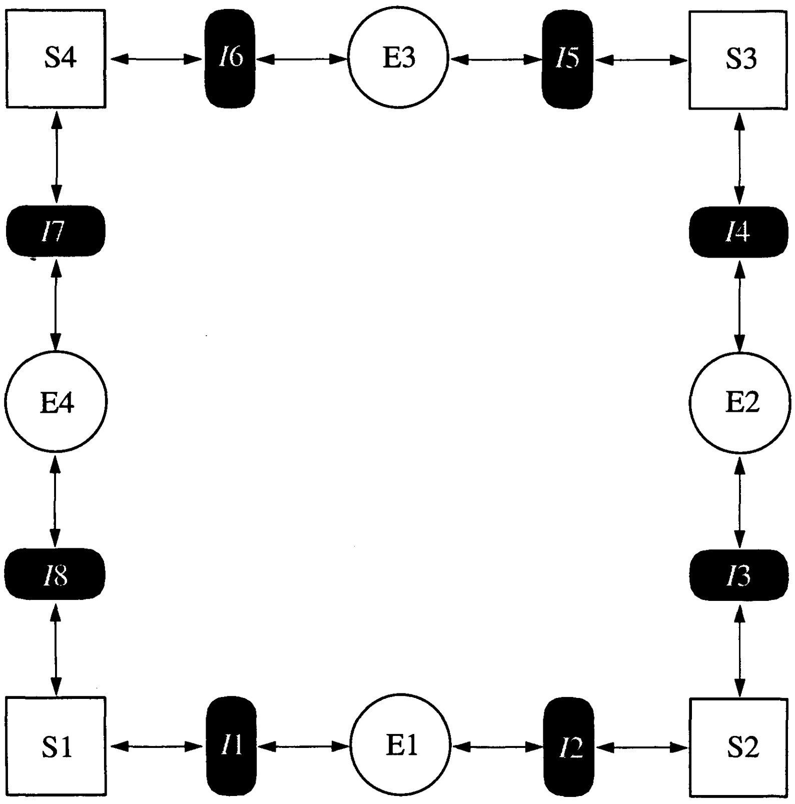


Figure 2.24: The incidence graph model

Chapter 3

The connectivity mapping

Figure 3.1 shows a typical mesh that could be obtained from the finite element method; this one was obtained from [10]. The mesh contains elements of different polynomial orders and element of different sizes next to each other. The construction of a conforming approximation from the mesh is not a simple matter and requires a flexible data structure. In this chapter ideas are discussed that have enabled such meshes to be managed simply and efficiently. We first develop the classical data structure, [16, 57], through a series of refinements: the *full blown* data structure will be a generalisation [12] of the classical data structure — this is the key idea behind this approach.

3.1 The classical approach

Consider the simple mesh in Figure 3.2 consisting of two linear elements. The global degree of freedom numbers are circled and the parenthesised numbers are the local degree of freedom numbers, local to each of the elements. There is a global approximation vector, which for this mesh consists of the six global degree of freedom values. Similarly each element has its own approximation vector consisting, in this case, of four values: the values of the local degrees of freedom.

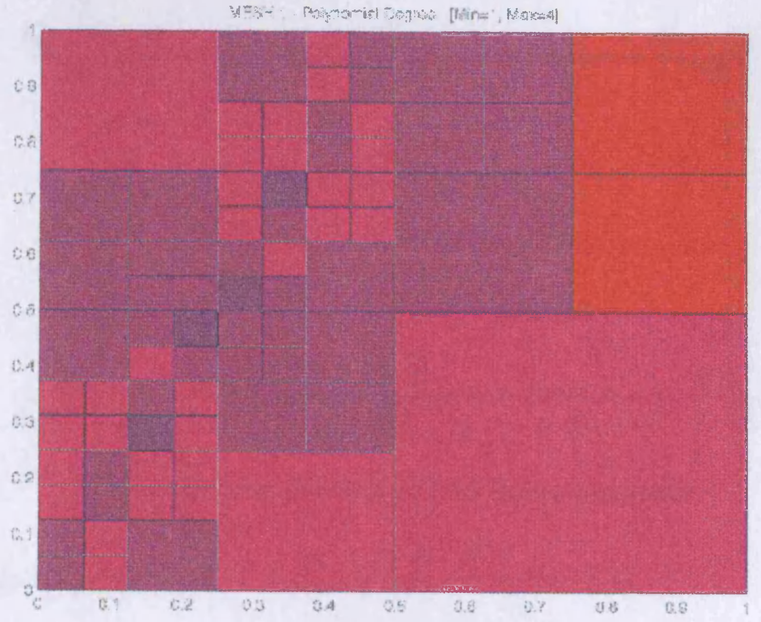


Figure 3.1: An actual *hp*-mesh, (red shading indicates low order elements, blue higher order elements).

So, the global approximation vector is

$$\boldsymbol{x} = (x_1, x_2, x_3, x_4, x_5, x_6) \tag{3.1}$$

and the element approximation vectors are

$$\boldsymbol{x}_I = (x_1, x_2, x_5, x_6) \tag{3.2}$$

$$\boldsymbol{x}_{II} = (x_2, x_3, x_4, x_5) \tag{3.3}$$

3.1.1 Assembling the stiffness matrix

From the mesh, in Figure 3.2 we have the following stiffness matrix. The superscripts and subscripts on each entry correspond to the element number and local

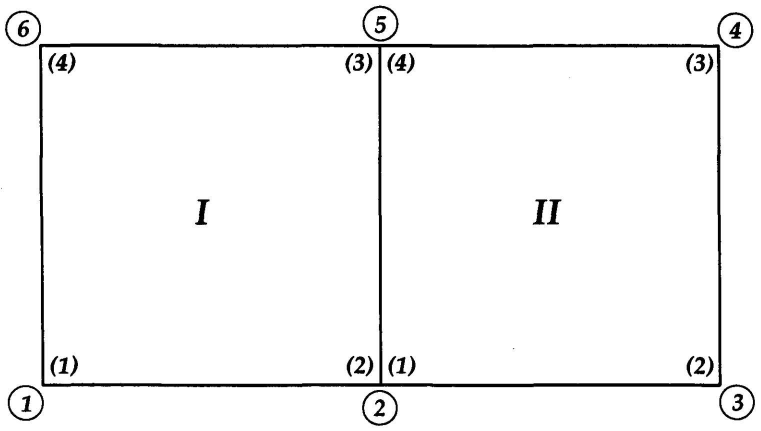


Figure 3.2: A Domain of two linear elements

node number respectively.

$$K_{global} = \begin{bmatrix} k^I_{1,1} & k^I_{1,2} & 0 & 0 & k^I_{1,3} & k^I_{1,4} \\ k^I_{2,1} & k^I_{2,2}+k^{II}_{1,1} & k^{II}_{1,2} & k^{II}_{1,3} & k^I_{2,3}+k^{II}_{1,4} & k^I_{2,4} \\ 0 & k^{II}_{2,1} & k^{II}_{2,2} & k^{II}_{2,3} & k^{II}_{2,4} & 0 \\ 0 & k^{II}_{3,1} & k^{II}_{3,2} & k^{II}_{3,3} & k^{II}_{3,4} & 0 \\ k^I_{3,1} & k^I_{3,2}+k^{II}_{4,1} & k^{II}_{4,2} & k^{II}_{4,3} & k^I_{3,3}+k^{II}_{4,4} & k^I_{3,4} \\ k^I_{4,1} & k^I_{4,2} & 0 & 0 & k^I_{4,3} & k^I_{4,4} \end{bmatrix} \tag{3.4}$$

This can be decomposed into a sum of the two element stiffness matrices:

$$K_{global} = \begin{bmatrix} k^I_{1,1} & k^I_{1,2} & 0 & 0 & k^I_{1,3} & k^I_{1,4} \\ k^I_{2,1} & k^I_{2,2} & 0 & 0 & k^I_{2,3} & k^I_{2,4} \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ k^I_{3,1} & k^I_{3,2} & 0 & 0 & k^I_{3,3} & k^I_{3,4} \\ k^I_{4,1} & k^I_{4,2} & 0 & 0 & k^I_{4,3} & k^I_{4,4} \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & k^{II}_{1,1} & k^{II}_{1,2} & k^{II}_{1,3} & k^{II}_{1,4} & 0 \\ 0 & k^{II}_{2,1} & k^{II}_{2,2} & k^{II}_{2,3} & k^{II}_{2,4} & 0 \\ 0 & k^{II}_{3,1} & k^{II}_{3,2} & k^{II}_{3,3} & k^{II}_{3,4} & 0 \\ 0 & k^{II}_{4,1} & k^{II}_{4,2} & k^{II}_{4,3} & k^{II}_{4,4} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.5}$$

Consider the *raw* element stiffness matrix from element I. This is assembled in the usual way:

$$K_I = \begin{bmatrix} k^I_{1,1} & k^I_{1,2} & k^I_{1,3} & k^I_{1,4} \\ k^I_{2,1} & k^I_{2,2} & k^I_{2,3} & k^I_{2,4} \\ k^I_{3,1} & k^I_{3,2} & k^I_{3,3} & k^I_{3,4} \\ k^I_{4,1} & k^I_{4,2} & k^I_{4,3} & k^I_{4,4} \end{bmatrix} \tag{3.6}$$

What is needed is a mapping that takes each entry in the element stiffness matrix and maps it to the correct position in the global stiffness matrix:

$$\tilde{K}_I = \begin{bmatrix} k^I_{1,1} & k^I_{1,2} & 0 & 0 & k^I_{1,3} & k^I_{1,4} \\ k^I_{2,1} & k^I_{2,2} & 0 & 0 & k^I_{2,3} & k^I_{2,4} \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ k^I_{3,1} & k^I_{3,2} & 0 & 0 & k^I_{3,3} & k^I_{3,4} \\ k^I_{4,1} & k^I_{4,2} & 0 & 0 & k^I_{4,3} & k^I_{4,4} \end{bmatrix} \tag{3.7}$$

This is achieved by what is known as a *Boolean matrix.* The Boolean matrix for element I is,

$$\Lambda_I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.8}$$

Given the element stiffness matrix, see equation (3.6), for example, a pre- and post-multiplication of K_I by Λ_I, as show in equation (3.9), gives equation (3.7), the *distributed* element stiffness matrix

$$\tilde{K}_I = \Lambda_I K_I \Lambda_I^t. \tag{3.9}$$

Similarly for element II, we get:

$$\Lambda_{II} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.10}$$

which can then be used to give $\tilde{K}_{II} = \Lambda_{II} K_{II} \Lambda_{II}^t$. So to get equation (3.4) we have to sum the mapped element stiffness matrix:

$$\begin{aligned} K_{global} &= \tilde{K}_I + \tilde{K}_{II} \\ &= \Lambda_I K_I \Lambda_I^t + \Lambda_{II} K_{II} \Lambda_{II}^t \\ &= \sum_{e \in \{I, II\}} \Lambda_e K_e \Lambda_e^t \end{aligned}$$

or more generally:

$$K_{global} = \sum_{e \in \Omega} \Lambda_e K_e \Lambda_e^t \tag{3.11}$$

A similar process can be developed for the load vector, resulting in:

$$\boldsymbol{G}_{global} = \sum_{e \in \Omega} \Lambda_e \boldsymbol{G}_e. \tag{3.12}$$

Also, the distribution of the approximation to each of the elements is done as

$$\boldsymbol{x}_I = \Lambda_I^t \boldsymbol{x}, \quad \boldsymbol{x}_{II} = \Lambda_{II}^t \boldsymbol{x}. \tag{3.13}$$

Because they are very sparse, storing the full Boolean matrix would be a very inefficient use of the available storage. Also, as the only nonzero entries are ones, it is possible to devise a data structure that minimises the overall storage for the Boolean matrices.

3.1.2 The classical storage scheme

Looking at the two boolean matrices in equations (3.8) and (3.10), the only non zero entries are ones, and there is only one non zero entry per column. All that we need to do is to store the row position of this entry. The classical method, [16, 57], for storing these is just an array of integers, where the index number is the column position and the value stored at this index is the row position. so, for example, for the mesh shown in Figure 3.2 we have:

$$\begin{aligned}\Lambda_I &= (1,2,5,6)\\ \Lambda_{II} &= (2,3,4,5)\end{aligned}$$

although this is usually stored as a two dimensional array, see equation (3.14), where the first index, the row, is the element number and the second index is the usual Boolean matrix, as described above. This is usually called *ELNODE*, see [16].

$$ELNODE = \begin{bmatrix} 1 & 2 & 5 & 6 \\ 2 & 3 & 4 & 5 \end{bmatrix} \tag{3.14}$$

The problem with this representation is that the mesh must be uniform, must have a uniform degree throughout the mesh and there must be no constrained degrees of freedom.

In most finite element codes the Boolean mappings take the form of an array, which can represent the data in a convenient and very compact form. They are essential to *all* finite element codes.

$$\text{Rep } (\Lambda_I) = [\{1\},\{2\},\{5\},\{6\}] \tag{3.15}$$

$$\text{Rep } (\Lambda_{II}) = [\{2\},\{3\},\{4\},\{5\}] \tag{3.16}$$

3.2 Nonuniform *p*-refinement

Suppose now that the mesh in Figure 3.2 has its element II p-refined to a quadratic, with element I remaining the same to give Figure 3.3. The problem that

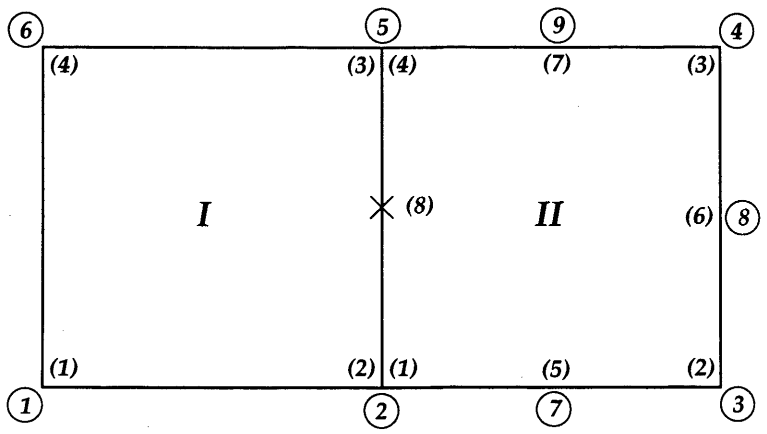


Figure 3.3: A simple nonuniformly p-refined mesh

we have now is to maintain continuity across the interelement boundary; there are two things that could be done:

1. increase the approximating degree along the interelement boundary in element I [31, 32]; or
2. decrease the approximating degree along the interelement boundary in element II.

The preferred approach is to choose 2. This will greatly simplify the implementation for, possibly, a small loss in accuracy. Therefore, the value of the degree of freedom (locally numbered with an eight) must be constrained to be zero, so the the approximation along the interface can be linear; thus it is not assigned a global degree of freedom number. We will *always* do this type of constraining for the p-refined case, i.e. set the order of approximation along a common interelement boundary to the minimum of the elements that share the interelement boundary.

However, this kind of constraining requires a modification in the Boolean matrix (connectivity mapping) and the connectivity mappings for both elements require updating, especially that of element II (obviously due to the increase in the number of degrees of freedom) thus:

$$\Lambda_I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \Lambda_{II} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

notice that the last column is all zeros, this is because the corresponding degree of freedom is constrained to zero. They have the corresponding representation:

$$\begin{aligned} \texttt{Rep}\ (\Lambda_I) &= [\{1\},\{2\},\{5\},\{6\}] \\ \texttt{Rep}\ (\Lambda_{II}) &= [\{2\},\{3\},\{4\},\{5\},\{7\},\{8\},\{9\},\{\}]. \end{aligned}$$

Notice in particular the $\{\}$ in the last entry for Rep (Λ_{II}). This empty set indicates the fact that the degree of freedom *has not* been assigned a global degree of freedom number. Also notice that Rep (Λ_I) has remained unchanged; the same is also true for the contributions to the global stiffness matrix and load vector. So it is possible to use the same matrix and vector from the previous solve step; had we increased the order of approximation along the interelement boundary for element I, this would not be the case. This is another reason for making this choice. As soon as we have the connectivity mappings the usual assembly procedure is followed, as described in equation (3.11); as is the procedure for distribution of the approximation vector.

3.3 Generalising to nonuniform h-refinements

In the previous section we supposed a nonuniform p-refinement of the mesh; but alternatively assume that the element had been h-refined giving the mesh in Figure 3.4. Again, the key problem is to maintain continuity across the interele-

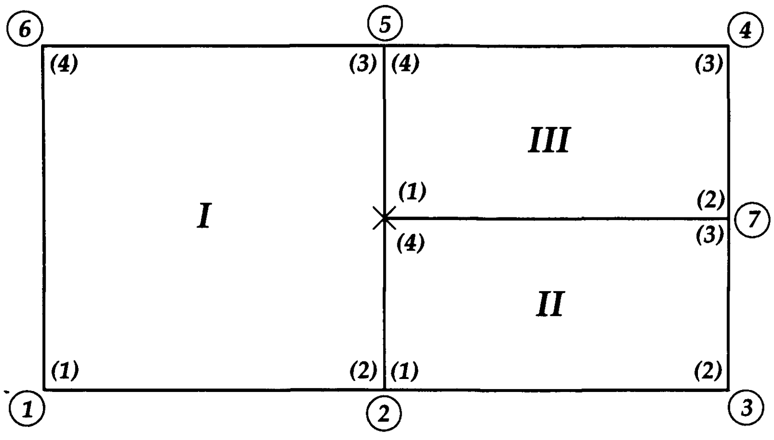


Figure 3.4: A simple nonuniformly h-refined mesh

ment boundary, so therefore the degree of freedom, labelled in Figure 3.4 with a $\times$, must be appropriately constrained. So again, it is not assigned a global degree of freedom number. These degrees of freedom are usually described as being *constrained* or *hanging.*

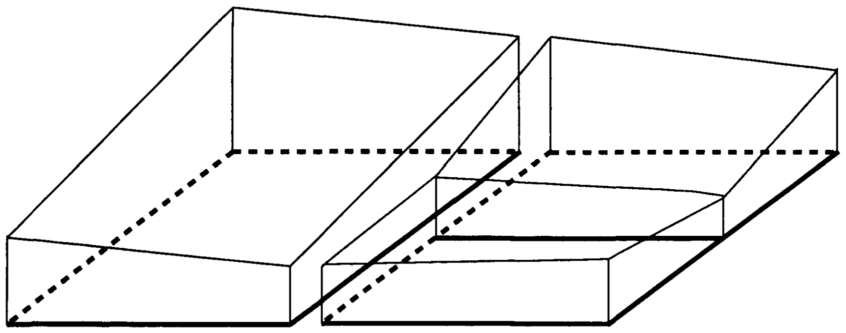


Figure 3.5: Problem showing an example of unconstrained discontinuous approximation

However, this time it is not just a simple matter of setting the constrained value to be zero as in the previous case, because element I requires that the approximation across the interelement boundary be linear. Thus on distribution of a global approximation vector we would assign the local degree of freedom numbered 4 for element *II* the value of an average of the values of the global degrees of freedom numbered 2 and 5; similarly for the local degree of freedom numbered 1 in element *III*. We would then get the following connectivity mappings:

$$\Lambda_I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \Lambda_{II} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & \frac{1}{2} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \Lambda_{III} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{1}{2} & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{3.17}$$

with their corresponding representations:

$$\begin{aligned} \text{Rep}\ (\Lambda_I) &= [\{1\},\{2\},\{5\},\{6\}] \\ \text{Rep}\ (\Lambda_{II}) &= [\{2\},\{3\},\{7\},\{(2,1/2),(5,1/2)\}] \\ \text{Rep}\ (\Lambda_{III}) &= [\{(2,1/2),(5,1/2)\},\{7\},\{4\},\{5\}] \end{aligned} \tag{3.18}$$

It should be noticed that the contributions from element *I* remain unchanged so again, as in the previous example, we can reuse the stiffness matrix and load vector.

3.3.1 Explanation of the notation

From equation (3.18) for element *II* we have

$$\text{Rep}\ (\Lambda_{II}) = [\{2\},\{3\},\{7\},\{(2,1/2),(5,1/2)\}] \tag{3.19}$$

We introduce some notation for sparse matrices. Assume that we represent the matrices in a column-wise scheme. Column 4 of Λ_{II} from equation (3.17) may be represented by:

$$\{(2,1/2),(5,1/2)\}. \tag{3.20}$$

What we have here is a simple set of $(\mathbb{N},\mathbb{R})$ pairs (R,V), where R represents the row number and V the value of the matrix entry. So the whole matrix has the form

$$\texttt{Rep}\ (\Lambda_{II}) \;=\; [\{2\},\{3\},\{7\},\{(2,1/2),(5,1/2)\}] \tag{3.21}$$

Notice that the first three (in this case) entries in the Rep (Λ_{II}) are of a different form from the last. We could write them down, and store them using the sparse matrix storage scheme, as

$$\{(2,1)\}$$

for example, but as the matrix entry 1 *will always* be a 1, storing it and performing its associated floating point operation is not necessary. The vast majority of entries in the connectivity mappings will be of this form, because the vast majority of the degrees of freedom will be "normal" degrees of freedom. So, for example, taking Λ_I in equation (3.17) and Rep (Λ_I) from equation (3.18) we can see that we have the classical storage scheme which is very efficient in terms of its use of storage and operations required.

Thus we would like any implementation to be able to take account of this requirement. This is a good candidate for an object orientated approach: we have the *base type* performing the classical accumulation, as done by the normal Boolean matrices, with some types derived from this implementing the desired p-constraint and h-constraint behaviour. However, the cost of this kind of implementation in terms of both storage and the number of dispatching operations that will be required could prove prohibitive. Also as the number of types is known and will not change, another approach is suggested: that of variant programming, [22].

3.3.2 Generalising to arbitrary h-refinements

Around corner singularities it can be shown that the traditional type of refinements here are not optimal, [60], and that the optimal refinement strategy involves both h and p-refinements. In addition the optimal refinement is not of subdividing the element edge in half but in the ratio of 3:20[1][60] — for examples of this see Figure 3.6 and Chapter 5. As before, the goal is to constrain the hanging

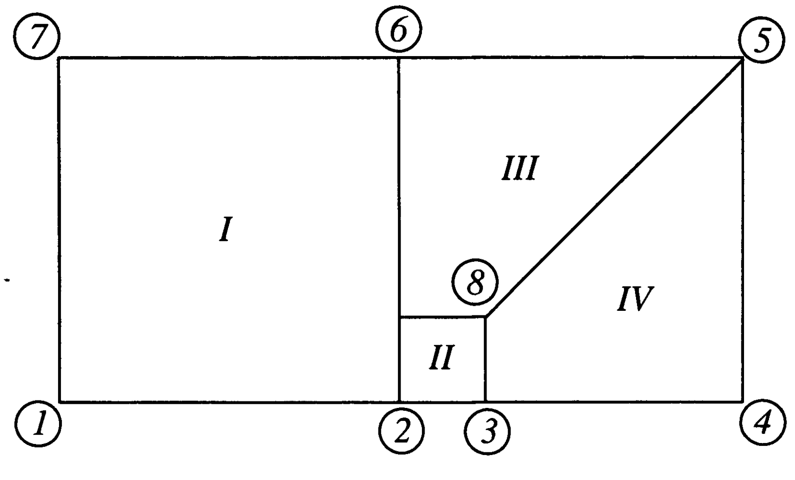


Figure 3.6: Optimally graded nonuniform h–refinement

node to maintain continuity across the interelement boundary. This time we distribute the global approximation vector with a weighted average, 3:20, giving the following connectivity mappings

$$\Lambda_I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \Lambda_{II} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & \frac{17}{20} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{3}{20} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad \Lambda_{III} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{17}{20} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{3}{20} & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} ; \quad (3.22)$$

[1]Actually the optimal is slightly larger than this

similarly the representation is

$$\begin{aligned}
\text{Rep }(\Lambda_I) &= [\{1\},\{2\},\{6\},\{7\}] \\
\text{Rep }(\Lambda_{II}) &= [\{2\},\{3\},\{8\},\{(2,17/20),(6,3/20)\}] \\
\text{Rep }(\Lambda_{III}) &= [\{(2,17/20),(6,3/20)\},\{8\},\{5\},\{6\}].
\end{aligned}$$

3.4 Generalising to *hp*-refinements

Consider now the *hp*-refined mesh in Figure 3.7. Here we have both *h*- and *p*-constraints.

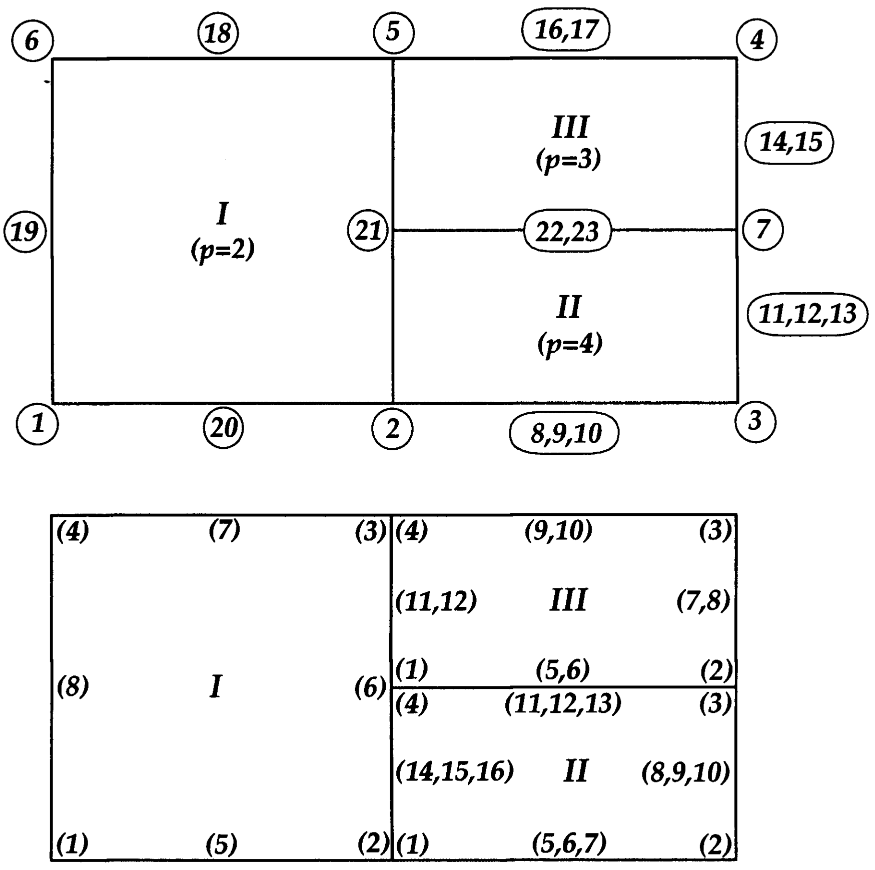


Figure 3.7: Generalised *hp*−refined domain

We have already seen how to constrain the degrees of freedom along the interelement boundary between elements II and III. So we must therefore determine the connectivity mapping for the interface between element I and elements II and III. First, we must identify which of the degrees of freedom are to be

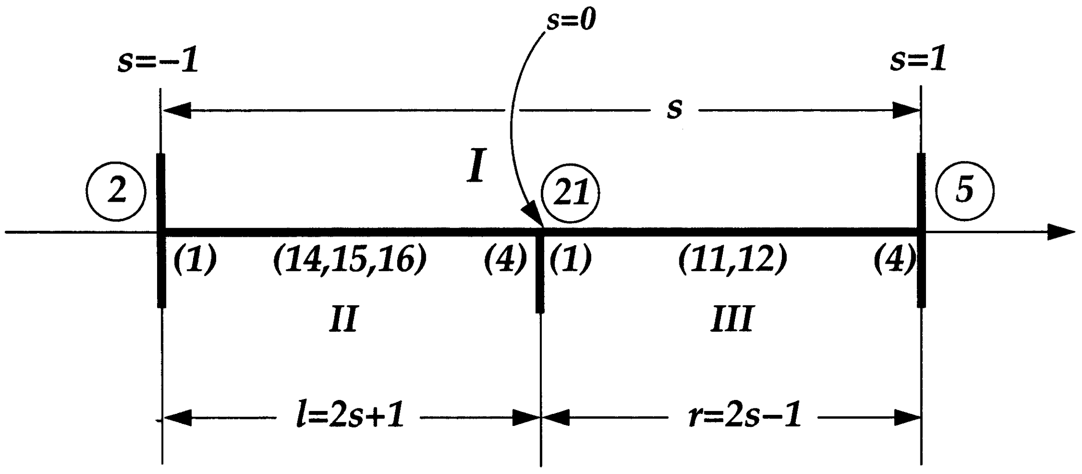


Figure 3.8: A edge refined traditionally in isolation

constrained: the lowest order element, element I, is a quadratic and this determines that the approximation along the interface must also be a quadratic. Consequently there are only three unconstrained degrees of freedom along the interface numbered 2, 5, (the linear degrees of freedom) and 21 (the quadratic degree of freedom). Construction of the connectivity mapping for element I is rather simple, it is just:

$$\Lambda = [\{1\}, \{2\}, \{5\}, \{6\}, \{20\}, \{21\}, \{18\}, \{19\}]$$

But the construction of the connectivity mappings for elements II and III is rather more complicated. The values of the constraints will depend on the basis functions that have been used and any scaling of them. The following procedure will work for any hierarchical basis, the only difference will be the choice of constraint constants that are to be used.

Suppose that the basis functions used are formed by taking tensor products of Lagrange type, for the linear degrees of freedom, and the integrated Legendre polynomials, for the higher degree polynomials, defined by:

$$\phi_0(s) = \frac{1}{2}(1-s) \qquad \phi_1(s) = \frac{1}{2}(1+s)$$

and

$$\phi_k(s) = \int_{-1}^{s} P_{k-1}(t)\, dt; \quad k = 2, 3, \ldots$$

where $P_k(t)$ is the k^{th} Legendre polynomial [60].

Figure 3.8 shows the interface in more detail. We first introduce a local coordinate $s \in [-1,1]$ along the edge. We will denote the values of the global degrees of freedom along the interface by x_2, x_5 and x_{21}. This gives the *global approximation* along the interface as:

$$\cdot\ u(s) = x_2\phi_0(s) + x_5\phi_1(s) + x_{21}\phi_2(s), \ s \in [-1,1]. \tag{3.23}$$

We will also denote the values of the local degrees of freedom for element II by $x_1^{II}, x_4^{II}, x_{14}^{II}, x_{15}^{II}$ and x_{16}^{II}; thus the local approximation along the interface for element II is given by

$$u^{II}(l) = x_1^{II}\phi_0(l) + x_4^{II}\phi_1(l) + x_{14}^{II}\phi_2(l) + x_{15}^{II}\phi_3(l) + x_{16}^{II}\phi_4(l), \ l \in [-1,1]. \tag{3.24}$$

where

$$l = 2s + 1$$

For a conforming approximation, we must have:

$$u(s) = u^{II}(l), \ s \in [-1,0] \tag{3.25}$$

where we have $l = 2s+1$. For this particular case we get the following solution:

$$\begin{cases} x_1^{II} &= x_2, \\ x_4^{II} &= \frac{1}{2}x_2 + \frac{1}{2}x_5 - \frac{1}{2}x_{21}, \\ x_{14}^{II} &= \frac{1}{4}x_{21}, \\ x_{15}^{II} &= 0, \\ x_{16}^{II} &= 0, \end{cases} \tag{3.26}$$

The connectivity mapping for element III is constructed in a similar manner, but instead there is a local coordinate r which is related to s by:

$$r = 2s - 1$$

The constraint constants for the degrees of freedom needed to satisfy (3.25) may easily be determined by the following procedure. Firstly, suppose a matrix $\boldsymbol{L}$ exists such that

$$\boldsymbol{\phi}(s) = \boldsymbol{L}\boldsymbol{\phi}(2s+1), \tag{3.27}$$

where

$$\boldsymbol{\phi}(s) = \begin{bmatrix} \phi_0(s) \\ \phi_1(s) \\ \vdots \\ \phi_p(s) \end{bmatrix} \tag{3.28}$$

for some p, the maximum polynomial degree of the basis functions. We may formulate condition (3.25) equally as:

$$\boldsymbol{x}^t\boldsymbol{\phi}(s) = \boldsymbol{x}^t_{II}\boldsymbol{\phi}(2s+1) \tag{3.29}$$

Substituting this into the above we get:

$$\begin{aligned} \boldsymbol{x}^t_{II}\boldsymbol{\phi}(2s+1) &= \boldsymbol{x}^t\boldsymbol{L}\boldsymbol{\phi}(2s+1) \\ \Rightarrow (\boldsymbol{x}^t_{II} - \boldsymbol{x}^t\boldsymbol{L})\boldsymbol{\phi}(2s+1) &= 0. \end{aligned} \tag{3.30}$$

However, because the basis functions are linearly independent we must have that:

$$\boldsymbol{x}_{II} = \boldsymbol{L}^t\boldsymbol{x} \tag{3.31}$$

Therefore once the matrix $\boldsymbol{L}$ is known, the constraint values can be simply read off using this result. Equally using the same procedure for element III from Figure 3.8 where $r = 2s - 1$,

$$\begin{aligned} \boldsymbol{x}^t_{III}\boldsymbol{\phi}(2s-1) &= \boldsymbol{x}^t\boldsymbol{R}\boldsymbol{\phi}(2s-1) \\ \Rightarrow \boldsymbol{x}_{III} &= \boldsymbol{R}^t\boldsymbol{x}, \end{aligned} \tag{3.32}$$

the matrix $\boldsymbol{R}$ can be constructed. Naturally, the matrices $\boldsymbol{L}$ and $\boldsymbol{R}$ are computed *a priori* and stored in the code.

Because of our use of hierarchical basis functions only two of these matrices are required: one for the left constraints and another for the right constraints. The matrices for $p = 4$ are show in equation (3.33):

$$\boldsymbol{L} = \begin{bmatrix} 1 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & -\frac{1}{2} & \frac{1}{4} & 0 & 0 \\ 0 & 0 & -\frac{3}{8} & \frac{1}{8} & 0 \\ 0 & \frac{1}{8} & \frac{3}{16} & -\frac{5}{16} & \frac{1}{16} \end{bmatrix}, \quad \boldsymbol{R} = \begin{bmatrix} \frac{1}{2} & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 1 & 0 & 0 & 0 \\ -\frac{1}{2} & 0 & \frac{1}{4} & 0 & 0 \\ 0 & 0 & \frac{3}{8} & \frac{1}{8} & 0 \\ \frac{1}{8} & 0 & \frac{3}{16} & \frac{5}{16} & \frac{1}{16} \end{bmatrix}. \tag{3.33}$$

Thus the solution for the conforming approximation can be assembled, after obtaining the leading submatrix from $\boldsymbol{L}$, as follows

$$\begin{bmatrix} x_1^{II} \\ x_4^{II} \\ x_{11}^{II} \\ x_{15}^{II} \\ x_{16}^{II} \end{bmatrix} = \begin{bmatrix} 1 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & -\frac{1}{2} & \frac{1}{4} & 0 & 0 \end{bmatrix}^t \begin{bmatrix} x_2 \\ x_5 \\ x_{21} \end{bmatrix} \tag{3.34}$$

which is the same as that found in equation (3.26). The constraint mapping for element III is obtained similarly — except the $\boldsymbol{R}$ matrix is used instead of the $\boldsymbol{L}$ matrix.

We have a series of steps that allow the construction of the constraint mapping for this type of mesh:

1. find the minimum order in the elements that lie along the interface;

2. all of the degrees of freedom whose order is greater than this are assigned to zero; and

3. constrain the remaining degrees of freedom using the process described previously.

For 2 to be true it is important that we are using a hierarchical basis. If we were not using a hierarchical basis but a Lagrange type basis, then every degree of freedom would have a higher degree polynomial associated with it and we would not be able to just assign particular ones to be zero to maintain continuity, we would have to compute all of the contributions along the edge at the lower degree. Again, after the construction of the constraint mappings is completed the standard assembly and distribution procedures can be followed giving a conforming approximation.

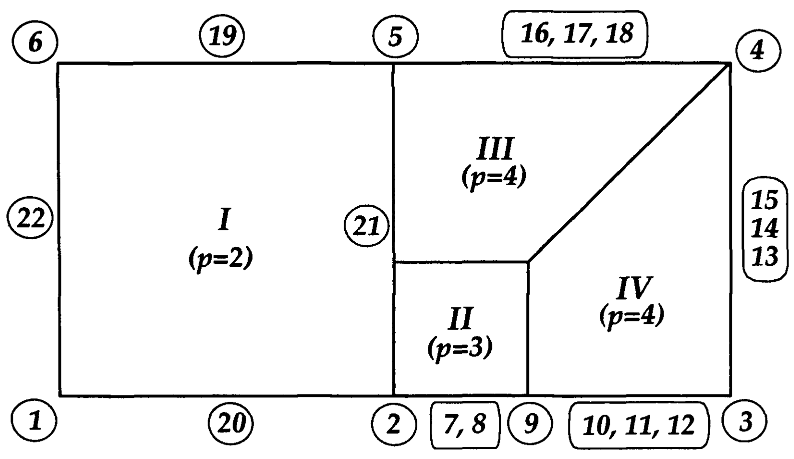
This may not always be the type of constraints that we will have to deal with. Earlier it was mentioned that around singularities a different type of refinement would be optimal; if we were to have that type of constraint here how would they be dealt with?

3.4.1 Generalising to completely arbitrary *hp*-meshes

Sometimes we may require a more general type of refinement where the edge of the element is not divided into the ratio of 0.5 or even 0.15, but a variable amount, which is denoted by $\sigma \in (0,1)$. Consider the mesh in Figure 3.9: this has a σ type refinement and has also been nonuniformly p-refined. The Figure 3.10 shows the interface between elements I and II and elements I and III, from Figure 3.9, in more detail.

Again the steps to construct a conforming approximation are similar to those for the traditional refinements, as described in the previous section; we have the usual global approximation:

$$u(s) = x_2\phi_0(s) + x_5\phi_1(s) + x_{21}\phi_2(s),\ s \in [-1,1] \tag{3.35}$$



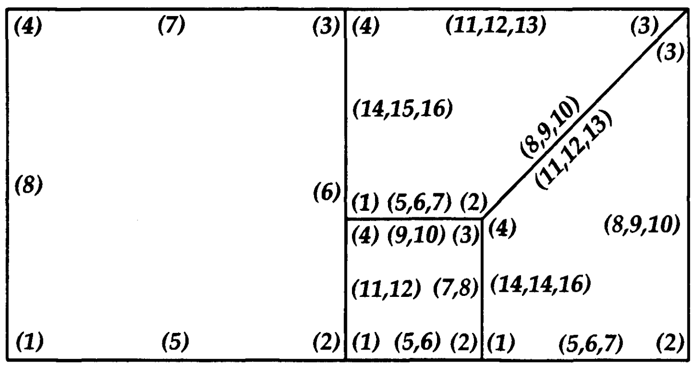


Figure 3.9: Nonuniform *hp*-refinement. Local and global degree of freedom numbers.

and the local approximation for element II is given by:

$$u^{II}(l) = x_1^{II}\phi_0(l) + x_4^{II}\phi_1(l) + x_{11}^{II}\phi_2(l) + x_{12}^{II}\phi_3(l),\ l \in [-1, 2\sigma - 1]. \tag{3.36}$$

This time the local coordinates are given by:

$$l = \frac{s + 1 - \sigma}{\sigma}. \tag{3.37}$$

Again they must be continuous across the interelement boundary, therefore:

$$\begin{aligned} u(s) &= u^{II}(l) \\ &= u^{II}\left(\frac{s+1-\sigma}{\sigma}\right),\ s \in (-1, 2\sigma - 1). \end{aligned} \tag{3.38}$$

Similarly for the right hand element, element III:

$$r = \frac{s - \sigma}{1 - \sigma},\ s \in (2\sigma - 1, 1). \tag{3.39}$$

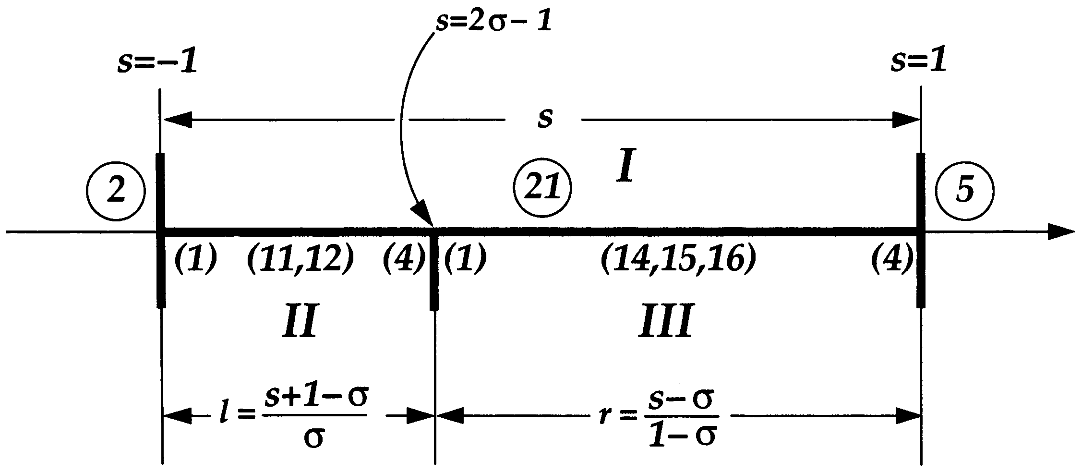


Figure 3.10: A general edge refinement in isolation

As in the previous section we end up with:

$$\boldsymbol{x}_{II} = \boldsymbol{L}^t(\sigma)\boldsymbol{x}. \tag{3.40}$$

Again this is a system of linear equations whose result is given in equation (3.41). The difference this time, however, is that the result is in the form of polynomials. Now we simply pull out the required block from the matrix, evaluate this block with the appropriate σ and apply the same procedure as before to get the connectivity mapping:

$$L(\sigma) = \begin{bmatrix} l_{0,0}(\sigma) & l_{0,1}(\sigma) & 0 & 0 & 0 & 0 \\ 0 & l_{1,1}(\sigma) & 0 & 0 & 0 & 0 \\ 0 & l_{2,1}(\sigma) & l_{2,2}(\sigma) & 0 & 0 & 0 \\ 0 & l_{3,1}(\sigma) & l_{3,2}(\sigma) & l_{3,3}(\sigma) & 0 & 0 \\ 0 & l_{4,1}(\sigma) & l_{4,2}(\sigma) & l_{4,3}(\sigma) & l_{4,4}(\sigma) & 0 \\ 0 & l_{5,1}(\sigma) & l_{5,2}(\sigma) & l_{5,3}(\sigma) & l_{5,4}(\sigma) & l_{5,5}(\sigma) \end{bmatrix}, \tag{3.41}$$

i	$l_{i,j}(\sigma)$	$r_{i,j}(\sigma)$
$i=0$	1	$1-\sigma$
	$1-\sigma$	0
$i=1$	0	σ
	σ	1
$i=2$	0	$-2\sigma+2\sigma^2$
	$-2\sigma+2\sigma^2$	0
	σ^2	$1-2\sigma+\sigma^2$
$i=3$	0	$2\sigma-6\sigma^2+4\sigma^3$
	$2\sigma-6\sigma^2+4\sigma^3$	0
	$-3\sigma^2+3\sigma^3$	$3\sigma-6\sigma^2+3\sigma^3$
	σ^3	$1-3\sigma+3\sigma^2-1\sigma^3$
$i=4$	0	$-2\sigma+12\sigma^2-20\sigma^3+10\sigma^4$
	$-2\sigma+12\sigma^2-20\sigma^3+10\sigma^4$	0
	$6\sigma^2-15\sigma^3+9\sigma^4$	$-3\sigma+15\sigma^2-21\sigma^3+9\sigma^4$
	$-5\sigma^3+5\sigma^4$	$5\sigma-15\sigma^2+15\sigma^3-5\sigma^4$
	σ^4	$1-4\sigma+6\sigma^2-4\sigma^3+\sigma^4$
$i=5$	0	$2\sigma-20\sigma^2+60\sigma^3-70\sigma^4+28\sigma^5$
	$2\sigma-20\sigma^2+60\sigma^3-70\sigma^4+28\sigma^5$	0
	$-10\sigma^2+45\sigma^3-63\sigma^4+28\sigma^5$	$3\sigma-27\sigma^2+73\sigma^3-77\sigma^4+28\sigma^5$
	$15\sigma^3-35\sigma^4+20\sigma^5$	$-5\sigma+35\sigma^2-75\sigma^3+65\sigma^4-20\sigma^5$
	$-7\sigma^4+7\sigma^5$	$7\sigma-28\sigma^2+42\sigma^3-28\sigma^4+7\sigma^5$
	σ^5	$1-5\sigma+10\sigma^2-10\sigma^3+5\sigma^4-\sigma^5$

Table 3.1: Components, $l_{i,j}(\sigma)$ and $r_{i,j}(\sigma)$, of the matrices $\boldsymbol{L}(\sigma)$ and $\boldsymbol{R}(\sigma)$

$$R(\sigma) = \begin{bmatrix} r_{0,0}(\sigma) & r_{0,1}(\sigma) & 0 & 0 & 0 & 0 \\ r_{1,0}(\sigma) & 0 & 0 & 0 & 0 & 0 \\ r_{2,0}(\sigma) & 0 & r_{2,2}(\sigma) & 0 & 0 & 0 \\ r_{3,0}(\sigma) & 0 & r_{3,2}(\sigma) & r_{3,3}(\sigma) & 0 & 0 \\ r_{4,0}(\sigma) & 0 & r_{4,2}(\sigma) & r_{4,3}(\sigma) & r_{4,4}(\sigma) & 0 \\ r_{5,0}(\sigma) & 0 & r_{5,2}(\sigma) & r_{5,3}(\sigma) & r_{5,4}(\sigma) & r_{5,5}(\sigma) \end{bmatrix}. \quad (3.42)$$

The values for $l_{ij}(\sigma)$ and $r_{ij}(\sigma)$ are given in Table (3.1). The approximation for element II can now be assembled, again by obtaining the leading submatrix from $\boldsymbol{L}$, as follows:

$$\begin{bmatrix} x_1^{II} \\ x_4^{II} \\ x_{11}^{II} \\ x_{12}^{II} \end{bmatrix} = \begin{bmatrix} l_{00}(\sigma) & l_{01}(\sigma) & 0 & 0 \\ 0 & l_{11}(\sigma) & 0 & 0 \\ 0 & l_{21}(\sigma) & l_{22}(\sigma) & 0 \end{bmatrix}^t \begin{bmatrix} x_2 \\ x_5 \\ x_{21} \end{bmatrix}. \tag{3.43}$$

This gives

$$\begin{cases} x_1^{II} &= x_2, \\ x_4^{II} &= (1-\sigma)x_2 + \sigma x_5 - 2\sigma(1-\sigma)x_{21}, \\ x_{11}^{II} &= \sigma^2 x_{21}, \\ x_{12}^{II} &= 0. \end{cases} \tag{3.44}$$

Note that for $\sigma = 0.5$ we will get the same results as in the previous section.

So again we have a sequence of steps that will result in the connectivity mapping for a conforming mesh:

1. find the minimum order in the elements that lie along the interface;

2. all of the degrees of freedom whose order is greater than this are assigned to zero;

3. determine the value of the grading factor σ; and

4. the remaining local degrees of freedom are constrained in terms of the global degrees of freedom using the coefficients read off from the correctly evaluated matrix $\boldsymbol{L}(\sigma)$ or $\boldsymbol{R}(\sigma)$ as appropriate.

3.5 Generalising to multilevel *hp*-refinements

Suppose now that the mesh in Figure 3.7 has its element *II* *h*-refined to give us Figure 3.11. The interface between elements *III* and elements *V* and *VII* has been dealt with in the previous section. Now the main problem is to construct the

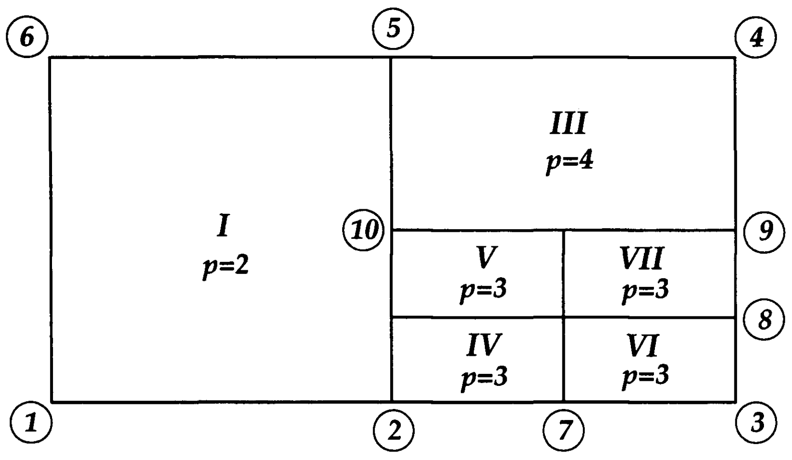


Figure 3.11: Generalised $hp-$refined domain

connectivity mapping for the interface between element I and elements III, V and IV. Figure 3.12 shows the interface in more detail. We could develop a procedure

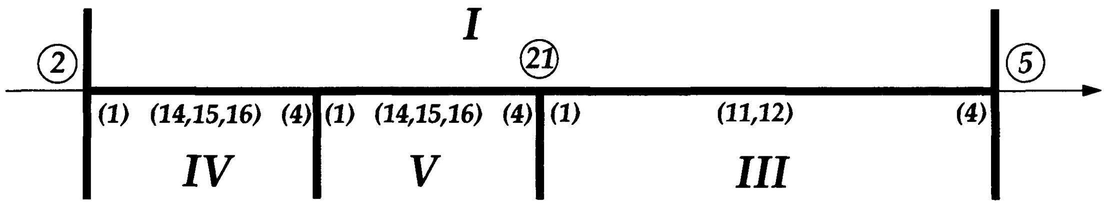


Figure 3.12: Isolated irregular edge from Figure 3.11

similar to that described for the single level case, but this is not very practical as it would require the storage of many matrices similar to $\boldsymbol{L}$ and $\boldsymbol{R}$ developed previously for the many possible configurations that may exist. Another way of constraining these meshes is required. Figure 3.13 shows how the element IV and V were derived from h-refining element II. But of course it does not exist any more and so it is described as being *imaginary*. We can make use of these

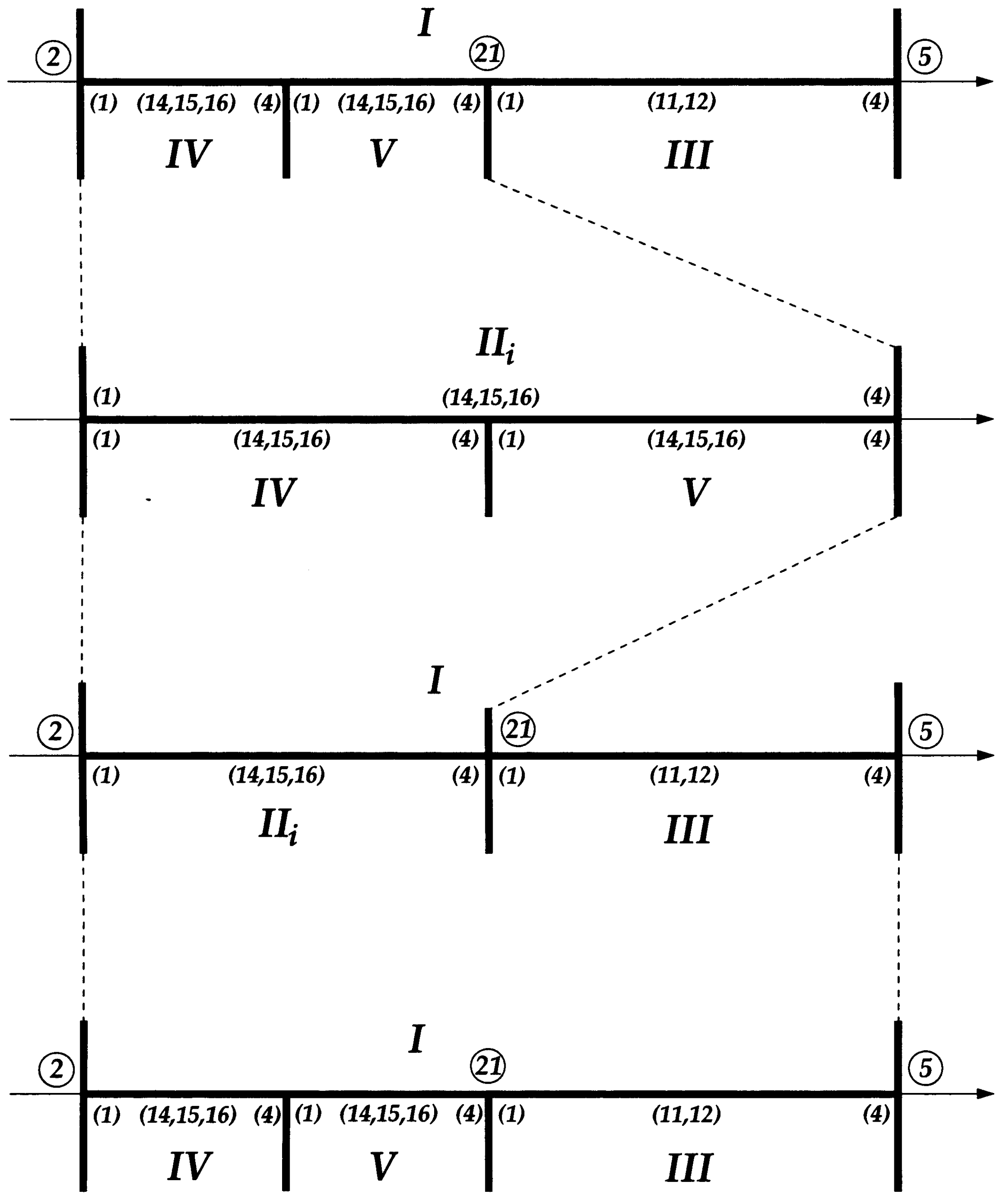


Figure 3.13: Edge showing use of the imaginary element in constructing the connectivity mapping

imaginary degrees of freedom; we had previously that:

$$\begin{bmatrix} x_1^{II} \\ x_4^{II} \\ x_{14}^{II} \\ x_{15}^{II} \\ x_{16}^{II} \end{bmatrix} = \begin{bmatrix} 1 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & -\frac{1}{2} & \frac{1}{4} & 0 & 0 \end{bmatrix}^t \begin{bmatrix} x_2 \\ x_5 \\ x_{21} \end{bmatrix} \tag{3.45}$$

The same can be used to get the constraints for elements IV and V in terms of the imaginary element II, thus:

$$\begin{bmatrix} x_1^{IV} \\ x_4^{IV} \\ x_{14}^{IV} \\ x_{15}^{IV} \\ x_{16}^{IV} \end{bmatrix} = \begin{bmatrix} 1 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & -\frac{1}{2} & \frac{1}{4} & 0 & 0 \\ 0 & 0 & -\frac{3}{8} & \frac{1}{8} & 0 \\ 0 & \frac{1}{8} & \frac{3}{16} & -\frac{5}{16} & \frac{1}{16} \end{bmatrix}^t \begin{bmatrix} x_1^{II} \\ x_4^{II} \\ x_{14}^{II} \\ x_{15}^{II} \\ x_{16}^{II} \end{bmatrix} \tag{3.46}$$

Then equations (3.46) are substituted into equations (3.45) to eliminate the imaginary degrees of freedom, to get:

$$\begin{cases} x_1^{IV} &= x_2, \\ x_4^{IV} &= \frac{3}{4}x_2 + \frac{1}{4}x_5 - \frac{3}{8}x_{21}, \\ x_{14}^{IV} &= \frac{1}{16}x_{21}, \\ x_{15}^{IV} &= 0, \\ x_{16}^{IV} &= 0. \end{cases} \tag{3.47}$$

Similarly for the element V we get

$$\begin{cases} x_1^{IV} &= \frac{3}{4}x_2 + \frac{1}{4}x_5 - \frac{3}{8}x_{21}, \\ x_4^{IV} &= \frac{1}{2}x_2 + \frac{1}{2}x_5 - \frac{1}{2}x_{21}, \\ x_{14}^{IV} &= \frac{1}{16}x_{21}, \\ x_{15}^{IV} &= 0, \\ x_{16}^{IV} &= 0. \end{cases} \tag{3.48}$$

Thus it is now just the simple case of constructing the full connectivity mappings for all the elements in the mesh. The process described here can be applied to any number of levels of refinement because all that is required is the product of the single level constraint constants.

3.5.1 Generalising to completely arbitrary multilevel *hp*-meshes

As described in the previous section we can construct the multilevel constraint mappings. The ratio used for a partition need not be 0.5 and so it is now possible to construct the constraints for the mesh in Figure 3.14. Indeed more general

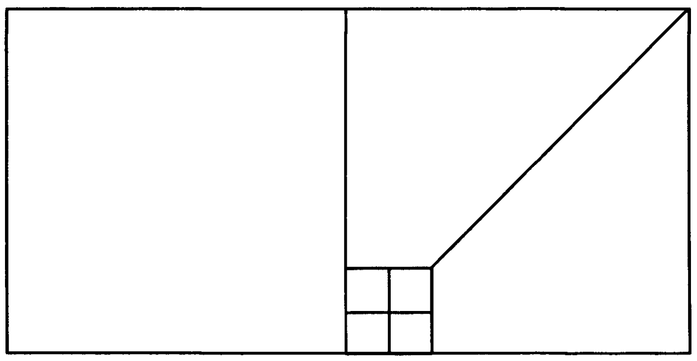


Figure 3.14: The 0.15 type refinement followed by a 0.5

meshes are possible: here we have a 0.15 type refinement followed by a traditional type refinement. Figure 3.15 shows the interface in more detail. This again re-

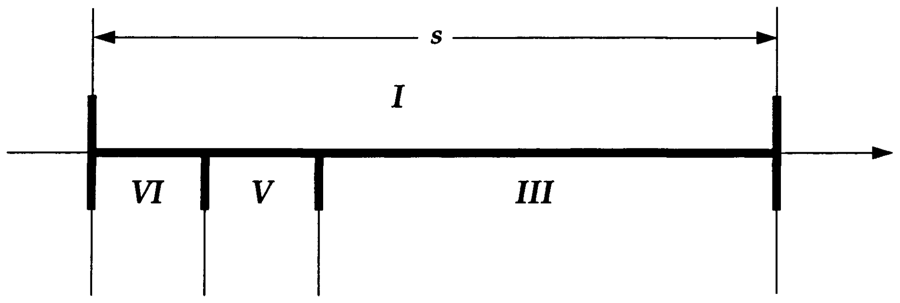


Figure 3.15: The inter-element interface from Figure 3.14

quires the procedure described previously using imaginary elements but this time we evaluate the constraint constants matrices at each level with the appropriate value for σ. So for the interface between element I and element III and the imaginary element II we would evaluate $\boldsymbol{L}(\sigma)$ with $\sigma = 0.15$ giving:

$$\begin{bmatrix} x_1^{II} \\ x_4^{II} \\ x_{14}^{II} \\ x_{15}^{II} \\ x_{16}^{II} \end{bmatrix} = \begin{bmatrix} 1 & \frac{17}{20} & 0 & 0 & 0 \\ 0 & \frac{3}{20} & 0 & 0 & 0 \\ 0 & -\frac{51}{200} & \frac{9}{400} & 0 & 0 \end{bmatrix}^t \begin{bmatrix} x_2 \\ x_5 \\ x_{21} \end{bmatrix}. \tag{3.49}$$

Then for the lower level, we would evaluate with $\sigma = 0.5$ giving

$$\begin{bmatrix} x_1^{VI} \\ x_4^{VI} \\ x_{14}^{VI} \\ x_{15}^{VI} \\ x_{16}^{VI} \end{bmatrix} = \begin{bmatrix} 1 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & -\frac{1}{2} & \frac{1}{4} & 0 & 0 \\ 0 & 0 & -\frac{3}{8} & \frac{1}{8} & 0 \\ 0 & \frac{1}{8} & \frac{3}{16} & -\frac{5}{16} & \frac{1}{16} \end{bmatrix}^t \begin{bmatrix} x_1^{II} \\ x_4^{II} \\ x_{14}^{II} \\ x_{15}^{II} \\ x_{16}^{II} \end{bmatrix}. \tag{3.50}$$

The elimination procedure is the same as for the previous case giving the required result of

$$\begin{bmatrix} x_1^{VI} \\ x_4^{VI} \\ x_{14}^{VI} \\ x_{15}^{VI} \\ x_{16}^{VI} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \frac{37}{40} & \frac{3}{40} & -\frac{111}{800} \\ 0 & 0 & \frac{9}{1600} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_2 \\ x_5 \\ x_{21} \end{bmatrix}. \tag{3.51}$$

Thus we have the approximation

$$\begin{cases} x_1^{VI} &= x_2, \\ x_4^{VI} &= \frac{37}{40}x_2 + \frac{3}{40}x_5 - \frac{111}{800}x_{21}, \\ x_{14}^{VI} &= \frac{9}{1600}x_{21}, \\ x_{15}^{VI} &= 0, \\ x_{16}^{VI} &= 0. \end{cases}$$

Again, as with the all the previous examples, as soon as the connectivity mappings are obtained the standard assembly and distribution procedures can be used, equations (3.11) and (3.12).

As can be seen the simple evaluation of the constraint constant matrices with the appropriate grading factor, σ, for the single constraint, followed by the composition of these, leads to the construction of the connectivity mapping for these kinds of refinements also. This process extends to any number of levels of refinement, possibly with each level having a different grading factor. As with all previous refinements once the connectivity mapping is constructed the normal assembly and distribution procedures can be used.

3.6 Conclusions

If a finer mesh is introduced over part of the domain it has been shown in this chapter that it is easy to match the solutions across the boundary between the two densities of mesh. If the degree of the basis functions is varied across the domain, again it has been shown that there is a simple strategy to match the solutions. What is surprising is that if both mesh density and basis function degree are varied, the matching can be derived by just combing the two strategies.

There are further advantages to be gained when introducing a different density of mesh — rather than just subdividing an element by half, the division point along an edge can be σ where $\sigma \in (0, 1)$; σ is not just restricted to be $\frac{1}{2}$. The strategy for dealing with non-uniform meshes was just applied with some minor adjustments to deal with the non-uniform division of boundaries.

The solution strategy does not build the stiffness matrix directly, but rather calculates the connectivity mapping: the stiffness matrix is calculated from that. Thus the problem of deriving the the stiffness matrix is reduced to constructing the connectivity mapping. A set of systematic techniques for deriving the connectivity mapping have been derived from consideration of each of the simple possibilities. We have show that there is a way of providing a simple, efficient assembly of the connectivity mappings for the *hp*-refinement case. We have also shown that the partition of part of the domain by a mesh can be generalised to a variable grading factor and that any number of levels of refinement is simple to construct.

Chapter 4

The matrix library and solvers

There are a variety of matrices used throughout any finite element code which usually consist of either dense or sparse matrices. The dense matrices are usually used for assembling the element contributions. Dense matrices are also required when we do static condensation at any level (element, subdomain or domain) because this causes *fill in*. At all other times we will use a sparse matrix, due to the large number of zero entries in the matrix, perhaps more that 99% in some cases.

4.1 Introduction

The matrix library is often an integral component of any finite element implementation. Often, in non-commercial implementations the matrix library will just deal with simple two dimensional arrays; however, this is very often inappropriate, especially when large problems are to be attempted. The finite element method generates very large sparse matrices, so obviously this array based approach is unsuitable. There is no single standard approach for the management of sparse matrices — many methods exist for their implementation [2, 16, 66], including some object orientated matrix libraries [35, 50, 55].

There are a number of matrices used throughout most finite element programmes. In addition to the dense, sparse matrices already mentioned, there are triangular matrices and there is also the possibility of distributing matrices across different computers. Each of these matrix representations, with the exception perhaps of the triangular matrix, should be interchangeable throughout the code without any other changes being made. Because of this need, the object orientated approach is used throughout the designing and implementation of the main part of the finite element code.

Several matrix libraries already exist that could have been used, such as Lapack [15, 33, 34] and some of the NAG routines [2], for example. Note that in the interests of developing a completely independent code it was decided not to use these. Reasons for this include the fact that users may not have access to the NAG library, or some other matrix library, so it was decided to create a matrix library specifically for this finite element implementation. A further problem with some of the existing libraries is their inflexibility.

4.2 Matrix implementation

Figure 4.1 shows the hierarchy of the matrix types, where we use the convention that the arrows point towards the type from which it is *derived.* Those types in the dashed boxes show the *abstract* types, the solid lined boxes show the *concrete* types.

The base type in the hierarchy, *Matrix*, must be carefully designed to allow all of the necessary operations that are likely to be required by any finite element implementation. It turns out that this is a relatively small and simple set of operations, just the access operations and a matrix vector product (this is used in the iterative solvers). Although some of the usual mathematical operations are also there, $+, \times$, multiplication by a scalar, $\ldots$, but are only there for completeness. They are abstract operations so that each concrete type has to implement its

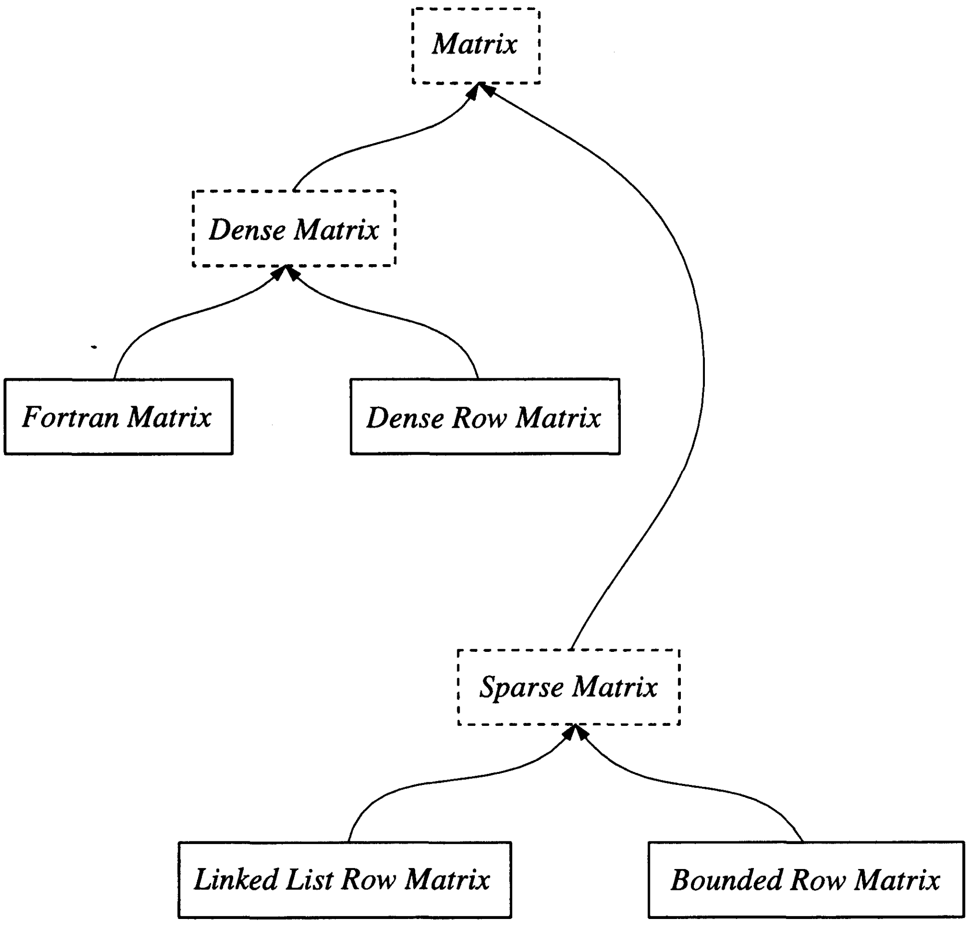


Figure 4.1: The matrix hierarchy

own optimised version. Other operations exist for construction, assignment and destruction, but these are private operations that are not seen by the user.

4.3 Sparse matrix

A sparse matrix is a matrix which has a large percentage of its entries with zero value; this percentage is usually greater than about 80%. In the matrices generated by the finite element method, many generated are over 95% sparse. However, to simplify the storage of the matrix and speed up the operations on the matrix (the most important of which, for iterative solvers, is matrix vector product), we will make use of the knowledge that each row of the matrix has at least one entry. Therefore, we can store each row as a sparse vector whereby we only have to store the position in the vector, which will be the column in the matrix, and the value at the position. This again is an abstract type. It does not add any more functionality, it is just the base type for the sparse matrix class.

4.3.1 Linked row matrix

As previously mentioned we will have at least one entry in every row of the matrix, so our main task is how to store the entries of the row. The simplest implementation is a linked list. Figure 4.2 shows the lists that make up the matrix, with each of the elements in the list having the following three pieces of information:

1. the column number; and
2. the entry value; and
3. the next entry *pointer*.

Using a linked list makes it simple to add entries to the matrix during its construction. Although the order of the entries is not important, we will use an ordered

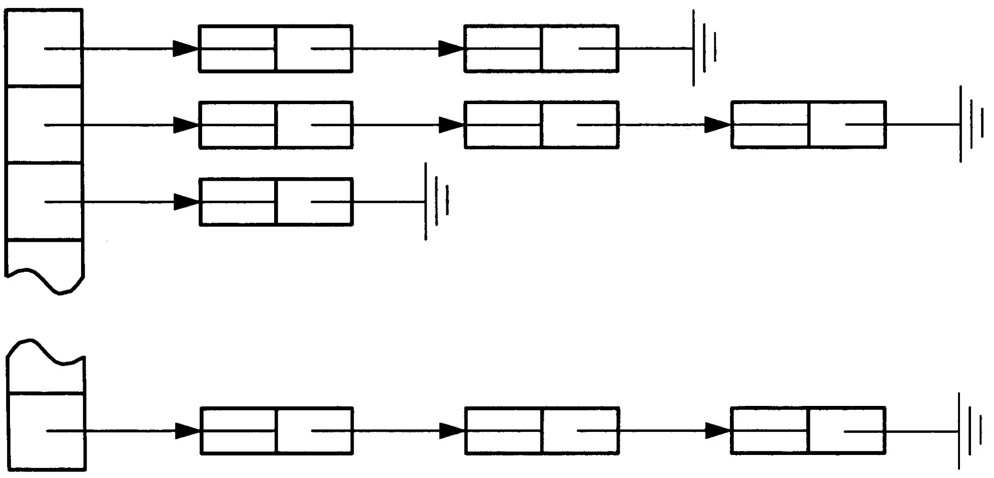


Figure 4.2: Storage representation of the linked row matrix

list, ordered by column number, so as to simplify some of the other operations.

4.3.2 Compressed row matrix

This is similar to the previous matrix, in that each row contains an entry, but this time the representation of the rows is different. Rather than have them as

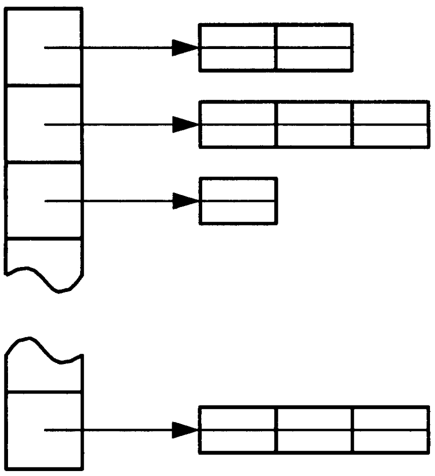


Figure 4.3: Storage representation of the bounded row matrix

linked lists we store them in an array, as shown in Figure 4.3, but each entry is basically the same:

1. the column number; and

2. the entry value.

This representation speeds up the matrix vector product a great deal, because we can bring most, if not all, of the information about the row into the CPU cache rather than a single entry at a time as in the linked list case. Another reason for storing the matrix as rows is to enable a simple parallel matrix–vector product. Each processor can be given a set of the rows from the matrix and the result vector, perform the inner–products and update the result *without* any worry about overwrites.

4.4 Dense matrix

A dense matrix, as suggested by its name, stores all of the entries of the matrix. This is almost always represented as a two dimensional array or an array of arrays. In this implementation both kinds of matrix are employed. We can define many more operations for this particular kind of matrix, ones that can be used for dense type matrices such as factorisation, inversion, Gaussian elimination, etc. The reason why these were not also defined for sparse matrices is, as already mentioned, that when we perform these operations the matrix suffers with fill in so the matrix becomes dense. If we still stored them in the sparse format then we would be using more resources in either the storage or the operations, or both of these. There have been two types implemented, as outlined below.

4.4.1 Fortran matrix

This is stored as a two dimensional array, with some additional compiler directives (pragmas, in Ada), instructing the compiler to store the matrix in column major order, the same as Fortran stores its matrices. The reason for this is that if a user has some special Fortran code written, it can then be imported simply and safely into the programme.

4.4.2 Dense row matrix

This matrix is represented as an array of arrays. There are several reasons for this:

- it should speed up the solving process, as only a single row and the vector need be brought into the cache at any one time; and
- it is conceivable to store some, or all, of the rows on the backing store if the matrix is exceptionally large; and
- there may be problems on some systems with having large contiguous blocks of memory.

4.5 Other matrix types

Using the object orientated approach for the matrix library could enable us to add other matrix implementations, especially different sparse representations, to the matrix class hierarchy such as those from [50]. This would enable the use of a representation that enabled the simple implementation of different preconditioners such as, incomplete factorisation or ILU [16, 50], while still keeping the code for the construction of stiffness matrix and the solvers the same. Any of these new matrix or preconditioner types could be simply added to the existing code.

4.6 Problems with the matrices generated by the finite element method

There are several problems with the matrices that are generated by the finite element method, but a major one of these is shear size, sometimes many $100,000$'s square. It is obvious from this that they cannot be stored as a two dimensional array, and so some sparse storage scheme must be employed. Many such schemes exist and some involve knowing the number of matrix entries or the bandwidth *a priori.* These are not useful in a fully adaptive finite element code, where these are not known and in fact would change after each adaptive refinement operation. Therefore it is desirable to have fully a adaptive matrix library, one where we need not know the number of entries or the maximum bandwidth of the matrix until we need to construct the matrix.

4.6.1 Sensitivity to *rounding errors*, the condition number

Those matrices that are generated can be extremely sensitive to small perturbations that are inevitably introduced during the solving process, (errors also occur during the assembling of the system, because the integral techniques used only approximate the integrals of the given functions). These small perturbations, *rounding discrepancies* or *rounding errors*, are introduced because we can use only a finite precision arithmetic. Many books exist that explain the problems with rounding errors and how to understand and deal with them, see [29, 44, 64]

Further insight into rounding errors can be gained by considering the system $K\xi = b$, then using $b + \delta b$ for the right hand side. We would expect, after solving the system, the solution to be $\xi + \delta\xi$, where δb is small relative to b and $\delta\xi$ is small relative to ξ. Is there any useful or convenient measure of this sensitivity? The *condition number* usually written as κ, is defined to be:

$$\kappa = ||K|| \cdot ||K^{-1}||$$

in some matrix norm. From this we get the standard relationship:

$$\frac{||\delta\xi||}{||\xi||} \leq \kappa \frac{||\delta b||}{||b||}$$

It is usual to use the 2-norm, and it can be shown that, for a symmetric matrix,

$$\kappa_2 = ||K||_2 \cdot ||K^{-1}||_2 = \frac{\lambda_{max}}{\lambda_{min}}$$

where λ_{max} and λ_{min} are the maximum and minimum eigenvalues respectively. This is, in turn, dependent on h, the mesh size, and p, the polynomial degree of the elements. This number affects all aspects of solving, both direct and iterative solvers. For iterative solvers the condition number determines the number of iterations a solution will take to converge to the tolerance required by the user, if indeed it converges at all.

4.7 Solving the system

After the system of linear equations has been assembled, it must be solved. There are many ways of doing this:

- direct solvers, which may be one of the following:
  - find the inverse, using elementary row operations;
  - Gaussian elimination;
  - one of the many factorisation techniques;
  - static condensation (this is not really a complete solver but helps solvers by giving smaller condition number and a smaller matrix);
- iterative solvers, four of which are:
  - Gauss Seidel; and
  - conjugate gradient, with or without preconditioning; and
  - multilevel; and
  - generalised minimum residual.

Both of the above methods, direct and iterative, have their own advantages and disadvantages. Obviously finding the inverse directly is impossible for these kind of problems, but we can make use of elementary row operations. The use of direct solvers is generally used when there is an asymmetric system or when the matrix is very ill conditioned, though now there are some powerful iterative methods for solving such asymmetric systems.

The solving technique that we use is a hybrid version containing both a partial static condensation *and* an iterative solver for the reduced system. This way we help to reduce the condition number of the matrix, by the partial static condensation, that is passed to the iterative solver, enabling a smaller number of iterations for convergence.

4.7.1 Direct solvers

Several different types of direct solvers are frequently used for solving matrix equations including those of Gaussian elimination and various factorisation techniques. One factorisation technique that is frequently used is the LU factorisation. In this we factorise the matrix into lower and upper triangular matrices, such that

$$LU = A$$

holds. So now, rather that having to solve $x = A^{-1}b$, we have to first solve $y = L^{-1}b$, followed by $x = U^{-1}y$. The solutions to each of these triangular matrix equations is easily computed. Another common direct solving technique, as already mentioned, is that of Gaussian elimination. The solution is achieved after a series of row operations usually with some *pivoting* strategy [29, 40]. Using a pivoting strategy can help to reduce the effect of the high condition number and stabilise the method from the effects of rounding errors. Analysis of these and many other direct solving schemes can be found in [29, 40, 44]. Both of these methods are related and have, in the worst case, $\mathcal{O}(n^3)$ floating point operation count, so clearly with very large n these methods can prove prohibitive, although some direct solvers for sparse systems exist [16, 37] and in these methods that sue fewer than $\mathcal{O}(n^3)$ are described. Factorisation methods do have other advantages though in that we only need to factorise the system once and we can solve for any number of *right hand sides.*

4.7.2 Static condensation, Schur complement

Consider the refinement of element in the subdomain in Figure 4.4. Having

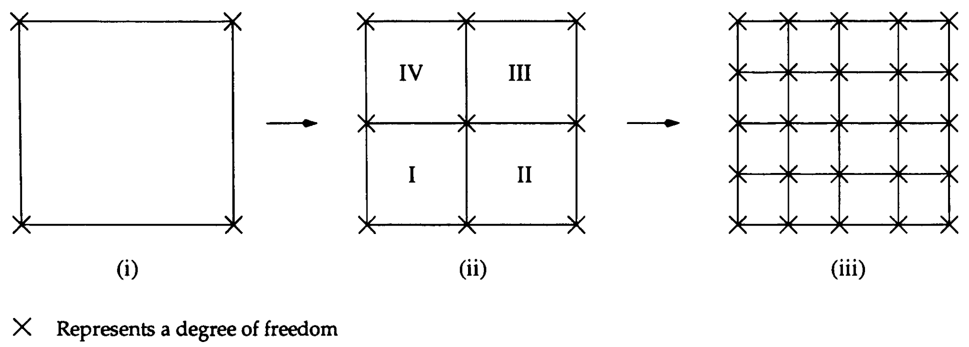


Figure 4.4: Domain refinement

assembled the stiffness matrix, we can improve its condition number and give ourselves a smaller system by eliminating the interior degrees of freedom. By this we mean a decoupling of the boundary and interior degrees of freedom. This

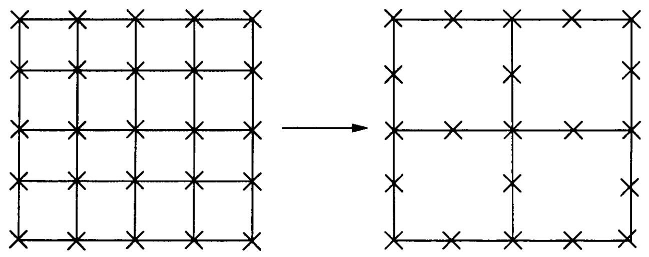


Figure 4.5: Elimination of interior degrees of freedom

process can also take place at the subdomain level, giving a further improvement in the condition number. We can also eliminate further interior degrees of freedom, leaving just the degrees of freedom associated with the boundary of the domain, as in Figure 4.6.

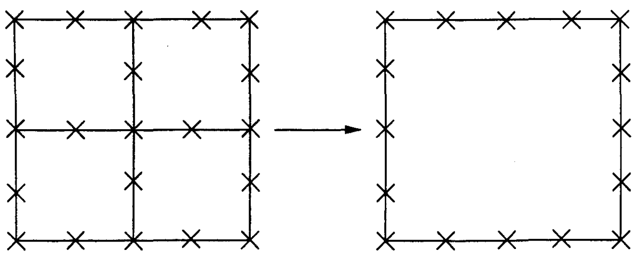


Figure 4.6: Further elimination of interior degrees of freedom

Mathematically, we have the following process. The uncondensed global system $K\xi = G$ obtained effectively, has this block matrix form:

$$K = \begin{bmatrix} K_{BB} & K_{BI} \\ K_{IB} & K_{II} \end{bmatrix}, \boldsymbol{G} = \begin{bmatrix} \boldsymbol{G}_B \\ \boldsymbol{G}_I \end{bmatrix}, \boldsymbol{\xi} = \begin{bmatrix} \boldsymbol{\xi}_B \\ \boldsymbol{\xi}_I \end{bmatrix} \tag{4.1}$$

where K_{BB} is the interaction of boundary degrees of freedom with boundary degrees of freedom, K_{BI} and K_{IB} is the interaction of boundary degrees of freedom with interior degrees of freedom and K_{II} is the interaction of interior degrees of freedom with interior degrees of freedom. So effectively we have:

$$K_{BB}\boldsymbol{\xi}_B + K_{BI}\boldsymbol{\xi}_I = \boldsymbol{G}_B \tag{4.2}$$

$$K_{IB}\boldsymbol{\xi}_B + K_{II}\boldsymbol{\xi}_I = \boldsymbol{G}_I \tag{4.3}$$

Rearranging equation (4.3) we get

$$\boldsymbol{\xi}_I = K_{II}^{-1}\boldsymbol{G}_I - K_{II}^{-1}K_{IB}\boldsymbol{\xi}_B \tag{4.4}$$

Substituting this into equation (4.2) we get

$$K_{BB}\boldsymbol{\xi}_B = \boldsymbol{G}_B - K_{BI}(K_{II}^{-1}\boldsymbol{G}_I - K_{II}^{-1}K_{IB}\boldsymbol{\xi}_B) \tag{4.5}$$

$$\Rightarrow (K_{BB} - K_{BI}K_{II}^{-1}K_{IB})\boldsymbol{\xi}_B = \boldsymbol{G}_B - K_{BI}K_{II}^{-1}\boldsymbol{G}_I. \tag{4.6}$$

$$\tag{4.7}$$

Doing this enables the boundary problem to be solved first (the Schur complement problem). Let $S = K_{BB} - K_{BI}K_{II}^{-1}K_{IB}$, be the Schur Complement, and $\boldsymbol{G}_S =$

$\boldsymbol{G}_B - K_{BI}K_{II}^{-1}\boldsymbol{G}_I$, so we now have to solve

$$S\boldsymbol{\xi}_B = \boldsymbol{G}_S. \tag{4.8}$$

This is a better conditioned system so should not take as many iterations to solve. It is also a smaller system so each iteration will not take as long. To calculate the interior solution, $\boldsymbol{\xi}_I$, we just do a *back solve*, substituting the boundary solution, $\boldsymbol{\xi}_B$, obtained from solving equation (4.8), into equation (4.4):

$$\begin{aligned}\boldsymbol{\xi}_I &= K_{II}^{-1}\boldsymbol{G}_I - K_{II}^{-1}K_{IB}\boldsymbol{\xi}_B \\ &= K_{II}^{-1}(\boldsymbol{G}_I - K_{IB}\boldsymbol{\xi}_B).\end{aligned}$$

This process can be done before we accumulate the local stiffness matrices into the more global system. This makes the process simpler because the system is smaller, although we have to do this four times, but overall we do save a considerable amount of time. Because of this, the order of static condensation and subsequent accumulation of the local systems into the global system does not matter. Therefore, this process can be done in parallel, so what we have in effect is a parallel direct solve. As we have gained so much from this approach, why do we not eliminate all of the interior nodes in the mesh as this would result in a much better conditioned and smaller system that would be easier to solve? The problem is, that doing this static condensation is an expensive process as we have many elementary row operations on a large matrix. We are effectively calculating the inverse of the sub matrix K_{II}. Therefore there must be some optimal amount of condensation that can be done, where doing more elimination will result in a system with a lower solve time but whose assembly time is higher, due to the excessive condensation, thereby making the overall run time greater.

4.7.3 Iterative solvers

In iterative solvers we are *not* interested in the inverse itself but only in the action of the inverse. They usually take the form

$$\boldsymbol{\xi}_{n+1} = T_n \boldsymbol{\xi}_n + \boldsymbol{c}_n, \tag{4.9}$$

for some ξ_0, often $\xi_0 = 0$, where T_n and c_n are dependent on the chosen method. Iteration is continued until some specified tolerance is reached. This is usually specified as

$$\frac{||A\boldsymbol{\xi}_n - \boldsymbol{b}||}{||\boldsymbol{b}||} < \epsilon. \tag{4.10}$$

Direct solvers usually give the exact solution (to the accuracy of the underlying floating point number, this will also depend on the condition number of the matrix). However, we frequently do not require such a high accuracy (something in the range 0.1% to 1% is usually sufficient) and so again by using an iterative scheme we have reduced the work required to solve the equations.

Though direct solvers can produce very accurate results they cannot usually take advantage of the sparsity that exists in the matrix obtained from the finite element method. Both Gaussian elimination and factorisation produce *fill in*, which results in a more populated inverse matrix — this can be a problem with the very large matrices that are the result of calculating of using direct solvers. It should be noted that, as previously mentioned, this may be the only way to solve certain systems. However large systems are usually solved more efficiently to a required tolerance using an iterative solver.

Iterative solvers can also be more effective if they are given a good initial approximation, ξ_0, of the solution to the system, in equation (4.9). This will help to further reduce the number of iterations required. If we refine the mesh, we can then make use of the solution from the previous mesh as the starting solution, ξ_0, in the current solve. The only time that this cannot be done, obviously, is for the first mesh; but this usually will be a smaller, perhaps better conditioned system anyway, so finding the first solution will not be so difficult.

There are now some powerful methods for iteratively solving asymmetric systems, for example, generalised minimum residual [54], and for other iterative solvers [20, 36]

One of the most powerful iterative solvers used in the finite element method, for positive definite systems, is the *conjugate gradient* method, which is detailed below for the solution of the system $A\boldsymbol{\xi} = \boldsymbol{b}$.

The conjugate gradient algorithm

Further details of this are given in [16, 24, 40]

$$\begin{aligned} \tau_i &= \frac{\boldsymbol{g}_i \cdot \boldsymbol{g}_i}{\boldsymbol{d}_i \cdot A\boldsymbol{d}_i}, && (4.11)\\ \boldsymbol{\xi}_{i+1} &= \boldsymbol{\xi}_i + \tau_i \boldsymbol{d}_i, && (4.12)\\ \boldsymbol{g}_{i+1} &= \boldsymbol{g}_i + \tau_i A\boldsymbol{d}_i, && (4.13)\\ \beta_i &= \frac{\boldsymbol{g}_{i+1} \cdot \boldsymbol{g}_{i+1}}{\boldsymbol{g}_i \cdot \boldsymbol{g}_i}, && (4.14)\\ \boldsymbol{d}_{i+1} &= -\boldsymbol{g}_{i+1} + \beta_i \boldsymbol{d}_i, && (4.15) \end{aligned}$$

where $\boldsymbol{g}_0 = A\boldsymbol{x}_0 - \boldsymbol{b}$ and $\boldsymbol{d}_0 = -\boldsymbol{g}_0$. As can be seen, the most computationally expensive step is the matrix vector product, steps involving $A\boldsymbol{d}_n$, so we would like to minimise the number of iterations so as to minimise the amount of work. This method converges within n iterations (if we use exact arithmetic) where n is the dimension of the matrix, although it usually takes less than this to converge to within the required tolerance, ϵ. However, as already mentioned, the number of iterations is greatly influenced by the condition number. The amount by which the error is reduced with each iteration can be shown to be the ratio:

$$\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}. \tag{4.16}$$

Obviously a smaller condition number implies a smaller number of iterations have to be performed to get to within the required tolerance. We must therefore ask the question, can the stiffness matrix be modified in some way to reduce the condition number?

4.7.4 Preconditioning

The process of reducing the condition number of a given matrix is called *preconditioning.* To get the most computational benefit from the preconditioning step the conjugate gradient method must be modified slightly. A matrix, C, the *preconditioning matrix* is introduced, to use the preconditioned conjugate gradient method then C *must* be symmetric positive definite. For it to be an effective preconditioner we should have that $\kappa(C^{-1}K) \ll \kappa(K)$

It is well known that the process of static condensation reduces the condition number, so this is also an important part of the preconditioning step.

The *preconditioned* conjugate gradient algorithm

For details of this see [16, 24, 40].

$$\begin{aligned}
\tau_i &= \frac{\boldsymbol{g}_i \cdot \boldsymbol{h}_i}{\boldsymbol{d}_i \cdot K\boldsymbol{d}_i}, && (4.17)\\
\boldsymbol{\xi}_{i+1} &= \boldsymbol{\xi}_i + \tau_i \boldsymbol{d}_i. && (4.18)\\
\boldsymbol{g}_{i+1} &= \boldsymbol{g}_i + \tau_i K\boldsymbol{d}_i. && (4.19)\\
\boldsymbol{h}_{i+1} &= C^{-1}\boldsymbol{g}_{i+1}. && (4.20)\\
\beta_i &= \frac{\boldsymbol{g}_{i+1} \cdot \boldsymbol{h}_{i+1}}{\boldsymbol{g}_i \cdot \boldsymbol{h}_i}, && (4.21)\\
\boldsymbol{d}_{i+1} &= -\boldsymbol{h}_{i+1} + \beta_i \boldsymbol{d}_i, && (4.22)
\end{aligned}$$

where $\boldsymbol{g}_0 = A\boldsymbol{x}_0 - \boldsymbol{b}$, $\boldsymbol{h}_0 = C^{-1}\boldsymbol{g}_0$ and $\boldsymbol{d}_0 = -\boldsymbol{h}_0$. The step (4.20) is the preconditioning step. A simple, yet effective, preconditioner is *diagonal scaling.* This is just the inverse of the leading diagonal of the matrix, $C = diag\ K$. A very effective preconditioner is described in [5, 6] and [9, 11] show the preconditioner in use on some test problems. The structure of the refinements in the mesh is used to construct the preconditioner; the tree structure that is discussed in this thesis enabled it to be implemented easily.

4.8 Conclusions

The solvers detailed above form an integral part of the matrix library and the finite element implementation as a whole. The solvers were implemented in such a way that the addition of another matrix type would not require any modification. Indeed, a distributed matrix has been implemented and used successfully in several test problems.

The matrix library detailed here forms an easily expandable, efficient and implementation and platform independent library, that can be used throughout any finite element implementation, or indeed any other time that a flexible matrix library is required. For maximum flexibility and efficiency a number of matrix types has been implemented. Indeed, during the course of the research a distributed matrix and triangular matrix were also added to this library and have been used successfully in a number of test cases. The distributed matrix uses the process for solving described in Chapter 3. If the user wanted to use any of the many sparse matrix representations available then they would simply have to derive a new type, from the base type, that encoded the new representation. All of the operations (solvers, etc.) would be available for use immediately with no further effort. So what we have done is to implement a new object orientated matrix library in Ada. This has far wider applications than just the finite element programme which is the topic of discussion in this thesis.

With the introduction of further element types the extension of the matrix type will be essential, especially the implementation of matrices that have large zero and non zero blocks.

There is another powerful iterative solver that is often used for solving the matrix equation, this is multilevel [28, 56]. It has not been mentioned because it was not considered as a part of this thesis, but as the implementation of the subdomain is the tree structure then the implementation of this solver is a fairly simple one.

Chapter 5

Numerical examples

In this chapter we will show the ideas discussed previously in this thesis fitting together, and illustrating the benefits that are obtained by using a more flexible data structure. To get the meshes that are shown in various figures, we used an automatic refinement strategy [10, 11], this relies on a reliable estimate of the error [7, 8, 49] being known. To get this estimate of the error we used a technique based on the procedure described in [8]. Other refinement strategies exist, such as that described in [51], in here they make use of the data structures described in [32] and the error estimator from [49].

5.1 Scalar problems

Both of the examples in this section are scalar analogues for the equivalent problems in linear elasticity. Each example is over different domains, with boundary conditions chosen so that the analytical solution can be determined. Each of the solutions will have a singularity, but of different strengths. As the analytical solutions are known, this enable us to compare the results with the analytical solution to get the convergence rates for each of the methods that will be discussed.

5.1.1 Crack domain

Figure 5.1 shows the full domain for the crack problem, but because of symmetry along the x-axis shown we need only solve the problem on half of the domain. We chose the upper half of the domain, this can be seen in Figure 5.2. The

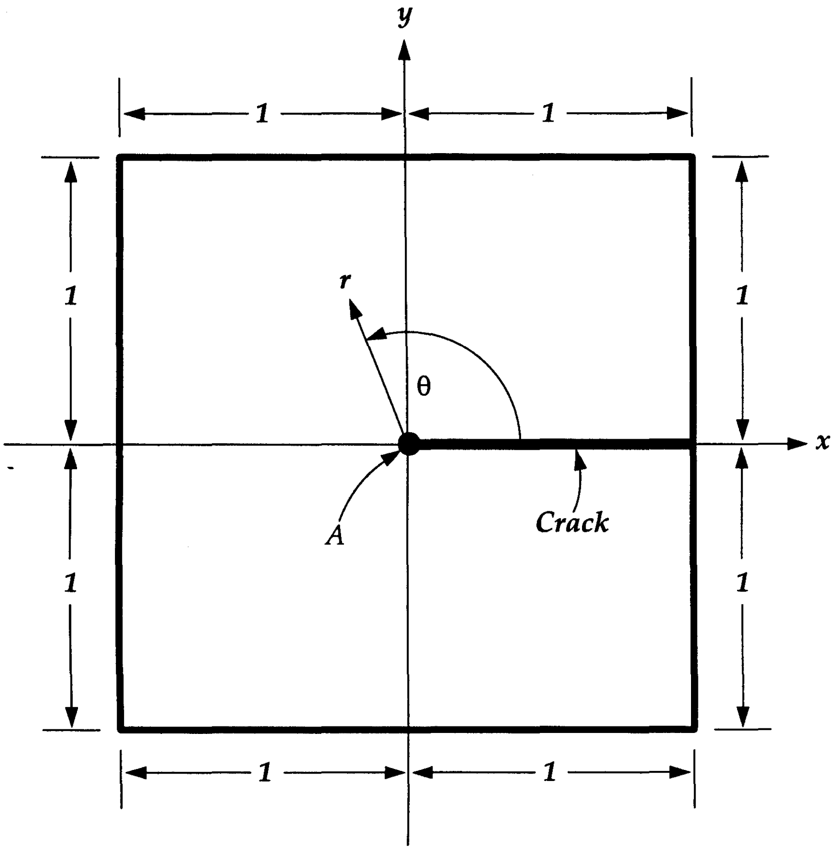


Figure 5.1: The full domain for the crack problem.

boundary conditions were chosen so that the true solution has the form:

$$u(r,\theta) = r^{\lambda}\cos\lambda\theta. \tag{5.1}$$

We will solve the domain with $\lambda = 1.5$ and $\lambda = 0.5$. This corresponds to different strengths of singularity.

The meshes that are shown in Figures 5.4, 5.5, 5.7 and 5.8 were obtained using the adaptive refinement strategy and error estimation procedure described earlier.

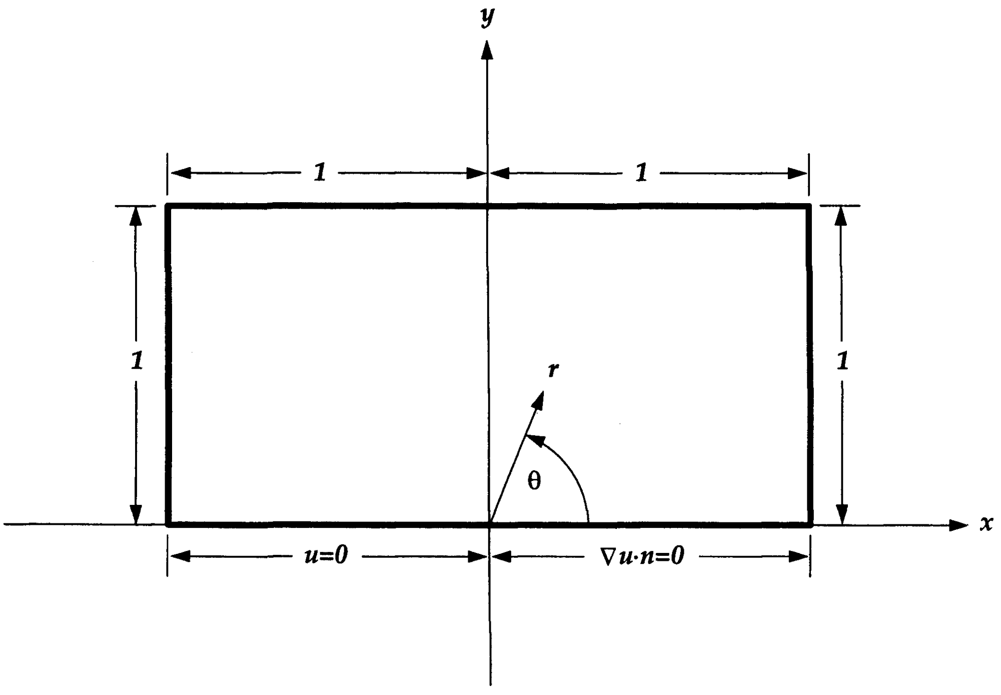


Figure 5.2: Half slit domain for scalar case.

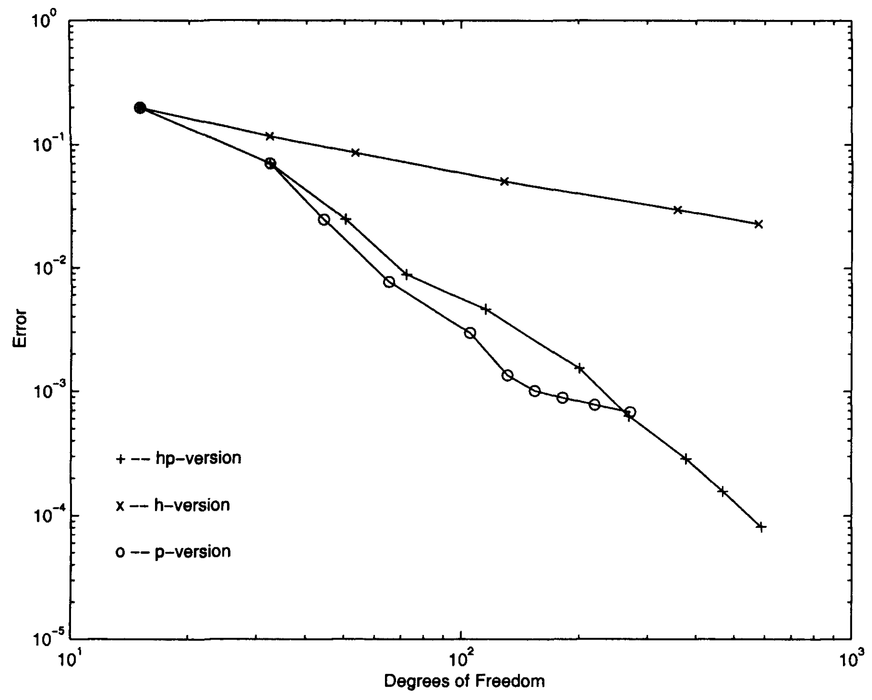


Figure 5.3: Convergence for the crack domain with $\lambda = 1.5$.

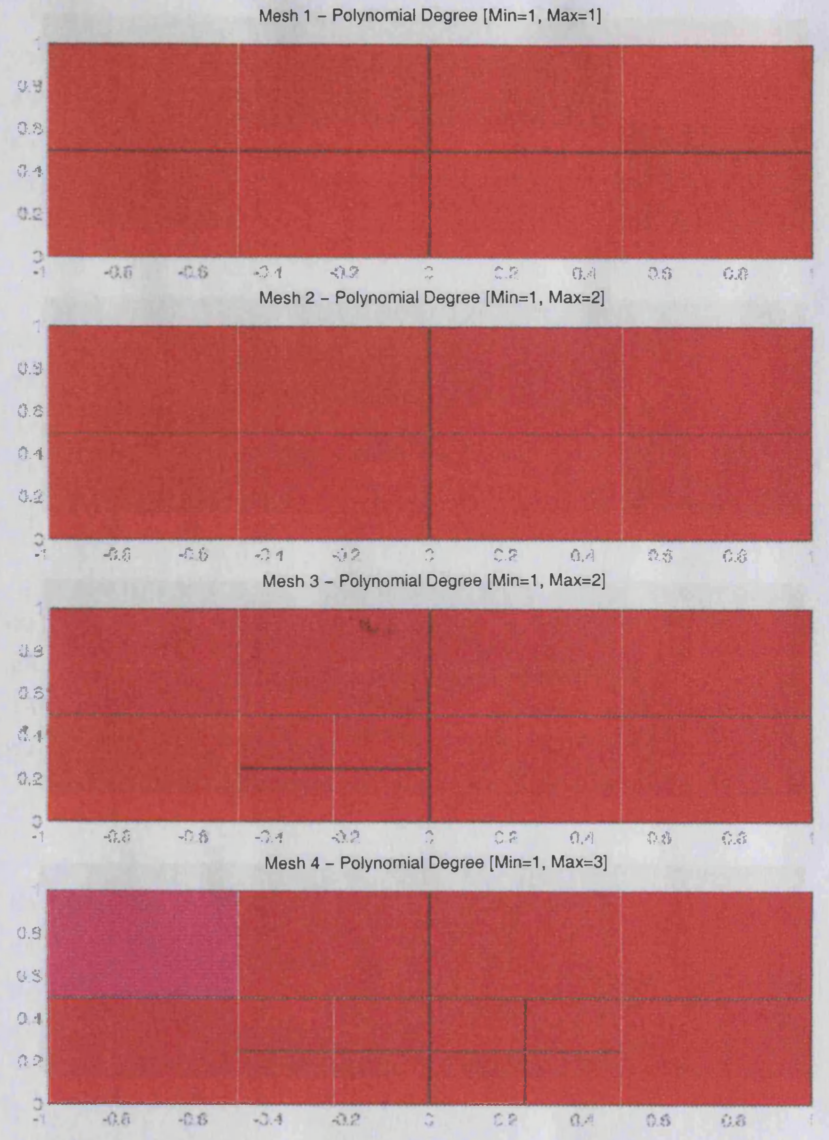


Figure 5.4: Figures for the crack domain with $\lambda = 1.5$. (red shading indicates lower order, blue high order).

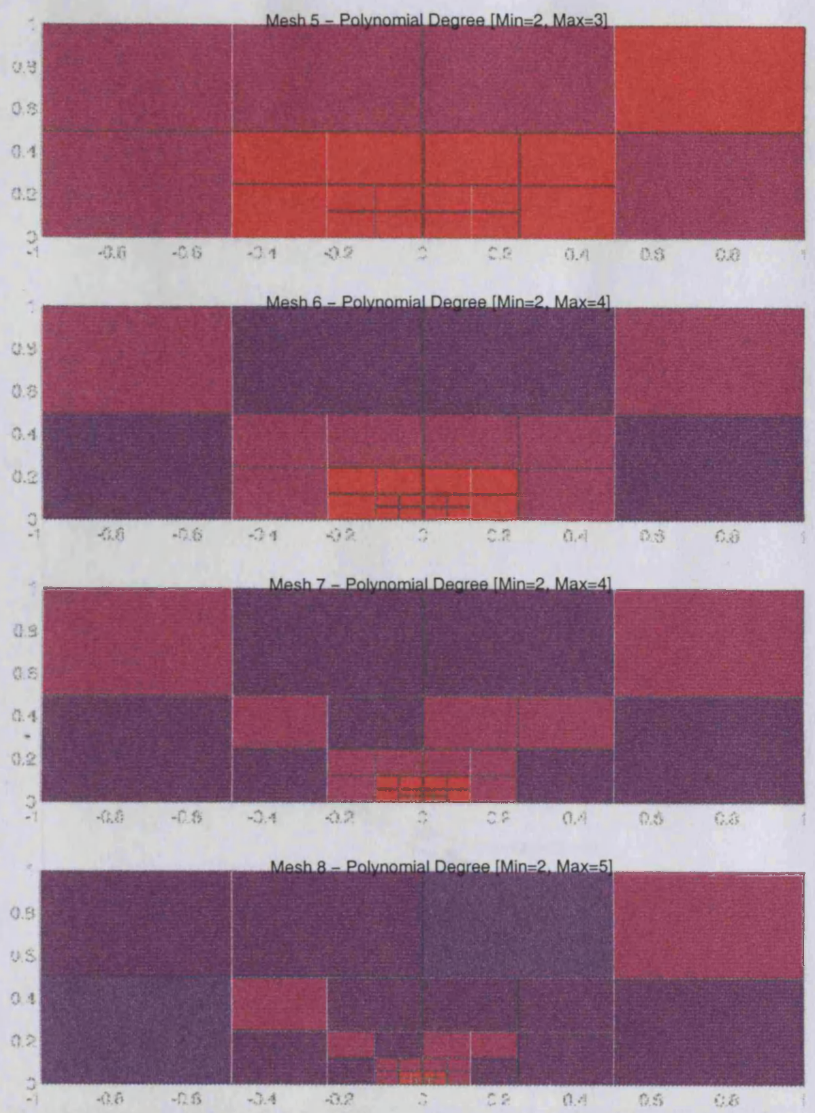

Mesh 5 – Polynomial Degree [Min=2, Max=3]
Mesh 6 – Polynomial Degree [Min=2, Max=4]
Mesh 7 – Polynomial Degree [Min=2, Max=4]
Mesh 8 – Polynomial Degree [Min=2, Max=5]

Figure 5.5: Refinements for the crack domain with $\lambda = 1.5$. (red shading indicates lower order, blue high order).

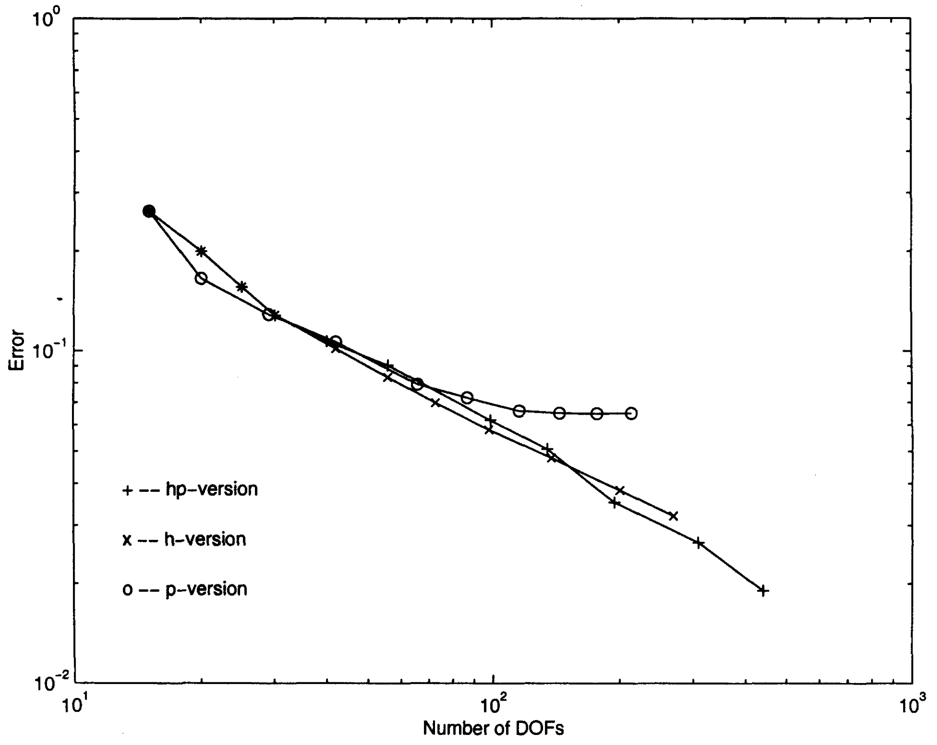


Figure 5.6: Convergence for the crack domain with $\lambda = 0.5$.

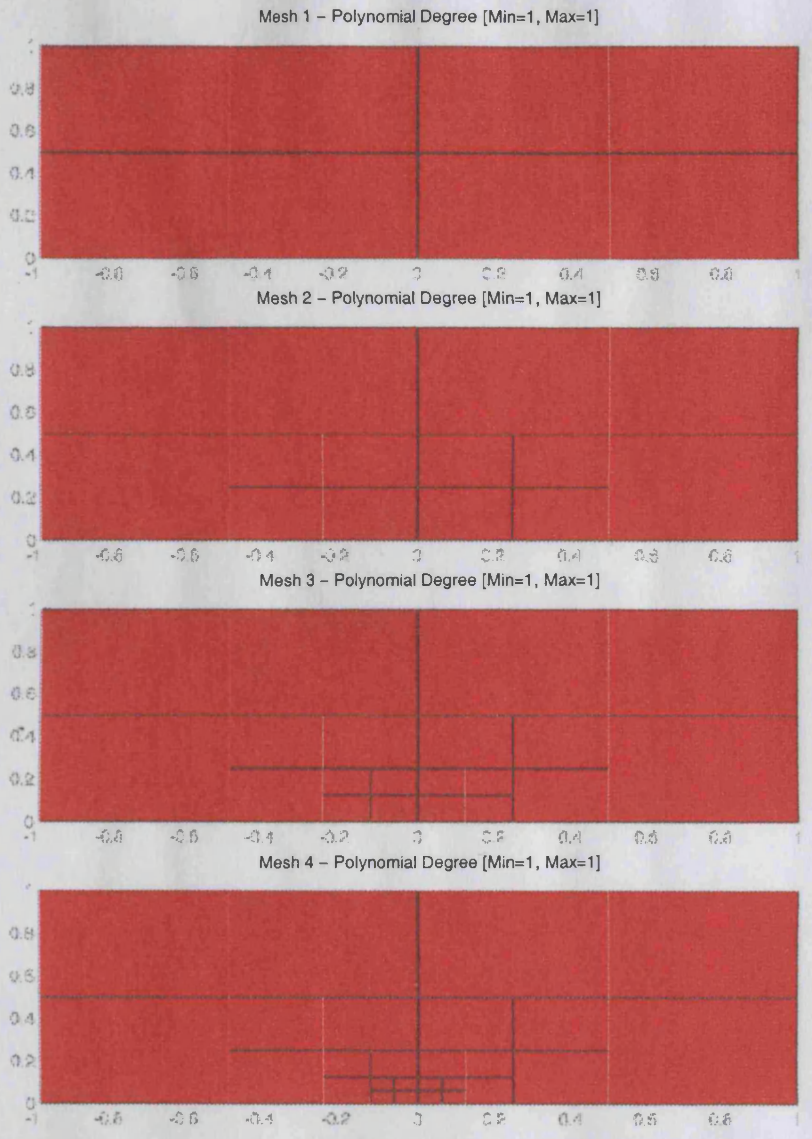
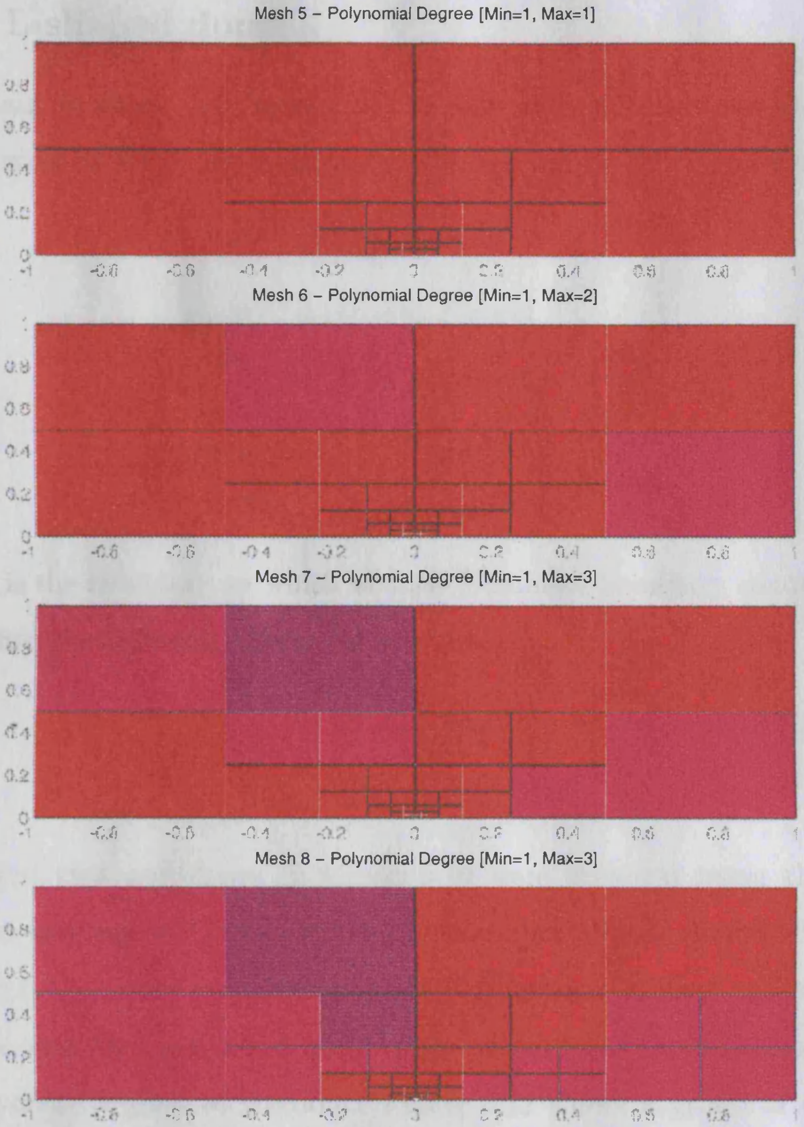

Mesh 1 - Polynomial Degree [Min=1, Max=1]
Mesh 2 - Polynomial Degree [Min=1, Max=1]
Mesh 3 - Polynomial Degree [Min=1, Max=1]
Mesh 4 - Polynomial Degree [Min=1, Max=1]

Figure 5.fi: Refinements for the crack domain with $\lambda = 0.5$. (red shading indicates lower order, bluer high order).



Mesh 5 - Polynomial Degree [Min=1, Max=1]
Mesh 6 - Polynomial Degree [Min=1, Max=2]
Mesh 7 - Polynomial Degree [Min=1, Max=3]
Mesh 8 - Polynomial Degree [Min=1, Max=3]

Figure 5.fi: Refinements for the crack domain with $\lambda = 0.5$. (red shading indicates lower order, bluer high order).

5.1.2 L-shaped domain

The domain is shown in Figure 5.9. It represents a corner singularity. The problem, with its boundary conditions, to be solved on this domain is:

$$\begin{aligned}
-\Delta u &= 0 \text{ in } \Omega, && (5.2)\\
u(r,0) &= 0 \text{ on } 0 < r < 1, && (5.3)\\
u(r,\frac{3\pi}{2}) &= 0 \text{ on } 0 < r < 1, && (5.4)\\
\frac{\partial u}{\partial n} &= r^{-\frac{1}{3}} \begin{pmatrix} \sin\frac{2}{3}\theta \\ \cos\frac{2}{3}\theta \end{pmatrix} \cdot n \text{ on} \partial_n, && (5.5)
\end{aligned}$$

where ∂_n is the boundary on which we have Neumann boundary conditions. This problem has the following analytical solution:

$$u = r^{\frac{2}{3}} \cos\left(\frac{2}{3}\theta\right). \qquad (5.6)$$

The meshes that are shown in Figure 5.10 were obtained using the adaptive refinement strategy and error estimation procedure already described.

Figure 5.11 shows a comparison in the performance of the h and hp methods. As can be seen the hp-method quite easily outperforms the h-method, in terms of error versus degrees of freedom. Figure 5.12 shows a group of curves that correspond to increasing the polynomial degree uniformly on a sequence of meshes refined with the 0.15 type refinements respectively.

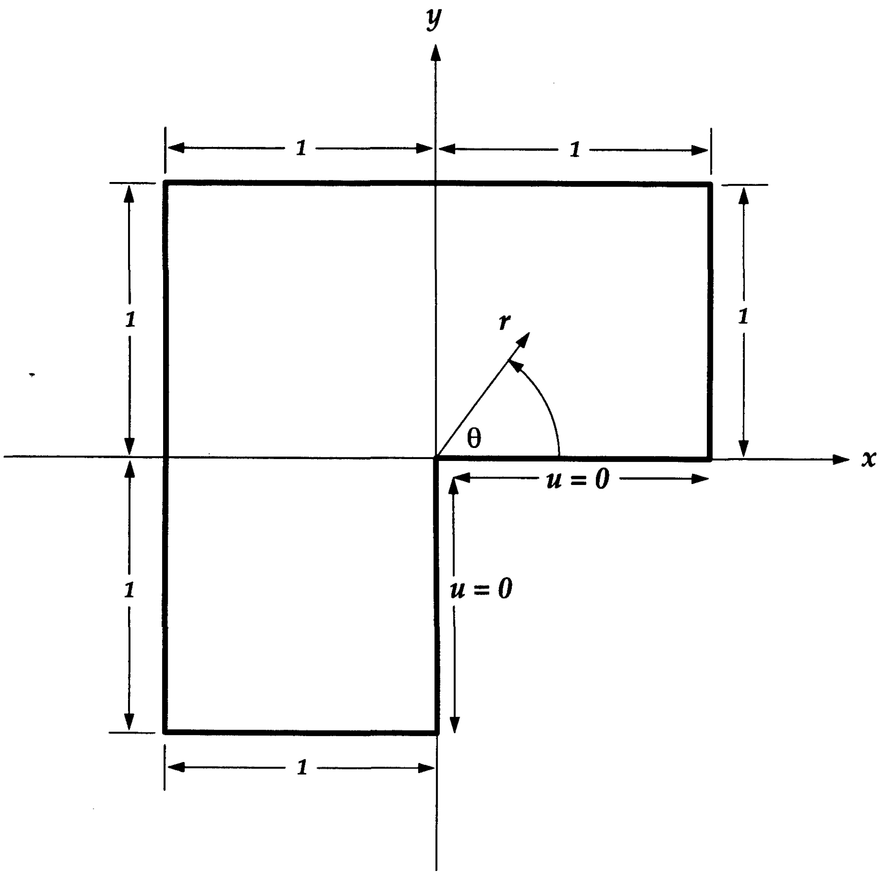


Figure 5.9: The domain for the L-shaped domain for scalar problems.

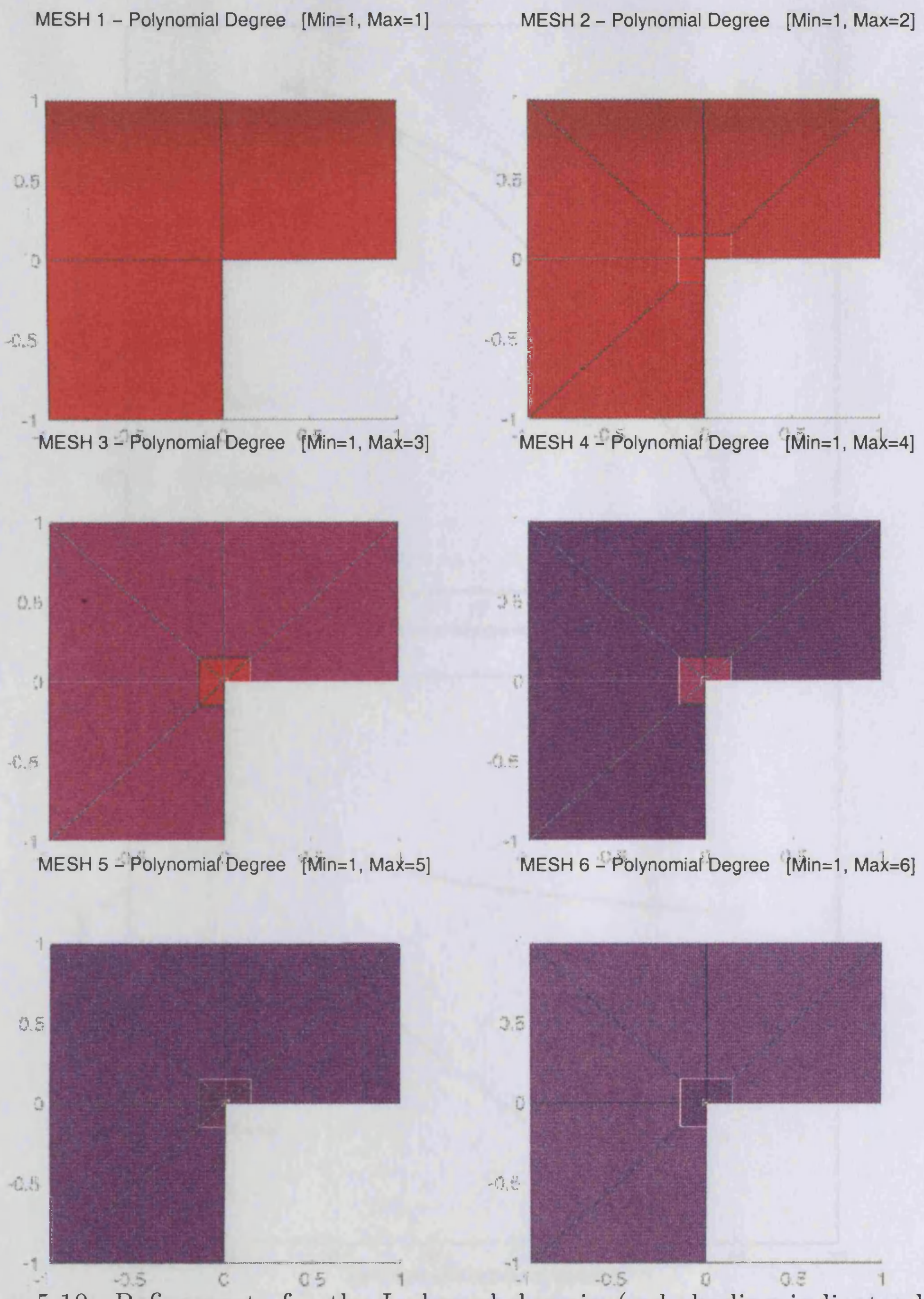


Figure 5.10: Refinements for the L-shaped domain (red shading indicates lower order, blue high order).

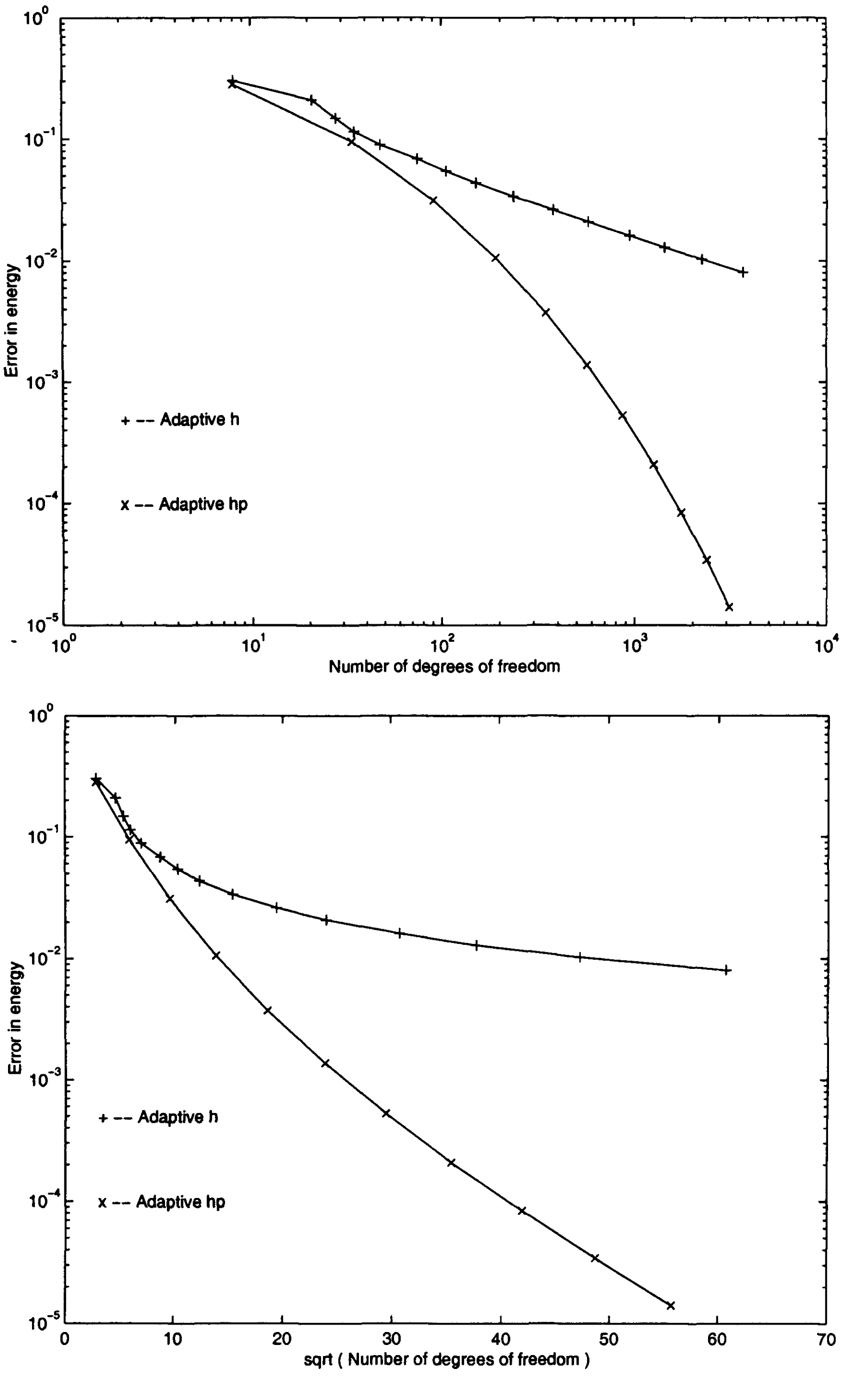
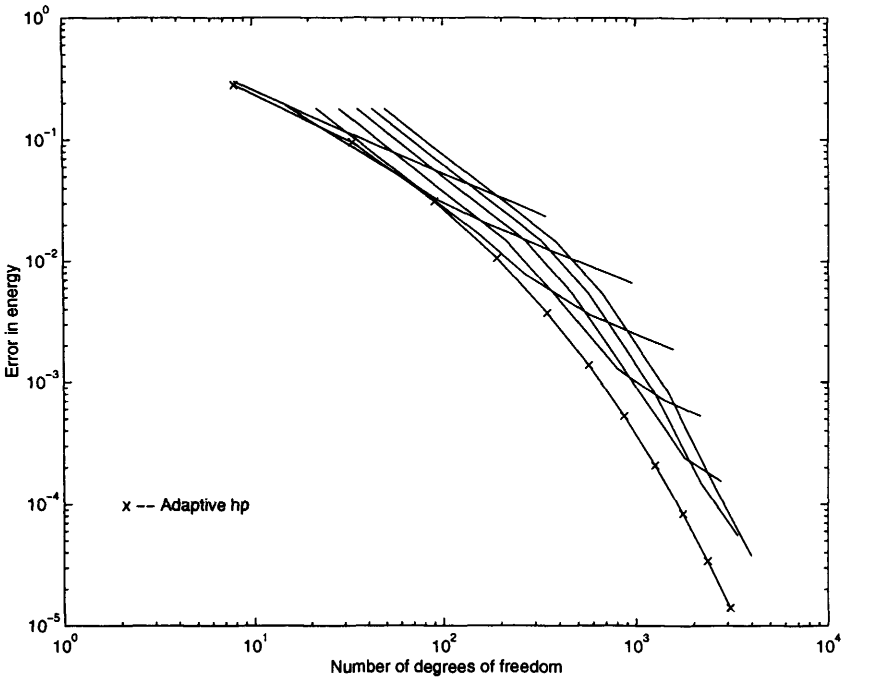


Figure 5.11: Comparison of rates of convergence obtained using the automatic h and hp-refinement strategies to solve the L-shaped domain problem.



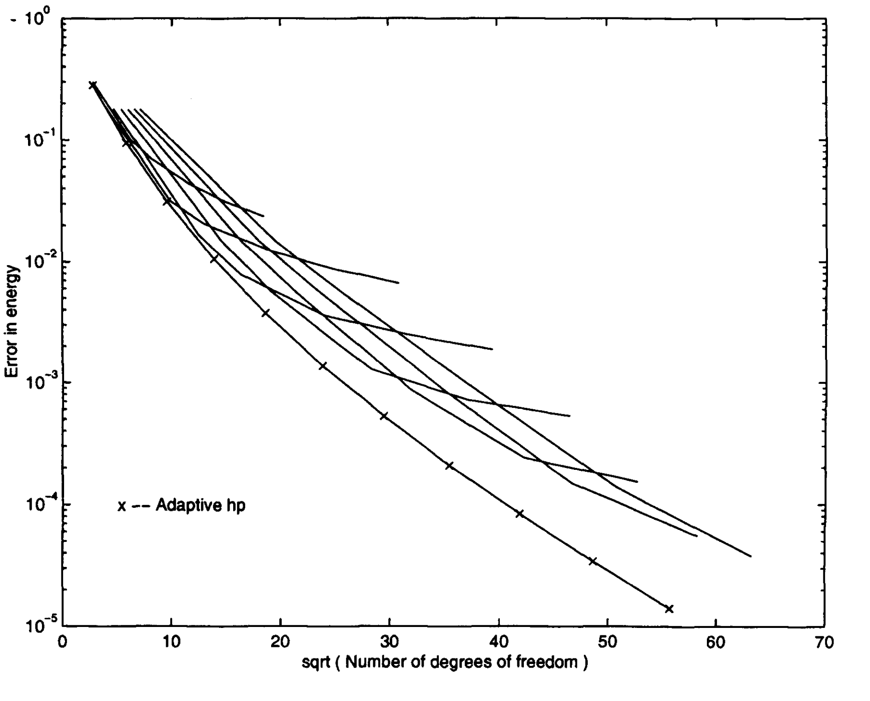


Figure 5.12: Convergence curves for the various versions of the finite element method applied the the scalar L-shaped domain problem. The group of curves correspond to a uniformly increasing p on a sequence of meshes refined geometrically towards the origin with a grading factor of 0.15. The single curve is that obtained from using the automatic *hp*-refinement strategy. The plots are shown on different scales.

5.1.3 Conclusions

The *hp*-method is usually superior, in terms of degrees of freedom versus error, to the other methods used. When it is not it is as good as the better of the two methods.

5.2 Linear elasticity L-shaped domain problems

We now show the data structure and the refinement strategy applied to a system of equations, that of linear elasticity. The equations for linear elasticity, for plane strain, are:

$$\frac{E(1-\nu^2)}{1-2\nu}\begin{bmatrix} \frac{\partial^2 u}{\partial x^2} + \frac{1-2\nu}{2(1-\nu)}\frac{\partial^2 u}{\partial y^2} + \frac{1}{2(1-\nu)}\frac{\partial^2 v}{\partial x \partial y} \\ \frac{1}{2(1-\nu)}\frac{\partial^2 u}{\partial x \partial y} + \frac{1-2\nu}{2(1-\nu)}\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \end{bmatrix} = \begin{bmatrix} -F_x \\ -F_y \end{bmatrix} \tag{5.7}$$

where u and v represent the $x-$ and $y-$displacements respectively. E is Young's modulus and ν is Poisson's ratio. For our problems we have chosen $E = 1$ and $\nu = 0.3$. Derivation of the equation (5.7) can be found in [60, 61, 65], and analysis of the following solution to the elasticity equations can be found in [60]. We will see numerics for what are known as mode 1 and mode 2 solutions, these correspond to varying strengths of singularity in the solution.

5.2.1 L-shaped domain, mode 1 solution

The domain is shown in Figure 5.13. In both of the examples, this one and the mode 2 solution, we have the definition (as we are doing plane strain)

$$\kappa = 3 - 4\nu$$

and

$$G = \frac{E}{2(1+\nu)}.$$

We have the following solution of the equation (5.7) in polar coordinates (r, θ):

$$\begin{aligned} u &= \frac{1}{2G} r^\lambda \left[(\kappa - Q(\lambda+1))\cos\lambda\theta - \lambda\cos(\lambda-2)\theta\right] \\ v &= \frac{1}{2G} r^\lambda \left[(\kappa + Q(\lambda+1))\sin\lambda\theta + \lambda\sin(\lambda-2)\theta\right] \end{aligned}$$

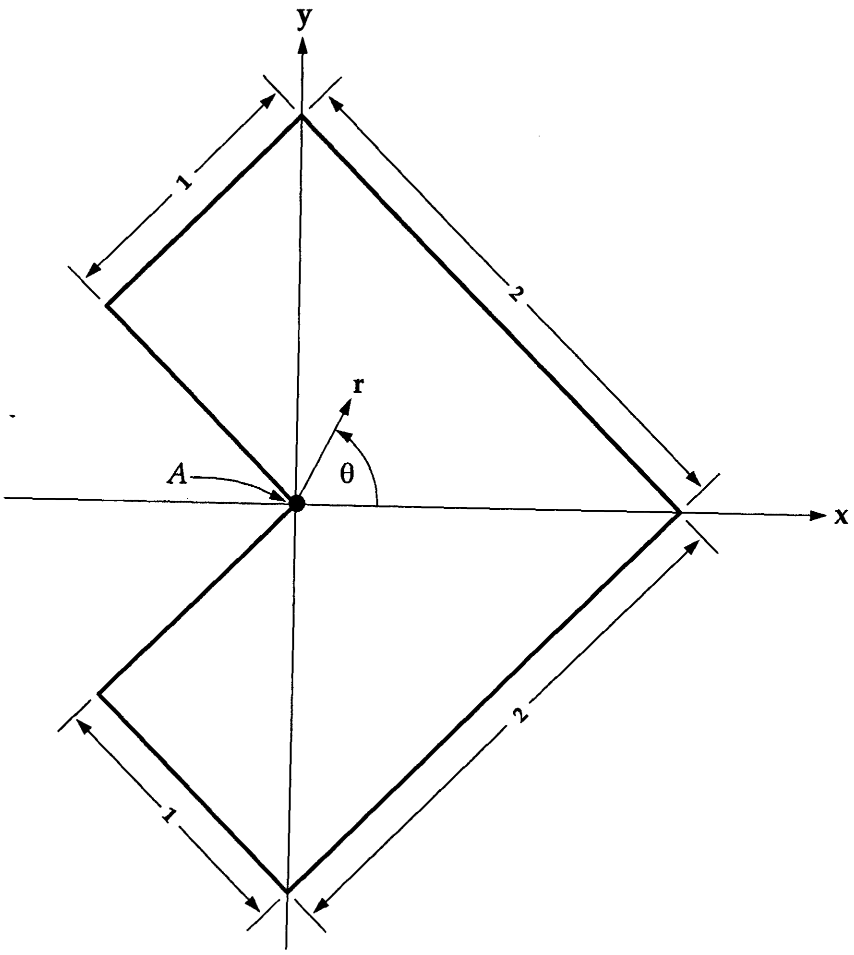


Figure 5.13: Geometry for L-shaped domain.

Mode	λ	Q	Energy
1	0.5444837367825	0.5430755788367	4.154544
2	0.9085291898461	-0.2189232362488	0.655806

Table 5.1: Table of values, $\nu = 0.3, E = 1.0$.

this gives $F_x = F_y = 0$. At the point A on the domain we have $\theta = 0$, this enables us to fix v to be zero at this point. The stress components for mode 1 are

$$\begin{aligned}
\sigma_x &= \lambda r^{\lambda-1}\left[(2 - Q(\lambda+1))\cos(\lambda-1)\theta - (\lambda-1)\cos(\lambda-3)\theta\right], && (5.8)\\
\sigma_y &= \lambda r^{\lambda-1}\left[(2 + Q(\lambda+1))\cos(\lambda-1)\theta + (\lambda-1)\cos(\lambda-3)\theta\right], && (5.9)\\
\tau_{xy} &= \lambda r^{\lambda-1}\left[(\lambda-1)\sin(\lambda-3)\theta + Q(\lambda+1)\sin(\lambda-1)\theta\right], && (5.10)
\end{aligned}$$

where λ is a solution of the following equation

$$\sin\lambda\frac{3\pi}{2} + \lambda\sin\frac{3\pi}{2} = 0,$$

and Q is defined to be:

$$Q = -\frac{\cos(\lambda-1)\frac{3\pi}{4}}{\cos(\lambda+1)\frac{3\pi}{4}} = -\frac{\lambda-1}{\lambda+1}\frac{\sin(\lambda-1)\frac{3\pi}{4}}{\sin(\lambda+1)\frac{3\pi}{4}}.$$

The choice of λ here, taken from Table (5.1), gives a very severe singularity at the point A in Figure 5.13. Equations (5.8) to (5.10) give us the following boundary conditions:

$$\frac{\partial \boldsymbol{u}}{\partial n} = \begin{bmatrix} n_x & 0 & n_y \\ 0 & n_y & n_x \end{bmatrix}\begin{bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{bmatrix}, \qquad (5.11)$$

where $\boldsymbol{u} = (u, v)$ and $\boldsymbol{n} = (n_x, n_y)$ is the outward normal. The numerical values for λ and Q and the strain energy can be found in Table (5.1). The meshes obtained from using the adaptive refinement strategy are shown in Figure 5.14, from these we get the convergence plots, Figure 5.15.

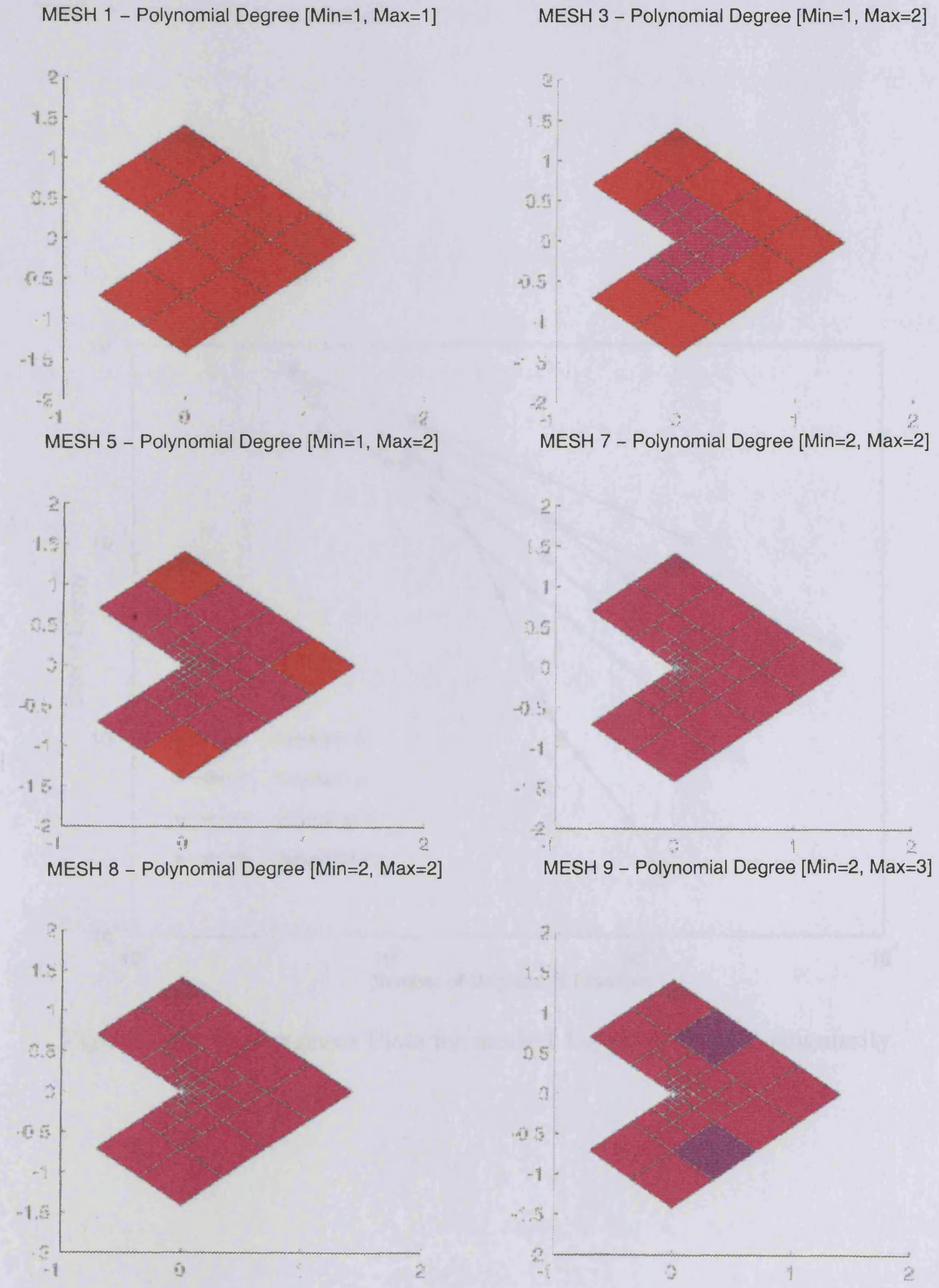

MESH 1 - Polynomial Degree [Min=1, Max=1]
MESH 3 - Polynomial Degree [Min=1, Max=2]
MESH 5 - Polynomial Degree [Min=1, Max=2]
MESH 7 - Polynomial Degree [Min=2, Max=2]
MESH 8 - Polynomial Degree [Min=2, Max=2]
MESH 9 - Polynomial Degree [Min=2, Max=3]

Figure 5.14: Adaptive *hp* meshes for mode 1 singularity, red elements indicate low order elements, blue high order.

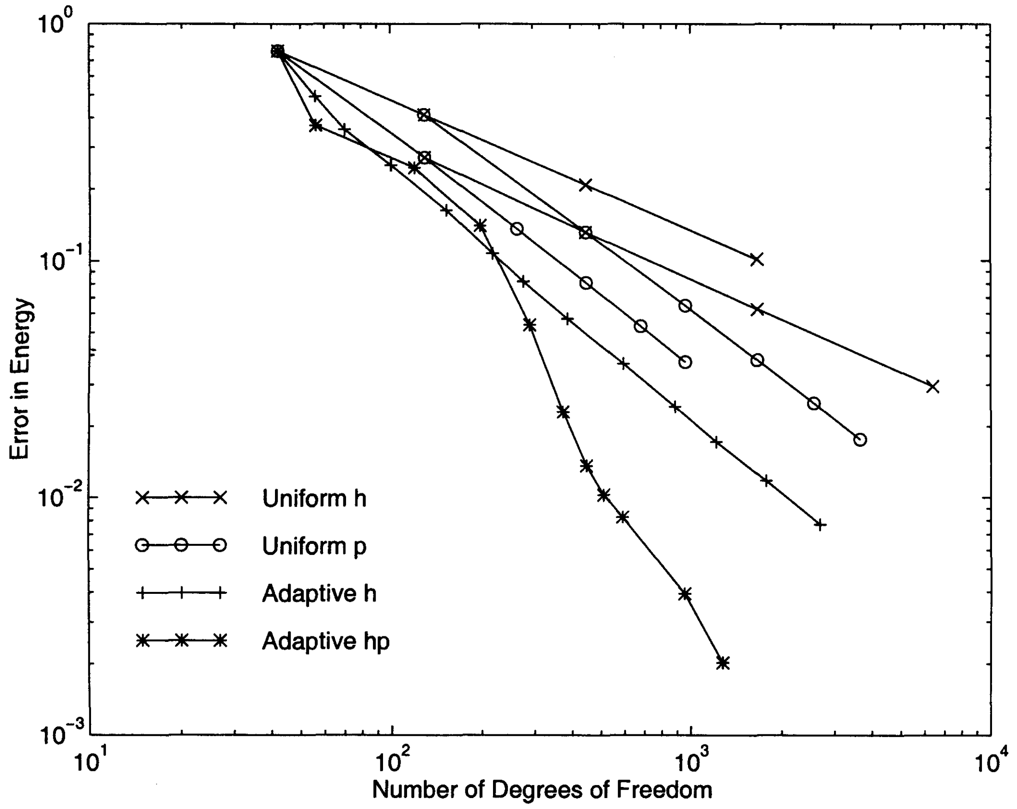


Figure 5.15: Convergence Plots for mode 1 L-shaped domain singularity.

5.2.2 L-shaped domain, mode 2 solution

We also have the following solution of equation (5.7) corresponding to the mode 2 solution in polar coordinates (r, θ):

$$\begin{aligned} u &= \frac{1}{2G} r^{\lambda} \left[(\kappa - Q(\lambda+1)) \sin \lambda\theta - \lambda \sin(\lambda - 2)\theta \right] \\ v &= -\frac{1}{2G} r^{\lambda} \left[(\kappa + Q(\lambda+1)) \cos \lambda\theta + \lambda \cos(\lambda - 2)\theta \right] \end{aligned}$$

At the points A in the domain we have that $\theta = 0$, this enables us to set $u = 0$, the $x-$displacement, here. The stress components for mode 2 are:

$$\sigma_x = \lambda r^{\lambda-1} \left[(2 - Q(\lambda+1)) \sin(\lambda-1)\theta - (\lambda-1)\sin(\lambda-3)\theta \right] \quad (5.12)$$

$$\sigma_y = \lambda r^{\lambda-1} \left[(2 + Q(\lambda+1)) \sin(\lambda-1)\theta + (\lambda-1)\sin(\lambda-3)\theta \right] \quad (5.13)$$

$$\tau_{xy} = -\lambda r^{\lambda-1} \left[(\lambda-1)\cos(\lambda-3)\theta + Q(\lambda+1)\cos(\lambda-1)\theta \right] \quad (5.14)$$

This time λ is a solution of the following equation:

$$\sin \lambda \frac{3\pi}{2} - \lambda \sin \frac{3\pi}{2} = 0$$

and Q is defined to be:

$$Q = -\frac{\sin(\lambda-1)\frac{3\pi}{4}}{\sin(\lambda+1)\frac{3\pi}{4}} = -\frac{\lambda-1}{\lambda+1} \frac{\cos(\lambda-1)\frac{3\pi}{4}}{\cos(\lambda+1)\frac{3\pi}{4}}$$

The choice of λ here, taken from Table (5.1), gives a less severe singularity than the mode 1 solution. Equations (5.12) to (5.14) give us the following boundary conditions:

$$\frac{\partial \boldsymbol{u}}{\partial n} = \begin{bmatrix} n_x & 0 & n_y \\ 0 & n_y & n_x \end{bmatrix} \begin{bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{bmatrix}, \quad (5.15)$$

where $\boldsymbol{u} = (u, v)$ and $\boldsymbol{n} = (n_x, n_y)$ is the outward normal. Again, the numerical values for λ, Q and the energy can be found in Table (5.1). The meshes obtained from performing the adaptive refinement strategy are shown in Figure 5.16, from these we get the convergence plots, Figure 5.17.

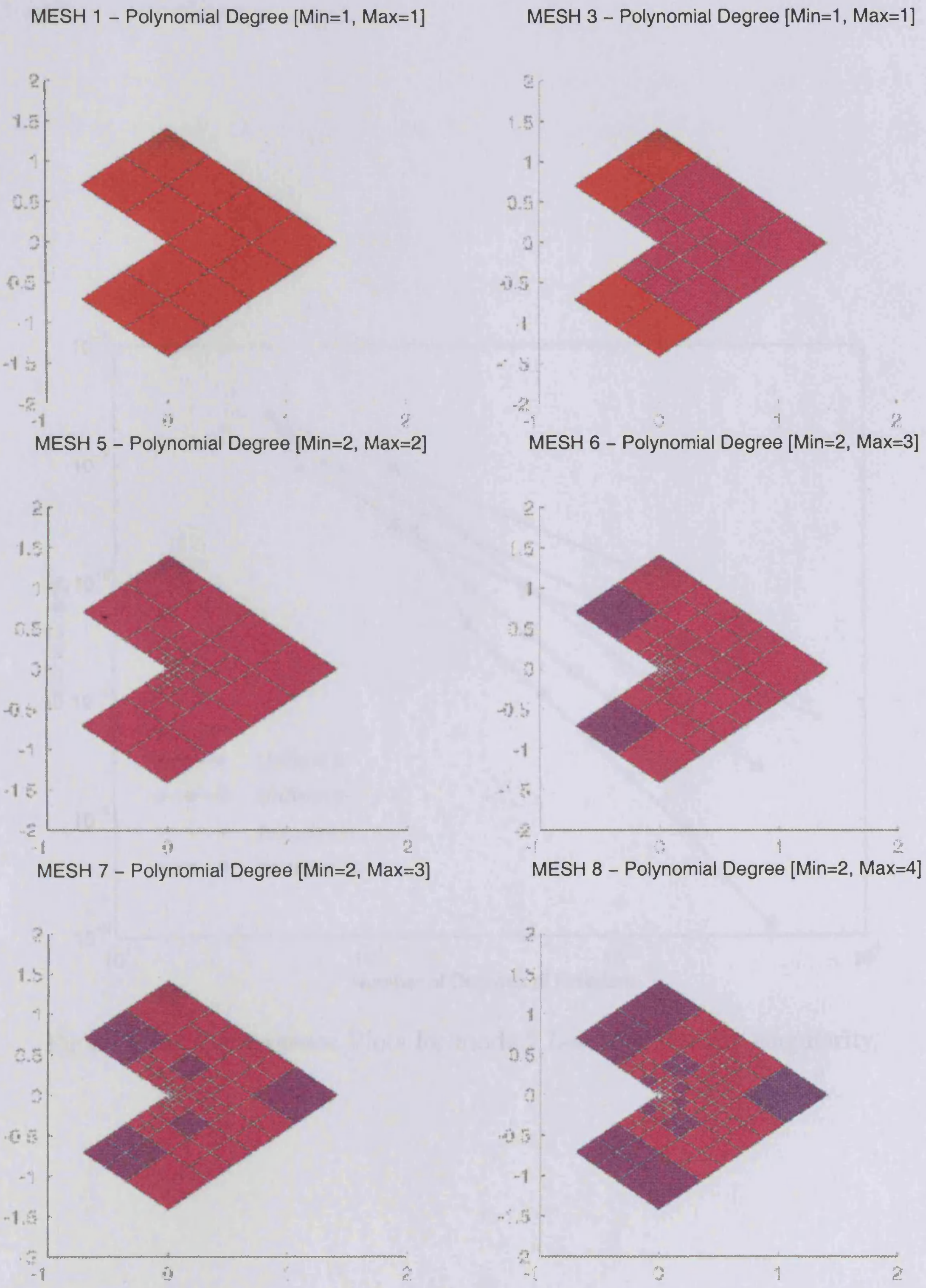

MESH 1 - Polynomial Degree [Min=1, Max=1]
MESH 3 - Polynomial Degree [Min=1, Max=1]
MESH 5 - Polynomial Degree [Min=2, Max=2]
MESH 6 - Polynomial Degree [Min=2, Max=3]
MESH 7 - Polynomial Degree [Min=2, Max=3]
MESH 8 - Polynomial Degree [Min=2, Max=4]

Figure 5.16: Adaptive *hp* meshes for mode 2 singularity, red elements indicate low order elements, blue high order.

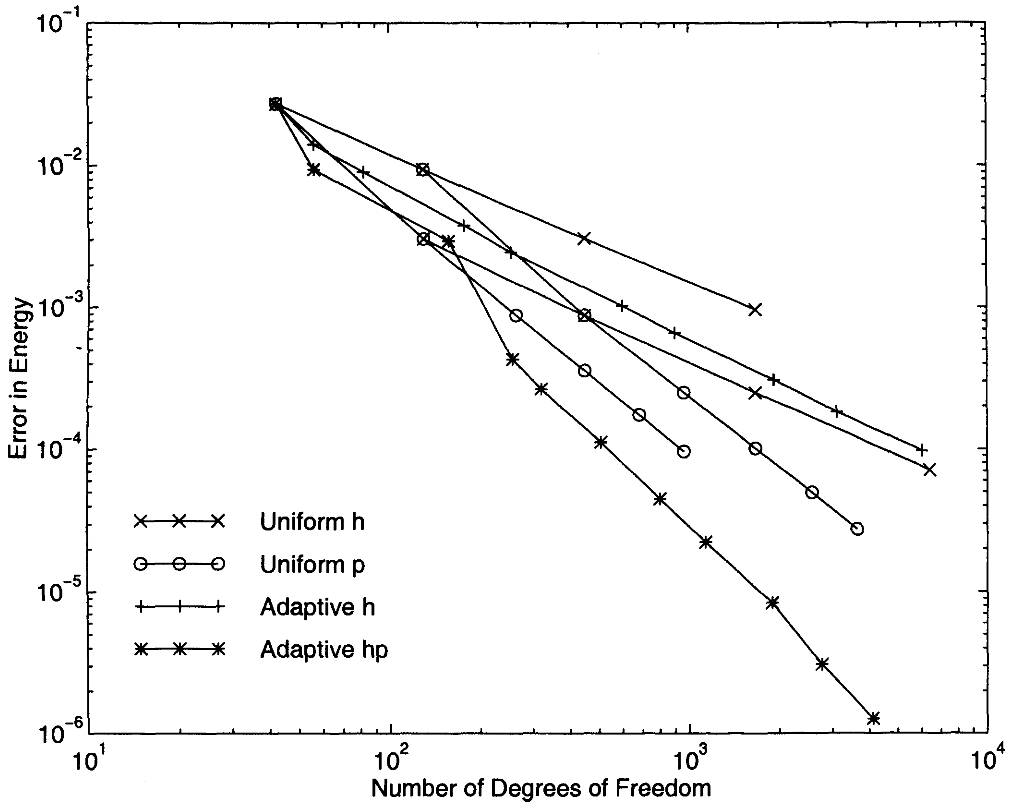


Figure 5.17: Convergence Plots for mode 2 L-shaped domain singularity.

5.2.3 Conclusions

We have to conclude that the best approach for the refinements, in terms of the minimum error for the minimum numbers of degrees of freedom, is to use an *hp*-refinement.

5.3 Linear elasticity cracked domain problems

As in the previous section, we will be testing the data structure and the refinement strategy for a system of equations, and as before it will be for plain strain linear elasticity. Our equation is

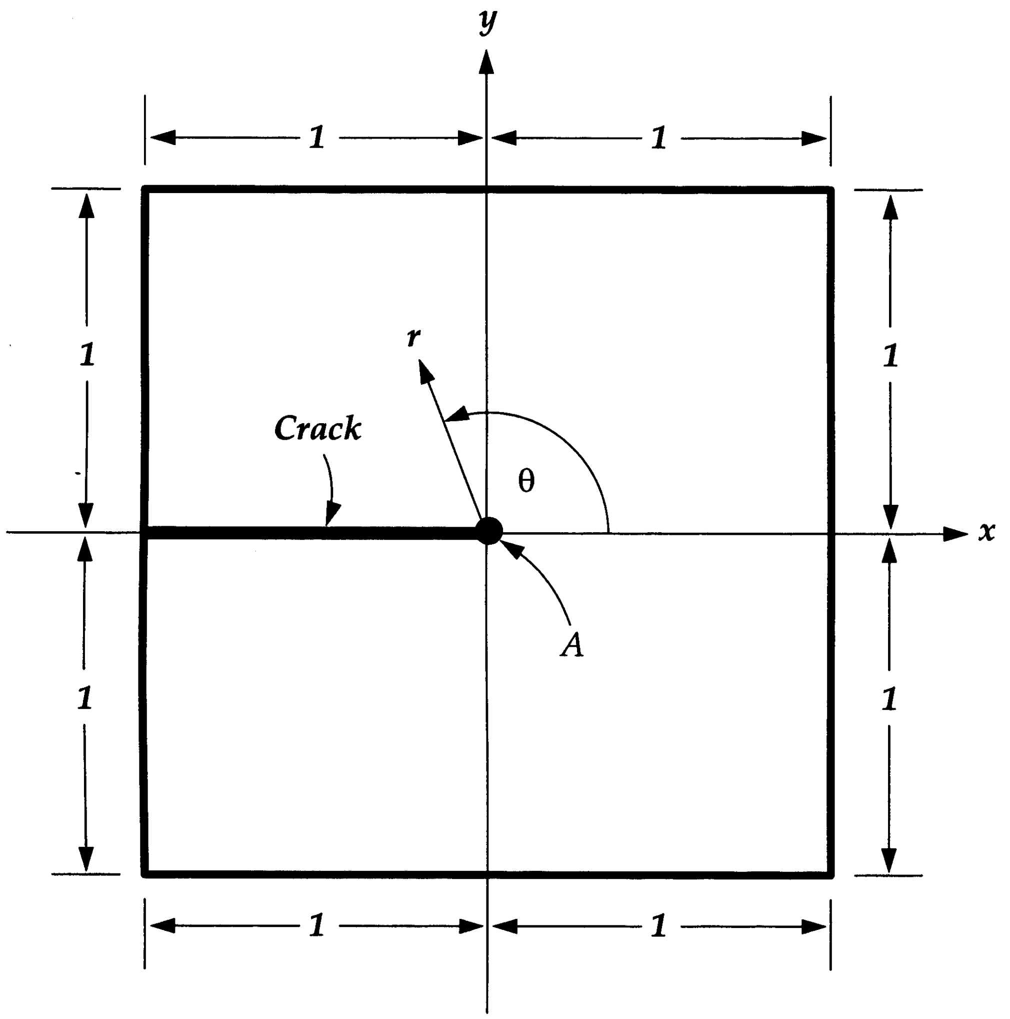


Figure 5.18: The geometry of the domain for the crack problem.

$$\frac{E(1-\nu^2)}{1-2\nu}\begin{bmatrix} \frac{\partial^2 u}{\partial x^2} + \frac{1-2\nu}{2(1-\nu)}\frac{\partial^2 u}{\partial y^2} + \frac{1}{2(1-\nu)}\frac{\partial^2 v}{\partial x \partial y} \\ \frac{1}{2(1-\nu)}\frac{\partial^2 u}{\partial x \partial y} + \frac{1-2\nu}{2(1-\nu)}\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \end{bmatrix} = \begin{bmatrix} -F_x \\ -F_y \end{bmatrix}, \qquad (5.16)$$

where u and v represent the $x-$ and $y-$displacements respectively, E is Young's modulus and ν is Poisson's ratio, which for our problems we have chosen to be

$\nu = 0.3$ and $E = 1.0$.

5.3.1 Crack domain, mode 1 solution

We have the following solution of the equation (5.16) in polar coordinates (r, θ):

$$\begin{aligned} u &= \frac{1}{2G} r^{\lambda} \left[(\kappa - Q(\lambda+1)) \cos \lambda\theta - \lambda \cos(\lambda - 2)\theta \right] \\ v &= \frac{1}{2G} r^{\lambda} \left[(\kappa + Q(\lambda+1)) \sin \lambda\theta + \lambda \sin(\lambda - 2)\theta \right] \end{aligned}$$

this gives $F_x = F_y = 0$. At the point A on the domain we have $\theta = 0$ this enables us to fix v to be zero at this point. The stress components for mode 1 are:

$$\begin{aligned} \sigma_x &= \lambda r^{\lambda-1} \left[(2 - Q(\lambda+1)) \cos(\lambda-1)\theta - (\lambda-1)\cos(\lambda-3)\theta \right], \\ \sigma_y &= \lambda r^{\lambda-1} \left[(2 + Q(\lambda+1)) \cos(\lambda-1)\theta + (\lambda-1)\cos(\lambda-3)\theta \right], \\ \tau_{xy} &= \lambda r^{\lambda-1} \left[(\lambda-1)\sin(\lambda-3)\theta + Q(\lambda+1)\sin(\lambda-1)\theta \right]. \end{aligned}$$

From (5.11) we again get the boundary conditions. Here λ is a solution of the following equation:

$$\sin \lambda 2\pi = 0;$$

this gives (for both mode 1 and mode 2 solutions):

$$\lambda = \pm\frac{1}{2}, \pm\frac{3}{2}, \pm 2, \pm\frac{5}{2}, \ldots \tag{5.17}$$

and Q is defined to be:

$$Q = -\frac{\cos(\lambda-1)\pi}{\cos(\lambda+1)\pi} = -\frac{\lambda-1}{\lambda+1}\frac{\sin(\lambda-1)\pi}{\sin(\lambda+1)\pi}.$$

5.3.2 Crack domain, mode 2 solution

We also have the following solution of the equation corresponding to mode 2 solution in polar coordinates (r, θ):

$$\begin{aligned} u &= \frac{1}{2G} r^{\lambda} \left[(\kappa - Q(\lambda + 1)) \sin \lambda\theta - \lambda \sin(\lambda - 2)\theta \right] \\ v &= -\frac{1}{2G} r^{\lambda} \left[(\kappa + Q(\lambda + 1)) \cos \lambda\theta + \lambda \cos(\lambda - 2)\theta \right] \end{aligned}$$

At the point A on the domain we have $\theta = 0$ this enables us to set $u = 0$, the x–displacement, at this point. The stress components for mode 2 are:

$$\begin{aligned} \sigma_x &= \lambda r^{\lambda-1} \left[(2 - Q(\lambda + 1)) \sin(\lambda - 1)\theta - (\lambda - 1) \sin(\lambda - 3)\theta \right], \\ \sigma_y &= \lambda r^{\lambda-1} \left[(2 + Q(\lambda + 1)) \sin(\lambda - 1)\theta + (\lambda - 1) \sin(\lambda - 3)\theta \right], \\ \tau_{xy} &= -\lambda r^{\lambda-1} \left[(\lambda - 1) \cos(\lambda - 3)\theta + Q(\lambda + 1) \cos(\lambda - 1)\theta \right], \end{aligned}$$

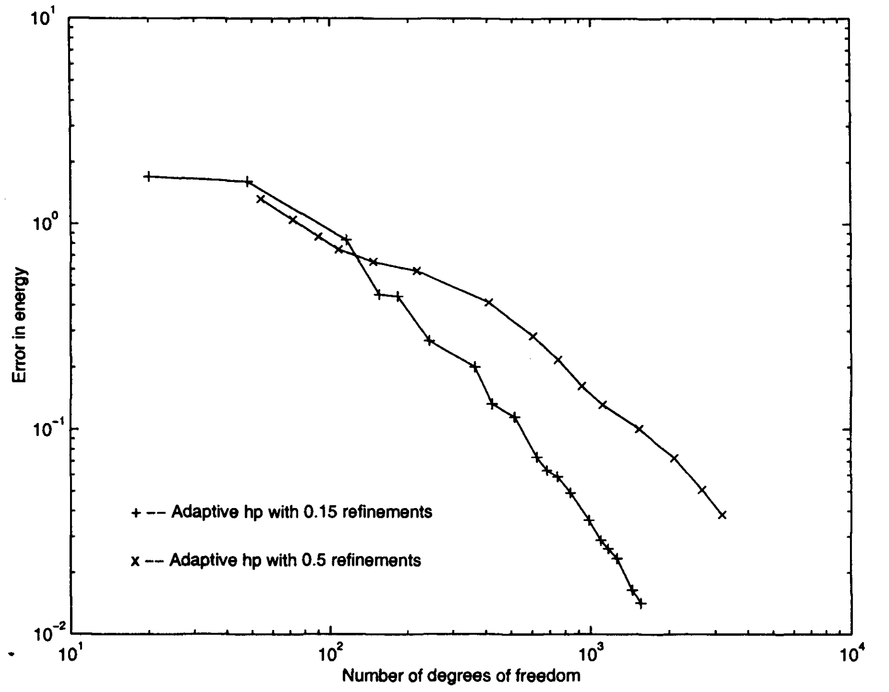
and Q is defined to be:

$$Q = -\frac{\sin(\lambda - 1)\pi}{\sin(\lambda + 1)\pi} = -\frac{\lambda - 1}{\lambda + 1} \frac{\cos(\lambda - 1)\pi}{\cos(\lambda + 1)\pi}.$$

from (5.15) we again get the boundary conditions. Figure 5.18 shows the domain approximated on. Figure 5.19 shows a comparison of the rates of convergence of the 0.15 type grading and 0.5 grading; both are *hp*-refined meshes. Figures 5.20 and 5.21 show the meshes obtained.

5.3.3 Conclusions

As in the previous chapter we conclude again that the best approach is to do a combination of *h*- and *p*-refinements. The best refinement strategy around singularities is that to use the 0.15 type refinements and increase p the polynomial degree in the outer elements, this is in agreement with the theory, predicted in [18]. Figure 5.21 shows the use of the connectivity mapping for a 0.15 constraint.



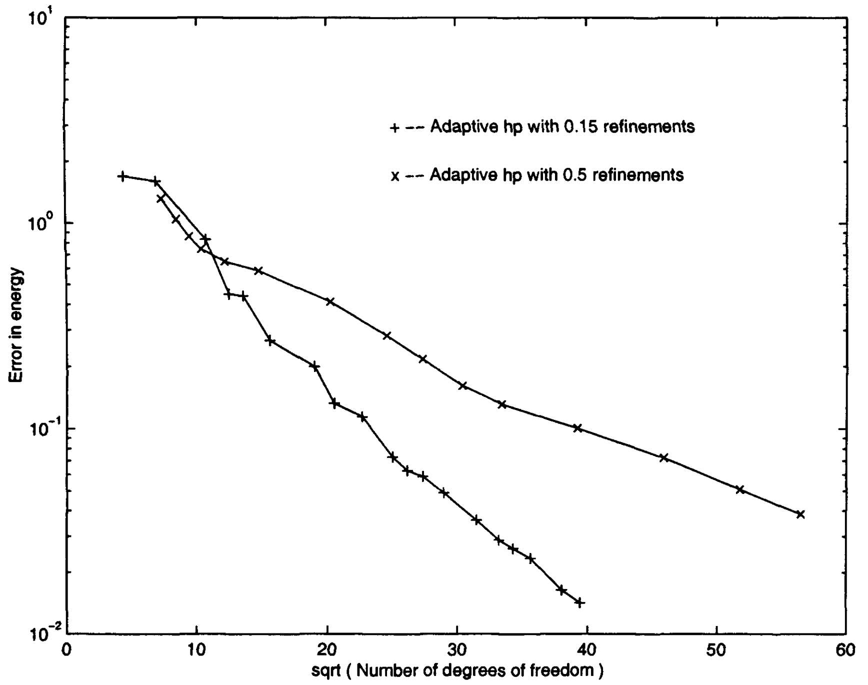


Figure 5.19: Comparison of the rates of convergence of the 0.15 type grading and 0.5 grading, both are *hp*-refined meshes.

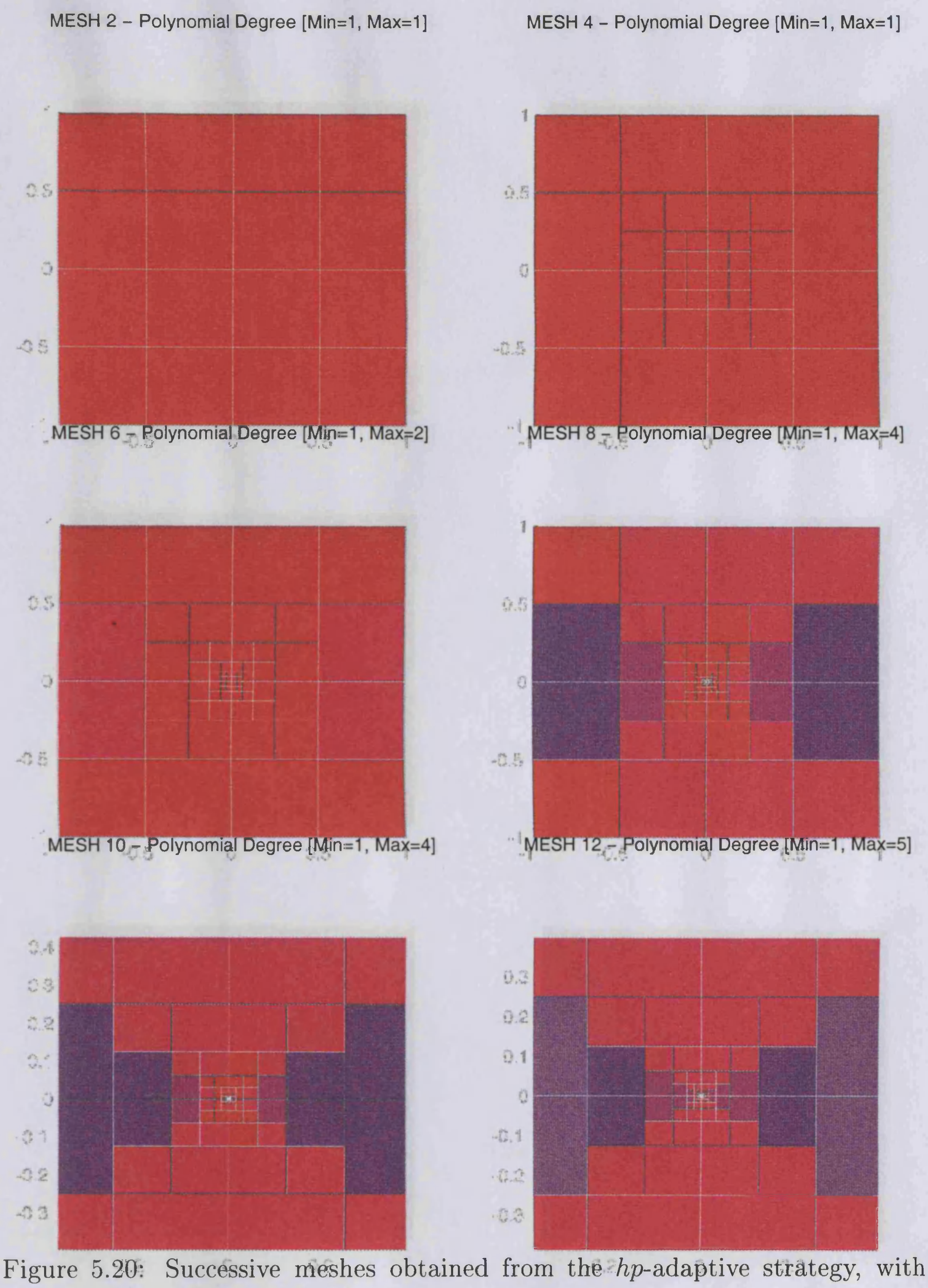

MESH 2 - Polynomial Degree [Min=1, Max=1]
MESH 4 - Polynomial Degree [Min=1, Max=1]
MESH 6 - Polynomial Degree [Min=1, Max=2]
MESH 8 - Polynomial Degree [Min=1, Max=4]
MESH 10 - Polynomial Degree [Min=1, Max=4]
MESH 12 - Polynomial Degree [Min=1, Max=5]

Figure 5.20: Successive meshes obtained from the *hp*-adaptive strategy, with grading factor, $\sigma = 0.5$, red elements indicate low order elements, blue high order.

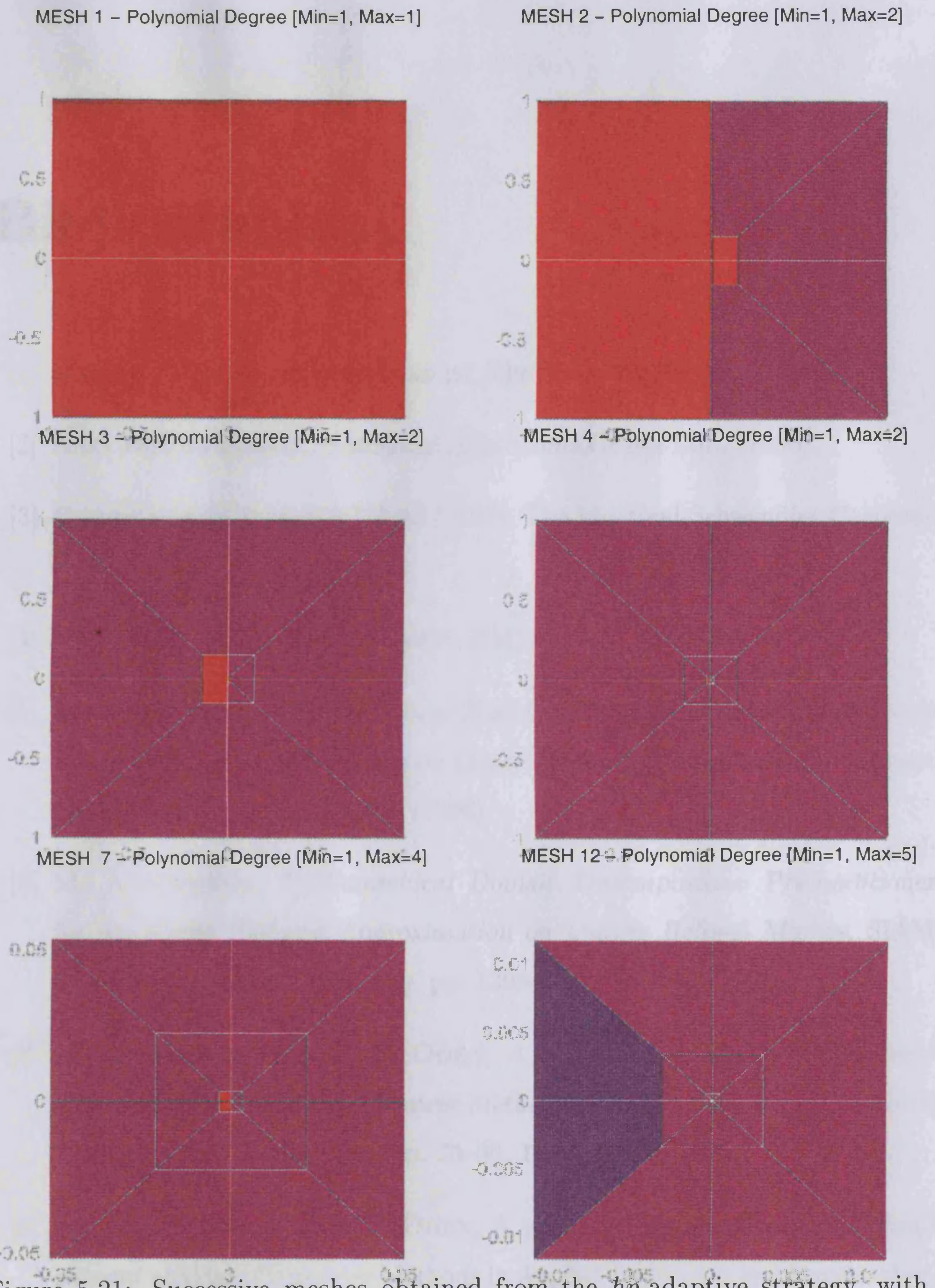

MESH 1 - Polynomial Degree [Min=1, Max=1]
MESH 2 - Polynomial Degree [Min=1, Max=2]
MESH 3 - Polynomial Degree [Min=1, Max=2]
MESH 4 - Polynomial Degree [Min=1, Max=2]
MESH 7 - Polynomial Degree [Min=1, Max=4]
MESH 12 - Polynomial Degree [Min=1, Max=5]

Figure 5.21: Successive meshes obtained from the *hp*-adaptive strategy, with grading factor, $\sigma = 0.15$, red elements indicate low order elements, blue high order.

Bibliography

[1] *Matlab, Language reference manual*, The Math Works, Inc., (1996).

[2] *NAG Fortran Library*, Numerical Algorithms Group Ltd., (1990).

[3] *Getting started with MSC/NASTRAN*, The MacNeal-Schwendler Corporation.

[4] *ProPHLEX User manuals*, Hibbitt, Karlsson and Sorensen, Inc.

[5] M. AINSWORTH, *A Preconditioner Based on Domain Decomposition for h–p Finite–Element approximation on Quasi–Uniform Meshes*, SIAM J. Numer. Anal., **33**, (4), pp. 1358–1376, (1996).

[6] M. AINSWORTH, *A Hierarchical Domain Decomposition Preconditioner for h-p Finite Element Approximation on Locally Refined Meshes*, SIAM Journal of scientific computing, pp. 1395–1413, **17**, (6), (1996).

[7] M. AINSWORTH AND J. T. ODEN, *A Procedure for A Posteriori Error Estimation for $h-p$ Finite Element Methods*, Computer Methods in Applied Mechanics and Engineering, pp. 73–96, **101**, (1–3), (1992).

[8] M. AINSWORTH AND J. T. ODEN, A posteriori *error estimation in finite element analysis*, Computer Methods in Applied Mechanics and Engineering, pp. 1–88, **142**, (1–2), (1997).

[9] M. Ainsworth, D. Andrews and B. Senior, *Preconditioners for the Adaptive hp–version Finite Element Method*, Proceedings of The Mathematics of Finite Elements and Applications, John Wiley and Sons, pp. 81–91, (1997).

[10] M. Ainsworth and B. Senior, *An Adaptive Refinement Strategy for hp–Finite Element Computations*, Applied Numerical Mathematics, pp. 165–178, **26**, (1–2), (1998).

[11] M. Ainsworth and B. Senior, *Aspects of an Adaptive hp–Finite Element Method: Adaptive Strategy, Conforming Approximation and Efficient Solvers*, Computer methods in applied mechanics and engineering, pp. 65–87, **150**, (1–4), (1997).

[12] M. Ainsworth and B. Senior, *hp–Finite Element Procedures on Non-Uniform Geometric Meshes: Adaptivity and Constrained Approximation*, To appear in Proceedings of IMA Workshop on Mesh Generation and Adaptive Methods for PDEs (to appear).

[13] J. E. Akin, *Finite element for analysis and design*, Academic Press, (1994).

[14] J. E. Akin, *Finite element analysis for undergraduates*, Academic Press, (1986).

[15] E. Anderson, Z. Bai, C. Bischof, J. W. Demmel, J. J. Dongarra, J. Du Croz A. Greenbaum, S. Hammarling, A. McKenne and D. Sorensen, LAPACK: *A portable linear algebra library for high-preformance computers*, Computer science dept. technical report CS-90-105 University of Tennessee, (1990).

[16] O. Axelsson and V. A. Barker, *Finite Element Solution of Boundary Value Problems*, Academic Press, Inc., (1984).

[17] I. Babuška, H. C. Elman and K. Markley, *Parallel Implementation of the hp–Version of the Finite Element Method on a Shared Memory Architecture*, SIAM Journal on Scientific and Statistical Computing, pp. 1433–1459, **13**, (6), (1992).

[18] Babuška, I. and Suri, Manil, *The h-p version of the finite element method with quasi-uniform meshes*, RAIRO Modélisation Mathématique et Analyse Numérique, pp. 199–238, **21**, (2), (1987).

[19] Babuška, Ivo and Suri, Manil, *The p- and h-p versions of the finite element method, an overview*, Computer Methods in Applied Mechanics and Engineering, **80**, (1-3), pp. 5–26, (1990).

[20] Z. J. Bai, D. Day, J. Demmel, J. Dongarra, M. Gu and A. Ruhe, *Templates for Linear Algebra Problems*, Lecture Nodes in Computer Science, pp. 115–140, **1000**, (1995)

[21] J. Barnes (Ed.), *Ada 95 rationale, The language, The standard libraries*, Springer, (1997).

[22] J. Barnes, *Programming in Ada 95*, Addison-Wesley Publishing Company, (1996).

[23] E. Barragy and F. Carey, *A Parallel Element by Element Solution Scheme*, International journal for numerical methods in engineering, pp. 2367–2382, **26**, (1988).

[24] R. Barret, M. Berry, T. F. Chan, J. Demmel, J. Dongarra, V. Eijkhout, R. Pozo, C.Romine and H. Van der Vorst, *Templates for the solution of linear systems: Building blocks for iterative methods*, SIAM, (1994).

[25] E. B. Becker, G. F. Carey and J. T.Oden, *Finite elements: An introduction, Volume 1*, Prentice-Hall, Inc, (1981).

[26] G. Booch, *Object-Oriented Analysis and Design with Applications, second edition*, The Benjamin/Cummings Publishing Company, Inc., (1994).

[27] G. Booch, *Software Components with Ada, Structures, Tools, and Subsystems*, The Benjamin/Cummings Publishing Company, Inc., (1987).

[28] J. H. Bramble, *Multigrid methods*, Pitman, (1993).

[29] R. L. Burden and J. D. Faires, *Numerical Analysis, Fourth Edition*, PWS-Kent, (1989).

[30] A. Burns and A. Wellings, *Concurrency in Ada*, Cambridge University Press, (1995).

[31] L. Demkowicz, *A modified FE assembling procedure with applications to electromagnetics, acoustics and hp–adaptivity*, The Mathematics of Finite Element Applications, pp. 93–102, (1997).

[32] L. Demkowicz, J. T. Oden, W. Rachowicz and O. Hardy, *Towards a Universal h-p Adaptive Finite Element Strategy, Part 1. Constrained Approximation and Data Structure*, Computer methods in applied mechanics and engineering, pp. 79-112, **77**, (1–2), (1989).

[33] J. Demmel, *LAPACK – A Portable Linear Algebra Library for High–Performance Computers*, Concurrency–Practice and Experience, pp. 655–666, **3**, (6), (1991)

[34] J. Dongarra and J. Demmel, *LAPACK – A Portable High–Performance Numerical Library for Linear Algebra*, Supercomputer, pp. 33–38, **8**, (6), (1991)

[35] J. Dongarra, R. Pozo and D. Walker, LAPACK++: *High performance linear algebra*, University of Tennessee, (1996).

[36] JACK DONGARRA AND ANDREW LUMSDAINE, *IML++: Iterative methods library reference guide*, (1996).

[37] I. S. DUFF, A. M. ERISMAN AND J. K. REID, *Direct methods for sparse matrices*, Oxford Science Publications, (1992).

[38] B. W. R. FORDE, R. O. FOSCHI AND S. F. STIEMER, *Object-oriented finite elememt analysis*, Computers and Structures, pp. 355–374, **34**, (1990).

[39] AL GEIST, ADAM BERGUELIN, JACK DONGARRA, WEICHENG JIANG, ROBERT MANCHEK AND VAIDY SUNDERAM, *PVM: Parallel Virtual Machine, A users guide and tutorial for networked parallel computing*, The MIT Press, (1994).

[40] G. H. GOLUB AND C. F. VAN LOAN, *Matrix Computations*, The Johns Hopkins University Press, (1989).

[41] D. M. HAWKEN, P. TOWNSEND AND M. F. WEBSTER, *The use of dynamic data structures in finite element applications*, International journal for numerical methods in engineering, pp. 1795–1811, **33**, (1992).

[42] C. JOHNSON, *Numerical solution of partial differential equations by the finite element method*, Cambridge University Press, (1987).

[43] D. KAY, *The p– and hp–finite element method applied to a class of non-linear elliptic partial differential equations*, Thesis, Leicester University, (1997).

[44] D. KINCAID AND W. CHENEY, *Numerical analysis, second edition*, Brooks/Cole publishing company, (1996).

[45] EDITED BY WILL LIGHT, *Advances in Numerical Analysis*, Oxford University Press, (1991).

[46] T. J. Liszka, *An introduction to hp–adaptive finite element method*, A COMCO technical report, (1995).

[47] T. J. Liszka, W. W. Tworzydlo, J. M. Bass, S. K. Sharma, T. A. Westermann and B. B. Yavari, *ProPHLEX - An hp-adaptive finite element kernel for solving coupled systems of partial differential equations in computational mechanics*, Computer Methods in Applied Mechanics and Engineering, pp. 251–271, **150**, (1–4), (1997).

[48] Mackenzie, J. A. and Morton, K. W., *Finite volume solutions of convection-diffusion test problems*, Mathematics of Computation, pp. 189–220, **60**, (201), (1992).

[49] 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*, Computer methods in applied mechanics and engineering, pp. 113-180, **77**, (1–2), (1989).

[50] R. Pozo and K. A. Remington, *SparseLib++ v. 1.5, sparse matrix class library reference guide*, (1996).

[51] W. Rachowicz, J. T. Oden and L. Demkowicz, *Towards a Universal h-p Adaptive Finite Element Strategy, Part 3. Design of h-p Meshes*, Computer methods in applied mechanics and engineering, pp. 181-212, **77**, (1–2), (1989).

[52] U. Rüde, *Data Structures for Multilevel Adaptive Methods and Iterative Solvers*, www.mgnet.org/mgnet-papers.html, (1992).

[53] U. Rüde, *Data Abstraction Techniques for Multilevel Algorithms*, Prepared for the proceedings of tha GAMM seminar on Multigrid methods, (1993).

[54] SAAD, YOUCEF AND SCHULTZ, MARTIN H., *GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM Journal on Scientific and Statistical Computing, pp. 856–869, **7**, (3), (1986).

[55] S. P. SCHOLZ, *Elements of an object-oriented FEM++ program in C++*, Computers and Structures, pp. 517–529, **43**, (1992).

[56] B. SMITH, P. BJØRSTAD AND W. GROPP, *Domain decomposition: Parallel multilevel methods for elliptic partial differential equations*, Cambridge University Press, (1996).

[57] I. M. SMITH AND D. V. GRIFFITHS, *Programming the Finite Element Method*, John Wiley and Sons, (1982).

[58] A. H. STROUD AND D. SECREST, *Gaussian Quadrature Formulas*, Prentice-Hall, Series in Automatic Computation, (1966).

[59] B. STROUSTRUP, *The C++ programming language second edition*, Addison-Wesley publishing company, (1991).

[60] B. SZABÓ AND I. BABUŠKA, *Finite Element Analysis*, John Wiley and Sons, Inc., (1991).

[61] S. P. TIMOSHENKO AND J. N. GOODIER, *Theory of Elasticity*, McGraw-Hill Book Company, (1970).

[62] S. TUCKER TAFT AND R. A. DUFF (EDS.), *Ada 95 reference manual, Language and standard libraries*, Springer, (1997).

[63] A. J. WATHEN, *An Analysis of some Element by Element Techniques*, Computer methods in applied mechanics and engineering, pp. 271–287, **74**, (1989).

[64] D. S. WATKINS, *Fundamentals of Matrix Computations*, Wiley, (1991).

[65] M. L. Williams, *Stress Singularities Resulting from Various Boundary Conditions in Angular Corners of Plates in Extension*, Journal of Applied Mechanics, (1952).

[66] Zeglinski Gordon W. and Han Ray P. S., *Object oriented matrix classes for the use in a finite element code using C++*, International journal for the numerical methods in engineering methods in engineering, pp. 3921–3937, **37**, (1994).

[67] T. Zimmermann, Y. Dubois-Pelerin and P. Bomme, *Object-oriented finite element programming: I. Governing principles*, Computer methods in applied mechanics and engineering, pp. 291–303, (1992).

[68] T. Zimmermann, Y. Dubois-Pelerin and P. Bomme, *Object-oriented finite element programming: II. A prototype in Smalltalk*, Computer methods in applied mechanics and engineering, pp. 361–397, (1992).