INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
NOTE TO USERS

The original manuscript received by UMI contains pages with slanted print. Pages were microfilmed as received.

This reproduction is the best copy available

UMI
OBJECT-ORIENTED IMPLEMENTATION OF
P-ADAPTIVE FINITE ELEMENT METHOD

by

Mandeep Singh

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Dr. John E. Akin, Chairman
Professor
Mechanical Engineering and Materials
Science

Dr. Andrew J. Meade,
Associate Professor
Mechanical Engineering and Materials
Science

Dr. Scott Collis,
Assistant Professor
Mechanical Engineering and Materials
Science

Houston, Texas

May, 1999
Object-Oriented Implementation of P-Adaptive
Finite Element Method

Mandeep Singh

Abstract

A finite element analysis methodology employing p-adaptivity is proposed. Object-oriented design and methodologies are used to implement the finite element package. The p-refinement phase is facilitated through the addition and deletion of general serendipity element edge nodes. Zienkiewicz-Zhu (ZZ) error estimator is used to determine the localized error. A modified superconvergent patch recovery technique is implemented to recover highly accurate nodal gradients utilized in the error estimation phase. Another variation to the ZZ error estimator, suggested by Blacker [12], is also tested.

The object-oriented design leads to easier maintainability and extensibility. The advantage of object-oriented design is the ability to try new solvers, new elements and new problem types with minimum programming effort and time.

The source code for the thesis is written in Fortran 90, with a graphical user interface (GUI) written in Java. The user interface performs pre-processing and post-processing. The Java user interface can add networking capabilities to the program. This method has been successfully applied to some benchmark problems.
To my parents Rajinder and Harjeet,

&

To my sisters Harpreet and Harmeet,

for

Their affection, care, inspiration and love.
Acknowledgments

I would like to express sincere thanks to my thesis advisor Dr. John E. Akin, whose support and guidance have been indispensable to this research. I am appreciative of his patience, understanding, and his positive outlook. I would also like to thank Dr. Scott Collis and Dr. Andrew J. Meade for their help and advice as well as for serving on my committee.

I would like to extend thanks to Alvaro Fernandez and Ravi Rao for the innumerable discussions regarding my research and their help in revising drafts of the thesis. Thanks are also due to Greg Lind and Powtawche Williams for their encouragement and support.

Most of all I would like to thank my parents. Without their support I would not have reached at this position. I would like to thank them for all their sacrifices and lessons, which have always been a guiding light. I would like to mention the special role of my sisters who have inspired me through this project. I would like to express utmost gratitude to the Almighty for his grace and strength.
Table of Contents

Abstract ii
Acknowledgments iv
List of Figures ix
List of Tables xiii
Nomenclature xiv

1 Introduction 1
1.1 Background ............................................. 3
1.1.1 Adaptivity ........................................... 4
1.1.2 Error Estimation .................................... 5
1.1.3 Object-Oriented Paradigm - Concepts and Terminology . 6
1.2 Thesis Objective ....................................... 10
1.3 Organization of This Manuscript ......................... 11

2 Literature Review 12
2.1 Adaptivity ............................................. 12
2.2 Error Estimators ....................................... 13
2.3 Object-Oriented Finite Element Analysis .................. 14

3 Adaptive Mesh Refinement Methodology 15
3.1 Accurate Gradient Recovery ............................ 15
3.1.1 Enhanced Superconvergent Patch Recovery Technique . 16
3.1.2 Implementation Details ........................................ 22
3.2 Error Estimation .................................................. 24
  3.2.1 Definition of Acceptable Solution ......................... 26
  3.2.2 Refinement Indicators ....................................... 26
  3.2.3 Measure of Effectiveness ................................... 27
3.3 P-Refinement .................................................... 27

4 Object-Oriented Design of the Program ....................... 29
  4.1 Object-Oriented Programming in Fortran 90 ................ 29
    4.1.1 Abstraction and Encapsulation ........................... 29
    4.1.2 Polymorphism ............................................ 31
    4.1.3 Inheritance ............................................. 34
    4.1.4 Safety Features ......................................... 35
  4.2 Object-Oriented Development Process ....................... 38
  4.3 Design of the Finite Element Program ..................... 38
    4.3.1 Problem Module .......................................... 39
    4.3.2 Adaptor Module .......................................... 39
    4.3.3 Solver Module ........................................... 39
    4.3.4 Domain Module ........................................... 41
    4.3.5 Element Module .......................................... 42
    4.3.6 Edge Module ............................................. 42
    4.3.7 Grid Point Module ....................................... 42
    4.3.8 Constraint Module ....................................... 43
  4.4 Discussion .................................................... 46

5 Data Structures .................................................. 49
5.1 Tiered Linked List Data Representation ........................................... 50
5.2 Singly Linked Element List ................................................................. 52
5.3 Singly Linked Edge List ................................................................. 52
5.4 Doubly Linked Circular Linked Grid Point List ................................. 54
5.5 Concept of an Iterator ................................................................. 54
5.6 Efficient Search Algorithm for Element and Grid Point Lists .......... 56
5.7 Performance Analysis ................................................................. 58
5.8 Discussion ................................................................. 59

6 Numerical Results ......................................................... 60
6.1 Element Validation ................................................................. 60
  6.1.1 First Patch Test ................................................................. 60
  6.1.2 Second Patch Test ................................................................. 62
6.2 Potential Flow Around a Cylinder ................................................. 64
  6.2.1 Analysis for the Potential Flow Test Case ...................................... 66
6.3 Babuška Test Case ................................................................. 72
  6.3.1 Analysis for the Babuška Test Case ...................................... 73
6.4 Strong Diagonal Gradient Test Case ........................................... 76
  6.4.1 Analysis for the Strong Gradient Test Case ...................................... 77
6.5 Further Tests ................................................................. 81
6.6 Discussion ................................................................. 82

7 Conclusions and Future Work ......................................................... 84

A Graphical User Interface Using Java ......................................................... 86
A.1 Choice of Java for the GUI ......................................................... 87
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.2 Combining Java and Fortran 90</td>
<td>88</td>
</tr>
<tr>
<td>A.3 Discussion</td>
<td>90</td>
</tr>
<tr>
<td>Bibliography</td>
<td>94</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Steps in an Adaptive Finite Element program. .................. 3
1.2 Illustrating the Notebook Class hierarchy. ..................... 9

3.1 Typical patch defined using the adjacent element approach: △ Gauss point; • nodal gradient values determined by recovery procedure. . . . 20
3.2 Typical patch defined using the surrounding element approach: △ Gauss point; • nodal gradient values determined by recovery procedure. 21
3.3 Corner element patch example. ............................... 23

4.1 Illustrating the program design. ............................... 40
4.2 The Problem, Adaptor, Solver and Domain classes ................ 41
4.3 The Element, Edge, Grid point and DirEcBc classes ............. 44

5.1 A singly linked list. ........................................... 50
5.2 A circular doubly linked list. ................................... 51
5.3 Tiered list data representation of a solitary, unconstrained linear rectangular serendipity element. ............................... 53
5.4 Relationship between a singly linked list and its iterator. ........ 55
5.5 A print routine illustrating the iterator traverse. ............... 56
5.6 Binary search type algorithm for the linked lists. ........................................ 57

6.1 Initial mesh for the patch tests. ................................................................. 61
6.2 Mesh after the second refinement iteration, for the second patch test case. ........ 62
6.3 Mesh after the fifth iteration for the second patch test case. .................. 63
6.4 Exact error versus degree of freedom of the system, for the second patch test. .......... 63
6.5 Potential flow around a cylinder problem. ................................................. 65
6.6 Potential flow finite element solution problem domain. .......................... 66
6.7 Initial (Q8 4X6) mesh used for the finite element solution, with 24 elements and 35 degree of freedoms. ......................................................... 67
6.8 Mesh for the third iteration for the potential flow test case (Q8 4X6), with combination boundary conditions. ........................................... 67
6.9 Mesh for the sixth iteration for the potential flow test case (Q8 4X6), with combination boundary conditions. ........................................ 68
6.10 Convergence rates for runs conducted using different boundary conditions, measured in terms of norm of estimated error for a q8 4X6 mesh. ........................................... 68
6.11 Convergence rates for runs conducted using different boundary conditions, measured in terms of norm of exact error for a q8 4X6 mesh. 69
6.12 Comparison of convergence rates for potential flow test case between Q8 4X6 mesh and Q8 16X24 mesh, measured in terms of norm of estimated error. ................................. 70
6.13 Variation of the effectivity index for Potential flow test case (Q8
16x24 mesh). ........................................... 71
6.14 Gradient mesh plots for the Babuška test case. ...................... 73
6.15 Initial finite element mesh for Babuška test case, a 4-node
isoparametric mesh. ...................................... 74
6.16 Mesh for Babuška test case, for the third refinement iteration (Q4
20X40). .................................................. 75
6.17 Mesh for Babuška test case, for the fifth refinement iteration (Q4
20X40). .................................................. 75
6.18 Convergence for Babuška test case (Q4 20X40). ..................... 76
6.19 Gradient contours for the strong diagonal gradient test case. ...... 77
6.20 Initial finite element mesh for strong diagonal gradient test case, a
4-node isoparametric mesh. ................................ 78
6.21 Mesh for strong diagonal gradient test case, for the third refinement
iteration (Q4 10X10). .................................... 78
6.22 Mesh for strong diagonal gradient test case, for the fifth refinement
iteration (Q4 10X10). .................................... 79
6.23 Convergence for strong diagonal gradient test case, Q4 10X10. .... 79
6.24 Variation of the effectivity index for strong diagonal gradient test
case Q4 10x10 mesh. .................................... 80
6.25 Convergence tests for Equilibrium Residual enhancement and odd v/s
even degree polynomial degree, using potential flow test case on a Q8
16X24 mesh with combination bcs. .......................... 82
A.1 The Java GUI and the Fortran 90 program combine to form a new application. .................................................. 89
A.2 Java GUI screen shot. .................................................. 91
A.3 Screen shot of ancillary windows. ................................. 92
List of Tables

4.1 Point class demonstrating various Object-oriented concepts, declarations. ........................................ 31
4.2 Point class continued, methods. ........................................ 32
4.3 Illustrating the Fortran 90 array features. ......................... 33
4.4 Example of Inheritance by Composition .......................... 34
4.5 Example of Inheritance using Subtyping, declarations .......... 36
4.6 Example of Inheritance using Subtyping continued, methods. .. 37
**Nomenclature**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>polynomial expansion coefficients vector</td>
</tr>
<tr>
<td>$M$</td>
<td>least squares problem design matrix</td>
</tr>
<tr>
<td>$g$</td>
<td>least squares problem right-hand side vector</td>
</tr>
<tr>
<td>$C$</td>
<td>material constitutive matrix, identity matrix</td>
</tr>
<tr>
<td>$\hat{e}_m$</td>
<td>mean allowed element error contribution</td>
</tr>
<tr>
<td>$e$</td>
<td>error vector</td>
</tr>
<tr>
<td>$|e|$</td>
<td>error in the energy norm</td>
</tr>
<tr>
<td>$e_e$</td>
<td>strain error vector</td>
</tr>
<tr>
<td>$e_\sigma$</td>
<td>exact gradient error vector</td>
</tr>
<tr>
<td>$|e_\sigma|$</td>
<td>exact gradient error in the energy norm</td>
</tr>
<tr>
<td>$e_\sigma^*$</td>
<td>approximate gradient error vector</td>
</tr>
<tr>
<td>$|e_\sigma^*|$</td>
<td>approximate gradient error in the energy norm</td>
</tr>
<tr>
<td>$|e_\sigma^*|_i$</td>
<td>approximate gradient error in the energy norm for an element</td>
</tr>
<tr>
<td>$|e_\sigma^*|_N$</td>
<td>approximate gradient error in the energy norm for a neighbor</td>
</tr>
</tbody>
</table>
\( f \)  
function

\( f \)  
forcing vector

\( h \)  
length of an element edge in parametric space

\( H \)  
interpolation function

\( k \)  
constant

\( L \)  
linear differential operator

\( m \)  
number of elements, number of sampling points

\( M \)  
coefficient

\( n \)  
number of least squares problem basis functions

\( N \)  
coefficient

\( N \)  
set of all elements containing a node

\( N \)  
interpolation function vector

\( p \)  
serendipity element edge degree

\( p_{old} \)  
previous serendipity element edge degree

\( p_{new} \)  
new serendipity element edge degree

\( P \)  
general serendipity element corner node, coefficient

\( P \)  
polynomial expansion terms vector
$r$  unit parametric coordinate, radial polar coordinate

$R$  radius of cylinder

$u$  flow velocity in the x-direction

$u$  field variable vector

$\bar{u}$  nodal field variable vector

$u_h$  finite element solution field variable vector

$v$  flow velocity in the y-direction

$v_r$  flow velocity in the radial direction

$v_\theta$  flow velocity in the tangential direction

$(x_i, y_i)$  Cartesian coordinates of sampling point or node

$(x_m, y_m)$  Cartesian coordinates of moved boundary node

$X_i$  least squares problem basis function

Greek Letters

$\Delta$  parametric distance between analysis nodes on edge

$\Delta_{new}$  new parametric distance between analysis nodes on edge

$\eta$  natural parametric coordinate, relative percentage error

$\bar{\eta}$  maximum permissible relative percentage error

$\theta$  effectivity index, angular polar coordinate
\( \xi \)  
\text{natural parametric coordinate}

\( \xi_i \)  
\text{element refinement indicator}

\( \sigma \)  
\text{exact gradient vector}

\( \sigma_h \)  
\text{finite element solution gradient vector}

\( \sigma^* \)  
\text{approximate gradient vector}

\( ||\sigma^*|| \)  
\text{approximate gradient vector in the energy norm}

\( \ddot{\sigma}^* \)  
\text{recovered nodal gradient vector}

\( \sigma_p^* \)  
\text{gradient approximation polynomial expansion}

\( \phi \)  
\text{constraint, velocity potential}

\( \Omega \)  
\text{problem domain}
Chapter 1

Introduction

Most engineering problems are solved by simulation of mathematical models, which are governed by partial differential equations. These partial differential equations are too complex to solve analytically. Hence, they are solved numerically, using methods like finite difference, finite element and finite volume. The finite element method, due to its diversity and flexibility, is the most popular numerical method used to solve these equations.

Since its birth in the 1940s the finite element method has matured and is now employed in a wide range of problems, from solid mechanics (elastic, plastic, static and dynamic) to heat transfer, fluid mechanics, acoustics, and electromagnetism. The main advantage of finite element analysis is its ability to model complex and arbitrary shaped domains. It also supports modeling of general boundary conditions and non-homogeneous materials.

In most industries finite element analysis is performed on an already designed product. Consequently, the analysis may only be used to verify the design. This comes too late in the development stage, rendering it unprofitable and time consuming to modify the design according to the results of the analysis. To reduce the design costs and product cycle time, finite element analysis needs to be integrated early in the design process. This integration requires further development of various technologies including automatic mesh generation, automatic error control via self-
adaptive methods, design optimization based on results of the analysis and software interoperability.

Traditionally, engineering judgment has been the sole criterion used to arrive at an accurate and cost-effective solution. This makes the refinement process error-prone and slow. As a result, self-adaptive finite element techniques have been developed. A self-adaptive finite element method is driven by the use of an error estimator, which provides numerical estimates of the error in the solution. The flowchart in Fig. 1.1 outlines the steps involved in an adaptive finite element method.

In finite element programming issues of code re-usability, maintainability and easy extensibility along with efficiency are of prime importance. Traditionally, finite element methods have been programmed procedurally in Fortran 77. In a procedural style, also termed as water-fall technique [21], a program is viewed as set of algorithms. Therefore, the program depends closely on the algorithm being implemented, creating a rigid framework where a change becomes costly and sometimes unmanageable. A different programming paradigm has emerged from its relative isolation in the past decade. Object-oriented programming is capable of overcoming some of the limitations of procedural programming. This will be demonstrated in subsequent chapters.

The basic aim of developing a finite element package is to use it to analyze and solve physical problems. Typically, the task of performing post-processing of the results obtained from the finite element program is tedious, and has to be repeated every time the program is run. This can be automated using a graphical user interface (GUI) which drives the entire program and performing both pre- and post-processing. Among the languages supporting GUI development- (C++, Java, Matlab and X-windows), Java offers additional benefit of being (more) platform independent
and easily extensible to run across a network. The GUI and the analysis code need not be programmed in the same language. It will be shown in Appendix A, a Java GUI and a Fortran 90 analysis code can be interfaced effectively.

1.1 Background

This section will introduce the basic concepts used in this thesis.
1.1.1 Adaptivity

Adaptive methods are employed to minimize the discretization error in a finite element solution. Out of many adaptive approaches, one way is to simply regenerate the mesh using the computed errors, to get a more accurate discretization. This method is computationally inefficient. Another approach is to refine an already generated mesh. In case of self-adaptive methods the refinement is performed automatically on the basis of the results provided by the error estimator. The different types of refinement techniques are h-refinement, p-refinement, r-refinement and mixed approaches. The approach to be applied depends on the smoothness of the solution and geometry of the problem domain. The hp-method is the most efficient approach.

**H-refinement** subdivides the selected element into smaller elements. Usually the length of the side of element ('h') is reduced by a factor of two. Though accurate for many problems, this method can lead to badly distorted elements. When selective refinement is employed, 'irregular nodes' are produced, which must be accounted by using constraint equations to preserve inter-element continuity. Complex data structures, such as trees, are needed to implement h-refinement.

**P-refinement** reduces the error in the solution by increasing the polynomial order ('p') of the interpolating functions, with the same geometric element. The advantage of p-refinement is that the higher order elements can better approximate the curved, irregular boundaries and are less sensitive to aspect ratio and skew. This makes p-refinement a good candidate for use with structural shape optimization. However a very high polynomial order in the interpolation function can cause oscillations in the solution. Special consideration needs to be taken at the boundary to preserve
geometric properties, boundary conditions and loading conditions as the solution adapts.

**R-refinement** relocates the position of nodes in a finite element mesh. It is used as a secondary method to improve the existing discretization in a conceptually simple way while still reducing the error. The obtained accuracy is limited, as the method does not introduce new degrees of freedom in the solution. Issues of stability and validity of the mesh need to be considered while employing this method.

Mixed approaches include schemes employing a combination of above mentioned methods. Of these, the **hp-method** has been shown to yield exponential rates of convergence in the energy norm. A set of heuristics is needed to arrive at an optimal strategy for hp-refinement. Maddox [27] used **rp-method**, which is computationally simpler, and still effective.

### 1.1.2 Error Estimation

Success of any adaptive scheme depends upon the reliability of the error estimator used. Generally, there are two types of error estimates: *a priori* and *a posteriori*. *A priori* estimates give advance notice on how a scheme may behave in worst case, by utilizing the characteristics of the solution such as smoothness. They are an important tool for the theoretical study of the finite element method but cannot provide practical results. *A posteriori* estimates use the information obtained during the solution process to drive the adaptive process and provide a basis for terminating a calculation when a target error level is attained. The majority of research work has naturally concentrated on *a posteriori* method.

*A posteriori* estimators may be classified into two families: residual-type and averaging-type. Residual-type estimators use the residual of the finite element so-
olution as a source term in local boundary value problems associated with the error. Averaging-type estimators are based on recovery of the superconvergent gradients obtained by averaging values computed at the superconvergent points. This thesis will utilize an averaging type error estimator originally proposed by Zienkiewicz and Zhu [45, 46]. A suggested enhancement to this error estimator will be tested, which involves addition of a residual term, suggested by Blacker et al. [12].

1.1.3 Object-Oriented Paradigm - Concepts and Terminology

Object-oriented programming can be considered as viewing a program, as an analog model of real-world phenomenon. A program thus builds an environment containing objects (such as beams, floors) with attributes analogous to real-world counterparts (such as weight, lengths and properties) and analogous behavior (deformation under loads). An object can also be an abstract entity (such as matrix, vector etc.). The rest of this section will explain the concepts and terminology of object-oriented design and programming.

1.1.3.1 Class

A class is an abstract data type definition. An instance of a class is known as object. A class is in fact a new type, like the classical data types integer, real, character. A class is a self-contained entity, composed of data and methods (procedures, functions) that are used to operate on that data. Classes arise from the developers abstraction of the problem in question. For example, an engineer interested in solving linear algebra problems realizes that one of the main concepts is that of the matrix. Hence, he can
abstract the concept of a matrix such that he creates a matrix class, from which matrix objects can be instantiated for later manipulation. It follows very naturally:

\[
\begin{align*}
\text{integer } & a, b \\
\text{matrix } & c\_\text{matrix}, d\_\text{matrix}
\end{align*}
\]

The attribute of object \(c\_\text{matrix}\) can be number of rows, number of columns, and the methods can be \(\text{ComputeDet}()\), \(\text{Invert\_Matrix}()\), \(\text{GetValue}(\text{row}, \text{column})\).

### 1.1.3.2 Abstraction and Encapsulation

**Abstraction** has been defined by Booch [13] as

"An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer."

A good abstraction emphasizes details that are significant to the reader or user and suppresses details that are, at least for the moment, immaterial or diversionary. This restricts the proliferation of the complexity in the program. For example, when building a finite element application an important class needed would be an element class. In order to generate the element stiffness matrix, the user only needs to know the interface of the method \(\text{generate\_Stiffness}\) while the details of its implementation are hidden.

**Encapsulation** focuses on implementation to build abstractions; thus abstraction and encapsulation are complementary concepts. Encapsulation is often achieved through information hiding, which is the process of hiding the details that do not contribute to the essential characteristics of the object. In practice, this means each class has two parts, an interface (function signatures, arguments the function takes
and the return types) and an implementation (of the methods). The interface enables the exchange of information between different objects. Keeping the interface the same, we can change the implementation without affecting other objects, thus removing interdependence from the application. In the previous example of the element class, changing the method for generating the stiffness matrix will not affect the user if the interface remains unchanged.

1.1.3.3 Inheritance

One of the more powerful features of object-oriented programming is the capability of sharing attributes and methods. By means of inheritance, new object types can be build up from existing types. Moreover, the new types obtain the capabilities (methods) and the knowledge (attributes) to do everything the parent or generating objects can do without additional coding. This leads to class hierarchies, a tree like structure, which removes the need of re-coding the common functionality. Only new data and new methods need to be defined providing a flexible and extensible environment. A good example in [29], is that of a red notebook and a blue notebook. They are both notebooks, sharing all their characteristic behavior and properties except for color. In the object-oriented approach, we can have a base Class Notebook with all the common attributes and methods needed to manipulate it. The red notebook and blue notebook both inherit from Notebook class. This is shown in Fig. 1.2. The Red Notebook and Blue Notebook have an extra attribute color to store the color of the notebook.
Figure 1.2: Illustrating the Notebook Class hierarchy.

1.1.3.4 Polymorphism

In the notebook example described above there can be similar methods, in both red notebook and blue notebook, for example the `What_Is_Your_Color()` method. In common procedural languages function or subroutine names are unique, making data and functions independent. On the other hand, in object-oriented languages data and functions (methods) can be related in an object. Object-oriented languages allow the use of identical functions names for identical methods in different classes.
This is how function (methods) and data (attributes) can be combined within a class, Fig. 1.2.

Object-oriented software can be developed in any high level language, even in C, FORTRAN 77. However languages like Ada, C++, Fortran 90, Java and Smalltalk have enhanced features that make object-oriented programming natural and practical. These languages differ in their implementation of various object-oriented concepts.

1.2 Thesis Objective

The objective of the thesis is to develop a self-adaptive finite element package which is user-friendly and researcher-friendly. This research is a continuation of an earlier work done by Maddox [27]. Maddox developed an rp-adaptive package, which was designed and programmed in a procedural style. Here, we develop an object-oriented version of the code for easier extensibility and maintainability. The versatility of the object-orientation is demonstrated by the ease with which different solvers can be used and different problems are solved. The setup can be easily extended to lower or higher dimensions, to use different types of elements, and to solve different types of problems.

This work explores enhancements to the superconvergent patch recovery technique to achieve more accuracy in the recovered gradients. The analysis capabilities of the package have been increased to solve a larger family of problems. Special attention has been paid to issues of efficiency to compensate for the necessary computational expense of additional function calls in this object-oriented setup. Efficient data-structures designed in an object-oriented style have been developed. The language used is Fortran 90, which provides support for object-oriented programming and offers features like advanced array syntax and dynamic memory management, discussed in
Chapter 4. The package is driven by a graphical user interface (GUI). The GUI, which is written in Java, encapsulates the Fortran 90 analysis code. It has visualization capabilities to perform preprocessing and post-processing. Thus saving a significant amount of research time by automating the process of analyzing the results generated by the program.

1.3 Thesis Accomplishments

The major accomplishments of this thesis are been summarized here:

- Development of an object-oriented P-adaptive finite element program.
- Developing a Java GUI for the program.
- Testing the variation in the algorithm for the refinement process.
- Implementation of a binary search type algorithm for the link lists.

1.4 Organization of This Manuscript

Chapter 1 is a brief introduction to self-adaptive finite elements and object-oriented programming paradigm. Chapter 2 surveys relevant literature. Chapter 3 discusses the adaptive strategies employed in this thesis. Chapter 4 describes object-oriented programming in Fortran 90 language along with the object-oriented design of this program. Chapter 5 outlines the data structures used. Chapter 6 validates this program with the help of numerical studies and some benchmark problems. Chapter 7 is a closure to this work. Appendix A explains about the graphical user interface development.
Chapter 2

Literature Review

This chapter reviews the previous work done leading to this prototype effort. It has been divided into individual sections, each focusing on a separate area.

2.1 Adaptivity

The goal of adaptive analysis is to seek an optimal combination of accuracy and computational cost to obtain the solution. The h-adaptive scheme, which adds more elements to the domain to achieve a more accurate discretization [28, 36], is the most commonly used adaptive scheme. An important consideration in a h-refinement scheme is the control of element shape after several levels of refinement. Traditional p-adaptive schemes [32, 42, 43], typically use hierarchical finite elements. A hierarchical finite element is an efficient transition element, as the element stiffness matrix need not be fully recomputed as the element adapts. P-adaptivity using hierarchical elements is also implemented in some commercial codes like Phlex [33]. R-adaption is not as commonly used. Much of the work done in r-adaption tries to arrive at an optimal distribution of grid points resulting in a least upper bound on the potential energy. Maddox [27] in his effort surveys the efforts in r-adaption method.

Hp-adaptive schemes have been shown to give exponential rates of convergence in the energy norm, with respect to the number of degrees of freedom. It is an optimal choice despite the additional computational and database complexities involved.
Major developments in hp-method, in the last decade follows from the work of Oden et al. [30] and Babuška et al. [9]. The software package, Phlex [33] was first to introduce hp-adaption commercially.

This effort is based on rp-adaptive proposed by Maddox [27] using serendipity elements. This rp scheme has an advantage over conceptually difficult traditional p-refinement approaches using hierarchical elements.

2.2 Error Estimators

The concept of a posteriori error estimation was first introduced in 1978 [2, 3]. Residual estimators though rigorous are computationally expensive and not trivial to implement. Babuška has done an extensive review [6, 10] on the development of residual estimators. The remainder of this section will focus on averaging-type estimators.

Zienkiewicz and Zhu [44] first introduced an averaging-type estimator, which was simple and inexpensive to compute. As the reliability of this approach was highly dependent on the accuracy of the recovered gradients, the proposed method suffered because of the inadequacy of the recovery process. Subsequently Zienkiewicz and Zhu proposed a more effective superconvergent patch recovery scheme [45, 46] to recover the solution gradients. Many variations [12, 10] have been suggested for the original error estimation scheme and the superconvergent patch recovery method developed by Zienkiewicz and Zhu. Oñate et al. [31] showed that considering the global error indicator in the error estimation process, along with the local error indicator originally proposed by Zienkiewicz and Zhu, improved efficiency. Lee et al. [25] introduced an implicit refinement method to satisfy the global error conditions.
2.3 Object-Oriented Finite Element Analysis

The idea of object-oriented programming can be traced back to Simula 67 language. It was realized that object-oriented techniques could solve the problems of code re-usability and maintainability, encountered in engineering software development. Consequently, there has been considerable interest, over the last decade, in this alternate philosophy of programming. Rehal et al. [35] and Forde et al. [20] first applied object-oriented principles to finite element programming. Gang [21] in his effort reviews early progress in object-oriented finite elements. Zimmermann et al. [47, 17, 18] in a series of papers presented an object-oriented finite element toolkit, implemented in Smalltalk and C++. Dubois-Pèlerin et al. [16] proposed further modularity by delineating two kinds of behavior of a finite element software: analysis type (solvers used, static or dynamic analysis) and domain information (element, grid Points, boundary conditions).

Liu et al. [26] introduced an object-oriented implementation of adaptive finite element and finite volume methods. The program was implemented in C++, performed h-refinement and offered a choice of solvers- direct, iterative or SOR. While much of the previous work is in C++, Smalltalk and Pascal, there is no reported implementation in Fortran 90.
Chapter 3

Adaptive Mesh Refinement Methodology

The adaptive mesh refinement method employed here consists of recovering accurate gradients, using these recovered gradients for estimating error in the finite element solution, and utilizing these estimated error values as an indicator for adapting (refining/coarsening) the solution interpolation functions. This section will explain these aspects of the adaptive mesh refinement process.

3.1 Accurate Gradient Recovery

Approximate solution of second order partial differential equations by the Galerkin finite element method typically results in derivatives which are discontinuous across the element boundaries. Though these discontinuities can be smoothed out by averaging, the resulting gradients are still least accurate at grid points and element boundaries where accuracy is usually desired. Various local and global projection schemes to recover accurate gradients have been used. These schemes are either sensitive to the element type or too expensive and difficult to implement.

A powerful gradient recovery method known as the superconvergent patch recovery method, was developed by Zienkiewicz and Zhu [46]. The finite element gradients are most accurate at certain element locations known as gauss points or superconvergent points. The Superconvergent patch recovery scheme (SPR) uses the gradient values at the superconvergent points to recover the highly accurate nodal gradients. In this
method a continuous polynomial function of gradients is constructed on a patch of elements. The patch defined by Zienkiewicz and Zhu consists of elements surrounding the node for which the recovery is being done. This original scheme can have ill-conditioned extraction equations for higher order polynomials. A newer element based patch definition reduces the conditioning problem and is more suitable for the irregular mesh arising in p-adaptive schemes. This effort uses an enhanced version of the superconvergent patch recovery scheme with an element driven patch definition.

### 3.1.1 Enhanced Superconvergent Patch Recovery Technique

The method can be illustrated by considering a linear elliptic problem of this form

\[
\varepsilon = Du \in \Omega 
\]

\[
\sigma = C\varepsilon \in \Omega 
\]

\[
D^T \sigma + b = 0 \in \Omega 
\]

In a combined form,

\[
Lu + b = 0 
\]

where,

\[
L = D^T CD 
\]

In the case of two dimensional Poisson equation the above are,

\[
\varepsilon = [\varepsilon_x, \varepsilon_y]^T 
\]

\[
D = \begin{bmatrix} \partial / \partial x \\ \partial / \partial y \end{bmatrix} 
\]

\[
C = \text{Material matrix} 
\]

u, b are scalars.

The boundary conditions are,
\[ u = \hat{u} \quad \text{on} \quad \Gamma \]  \hfill (3.9)

The finite element solution, denoted by \( u_h \) can be written as,
\[ u_h = N\hat{u} \in \Omega^e \]  \hfill (3.10)

Where \( N \) represents the finite element solution interpolation functions and \( \hat{u} \) denotes the nodal values of the field variables. The solution gradients are,
\[ \sigma_h = CDu_h \in \Omega^e \]  \hfill (3.11)

These gradients are discontinuous across the element boundaries. The superconvergent patch recovery method utilizes the fact that finite element gradients are most accurate at the gauss points, and tries to find nodal gradient values \( \bar{\sigma}^* \), such that the smoothed, continuous gradient field \( \sigma^* \), defined as
\[ \sigma^* = N\bar{\sigma}^* \]  \hfill (3.12)

is more accurate than \( \sigma_h \). This is accomplished by assuming that the nodal gradient values, \( \sigma^* \) belong to a polynomial expansion, \( \sigma_p^* \) which is defined over a patch of elements surrounding the concerned element. This polynomial \( P(\xi, \eta) \) is evaluated on the non-dimensionalized patch domain.
\[ \sigma_p^* = P(\xi, \eta)a \in \Omega^p, \Omega^p \cup \Omega^e \]  \hfill (3.13)

Where \( \sigma_p^* \) is a vector of all the derivative components and \( a \) is a set of unknowns. This prototype uses the same polynomial function \( P \) for each derivative component. Different derivative components are obtained using a different set of coefficient \( a \). For example, in the case of potential flow there will be two derivative components written as,
\[ \sigma_{p_x}^* = P(\xi, \eta)a_x \]
\[ \sigma_{p_y}^* = P(\xi, \eta)a_y \]
The gradients thus obtained are more accurate and smooth. The choice of this polynomial will be discussed in a later section. The determination of the unknown coefficients $a$ can be done by minimizing the merit function $F(a)$, where $F(a)$ is,

$$
F(a) = \frac{1}{2} \sum_{i=1}^{m} (\sigma_h(x_i, y_i) - \sigma_p^*(x_i, y_i))^2 + \frac{1}{2} \phi_1 \|D^T \sigma_p^*(x, y) + b(x, y)\|_{\Omega_p}^2
$$

(3.14)

With, $0 \leq \phi_1 \leq 1$ being a weight chosen by the analyst. The first term, originally proposed by Zienkiewicz and Zhu, is the least square fit of the expansion $P$ to the superconvergent gradient values. It is given by

$$
\frac{1}{2} \sum_{i=1}^{m} (\sigma_h(x_i, y_i) - \sigma_p^*(x_i, y_i))^2 = \frac{1}{2} \sum_{i=1}^{m} (\sigma_h(x_i, y_i) - P(x_i, y_i)a)^2
$$

(3.15)

The second term, proposed by Blacker et al. [12], is the norm of the equilibrium residual. The gradients recovered by using only the least square term do not satisfy the equilibrium equation 3.3. The second term introduces a penalty on the recovered derivatives for violating the equilibrium equation,

$$
\frac{1}{2} \phi_1 \|D^T \sigma_p^*(x, y) + b(x, y)\|_{\Omega_p}^2 =
\frac{1}{2} \phi_1 \int_{\Omega_p} (D^T P(x, y)a + b(x, y))^T (D^T P(x, y)a + b(x, y)) d\Omega
$$

(3.16)

The equilibrium residual term has been shown to significantly improve the accuracy of the derivatives recovered [12, 25]. The necessary condition for minimization of $F$ gives,

$$
\frac{\partial F}{\partial a} = 0 \Rightarrow (M_1 + M_2)a = g_1 + g_2
$$

(3.17)

where,

$$
M_1 = \sum_{i=1}^{m} P(x_i, y_i)^T P(x_i, y_i)
$$

(3.18)

$$
M_2 = \phi_1 \int_{\Omega_p} (D^T P(x, y))^T (D^T P(x, y))d\Omega
$$

(3.19)
\[
g_1 = \sum_{i=1}^{m} P(x_i, y_i)^T (\sigma_h(x_i, y_i)) \quad (3.20)
g_2 = -\phi_1 \int_{\Omega_p} (D^T P(x, y))^T b(x, y) d\Omega. \quad (3.21)
\]

The resulting system of equations is of the form,

\[
Ma = g \quad (3.22)
\]

If there are two components of gradients (for example, potential flow discussed earlier) \(\sigma_x, \sigma_y\), there will be two sets of coefficients \(a\) and consequently, two sets of system of equations.

\[
(M_1 + M_{2, x}) a_x = g_{1, x} + g_{2, x} \quad (3.23)
\]
\[
(M_1 + M_{2, y}) a_y = g_{1, y} + g_{2, y} \quad (3.24)
\]

The \(M_1\) matrix is same for all components. These equations are solved to recover the unknown coefficients \(a\). Two gradient recovery methods were explored. The first is the original Zienkiewicz and Zhu recovery scheme which involves only the first term of equation 3.14. The second method considers the whole of equation 3.14. The comparison of these schemes is discussed in numerical results chapter (6).

### 3.1.1.1 Choice of Patch

This effort uses element-driven patches for the gradient recovery process, in order to circumvent some of the problems presented by the use of general serendipity elements. As a result, two element-driven patches are studied. The first scheme defines the patch consisting of the central element and all elements sharing a common edge with it. Fig. 3.1 shows a typical patch formed by this approach. The second scheme defines the composition of the patch as the central element and all its surrounding elements. Fig. 3.2 shows the second approach; it has access to more sampling point information, but is somewhat more expensive,
Figure 3.1: Typical patch defined using the adjacent element approach:
- △ Gauss point; ○ nodal gradient values determined by recovery procedure.

Both of these approaches do not require special treatment for the boundary corner grid points. In the Zienkiewicz and Zhu vertex driven approach, corner grid points may produce an under-determined least square problem when solving for the coefficients $a$. The present definitions of the patch provide access to a sufficient number of sampling points.
Figure 3.2: Typical patch defined using the surrounding element approach: \( \Delta \) Gauss point; \( \circ \) nodal gradient values determined by recovery procedure.

### 3.1.1.2 Choice of Gradient Polynomial Expansion

Three different polynomial expansions were tried in this effort.

- Polynomial expansion Method A: The polynomial used for recovering nodal gradient values is a complete polynomial of the same degree as the central patch element's highest edge degree.
• Polynomial expansion Method B: The polynomial used for recovering nodal gradient values is a complete polynomial of one degree higher than the central patch element’s highest edge degree.

• Polynomial expansion Method C: This approach uses the corner node interpolation function for a homogeneous serendipity element having edges of the same order as the central patch element’s highest order edge.

An example is shown in Fig. 3.3 The element in upper left corner is the central patch element, whose highest degree edge is quadratic. The darkened regions shows the patch for this element. The different methods result in following polynomial choices:

\[
P_A = \begin{bmatrix} 1 & x & y & x^2 & xy & y^2 \end{bmatrix} \quad (3.25)
\]

\[
P_B = \begin{bmatrix} 1 & x & y & x^2 & xy & y^2 & x^3 & x^2y & xy^2 & y^3 \end{bmatrix} \quad (3.26)
\]

\[
P_C = \begin{bmatrix} 1 & x & y & x^2 & xy & y^2 & x^2y & xy^2 \end{bmatrix} \quad (3.27)
\]

These polynomial expansions can be potentially problematic along lower degree edges due to the presence of some higher order terms.

3.1.2 Implementation Details

The process of recovering gradients with this enhanced superconvergent patch recovery scheme can be divided into the following steps. First the patch domain is defined. Then the sampling point data (gradient values at the gauss point) is used to formulate the least square problem. If indicated by the user, the equilibrium residual term is evaluated by carrying out integrations over the patch elements. The resulting system of equations is solved to determine the unknown coefficients. These coefficients are used to evaluate the superconvergent gradient at the nodes.
Figure 3.3: Corner element patch example.

Constructing the minimization problem requires repeated access to element sampling point data. Thus for efficient implementation, the sampling point data of each element is stored in a link list. Each sampling point data list node stores the information of one sampling point. This list is included in the element class, which will be explained in Chapter 4. The system of equations is solved using singular value decomposition method (SVD), as this method is robust against round-off error and ill-conditioned systems.
3.2 Error Estimation

The Zienkiewicz-Zhu error estimator used in this prototype uses the energy norm to quantify the error in a finite element solution. In the specific case of linear elasticity, the error norm can be defined as,

$$ ||e|| = \left( \int_{\Omega} e_u^T L e_u \, d\Omega \right)^{1/2} $$  \hspace{1cm} (3.28)

The definition of the $L$ operator is restated here for clarity:

$$ L = D^T C D $$

where,

$D$ = Differential operator.

$C$ = Material matrix.

$e_u$ = Difference between exact and calculated displacements.

$= u - u_h$

Thus, the error norm can be written out as

$$ ||e|| = \left( \int_{\Omega} e_u^T D^T C De_u \, d\Omega \right)^{1/2} $$

$$ = \left( \int_{\Omega} (De_u)^T C (De_u) \, d\Omega \right)^{1/2} $$

$$ = \left( \int_{\Omega} e_\varepsilon^T C e_\varepsilon \, d\Omega \right)^{1/2} $$  \hspace{1cm} (3.29)

where

$$ e_\varepsilon = De_u, $$  \hspace{1cm} (3.30)

and using the constitutive relation,

$$ e_\varepsilon = C^{-1} e_\sigma $$  \hspace{1cm} (3.31)
the error norm is,
\[ \| e_\sigma \| = \left( \int_\Omega e_\sigma^T C^{-1} e_\sigma \, d\Omega \right)^{1/2}, \]  
(3.32)

The error in stress \( e_\sigma \) is defined as,
\[ e_\sigma = \sigma - \sigma_h. \]  
(3.33)

Here \( \sigma \) is the exact stress value and \( \sigma_h \) the approximate stress value. Since we don’t know the exact solution, Zienkiewicz and Zhu proposed the substitution of superconvergent recovered gradients in place of exact gradients to obtain an approximation to error. Thus the error measure becomes
\[ e_\sigma \approx e_\sigma^* = \sigma^* - \sigma_h, \]  
(3.34)

and the approximate energy norm,
\[ \| e^* \| = \left( \int_\Omega e_\sigma^*^T C^{-1} e_\sigma^* \, d\Omega \right)^{1/2} \]  
(3.35)

is used as the measure of error in the solution. A more practical way of expressing the error is in terms of relative percentage error \( \eta \)
\[ \eta = \left( \frac{\| e_\sigma^* \|^2}{\| \sigma^* \|^2 + \| e_\sigma^* \|^2} \right)^{1/2} \]  
(3.36)

The error in the energy norm can be defined over the entire problem domain or the local element domain. Consequently, global and elemental percentage errors can be obtained. The global norms are constructed as summation of the elemental norms.
\[ \| e_\sigma^* \|^2 = \sum_{i=1}^m \| e_\sigma^* \|^2_i, \]  
(3.37)
\[ \| \sigma^* \|^2 = \sum_{i=1}^m \| \sigma^* \|^2_i, \]  
(3.38)

where \( m \) is the number of elements in the domain. The elemental norms are defined as,
\[ \| e_\sigma^* \|_i = \left( \int_{\Omega_i} e_\sigma^*^T C^{-1} e_\sigma^* \, d\Omega_i \right)^{1/2} \]  
(3.39)
\[ \|\sigma^*\|_i = \left( \int_{\Omega_i} \sigma^* C^{-1} \sigma^* \, d\Omega_i \right)^{1/2} \]  
(3.40)

### 3.2.1 Definition of Acceptable Solution

The acceptable solution is reached when the following two conditions are satisfied, as pointed by Oñate [31].

- **Global Error Condition**: Global percentage error should not be greater than a user specified maximum percentage error value \( \bar{\eta}_{\text{perm}} \).
  
  \[ \eta \leq \bar{\eta}_{\text{perm}}. \]  
  (3.41)

- **Local Error Condition**: Local relative percentage error of all the elements in the domain must satisfy the relationship,
  
  \[ \|\varepsilon^*\|_i \leq \bar{\varepsilon}_{\text{avg}}. \]  
  (3.42)

Since optimal convergence rates of an adaptive scheme can be achieved by equally distributing the error among elements. So, \( \bar{\varepsilon}_{\text{avg}} \) is defined as,

\[ \bar{\varepsilon}_{\text{avg}} = \bar{\eta}_{\text{perm}} \cdot \left( \left( \|\sigma^*\|^2 + \|\varepsilon^*_\sigma\|^2 \right) / m \right)^{1/2} \]  
(3.43)

### 3.2.2 Refinement Indicators

The global error indicator is used to determine if another iteration of the program needs to be performed. That is,

If \( \eta > \bar{\eta}_{\text{perm}} \) \( \Rightarrow \) perform another iteration.
Local refinement indicator targets the individual elements to be refined,

\[ \xi_i = \frac{\| e_r^i \|}{\bar{e}_{avg}} \]  \hspace{1cm} (3.44)

where \( \xi_i > 1 \) signals refinement, \( \xi_i < 1 \) signals derefinement and \( \xi_i = 1 \) signals no change. This way both local and global error requirements are satisfied.

### 3.2.3 Measure of Effectiveness

The performance of an error estimator is measured by the effectivity index,

\[ \theta = \frac{\| e_r^i \|}{\| e_e^i \|} \]  \hspace{1cm} (3.45)

which is the ratio of estimated error to exact error. Zienkiewicz and Zhu [46] have proved that if the effectivity index approaches unity as the polynomial order in the mesh tends to infinity, the error estimator is asymptotically exact and will converge to the exact error. This index will be analyzed in Chapter 6, for various test cases.

### 3.3 P-Refinement

It was pointed out earlier that computation of the error for each element allows a refinement indicator to be derived. Indeed, the ratio

\[ \xi_i > 1 \]  \hspace{1cm} (3.46)

targets the elements to be refined. It's value can also be used to determine the required polynomial order change to achieve an assumed rate of convergence in the solution. The standard rate of convergence is \( O(\Delta^p) \), where \( \Delta \) is the current distance between analysis nodes on each edge of the element. Mathematically, \( \Delta \) is defined as

\[ \Delta = \frac{h}{p} \]  \hspace{1cm} (3.47)
where $h$ is the length of the element edge in parametric space and $p$ is the degree of a typical edge in the serendipity element. A new predicted distance between analysis nodes should be

$$
\Delta_{\text{new}} = \frac{\Delta_{\text{old}}}{\xi^{1/p_{\text{old}}}} = \frac{h}{p_{\text{old}} \xi^{1/p_{\text{old}}}}
$$

which can be rewritten as,

$$
\Delta_{\text{new}} = \frac{h}{p_{\text{new}}}
$$

where $p_{\text{new}} = p_{\text{old}} \xi^{1/p_{\text{old}}}$ Assuming a superconvergence in the solution this can be modified as,

$$
p_{\text{new}} = p_{\text{old}} \xi^{1/p_{\text{old}}+1}
$$

It is readily apparent this heuristic can be used to signal that an edge be enriched or degraded.
Chapter 4

Object-Oriented Design of the Program

This chapter will introduce the implementation of object-oriented concepts in Fortran 90 programming language and will present the design for this finite element package.

4.1 Object-Oriented Programming in Fortran 90

Szymanski et al. [38] and his student Norton [39] in their publications have elaborated on how to represent object-oriented concepts in Fortran 90. This section will only summarize the object-oriented features supported by Fortran 90:

4.1.1 Abstraction and Encapsulation

Fortran 90 provides many features to implement abstraction. The main features are:

4.1.1.1 Classes

Fortran 90 allows creation of user defined types to encapsulate the related concepts into a single form. The type thus created, needs to be associated with the related methods which provide an interface for communication with other objects. This is achieved in Fortran 90 by placing the type and its methods within a module construct, thus creating a class. Each module generally contains one class, but more than one logically related class may be in a module. Tables 4.1 and 4.2 shows a Point class within a module; this class encapsulates the coordinates and number of dimensions of
the **Point**. The **use** statement provides access to the methods of one object, and to other objects in different **modules** (this is not merely do a text-substitution, as done by **include** command in Fortran 77). This is seen in Table 4.1 as the **Point** class located in **Point module** gets access to the data and methods in the **Utils module** and **Double Precision module**. We can declare contents in a **module** as **private** or **public** to provide the outside user access only to relevant information and to hide the complexity within the **module**.

### 4.1.1.2 Advanced Array Features

Fortran 90 has dynamic memory management features, including allocatable arrays and pointer attributed objects. This is one of the major advancement over Fortran 77. The Fortran 90 arrays are not just collection of types but also encapsulate their size. One can query about the size of an array using the **size** statement. Taking advantage of this feature Fortran 90 allows use of an **assumed shape array**, which is an input array argument for which one does not specify the size in the declaration statement within a procedure.

Fortran 90, provides **automatic arrays**, which are created on entry and destroyed upon exiting a procedure. The **automatic arrays** can be sized according to the size of input array argument, so there is no need to pass extra array as an argument. Table 4.3 shows the usage of **assumed shaped arrays** and **automatic arrays**.
Module Point_Module
   Use DBL_PRECISION_MODULE ! Access additional features
   Implicit none

   Private        ! Everything is private by default
   Public :: Point, Init, Operator (==), GetX  ! Public stuff.

   Type Point
      Private    ! Subcomponents are hidden.
      Real(dp) :: x, y, z
      Integer :: Dim
   End Type Point

! Generic programming, binding to single call "Init"
Interface Init
   Module Procedure Init_Point, Init_point_Vctr
End Interface

! Overloading the "==" operator for point class.
Interface operator (==)
   Module procedure equality_operator_point
End Interface

Table 4.1: Point class demonstrating various Object-oriented concepts.
       declarations.

4.1.2 Polymorphism

Fortran 90 allows logically similar methods operating on different types, to be called
with the same name. This property can be seen in some intrinsic functions like
int(a), where a can be of integer, real or complex type. This can extended to
user defined objects using generic programming, operator overloading and the
interface construct. For Example, in Tables 4.1 and 4.2 Init_Point is bound
to Init using the interface construct. This binding can be performed in all the
classes in the program, with a result the user only utilizes one call Init instead
Contains

! Constructor
Subroutine Init_Point(p1, x_val, y_val, z_val)
    ! details omitted...
End Subroutine Init_Point

! Another version of Init taking a vector.
Subroutine Init_point_Vctr(p1, vec)
    ! details omitted...
End Subroutine Init_point_Vctr

! Defining the function to perform equality.
Function equality_operator_point(p1,p2) Result(res)
    Type(Point), Intent(In) :: p1, p2
    Logical :: res

    If((p1 % X == p2 % X) .AND. 
        ((p1 % Y == p2 % Y) .AND. 
        (p1 % Dim == p2 % Dim))) Then
        res = .true.
    Else
        res = .false.
    End if
End Function equality_operator_point

! Accessors, provide interaction with other objects.
Function GetX(p1)Result(res)
    ! details omitted...
End Function GetX

! other methods...

End Module Point_Module

Table 4.2: Point class continued, methods.
Function MatVectprod (mat, vec) Result (res)
    Implicit none
    ! Assumed shaped arrays.
    Real, Intent (In) :: mat(:,:)
    Real, Intent (In) :: vec(:)

    ! Automatic array.
    Real   :: res (Size(mat,1))
    Integer :: rows, cols

    rows = Size (mat,1)
    cols = Size (mat,2)

    If(cols /= Size (vec)) Call Errhandler &
        ('In MatVectprod,inconsistent sized' // &
        'arguments', Fatal)

    res = Matmul(mat, vec)
End Function MatVectprod

Table 4.3: Illustrating the Fortran 90 array features.

of Init_Element, Init_Point or Init_Point_Vector etc. At the compile time, the
compiler uses the information about the argument types to determine between which
procedure to execute.

With this language operators can be overloaded for user defined types (already
existing for implicit types) additional operators can also be defined. Tables 4.1 and 4.2,
show how the '==', equality operator is defined for the Point class. The assignment
operator '=' between two objects of same type is implicitly implemented by the
language. Using the assumed shape array, automatic array and size command
the procedures can be made independent of the length of the array arguments, lending
more flexibility to the code.
4.1.3 Inheritance

Inheritance is the ability to reuse and share the data and methods of existing classes to construct complex classes in a hierarchical fashion. Fortran 90 supports following types of inheritance:

- **Inheritance by Granting Access:** This is the simplest form of inheritance. The `use` statement in Fortran 90 allows objects in one module to get access to the public (Visible, Not hidden) methods and data, in the base module.

- **Inheritance by Composition:** This type of inheritance uses other classes to build bigger and complicated classes. This is also known as ‘has-a’ inheritance. This is demonstrated in Table 4.1.3 the `Gauss_Pt` class has an instance of `Point` class to define its position in the space. `Gauss_Pt` class is granted access to contents of the `Point` class using public declarations and `use` statement.

<table>
<thead>
<tr>
<th>Type Gauss_Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real(dp) :: Weight</td>
</tr>
<tr>
<td>Type(Point) :: Pos</td>
</tr>
<tr>
<td>End Type Gauss_Type</td>
</tr>
</tbody>
</table>

Table 4.4: Example of Inheritance by Composition

- **Inheritance By Subtyping:** This type of inheritance can be defined as ‘is-a’ inheritance, illustrated in Fig. 1.2. The ‘is-a’ inheritance is explicitly implemented in C++ and Java languages, while in Fortran 90 it can be simulated. The technique to implement this in Fortran 90 was introduced by Szymanski et al. [38], and can add dynamic capabilities to the program. One of the classes in the design of this thesis is `Solver` class, which represents the solver used
to solve the finite element system of equations. The goal would be to have flexibility to use either a matrix type solver or an iterative solver, based on an input parameter. Tables 4.5 and 4.6 demonstrate the procedure to achieve this in Fortran 90. First \texttt{MatrixSolver} and \texttt{IterativeSolver} classes are created. Then, a big \texttt{Solver} class, containing pointer attributed instances of both \texttt{MatrixSolver} and \texttt{IterativeSolver} classes, is created. Based on the values of an input parameter, one of the pointer sub-components of the big \texttt{Solver} class is initialized while the others are nullified. The big \texttt{Solver} class is visible to the outside program and encapsulates or hides the \texttt{MatrixSolver} and \texttt{IterativeSolver} classes. The methods of the big \texttt{Solver} class will execute the corresponding methods of only the live (non nullified), pointer attributed sub-component. This is a powerful but computationally expensive method.

4.1.4 Safety Features

Fortran 90 is a disciplined language and provides a framework to avoid errors in the program. One of the feature is that pointers are only an attribute and not a different entity (as in C++). For example we can declare a pointer attributed instance of an element class as,

\begin{verbatim}
Type(Element), Pointer :: An_Elem
\end{verbatim}

\texttt{An_Elem} is not a pointer nor its a number representing some memory location, but has a pointer-attribute enabling it point to other objects (of the same type) while also behaving like a non-pointer instance. Fortran 90 implicitly dereferences \texttt{An_Elem}, to get the data pointed by \texttt{An_Elem}. This hides the details of dereferencing and simplifies the code. Consequently, pointer arithmetic, which is another source of
errors is disallowed. While allocating or deallocating a pointer attributed object or an allocatable array, the language allows to check the success of the operation. One can also enquire about the state of a pointer using associated function and that of an allocatable array using allocated function. Another safety feature is the specification of intent when declaring arguments within a procedure. An argument declared with intent(in) cannot be modified within the procedure, while an argument declared with intent(out) must be defined within the procedure. The intent(inout) allows both.

Module Solver

! Defining the each type of solver.
Type Solver_Mat_type
  ! details omitted...
End Type Solver_Mat_type

Type Solver_Iter_Type
  ! details omitted...
End Type Solver_Iter_Type

! The big solver type contains instances of all the solvers.
Type Solver_big
  Type(Solver_Mat_type), Pointer :: solver_mat
  Type(Solver_Iter_type), Pointer :: solver_iter
End Type Solver_big

Interface Init
  Module Procedure Init_Big, Init_Mat, Init_Iter
End Interface

Interface Solve
  Module Procedure Solve_Big, Solve_Mat, Solve_Iter
End Interface

Table 4.5: Example of Inheritance using Subtyping, declarations
Contains

Subroutine Init_Big (this, choice)
  Type(Solver_big), Intent(Out) :: this
  Integer :: choice

  Nullify(this % solver_mat, this % solver_iter)

! Depending on choice only the needed subtype is initialized.
  if(Choice == ITER_SOLVER) THEN
    Call Init (this % solver_iter)
  Else
    Call Init (this % solver_mat)
  End if
End Subroutine Init_Big

Subroutine Solve_Big(this)
  Type(Solver_big), Intent(Inout) :: this

! Executing only the initialized component.
  if(Associated(this % solver_iter)) Then
    Call Solve (this % solver_iter)
  Else
    Call Solve (This % solver_mat)
  End if
End Subroutine Solve_Big

! Methods for Matrix solver and iterative solver...

End SolverModule

Table 4.6: Example of Inheritance using Subtyping continued, methods.

Another useful feature of Fortran 90 is its compatibility with programs written in Fortran 77, thus allowing an incremental transition from Fortran77 to Fortran 90. Since Fortran 90 is a subset of High Performance Fortran (HPF), it also serves as a migration path to parallel computers via HPF.
4.2 Object-Oriented Development Process

Creating an object-oriented program can be considered as a four step process.

- Determine the types of objects needed, based on a specific application.
- Define the interface, i.e what these objects need to be and how they interact with each other.
- Implementation: coding.
- Verification and testing.

This is not a one pass process, but rather an iterative process. For example, during implementation, we may realize a better way to design some of the classes.

4.3 Design of the Finite Element Program

For any application there is no one optimum design (of the Classes). A good design should exploit the benefits offered by the object-oriented approach. Along with that it needs to address the issues like efficiency and ease of implementation. The design of this prototype program consists of several modules. Each module contains one or more logically related classes. There are eight major classes- Problem, Adaptor, Solver, Domain, Element, Edge, Constraint and Grid Point. There are eight auxiliary classes namely: Skyline, Spr (superconvergent patch recovery), Grid_Point_Sublist, Gauss, Material, Dof (degree of freedom), Point and Solution. There are six modules containing data and procedures, used by more than one class, namely User_Specific, Interpolation, Element_Assembly, Control_Data, Utils and Precision. These common data and methods are grouped
on the basis of logical or functional similarities. Fig. 4.1 depicts the design of this prototype. The relationships can be of containment or only using the methods.

This is an appropriate place to note an important detail. To remove confusion between a finite element node and a node in the link (and circular) lists, the finite element node is referred as grid point while a list node is simply referred as node. Here are modules discussed in more detail.

4.3.1 Problem Module

This is the highest level class in the program. Fig. 4.2 shows the structure of the Problem class.

4.3.2 Adaptor Module

This module contains the Adaptor class, shown in Fig. 4.2. One of the subcomponents, Ptr_To_Domain establishes a link, for Adaptor class to access the data and methods in the Domain class.

4.3.3 Solver Module

The structure of the class contained by this module is shown in Fig. 4.2. Here also the Solver class is linked to Domain class using Ptr_To_Domain. The stiffness matrix is stored in Stiff_Mat, which is an instance of Skyline class. Dof_Arr, an array of type Dof, which stores the mapping between the local and the system numbering schemes. Soln, an instance of Solution class holds the solution values for each degree of freedom.
Global data:
Precision, Control_Data, Utils,
Point, User_specific.

Figure 4.1: Illustrating the program design.
### Figure 4.2: The Problem, Adaptor, Solver and Domain classes

#### 4.3.4 Domain Module

This module holds the Domain class. The Domain class contains the Element, Edge and grid Point lists and iterators for these lists. The iterators are used to implement efficient traverse and query algorithms discussed in Chapter 5.
4.3.5 Element Module

This module contains the Element class, Element link list class, Element list iterator class and Element Pointer class. The Element list class and the Element iterator class provide methods for list manipulation and traversal, discussed in Chapter 5. The Element class is shown in Fig. 4.3, has two arrays of pointers one containing pointers to the constituent edges, Edg and other containing pointers to the adjacent elements, Adj. The geometric grid point coordinates of an Element are calculated and stored in Gm_Grdpt_Crds, to avoid expensive recomputation for each iteration. Dof_Map contains the system degree of freedom numbers for the local element degree of freedom numbers. Smpl_Data is a link list containing the sampling Point data for the Element, used for accurate gradient recovery.

4.3.6 Edge Module

The Edge module contains the Edge class, Edge list class and the Edge list iterator class. The Edge class is depicted in Fig. 4.3. Each Edge contains a circular list, GrdptSubList, to keep track of the grid points on the Edge. The GrdptSubList stores pointers to the grid points lying on the Edge. IsBndryFlux is a logical field, used to distinguish Edges with flux boundary condition from rest of the Edges.

4.3.7 Grid Point Module

The Grid Point module also has a similar structure as the Element and Edge modules; it contains the Grid Point class, Grid Point list class and Grid Point list iterator class. The Grid Point class is shown in Fig. 4.3. Pos, an instance of Point class, keeps track of the coordinates of the Grid Point. Dof_info is an in-
teger flag which is used to classify analytic or geometric nature of the Grid Point. Due to the use of different interpolation functions for the geometry and the solution, a Grid Point could be either analytical, geometric or both. The functions Get_Is_PureGeometric(), Get_Is_PureAnalytic() make this transparent to the user. Dof_info also serves to uniquely identify each Grid Point. After each iteration the procedure Reset_Sys_Dof_info is executed, which traverses the Grid Point list and progressively increases (in an absolute sense) the integer values stored in Dof_Info. This is used as default numbering scheme for the system and can be used to implement the equation renumbering algorithm.

Constraints indicates what boundary conditions are applied on the Grid Point. A constraint code of ‘1’ indicates dirchlet boundary condition of type 1, that is:

\[ D_j = b_j \] (4.1)

The constraint code of ‘-1’ indicates neumann boundary condition, while a code of ‘0’ indicates an unconstraint Grid Point. Mat_prop stores the nodal properties, if provided.

4.3.8 Constraint Module

This module contains the Dirchlet and Boundary Flux classes. The Dirchlet class is illustrated in Fig. 4.3. The integer flag param tells which parameter of each grid point the boundary condition is applied to.

The auxiliary classes also deserve some description. The Skyline class is used to implement the skyline storage of a square matrix. The functions like Get(row,col), hides the implementation complexity from the user. This allows us the flexibility
### Element Class

<table>
<thead>
<tr>
<th>Attributes:</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Variable</td>
</tr>
<tr>
<td>Integer</td>
<td>Id</td>
</tr>
<tr>
<td>Integer</td>
<td>No_As_GrdPt</td>
</tr>
<tr>
<td>Material</td>
<td>Mat_Prop</td>
</tr>
<tr>
<td>Array of integer</td>
<td>CrvGrdPt_Seq</td>
</tr>
<tr>
<td>Array of Edge pointer</td>
<td>Edg</td>
</tr>
<tr>
<td>Array of Element pointer</td>
<td>Adj</td>
</tr>
<tr>
<td>Array of Real</td>
<td>Gm_Grds_Crds</td>
</tr>
<tr>
<td>Array of Real</td>
<td>Elem_Sqr_Mtrx</td>
</tr>
<tr>
<td>Array of Real</td>
<td>Elem_Col_Vcr</td>
</tr>
<tr>
<td>Array of Integer</td>
<td>Dof_msp</td>
</tr>
<tr>
<td>Sampling data list</td>
<td>Smp_data</td>
</tr>
</tbody>
</table>

**Tasks:**
- Read the element properties.
- Generate Dof, map.
- Generate element Stiffness matrix.
- Find element bandwidth.
- Find element Column height.
- Get number of Analysis grid points.
- Perform Postprocess.

### Edge Class

<table>
<thead>
<tr>
<th>Attributes:</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Variable</td>
</tr>
<tr>
<td>Integer</td>
<td>Id</td>
</tr>
<tr>
<td>Integer</td>
<td>No_As_GrdPt</td>
</tr>
<tr>
<td>List of pointers to Grid_Pts</td>
<td>GridPSublist</td>
</tr>
<tr>
<td>Logical</td>
<td>IsInbndryFlux</td>
</tr>
</tbody>
</table>

**Tasks:**
- Provide functionality for higher level classes.
- Get number of Analysis grid points on edge.

### Gridpt Class

<table>
<thead>
<tr>
<th>Attributes:</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Variable</td>
</tr>
<tr>
<td>Integer</td>
<td>Id</td>
</tr>
<tr>
<td>Pos</td>
<td></td>
</tr>
<tr>
<td>Material</td>
<td>Mat_Prop</td>
</tr>
<tr>
<td>Integer</td>
<td>Dof_info</td>
</tr>
<tr>
<td>Integer</td>
<td>Constnt_indicator</td>
</tr>
</tbody>
</table>

**Tasks:**
- Position itself in space.
- Provide functionality for other classes.
- Reset system degree of freedom.
- Store Information about the grid point.

### DirccBc Class

<table>
<thead>
<tr>
<th>Attributes:</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Variable</td>
</tr>
<tr>
<td>Array of Real</td>
<td>values</td>
</tr>
<tr>
<td>Array of Integer</td>
<td>gridPIds</td>
</tr>
<tr>
<td>Array of Integer</td>
<td>Param</td>
</tr>
</tbody>
</table>

**Tasks:**
- Read and initialize itself.
- Provide functionality for other classes.

---

**Figure 4.3:** The Element, Edge, Grid point and DirecBc classes
to use a different matrix storage method, since by implementing the corresponding Get(row,col) method, the rest of the program will remain unchanged. For recovering the superconvergent gradients we need a list to store the sampling point information of the element; this functionality is provided by Sampling Point Data class and the corresponding list and iterator classes in the Spr module. GridPtSublist class forms a circular list of Grid Point pointers. This list is used to represent the Grid Points on an Edge. The Gauss class, illustrated in Table 4.1.3 stores the position and the weight associated with a gauss point. Material class is used to read and store the properties. Dof (Degree of freedom) class is used to form mapping between system and local degree of freedoms. Dof class also provides equation renumbering algorithms. Point class which was illustrated in Tables 4.1 and 4.2, implements a point in one, two or three dimensional space. Solution class is used to hold the system solution.

This prototype program also has some modules, which contain common data and methods. Interpolation module provides methods for both geometric and solution interpolation. User Specific class holds procedures like exact solution for benchmark test cases and source functions etc. Element Assembly module provides the structures needed to generate the element stiffness matrix. Control module stores global level tags and flags, while Util module stores globally used procedures for implementing validation and error checking. Precision module defines the precision type.
4.4 Discussion

In this section we will analyze what do we gain by using object-oriented approach and demonstrate that it is worth the effort. Some of the advantages of following object-oriented policies are:

- The main advantage of an object-oriented scheme lies in data encapsulation. Firstly, the information is decentralized as such, it may be stored at easily reachable places (no complicated pointing devices) and processed at suitable time. Secondly objects possess more than integer identifiers, they can possess other objects, for example the `Element` class does not contain the number of material properties but the material properties themselves.

- The program is more organized and intuitive, as objects present themselves to the external world in a more meaningful way (for example, 'I am a beam on the third floor connecting such and such ..' as opposed to 'I am an array element 5 in element_array').

- We can easily modify the implementation of one part of code without affecting the rest of it. This is of great relevance to scientific research as the researcher can try new algorithms, new data structures and different problems with less programming effort. In this application the `Solver` is uncoupled from the domain information (`Element`, `Edges`,...). The `Domain` is also independent of the matrix storage technique and data structures used.

- It's easy for other people to understand the code written by us. This is gained by separating 'what we are doing' from 'how we are doing'.
• Concurrency issues - As object-oriented approach encourages localized, distributed grouping of data and tasks. It's relatively natural to see how data and tasks can be distributed over processors (useful for parallel machines) and among individuals (good for team environment).

• In object-oriented programming we group data (attribute) and functions (methods) in a class. We do not have to pass the arguments; they are inherently available to the methods, both being in the same class.

• By making use of the inheritance, we can add new components and reuse the earlier written code. In this application through the use of inheritance the type of solver used can be decided by an input parameter at the run-time.

One of the main criticisms of object-oriented programming is its overhead, which makes it inherently inefficient with respect to speed. Mostly because of the late (run-time) binding typically associated with object-oriented environments and addition of new procedure calls for implementing encapsulation. These are offset by the reduction in human time and effort, as human time is a more precious commodity than computer time. The number of function calls can be reduced by the use of pointers.

Though much have been said about object orientation making tasks easy for the software developer. However object-oriented programming does not obviate the need for forward thinking nor does it make it impossible to introduce a bug. The success of an object-oriented program heavily depends on its design. As the program is supposed to be flexible and easily extensible in future, the developer needs to think beyond the present application. Initially the program design can consume a large amount of total application development time, this was also the case with this prototype. As
the developer gains experience in object-oriented principles, the subsequent projects will spend less time on the design board.
Chapter 5

Data Structures

Adaptive finite element programs place demanding requirements on the data structures employed. Some of these requirements are: flexibility, capable of handling insertion, deletions at any location within the data structure; dynamic, data structures should have dynamic memory management capabilities to effectively utilize the available memory; and efficiency. Typical finite element programs employ complex data and control structures to store and manipulate information. In such a set-up several arrays and tables are necessary to store the characteristics of nodes, elements and material properties. This inconsistency in the data structures lead to programming errors that are very hard to detect. Consequently, the programs become inflexible and essentially static. Moreover, these programs are inefficient when dynamic alterations are done to the mesh.

Researchers have used different types of dynamic data structures for adaptive programs. Rheinboldt et al. [37] advocate the use of a more advanced tree structure to implement a localized h-refinement method. Demkowicz [15] augments this tree data structure with additional structures in order to implement a hybrid two-dimensional hp-refinement strategy. Hardy [22] extends this work to three-dimensions. Maddox [27] developed a tiered link list structure for rp-refinement that was flexible and yet simpler than the tree structures. This effort builds on Maddox's work.
5.1 Tiered Linked List Data Representation

Before venturing further, it will be useful to briefly define a link list. A link list can be considered as a train, where each compartment is known as a node (this is different from a finite element node). Analogous to a train compartment, a node contains some data and is connected to the next node. The link between the nodes is formed using pointers. Fig. 5.1 shows a singly linked list. In a singly linked list, nodes are connected in such a way that the list can be traversed in one direction only, from the beginning of the list to the end of the list. Some other variations of lists are doubly linked list and circular link list shown in Fig. 5.2

![Diagram of a singly linked list]

**Figure 5.1**: A singly linked list.

In this research, separate link lists are maintained for elements and their associated edges and grid points. This requires a computer programming language that supports derived types, pointers and dynamic memory management, like C, C++ and Fortran 90. One principal advantage of employing such list structures is their dynamic nature, list items can be inserted and deleted arbitrarily, so space is utilized only when needed. This is much better than the traditional notion of declaring worst case, maximum-sized arrays thereby wasting lot of storage space. There is some extra space used for
Figure 5.2: A circular doubly linked list.

the pointers, which is an acceptable trade-off as this space is only a small percentage of the total usage.

The conceptual idea is to have an element-driven data structure that is flexible enough to give lower level access. Thus, the tiered hierarchy was implemented where each element data node has pointers to its associated edge data nodes, which in turn point to the grid point data nodes defining the edge. Consequently, if an element identifier is known, all other information related to that element is immediately accessible. The edge and grid point lists can optionally be traversed individually, to
allow maximum flexibility while maintaining a high degree of hierarchical abstraction. A composite example of this tiered structure is shown in Fig. 5.3. Individual component lists are described in the subsequent sections.

5.2 Singly Linked Element List

The element list is a singly linked list that stores all the elements present in the domain. It is equipped with a header node (as are all the other lists) which does not represent an item in the list. Its main purpose is to produce a condition where the list is never truly empty. This simplifies the list insertion scheme by removing an algorithmic special case. The element list is ordered in ascending order of the element id. Using a modified search algorithm efficient insertions have been achieved; this will be discussed in a later section.

As discussed in Chapter 4, an element contains EDG, an array of pointers to specific members of the edge list that correspond to edges of the element. An element also stores an array of pointers to specific members of the element list that represent adjacent elements in the mesh. Each element also contains a single link list, Smp.data which stores pertinent quadrature point and derivative data computed from the gradient recovery algorithm.

5.3 Singly Linked Edge List

The edge list is also a singly linked list. The edge list is queried based on starting and ending grid point on the edge, which makes it difficult to order the edge list. Ordering based on id is of no advantage in this situation, hence the edge list is made an unordered list. The salient feature of this list is that one of the fields is
Figure 5.3: Tiered list data representation of a solitary, unconstrained linear rectangular serendipity element.
actually a doubly linked circular sub-list. The data items in this sub-list are pointers to particular grid points in the grid point list. Thus the sublist represents the grid points lying on that edge. Since each edge is shared by at most two elements, with both having opposite positive sense, the circular list can be traversed from either end making the computation of element interpolation functions and derivatives a simple task.

5.4 Doubly Linked Circular Linked Grid Point List

The grid point list is a circular doubly linked list. Thus it can be traversed in any order from the beginning of the list to the end or vice versa. This implementation enables efficient modification of the list. This list is ordered according to the grid point id. The ordering not only makes it easier to implement the refinement algorithm, but also results in the design of efficient search, insert and delete operations. The performance analysis is presented in a later section.

5.5 Concept of an Iterator

An iterator is an object, which can be used to traverse a data structure. It forms a wrapper around the data structure and encapsulates (hides) the actual mechanics of the data structure manipulation. The Fig. 5.4 shows a single link list iterator. It has as its components a pointer to the actual list- List, a Current pointer which stores the current position of the iterator over the list, a Trail pointer needed for the delete operation and a Middle pointer used for the search algorithm. It also provides methods like insert, delete etc., for manipulating the lists. Due to its object-oriented
design, the user only has to know the call sequence to these methods and not how they are implemented.

![Diagram of a singly linked list and its iterator]

**Figure 5.4**: Relationship between a singly linked list and its iterator.

Fig. 5.5 shows the source code for a traverse operation. This provides a simple, consistent and standard way to traverse all the different types of data structures. To switch to a different data structure, only the methods IsEnd, GetCurrent, Go_next need to be redefined. The traverse (Fig. 5.5) program remains unchanged. This design has been used consistently in this thesis.
Do While (.NOT.isEnd (itr) )
    Call MyPrint ( Get_Current (itr) )
    Call Go_Next (itr)
End Do

Figure 5.5: A print routine illustrating the iterator traverse.

5.6 Efficient Search Algorithm for Element and Grid Point Lists

The list lookup operation is used by many other routines like, query, insert and delete. Thus, the efficiency of the lookup operation is of paramount importance. Due to the dynamically incremental nature of the list, data is rarely stored contiguously in memory. For example, to get the fifth node, the list will have to be traversed through first four nodes. This can be significantly slow for a long list. For the ordered lists, a more efficient binary search type algorithm can be implemented.

The idea is to search in a small subset of the list instead, of searching in the whole list. This can be achieved by dividing the list into two parts using the Middle pointer. As the list is in order (in ascending order of the ids), the part to be searched can be selected by comparing the id of the Middle pointer with the lookup id. This is similar to the binary search algorithm for first level. Fig. 5.6 graphically depicts this algorithm.
Figure 5.6: Binary search type algorithm for the linked lists.

An important point is how to set the **Middle** pointer. One of the options is to traverse the list to get the middle location and then, keep modifying the **Middle** pointer as the list changes. This was tried initially but, the complexity increased as there were many exception situations to deal with. Moreover, the algorithm became expensive as the more operations were needed. An alternative approach is to guess the middle location of the list; this is done by setting the **Middle** pointer to the recent most node. Initially the **Middle** pointer is set to list header. After an insert operation it is pointed to the node just inserted. After a delete operation, it is set to one before the deleted node. After a query operation, the **Middle** is pointed to the queried node. These manipulations keep modifying the **Middle** pointer, increasing the probability of traversing a smaller list, resulting in fewer loop iterations required for retrieval.
5.7 Performance Analysis

Since the position of the middle pointer is arbitrary, an analysis based on average number of iterations in a loop is performed. The analysis for the singly linked list is adapted from the reference [41]. Let \( P \) be the position of the node being queried, \( M \) be the position of the middle node and \( N \) the list size. The average is taken over all \( N^2 \) combinations of \( P \) and \( C \): both \( P \) and \( C \) can take any value between 1 and \( N \).

In the original search operation, for each value of \( P \) the list is traversed from the start taking \( P - 1 \) iterations to get to the required node. Therefore, average number of iterations is,

\[
\frac{1}{N} \sum_{i=0}^{N-1} I = \frac{(N - 1)}{2}
\]  

(5.1)

In the binary search type algorithm used in this thesis, fewer iterations are needed whenever \( P \geq C \). For example, if \( P \) is 4 and \( P \geq 4 \), \( C \) can take on any value between 1 and 4. The number of iterations for each of those values is 3, 2, 1, 0, respectively. The savings over the original version of search are 1, 2, 3, and 4 iterations. In general for each value of \( P \) between 1 and \( N \), there is a saving of \( 1 + 2 + 3 + \ldots + P - 1 \) iterations over the original version of search. The average saving in iterations is therefore,

\[
\frac{1}{N^2} \sum_{P=1}^{N} \sum_{I=1}^{P-1} I = \frac{1}{N^2} \sum_{P=1}^{N} \frac{P(P - 1)}{2} \approx \frac{N}{6}
\]  

(5.2)

So, from equations 5.1 and 5.2, the average number of iterations in the new search algorithm is approximately, \( N/3 \).

For insertions in an ordered list, the worst case of sequentially adding all the list members in ascending order is an \( O(n^2) \) operation that requires a complete list traversal every time something is added. By using the iterator, the operation to insert
all $n$ items in the list is reduced to an $O(n)$ operation. Thus, inserting a single item is an $O(1)$ operation. This is possible because, the elements and the grid points are usually inserted into their respective lists in increasing order of ids. At the end of the previous insert the Middle pointer, of the iterator is set to the inserted grid point. Since the next item to be inserted has a greater id, so its insert location is just after the Middle pointer. Thus starting the list traverse from the Middle pointer, only one loop iteration is needed to reach the new insert location.

The efficiency of the doubly linked list is increased with the use of improved search algorithm. Still the doubly linked list is more expensive than its singly linked counterpart as it involves double the number of pointer manipulations for each basic operation.

5.8 Discussion

List iterators have proved useful in fulfilling the requirements of this adaptive program. The most significant advantage being the flexibility to switch to another data structure, without modifying rest of the program. The use of iterators also simplifies the program by confining the complexity to its methods (query, insert, and delete). The program thus becomes consistent and easy to debug. This setup allows advanced search algorithms to be implemented. After testing, the iterator methods can be used repeatedly and can be treated like a black box for the rest of the program. Thereby, resulting in a more robust program.
Chapter 6

Numerical Results

This section explains the numerical studies carried out to test the method. The foremost thing is to check the elements used.

6.1 Element Validation

Two most important tests for an element are the single element tests proposed by Robinson [34] and the patch tests introduced by Irons [24]. The single element test is used to determine the sensitivity of elements to geometrical parameters like aspect ratio. While the patch tests are used to establish the convergence for an element. Thusly, patch tests will be used here.

The patch test is based on the fact that in a finite element system, the solution derivatives become constant as element size is reduced to zero. Thus, to be valid in the limit, the element formulation must be able to yield correct results when derivatives in the governing integral statement, take on constant or zero values. Two patch tests have been used here.

6.1.1 First Patch Test

This is a patch test for the Laplace equation,

$$Lu = 0$$  \hspace{1cm} (6.1)
Where,

\[ u(x, y) = 3x - 4y \]  \hspace{1cm} (6.2)

**Figure 6.1:** Initial mesh for the patch tests.

The first order derivatives of \( u \) are constant. So, it is a candidate for patch test. Fig. 6.1 depicts the initial mesh used. This program always starts with a linear mesh. The desired accuracy of 0.1 % was achieved in the first iteration itself, without any need for further refinement. The global percentage error was reduced to \( 4.5 \times 10^{-6} \) % at end of the first iteration. The solution and solution derivatives in the interior of the mesh matched perfectly with the analytical solution. Hence the element passes this patch test.
6.1.2 Second Patch Test

This is for Poisson's equation. The governing equation is

\[ Lu = -8 \]  \hspace{1cm} (6.3)

Where,

\[ u(x, y) = 3x - 4y^2 \]  \hspace{1cm} (6.4)

Figure 6.2: Mesh after the second refinement iteration, for the second patch test case.

For this patch test, first order derivative with respect to \( x \) and second order derivative with respect to \( y \) are constant. Initial mesh is same as the one used in the first patch test. The mesh for second iteration is shown in Fig. 6.2. In five iterations global percentage error reduced from 11.85 % to 3.4 \( \times 10^{-6} \) %. Final mesh is shown in Fig. 6.3. Fig 6.4 depicts the reduction in the error as the program goes through the refinement iterations.
Figure 6.3: Mesh after the fifth iteration for the second patch test case.

Figure 6.4: Exact error versus degree of freedom of the system, for the second patch test.
After the element validation, more practical test cases were run which include a potential flow test case, a Babuška test case and a problem with strong diagonal gradients.

### 6.2 Potential Flow Around a Cylinder

This is an example of two-dimensional problem, of an ideal incompressible fluid around a cylinder, also discussed in [27]. Fig. 6.5 shows the cylinder and the fluid which extends to infinity in all directions. A Uniform flow assumption is made. The closed form solution to this problem in terms of velocity potential is,

\[ \phi = U \left( r + \frac{R^2}{r^2} \right) \cos \theta + C \]  

(6.5)

where \( r \) and \( \theta \) are polar coordinates (with origin at the center of the cylinder) of a point in space, \( C \) is an arbitrary constant, \( U \) is the uniform undisturbed velocity, and \( R \) is the radius of the cylinder.

The radial and tangential velocity components are independent of the constant \( C \) and are expressed

\[ v_r = U \left( 1 - \frac{R^2}{r^2} \right) \cos \theta \]  

(6.6)

and

\[ v_\theta = -U \left( 1 + \frac{R^2}{r^2} \right) \sin \theta. \]  

(6.7)

These velocity components can be resolved into Cartesian coordinates for comparison with the finite element solution by the expressions

\[ u = v_r \cos \theta - v_\theta \sin \theta \]  

(6.8)

and,

\[ v = v_r \sin \theta + v_\theta \cos \theta. \]  

(6.9)
Figure 6.5: Potential flow around a cylinder problem.

Taking advantage of the centerline symmetry and midstream antisymmetry only a quarter region of the whole domain is selected for finite element discretization, as shown in Fig. 6.6. The initial mesh used in the first run is shown in Fig. 6.7, its a superparametric mesh with a second order polynomial used to describe the geometry and a linear polynomial used to represent the solution. Higher order geometric mesh is needed to accurately represent curved edges. While, the refinement algorithm refines the solution interpolation from linear onwards. Two set of boundary conditions were tested,

- Essential boundary conditions on all the edges. Using the exact solution from eqn. 6.5.

\[ \phi = \bar{\phi} \quad \text{on all edges.} \] (6.10)
Figure 6.6: Potential flow finite element solution problem domain.

- Combination of natural and essential boundary conditions (Fig. 6.6.)

\[
\frac{\partial \phi}{\partial n} = 0 \quad \text{on edges } ab, bc. \tag{6.11}
\]

\[
\phi = \bar{\phi} \quad \text{on edges } cd, de, ea. \tag{6.12}
\]

6.2.1 Analysis for the Potential Flow Test Case

The meshes for third and sixth refinement iteration for the combination boundary condition case are shown in Fig. 6.8 and Fig. 6.9. The finite element solution has high error in the regions of sharply varying gradients, which in this case are around the stagnation point and along the curved edge. This trend is seen in the mesh refinement in Fig. 6.8 and Fig. 6.9. The global percentage error was reduced from 8.6 % to 2.5 % in the 3rd iteration and 1.2 % in the 6th iteration.

Rates of convergence using the different boundary conditions are seen in the Fig. 6.10. In the first approach, essential boundary conditions were provided only for the
Figure 6.7: Initial (Q8 4X6) mesh used for the finite element solution, with 24 elements and 35 degree of freedoms.

Figure 6.8: Mesh for the third iteration for the potential flow test case (Q8 4X6), with combination boundary conditions.
Figure 6.9: Mesh for the sixth iteration for the potential flow test case (Q8 4X6), with combination boundary conditions.

Figure 6.10: Convergence rates for runs conducted using different boundary conditions, measured in terms of norm of estimated error for a q8 4X6 mesh.
Figure 6.11: Convergence rates for runs conducted using different boundary conditions, measured in terms of norm of exact error for a q8 4X6 mesh.

original mesh. As the mesh is refined, boundary conditions for the new boundary grid points were obtained through quadratic interpolation of the original values. This case will be referred as the interpolated boundary condition case and is represented by the red line (with △ ticks) on the convergence plots. Interpolated boundary conditions usually have the effect of altering the system being solved. This happens in every iteration, whenever new grid points, lying on the boundary, are constrained. As a result, the estimated error at the boundary increased, resulting in excessive number of grid points to be placed on the edges near those boundaries and early saturation of the solution, as seen in Fig. 6.10 and Fig. 6.11. In the second approach essential boundary conditions for new boundary grid points were computed from the known
exact solution. This approach is referred as the exact essential boundary condition case, exhibits faster convergence than the first approach, shown in Fig. 6.10 and Fig. 6.11 as black line (with \( \circ \) ticks).

\[ \text{Log Estimated Error in Energy Norm} \]

\[ \text{Log Number of Unknowns} \]

**Figure 6.12:** Comparison of convergence rates for potential flow test case between Q8 4x6 mesh and Q8 16x24 mesh, measured in terms of norm of estimated error.

The accuracy of the solution was further increased by specifying natural boundary conditions on the curved edge \( ab \) and on the bottom edge \( bc \) seen in Fig. 6.6 (with \( \ast \) ticks). Fig. 6.11, shows the plot of log of the exact error versus log of number of degrees of freedom. Another trial run was conducted with a much finer mesh, Q8 16x24. Fig. 6.12 shows the comparison between the two meshes. In the finer mesh, essential boundary condition and combination boundary condition give almost identical results.
Figure 6.13: Variation of the effectivity index for Potential flow test case (Q8 16x24 mesh).

There is a loss of order of convergence which can be seen in the Fig. 6.12. This is due to the lack of polynomial completeness of the element interpolation functions. As the serendipity element employed here do not contain internal degree of freedom, hence the functions above quadratic order are incomplete. In this program, the maximum edge degree is limited to seven. This results in a saturation condition, when elements to be refined reach maximum edge degree, therefore running out of the elements that can be refined, as can be seen in Fig. 6.10.

The efficiency of the error estimates can be measured by the effectivity index. Effectivity index, introduced in Chapter 3, Eqn. 3.45, is the ratio of computed error over the exact error expressed in terms of energy norm. Effectivity index should converge to one as the mesh refines. Fig. 6.13 shows the plot of effectivity index for
the Q8 16x24 mesh. After the seventh iteration, effectivity index increases, as the estimated error becomes over-conservative. This is because of the incompleteness of the polynomial interpolation functions. Loss of the rate of convergence in Fig. 6.12 after the seventh iteration confirms this observation. For this test case, it is on the safe side, as the program overestimates the error.

6.3 Babuška Test Case

This is a two-dimensional steady state heat transfer problem with one side (side OA in Fig. 6.14) having a prescribed temperature of zero degrees. The heat transfer is driven by fluxes specified on all the rest sides of the domain. The differential system of equations governing the problem is:

$$L(T) = 0$$  \hspace{1cm} (6.13)

Analytic solution to this problem is,

$$T = C \cdot r^{1/2} \sin(\theta/2)$$  \hspace{1cm} (6.14)

With,

$$C = 0.0700754$$

$r$ and $\theta$ are the polar coordinates with origin at $O$ in Fig. 6.14. The exact gradients plotted in the same figure show a naturally occurring singularity at the point $x = 0, y = 0$ where $r = 0$. The order of this singularity, $\lambda$, is equal to 0.5.

Gradients of $T$ in polar coordinates are given by,

$$\frac{\partial T}{\partial r} = 0.5 \cdot C \cdot r^{-1/2} \sin(\theta/2)$$  \hspace{1cm} (6.15)

$$\frac{\partial T}{\partial \theta} = 0.5 \cdot C \cdot r^{1/2} \cos(\theta/2)$$  \hspace{1cm} (6.16)
Using the chain rule of differentiation gradients in Cartesian coordinates are obtained.

\[
\frac{\partial T}{\partial x} = \frac{\partial T}{\partial r} \frac{x}{r} - \frac{\partial T}{\partial \theta} \frac{y}{r^2}
\]

\[
\frac{\partial T}{\partial y} = \frac{\partial T}{\partial r} \frac{y}{r} + \frac{\partial T}{\partial \theta} \frac{x}{r^2}
\]

(6.17) (6.18)

The finite element discretization is shown in Fig. 6.15. It is a 4-node isoparametric mesh, that is linear analytic (solution space) and linear geometric mesh. Essential boundary conditions are applied on all boundary edges, using the known exact solution.

6.3.1 Analysis for the Babuška Test Case

The graphs for the third and fifth iteration are shown in Fig. 6.16 and Fig. 6.17. Due, to the presence of the singularity in the solution, the program puts lot of new nodes near the singular region. Generally, an error estimator sees the singularity to be an area of very high error and has a tendency to focus only on the singularity. It
Figure 6.15: Initial finite element mesh for Babuška test case, a 4-node isoparametric mesh.

It is reassuring to see that the error estimator used here, also recognizes error in other parts of the domain, where element sizes increase suddenly, Figs. 6.16 and 6.17.

As the Babuška test case has a singularity of strength 0.5. A more accurate solution can be obtained if the singularity is accounted for in the solution. One way to do so is to employ special elements which, in the vicinity of the singularity, display singular behavior. Akin [1] gives a conforming element which has a general form,

\[ u = O(\rho^{1-a}) \quad \text{for} \quad 0 < a < 1 \]

With these elements the Babuška problem can be solved more accurately and efficiently. The effectivity index for this test case did not perform as well. It increased to about 2 after five iterations. This can also be attributed to the presence of the singularity. The plot of convergence is shown in Fig. 6.18 shows a drop in the percentage error from 4.8 % in the first iteration to 3.2 % in the fifth iteration. After that there is a loss of convergence because of the polynomial incompleteness.
Figure 6.16: Mesh for Babuška test case, for the third refinement iteration (Q4 20X40).

Figure 6.17: Mesh for Babuška test case, for the fifth refinement iteration (Q4 20X40).

Saturation due to the degree limit, previously seen in the potential flow test case, is also seen here.
Figure 6.18: Convergence for Babuška test case (Q4 20X40).

6.4 Strong Diagonal Gradient Test Case

The governing equation of the problem is

\[-Lu = f \text{ in } \Omega\]  \hspace{1cm} (6.19)

with boundary condition

\[u = 0 \text{ on } \partial\Omega\]  \hspace{1cm} (6.20)

where \(\Omega\) is a unit square domain \(\Omega = (1, 0) \times (0, 1)\) and \(f\) is chosen so that the exact solution is of the form

\[u = x \, (1 - x) \, y \, (1 - y) \, \tan^{-1}(\alpha(\xi - \xi_0))\]  \hspace{1cm} (6.21)

where \(\xi = (x + y)/\sqrt{2}, \xi_0 = 0.8\) and \(\alpha = 20\).
This test case has been used by Oden [30] and Zienkiewicz [46] for testing various
error estimators. The solution exhibits a relatively sharp transition of gradients along
the diagonal of the square domain. The contours of the exact solution of \( \partial u / \partial x \) and
\( \partial u / \partial y \) are plotted in Fig. 6.19. The finite element discretization is shown in Fig.
6.20, which is 4 node iso-parametric mesh. The initial number of degrees of freedom
is 121.

![Image of gradient contours](image)

**Figure 6.19:** Gradient contours for the strong diagonal gradient test case.

6.4.1 Analysis for the Strong Gradient Test Case

Mesh for the third and fifth refinement iterations are plotted in Fig. 6.21 and Fig.
6.22. These plots show that the regions of high error are correctly identified and
refined by the program. The error reduced from 39.3% to 14.8% in the third iteration
and to 11.17% in the fifth iteration. The solution saturates after the fifth iteration
which can be seen in the convergence plot Fig. 6.23. Effectivity index for this test
case is shown in Fig. 6.24.
Figure 6.20: Initial finite element mesh for strong diagonal gradient test case, a 4-node isoparametric mesh.

Figure 6.21: Mesh for strong diagonal gradient test case, for the third refinement iteration (Q4 10X10).
Figure 6.22: Mesh for strong diagonal gradient test case, for the fifth refinement iteration (Q4 10X10).

Figure 6.23: Convergence for strong diagonal gradient test case, Q4 10X10.
**Figure 6.24:** Variation of the effectivity index for strong diagonal gradient test case Q4 10x10 mesh.
6.5 Further Tests

Additional tests where performed on the suggested enhancement to the ZZ-SPR (Zienkiewicz-Zhu error estimator and Super-convergent patch recovery process) error estimator. Blacker et al. [12], has proposed an Equilibrium Residual enhancement for the gradient recovery, discussed in Chapter 3. The potential flow problem, discretized on a Q8 16 X 24 mesh, with combination boundary conditions, was used to carry these tests. Fig. 6.25 shows result in terms of rates of convergence. The black line (with * ticks) is the ZZ-SPR error estimator, while the red line (with △ ticks) shows Blacker's variation. Blacker's variation did not perform as well, it had slower convergence and stagnated early. This observation agrees with another study [8] (not published at time of this writing) which summarizes that the ZZ-SPR estimator is more robust. Further tests need to be done in this regard.

Another test was carried to determine the effect of using even degree polynomials versus the odd degree polynomials as the element interpolation function. Zienkiewicz-Zhu suggested that even degree polynomial give faster rate of convergence. This model starts from a linear mesh. As the maximum degree jump of two is permitted, so for most cases the new polynomial degrees computed by the error indicator go from 1 to 3 to 5. To get an even degree solution an additional heuristic was added. The degree change from 1 to 3 was modified to 1 to 2. Again due to maximum degree limit, this resulted in most polynomial degrees varying from 1 to 2 to 4 to 6. In fact the process of shifting from odd to even degrees reduces the number of degrees of freedom.

The blue line (with + ticks) on the Fig. 6.25 shows the convergence rate using even degree polynomials, while the black line (with * ticks) uses the odd degree polynomials. The even degree solution is much cheaper until the sixth iteration, while
Figure 6.25: Convergence tests for Equilibrium Residual enhancement and odd v/s even degree polynomial degree, using potential flow test case on a Q8 16X24 mesh with combination bcs.

the error is almost identical to that in the odd degree solution. Early stagnation in the even degree is due to the reduction in the degree of freedom as a result of the shift from odd to even degrees.

6.6 Discussion

The adaptive method introduced in this prototype code has provided good results for the test cases investigated. The exact error in the gradients is reduced considerably, although after approximately seven iterations a saturation was noticed in the convergence. This is occurring because of the use incomplete serendipity elements,
as we do not include internal nodes in our element formulations. The interpolation functions obtained by adding nodes only to the edges result in incomplete functions beyond the third degree. This error becomes more substantial for higher degree of the polynomials. A change in heuristics was considered to circumvent this problem, by limiting the degree of the interpolation functions of an element and allowing more elements to be refined in an iteration. The speed of the refinement process can be controlled by changing number of elements to be refined in an iteration. Only ten percent of all eligible (to be refined) elements are refined. This is done to better analyze the refinement process.

An important observation is that the rate of convergence in a P-method greatly depends on quality of initial mesh, as seen in Fig. 6.12. very high rates of convergence were achieved using a finer Q8 16X24 mesh. The Effectivity index also performed as expected for these test cases, taking into account the effects of element incompleteness and presence of the singularity.
Chapter 7

Conclusions and Future Work

The P-adaptive finite element application developed for this thesis has shown good results. Benchmark problems were run to verify the method. They provided further insight into the refinement process and were used to test some variations to the original method. The work carried out on optimizing the data-structures, on the object-oriented design and on the development of the GUI, tied together to produce significant results. The experience gained here forms a good platform for future applications.

An important future enhancement is to add internal degrees of freedom to the element interpolation functions. This will allow polynomial completeness (even) for interpolation functions greater than third degree, which will improve accuracy of the solution and effectiveness of the error estimates, resulting in faster convergence. Another area of improvement is to investigate the significance of pollution error in a finite element solution. Pollution error is discussed in the publications of Babuška [5], [7] and [8].

The success of this object-oriented design stems from the identification and separation of different concepts in the program into independent objects. This adds flexibility and permits easy extensibility. For instance the solver and the data-structures can be changed, requiring only minor editing. Upgrading to hp-adaptive method should be considered. Hp-adaption has been shown to yield exponential rates of conver-
gence. The logic of h-adaption will have to be implemented and optimally combined
with the p-adaption process. A more suitable data-structure to represent h-adapted
elements is a tree structure. So, the element data-structure may be switched to a
tree data-structure. R-adaption may also be implemented, it will provided a more
accurate solution at a fraction of cost.

An area of expansion is the extension of this method to three dimensions. The
interpolation function routine can generate functions for three-dimensional general
serendipity elements. An element face object is an essential component of a three-
dimensional version of this program. The element face object will form an addi-
tional tier between the element object and the edge object. Even though the lists
implemented here were optimized, they are still one-dimensional, and considerable
improvement in speed can be achieved by using a more efficient data-structure like a
tree or a hash table. This will be relatively easy in light of the object-oriented design
of the program.

The experience and technology presented in this thesis conceiving the GUI can be
utilized to develop more interactive applications. When developing a GUI, building
a function level interaction between the GUI and the application should be consid-
ered. This will give more control to the user and will increase the capabilities of the
application while removing unnecessary overlap between the GUI and the application.
Appendix A

Graphical User Interface Using Java

Fortran has been the language of choice for scientists and engineers. It is simpler to use and implements features which are helpful for scientific computing. The newer versions Fortran 90 (and 95) added a great deal of flexibility and allows object-oriented development. Still Fortran (and C++) do not contain standardized and platform independent constructs for creating user-interfaces, managing threads, networking and variety of other tasks for creating applications.

This program is run using input files, which includes the initial mesh, the program will then run and generate a number of output files. These output files can be post-processed using packages like Matlab. The control over the program is limited to editing the input file and again executing the program. This is typically the how most programs are run in scientific and academic communities. A graphical user interface commonly known as GUI, can dramatically increase the capability of an application, by providing the user with more control over the program. A GUI can save significant amount of time and effort by automating the tasks of pre-processing (input verification) and post-processing (analysis and visualization of results). Thus the application becomes easier to use, which broadens its user base. For these very reasons it was decided to develop a GUI for this prototype code. The following sections explain why Java was chosen for GUI development and how it was interfaced with the program written in Fortran 90.
A.1 Choice of Java for the GUI

Java was introduced by Sun Microsystems in late 1995 [19]. It has become very popular since then. The following explains the suitability Java for GUI development [23],

- **Platform Independence:** Java is intended to be a hardware independent language (though not completely hardware independent). Java compilers translate the Java source code into byte-codes that are interpreted on a Java Virtual Machine (JVM). The JVM is implemented on each hardware platform and makes the applications hardware independent. So, the same application can run on various different platforms.

- **Distributed:** Java is also called a distributed language. It provides lot of high-level support for networking. Java has large number of classes and methods already implemented within the language, which makes it easier to develop a GUI and add other capabilities to the application like visualization, audio, video, virtual reality and more. One of the most exciting applications of Java has been to develop systems for the World Wide Web (WWW). A GUI built with Java can easily modified to run on a workstation or over a network. This adds another dimension to the software.

- **Multithreaded:** It provides support for one than one process to be run at same time. An important benefit of multithreading is that it improves the interactive performance of the GUI.
• **Promoting Better Programming:** Java effectively addresses difficult programming issues. Java eliminates several type of errors which plague complex systems as,

- Pointer errors are syntactically impossible.
- Memory leaks are not possible because of the use of automatic garbage collection.
- Array bounds are checked on all array accesses.

Java provides advanced exception handling system that enables the developer to easily build error handling into the software. Java also provides a utility for automatically generating HTML documentation. Java is a pure Object-Oriented language. Besides the benefits of being an Object-Oriented language, there is a logically consistent mapping between Java and Fortran 90 classes.

There are issues about efficiency and reliable security of Java. However, the “just in time compiler” (JIT), which translates Java byte code into native code on various computing platforms, is being developed by many companies and should address this concern. Other issues in the Java language can be attributed to relative infancy of the language and should be sorted out.

### A.2 Combining Java and Fortran 90

The combination of the Java GUI and the Fortran program will be referred to as an application. The relationship between the Java GUI and the Fortran program is illustrated in Fig. A.1. The GUI forms a wrapper around the Fortran program and
interacts with the user. Communication between the GUI and the program is at the

![Diagram showing interaction between User, FORTRAN 90 Program, and JAVA GUI]

**Figure A.1:** The Java GUI and the Fortran 90 program combine to form a new application.

To run this new application the GUI is started, which then takes control of execution. The user can select a input file, which is read by the GUI. The GUI performs input validations and if everything is ok, it draws the initial mesh. When there is an error, user is informed about it and given option to specify another input file or exit the application. When user clicks to solve the problem, the GUI writes out an input file for the Fortran program. GUI then spawns a new process in which the Fortran program is executed with the new input file.

The Fortran program will regenerate results in form of output files. When the program has finished running, the GUI reads in the output files to perform post-processing. Among the various options available, user can see the refined mesh for each iteration. The GUI also plots gradient vectors for each iteration. Another
interactive capability is that the user can click on a particular grid point or element, in the figure, and see the information about it. The user can also edit the adaptivity parameters and solve again. For example with different desired accuracy or using a different type of patch definition and so on. Fig. A.2 is the screen capture of the GUI, running the Potential Flow problem. This figure shows adapted grid point locations for the third iteration. The gradient vectors can also be seen in the same figure. Fig. A.3 is a screen shot of the post-processing window and edit properties window. Due to size restriction these windows have been captured as a separate screen shot.

### A.3 Discussion

Though the GUI developed in this thesis has limited capabilities, it achieved its goal of being a technology demonstrator. It showed how Fortran 90, which is good for scientific computing can be effectively interfaced with Java, which is good for GUI development. This results in an interactive and useful application. There was considerable amount of work involved in creating such a fusion.

This Fortran program can be run as stand alone or through the GUI. Due to the networking capabilities of Java, this application can also be easily adapted to run over a network. This opens lot of exciting capabilities. A good scenario will be a remote user (for example on a oil rig) logging on to a network to perform analysis (irrespective of the hardware platform). Applications running over WWW will be get a definite boost with the launch of WWW II, which is expected to be thousands of times faster.

The interaction between Fortran and Java is at the file level, which is not as flexible as the function level interaction. Function level interaction will also remove data redundancy between Java and Fortran. Such embedded application require
Figure A.2: Java GUI screen shot.
Figure A.3: Screen shot of ancillary windows.
early planning and involve resolving issues of software interoperability. Development of these features will allow users to create applications using the best parts from each language.
Bibliography


